导航

概览

介绍 架构 快速开始

发版日志 术语表

规划

架构、版本和组件 部署容量规划 已验证的模型

设备方案

安装

安装前配置 上架软件包 安装 Alauda

安装 **Workbench**

配置 **Alauda AI** 入口和品牌

Tools 菜单酉

升级

升级 **Alauda AI**

迁移到 **Knative Operator**

升级 **Workb**

卸载

平台管理

硬件配置

多租户

安全配置文件

开发

工作台

Kubeflow Notebooks

连接

Pipelines

分布式工作负载

使用 **Kubefl**

实验跟踪

Agentic MLOps

组件

训练

- 训练指南
- 配额与调度
- 组件

部署

- 模型管理
- 推理服务
- 推理网关
- **LLM** 压缩器
- **Model as a Service**
- 组件

构建 **AI** 应用

- 使用 **Core Profile** 快速开始
- 使用 **Secure Profile** 的 **Demo**
- **Demo Adva**
- **Envoy AI Gateway** 后端包含两个
- 组件

评测与安全

- 评估 **LLM**
- 使用 **Ragas** 评估 **RAG**
- 用于 **LLM** 安

NeMo Guardrails 组件

监控与运维

日志与链路追踪 资源监控 偏差与漂移监

API 参考

介绍 **Kubernetes APIs**

概览

介绍

架构

组件说明

快速开始

预计阅读时间

发版日志

Alauda AI 2.8.0

术语表

介绍

欢迎使用 Alauda AI！Alauda AI 提供了大模型和 MLOps 平台的最佳实践，帮助您在大模型应用、AI 应用和机器学习等多种场景中高效完成任务。利用 Alauda AI，您可以快速完成以下常见任务：

- 模型存储与版本管理
- 模型（大模型）推理服务发布与运维
- 智能体编排、AI 应用开发与发布

在本用户手册中，我们将向您介绍 Alauda AI 平台的各项功能，帮助您充分利用我们提供的工具和服务。无论您是数据科学家、机器学习工程师，还是应用开发人员，我们相信我们的平台都能为您的工作带来便利和效率！

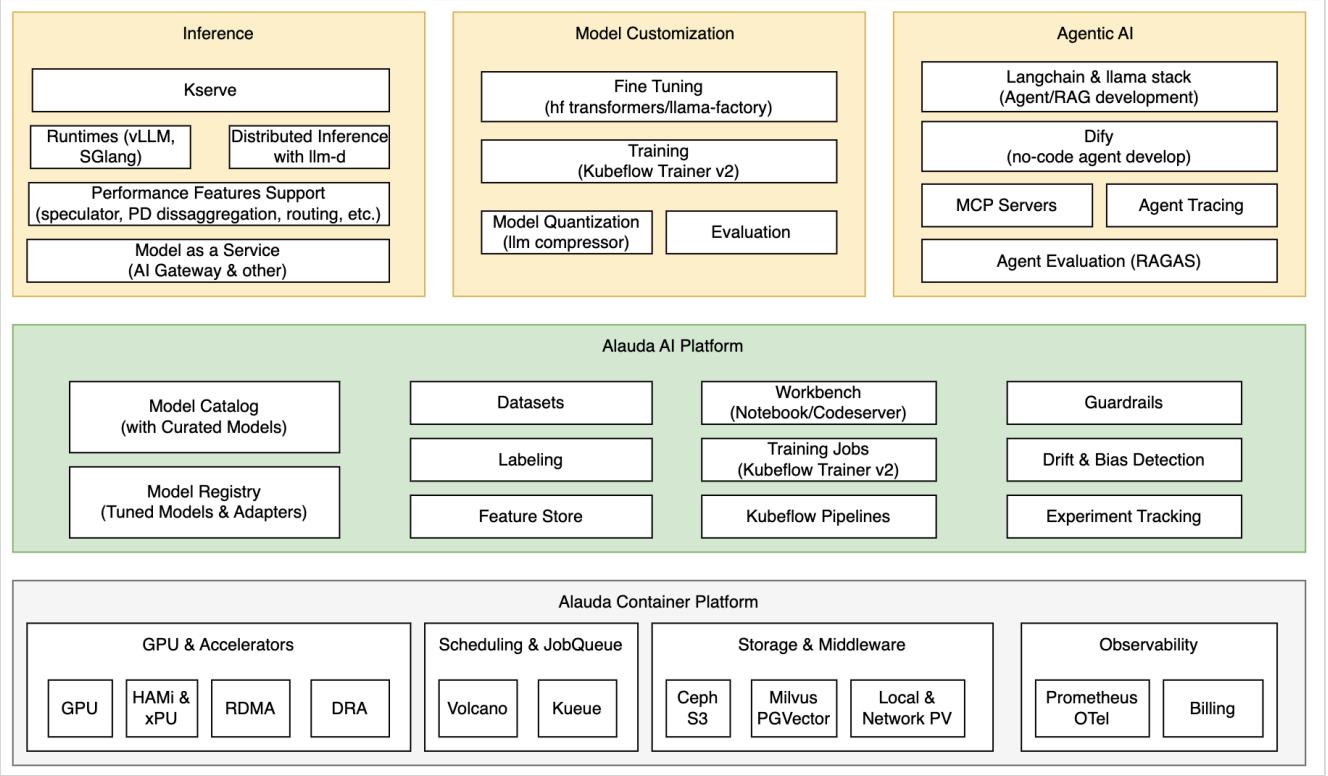
MLOps 与 LLMOps 介绍

MLOps（机器学习运维）定义了一系列在机器学习、大模型或其他 AI 应用的开发、训练、发布及运维过程中的原则和工具形态，成为高效、准确且可追溯地发布 AI 应用的最佳实践。

Alauda AI 不仅涵盖了 MLOps 流程中主体过程的工具支持，还为 LLMOps（大语言模型运维）的实践提供了更简洁的用户界面。从大模型推理服务的发布，到智能体和应用的构建，均以无代码的方式完成。

架构

下图展示了 Alauda AI 平台的架构。



目录

组件说明

Alauda Container Platform 层中的组件

AI Platform 层中的组件

GenAI Platform 层中的组件

组件说明

Alauda Container Platform 层中的组件

组件	描述	类型	许可证
GPU (Alauda Build of Nvidia GPU Device Plugin)	为 AI 工作负载提供 GPU 资源	开源	Apache Version 2.0
HAMi (Alauda Build of Hami, Alauda Build of Hami-WebUI)	GPU 资源切分、共享和调度	开源	Apache Version 2.0
Alauda Build of DCGM-Exporter	GPU 监控	开源	Apache Version 2.0
Alauda Build of NPU Operator	为 AI 工作负载提供 NPU 资源	开源	Apache Version 2.0
Alauda Build of Node Feature Discovery	检测集群节点的硬件特性	开源	Apache Version 2.0
DRA (Alauda build of NVIDIA DRA Driver for GPUs)	用于 GPU 共享的动态资源分配	开源	Apache Version 2.0
Volcano (Alauda support for Volcano)	为 AI 工作负载提供批处理任务调度	开源	Apache Version 2.0
Kueue (Alauda Build of Kueue)	为 AI 工作负载提供任务调度	开源	Apache Version 2.0
Milvus (Alauda Build of Milvus)	用于 embedding 存储和检索的向量数据库	开源	Apache Version 2.0
PGVector (Alauda support for PostgreSQL)	用于向量相似度搜索的 PostgreSQL 扩展	开源	The PostgreSQL License

AI Platform 层中的组件

组件	描述	类型	许可证
Model Catalog (Alauda AI/Alauda AI Essentials)	用于管理 AI 模型及其元数据的集中式仓库	专有	商业
Model Registry (Alauda support for Kubeflow Model Registry)	跟踪每个命名空间中的 AI 模型版本和元数据	开源	Apache Version 2.0
Datasets (Alauda AI/Alauda AI Essentials)	用于管理数据集及其元数据的集中式仓库	专有	商业
Labeling (Alauda support for Label Studio)	用于创建标注数据集的数据标注工具	开源	Apache Version 2.0
Feature Store (Alauda support for FeatureForm)	用于管理和提供机器学习特征的集中式仓库	开源	Mozilla Public License (MPL)
Workbench (Alauda AI Workbench)	用于管理 AI 项目的基于 Web 的界面，包括模型训练和推理	专有	商业
Training Jobs (Alauda support for Kubeflow Trainer v2)	Kubernetes 原生训练任务管理	开源	Apache Version 2.0
Kubeflow Pipelines (Alauda support for Kubeflow Base & Alauda support for Kubeflow Pipeline)	用于 AI pipeline 的工作流编排	开源	Apache Version 2.0
Guardrails (Coming soon)	AI 安全和治理框架	开源	Apache Version 2.0
Drift & Bias Detection (Alauda support for Evidently)	用于监控模型性能退化和偏差	开源	Apache Version 2.0
Experiment Tracking (Alauda support for MLFlow)	用于跟踪和比较机器学习实验	开源	Apache Version 2.0

GenAI Platform 层中的组件

组件	描述	类型	许可证
Kserve (Alauda AI/Alauda Build of KServe)	Kubernetes 原生模型服务框架	开源	Apache Version 2.0
vLLM (Alauda AI/Alauda Build of KServe)	适用于大语言模型的高性能模型推理引擎	开源	Apache Version 2.0
llm-d (Alauda Build of KServe)	适用于大语言模型的分布式推理引擎	开源	Apache Version 2.0
Model as a Service (Alauda build of Envoy AI Gateway)	用于将 AI 模型作为服务提供的 API gateway	开源	Apache Version 2.0
Fine-tuning	与 Workbench 集成的用于微调大语言模型的工具，例如 transformers、accelerate、llama-factory 等。	开源	-
Training (Alauda support for Kubeflow Trainer v2)	Kubernetes 原生训练任务管理	开源	Apache Version 2.0
Model Quantization	与 Workbench 集成的用于模型量化的工具，例如 llm-compressor 等。	开源	-
Evaluation	与 Workbench 集成的用于评估模型性能的工具，例如 lm-evaluation-harness 等。	开源	-
Llama Stack (Alauda build of Llama Stack)	用于构建大语言模型应用的框架	开源	MIT

组件	描述	类型	许可证
Langchain	与 Workbench 集成的用于使用 Langchain 构建 LLM 应用的工具	开源	MIT
Dify (Alauda support for Dify)	用于构建 AI 助手和聊天机器人的平台	开源	Apache Version 2.0 (modified)
MCP Servers	可与各种 MCP servers 集成	-	-
Agent Tracing (Alauda support for MLflow)	用于 AI Agent 的追踪和监控	开源	Apache Version 2.0
Agent Evaluation	与 Workbench 集成的用于评估 AI Agent 的工具，例如 RAGAS 等。	开源	-

快速开始

本文旨在帮助新用户快速了解如何在 Alauda AI 中部署推理服务。通过部署并体验一个简单的“文本生成”推理服务，您可以快速掌握平台的主要功能和使用方法。

目录

预计阅读时间
注意事项
前提条件
步骤概览
操作步骤
步骤 1：创建命名空间
步骤 2：管理命名空间并添加用户
步骤 3：准备并注册模型
步骤 4：发布推理服务
步骤 5：调用推理服务

预计阅读时间

预计完成本文的阅读和操作大约需要 20 分钟。

注意事项

本文仅演示基本流程。有关详细的参数配置，请参阅完整文档。

前提条件

- 您已拥有平台管理员帐户（用于创建和管理命名空间）。
- 您已准备好要部署的模型文件（可以提前从 Hugging Face 或 ModelScope 等网站下载）。
- 如果需要使用 GPU 推理，请确保已安装 GPU 插件。如果尚未安装，请在平台管理的插件中心安装 GPU 插件。
- 您了解 Kubernetes 和机器学习模型的基本概念。

步骤概览

步骤	操作	描述	注意事项
1	创建命名空间	在 Alauda AI 或容器平台中创建命名空间	如果已有命名空间，则跳过此步骤
2	管理命名空间并添加用户	将命名空间纳入 Alauda AI 管理，并向命名空间添加用户	如果命名空间已被管理且用户权限已配置，则跳过此步骤
3	准备并注册模型	将模型存储在 S3、PVC 或 OCI 存储中并注册其 Model URI，或使用平台共享的 Model Catalog 条目	如果已有已注册模型或使用目录模型，则跳过此步骤
4	发布推理服务	将模型发布为在线推理服务	
5	调用推理服务	通过 API 或“体验”功能调用推理服务	

操作步骤

步骤 1：创建命名空间

注意：如果已有命名空间，则跳过此步骤

命名空间是 Alauda AI 实现多租户隔离的基础，每个项目都应使用独立的命名空间。您可以直接在 Alauda AI 中创建命名空间，也可以从容器平台导入现有命名空间。

1. 进入 Alauda AI，并从顶部导航切换到 管理员视图。
2. 点击左侧导航栏中的 命名空间。
3. 点击 创建命名空间 并输入名称，例如“text-classification-demo”。
4. 点击 创建 完成命名空间创建。

如果命名空间已存在于容器平台中，请在 命名空间 页面点击 导入命名空间，然后选择要导入 Alauda AI 的命名空间。

步骤 2：管理命名空间并添加用户

注意：如果命名空间已被管理且用户权限已配置，则跳过此步骤

将创建的命名空间纳入 Alauda AI 管理，并向命名空间添加用户：

1. 进入 Alauda AI，并从顶部导航切换到 管理员视图。
2. 点击左侧导航栏中的 命名空间。
3. 如果命名空间尚未由 Alauda AI 管理，请点击 导入命名空间，选择新创建的“text-classification-demo”命名空间，然后完成导入。
4. 打开命名空间编辑页面并点击 成员管理。
5. 导入需要使用此命名空间的用户，并为用户分配以下角色之一：

   - Owner：可以管理命名空间，并将其他成员导入为 Editor 或 Viewer。Owner 不能导入另一个 Owner。
   - Editor：可以使用命名空间创建和管理 AI 资源。
   - Viewer：可以查看命名空间中的资源。

步骤 3：准备并注册模型

注意：如果已有已注册模型或使用平台共享的 **Model Catalog** 条目，则跳过此步骤

将文本分类模型存储在受支持的后端中并进行注册：

1. 进入 Alauda AI，在顶部导航中选择 用户视图，然后选择上一步管理的命名空间。
2. 将模型存储在 S3、已挂载的 PVC 或 OCI registry 中。记录其 Model URI，例如 `s3://models/Meta-Llama-3-8B-Instruct`。
3. 打开 **AI Hub > Model Registry**，选择 **Register Model**，选择一个 Model Registry，输入模型名称和版本，并提供 Model URI。
4. 根据需要在 registry 元数据中设置模型任务和框架。大多数热门开源 LLM 使用 `transformers` 框架和 `text-generation` 任务。
5. 如果模型已在 **AI Hub > Model Catalog** 中提供，请改为打开其目录详情，然后使用 **Create Inference API** 或 **Register Model**。

有关 Workbench 上传说明，请参阅使用 Workbench 上传模型。有关注册说明，请参阅 Model Registry。

步骤 4：发布推理服务

将模型发布为在线推理服务：

1. 在模型详情页面点击 **Create Inference API**。对于 Model Registry 模型，选择活动的 **Version**；对于目录模型，系统会自动选择目录构件。
2. 配置服务参数：
   - Name：meta-llama-3-8b-service
   - Model Source：Model Catalog 或 Model Registry
   - Model：Meta-Llama-3-8B-Instruct
   - Version：Version 1（使用 Model Registry 时）
   - Inference Runtimes：需要根据 GPU 节点中安装的 cuda 版本进行选择。例如，如果安装的是 cuda12.6 或更高版本，请选择“vllm-cuda12.6-x86”。
   - Resource Requests：2CPU/20Gi Memory
   - Resource Limits：2CPU/20Gi Memory

- GPU Acceleration：HAMi NVIDIA
  - gpu number：1
  - vgpu cores：50
  - GPU vmemory：23552
- Storage：使用构件配置的存储，或挂载现有 PVC
- Auto Scaling：关闭
- Number of instances：1

3. 点击 **Publish** 并等待服务启动。
4. 在 **Inference Services** 页面查看服务状态。

步骤 **5**：调用推理服务

测试已发布的推理服务：

1. 点击左侧导航栏中的 **Inference Services**，点击“Published Inference Service”的名称，然后在推理服务详情页面点击 **Experience**。
2. 输入测试文本，例如“推荐几本好书”。
3. 查看模型返回的生成文本和生成参数。

发版日志

目录

Alauda AI 2.8.0

升级支持

自 **Alauda AI 2.3.x** 以来的新功能和优化

平台架构和管理

模型服务和推理

训练和分布式工作负载

MLOps 和开发体验

硬件和模型优化

Alauda AI 2.8.x 的部署形态变更

已废弃功能

破坏性变更

部署和配置

自 **Alauda AI 2.3.x** 以来修复的问题

已知问题

Alauda AI 2.8.0

Alauda AI 2.8.x 总结了自 Alauda AI 2.3.x 之后引入的主要平台、model-serving、training 和 MLOps 变更。

升级支持

Alauda AI 2.8.x 支持从现有的 Alauda AI 2.3.x 环境升级。开始升级前，请先查看升级文档以及各组件特定的迁移要求。

自 **Alauda AI 2.3.x** 以来的新功能和优化

平台架构和管理

- 单集群应用架构 简化了 Alauda AI 2.6.x 及更高版本的部署模型。原有的 **Alauda AI Essentials** (`aml-global`) 组件已移除，Alauda AI 产品入口通过平台集成的产品配置进行配置。
- **Namespace** 管理和权限 允许管理员创建或导入 namespace，并分配 namespace 级别的 owner、editor 和 viewer 权限。原有的平台角色 `aml-namespace-editor`、`owner` 和 `viewer` 已废弃。
- 认证和访问 与 `oauth2-proxy` 集成，并支持独立的 OIDC provider 和访问入口。
- 监控 与 Prometheus 集成，并包含基于 Perses 的 dashboard。
- **Alauda AI Platform Control Plane** 通过 Alauda AI Operator 提供对受支持 AI 组件的集中管理。

模型服务和推理

- 基于镜像的 **Model Catalog** 提供了一个集中式 catalog，用于部署可直接使用的模型镜像，并支持可配置的持久化存储。
- **vLLM-ascend** 通过自定义推理 runtime 支持在 Ascend NPU 上进行模型服务。
- **vLLM Expert Parallel** 支持通过 expert parallelism 提供兼容的 Mixture-of-Experts 模型服务。
- **vLLM speculative decoding** 支持 N-gram 和 EAGLE-3 配置路径，以降低推理延迟。
- **Alauda Build of InferNex Bridge** 为推理工作负载提供弹性编排能力，包括 `InferNexService` 和多节点推理场景。

- **Model as a Service (MaaS)** 提供基于 Alauda Build of Envoy Gateway 和 Alauda Build of Envoy AI Gateway 的托管模型服务网关。管理员可以通过 subscription 发布 `LLMInferenceService` 和外部模型服务，配置令牌限制，为用户分配 subscription，并查看流量和使用情况报告。

训练和分布式工作负载

- **Kubeflow** 组件 升级至 Kubeflow 1.11 release line。Kubeflow Trainer v2 支持在 GPU 和 Ascend NPU 环境中运行分布式训练工作流。
- **JobSet Operator** 支持用于分布式训练、批处理和 HPC 工作负载的协调 Kubernetes Jobs 组。
- **TrainingHub** 增强 为单 GPU、多 GPU、多节点和 Ascend NPU 场景增加了 QLoRA 和 CPT 微调工作流。
- **Kueue** 增强 增加了配额管理、fair sharing、gang scheduling、preemption、cohort，以及支持 checkpoint 和 resume 工作流的可抢占 TrainJobs。
- 动态资源分配（**DRA**） 支持 GPU slicing 和共享 GPU 资源分配，用于训练工作负载。
- **ARM** 支持 扩展到了 Kubeflow Pipelines 和 training runtimes。

MLOps 和开发体验

- **MLflow** 增强 增加了 AI-agent tracing、外部 PostgreSQL 元数据存储和对象存储配置，同时保留平台 SSO 和 Kubernetes RBAC 集成。
- **MLflow Operator 3.13.0** 从 Alauda AI 2.8.x 开始通过 OperatorHub 提供。安装该 Operator 并创建一个 `MLflow` custom resource，即可部署和管理 tracking server。
- **Alauda AI Workbench v0.2.0** 在 Alauda AI 2.8.x 中通过 `default` `AmlCluster` 内的 `workbench` 组件进行管理。Alauda AI 会安装并协调 Workbench Operator 及其资源。
- **Elyra Pipeline Editor** 允许用户从 notebooks 和 scripts 以可视化方式编排 Kubeflow Pipelines workflows。
- 可复用 **pipeline** 组件 和扩展的 Kubeflow Pipeline 机制简化了 pipeline workflows 的编写和复用。
- **Llama Stack** 为 agent 和 retrieval workflows 增加了 PGVector 和 Milvus vector stores。
- **Alauda Build of Spark Operator** 支持将 Spark applications 作为 Kubernetes-native 分布式工作负载运行。

- **coding-agent** 和 **agentic MLOps workflows** 为将 coding agents 连接到本地推理服务提供指导。

硬件和模型优化

- **Alauda Build of NPU Operator** 通过 Alauda AI Operator 进行管理，并支持 Ascend 910B 和不可变操作系统环境。
- **Ascend NPU** 微调 支持 MindSpeed-LLM 和 Kubeflow Trainer v2 workflows，包括 checkpoint 转换、数据集预处理和 supervised fine-tuning。
- **ModelSlim** 集成 为 Ascend NPU 环境提供模型压缩和量化 workflows。

Alauda AI 2.8.x 的部署形态变更

Alauda AI Workbench、MLflow 和 Alauda Build of LeaderWorkerSet 通过 2.8.x 基于 Operator 的部署模型交付。Workbench 和 MLflow 作为独立 Operator 从 **OperatorHub** 安装，而 LWS 通过 `default` `AmlCluster` 中的 `lws` 组件安装并协调：

Component	Alauda AI 2.8.x deployment form	Migration note
MLflow	MLflow OLM Operator	Install the MLflow Operator and create an `MLflow` custom resource.
Alauda Build of LeaderWorkerSet (LWS)	Operator managed through Alauda AI and the `AmlCluster`	Enable the `lws` component in the `default` `AmlCluster`; Alauda AI installs and reconciles LWS.
Alauda AI Workbench	Workbench OLM Operator	Install the Workbench Operator and create a `Workbench` custom resource.

这些部署形态变更并不意味着应删除现有用户数据。请遵循各组件特定的迁移说明，并在迁移期间保留适用的 PVC 和 custom resource。

已废弃功能

- 原有的平台角色 `aml-namespace-editor`、`owner` 和 `viewer` 已废弃。请改用 namespace 级别权限。
- 基于 GitLab 的 Model Catalog 已废弃。对于新的模型交付工作流，请使用带有 OCI model artifacts 的基于镜像的 Model Catalog。
- 原有的 Alauda AI Model Serving operator 已废弃。对于受支持的 serverless 推理场景，请使用 Knative Operator。
- **Alauda Build of NVIDIA GPU Device Plugin** 和 **Alauda Build of DCGM-Exporter** 已废弃。建议使用 **Alauda GPU Management** 作为替代。

破坏性变更

部署和配置

- Alauda AI 2.8.x 使用单集群应用架构。**Alauda AI Essentials** (`aml-global`) 组件不再交付。
- AI header 中移除了 project 和 cluster 切换功能。请使用 namespace 切换来更改工作上下文。
- 登录和登出通过 `oauth2-proxy` 处理。
- MLflow、Alauda Build of LeaderWorkerSet 和 Alauda AI Workbench 从之前的部署形态改为 Operator。现有安装需要进行组件特定迁移，而不是就地包升级。

自 **Alauda AI 2.3.x** 以来修复的问题

- **Alauda Build of TrustyAI**： 修复了某些 TrustyAI 组件缺少 `arm64` 镜像的问题。
- **Alauda Build of Envoy AI Gateway**： 修复了一个问题：当 CRD 在 controller 启动后才安装时，controller 无法处理 `InferencePool` 资源。

以下问题在 Alauda AI 2.3.0 中被记录为已知问题，并已在后续版本中修复：

- **vLLM Ascend**： 修复了多卡 Ascend 推理部署可能无法变为 `Ready` 的问题，因为 HCCL 初始化不支持 root 和 non-root 两种模式。

已知问题

- 当使用 VictoriaMetrics 收集 Serverless 模式下推理服务的监控数据时，推理服务可能无法缩容到 0。

术语表

name	Description
Large Language Model	LLM（Large Language Model）是一种在海量文本数据上训练的 AI 模型，能够理解和生成自然语言，其参数规模通常从数十亿到数千亿不等。
Inference Service	Inference Service 是机器学习或深度学习领域中的一种服务，为已训练模型提供高性能、可扩展的预测或推理能力。
Inference Runtimes	Inference Runtime 为基于模型的推理服务提供高性能容器环境，优化资源利用率，加速推理过程，并降低延迟。
AI Agent	AI Agent 是一种能够感知环境、做出决策并自主执行任务的 AI 实体，具有自主性、适应性和目标导向性。
Text Generation	Text Generation 是使用自然语言处理（NLP）技术，基于给定输入（例如上下文、提示或规则）自动生成连贯且有意义文本内容的过程。
Text Classification	Text Classification 是将文本数据分配到预定义类别或标签中的过程，通常通过机器学习或深度学习模型实现，并用于信息检索、情感分析和垃圾邮件检测等各种应用。
Text-to-Image	Text-to-Image 是使用结合 NLP 和计算机视觉的 AI 技术，将输入文本描述自动生成图像并将文本转换为视觉内容的过程。
Virtual GPU	Virtual GPU（vGPU）是一种利用虚拟化技术将物理 GPU 资源拆分并分配给多个虚拟机的技术，从而实现图形处理能力的共享和高效利用。
Physical GPU	Physical GPU（pGPU）是指将宿主机上的整块物理 GPU 卡直接挂载到虚拟机，以便独占访问 GPU 资源的技术。

规划

在部署前使用本章：请在 受支持的配置 中确认组件版本和架构组合，选择一个 已验证的模型配方，并选择一个 GPU 或 NPU 选项。

受支持的配置表还会索引每个组件，并提供通往各自安装指南的链接。

架构、版本和组件

架构、版本和组件

x86_64 架构

ARM 架构

说明

部署容量规划

部署容量规划

1. Alauda AI 平台资源

2. MaaS SSE 容量参考

已验证的模型

Qwen3-32B

模型标识

已验证硬件 × 栈

模型配置

部署规格

部署

基准测试结果

Qwen3.6-27B (W8A8)

模型标识

已验证硬件 × 技术栈

模型配置

部署规格

部署

基准测试结果

DeepSeek-V

模型标识

已验证硬件 × 栈

模型配置

部署规格

部署

Benchmark 结果

DeepSeek-V4-Flash (W8A8)

模型标识

已验证硬件 × 软件栈

模型配置

部署规格

部署

基准测试结果

MiniMax-M2.5 (W8A8)

模型标识

已验证硬件 × 技术栈

模型配置

部署规范

部署

基准结果

真实 coding-agent 验证

设备方案

关于 Alauda Build of Hami

关于 NVIDIA GPU Device Plugin 的 Alauda Bui

架构、版本和组件

本页列出了组件矩阵中当前维护的 Alauda AI 版本：当前发布版本以及其之前最近的 Stable 版本。

目录

x86_64 架构

ARM 架构

说明

x86_64 架构

Components	Type	Alauda AI v2.3 Stable	Alauda AI v2.8 Stable
Alauda Container Platform Supported Versions		v4.0.x, v4.1.x, v4.2.x, v4.3.x	v4.1.x, v4.2.x, v4.3.x, v4.4.x (1)
Alauda AI Essentials (2)	Cluster Plugin	v2.3.0	Removed
Alauda AI	Operator	v2.3.0	v2.8.1
Alauda AI Workbench	Cluster Plugin	v0.1.7	Replaced (3)

Components	Type	Alauda AI v2.3 Stable	Alauda AI v2.8 Stable
Alauda AI Workbench Operator	Operator	-	v0.2.0
Alauda Build of KServe	Operator	v0.16.0	v0.19.0-1
Alauda Build of Serving Runtime	Operator	-	v1.1.1
Alauda Build of KubeRay Operator	Cluster Plugin	v1.6.0	v1.6.0
Alauda Build of NVIDIA GPU Device Plugin	Cluster Plugin	v0.18.4	Deprecated (7)
Alauda Build of NVIDIA DRA Driver for GPUs	Cluster Plugin	v25.8.1	v25.8.1
Alauda Build of DCGM-Exporter	Cluster Plugin	v4.2.3-413-1	Deprecated (7)
Alauda GPU Management	Operator	-	v26.3.3-1
Alauda Build of HAMi	Cluster Plugin	v2.8.1	v2.9.0
Alauda Build of HAMi-WebUI	Cluster Plugin	v1.10.0	v1.10.3
Alauda Build of Node Feature Discovery	Cluster Plugin	v0.17.4	v0.18.3
Alauda Build of Kueue	Cluster Plugin	v0.17.0	v0.17.0
Alauda Build of LeaderWorkerSet	Cluster Plugin	v0.8.0-1	Replaced
Alauda Build of LeaderWorkerSet	Operator	-	v0.9.0-1
Alauda Build of JobSet	Operator	-	v0.12.0

Components	Type	Alauda AI v2.3 Stable	Alauda AI v2.8 Stable
Volcano	Cluster Plugin	v1.12.4	v1.15.0
MLFlow	Cluster Plugin	v3.1.5	Replaced
MLFlow Operator	Operator	-	v3.13.0
Kubeflow Base	Cluster Plugin	v1.10.14-1	Replaced
Kubeflow Base Operator	Operator	-	v26.3.5
Kubeflow Pipelines	Cluster Plugin	v1.10.13	Replaced
Kubeflow Pipelines Operator	Operator	-	v26.3.5
Kubeflow Trainer v2	Cluster Plugin	v1.10.13	Replaced
Kubeflow Trainer Operator	Operator	-	v26.3.5
Data Science Pipeline Operator	Operator	-	v2.15.1
Kubeflow Model Registry	Operator	v1.10.13	Replaced
Alauda Build of Kubeflow Model Registry	Operator	-	v0.3.8-2
Alauda Build of Llama Stack	Operator	v0.8.0	v0.9.0
Label Studio	Helm Chart	v1.21.0-2	v1.21.0-2
Alauda Build of Envoy AI Gateway	Cluster Plugin	v0.4.1	Replaced
Alauda Build of Envoy AI Gateway	Operator	-	v0.6.0-2
Dify	Helm Chart	v1.11.4	Replaced

Components	Type	Alauda AI v2.3 Stable	Alauda AI v2.8 Stable
Dify Operator	Operator	-	v1.15.0
Langflow (4)	Helm Chart	v1.6.4-1	Deprecated
Evidently	Helm Chart	v0.7.14-1	v0.7.14-1
Featureform (4)	Helm Chart	v0.12.1-2	Deprecated
Alauda Build of Feast	Operator	v0.61.1	v0.61.1
Knative Operator	Operator	v1.19.3-260213	v1.19.3-260213
PostgreSQL (5)	Operator	v4.2.0	v4.3.3
Alauda Cache Service for Redis OSS (5)	Operator	-	v5.0.2
Alauda Build of Authorino (5)	Operator	-	v0.26.0-1
Milvus Operator	Helm Chart	v1.3.5	v1.3.5
Alauda Build of Gitlab (6)	Operator	v18.5.1	-
Alauda Build of Harbor (6)	Operator	-	v2.4.14
Alauda Build of TrustyAI	Operator	v3.4.1	v3.4.2-2
Alauda Build of MCP Lifecycle Operator	Operator	-	v0.2.0-1
Kagenti Operator	Operator	-	v0.3.0-rc.1
Kagenti UI Operator	Operator	-	v0.6.1

ARM 架构

Components	Type	Alauda AI v2.3 Stable	Alauda AI v2.8 Stable
Alauda Container Platform Supported Versions		v4.0.x, v4.1.x, v4.2.x, v4.3.x	v4.1.x, v4.2.x, v4.3.x, v4.4.x
Alauda AI Essentials	Cluster Plugin	v2.3.0	Removed
Alauda AI	Operator	v2.3.0	v2.8.1
Alauda AI Workbench	Cluster Plugin	v0.1.7	Replaced
Alauda AI Workbench Operator	Operator	-	v0.2.0
Alauda Build of KServe	Operator	v0.16.0	v0.19.0-1
Alauda Build of KubeRay Operator	Cluster Plugin	v1.6.0	v1.6.0
Alauda Build of NVIDIA GPU Device Plugin	Cluster Plugin	v0.18.4	Deprecated (7)
Alauda Build of NVIDIA DRA Driver for GPUs	Cluster Plugin	v25.8.1	v25.8.1
Alauda Build of DCGM-Exporter	Cluster Plugin	v4.2.3-413-1	Deprecated (7)
Alauda Build of NPU Operator	Cluster Plugin	v1.1.3	Replaced
Alauda Build of NPU Operator	Operator	-	v26.6.0
Alauda Build of HAMi	Cluster Plugin	v2.8.1	v2.9.0
Alauda Build of HAMi-WebUI	Cluster Plugin	v1.10.0	v1.10.3
Alauda Build of HAMi Ascend Device Plugin	Operator	-	v1.4.0

Components	Type	Alauda AI v2.3 Stable	Alauda AI v2.8 Stable
Alauda Build of InferNex Bridge	Operator	-	v26.6.0
Alauda Build of Node Feature Discovery	Cluster Plugin	v0.17.4	v0.18.3
Alauda Build of Kueue	Cluster Plugin	v0.17.0	v0.17.0
Alauda Build of LeaderWorkerSet	Cluster Plugin	v0.8.0-1	Replaced
Alauda Build of LeaderWorkerSet	Operator	-	v0.9.0-1
Alauda Build of JobSet	Operator	-	v0.12.0
Volcano	Cluster Plugin	v1.12.4	v1.15.0
MLFlow	Cluster Plugin	v3.1.5	Replaced
MLFlow Operator	Operator	-	v3.13.0
Kubeflow Base	Cluster Plugin	v1.10.14-1	Replaced
Kubeflow Base Operator	Operator	-	v26.3.5
Kubeflow Pipelines	Cluster Plugin	v1.10.13	Replaced
Kubeflow Pipelines Operator	Operator	-	v26.3.5
Kubeflow Trainer v2	Cluster Plugin	v1.10.13	Replaced
Kubeflow Trainer Operator	Operator	-	v26.3.5
Data Science Pipeline Operator	Operator	-	v2.15.1
Kubeflow Model Registry	Operator	v1.10.13	Replaced

Components	Type	Alauda AI v2.3 Stable	Alauda AI v2.8 Stable
Alauda Build of Kubeflow Model Registry	Operator	-	v0.3.8-2
Alauda Build of Llama Stack	Operator	v0.8.0	v0.9.0
Label Studio	Helm Chart	v1.21.0-2	v1.21.0-2
Alauda Build of Envoy AI Gateway	Cluster Plugin	v0.4.1	Replaced
Alauda Build of Envoy AI Gateway	Operator	-	v0.6.0-2
Dify	Helm Chart	v1.11.4	Replaced
Dify Operator	Operator	-	v1.15.0
Langflow	Helm Chart	v1.6.4-1	Deprecated
Evidently	Helm Chart	v0.7.14-1	v0.7.14-1
Alauda Build of Feast	Operator	v0.61.1	v0.61.1
Knative Operator	Operator	v1.19.3-260213	v1.19.3-260213
PostgreSQL	Operator	v4.2.0	v4.3.3
Alauda Cache Service for Redis OSS	Operator	-	v5.0.2
Alauda Build of Authorino	Operator	-	v0.26.0-1
Milvus Operator	Helm Chart	v1.3.5	v1.3.5
Alauda Build of Gitlab	Operator	v18.5.1	-
Alauda Build of Harbor	Operator	-	v2.4.14

Components	Type	Alauda AI v2.3 Stable	Alauda AI v2.8 Stable
Alauda Build of TrustyAI	Operator	v3.4.1	v3.4.2-2
Alauda Build of MCP Lifecycle Operator	Operator	-	v0.2.0-1
Kagenti Operator	Operator	-	v0.3.0-rc.1
Kagenti UI Operator	Operator	-	v0.6.1

说明

(1) **Alauda AI** v2.8 不再支持 **Alauda Container Platform** v4.0.x。直到 **Alauda Container Platform** v4.0.x 达到生命周期终止之前，仍继续支持 **Alauda AI** v2.3.x。

(2) **Alauda AI Essentials** 在 **Alauda AI** v2.6 之后移除。

(3) *Replaced* 表示组件的部署方式已变更；Operator 替代了 Cluster Plugin 或 Helm Chart。

(4) **Featureform** 和 **Langflow** 已弃用，不应用于新的部署。

(5) **PostgreSQL**、**Alauda Cache Service for Redis OSS** 和 **Alauda Build of Authorino** 是 MaaS 的依赖项。

(6) 模型存储已从 GitLab 改为 container registry，建议使用 Harbor 存储模型。

(7) **Alauda Build of NVIDIA GPU Device Plugin** 和 **Alauda Build of DCGM-Exporter** 已弃用，建议使用 **Alauda GPU Management** 进行替代。

部署容量规划

本页用于估算部署 Alauda AI 基础功能所需的组件资源，以及 MaaS 长流式 SSE 场景所需的额外资源。

目录

1. Alauda AI 平台资源
   1.1 资源建议
   1.2 存储规划
2. MaaS SSE 容量参考
   2.1 未启用 Guardrail
   2.2 启用 Guardrail
   2.3 解读压测数据
   2.4 MaaS 扩缩容参考

1. Alauda AI 平台资源

下列资源覆盖 Alauda AI 基础功能和固定平台组件，不包含 ACP 和 Kubernetes 资源。不同的 ACP 和 Kubernetes 版本、集群规模以及启用特性具有不同的基线资源需求，因此请在 Alauda AI 资源之外单独规划这些部分。

Alauda AI 平台资源还不包括以下工作负载：

- 训练、微调和评估任务；
- 模型推理服务和推理副本；
- Notebook、Workspace、PipelineRun 和 Agent 实例；
- 模型权重、训练数据、镜像、日志和备份的存储。

训练任务、推理服务及其他 AI 工作负载需要根据任务类型、模型大小、并发数和性能目标额外分配资源，例如 GPU/NPU 卡、CPU、内存、节点和业务存储。

1.1 资源建议

下表仅展示 Alauda AI 组件的参考值，而不是物理服务器的最终数量或规格。在准备服务器时，请在这些数值之上叠加 ACP、Kubernetes、操作系统、容器运行时、存储、监控和日志以及其他平台特性所需的资源。

部署场景	**Alauda AI** 组件资源	适用场景	说明
高可用部署	**48** 核 **CPU / 72 GiB** 内存	生产环境、多用户共享环境，以及需要控制平面高可用的场景	推荐部署在 3 台服务器上，每台为 Alauda AI 预留约 16 核 CPU / 24 GiB 内存；不包含 ACP 和 Kubernetes 基线资源
Demo / 试用最低配置	**24** 核 **CPU / 36 GiB** 内存	功能演示、兼容性验证和短期试用	采用单副本部署形态，包含 Alauda AI 基础平台组件以及训练/微调/Agent 控制平面，并预留演示数据和演示工作负载空间；不具备高可用性

高可用形态包含 Alauda AI 基础平台组件以及训练、微调和 Agent 控制平面，并在组件资源之上额外预留约 50% 的余量，用于应对节点故障后的重新调度、滚动升级、监控和日志波动以及业务增长。建议 Alauda AI 部署在 3 台服务器上，每台为 Alauda AI 预留约 16 核 CPU / 24 GiB 内存；ACP 和 Kubernetes 基线资源需根据实际环境在此基础上额外增加。

Demo / 试用形态采用单副本部署形态，包含 Alauda AI 基础平台组件以及训练、微调和 Agent 控制平面，并预留演示数据和演示工作负载空间；不具备高可用性。如果 Demo 或高可用部署同时运行真实训练任务、推理服务、MaaS 并发、用户 Notebook/Workspace/PipelineRun、Agent 实例或大规模业务数据，请根据实际工作负载额外增加资源。

以上数值仅代表 Alauda AI 组件资源，不包含 ACP 和 Kubernetes 的基线资源。

1.2 存储规划

Alauda AI 平台资源不包括业务数据存储容量。请根据数据量、增长速率、保留周期、IOPS、吞吐量和恢复目标，单独规划模型权重、训练数据、模型镜像、Workspace/PVC、Pipeline 制品、日志和备份。

2. MaaS SSE 容量参考

压测数据有助于确定各并发级别下 MaaS 组件的合理资源配置。它是在基础平台资源之外对 MaaS 容量进行定容的参考。数据来源于长流式 SSE 测试：每个流返回 200 个内容块，间隔约 100 ms，每次测试持续 600 秒，并使用关闭连接模式。测试使用集群内的 `mock-llm` 作为后端替代真实模型服务，以减少模型响应内容和持续时间的不确定性对结果的影响。

2.1 未启用 Guardrail

下表展示了各并发级别的压测结果；每次测试都在 600 秒内完成了全部请求，且无中断或错误。延迟单位为毫秒。`TTFT` 表示首个 token 时间，流持续时间表示单个 SSE 流从开始到结束的时长；两者均以 avg / p95 形式给出。

并发数	**Gateway** 配置	**Authorino** 配置	**Ratelimit** 配置	**TTFT (ms)**
500	2 个副本；每个 1 CPU / 512 MiB	2 个副本；每个 0.5 CPU / 256 MiB	2 个副本；每个 0.25 CPU / 256 MiB	avg 140 / p95 1,517
1,000	2 个副本；每个 2 CPU / 512 MiB	2 个副本；每个 1.5 CPU / 256 MiB	2 个副本；每个 0.25 CPU / 256 MiB	avg 179 / p95 470
2,000	4 个副本；每个 2 CPU / 512 MiB	3 个副本；每个 2 CPU / 256 MiB	2 个副本；每个 0.25 CPU / 256 MiB	avg 159 / p95 506

2.2 启用 Guardrail

Gateway、Authorino 和 Ratelimit 采用与未启用 Guardrail 的测试相同的配置：

- Gateway：2 个副本；每个 1 CPU / 512 MiB
- Authorino：2 个副本；每个 0.5 CPU / 256 MiB
- Ratelimit：2 个副本；每个 0.25 CPU / 256 MiB

Guardrail 测试额外引入了 NeMo engine 和 Guardrail `ext_proc` shim。每次测试都在 600 秒内完成了全部请求，且无中断或错误。延迟单位为毫秒；`TTFT` 和流持续时间均以 avg / p95 形式给出。

并发数	NeMo engine	Guardrail shim	TTFT (ms)	流持续时间 (ms)
100	3 个副本；每个 1 CPU / 1 GiB	2 个副本；每个 0.5 CPU / 256 MiB	avg 338 / p95 846	avg 20,391 / p95 20,906
200	6 个副本；每个 1 CPU / 1 GiB	2 个副本；每个 0.5 CPU / 256 MiB	avg 343 / p95 793	avg 20,392 / p95 20,848
500	15 个副本；每个 1 CPU / 1 GiB	2 个副本；每个 0.5 CPU / 256 MiB	avg 383 / p95 985	avg 20,432 / p95 21,034

2.3 解读压测数据

CPU 是一种弹性资源。单个 CPU 的实际处理能力在不同 CPU 型号、微架构、主频、NUMA 拓扑、虚拟化模式和运行环境之间可能存在显著差异。因此：

- 压测数据仅作为容量规划参考，不作为通用配置承诺；
- 不同 CPU 型号上的 `1 CPU` 并不一定带来相同的吞吐量和延迟；
- 请根据目标 CPU 型号、模型大小、请求长度、输出速度、连接模式、Guardrail 策略和目标并发数调整 CPU 配置；
- 在 ARM64、Kunpeng 或其他目标硬件上，请在目标硬件上运行业务压测；

- 当测试结果与本页面不一致时，以实际业务测试结果为准。

MaaS 压测不包含真实模型推理服务、模型权重或 GPU/NPU 资源。在部署真实推理服务时，请根据模型大小、量化方式、上下文长度、并发数和目标延迟，增加 GPU/NPU、CPU、内存和存储资源。

2.4 MaaS 扩缩容参考

关于各 MaaS 组件的扩缩容方式和资源配置入口，请参见 MaaS 组件扩缩容。

在扩缩容层面：

- **Gateway**：CPU 密集型组件。较高的 CPU 利用率属于正常现象；请配置告警以避免误报，并在利用率过高时增加副本数以保持性能；
- **Authorino**：对鉴权延迟较为敏感。即使 CPU 尚未饱和，请求也可能排队并超时，返回 `403`；此时应通过增加副本数进行横向扩展；
- **NeMo engine**：采用 Python 实现，因此单个实例无法高效利用多个 CPU 核；应通过增加副本数进行横向扩展。

已验证的模型

面向 Alauda AI 上 已验证的开放权重 **LLM** 的即用型部署方案。
本指南中的每个模型都已在真实集群上完成端到端部署并经过基准测试，因此你可以获得一个已知可用的部署清单、为其提供服务的运行时镜像，以及可预期的吞吐量。

这里的模型均在 **Huawei Ascend NPUs**（**910B4** 和 **910B3**） 上，使用社区 `vLLM-Ascend` 引擎进行验证，并通过 Alauda AI 的 **InferNex** 接入层部署——即由 InferNex-Bridge 将 KServe `LLMInferenceService` 协调为一个具备负载感知能力的路由器（hermes-router / EPP），位于 vLLM-Ascend 实例之前。大多数模型都通过了相同的 **InferNex** 聚合入口和相同的两个基准场景（两个 Qwen 模型共享完全相同的 2 × TP=4 拓扑，因此可以直接比较；DeepSeek-V4-Flash 和 MiniMax-M2.5 使用更大的 MoE 拓扑，最多达到 16 卡的 DP2 × TP=8 跨节点聚合）。关于运行时模型（KServe、ModelCar 存储、调度），请参见 部署。

目录

已验证的模型
运行时镜像
基准场景
部署已验证的模型
注意事项
验证 ModelCar 签名

已验证的模型

模型	类型	参数	**dtype**	设备	引
Qwen3-32B	Qwen3 dense	32B	BF16	Ascend 910B4 ×8	vl v(
Qwen3.6-27B (W8A8)	`qwen3_5` hybrid (GDN + MTP)	27.78B	W8A8	Ascend 910B4 ×8	vl ni
DeepSeek-V4-Flash (W4A8)	`deepseek_v4` MoE (MLA + DSA + MTP)	256-expert MoE	W4A8	Ascend 910B4 ×8	vl ni
DeepSeek-V4-Flash (W8A8)	`deepseek_v4` MoE (MLA + DSA + MTP)	256-expert MoE	W8A8	Ascend 910B3 ×16	vl ni
MiniMax-M2.5 (W8A8)	`minimax_m2` MoE (full-attn, 256-expert)	230B-A10B	W8A8 (QuaRot)	Ascend 910B3 ×16	vl v(

两个 Qwen 模型和 DeepSeek-V4-Flash（W4A8）均在 Ascend 910B4（每卡 32 GB）上完成验证，并通过具备负载感知路由的 KServe `LLMInferenceService`（InferNex-Bridge + hermes-router）进行驱动。两个 Qwen 模型运行在**8** 卡聚合上——2 个实例 × TP=4；DeepSeek-V4-Flash（约 151 GB W4A8）以**1** 个实例 **× TP=8** 的方式占满全部 8 卡，并额外验证了位于内部 KServe 入口旁的 **MaaS gateway**（**API-key**）入口。

DeepSeek-V4-Flash（W8A8，约 280 GB）在 **Ascend 910B3**（每卡 **64 GB**）双节点共 **16** 卡上进行了验证，作为单个 **DP2 × TP=8** 跨节点聚合运行，并使用 **mooncake** 跨 **rank KV** 存储，同时通过内部 KServe 入口和产品的 MaaS gateway 进行驱动。

MiniMax-M2.5（W8A8，约 230 GB，230B-A10B）在相同的 **Ascend 910B3 16** 卡 **DP2 × TP=8** 跨节点聚合上进行了验证，但作为一个纯 `agg` 基线——不使用 mooncake 存储，也不使用 speculative decoding。由于该模型属于“快速”类型（约 10B 激活

- full attention + W8A8），这两个附加功能都被测得会*降低*性能，因此生产配置只保留 decode graph + 本地 prefix cache。它还通过产品 MaaS gateway 额外验证为一个真实的编码 **Agent**（OpenCode / Pi）。

运行时镜像

引擎	设备	镜像（已验证标签）	由以下模型使用	说明
vLLM-Ascend v0.18.0	Huawei Ascend NPU	`quay.io/ascend/vllm-ascend:v0.18.0-openeuler`	Qwen3-32B	面向 Ascend 社区版 vLLM V1 引擎。可工接服务标准的 Qwen3 dens 模型。请将该签对应的 CA 版本与宿主机 NPU 驱动保持一致。
vLLM-Ascend nightly（固定版本）	Huawei Ascend NPU	`quay.io/ascend/vllm-ascend:nightly-releases-v0.22.1rc-openeuler`	Qwen3.6-27B (W8A8), DeepSeek-V4-Flash (W4A8)	该镜像包含 `qwen3_5` Gated DeltaN hybrid + MTP 及 `deepseek_v` MoE 栈——标准的 `v0.18.` 无法加载其中一者。请使用个固定版本标签，不要使用断变化的 `nightly-main-openeuler` 已漂移到一个坏的构建，TP worker 会崩溃）。
vLLM-Ascend nightly-main（指定日期快照）	Huawei Ascend NPU	`quay.io/ascend/vllm-ascend:nightly-main-openeuler-0709`	DeepSeek-V4-Flash (W8A8)	一个更晚的 `nightly-ma` 快照（vLLM 0.23.0），包 DP2 跨节点图

引擎	设备	镜像（已验证标签）	由以下模型使用	说明
				DSA-CP pref cache 修复，足 W8A8 跨节部署需求。请定使用像该镜这样的指定日快照，而不是断变化的 `nightly-ma` 标签。
vLLM-Ascend v0.23.0（固定版本）	Huawei Ascend NPU	`quay.io/ascend/vllm-ascend:v0.23.0-openeuler`	MiniMax-M2.5 (W8A8)	固定版本的 `nightly-releases-v0.23.0-openeuler` 建（vLLM 0.23.0），包 `minimax_m2` MoE 栈 + toc call/reasonin 解析器 以及 vLLM #1150 （流式 tool-c 参数会保持到 合的 `</paramete` 之前）。请优使用这个指定期的 **release** 签，而不是不变化的 `nightly-main`。

TIP

Ascend CANN 镜像为 arm64。始终将运行时镜像的 CANN 版本与节点上的宿主机 **NPU** 驱动保持一致。本指南仅列出实际使用到的引擎；其他引擎（MindIE、SGLang 等）未在该规模下进行基准测

试。

基准场景

这些模型使用 `aiperf` 在相同的两个场景下进行了测量，这两个场景模拟真实的服务模式。输出固定为 **128 tokens**，负载采用 **closed-loop**、并发 **4**（固定 4 个在途请求）；每个场景执行 240 个请求。DeepSeek-V4-Flash（W8A8）和 MiniMax-M2.5（W8A8）是例外——它们分别在 并发 **8 / 16 / 32** 下进行扫描测试（每个档位 480 个请求）。

场景	建模内容	数据集	请求形态
① 固定长度 **system-prompt** 复用	重复使用的 system prompt——衡量请求长度固定且 KV cache 重复命中时的性能。	60 个不同的 8k-token system prompt，每个复用 4 次 = 240 个请求。	ISL ~8k / OSL 128
② 多轮对话	多个用户在持续对话中交互，每个 session 包含多个请求。	60 个独立用户 × 4 轮 = 240 个请求。	第 1 轮发送 16k，模型回复 128 tokens；第 2 轮再增加 1k（总计约 17k）；依此类推共 4 轮（16k / 17k / 18k / 19k），平均 约 **17.5k** ISL / OSL 128。

NOTE

两个 Qwen 模型和 DeepSeek-V4-Flash（W4A8）在单节点 **8** 卡部署上运行这些场景（Qwen 模型：2 个实例 × TP=4；DeepSeek-V4-Flash W4A8：1 个实例 × TP=8）。延迟（TTFT / ITL / E2E）是在稳定的 2 在途请求负载下、以每个实例为单位的运行点；总吞吐量（TPS）是所有实例的聚合值，并随实例数量扩展。**TPS** 是总 **token**（输入 **+** 输出）口径；仅解码输出速率单独报告，并且在这些长输入工作负载下会小得多。

DeepSeek-V4-Flash（W8A8）和 MiniMax-M2.5（W8A8）则运行在 **16** 卡 **DP2 × TP=8** 跨节点部署上，并在 并发 **8 / 16 / 32** 下进行扫描（每个档位 480 个请求），因此它们的数值是按各自模型页面报告的运行区间。两者都通过内部 KServe 入口和产品 **MaaS gateway**（**API-key**）入口进行驱动。

部署已验证的模型

每个模型页面都链接了位于 `assets/` 下的自包含 YAML，这些 YAML 包含真实的 InferNex 部署——一个 KServe `LLMInferenceService`（`infernex.io/runtime: true`）以及两个 `LLMInferenceServiceConfig` 对象（engine template + hermes-router/EPP template），由 InferNex-Bridge 将它们协调为正在运行的实例。

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/plan/validated_models/assets

# 1. Edit the file first: set the namespace and the image tag (match your CANN /
#    host driver). model.uri already points at the public ModelCar on Docker Hub
#    (oci://docker.io/alaudadockerhub/...) — repoint it only if you mirror locally.
# 2. Apply — the InferNex-Bridge reconciles the LLMInferenceService into the
#    hermes-router (EPP) + the vLLM-Ascend instances.
kubectl apply -f $base/qwen3-32b/qwen3-32b-agg-base-llmisvc.yaml

# 3. Watch it come up (first start loads weights + compiles graphs — can take minutes).
kubectl -n <your-namespace> get llminferenceservice -w

# 4. Call the OpenAI-compatible endpoint through the gateway.
curl -s http://<gateway>/<namespace>/qwen3-32b-agg-base/v1/chat/completions \
  -H 'Content-Type: application/json' \
  -d '{"model":"Qwen/Qwen3-32B","messages":[{"role":"user","content":"hello"}]}'
```

注意事项

- 这些清单通过 **InferNex**（`LLMInferenceService` + InferNex-Bridge - hermes-router）进行部署。两个 `LLMInferenceServiceConfig` 对象位于 `kserve` 命名空间；`LLMInferenceService` 位于你的部署命名空间。

- 资源键适用于 **Ascend 910B4**（`huawei.com/Ascend910`）。请根据你实际的 NPU 型号调整资源键、镜像和版本字段。
- ModelCar 镜像在 Docker Hub 上通过 `alaudadockerhub` ↗ 公开提供——这些清单会在无需凭据的情况下拉取它们。如果你愿意，可以将其镜像到自己的 registry 并重新指向 `model.uri`；清单中的 modelcar pull secret 仅在私有 registry 场景下需要。
- 基准数据是在 8 卡上以 **closed-loop**（并发 4）方式测得。请将它们视为稳定负载下的每实例运行点，而不是饱和上限。

验证 ModelCar 签名

ModelCar 镜像使用 Cosign ↗ 进行签名。部署前，请使用已发布的公钥（`cosign.pub` ↗）验证镜像：

```
curl -sO https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/plan/validated_models/cosign.pub

cosign verify --key cosign.pub --insecure-ignore-tlog=true \
  docker.io/alaudadockerhub/modelcar-qwen3-32b:v0.1.0
```

三个已签名镜像及其摘要如下：

镜像	摘要
`docker.io/alaudadockerhub/modelcar-qwen3-32b:v0.1.0`	`sha256:eccffdb567038196638eb7e1c6bb9572d8bcc782`
`docker.io/alaudadockerhub/modelcar-qwen3-6-27b-w8a8:v0.1.0`	`sha256:5eb8841a29a4f3d659f8cef25c7f8b16208b3226`
`docker.io/alaudadockerhub/modelcar-deepseek-v4-flash-w4a8-mtp:v0.1.0`	`sha256:ef57a46726bec9c6505504262beaa55ec6c45d59`

> **NOTE**
>
> DeepSeek-V4-Flash（**W8A8**）和 MiniMax-M2.5（**W8A8**）不在此 Cosign 表中——它们的 ModelCar 以 **OCI Image Layout tar** 的形式分发（而不是已签名的 registry 镜像），因此没有可供验

证的 Cosign 签名。其完整性通过内容寻址来保证：每个 tar 的 `index.json` 都包含镜像摘要，且 `blobs/sha256/` 下的每个 blob 在导入时都会根据自身摘要进行校验（`skopeo copy` 会执行该检查）。请参见 DeepSeek-V4-Flash (W8A8) 和 MiniMax-M2.5 (W8A8) 页面。

之所以需要 `--insecure-ignore-tlog=true`，是因为这些镜像使用 `--tlog-upload=false` 签名（没有公开的 transparency log 条目）；验证仅依赖公钥本身。

Qwen3-32B

标准 **Qwen3 dense** 模型，32B 参数，以 **BF16** 提供服务（4 张卡合计约 ~62 GB）。已在 **Ascend 910B4**（**32 GB/卡**） 上，使用 vLLM-Ascend **v0.18.0**，并通过 Alauda AI 的 InferNex 接入层完成验证。已验证的 8 卡拓扑为 **TP=4 × 2** 个副本，在 两个部署规格 下进行了基准测试——`agg-base`（负载均衡）和 `agg-mc-kv`（跨实例 KV cache 存储 + 感知 KV-cache 的路由）。

目录

模型标识
已验证硬件 × 栈
模型配置
部署规格
部署
基准测试结果

模型标识

字段	值
发布方	Qwen (Alibaba)

字段	值
架构	Qwen3 dense（标准 attention，无 MTP）
参数	32B
原生 dtype	BF16 (~62 GB)
模型来源	https://huggingface.co/Qwen/Qwen3-32B ↗

已验证硬件 × 栈

平台	引擎	版本 / 配置	状态
Ascend 910B4 32 GB × 8（TP=4 × 2 个副本）	vLLM-Ascend	`v0.18.0-openeuler`（V1 engine）	✅ 闭环，2 场景性能测试，两个部署规格（agg-base + agg-mc-kv）

模型配置

定义服务边界的 vLLM-Ascend 引擎标志（两个部署规格完全相同）：

参数	值
Tensor parallelism (`tensor-parallel-size`)	4
副本（实例）	2（= 8 张卡）
`max-model-len`	40960
`max-num-batched-tokens`	40960
`gpu-memory-utilization`	0.8
`block-size`	128

参数	值
`max_tokens`（输出，基准测试固定）	128
Prefix caching	已启用（`--enable-prefix-caching`）

部署规格

两个规格都通过 InferNex 运行在同一组 8 张卡（2 × TP=4）上（KServe `LLMInferenceService` + InferNex-Bridge + hermes-router）。`agg-mc-kv` 在 `agg-base` 的基础上增加了跨实例 KV cache 存储和感知 KV-cache 的路由：

组件	`agg-base`	`agg-mc-kv`
hermes-router (EPP)	✅ 已启动	✅ 已启动
路由策略	`random`（负载均衡）	`kv-cache-aware`
cache-indexer	—	✅
mooncake KV store (AscendStoreConnector)	—	✅
tokenizer sidecar	—	✅
跨实例 KV 复用	否	是

部署

独立的 InferNex 清单（每个都打包了 engine + hermes-router `LLMInferenceServiceConfig` 和 `LLMInferenceService`，2 个副本 × TP=4）：

规格	文件
agg-base（负载均衡）	`qwen3-32b-agg-base-llmisvc.yaml` ↗

规格	文件
agg-mc-kv（KV store + 感知 KV-cache 的路由）	`qwen3-32b-agg-mc-kv-llmisvc.yaml` ↗

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/plan/validated_models/assets/qwen3-32b
# edit namespace / model.uri registry / image tag first, then:
kubectl apply -f $base/qwen3-32b-agg-base-llmisvc.yaml
```

基准测试结果

闭环 `aiperf 0.7.0`，**TP=4 × 2** 个副本（**8 × 910B4**），并发 4。两个 场景——① 8k system-prompt 复用和 ② 17.5k 多轮对话——每个 240 个请求， 输出固定为 128 tokens。TTFT / E2E 的单位为 ms，ITL 的单位为 ms，TPS = 总 tokens/s（输入 + 输出）。

场景 ① — 固定长度 **system-prompt** 复用（**ISL ~8k / OSL 128**）

部署规格	TTFT avg (ms)	ITL avg (ms)	E2E avg (ms)	TPS (in+out)
agg-base	1328	71.4	10391	3120
agg-mc-kv	445	68.7	9165	3558

场景 ② — 多轮对话（**ISL ~17.5k / OSL 128**）

部署规格	TTFT avg (ms)	ITL avg (ms)	E2E avg (ms)	TPS (in+out)
agg-base	4612	95.9	16791	4270
agg-mc-kv	1663	77.3	11481	6498

NOTE

如何解读这些结果。 所有运行都以每个实例稳定 2 个在途请求完成了 240/240， 且没有任何错误。`agg-base` 的数据来自一次具有代表性的运行；`agg-mc-kv` 是 3 次运行的平均值。

`agg-mc-kv` 的价值主要体现在 **TTFT**（首 token 延迟）上：跨实例 KV 存储允许 一个请求复用其他实例已经计算好的前缀 KV，从而跳过一次重新 prefill。该效果在 多轮对话 场景中最明显（TTFT 4612 → 1663 ms，E2E 16.8 → 11.5 s，TPS +52%），因为每一轮都共享一个持续增长的长前缀。在这 3 次运行中，场景 ② 的冷/热态差异 很大：第一次（冷态）运行 需要完整完成 16k base 的 prefill（TTFT ~3.5 s），而 热态运行 的 TTFT 可达到 ~0.76 s、E2E ~9.9 s——上面的均值同时包含了两者，而热态稳态才是持续服务中的实际运行点。decode step 延迟（ITL）在两个规格之间 基本相同——KV store 并不会加速这一部分。

在当前 2 实例规模下，`agg-mc-kv` 的大部分收益来自 KV **store**；若要单独隔离 路由带来的贡献，需要额外做一组 A/B 测试，而 store 的价值会随着实例数量增加而 提升（跨实例复用机会更多）。这些是半规模数据（8 张卡）；总体吞吐量会随实例数 扩展。

Qwen3.6-27B (W8A8)

`qwen3_5` 架构模型 — Gated DeltaNet 线性注意力混合架构（3:1 线性 / 全注意力），带原生 MTP speculative head，27.78B 参数，作为 **W8A8**（INT8 权重，约 33 GB）提供服务。已在 **Ascend 910B4**（每卡 **32 GB**） 上通过 Alauda AI 的 InferNex 接入面，使用 vLLM-Ascend **nightly** 引擎完成验证。已验证的 8 卡拓扑为 **TP=4 × 2 replicas**，部署规格为 `agg-base`。

目录

模型标识
已验证硬件 × 技术栈
模型配置
部署规格
部署
基准测试结果

模型标识

字段	值
发布方	Qwen (Alibaba)

字段	值
架构	`qwen3_5` — Gated DeltaNet 线性注意力混合架构 + 全注意力 + 原生 MTP（支持多模态）
参数量	27.78B（64 layers，hidden 5120）
原生 dtype	BF16（约 54 GB）；此处作为 **W8A8**（约 33 GB）提供服务
模型来源（W8A8）	https://www.modelscope.cn/models/Eco-Tech/Qwen3.6-27B-w8a8 ↗

已验证硬件 × 技术栈

平台	引擎	版本 / 配置	状态
Ascend 910B4 32 GB × 8（TP=4 × 2 replicas）	vLLM-Ascend	`nightly-releases-v0.22.1rc-openeuler`（vLLM 0.22.1，CANN 9.0.0）	✅ 闭环、2 场景性能测试（n=3），agg-base

NOTE

`qwen3_5`（Gated DeltaNet 混合 + 多模态）只能在 vLLM-Ascend **nightly** 镜像上加载。请使用固定到发布版本的 `nightly-releases-v0.22.1rc-openeuler` 标签——持续变化的 `nightly-main-openeuler` 已漂移到一个损坏的构建，TP workers 会崩溃。原生 `v0.18.0` 无法提供服务。W8A8 量化可以节省 HBM（从而获得更大的 KV cache），但其本身不会加速 decode——瓶颈在于 nightly 引擎中的 `qwen3_5` GDN/MTP decode 路径，而不是内存带宽。

模型配置

参数	值
Tensor parallelism (`tensor-parallel-size`)	4
Replicas (instances)	2 (= 8 cards)
`max-model-len`	24576
`max-num-batched-tokens`	8192
`gpu-memory-utilization`	0.85（不是 0.90——在启用 MTP 且高并发时会 OOM）
`max-num-seqs`	32（必需的保护上限）
`max_tokens`（输出，benchmark 固定）	128
Quantization	`ascend`（W8A8）
Prefix caching	禁用（`--no-enable-prefix-caching`，GDN 要求）
Speculative decoding (MTP)	`qwen3_5_mtp`，3 tokens

部署规格

此模型仅以 **`agg-base`** 方式提供服务——聚合，hermes-router 策略为 `random`（负载均衡），不使用 mooncake KV store。跨实例 KV store / 支持 KV-cache 感知路由的方式（`agg-mc-kv`）目前还不能用于 Ascend 上的 `qwen3_5` GDN 混合架构：该混合线性注意力 KV 池的 aligned-store 支持仍在上游开发中，因此启用后会导致引擎崩溃。

组件	**`agg-base`**
hermes-router (EPP)	✅ 已启动
路由策略	`random`（负载均衡）

组件	`agg-base`
cache-indexer	—
mooncake KV store	—（`qwen3_5` 不支持）

部署

自包含的 InferNex manifest（engine + hermes-router `LLMInferenceServiceConfig`，以及 `LLMInferenceService`，2 replicas × TP=4）：

规格	文件
agg-base（负载均衡）	`qwen3-6-27b-w8a8-agg-base-llmisvc.yaml` ↗

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/plan/validated_models/assets/qwen3-6-27b-w8a8
# edit namespace / model.uri registry / image tag first, then:
kubectl apply -f $base/qwen3-6-27b-w8a8-agg-base-llmisvc.yaml
```

WARNING

在开始接收流量之前，请务必先预热 replicas（施加少量并发）。冷 replica 只会捕获 `batch=1` 的 decode graph；随后第一次并发突发会在热路径上捕获更大的 graph，并进入较慢的稳定状态。`--max-num-seqs 32` 上限是必需的保护措施——如果没有它，高并发可能会级联恶化到 256 并发 / >1 s ITL 的状态。请将 `--gpu-memory-utilization` 保持在 **0.85**，不要使用 0.90；在启用 MTP 且高并发时，0.90 会导致 OOM。

基准测试结果

闭环 `aiperf 0.7.0`，**TP=4 × 2 replicas**（**8 × 910B4**），并发 4，agg-base，开启 MTP。两个场景——① 8k 系统提示词复用和 ② 17.5k 多轮对话——每个场景 240 个请求，输出固定为 128 tokens。每个场景运行 **3** 次（n=3，相同 trace + seed）；全部 6 次运行均返回 240/240，

且无错误。数值为跨运行均值。TTFT / E2E 的单位为 ms，ITL 的单位为 ms（按 chunk 统计；MTP 每个 chunk 约输出 ~3 tokens），TPS = total tokens/s。

场景 ① — 固定长度系统提示词复用（**ISL ~8k / OSL 128**）

部署规格	**TTFT** 平均值 **(ms)**	**ITL** 平均值 **(ms)**	**E2E** 平均值 **(ms)**	**TPS**（**in+out**）
agg-base	1509	32.0	5577	5807

场景 ② — 多轮对话（**ISL ~17.5k / OSL 128**）

部署规格	**TTFT** 平均值 **(ms)**	**ITL** 平均值 **(ms)**	**E2E** 平均值 **(ms)**	**TPS**（**in+out**）
agg-base	3999	45.3	9751	7408

NOTE

如何解读这些结果。 所有 6 次运行都在每个实例稳定保持 2 个 in-flight 请求的情况下完成了 240/240，且无错误；ITL 的可重复性很高（不同运行之间的波动约为 ~3–14%）。`agg-base` 是此模型唯一可用的部署规格——跨实例 KV store + 支持 KV-cache 感知路由的方式（`agg-mc-kv`）不能用于 `qwen3_5` GDN 混合架构。ITL 按每个 **chunk**报告——由于 MTP speculative head 每个流式 chunk 约输出 ~3 tokens，按 token 计算的有效延迟大约是其三分之一。仅解码输出速率为 **91.3 tok/s**（场景 ①）/ **52.5 tok/s**（场景 ②）；TPS 列表示的是总 token（输入 + 输出）口径。ITL p90 为 45.0 ms（①）/ 78.4 ms（②），TTFT p90 为 2192 ms（①）/ 6290 ms（②）。在 17.5k 负载下，随着逐轮上下文被重新 prefill，TTFT 上升且尾部变宽（该模型不能使用 prefix caching）。这些是半规模数据（8 卡）；总吞吐量随实例数量扩展。

DeepSeek-V4-Flash (W4A8)

`deepseek_v4` 架构模型——一个 256-expert MoE，结合 MLA + DSA 稀疏注意力 和原生 MTP speculative head，以 **W4A8** 形式提供（~151 GB 权重，43 层）。 已在 **Ascend 910B4**（**32 GB/卡**） 上通过 Alauda AI 的 InferNex 接口 使用 vLLM-Ascend **nightly** 引擎完成验证。由于权重需要占用全部 8 张卡， 拓扑为 **1** 个实例 **× TP=8**（**+ expert parallel**），两个 benchmark 场景还分别通过 内部 KServe ingress 和 **MaaS gateway**（**API-key**） ingress 进行了驱动。

目录

模型标识
已验证硬件 × 栈
模型配置
部署规格
部署
Benchmark 结果

模型标识

字段	值
发布方	DeepSeek

字段	值
架构	`deepseek_v4` — 256-expert MoE（6 experts/令牌 + 1 shared）+ MLA + DSA 稀疏注意力 + 原生 MTP
量化	W4A8（~151 GB 权重，43 层）；MIT 许可证
模型来源（W4A8）	https://www.modelscope.cn/models/gdydems/DeepSeek-V4-Flash-w4a8-mtp ↗

已验证硬件 × 栈

平台	引擎	版本 / 配置	状态
Ascend 910B4 32 GB × 8（1 个实例 × TP=8 + EP）	vLLM-Ascend	`nightly-releases-v0.22.1rc-openeuler`（vLLM 0.22.1，CANN 9.0.0）	✅ 闭环，2 场景性能测试（n=3），agg-base，双 ingress（KServe + MaaS）

NOTE

发布固定的 nightly `nightly-releases-v0.22.1rc-openeuler`（与 Qwen3.6-27B 相同的镜像）包含完整的 `deepseek_v4` 栈。`enable_dsa_cp` 被有意设置为 **off**（上游 DSA-CP index-buffer bug 会导致该构建上的持续 8k MTP decode 崩溃）。Prefix caching 已启用， 但 命中率为 **0**——DSA 稀疏注意力会按每个 query 选择 KV，因此共享前缀不能直接复用； 它几乎没有额外成本，而相较旧版本的加速来自 nightly 栈本身，而不是 prefix caching。

模型配置

参数	值
Tensor parallelism (`tensor-parallel-size`)	8
副本（实例）	1（= 8 张卡）

参数	值
Expert parallelism (`enable-expert-parallel`)	on
`max-model-len`	32768
`max-num-batched-tokens`	2048
`max-num-seqs`	4
`gpu-memory-utilization`	0.90（32 GB 卡 KV/activation 平衡）
`block-size`	128
`max_tokens`（输出，benchmark 固定）	128
Quantization	`ascend`（W4A8）
Speculative decoding（MTP）	`mtp`，1 token
Prefix caching	已启用，但命中率为 0（DSA 稀疏注意力）

部署规格

仅以 **`agg-base`** 方式提供——aggregation，hermes-router 策略为 `random`。在单个 TP=8 实例下， router 只有一个 endpoint（无实际作用）；保留该结构仅为一致性。跨实例 KV store / KV-cache-aware routing （`agg-mc-kv`）不适用：~151 GB 权重会占满全部 8 张卡，因此只能放下一个副本，也没有第二个实例可供共享 KV。

组件	**`agg-base`**
hermes-router (EPP)	✅ 已启动（单 endpoint，无实际作用）
路由策略	`random`
cache-indexer / mooncake KV store	—（不适用于单实例）

部署

自包含的 InferNex manifest（engine 内联在 `LLMInferenceService` + hermes-router preset 中，1 个副本 × TP=8）：

规格	文件
agg-base, TP=8	`deepseek-v4-flash-w4a8-agg-base-llmisvc.yaml` ↗

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/plan/validated_models/assets/deepseek-v4-flash-w4a8
# edit namespace / model.uri registry / image tag first, then:
kubectl apply -f $base/deepseek-v4-flash-w4a8-agg-base-llmisvc.yaml

# Internal KServe ingress (no auth):
curl -s http://<gateway>/<namespace>/deepseek-v4-flash-w4a8-agg-base/v1/chat/completions \
  -H 'Content-Type: application/json' \
  -d '{"model":"gdydems/DeepSeek-V4-Flash-w4a8-mtp","messages":[{"role":"user","content":"hello"}]}'

# Product MaaS gateway (OpenAI-compatible, API-key auth + token rate limiting):
curl -s http://<maas-gateway>/v1/chat/completions \
  -H "Authorization: Bearer $MAAS_API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"model":"gdydems/DeepSeek-V4-Flash-w4a8-mtp","messages":[{"role":"user","content":"hello"}]}'
```

Benchmark 结果

闭环 `aiperf 0.7.0`，**TP=8 × 1** 个副本（**8 × 910B4**），并发 4，agg-base。 两个场景——① 8k system-prompt 复用和 ② 17.5k multi-turn——各发送 240 个请求， 输出固定为 128 个令牌，n=3（所有运行均为 240/240，零错误）。每个场景都通过两个 **ingress** 进行驱动： 内部 **KServe** Service（无认证）和产品 **MaaS gateway**（Envoy + API-key 认证 + 令牌速率限制——面向客户的 OpenAI endpoint）。 TTFT / E2E 以 ms 计，ITL 以 ms 计，TPS = 总令牌/s。

场景 ①——固定长度 **system-prompt** 复用（**ISL ~8k / OSL 128**）

入口	**TTFT** 平均值 **(ms)**	**ITL** 平均值 **(ms)**	**E2E** 平均值 **(ms)**	**TPS**（**in+out**）
KServe（内部）	3758	90.9	15300	2124
MaaS gateway（API key）	3872	91.4	15481	2099

场景 ②——多轮对话（**ISL ~17.5k / OSL 128**）

入口	**TTFT** 平均值 **(ms)**	**ITL** 平均值 **(ms)**	**E2E** 平均值 **(ms)**	**TPS**（**in+out**）
KServe（内部）	9789	182.1	32918	2189
MaaS gateway（API key）	9003	175.9	31338	2306

NOTE

如何解读这些结果。 所有运行都在每个实例稳定保持 2 个 in-flight 请求的情况下完成了 240/240，且零错误（n=3 平均值）。**MaaS gateway** 与内部 **KServe** 的差异都在运行间抖动范围内 （场景 ① 在 gateway 上约高出 +1–3%——多出几百毫秒的 hop；场景 ② 的两组 n=3 数据在不同时间采集，结果又反向偏了几个百分点）。结论是：**API-key +** 速率限制的 **MaaS gateway** 在典型请求和长上下文请求上都不会带来 超出噪声范围的可测额外开销，因此可以作为生产入口提供服务。仅解码输出速率为 33 tok/s（场景 ①）/ 16 tok/s（场景 ②）； TPS 列表示总令牌（输入 + 输出）口径。MTP speculative decoding 已开启。由于这是单个 TP=8 实例，因此 TPS 不能与本指南中的 2-副本模型直接比较。

WARNING

通过 MaaS gateway 发送长上下文请求时，需要提高网关的 request-body buffer——17.5k 场景（~85 KB streaming body） 超过了默认限制，会一直挂起，直到提高 MaaS gateway 上的

`ClientTrafficPolicy.connection.bufferLimit`。 内部 KServe ingress 没有此类限制。上面的数值是在完成该修复后得到的。

DeepSeek-V4-Flash (W8A8)

`deepseek_v4` 架构模型 — 256 专家 MoE，采用 MLA + DSA 稀疏注意力 以及原生 MTP speculative head，作为 **W8A8** 提供服务（int8 attention + expert， ~280 GB 权重，70 个 shard，43 层）。由于 ~280 GB 的权重无法装入单个 8 × 32 GB 节点，因此它在 **Ascend 910B3**（**64 GB/**卡）跨两个 节点 **= 16** 张卡 上进行了验证，作为一个单一聚合的 **DP2 × TP=8 + EP16** 服务， 通过 Alauda AI 的 InferNex 界面，使用 vLLM-Ascend **nightly** 引擎提供服务。它增加了一个 **mooncake** 跨 **rank KV** 存储，使两个数据并行 rank 共享前缀 KV；两种基准场景都通过 **MaaS gateway**（**API-key**）入口 以并发扫测方式驱动（8 / 16 / 32）。

目录

模型标识
已验证硬件 × 软件栈
模型配置
部署规格
部署
基准测试结果

模型标识

字段	值
发布方	DeepSeek
架构	`deepseek_v4` — 256-expert MoE（每个 token 6 个 expert + 1 个 shared）+ MLA + DSA 稀疏注意力 + 原生 MTP
量化	W8A8（`W8A8_DYNAMIC`，int8 attention + expert；~280 GB 权重，70 个 shard，43 层）；MIT license
模型来源（W8A8）	https://www.modelscope.cn/models/Eco-Tech/DeepSeek-V4-Flash-w8a8-mtp↗
ModelCar（OCI Image Layout tar）	`http://<model-package-host>/packages/aml-models-oci/v0.1.0/DeepSeek-V4-Flash-w8a8-mtp/DeepSeek-V4-Flash-w8a8-mtp.oci.tar`

NOTE

W8A8 ModelCar 以 **OCI Image Layout tar** 的形式分发在内部 package store 上，而不是公共 Docker Hub 镜像。请将上面链接中的 `<model-package-host>` 替换为 Alauda 为该环境提供的 package store 地址。集群拉取的是 OCI *image*，不是 tar，因此请先将其导入你自己的 **registry** 一次，然后将 `model.uri` 指向该地址—— 见下方 部署 部分。基准测试本身使用的是按节点预置在 **node-local PVC** 上的权重；ModelCar 只是同一份权重的重新封装（manifest 头部记录了该 PVC 备选方案）。

已验证硬件 × 软件栈

平台	引擎	版本 / 配置	状态
Ascend 910B3 64 GB × 16，2 节点（1 个实例，DP2 × TP=8 + EP16）	vLLM-Ascend	`nightly-main-openeuler`（vLLM 0.23.0）	✅ 闭环，2 场景 并发扫测（**8/16/32**），每档 480 次请求，0 error / 0 mismatch，聚合 + mooncake KV store，通过 MaaS gateway

NOTE

nightly-main 镜像包含完整的 `deepseek_v4` 栈 以及 DP2 跨节点 graph + DSA-CP prefix-cache 修复。`enable_dsa_cp` 被刻意设置为 **on** —— 它是 非零 prefix-cache 命中的前提，并且该构建在开启它时保持稳定。在满足完整命中门槛 （见配置表）的情况下，**prefix caching** 可正常工作 —— 在 21k prefix 上单 rank 命中率约 ~90.9 %。

模型配置

参数	值
Tensor parallelism (`tensor-parallel-size`)	8
Data parallelism (`data-parallel-size`)	2（每个节点 1 个 rank，跨节点）
Replicas (instances)	1 个聚合服务（leader + worker = 16 张卡）
Expert parallelism (`enable-expert-parallel`)	on（EP16）
`max-model-len`	200000（agent 长上下文）
`max-num-batched-tokens`	8192
`max-num-seqs`	32
`gpu-memory-utilization`	0.92（64 GB 卡 — ~20.6 GB/card 的权重为 KV 留出充足空间）
`block-size`	128
Quantization	`ascend`（W8A8_DYNAMIC）

参数	值
Speculative decoding (MTP)	`mtp`，1 token（draft head 上启用 `enforce_eager`）
Decode graph	`FULL_DECODE_ONLY`（`enable_npugraph_ex`）
DSA context parallel (`enable_dsa_cp`)	on（prefix-cache 命中前提）
Prefix caching	已启用且有效（单 rank 命中率约 ~90.9 %；需要 `VLLM_ASCEND_ENABLE_FLASHCOMM1=1` + `VLLM_PREFIX_CACHE_RETENTION_INTERVAL=16384` + prefix ≥ 16384）
Tool calling / reasoning	`--tool-call-parser deepseek_v4` + `--enable-auto-tool-choice` + `--reasoning-parser deepseek_v4`（默认关闭 thinking）

部署规格

作为 **`agg-mc-kv`** 提供服务——一个单一的 DP2 × TP=8 聚合 加上 由两个 DP rank 共享的 mooncake KV store。DP2 的随机路由会将每个用户不断增长的历史分散到两个 rank 上，因此每个 rank 的本地多轮命中率只有约 ~41 %；mooncake store 允许一个 rank 拉取另一个 rank 已经计算出的 prefix KV，将合并后的有效 **prefix** 复用率提升到约 **~84 %**。路由器保持为 `random`，cache-indexer 关闭：单个聚合 endpoint 不需要全局 KV 索引或感知 KV-cache 的路由—— 这与 GLM-5.2 / Qwen3-8B 的聚合范式一致（store 共享 KV；router 保持 random）。

组件	**`agg-mc-kv`**
hermes-router (EPP)	✅ 已启动（单一 leader endpoint，`random` 无实际作用）
路由策略	`random`
mooncake KV store (AscendStoreConnector)	✅（在两个 DP rank 之间共享）

组件	`agg-mc-kv`
cache-indexer / KV-cache-aware routing	—（单个聚合 endpoint 不需要）
跨 rank prefix KV 复用	yes（合并后有效约 ~84 %）

> **NOTE**
>
> store 以异步方式写入（21k prefix 落盘约需 ~30–60 s）。跨 rank 命中会出现在多轮 agent 会话中，turn 间隔大约一分钟或更长时；连续的亚秒级复用只会命中同一 rank 的本地 prefix cache。 我们还评估过一种单独的 KV-cache-aware routing 方案（将后续请求引导回持有 cache 的 rank），并已回退——在该规模下，这个栈没有带来收益，因此生产形态保持为 `random` + mooncake store。

部署

自包含的 InferNex manifest（engine 内联在 `LLMInferenceService` 的 leader + worker templates + hermes-router preset 中，跨两个节点的 DP2 × TP=8，并使用 mooncake store）：

规格	文件
agg-mc-kv, DP2 × TP=8	`deepseek-v4-flash-w8a8-agg-mc-kv-llmisvc.yaml` ↗

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/plan/validated_models/assets/deepseek-v4-flash-w8a8

# 1. Import the ModelCar (OCI Image Layout tar) into your own registry once — the cluster
#    pulls an OCI image, not the tar. skopeo does the multi-arch (amd64+arm64) push in one go:
curl -sO http://<model-package-host>/packages/aml-models-oci/v0.1.0/DeepSeek-V4-Flash-w8a8-mtp/DeepSeek-V4-Flash-w8a8-mtp.oci.tar
skopeo copy --all \
  oci-archive:DeepSeek-V4-Flash-w8a8-mtp.oci.tar:v0.1.0 \
  docker://<your-registry>/modelcar-deepseek-v4-flash-w8a8-mtp:v0.1.0

# 2. Edit the manifest: set the namespace, set model.uri to the registry ref you just
#    pushed, and the image tag (match your CANN / host driver). Then apply — the
#    storageInitializer pulls the ModelCar to /mnt/models in each DP rank.
#    (Alternative: stage the weights on a node-local PVC per node — see the manifest header.)
kubectl apply -f $base/deepseek-v4-flash-w8a8-agg-mc-kv-llmisvc.yaml

# Internal KServe ingress (no auth):
curl -s http://<gateway>/<namespace>/deepseek-v4-flash-w8a8-agg-mc-kv/v1/chat/completions \
  -H 'Content-Type: application/json' \
  -d '{"model":"deepseek-v4-flash-w8a8","messages":[{"role":"user","content":"hello"}]}'

# Product MaaS gateway (OpenAI-compatible, API-key auth + token rate limiting):
curl -s http://<maas-gateway>/v1/chat/completions \
  -H "Authorization: Bearer $MAAS_API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"model":"deepseek-v4-flash-w8a8","messages":[{"role":"user","content":"hello"}]}'
```

基准测试结果

闭环 `aiperf 0.7.0`，**DP2 × TP=8**（**16 × 910B3 64 GB**），通过产品 **MaaS gateway** 驱动。并发扫测 **8 / 16 / 32**，每档 480 次请求（所有档位均为 0 error / 0 mismatch）。这些核心数字对应的是生产级 `agg-mc-kv` 配置——FULL_DECODE_ONLY graph + mooncake 跨 rank KV store，也就是上方 manifest 部署的 精确配置——场景 ②（多轮嵌套长上下文），也就是该 store 所针对的 agent 风格工作负载。TTFT / E2E 的单位为 s，ITL 的单位为 ms；decode = 仅输出 tok/s（实测），TPS = 总 token/s（输入 + 输出，计算得到；以 prefill 为主）。

场景 ② — 多轮对话，生产级 `agg-mc-kv`（**ISL ~17.5k / OSL 128**）

并发	**TTFT** 平均值 **(s)**	**ITL** 平均值 **(ms)**	**E2E** 平均值 **(s)**	**Decode (tok/s)**
8	1.76	72	10.9	93.6
16	1.36	90	12.8	159.3
32	1.94	123	17.6	230.9

mooncake 跨 rank KV store 正是让这一方案变快的关键：它将多轮 prefix 复用率从约 ~41 %（每个 rank 本地）提升到约 ~84 %（合并后有效）。在并发 32 下，将同一 场景 ② 按部署演进的各个配置依次跑一遍，可以看出收益来自哪里：

配置演进 — 场景 ②，并发 **32**

配置	**TTFT** 平均值 **(s)**	**E2E** 平均值 **(s)**	**Decode (tok/s)**	**TPS (in+ou**
agg（基础版 — eager，无 store）	10.2	83.7	48.1	678
+ FULL_DECODE_ONLY graph	3.2	29.3	138.7	1955
+ mooncake KV store（生产版）	**1.9**	**17.6**	**230.9**	**3255**

→ graph + store 共同作用，相比相同并发下的基础聚合，**TTFT** 约提升 **5.3× / E2E** 约提升 **4.8× / TPS** 约提升 **4.8×**。

场景 ① **—** 固定长度 **system-prompt** 复用（**ISL ~8k / OSL 128**），基础 **agg**

8k 固定 prompt 场景仅在基础聚合配置上进行了测量（graph

- mooncake 没有为它单独复跑），因此请将这些结果视为下限——生产配置会像场景 ② 那样显著提升它们：

并发	**TTFT** 平均值 **(s)**	**ITL** 平均值 **(ms)**	**E2E** 平均值 **(s)**	**Decode (tok/s)**
8	3.19	305	41.9	24.2
16	3.42	350	47.8	42.3
32	5.71	493	68.3	59.2

NOTE

如何解读这些结果。 每个档位都以 480/480 完成，零错误、零输出不匹配。随着并发上升，系统吞吐量（**decode / TPS**）会增加 （次线性，逐渐饱和），而尾延迟（TTFT / ITL / E2E）也会上升——这是更多 in-flight 请求带来的排队成本。TPS 是总 **token**（输入 **+** 输出） 层面的指标，并且以 prefill 为主（ISL 8k / 17.5k，OSL 128）；仅解码输出速率则是单独的“Decode”列。

这些数字不能直接与本指南中的其他模型进行比较：它是并发扫测（而不是 Qwen 模型使用的固定并发 4 运行），运行在 **910B3 64 GB** 卡上（而不是 910B4 32 GB），并且采用 **16** 卡 **DP2 × TP=8** 拓扑。请将它们视为这个特定大型 MoE 部署的运行包络，而不是跨模型排名。

MiniMax-M2.5 (W8A8)

`minimax_m2`-架构模型（`MiniMaxM2ForCausalLM`）— 一个 256 专家 MoE（每个 token 8 个专家，无共享专家），具有 **62** 层全注意力（GQA 48/8 heads，head_dim 128 — 一个纯全注意力模型，不是 hybrid-KV / sliding-window 模型）以及原生 MTP head，以 **W8A8**（`W8A8_DYNAMIC`，QuaRot int8 attention + expert，无 KV quant；~230 GB 权重，71 个 shard，62 层）形式提供服务。它是 总计 **230B**，且每个 **token** 激活约 **~10B**。约 ~230 GB 的权重可以放入单个 8 × 64 GB 节点中，但它已在 **Ascend 910B3**（**64 GB/**卡）跨两个节点 **= 16** 张卡 的条件下验证为一个聚合的 **DP2 × TP=8 + EP16** 服务——与 DeepSeek-V4-Flash 的 16 卡拓扑进行同类对比，其中 DP2 在偏 prefill 的基准上大约将 prefill 吞吐量翻倍。它通过 Alauda AI 的 InferNex surface 运行 vLLM-Ascend **v0.23.0** release engine，并且两个基准场景都通过 **MaaS gateway**（**API-key**）入口 作为并发扫描（8 / 16 / 32）来驱动。

不同于 DeepSeek-V4-Flash（W8A8），MiniMax-M2.5 以不带任何加速附加组件的纯聚合方式提供服务——没有 mooncake KV store，也没有 speculative decoding。这是有意采用的生产配置：因为这个模型“很快”（~10B activated + full attention + W8A8），这两个附加组件都被测出会*拖慢*性能（见 部署规范）。

目录

模型标识
已验证硬件 × 技术栈
模型配置
部署规范
部署
基准结果
真实 coding-agent 验证

模型标识

字段	值
发布方	MiniMax
架构	`minimax_m2`（`MiniMaxM2ForCausalLM`）— 256 专家 MoE（8 experts/token，无共享专家）+ 62 层 全注意力（GQA 48/8，head_dim 128；非 hybrid-KV）+ 原生 MTP head；总计 **230B /** 激活约 **~10B**
量化	W8A8（`W8A8_DYNAMIC`，**QuaRot**；int8 attention + expert，无 KV quant；~230 GB 权重，71 个 shard，62 层）；MIT license
Context / vocab	196,608（~192k，`max_position_embeddings`，rope_theta 5e6）/ 200,064
模型来源（W8A8）	https://www.modelscope.cn/models/Eco-Tech/MiniMax-M2.5-w8a8-QuaRot ↗
ModelCar（OCI Image Layout tar）	`http://<model-package-host>/packages/aml-models-oci/v0.1.0/MiniMax-M2.5-w8a8-QuaRot/MiniMax-M2.5-w8a8-QuaRot.oci.tar`

NOTE

内置 **MTP head** 在这些权重上不可用。 配置声明了 `use_mtp: true` / `num_mtp_modules: 3`，但 QuaRot 量化的 checkpoint 不包含 **MTP tensors**（143,967 个 tensor 的 state dict 在 layer 61 处结束），因此 `--speculative-config method=mtp` 无法启动。MiniMax 唯一可用的 speculative 路径是 **eagle3**（一个独立的 draft model），而 eagle3 已作为该模型生产配置之外的方案进行测量——见 部署规范。

NOTE

W8A8 ModelCar 以 **OCI Image Layout tar** 的形式分发在内部包存储上，而不是公共 Docker Hub image。请将上方链接中的 `<model-package-host>` 替换为 Alauda 为该环境提供的包存储地址。基准测试本身使用的是已在每个节点本地 **PV/PVC** 上落盘的权重；下方 manifest 挂载的也是这条路

径。ModelCar 只是将同一套权重重新打包，便于跨环境分发——先导入到你自己的 registry，然后重新指向 `model.uri`（见 **Deploy** 章节）。由于这个 ModelCar 超过 **200 GB**，`oci://` `storageInitializer` 拉取需要额外处理临时磁盘。

已验证硬件 × 技术栈

平台	引擎	版本 / 配置	状态
Ascend 910B3 64 GB × 16，2 个节点（1 个实例，DP2 × TP=8 + EP16）	vLLM-Ascend	`v0.23.0-openeuler`（vLLM 0.23.0，release 固定）	✅ 闭环、2 场景 并发扫描（**8/16/32**）、每档 480 次请求、0 error；纯聚合基线（无 spec decode、无 mooncake）；+ 真实 coding-agent 验证（OpenCode / Pi），全部通过 MaaS gateway

NOTE

`v0.23.0-openeuler` release engine 包含完整的 `minimax_m2` 技术栈（model + `minimax_m2` tool-call / reasoning parsers）以及 vLLM #11505，该修复会将流式 tool-call 参数一直保留到结束的 `</parameter>` 之前，因此不会把尾部标签碎片拼接进命令参数中。之所以选择它而不是更早的 `nightly-main-0709` 快照，是为了这个 parser 修复（已在 3,123 次 tool 执行中验证，尾部污染为 0）；代价是相较 0709 的 decode 有小幅回退——见 基准结果。

模型配置

参数	值
Tensor parallelism (`tensor-parallel-size`)	8

参数	值
Data parallelism (`data-parallel-size`)	2（每个节点 1 个 rank，跨节点）
Replicas (instances)	1 个聚合服务（leader + worker = 16 张卡）
Expert parallelism (`enable-expert-parallel`)	on（EP16）
`max-model-len`	196608（agent 长上下文；基准场景 ≤ ~17.5k）
`max-num-batched-tokens`	16384
`max-num-seqs`	32
`gpu-memory-utilization`	0.85（官方值——长上下文，刻意未拉满）
Quantization	`ascend`（W8A8_DYNAMIC / QuaRot）
Speculative decoding	无（QuaRot 权重中缺少内置 MTP；eagle3 已测出不划算——见下文）
Decode graph	`FULL_DECODE_ONLY`（ITL ~34 ms，相比 eager 的 ~300 ms）
Prefix caching	已启用，本地（在真实 agent trace 上命中率约 ~62.6 %；agent 窗口内 Prompt-Token 命中率为 91.8–95.5 %）
Tool calling / reasoning	`--tool-call-parser minimax_m2` + `--enable-auto-tool-choice` + `--reasoning-parser minimax_m2`
典型环境	`VLLM_ASCEND_ENABLE_FLASHCOMM1=1`。不要设置 `enable_fused_mc2`——它是“A3 推荐，不适用于 A2”，而 910B3 属于 A2 级别。

部署规范

以 `agg` 方式提供服务——一个单独的 DP2 × TP=8 聚合，不使用 mooncake KV store，不使用 speculative decoding。router 保持 `random`，cache-indexer 为关闭：单个聚合 endpoint 不需要全局 KV index 或感知 KV-cache 的 routing（与 GLM-5.2 / Qwen3-32B 的聚合范式一致）。真正带来净收益、并且该配置保留的是 `FULL_DECODE_ONLY` decode graph 加上引擎的本地 **prefix cache**。

组件	`agg`
hermes-router (EPP)	✅ 已启动（单 leader endpoint，`random` 无操作）
Routing strategy	`random`
mooncake KV store	—（已测出无收益；见下文）
Speculative decoding (eagle3 / MTP)	—（已测出无收益；见下文）
cache-indexer / KV-cache-aware routing	—（单个聚合 endpoint 不需要）
Decode graph + local prefix cache	✅（该配置保留的净收益加速项）

NOTE

为什么不使用 **speculative decoding** 和 **KV store** —— 因为在这个模型上两者都被测出会拖慢。 由于 MiniMax-M2.5 “很快”（只有 ~10B activated + full attention + W8A8），这两个附加组件都会造成负收益：

- **eagle3**（**speculative decoding**）——被排除，有两条独立证据。 ① 批处理吞吐量是净亏损：场景 ① 在并发 32 时，decode 为 **377 tok/s**（**eagle3**）对比 **631 tok/s**（基线）（越多 speculative tokens，越差）。② 一个真实的 **27k-prefill agent** 请求返回 **HTTP 500**（engine 保持运行，但每个这类请求都会失败），而基线可以 0 error 地服务同一请求。机制上：speculation 用计算换延迟——只有在 decode 受内存带宽限制（单流）时才是净收益；一旦 batch 使计算饱和，就会变成净亏损；同时 W8A8 基座与全精度 eagle draft 也不匹配，导致 pos-0 仅约 ~44 % 的接受率（每个 draft 仅接受 0.56 个）。内置 MTP head 不可用（量化权重中没有 MTP tensors）。
- **mooncake KV store**——不值得。 即使 store 完全 warm（约 ~91 % merged prefix hit），仍然是净亏损：场景 ② 在并发 16 / 32 时吞吐量分别 下降 **14 % / 13 %**，TTFT 在各方面都更差。MiniMax 的 prefill 快到这种程度：通过网络重新拉取 KV 的成本都大于在本地重新计算——这与

DeepSeek 相反，后者较慢的 prefill 使 store 成为明确收益项。在真实 agent trace 上，本地 prefix cache 已经覆盖 62.6 %，store 只是在其基础上再增加约 ~3.6 %。

部署

自包含的 InferNex manifest（engine 内联在 `LLMInferenceService` 的 leader + worker templates + hermes-router preset 中，跨两个节点的 DP2 × TP=8，纯 `agg` 基线）：

规格	文件
agg, DP2 × TP=8	`minimax-m2.5-w8a8-agg-llmisvc.yaml` ↗

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/plan/validated_models/assets/minimax-m2.5-w8a8

# 1. Stage the ~230 GB weights on a node-local directory on BOTH nodes (byte-identical),
#    then edit the manifest: set the namespace, pin the two hostPath PVs to your node
#    hostnames (<node-0> / <node-1>) and weights path, and set the image tag (match your
#    CANN / host driver). Then apply — the engine mounts the on-node weights at /mnt/models
#    (the storageInitializer is off, so there is no 230 GB pull).
kubectl apply -f $base/minimax-m2.5-w8a8-agg-llmisvc.yaml

#    Cross-environment alternative — import the OCI ModelCar tar into your own registry
#    once (skopeo does the multi-arch amd64+arm64 push in one go), then set the manifest
#    model.uri to oci://<your-registry>/modelcar-minimax-m2.5-w8a8-quarot:v0.1.0:
#      curl -sO http://<model-package-host>/packages/aml-models-oci/v0.1.0/MiniMax-M2.5-w8a8-QuaRot/MiniMax-M2.5-w8a8-QuaRot.oci.tar
#      skopeo copy --all \
#        oci-archive:MiniMax-M2.5-w8a8-QuaRot.oci.tar:v0.1.0 \
#        docker://<your-registry>/modelcar-minimax-m2.5-w8a8-quarot:v0.1.0
#    (This ModelCar is >200 GB — keep storageInitializer.enabled:false and add an init
#    container that extracts the pulled layout to tmpfs.)

# Internal KServe ingress (no auth):
curl -s http://<gateway>/<namespace>/minimax-m25-w8a8-agg/v1/chat/completions \
  -H 'Content-Type: application/json' \
  -d '{"model":"minimax-m2.5-w8a8","messages":[{"role":"user","content":"hello"}]}'

# Product MaaS gateway (OpenAI-compatible, API-key auth + token rate limiting):
curl -s http://<maas-gateway>/v1/chat/completions \
  -H "Authorization: Bearer $MAAS_API_KEY" \
  -H 'Content-Type: application/json' \
  -d '{"model":"minimax-m2.5-w8a8","messages":[{"role":"user","conten
```

```
t":"hello"}]}'
```

基准结果

闭环 `aiperf 0.7.0`，**DP2 × TP=8**（**16 × 910B3 64 GB**），通过产品 **MaaS gateway** 驱动，在生产 `v0.23.0-openeuler` engine 上运行。并发扫描为 **8 / 16 / 32**，每档 480 个请求（所有档位 0 error / 0 mismatch）。TTFT / E2E 以秒计，ITL 以毫秒计；decode = 仅输出 tok/s（实测），TPS = 总 tokens/s（输入 + 输出，计算得出；prefill 主导）。输出长度固定为 128 tokens。

场景 ① — 固定长度 **system-prompt** 复用（**ISL ~8k / OSL 128**）

并发	**TTFT** 平均值（**s**）	**ITL** 平均值（**ms**）	**E2E** 平均值（**s**）	**Decode**（**tok/s**）
8	1.17	33.5	5.4	189
16	1.35	36.2	5.9	344
32	1.44	45.6	7.2	564

场景 ② — 多轮对话，长上下文（**ISL ~17.5k / OSL 128**）

并发	**TTFT** 平均值（**s**）	**ITL** 平均值（**ms**）	**E2E** 平均值（**s**）	**Decode**（**tok/s**）
8	2.95	44.0	8.5	120
16	3.81	56.7	11.0	186
32	5.42	87.9	16.6	247

NOTE

引擎选择—— `v0.23.0` 与更早的 `nightly-main-0709`。 在相同硬件 / 配置 / caliber 下重新测试后，`v0.23.0` 相比 0709 表现出小幅 **decode** 回退（decode 吞吐量 **-3…11 %**，ITL **+5…18 %**，且随并发增大而加剧；例如场景 ① 并发 32 时 decode 由 631 → 564 tok/s），而 **TTFT**（**prefill**）持平或更好。之所以接受该回退，是为了换取一个**release** 固定（不漂移）的 tag、统一的 `minimax_m2`

parser，以及 #11505 的 streaming tool-call 修复。0709 image 仍保留在节点上，作为对 decode 吞吐敏感的交付场景的回退方案。

配置决策——**eagle3** 和 **mooncake** 均不划算（基于基线测量）

配置（场景 · 并发）	基线	**eagle3**	**mooncake store**
S① conc 32 — decode (tok/s)	**631**	377（1-tok）/ 300（3-tok）	—
S① conc 32 — ITL (ms)	**39.5**	75.7 / 97.7	—
S② conc 16 — TPS (in+out)	**27,792**	—	23,902（**−14 %**）
S② conc 32 — TPS (in+out)	**37,458**	—	32,421（**−13 %**）
真实 27k-prefill agent 请求	0 error	**HTTP 500**	—

→ 纯 `agg` 基线在所有维度上都获胜；这两个加速层在这个快模型上都适得其反。（eagle3 / mooncake 的 A/B 测试是在更早的 0709 基线上做的，早于 `v0.23.0` 的重新测试；但结论在两个 engine 上都成立。）

NOTE

如何解读这些数据。 每个档位都完成了 480/480，0 error，0 输出不匹配。随着并发提升，系统吞吐量（**decode / TPS**）上升（次线性、逐渐饱和），同时尾延迟（TTFT / ITL / E2E）也会上升——这是更多 in-flight requests 带来的排队成本。TPS 是**总 token（输入 + 输出）**口径，并且是 prefill 主导（ISL 8k / 17.5k，OSL 128）；仅 decode 的输出速率则是单独的“Decode”列。

这些数字不能直接与本指南中的其他模型对比：这是一轮并发扫描（而不是 Qwen 模型使用的固定并发 4 运行），运行在 **910B3 64 GB** 卡上，采用 **16** 卡 **DP2 × TP=8** 拓扑。请将其视为这个特定大型 MoE 部署的运行边界，而不是跨模型排序。

真实 **coding-agent** 验证

除了合成的 aiperf 扫描之外，还通过 MaaS gateway 将 MiniMax-M2.5 作为真实 **coding agent** 进行驱动。**Envoy AI Gateway v0.6.0** 将 vLLM Chat Completions 流通过 OpenAI→OpenAI 原样透传，因此 `minimax_m2` parser 的流式 `reasoning_content` 和结构化 tool call 能完整到达客户端。两个 agent（**OpenCode 1.17.18** 和 **Pi 0.80.6**）在并发 **1 / 8 / 32** 下运行了同一套固定的 14 任务 Terminal-Bench 集合（分别运行 14 / 14 / 42 次），并由客观 verifier 评分。

Agent	通过数（并发 **1 / 8 / 32**）	峰值正确任务吞吐量
OpenCode 1.17.18	6 / 14 · 4 / 14 · 14 / 42	24.65 tasks/hr
Pi 0.80.6	7 / 14 · 8 / 14 · 18 / 42	**34.98 tasks/hr**

- 旧的 `</parameter>` 命令尾部污染已修复： 在 3,123 次真实 tool 执行中，没有任何一次执行参数包含闭合标签碎片（这也是 engine 固定到 `v0.23.0` / #11505 的原因）。
- 在并发 **32** 下链路很快且未饱和： 0 waiting requests，KV-cache 峰值 14.6 %，16 卡 AICore 平均/p95 为 17.2 % / 34 %，vLLM 0 error / abort / preemption——因此 32 是已验证的下界，而不是容量上限。
- 复杂链路上仍存在残余 **parser** 风险： OpenCode 在 **10 / 70** 次试验中把原始 tool XML 误解析为文本/reasoning；Pi 在 **2 / 70** 次试验中合并了相邻 tool 边界。因此结论是可用于 **POC**，但尚未宣称生产稳定，而且在这个模型上 Pi 的表现优于这两个 agent。

设备方案

关于 **Alauda Build of Hami**

关于 **NVIDIA GPU Device Plugin** 的 **Alauda Bui**

关于 Alauda Build of Hami

Heterogeneous AI Computing Virtualization Middleware（HAMi），原名 k8s-vGPU-scheduler，是一个“all-in-one”图表，旨在管理 k8s 集群中的 Heterogeneous AI Computing Devices。它可以提供在任务之间共享 Heterogeneous AI devices 的能力。

> **Note**
>
> 因为 Alauda Build of Hami 的发版周期与灵雀云容器平台不同，所以 Alauda Build of Hami 的文档现在作为独立的文档站点托管在 Alauda Build of Hami ↗。

关于 NVIDIA GPU Device Plugin 的 Alauda Build

NVIDIA 用于 Kubernetes 的 device plugin 是一个 DaemonSet，可让你自动执行以下操作：

- 暴露集群中每个节点上的 GPU 数量
- 跟踪 GPU 的运行状况
- 在你的 Kubernetes 集群中运行支持 GPU 的容器

> **Note**
>
> 因为 Alauda Build of NVIDIA GPU Device Plugin 的发版周期与灵雀云容器平台不同，所以 Alauda Build of NVIDIA GPU Device Plugin 的文档现在作为独立的文档站点托管在 Alauda Build of NVIDIA GPU Device Plugin↗。

安装

目录

Requirements

Hardware

Software

安装

Requirements

Hardware

- 至少需要两个节点，总计具备 16 核和 32 GB 内存。
- 运行时服务所需的额外资源取决于实际业务规模：同时运行 10 个 7B 规模的 LLM 推理实例，至少需要 10 块 GPU 以及相应的 CPU、内存、磁盘存储和对象存储。
- 每个 worker 节点需要 200G 可用磁盘空间。

Software

- CUDA Toolkit 版本：12.6 或更高。

INFO

如果你的 GPU 不支持 CUDA 12.6，仍然可以使用较低版本的 CUDA Toolkit。不过，在部署 Alauda AI 之后，需要添加一个适配旧版 CUDA 的自定义推理 runtime。可参考 扩展 LLM 推理 runtimes，因为内置的 vLLM 推理 runtime 仅支持 CUDA 12.6 或更高版本。

安装

安装 Alauda AI 涉及以下高级任务：

1. 确认并配置你的集群以满足所有要求。请参考 安装前配置。
2. 为 Alauda AI 配置平台入口、认证资源以及可选的 UI 品牌设置。请参考 配置 Alauda AI 入口和品牌。
3. 安装 Alauda AI。请参考 安装 Alauda AI。

完成后，Alauda AI 的核心能力就已成功部署。如果你想快速体验该产品，请参考 快速开始。

安装前配置

准备 **Harbor** 服务

上架软件包

下载软件包

下载 `violet` 工具

安装 **Alauda**

下载软件包

前提条件

安装 **Workbench**

前提条件

启用 Workbench

验证

配置 **Alauda AI** 入口和品牌

前提条件

Tools 菜单配置

添加 Tools 菜单项

结果

安装前配置

目录

准备 **Harbor** 服务

准备 **Harbor** 服务

如果你计划使用 **Model Catalog** 功能，请在安装 Alauda AI 之前准备一个 **Alauda Build of Harbor** 服务。该镜像仓库必须满足以下要求：

- 以生产模式运行。生产环境部署请使用 HTTPS。
- 允许推理集群节点在不使用镜像拉取凭证的情况下拉取 Model Catalog 镜像。

如果无法在目标环境中部署 HTTPS 镜像仓库，可以使用 HTTP 镜像仓库作为替代，但必须在部署模型之前配置推理集群容器运行时。有关详细的 containerd 配置，请参见创建 Alauda AI 实例时设置 **Model OCI Registry Address** 的相关说明。

有关安装操作步骤，请参见 部署 Alauda Build of Harbor↗。

上架软件包

Alauda AI 组件以平台软件包的形式交付。在安装 operator 或集群插件之前，请从 灵雀云 customer portal 下载软件包，并使用 `violet` 打包工具将其上架到目标集群。

本页介绍通用的软件包工作流。有关组件特定的前置条件、安装设置和自定义资源，请在上传软件包后，继续查看该组件的安装页面。

目录

下载软件包

下载 `violet` 工具

上传软件包

上传到外部 registry

上传多个软件包

安装已上传的组件

下载软件包

从 灵雀云 Customer Portal 的 **Marketplace** 下载所需组件和版本的软件包：

- 下载与 Alauda AI release 和目标架构匹配的软件包。
- 对于从 **OperatorHub** 安装的组件，请使用 **OLM operator bundle**。
- 对于从 **Cluster Plugins** 安装的组件，请使用 **Cluster Plugin** 软件包。

- 将下载的软件包保存在你运行 `violet` 的机器上。

确切的软件包文件名和版本会随对应的产品 release 一同提供。不要用其他 Alauda AI release 中的软件包替代。

下载 `violet` 工具

如果用于上架软件包的机器上尚未提供 `violet`，请先下载它。

在 Web Console 中，切换到 **Administrator** 视图：

1. 单击 **Marketplace** > **Upload Packages**。
2. 单击 **Download Packaging and Listing Tool**。
3. 在 **Execution Environment** 下找到正确的操作系统和 CPU 架构。
4. 单击 **Download**。
5. 使下载的文件可执行：

```
chmod +x <path-to-violet>
```

上传软件包

使用以下命令将 operator 或 Cluster Plugin 软件包上传到目标集群：

```
violet push \
  <package> \
  --platform-address=<platform-access-address> \
  --platform-username=<platform-admin> \
  --platform-password=<platform-admin-password> \
  --clusters=<target-cluster>
```

将占位符替换为平台地址、平台管理员凭据、目标集群名称以及本地软件包路径。软件包会上传到平台仓库，并可在所选集群上进行安装。

INFO

对于平台内置的 registry，不需要额外的 registry 参数。`violet` 会将软件包中包含的镜像重新定位到平台 registry 中，这样在安装 operator 或 Cluster Plugin 时，就无需从其原始 registry 拉取这些镜像。

上传到外部 registry

如果目标集群使用外部私有或公共 registry，而不是平台内置的 registry，请提供目标 registry 及其凭据：

```
violet push \
  <package> \
  --platform-address=<platform-access-address> \
  --platform-username=<platform-admin> \
  --platform-password=<platform-admin-password> \
  --clusters=<target-cluster> \
  --dst-repo=<external-registry-address> \
  --username=<registry-username> \
  --password=<registry-password>
```

请使用目标集群可以访问并拉取的 registry 地址。在安装组件之前，请确认软件包所需的所有镜像都已存在于目标 registry 中。

上传多个软件包

你可以使用 shell 脚本上传多个软件包。设置平台变量，并将每个下载的软件包添加到 `PACKAGES` 数组中：

uploading-packages.sh

```
#!/usr/bin/env bash
set -euo pipefail

PLATFORM_ADDRESS=https://<platform-access-address>
PLATFORM_ADMIN_USER=<platform-admin>
PLATFORM_ADMIN_PASSWORD=<platform-admin-password>
CLUSTER=<target-cluster>

PACKAGES=(
  <path-to-package-1>
  <path-to-package-2>
)

VIOLET_EXTRA_ARGS=()
# Uncomment and set these values when the target cluster uses an external
registry.
# VIOLET_EXTRA_ARGS+=(
#   --dst-repo=<external-registry-address>
#   --username=<registry-username>
#   --password=<registry-password>
# )

for PACKAGE in "${PACKAGES[@]}"; do
  violet push \
    "$PACKAGE" \
    --platform-address="${PLATFORM_ADDRESS}" \
    --platform-username="${PLATFORM_ADMIN_USER}" \
    --platform-password="${PLATFORM_ADMIN_PASSWORD}" \
    --clusters="${CLUSTER}" \
    "${VIOLET_EXTRA_ARGS[@]}"
done
```

使用以下命令运行脚本：

```
bash ./uploading-packages.sh
```

WARNING

请保护平台和 registry 密码。不要将包含真实凭据的脚本提交到源代码管理中，也不要将其分享给授权安装团队以外的人员。

安装已上传的组件

上传完成后，在 Web Console 中打开目标集群：

- 对于 OLM operator，请转到 **Administrator** > **Marketplace** > **OperatorHub**，选择已上传的组件并安装。然后根据组件文档，按需创建其自定义资源。
- 对于 Cluster Plugin，请转到 **Administrator** > **Marketplace** > **Cluster Plugins**，并根据平台文档安装已上传的组件。

如果该组件或版本未显示，请验证软件包上传是否成功、所选集群是否正确，以及软件包是否与目标平台架构和 release 匹配。

安装 Alauda AI

Alauda AI 通过 Alauda AI Operator 提供模型管理、推理、训练和 MLOps 能力。系统会自动创建 `default` `AmlCluster` 实例，用于控制集群中启用的组件。默认情况下，Alauda AI 在推理后端使用 **KServe** 的 `Standard` 模式，这种模式特别推荐用于资源密集型的生成式工作负载。该模式通过利用基础 Kubernetes 功能，提供了一种直接的模型部署方式，并具备强大且可定制的部署能力。

可选能力（包括用于将推理服务缩容至零的 **Knative** 功能）可通过更新 `AmlCluster` 实例中的组件开关来启用。所需的 Operator 和 operand 会根据这些设置自动安装。

> **INFO**
>
> 推荐的部署选项↗：对于生成式推理工作负载，推荐使用 **Standard** 方式（之前称为 RawKubernetes Deployment），因为它可对资源分配和扩缩容提供最强控制。

目录

- 下载软件包
- 前提条件
- 上传软件包
- 安装 Alauda AI
  - 操作步骤
  - 验证
- 配置 Alauda AI 实例
  - 部署 Alauda AI

组件管理状态
配置 KServe
配置自定义 OIDC 提供方（可选）
配置控制台语言（可选）
配置 Model Catalog
验证
导入用于 Catalog 的内置模型镜像
获取 OCI 镜像 tar 包
推送到 Harbor
验证 Harbor 导入
FAQ
1. 为 aml-skipper 配置审计输出目录
2. 为 vLLM-ascend 设置 KServe Modelcar UID

下载软件包

有关下载安装包和 `violet` 工具的通用操作步骤，请参见 Upload Packages。

Alauda AI 所需的软件包为：

- **Alauda AI** — 用于模型管理、推理服务和组件生命周期管理的主平台组件。

下载与 Alauda AI 版本和目标集群架构相匹配的软件包版本。

前提条件

下面列出的依赖组件以软件包形式交付。你无需在目标集群上单独安装它们——在安装 Alauda AI 之前先将这些软件包上传到平台仓库，Alauda AI 安装完成后，`AmlCluster` 会自动安装并管理这些组件。

默认需要以下组件：

- **PostgreSQL**
- **Alauda Cache Service for Redis OSS**
- **Alauda Build of Authorino**
- **Alauda Build of Envoy AI Gateway**
- **Alauda Build of KServe**
- **Alauda Build of LeaderWorkerSet**

上传软件包

将 **Alauda AI** 软件包上传到将运行 Alauda AI 的集群。按照 Upload Packages 准备 `violet`，配置平台或外部 registry 凭据，并上传软件包。

上传完成后，按如下所述从 **OperatorHub** 安装 Alauda AI。可选能力会在安装后通过 `AmlCluster` 组件配置启用。

安装 **Alauda AI**

1 操作步骤

在 **Administrator** 视图中：

1. 单击 **Marketplace / OperatorHub**。
2. 在控制台顶部的 **Cluster** 下拉列表中，选择要安装 Alauda AI 的目标集群。
3. 选择 **Alauda AI**，然后单击 **Install**。
   Install Alauda AI 窗口将弹出。
4. 然后在 **Install Alauda AI** 窗口中进行操作。
5. 保持 **Channel** 不变。
6. 检查 **Version** 是否与要安装的 **Alauda AI** 版本匹配。
7. 保持 **Installation Location** 不变，默认应为 `aml-operator`。
8. 将 **Upgrade Strategy** 选择为 **Manual**。
9. 单击 **Install**。

2 验证

确认 **Alauda AI** 磁贴显示以下状态之一：

- `Installing`：安装正在进行中；请等待其变为 `Installed`。
- `Installed`：安装已完成。

配置 Alauda AI 实例

Alauda AI 安装完成后，operator 会自动创建集群级别的 `default` `AmlCluster` 实例。你无需手动创建 `default` 实例。请根据你的环境检查并更新自动创建的实例。

1 部署 Alauda AI

在 **Administrator** 视图中：

1. 单击 **Marketplace / OperatorHub**。
2. 在控制台顶部的 **Cluster** 下拉列表中，选择要安装 Alauda AI Operator 的目标集群。
3. 选择 **Alauda AI**，然后单击它。
4. 在 **Alauda AI** 页面中，单击选项卡中的 **All Instances**。
5. 等待 `default` `AmlCluster` 实例出现，然后编辑它。
6. 从下拉列表中选择 **Deploy Flavor**：
    1. `single-node` 适用于非 HA 部署。
    2. `ha-cluster` 适用于 HA 集群部署（生产环境推荐）。
7. 在 **Domain** 字段中输入一个有效域名。

> **INFO**
>
> 该域名由 ingress gateway 用于暴露模型服务。 大多数情况下，你会希望使用泛域名，例如 *.example.com。
>
> 你可以通过更新 **Domain Certificate Type** 字段来指定以下证书类型：
>
> - `Provided`

- `SelfSigned`
- `ACPDefaultIngress`

默认情况下，配置使用 `SelfSigned` 证书类型来保护到集群的 ingress 流量，证书存储在 **Domain Certificate Secret** 字段中指定的 `knative-serving-cert` secret 里。

2 组件管理状态

Alauda AI 2.8 会自动安装并管理 `AmlCluster` 配置中选择的组件。以下值显示默认组件设置。请将你需要的条目合并到现有的 `default` `AmlCluster` 中；不要创建第二个实例。

```
spec:
  components:
    authorino:
      managementState: Managed
    envoyAIGateway:
      managementState: Unmanaged
    envoyGateway:
      managementState: SharedManaged
    feast:
      managementState: Unmanaged
    knativeServing:
      managementState: Unmanaged
      providerType: Operator
    llamaStack:
      managementState: Unmanaged
    lws:
      managementState: Managed
    mlflow:
      managementState: Unmanaged
    npuOperator:
      managementState: Unmanaged
      values:
        clusterd:
          enabled: false
        devicePlugin:
          enabled: true
        driver:
          enabled: true
          recoveryPolicy:
            autoRecover: false
          upgradePolicy:
            autoUpgrade: false
          version: 25.5.0
        exporter:
          enabled: true
        mindioacp:
          enabled: false
        mindiotft:
          enabled: false
        nodeD:
          enabled: false
        ociRuntime:
          enabled: true
```

```
      rscontroller:
        enabled: false
      trainer:
        enabled: false
  postgres:
    managementState: SharedManaged
  redis:
    managementState: SharedManaged
  servingRuntimeOperator:
    managementState: Unmanaged
  trustyAI:
    managementState: Unmanaged
  workbench:
    managementState: Unmanaged
    values:
      global:
        istio:
          enabled: false
```

`spec.components` 字段使用与以下产品和 Operator 对应的配置键：

Component key	Product or Operator
`authorino`	**Alauda Build of Authorino**
`envoyAIGateway`	**Alauda Build of Envoy AI Gateway**
`envoyGateway`	**Alauda Build of Envoy Gateway** (SharedManaged)
`feast`	**Alauda Build of Feast**
`knativeServing`	**Knative Serving**
`llamaStack`	**Alauda Build of Llama Stack**
`lws`	**Alauda Build of LeaderWorkerSet**
`mlflow`	**MLflow**
`npuOperator`	**Alauda Build of NPU Operator**
`postgres`	**PostgreSQL** (SharedManaged)

Component key	Product or Operator
`redis`	**Alauda Cache Service for Redis OSS** (SharedManaged)
`servingRuntimeOperator`	**Alauda Build of Serving Runtime**
`trustyAI`	**Alauda Build of TrustyAI**
`workbench`	**Alauda AI Workbench**

管理状态含义如下：

Value	Meaning
`Managed`	Alauda AI 管理该组件及其生命周期。对于正常安装，请使用 `Managed`，这样 Alauda AI 会安装并协调该组件。
`Unmanaged`	Alauda AI 不会安装或协调该组件。仅在临时规避、兼容性需求或高级部署场景中使用；必要时请单独管理该组件。
`SharedManaged`	Alauda AI 会部署该组件供共享使用，但不会锁定 Operator 版本。
`Removed`	Alauda AI 会移除该组件及其受管资源。该值不能用于处于 `SharedManaged` 状态的组件；请改为将这些组件设置为 `Unmanaged`。

对于常见的 2.8 部署，当需要相应能力时，请将以下组件开关设置为 `Managed`。`servingRuntimeOperator` 的设置取决于你环境中的加速器和 serving-runtime 要求。

```
spec:
  components:
    envoyAIGateway:
      managementState: Managed
    servingRuntimeOperator:
      managementState: Managed 1
    workbench:
      managementState: Managed
```

1 `servingRuntimeOperator` 控制 **Alauda Build of Serving Runtime**，其中包含用于 Nvidia GPU 的 vLLM 和 llm-d serving runtime，仅适用于 x86。对于需要这些 runtime 的 Nvidia/x86 部署，请启用它。如果你的环境使用 Ascend 或其他非 Nvidia 加速器，请不要启用或安装此组件；应改用适用于该硬件的 serving runtime。

INFO

启用 **Knative** 功能

在现有的 `default` `AmlCluster` 中，将 `spec.components.knativeServing.managementState` 设置为 `SharedManaged`：

```
spec:
  components:
    knativeServing:
      managementState: SharedManaged
```

3 配置 KServe

Alauda Build of KServe 由 Alauda AI 安装并管理。`default` `AmlCluster` 实例中的 `kserve` 组件默认值为 `Managed`，因此 Alauda AI 会安装 KServe Operator 并自动创建 `KServe` 实例。你无需上传 KServe Operator 软件包、从 OperatorHub 安装 Operator，或手动创建 `KServe` 自定义资源。

KServe 参数通过 `spec.components.kserve.values` 暴露。请将所需设置合并到现有的 `default` `AmlCluster` 中：

```yaml
spec:
  components:
    kserve:
      managementState: Managed
      values:
        defaultDeploymentMode: RawDeployment
        domain: example.com
        ingressGateway: knative-serving/knative-ingress-gateway
        kserve:
          storage:
            uidModelcar: 1000
        preset:
          envoyAIGateway:
            port: 1063
            service: ai-gateway-controller
          envoyGateway:
            createInstance: true
            deployType: ControllerNamespace
            instanceName: aieg
            saNamespace: envoy-gateway-system
            serviceAccount: envoy-gateway
          gie:
            enabled: true
          kserveGateway:
            enabled: true
            name: kserve-ingress-gateway
            namespace: kserve
            port: 80
            service_type: NodePort
```

主要的 `kserve` 组件设置：

Field	Description	Default
`managementState`	Alauda AI 是否管理 KServe Operator 和实例。对于正常安装，请使用 `Managed`。	`Managed`
`values.defaultDeploymentMode`	推理服务的部署模式：用于无服务器能力（如	`RawDeployment`

Field	Description	Default
	scale-to-zero）的 `RawDeployment`（`Standard`）或 `Knative`。	
`values.domain`	ingress gateway 用于暴露 inference-service 端点的域名。请使用泛域名，例如 `*.example.com`。	None
`values.ingressGateway`	绑定到 KServe 的 ingress gateway，格式为 `<namespace>/<gateway-name>`。	`knative-serving/knative-ingress-gateway`
`values.kserve.storage.uidModelcar`	用于 Modelcar 工作负载的 UID。如果你计划使用 `vLLM-ascend`，请将其设为 `0`；默认值为 `1000`。	`1000`

gateway 预设会配置位于推理流量前端的 AI gateway 栈：

Field	Description	Default
`preset.envoyGateway.createInstance`	创建一个 Envoy Gateway 实例，使用捆绑的扩展来管理推理流量。	`true`
`preset.envoyGateway.instanceName`	要创建的 Envoy Gateway 实例名称。	`aieg`
`preset.envoyGateway.saNamespace`	Envoy Gateway service account 所在的命名空间。	`envoy-gateway-system`
`preset.envoyGateway.serviceAccount`	Envoy Gateway 使用的 service account 名称。	`envoy-gateway`

Field	Description	Default
`preset.envoyAIGateway.service`	Envoy AI Gateway 的 Kubernetes service 名称。	`ai-gateway-controller`
`preset.envoyAIGateway.port`	Envoy AI Gateway 使用的端口号。	`1063`
`preset.gie.enabled`	启用捆绑的 Gateway API Inference Extension。如果集群中已单独安装 GIE，请将其设置为 `false`。	`true`
`preset.kserveGateway.enabled`	为 InferenceService 流量部署一个 KServe Gateway 实例。	`true`
`preset.kserveGateway.name`	KServe Gateway 的名称。	`kserve-ingress-gateway`
`preset.kserveGateway.namespace`	KServe Gateway 部署所在的命名空间。	`kserve`
`preset.kserveGateway.port`	KServe Gateway 使用的端口号。	`80`
`preset.kserveGateway.service_type`	KServe Gateway 的 service 类型。	`NodePort`

验证 `AmlCluster` 是否已完成协调，以及 `KServe` 实例是否由 Alauda AI 创建：

```
kubectl get amlcluster default

kubectl get kserve default-kserve -n kserve-operator
```

当状态显示 `DEPLOYED: True` 时，`AmlCluster` 应报告 `Phase=Ready` 且 `Reason=Reconciled`，此时 `KServe` 实例即为就绪状态。

4 配置自定义 OIDC 提供方（可选）

默认情况下，Alauda AI 使用 ACP Dex 作为 OIDC 提供方。在此默认配置下，`AmlCluster` 实例中不需要额外的 `spec.oidc` 配置。

如果你希望 Alauda AI 使用其他 OIDC 提供方，请在该提供方中注册 OAuth2/OIDC 客户端，允许 Alauda AI 回调 URL，然后更新 `AmlCluster` YAML 中的 `spec.oidc`。回调 URL 为：

```
https://<platform-address>/clusters/<cluster-name>/aml/oauth2/callback
```

Alauda AI 会从 Alauda AI 安装集群中 `kubeflow` 命名空间内的 Kubernetes Secret 读取 OIDC client secret。默认 Secret 名称为 `aml-oidc-secret`，Secret key 必须为 `client-secret`。请使用 OIDC 提供方中的 client secret 更新此 Secret：

```
kubectl create secret generic aml-oidc-secret \
  -n kubeflow \
  --from-literal=client-secret='<oidc-client-secret>' \
  --dry-run=client -o yaml | kubectl apply -f -
```

然后配置 `spec.oidc`：

```
spec:
  oidc:
    # OIDC issuer URL. This must match the issuer value advertised by
the
    # provider, for example a Keycloak realm URL.
    issuerURL: https://<oidc-provider-issuer>
    # OAuth2/OIDC client ID registered in the external provider.
    clientID: <oidc-client-id>
    # Kubernetes Secret name in the kubeflow namespace. The Secret mu
st
    # contain a client-secret key. Default: aml-oidc-secret.
    clientSecretName: aml-oidc-secret
    # OAuth2 scopes requested during login. Keep this minimal to avoi
d
    # oversized ID/access tokens and oversized login cookies.
    # Default: openid profile email.
    scope: openid profile email
    # Whether to use the preferred_username claim as the email value.
    # Default: true.
    preferredUsernameAsEmail: true
    main:
      # OIDC authorization endpoint. If you use discovery, this maps
from
      # authorization_endpoint.
      loginURL: https://<oidc-provider-authorization-endpoint>
      # OAuth2 callback URL registered in the external provider.
      redirectURL: https://<platform-address>/clusters/<cluster-name
>/aml/oauth2/callback
```

如果提供方在 `<issuerURL>/.well-known/openid-configuration` 暴露了标准 OIDC discovery 文档，那么当这些字段未设置时，Alauda AI 会从 discovery 中自动填充 `redeemURL`、`jwksURL` 和 `profileURL`。如果 discovery 不可用，请显式配置这些字段：

Discovery field	`spec.oidc` field
`authorization_endpoint`	`loginURL`
`token_endpoint`	`redeemURL`

Discovery field	`spec.oidc` field
`userinfo_endpoint`	`profileURL`
`jwks_uri`	`jwksURL`

将映射后的 `loginURL` 值用于 `main.loginURL`；如果你配置了 secondary endpoint，则用于 `secondary.loginURL`。

```
spec:
  oidc:
    redeemURL: https://<oidc-provider-token-endpoint>
    jwksURL: https://<oidc-provider-jwks-endpoint>
    profileURL: https://<oidc-provider-userinfo-endpoint>
```

Keycloak 配置示例：

- 在目标 realm 中创建一个 **OpenID Connect** client。
- 将 **Client ID** 设置为 `spec.oidc.clientID` 中使用的值，例如 `aml`。
- 打开 **Client authentication**。
- 在 **Authentication flow** 下，选择 **Standard flow**。
- 打开 **Require PKCE**，并将 **PKCE Method** 设置为 `S256`。
- 将 **Valid redirect URIs** 设置为 `https://<platform-address>/clusters/<cluster-name>/aml/*`。
- 从 Keycloak client 的 **Credentials** 选项卡复制 client secret，并更新上面显示的 `aml-oidc-secret` Secret。
- 在 client 的 **Client scopes** 设置中，将 `basic`、`email` 和 `profile` 设为 **Default**，并将其他 scopes 设为 **Optional**，除非你的环境明确需要它们。除非必要，请避免添加较大的 claim mapper，例如 groups、realm roles、client roles、address、phone、offline access 以及其他应用特定的 claims。过大的 token 可能会导致 oauth2-proxy cookie 超过浏览器或 ingress header 大小限制，并引发登录循环或 HTTP 431/400 错误。

5 配置控制台语言（可选）

Alauda AI 会从 `AmlCluster` 实例中的 `spec.i18n.languages` 读取可用的控制台语言。默认值为 `en`。

支持的语言代码如下：

Code	Language
`zh`	Chinese (Simplified)
`en`	English
`fr`	French
`ru`	Russian
`ja`	Japanese
`ko`	Korean
`de`	German
`es`	Spanish
`it`	Italian
`pt-BR`	Portuguese (Brazil)
`zh-TW`	Chinese (Traditional)

`en` 和 `zh` 是内置语言。其他语言仅会添加为可选项。若要成功切换到这些语言，你还需要提供对应的翻译文件。

你可以在为其他语言准备翻译文件时，参考内置的英文和中文翻译文件：

```
/clusters/<cluster-name>/aml/console-aml/assets/i18n/en/lich-single.json
/clusters/<cluster-name>/aml/console-aml/assets/i18n/zh/lich-single.json
```

从这些文件生成目标语言的翻译，或者联系 Alauda 获取所需语言的最新翻译内容。

例如，若要在控制台中提供英文、中文和法语可选项，请更新 `AmlCluster` YAML：

```yaml
spec:
  i18n:
    languages:
      - en
      - zh
      - fr
```

然后在 `cpaas-system` 命名空间中为法语创建一个翻译文件 `ConfigMap`：

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    image-load-config: "true"
  name: aml-i18n-fr
  namespace: cpaas-system
data:
  overrides: |
    [{
      "dest": "/aml-i18n/fr/lich-single.json",
      "configMap": {
         "name": "aml-i18n-fr",
         "key": "lich-single.json"
      }
    }]
  lich-single.json: |
    {
      "nav_model_repo": "Referentiel de modeles",
      "nav_infer_svc": "Service d'inference",
      "workbench": "Workbench",
      "tool": "Outils"
    }
```

其他非内置语言可使用相同模式：将 `fr` 替换为目标语言代码，并提供翻译后的 `lich-single.json` 内容。

6 配置 Model Catalog

- **Model OCI Registry Address**：为 Model Catalog 托管模型 OCI 产物的 registry 地址。该字段没有默认值，必须根据你的环境进行配置。
  该 registry 存储 Model Catalog 使用的模型 OCI 镜像。请使用 Harbor 或其他启用 HTTPS 访问的生产模式 OCI registry。Model Catalog 不支持为拉取模型 OCI 镜像配置 `imagePullSecret`，因此用于 Model Catalog 的 Harbor project 或 repository 必须允许推理集群节点匿名拉取。在 Harbor 中，将存储 Model Catalog 镜像的 project 设为 **Public**。
  如果目标环境中无法部署支持 HTTPS 的 registry，可以将 HTTP registry 作为备用方案。在部署模型之前，请先在推理集群中的每个节点上配置容器运行时。对于 containerd，可为该 registry 地址添加 insecure registry mirror，例如创建 `/etc/containerd/certs.d/<registry-host:port>/hosts.toml`：

  ```
  server = "http://<registry-host:port>"

  [host."http://<registry-host:port>"]
    capabilities = ["pull", "resolve"]
  ```

  然后重启 containerd，或通过你的集群管理系统应用等效的节点运行时配置。该配置必须存在于调度推理服务 Pod 的节点上；否则，即使 Model Catalog 能列出该模型，Pod 的镜像拉取仍会失败。containerd 的具体配置路径可能因 Kubernetes 发行版而异；应用配置后，请验证节点能够拉取 Model Catalog 镜像，例如使用 `crictl pull <registry-host:port>/<repository>:<tag>`。
- **Source of PVC**：选择复用现有 PVC 还是创建新的 PVC。使用 `CreateNew` 可让安装过程创建 PVC。
- **StorageClass Name**：创建新 PVC 时使用的 StorageClass。

7 验证

检查配置并保存 `default` `AmlCluster` 实例，然后查看其状态：

```
kubectl get amlcluster default
```

该资源应为 `Ready`：

```
NAME      PHASE   READY   REASON
default   Ready   True    Reconciled
```

导入用于 **Catalog** 的内置模型镜像

Alauda AI 中的 **Catalog** 功能自带一组内置模型 OCI 镜像，用户可以从 Web Console 将其作为推理服务部署。这些镜像必须先导入到 **Model Catalog** 配置的 **OCI registry** 中，**Catalog** 才能提供这些镜像。如果不执行这一步，安装虽然会成功完成，但之后从 Catalog 部署内置模型时会失败，并出现 `ImagePullBackOff`。

1 获取 **OCI** 镜像 **tar** 包

内置模型镜像以 OCI archive tar 包（符合 OCI Image Layout Specification 的 `.tar` 文件）形式交付。每个 tar 包都包含一个模型的多架构镜像（`linux/amd64` + `linux/arm64`）。

你可以从 Customer Portal Marketplace 下载这些 tar 包，或者联系你的 Alauda 支持代表获取与 Alauda AI 版本匹配的软件包。

2 推送到 **Harbor**

推荐的目标 registry 是 Harbor。下面的示例使用 HTTP Harbor registry。如果你的 Harbor registry 使用 HTTPS，请省略 `--plain-http`，并将 API URL 从 `http://` 改为 `https://`。

在安装了 `ctr`、`curl` 和 `jq` 且可以访问 Harbor 的节点上运行这些命令。

首先，设置环境变量：

```
export REG=<harbor-address>:<port> 1
export REPO=mlops/modelcar-qwen3.5-0.8b 2
export TAG=v0.1.0 3
export TAR=./Qwen3.5-0.8B.oci.tar 4
export AUTH='user:password' 5
```

1 Harbor registry 端点，不包含 URL scheme。

2 Harbor 中的目标 repository 路径，格式为 `<project>/<image-name>`。例如，`mlops/modelcar-qwen3.5-0.8b` 使用 Harbor project `mlops` 和 repository `modelcar-qwen3.5-0.8b`。

3 OCI archive 携带的镜像标签。如果你不知道它，可使用下面的命令从 tar 包中提取。

4 上一步获取的 OCI archive tar 包路径。

5 形式为 `user:password` 的 Harbor 凭据。如果你没有这些信息，请联系平台管理员。

tar 包通常在 OCI image layout 中自带其标签（例如 `v0.1.0`）。如有需要，可从 tar 包中提取它：

```
export TAG=$(tar -xOf "$TAR" index.json \
  | jq -r '.manifests[0].annotations["org.opencontainers.image.ref.name"]')
echo "$TAG"   # should print something like v0.1.0
```

检查该镜像标签是否已存在于 Harbor 中：

```
URL="http://$REG/api/v2.0/projects/${REPO%%/*}/repositories/$(printf '%s' "${REPO#*/}" | sed 's|/|%2F|g')/artifacts/$TAG"

HTTP=$(curl -s -u "$AUTH" -o /tmp/harbor-artifact.json -w '%{http_code}' "$URL")

echo "HTTP=$HTTP  URL=$URL"

[ "$HTTP" = 200 ] && jq '{digest, size, push_time, arch: .extra_attrs.architecture, tags: [.tags[].name], platforms: [.references[]?.platform]}' /tmp/harbor-artifact.json \
  || jq -r '.errors[]?.message' /tmp/harbor-artifact.json
```

如果 Harbor project 还不存在，请在推送前先创建它：

```
PROJECT="${REPO%%/*}"

curl -s -u "$AUTH" -X POST "http://$REG/api/v2.0/projects" \
  -H 'Content-Type: application/json' \
  -d "{\"project_name\":\"$PROJECT\",\"public\":true}" \
  -w '\nHTTP %{http_code}\n'
```

如果 project 已存在，Harbor 会返回非 2xx 状态码。确认 project 存在后，请确保其配置为 public project，然后继续导入和推送。Model Catalog 不支持在部署模型 OCI 镜像时配置 `imagePullSecret`，因此推理集群节点必须能够匿名拉取这些镜像。

然后运行导入和推送操作步骤：

```
# 1. Import into the node's containerd content store.
#    --base-name prepends $REG/$REPO to the tag carried inside the ta
rball,
#    producing a fully-qualified reference $REG/$REPO:$TAG.
ctr -n k8s.io images import \
  --all-platforms \
  --base-name "$REG/$REPO" \
  "$TAR"

# 2. Verify the import. You should see "$REG/$REPO:$TAG".
ctr -n k8s.io images ls -q | grep "$REPO"

# 3. Push to Harbor. Use --plain-http only for HTTP Harbor.
ctr -n k8s.io images push \
  --plain-http \
  --user "$AUTH" \
  "$REG/$REPO:$TAG"

# 4. Clean up the local reference on the node. Blob data is reclaimed
by
#    containerd's garbage collector, leaving no persistent state on t
he node.
ctr -n k8s.io images rm "$REG/$REPO:$TAG"
```

对每个内置模型 tar 包重复此操作步骤，并分别变更每个模型对应的 `$REPO`、`$TAG` 和 `$TAR`。

INFO

`--all-platforms` 在 **import** 步骤中至关重要：如果省略它，就只会导入节点宿主架构对应的内容，后续 push 会悄悄遗漏其他平台的 blob。`push` 步骤不需要该参数——推送多架构索引时会自动推送其引用的所有平台。

3 验证 Harbor 导入

确认 Harbor 现在可以提供该镜像：

```
URL="http://$REG/api/v2.0/projects/${REPO%%/*}/repositories/$(printf '%s' "${REPO#*/}" | sed 's|/|%2F|g')/artifacts/$TAG"

HTTP=$(curl -s -u "$AUTH" -o /tmp/harbor-artifact.json -w '%{http_code}' "$URL")

echo "HTTP=$HTTP  URL=$URL"

[ "$HTTP" = 200 ] && jq '{digest, size, push_time, arch: .extra_attrs.architecture, tags: [.tags[].name], platforms: [.references[]?.platform]}' /tmp/harbor-artifact.json \
  || jq -r '.errors[]?.message' /tmp/harbor-artifact.json
```

`HTTP=200` 表示镜像已成功导入到 Harbor。期望输出中包含 digest、size、push time、tag 和平台信息：

```
{
  "digest": "sha256:...",
  "size": 123456789,
  "push_time": "2026-05-06T00:00:00.000Z",
  "arch": "amd64",
  "tags": ["v0.1.0"],
  "platforms": [
    {"architecture": "amd64", "os": "linux"},
    {"architecture": "arm64", "os": "linux"}
  ]
}
```

至此，Alauda AI 的核心能力已成功部署。如果你想快速体验产品，请参阅 Quick Start。

FAQ

1. 为 aml-skipper 配置审计输出目录

默认的审计输出路径为宿主机上的 `/cpaas/audit`。但是，在某些操作系统上（例如 Alauda OS），宿主机的根路径是只读的，无法创建 `/cpaas` 目录。在这种情况下，用户需要修改审计输出路径。

要修改审计输出路径，请更新 `AmlCluster` 默认资源，并在 `spec.values` 下添加 `amlSkipper.auditLogHostPath.path` 配置。例如：

```yaml
apiVersion: amlclusters.aml.dev/v1alpha1
kind: AmlCluster
metadata:
  name: default
  ...
spec:
  ...
  values:
    amlSkipper:
      auditLogHostPath:
        path: /var/lib/audit
```

> **NOTE**
>
> 具体路径应与 `Alauda Container Platform Log Collector` 的采集配置保持一致。

2. 为 vLLM-ascend 设置 KServe Modelcar UID

如果你计划使用 `vLLM-ascend`，请在 `default` `AmlCluster` 中将 KServe Modelcar UID 设置为 `0`（默认值为 `1000`）：

```yaml
spec:
  components:
    kserve:
      values:
        kserve:
          storage:
            uidModelcar: 0
```

该设置作用于集群级别，并会影响 Alauda AI 安装集群中的所有 Modelcar 工作负载。

安装 Workbench

在 Alauda AI 2.8.x 中，Workbench 通常由 Alauda AI 安装和管理。请在集群范围的 `default` `AmlCluster` 中启用 `workbench` 组件；你无需上传 Workbench Operator 包、单独安装 Operator，或手动创建 `Workbench` 自定义资源。

目录

前提条件

启用 Workbench

验证

前提条件

- 目标集群上已安装 Alauda AI。请参见 安装 Alauda AI。
- 目标集群具有足够的资源来运行 Workbench 工作负载。
- 可选： 仅当需要为 Elyra Kubeflow Pipelines run-URL 重定向时，才需要 Alauda Service Mesh (Istio)。默认情况下会禁用 Istio 集成。

启用 Workbench

在 管理员 视图中：

1. 打开 **Marketplace** > **OperatorHub**。
2. 选择目标集群并打开 **Alauda AI**。
3. 打开 **All Instances** 选项卡，编辑 `default` `AmlCluster` 实例。
4. 将 `spec.components.workbench.managementState` 设置为 `Managed`。
5. 保存 `AmlCluster` 实例，并等待 Workbench 就绪。

你可以在 `AmlCluster` YAML 中应用该组件设置：

```yaml
spec:
  components:
    workbench:
      managementState: Managed
      values:
        global:
          istio:
            enabled: false
```

仅当目标集群运行 Alauda Service Mesh 且你需要 Elyra 生成的 KFP run URL 通过 Kubeflow Istio ingress gateway 重定向时，才将 `spec.components.workbench.values.global.istio.enabled` 设置为 `true`。

> **NOTE**
>
> 默认的 `workbench` 状态为 `Unmanaged`，因此在你显式将该组件设置为 `Managed` 之前，不会安装 Workbench。对于常规部署，请使用 `Managed`，并让 Alauda AI 安装和协调 Workbench。

验证

检查 `AmlCluster` 状态：

```
kubectl get amlcluster default
```

该资源应报告 `Phase=Ready` 和 `Reason=Reconciled`。你还可以验证由 Alauda AI 创建的 Workbench 资源：

```
# The Workbench instance is ready.
kubectl get workbench -A

# The workspace controller is available.
kubectl -n kubeflow get deploy workspace-controller-controller-manager

# Built-in IDE templates are registered.
kubectl get workspacekind
```

部署就绪后，打开 **Alauda AI**，确认 **Workbench** 已显示，并且你可以打开 Workbench 列表。

配置 Alauda AI 入口和品牌

从 Alauda AI 2.6 开始，Alauda AI 以单集群应用的形式部署。全局资源（OIDC client 和产品入口）不再作为集群插件交付，而是通过 `global-install.sh` 脚本进行配置。

> **WARNING**
>
> 如果在早期版本中安装了 **Alauda AI Essentials** 集群插件，则在升级到 Alauda AI 2.6 或更高版本后必须将其卸载。在 管理员 视图中，导航到 **Marketplace / Cluster Plugin**，选择 **global** 集群，找到 **Alauda AI Essentials**，然后单击 **Uninstall**。

该脚本会在 global 集群上创建以下资源：

- **OAuth2Client**：通过平台的 Dex Identity Provider 为 Alauda AI 启用 SSO 认证。
- **Secret**：存储 OIDC client 凭据。
- **ProductEntry**：将 Alauda AI 注册为平台控制台中的产品入口，使其显示在左侧导航菜单中。

目录

前提条件
操作步骤
下载脚本
运行脚本
重要说明
验证

前提条件

- 已安装并配置 `kubectl`，可用于访问 **global** 集群。
- global 集群中存在 `productbase.product.alauda.io/base` 资源（脚本会从中读取平台 URL 和版本）。

操作步骤

1 下载脚本

下载 global-install.sh 脚本。

2 运行脚本

以目标集群名称作为参数运行脚本：

```
bash global-install.sh \ 1
    <cluster-name> 2
```

1 `<cluster-name>` 是安装 Alauda AI 的集群名称，或者将要安装 Alauda AI 的集群名称。

脚本会从 global 集群上的 `productbase.product.alauda.io/base` 中读取平台 URL 和版本，然后创建所需的 OIDC 和产品入口资源。如果已存在同名 `OAuth2Client`，则会安全地跳过。

3 重要说明

OIDC 配置：global 集群中的 `OAuth2Client` 资源与目标集群中的 `AmlCluster` 之间必须匹配 OIDC client 凭据。`OAuth2Client` 在 `cpaas-system` 命名空间下命名为 `aml`。其 `secret`（或 `secretRef`）必须与 `AmlCluster` `spec.oidc.clientSecretName` 所引用的 Secret 中的 `data.client-secret` 保持一

致（位于 `kubeflow` 命名空间下）。如果需要自定义，请同时更新两端。一旦 `OAuth2Client` 存在，脚本会跳过它——后续更改应直接在该资源上进行。

多个 **Alauda AI** 安装：ProductEntry 资源命名为 `aml-<cluster-name>`。如果在不同集群中部署了多个 Alauda AI 实例，它们都会拥有一个单独的 ProductEntry，但显示名称相同，均为 “Alauda AI”。要在平台控制台导航菜单中区分它们，请编辑对应 ProductEntry 的 `spec.displayName` 字段。例如：

```
kubectl patch productentry aml-<cluster-name> --type=merge -p '{"spe
c":{"displayName":{"en":"Alauda AI (Prod)","zh":"Alauda AI (Pro
d)"}}}'
```

产品内品牌设置：此设置应用于安装 Alauda AI 的集群，而不是上面资源所使用的 global 集群。在运行命令之前，请将 `kubectl` 上下文切换到 Alauda AI 安装集群。要自定义 Alauda AI 内部显示的产品名称、logo 和浏览器 favicon，请更新 `kubeflow` 命名空间中的 `ui-config` `ConfigMap`。将 `data.override` 设置为 `"true"`，然后配置 `data.productName`、`data.logo` 和 `data.favicon`。`logo` 和 `favicon` 的值必须是完整的 data URI 值，例如 `<base64-encoded-svg>` 或 `<base64-encoded-png>`。

```
kubectl patch configmap ui-config -n kubeflow --type=merge -p '{
  "data": {
    "override": "true",
    "productName": "My AI Platform",
    "logo": "<base64-encoded-logo>",
    "favicon": "<base64-encoded-favicon>"
  }
}'
```

4 验证

运行脚本后，确认已创建这些资源：

```
# Check the OAuth2Client
kubectl get oauth2client -n cpaas-system

# Check the ProductEntry
kubectl get productentry
```

经过短暂延迟后，**Alauda AI** 入口应出现在平台控制台左侧导航菜单中。

Tools 菜单配置

本文档描述了如何向左侧导航中的 **Tools** 部分添加自定义工具链接，例如集成 **MLFlow**。

从 AML 1.3 开始，Tools 菜单不再通过手动编辑 `kube-public/centraldashboard-config` 来维护。取而代之的是使用一种 自动发现机制：系统会扫描所有带有 `aml.cpaas.io/centralMenuItem: "true"` 标签的 `ConfigMap` 资源，并自动合并它们以生成 Tools 菜单配置。

当安装 MLflow Operator 插件时，它会根据 operator 配置自动创建 MLflow Tools 菜单 `ConfigMap`。仅在自定义集成或排障时使用下面的手动示例。

> **WARNING**
>
> 请勿手动编辑 `kube-public/centraldashboard-config`。任何手动更改都会在升级或重启期间被还原。

目录

添加 Tools 菜单项

字段说明

结果

添加 Tools 菜单项

您可以通过创建符合要求格式的 `ConfigMap` 来向 Tools 菜单添加新条目。

示例（MLFlow）：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: aml-mlflow-menu-config
  namespace: kubeflow
  labels:
    aml.cpaas.io/centralMenuItem: "true"
data:
  menu: |-
    - link: "https://<platform-access-address>/clusters/<cluster-name>/ml
flow/"
      parentUid: advanced
      title: MLFlow
      titleI18N:
        en: MLFlow
      tags: [kubeflow]
      order: 200040
```

字段说明

- `link`：单击菜单项时打开的 URL
- `parentUid`：必须设置为 `advanced`，这样该项会挂载到 **Tools** 菜单下
- `title / titleI18N`：菜单项的显示名称，支持国际化
- `tags`：在安装 Alauda AI 时，系统会将多个 `ConfigMap` 聚合为一个 `ConfigMap`。在此聚合过程中，系统会自动向合并后的配置中添加一个 `aml-feature-tags` 字段。默认情况下，该字段包含内置标签，例如 `addon`、`core`、`datasets`、`pretrain`、`tune-models` 和 `workbench`。
  在上面的示例中，如果某个菜单项将 `tags` 定义为 `kubeflow`，那么 `aml-feature-tags` 中也会包含这个 `kubeflow` 条目。 左侧导航的聚合机制采用 白名单 设计。例如，运行以下命令后，页面上只会显示 `tags` 中包含 `kubeflow` 的导航项：

  ```
  kubectl annotate configmap aml-centraldashboard-config -n kube-public a
  ml.cpaas.io/centralTagsFilter="kubeflow" --overwrite
  ```

在过滤过程中，所有不匹配 `kubeflow` 的其他标签都会被判定为 `false`，因此不会显示在导航中。

- `order`：菜单排序值；数字越小越靠前

结果

- 创建 `ConfigMap` 后，其内容会自动合并到 `centraldashboard-config` 的 `data` 部分，例如：

```
apiVersion: v1
data:
  aml-feature-tags: |-
    {
      "addon": true,
      "core": true,
      "datasets": true,
      "kubeflow": true,
      "pretrain": true,
      "tune-models": true,
      "workbench": true
    }
  links: |-
    {
      "menuLinks": [
        {
          "type": "item",
          "link": "/model-repo/model",
          "i18nKey": "nav_model_repo",
          "text": "模型仓库",
          "en": "Model Repository",
          "icon": "model-repo"
        },
        {
          "type": "item",
          "link": "/model-repo/service-manage",
          "i18nKey": "nav_infer_svc",
          "text": "推理服务",
          "en": "Inference Service",
          "icon": "service-manage"
        },
        {
          "type": "item",
          "link": "workbench",
          "i18nKey": "workbench",
          "text": "Workbench",
          "en": "Workbench",
          "icon": "sitemap"
        },
        {
          "type": "item",
          "link": "",
          "i18nKey": "nav_observe",
```

```
    "text": "可观测",
    "en": "Observe",
    "icon": "observe",
    "children": [
      {
        "type": "item",
        "link": "monitor",
        "i18nKey": "nav_monitor",
        "text": "监控面板",
        "en": "Monitor",
        "icon": ""
      }
    ]
  },
  {
    "type": "item",
    "link": "/model-repo/dataset",
    "i18nKey": "nav_dataset",
    "text": "数据集",
    "en": "Dataset",
    "icon": "data-set"
  },
  {
    "type": "item",
    "link": "",
    "i18nKey": "nav_model_optimization",
    "text": "模型优化",
    "en": "Model Optimization",
    "icon": "model-repo",
    "children": [
      {
        "type": "item",
        "link": "/model-repo/training",
        "i18nKey": "nav_pre_train",
        "text": "预训练",
        "en": "PreTraining",
        "icon": ""
      },
      {
        "type": "item",
        "link": "/model-repo/task-template",
        "i18nKey": "nav_task_temp",
        "text": "任务模板",
        "en": "Task Template",
```

```
              "icon": ""
            },
            {
              "type": "item",
              "link": "/model-repo/fine-tuning",
              "i18nKey": "nav_fine_tuning",
              "text": "模型微调",
              "en": "Fine-Tuning",
              "icon": ""
            }
          ]
        },
        {
          "type": "item",
          "link": "",
          "i18nKey": "tool",
          "text": "工具",
          "en": "Tools",
          "icon": "advanced",
          "children": [
            {
              "type": "item",
              "link": "https://<platform-access-address>/clusters/<clus
ter-name>/mlflow/",
              "i18nKey": "",
              "text": "MLFlow",
              "en": "MLFlow",
              "icon": ""
            }
          ]
        }
      ]
    }
kind: ConfigMap
metadata:
  name: centraldashboard-config
  namespace: kube-public
```

升级

升级 Alauda AI

准备升级软件包

升级前操作

移除 Cluster Plugins

准备 global 集群

升级后操作

迁移到 Knative Operator

1. 移除旧版 Serving 实例

2. 安装 Knative Operator 并创建 Serving

升级 Workb

从 Workbench

从 Kubeflow No

操作步骤

卸载

卸载流程概览

卸载由 AmlCluster 管理的组件

卸载未由 AmlCluster 管理的组件

卸载 Alauda AI 核心

卸载后保留的数据

常规说明

升级 Alauda AI

INFO

从 **Alauda AI 2.3.x** 升级到 **Alauda AI 2.8.x**。

本文档描述了从 Alauda AI 2.3.x 部署模型迁移到 Alauda AI 2.8.x 部署模型的过程。开始之前，请先通读完整的操作步骤。此次升级会变更若干组件的部署形式，其中一些组件需要执行迁移，而不是原地升级。

WARNING

Alauda AI 2.8.x 不支持 **ACP 4.0.x**。 如果目标环境运行的是 ACP 4.0.x，请先将 ACP 升级到 **ACP 4.1.x** 到 **4.3.x** 范围内的受支持版本，然后再升级 Alauda AI。

WARNING

在卸载 Cluster Plugin 或 Operator 之前，请先查看对应组件的迁移要求，并保留重要的用户数据和配置。尽可能导出自定义资源和配置，并确认用户 PVC、数据库和对象存储制品将被保留。仅在环境确实需要且相关迁移操作明确要求时，才使用存储快照或数据库备份。除非相关迁移步骤明确要求，否则不要删除 PVC 或自定义资源。

目录

准备升级软件包

上传 operator 软件包
升级前操作
保留现有资源
检查目标集群
移除 Cluster Plugins
准备 global 集群
运行 `global-install.sh`
运行 `migrate-roles.sh`
升级 Alauda AI
升级后操作
验证 Alauda AI
验证 KServe
验证 Operators 和已迁移组件
验证数据保留

准备升级软件包

在开始升级之前，请上传 Alauda AI 2.8.x operator 软件包以及所有必需的依赖软件包。将软件包上传到对应组件将运行所在的集群。关于公共软件包下载、`violet` 配置和上传操作，请参见 Upload Packages。

所需的软件包集合取决于现有环境中启用的功能。对于 Alauda AI 2.8.x 环境，至少请准备以下软件包：

软件包或组件	用途
Alauda AI	Alauda AI 平台核心组件。
Alauda Build of KServe	模型服务组件和推理服务。
Alauda Build of Serving Runtime	受支持推理工作负载的 Serving runtime。
Alauda Build of LeaderWorkerSet	LeaderWorkerSet 工作负载。

软件包或组件	用途
PostgreSQL	受支持组件所需的 PostgreSQL 服务。
Alauda Build of Authorino	MaaS 的认证和授权。
Alauda Cache Service for Redis OSS	受支持组件所需的缓存服务。
Alauda AI Workbench Operator	Workbench 环境和工作区管理。
MLflow Operator	MLflow 跟踪和制品管理。
选择的 Alauda AI 2.8.x 功能所需的其他 operator	例如，启用训练、gateway 或 MLOps 功能所需的 operator。

准确的软件包文件名和版本随 Alauda AI 2.8.x 发布包提供。不要使用 Alauda AI 2.3.x 的软件包进行替代。

上传 **operator** 软件包

公共软件包指南包含 `violet push` 命令、外部 registry 选项以及多软件包上传示例。请使用 Upload Packages 中的方法，结合 Alauda AI 2.8.x 发布包提供的文件名和目标集群信息上传各个软件包。

升级前操作

在升级 **Alauda AI** 之前，请完成以下操作。

保留现有资源

查看每个组件的迁移要求，并保留迁移过程中必须继续存在的资源。此升级中的组件设计为保留其数据；在可行的情况下，请保持数据原位，而不要重新创建。至少需要保留并记录以下内容：

- `AmlCluster` 资源及其当前 YAML 配置。
- 由 Workbench、MLflow、LWS、KServe 及其他已启用组件管理的自定义资源。

- Workspaces、`WorkspaceKind` 资源、`aml-workbench-config` `ConfigMap` 和用户 PVC。
- MLflow 元数据和制品存储配置。
- 推理服务配置和模型存储配置。
- 任何组件特定的 Secret、ConfigMap、RoleBinding 和授权策略。

在卸载旧的部署形式之前，请使用对应组件的备份说明。例如，Workbench 升级指南 会备份 Workbench 资源，并要求保留用户 PVC。

检查目标集群

确认以下事项：

1. 目标集群运行正常，并且有足够的容量容纳新的 operators 及其 operands。
2. 目标集群可以从已配置的 registry 拉取镜像。
3. 所需的 StorageClass 和 PersistentVolume 可用。
4. 平台管理员凭据可以通过 OperatorHub 上传软件包并安装 Operator。
5. 可以使用 `kubectl` 访问 global 集群，以执行下面描述的全局迁移步骤。

移除 **Cluster Plugins**

Alauda AI 2.8.x 将多个组件从 Cluster Plugin 形态改为 Operator 形态。这些部署形式不支持原地升级。在升级 **Alauda AI** 之前，请移除现有环境中已安装组件的旧版 Cluster Plugin：

- **Alauda AI Essentials**，位于 **global** 集群。
- **Alauda Build of LeaderWorkerSet**，位于目标集群。
- **Alauda AI Workbench**，位于目标集群。
- **MLflow**，位于目标集群。

请按以下通用步骤操作：

1. 记录已安装组件的现有自定义资源和配置。
2. 保留相关用户数据，包括 Workbench PVC、MLflow 数据库和制品，以及由 LeaderWorkerSet 管理的工作负载。

3. 在 **Administrator** > **Marketplace** > **Cluster Plugins** 中，选择相应集群：
   - 选择 **global** 集群，并卸载 **Alauda AI Essentials**（如果已安装）。
   - 选择目标集群，并卸载已安装的 **Alauda Build of LeaderWorkerSet**、**Alauda AI Workbench** 和 **MLflow** Cluster Plugin。
4. 除非相关组件的迁移步骤明确要求，否则不要删除已保留的 PVC、数据库、对象存储制品或自定义资源。
5. 在移除旧的 Cluster Plugin 之后，通过 Alauda AI 2.8.x 部署模型启用替代组件。升级后的 `AmlCluster` 会在 `lws.managementState` 为 `Managed` 时安装并协调 **Alauda Build of LeaderWorkerSet**。关于 **Alauda AI Workbench** 和 **MLflow**，请按照其组件特定文档安装替代 Operator 并创建对应的自定义资源。

现有的集群级 `AmlCluster` 资源 `default` 负责控制 Alauda AI 的部署配置。升级 **Alauda AI** 后，请使用该资源配置由 Alauda AI 管理的组件，包括 **Alauda Build of LeaderWorkerSet**。作为独立 Operator 提供的组件（例如 Workbench 和 MLflow）则通过各自的 Operator 自定义资源进行部署和配置。

> **WARNING**
>
> 在卸载 Cluster Plugin 之前，请确认其 CRD、自定义资源或依赖资源是否会被删除。如果卸载可能会删除 CRD，请在继续之前确认用户数据已被保留。

关于 Workbench 特定的数据保留要求，请参见 Migrating from the Workbench Cluster Plugin。关于替代 Operator 的自定义资源，请参见 Install Workbench 和 Install MLflow。

准备 global 集群

Alauda AI 2.8.x 使用单集群应用架构。全局资源不再由旧的 **Alauda AI Essentials** (`aml-global`) Cluster Plugin 提供。

以下命令必须针对 **global** 集群 执行，而不是 Alauda AI 工作负载运行所在的集群。请在卸载旧的 **Alauda AI Essentials** Cluster Plugin 后完成这些操作。

运行 `global-install.sh`

下载随 Alauda AI 2.8.x 文档或发布包提供的 `global-install.sh` 脚本。在 global 集群上，使用 Alauda AI 所安装的集群名称运行该脚本：

```
kubectl config use-context <global-cluster-context>
bash global-install.sh <target-cluster-name>
```

该脚本会创建或验证 Alauda AI 使用的全局 `OAuth2Client`、OIDC Secret 和 `ProductEntry` 资源。脚本完成后，请验证这些资源：

```
kubectl get oauth2client -n cpaas-system
kubectl get productentry
```

平台同步新的 ProductEntry 后，平台控制台中应可见 **Alauda AI** 条目。

运行 `migrate-roles.sh`

在旧的 Alauda AI Essentials plugin 移除后，请在 global 集群上运行随 Alauda AI 2.8.x 发布包提供的 `migrate-roles.sh` 脚本。

请使用该脚本提供的命令行用法和参数。由于发布脚本尚未添加到文档包中，因此此处暂不提供确切的调用方式。请确认脚本成功完成，并且迁移后的命名空间权限已存在，然后再继续。

升级 Alauda AI

1. 登录 Web Console，并打开 **Administrator** 视图。
2. 转到 **Marketplace** > **OperatorHub**。
3. 选择目标集群。
4. 打开 **Alauda AI**。
5. 查看可用版本，并选择 Alauda AI 2.8.x 版本。
6. 确认升级，并等待 OperatorHub 安装状态变为 `Installed`。

在升级过程中，现有的集群级 `AmlCluster` 资源 `default` 会自动升级。升级后的 `AmlCluster` 会自动启用以下组件：

- **PostgreSQL**

- **Alauda Cache Service for Redis OSS**
- **Alauda Build of Authorino**
- **Alauda Build of KServe**
- **Alauda Build of LeaderWorkerSet**

如果需要，您可以在 `AmlCluster` 配置中手动启用以下可选组件：

- **MLflow Operator**
- **Alauda AI Workbench Operator**
- **Alauda Build of Serving Runtime**

关于 `AmlCluster` 配置、registry 设置、组件选项和安装详情，请参见 Install Alauda AI。

升级后操作

验证 **Alauda AI**

检查集群级 `AmlCluster` 资源的状态：

```
kubectl get amlcluster default
```

该资源应为 `Ready` ：

```
NAME      PHASE   READY   REASON
default   Ready   True    Reconciled
```

验证 **KServe**

如果已安装 Alauda Build of KServe，请验证 KServe 实例：

```
kubectl get kserve default-kserve -n kserve-operator
```

该实例应报告 `DEPLOYED: True`：

```
NAME             DEPLOYED   REASON
 default-kserve   True       UpgradeSuccessful
```

验证 **Operators** 和已迁移组件

验证每个已安装依赖项的 OperatorHub 和工作负载状态：

- 确认每个必需组件均为 `Installed`，且其 ClusterServiceVersion 报告 `Succeeded`。
- 确认受启用功能所需的 PostgreSQL、Redis 和 Authorino 服务均已就绪。
- 确认 LWS controller 正在运行，并且一个代表性的分布式工作负载可以启动。
- 确认 MLflow tracking server 已就绪，并且可以访问现有元数据和制品。
- 确认 Workbench 自定义资源已就绪，现有 Workspaces 和 PVC 已存在，并且可以打开一个测试 Workbench。
- 确认现有 KServe 推理服务和具有代表性的模型请求均按预期工作。
- 在 global 集群上，确认 `global-install.sh` 创建的 `OAuth2Client` 和 `ProductEntry` 资源，并验证用户可以通过平台入口打开 Alauda AI。

验证数据保留

在确认升级完成之前，请验证以下数据仍然可用：

- 用户 PVC 和 Workbench 主目录。
- 现有 Workspaces 和已保留的 `WorkspaceKind` 资源。
- MLflow 元数据和模型制品。
- 现有推理服务配置和模型数据。
- 分布式工作负载配置和检查点。

如果某个组件未能成功协调，请停止后续迁移，并使用该组件特定的备份和回滚步骤。不要在仍存在自定义资源时删除 CRD；删除 CRD 也可能会删除存储在集群中的自定义资源。

迁移到 Knative Operator

在 1.x 系列产品中，推理服务的 serverless 能力由 `Alauda AI Model Serving` operator 提供。在 2.x 系列中，该能力由 `Knative Operator` 提供。本节将指导您将 serverless 能力从旧版 operator 迁移到新 operator。

目录

1. 移除旧版 Serving 实例

操作步骤

2. 安装 Knative Operator 并创建 Serving 实例

1. 移除旧版 Serving 实例

1 操作步骤

在 管理员 视图中：

1. 单击 **Marketplace / OperatorHub**。
2. 在控制台顶部，从 **Cluster** 下拉列表中选择已安装 **Alauda AI** 的目标集群。
3. 选择 **Alauda AI**，然后单击 **All Instances** 选项卡。
4. 找到 `default` 实例并单击 **Update**。
5. 在更新页面中，切换到 **YAML** 视图。

6. 将 `spec.components.knativeServing.managementState` 设置为 `Removed`，例如：

```yaml
spec:
  components:
    knativeServing:
      managementState: Removed
```

7. 单击 **Update** 以应用更改。

2. 安装 Knative Operator 并创建 Serving 实例

从 Marketplace 安装 **Knative Operator**，并创建 `KnativeServing` 实例。有关详细说明，请参阅启用 Knative 功能部分。

INFO

完成上述步骤后，Knative serving 控制平面的迁移即完成。

- 如果您正在从 **Alauda AI 2.0** + **Alauda AI Model Serving** 组合迁移，则此处迁移已完全完成。业务服务将很快自动切换其配置。
- 如果您正在从 **Alauda AI 1.x** + **Alauda AI Model Serving** 组合迁移，请确保 **Alauda AI** 同时升级到 **2.x** 版本。

升级

目录

从 Workbench 集群插件迁移

从 Kubeflow Notebook 迁移

操作步骤

从 **Workbench** 集群插件迁移

从 v0.2.0 开始，Workbench 作为 OLM Helm operator 提供，而不是作为 Cluster Plugin 提供。这两种形式之间不支持原地升级。

1. 备份现有的 Workbench 资源：

   ```
   kubectl get workspacekinds,workspaces -A -o yaml > workbench-backup.yaml
   ```

2. 在 管理员 > **Marketplace** > **Cluster Plugins** 中，卸载之前的 **Workbench** 集群插件。保留用户 PVC、保留的 `WorkspaceKind` 资源以及 `aml-workbench-config` `ConfigMap`；operator 会接管或重新应用这些资源。

3. 在 `default` `AmlCluster` 中将 `spec.components.workbench.managementState` 设置为 `Managed`，以启用 Workbench。Alauda AI 会安装并管理 Workbench Operator 及其资

源；请勿手动创建 `Workbench` 自定义资源。有关正常安装路径，请参见安装 Workbench。

4. 验证现有的 Workspaces 及其 PVC 仍然存在，然后创建并连接到一个测试 Workbench。

如果之前的安装依赖 Elyra KFP run-URL 重定向，请在 `default` `AmlCluster` 中设置 `spec.components.workbench.values.global.istio.enabled: true`。Istio 集成是可选的，默认处于禁用状态。

从 **Kubeflow Notebook** 迁移

Workbench 与“Kubeflow Notebook”（Alauda AI <= 1.3）不兼容。您需要创建新的“workbench”实例，“Kubeflow Notebook”将移至左侧导航栏中的“Advanced - Kubeflow”。

> **WARNING**
>
> 我们建议迁移到 Workbench，因为在后续的上游 Kubeflow 版本中，“Kubeflow Notebook”将被弃用。

> **NOTE**
>
> 您可以保留 Alauda AI 1.3 中 “Kubeflow Notebook” 实例使用的 PVC。删除 **Notebook** 实例，并将该 PVC 挂载到新的 Workbench 实例中，这样 Notebook 实例中的数据仍然可用。(注意 容器中的数据将会丢失，就像您使用 `pip install` 将包安装到系统而不是虚拟环境中一样。)

操作步骤

1. 登录并打开 **Alauda AI** 页面。
2. 转到 **Workbench** 打开列表。
3. 单击 **Create** 并填写所需字段。
4. 在 **Home Directory** 中，选择之前 Notebook 使用的 PVC。

卸载

本文档将指导你如何从集群中完整卸载 Alauda AI 及其相关组件。

在 Alauda AI 2.8 中，组件通过集群范围的 `default` `AmlCluster` 实例由 Alauda AI 安装和管理。请按照本文档中的顺序，先卸载组件，再卸载 Alauda AI 核心。卸载 Alauda AI 及其 Operator 不会自动移除自定义资源或 CRD；在继续之前，请先查看 常规说明。

> **WARNING**
>
> 在卸载之前，请备份所有重要数据和模型。与数据相关的资源在卸载后可能仍会保留，删除这些资源可能会永久删除关联数据。删除任何资源之前，请先查看 卸载后保留的数据。

目录

- 卸载流程概览
- 卸载由 AmlCluster 管理的组件
  - Managed 组件：设置为 `Removed`
  - SharedManaged 组件：设置为 `Unmanaged`，然后在 OperatorHub 中卸载
- 卸载未由 AmlCluster 管理的组件
- 卸载 Alauda AI 核心
  - 删除 `default` `AmlCluster` 实例
  - 卸载 Alauda AI Operator
- 卸载后保留的数据
- 常规说明

在卸载 Operator 之前先删除 operator 实例

卸载不会移除 CRD

卸载流程概览

组件类别	卸载方式
`default` `AmlCluster` 中处于 `Managed` 状态的组件（例如 `authorino`、`lws`、`kserve`）	将 `managementState` 设置为 `Removed`
处于 `SharedManaged` 状态的组件（例如 `envoyGateway`、`postgres`、`redis`）	将 `managementState` 设置为 `Unmanaged`，然后在 OperatorHub 中卸载对应的 Operator
未由 `AmlCluster` 管理的组件	在 OperatorHub 中卸载，或者如果作为集群插件安装，则在 集群插件 中卸载
Alauda AI 核心	删除 `default` `AmlCluster` 实例，然后在 OperatorHub 中卸载 Alauda AI Operator

卸载由 **AmlCluster** 管理的组件

Managed 组件：设置为 `Removed`

由 Alauda AI 管理的组件（`managementState: Managed`）可以通过将 `default` `AmlCluster` 中的状态改为 `Removed` 来卸载。随后，Alauda AI 会移除该组件及其所管理的资源。

例如，要卸载 KServe：

```
spec:
  components:
    kserve:
      managementState: Removed
```

SharedManaged 组件：设置为 `Unmanaged`，然后在 OperatorHub 中卸载

处于 `SharedManaged` 状态的组件（`envoyGateway`、`postgres`、`redis`）不支持 `Removed` 值。请先将其 `managementState` 设置为 `Unmanaged`，然后在 OperatorHub 中卸载对应的 Operator。

例如，要卸载 PostgreSQL：

```
spec:
  components:
    postgres:
      managementState: Unmanaged
```

然后，在 管理员 视图中：

1. 点击 **Marketplace / OperatorHub** 并选择目标集群。
2. 找到该组件的 Operator（例如 **PostgreSQL**）。
3. 点击 卸载 并确认。

有关管理状态和组件键，请参见 组件管理状态。

卸载未由 AmlCluster 管理的组件

有些组件不受 `default` `AmlCluster` 管理。这包括 `managementState` 为 `Unmanaged` 的组件，以及独立安装的组件，例如 **Kagenti**、**Alauda Build of MCP Lifecycle Operator**、**Dify** 和 **Alauda Build of InferNex Bridge**。请在 OperatorHub 中卸载这些组件；如果该组件是作为集群插件安装的，则在 集群插件 中卸载。

在 管理员 视图中：

1. 点击 **Marketplace / OperatorHub**（或 **Marketplace /** 集群插件）并选择目标集群。
2. 找到要卸载的组件。
3. 点击 卸载 并确认。

卸载 **Alauda AI** 核心

卸载核心 Alauda AI 分为两步：先删除 `default` `AmlCluster` 实例，然后在 OperatorHub 中卸载 Alauda AI Operator。

删除 `default` `AmlCluster` 实例

1. 在 管理员 视图中，点击 **Marketplace / OperatorHub** 并选择目标集群。
2. 选择 **Alauda AI**，然后打开 **All Instances** 选项卡。
3. 找到名为 `default` 的实例。
4. 点击 **...** 菜单，然后选择 **Delete**，并确认。

验证该实例是否已删除：

```
kubectl get amlcluster default
```

该命令会返回 `No resources found` 或类似信息。

卸载 **Alauda AI Operator**

1. 在 管理员 视图中，点击 **Marketplace / OperatorHub** 并选择目标集群。
2. 选择 **Alauda AI**，然后点击 卸载。
3. 在确认窗口中，确认安装位置（默认是 `aml-operator`），然后点击 卸载。

验证 **Alauda AI** 磁贴是否从 `Installed` 变为 `Not Installed`，或者从列表中消失。

卸载后保留的数据

WARNING

在卸载 Alauda AI 后，`kubeflow` 和 `aml-maas` 命名空间中仍会保留与数据相关的资源。删除这些资源可能会导致数据丢失。删除之前请先检查这些资源，只有在确认不再需要相关数据时才删除。

常规说明

在卸载 **Operator** 之前先删除 **operator** 实例

对于每个 Operator，如果你希望删除其管理的自定义资源实例，请务必在卸载 Operator 之前删除这些实例。卸载 Operator 不会删除其实例；这些实例以及它们引用的数据仍会保留在集群中。

卸载不会移除 **CRD**

从 OperatorHub 卸载 Cluster Plugin 或 Operator 不会删除其安装的 CRD。如果你希望删除 CRD，需要单独删除。请注意，删除 CRD 也会删除该类型的所有自定义资源，并可能移除数据。

平台管理

本章面向平台管理员。它涵盖硬件配置、namespace 和多租户管理，以及跨组件管理任务，例如 MLflow 工作区权限和 Kagenti 安全配置文件。

本章所使用的组件位于其他位置：在 Develop 中的 MLflow，以及在 Build AI Applications 中的 Kagenti。训练工作负载的配额和调度治理位于 Train。

硬件配置

介绍

为什么需要 Hardware Profile？

硬件配置文件管理

创建硬件配置文件

更新硬件配置文件

使用 CLI 创建

前提条件

创建 HardwareP

rofile

将工作负载调度到特定的 GPU 节点

使用节点选择器

使用污点和容忍

组合配置

创建仅 CPU

示例 1：仅 CPU

示例 2：GPU 加

多租户

Namespace 管理

MLflow 工作区与访问控制

将 namespace 暴露为工作区

授予用户访问权限

从客户端选择工作区

故障排查

安全配置文件

安装 Secure Profile 依赖项

通用安装模式

SPIRE

Keycloak

Istio（ambient 模式）

下一步

启用安全 Profile

前提条件

1. 在 operand 上启用安全 profile

2. 将命名空间纳入安全 profile

3. 部署受保护的 agent

4. 验证

硬件配置

介绍

为什么需要 Hardware Profile？

硬件配置文件管理

创建硬件配置文件

更新硬件配置文件

使用 CLI 创建

前提条件

创建 HardwareP

rofile

将工作负载调度到特定的 GPU 节点

使用节点选择器

使用污点和容忍

组合配置

创建仅 CPU

示例 1：仅 CPU

示例 2：GPU 加

介绍

硬件配置文件使平台管理员能够集中为特定场景预配标准化的硬件配置。这些配置将计算资源限制、节点选择器和节点容忍度紧密封装为一个统一的整体，平台用户在部署不同的模型推理服务时可以轻松选择。

使用硬件配置文件可显著减少通过 YAML 直接配置所导致的人工错误，防止意外调度到错误的拓扑组，并全面确保集群工作负载的稳健资源管理。

硬件配置文件可原生支持并与平台的 `InferenceService` 和 `LLMInferenceService` 资源进行深度交互。

目录

为什么需要 Hardware Profile？

Hardware Profile 的关键方面

为什么需要 **Hardware Profile**？

虽然标准 Kubernetes 通过 Pod 规范提供了资源请求和限制，但构建和部署 AI 推理工作负载（例如 Large Language Models 或特定的 KServe predictor）会带来独特的运维挑战。我们的 Hardware Profiles 实现专门针对这些挑战进行了设计，并具备以下平台特定特性：

1. 拓扑与专用加速器抽象
   数据科学家更关注模型性能和业务逻辑，而不是底层集群拓扑。他们可能并不清楚将工作负

载调度到特定 GPU 节点、vGPU 资源或互连网络所需的准确节点标签或污点。Hardware Profile 将这些技术复杂性抽象掉。管理员可以将精确的 `Node Selectors` 和 `Tolerations` 直接嵌入配置文件中，确保当用户从 UI 中选择 “High-End NVIDIA A100” 配置文件时，工作负载会自动定位到正确的物理机器池。

2. 动态边界化自定义（不只是固定配额）
   与严格执行单一、不可变资源规格（t-shirt sizing）的平台不同，我们的系统为每种资源类型定义了一个动态、可扩展的边界。管理员会配置 最小允许值、默认值 和 最大允许值。当用户选择某个配置文件时，会立即继承 *Default* 设置。不过，通过 **Customize Data** 选项，他们仍然可以灵活地手动微调各自的 Requests 和 Limits。只要这些值落在授权的配置文件边界内，就能够成功通过——从而为不同模型提供弹性，同时又不会带来过度占用集群的风险。

3. 智能 **Webhook** 验证与非对称自动修正
   我们的平台使用专用的 Mutating Webhook，与模型服务流水线深度集成。它不会依赖用户完美地编写 YAML 清单，而是会优雅地拦截请求，并安全地将配置文件的约束注入到工作负载运行时中。此外，它还能智能地保护集群——例如，如果用户指定了 limits 但遗漏了 requests（或反之），Webhook 会原生执行智能语义调整（将 requests 上限裁剪到 limits，或提升默认值），并在任何 Pod 被创建之前，全面阻止违反配置文件定义的最小值或最大值限制的配置。

4. 与自定义 **Serving Engine** 的原生互操作性
   无论是部署标准的 `InferenceService` 还是高度定制的 `LLMInferenceService`，硬件配置文件引擎都会在后台原生跟踪复杂的 Pod/Container 结构，并将配置精确注入到当前活动的预测容器资源中。

Hardware Profile 的关键方面

- 资源标识符（**Limits** 与 **Requests**）： 配置文件会安全地管理原生 Kubernetes 限制（例如最小 CPU 阈值、默认可用内存分配以及严格的最大 GPU 加速限制），以防止系统过载，同时保持运行稳定性。
- **Taints** 与 **Tolerations**： Hardware profiles 会天然地精确指示工作负载 Pod 能够容忍哪些节点（例如容忍专用异构硬件污点）。
- **Node Selectors**： 它们会严格约束工作负载只能调度到特定的节点标签选择器，以匹配正确的机器架构，而不会进行任何隐式猜测。
- 后端 **Webhook** 注入： 通过安装在集群中的自动拦截机制，硬件约束会直接从管理命名空间透明地合并并附加到已提交的工作负载中。

硬件配置文件管理

为了为您的数据科学家和工程师配置特定的硬件配置和约束，以便他们在平台上部署模型推理服务时使用，您必须创建并管理关联的硬件配置文件。硬件配置文件将节点亲和性、容忍度和资源约束封装为一个可复用的实体。

目录

- 创建硬件配置文件
  - 步骤 1：导航到硬件配置文件
  - 步骤 2：开始创建硬件配置文件
  - 步骤 3：配置基本详情
  - 步骤 4：配置资源标识符（requests 和 limits）
  - 步骤 5：配置节点调度规则
  - 步骤 6：完成创建
- 更新硬件配置文件
  - 步骤 1：定位硬件配置文件
  - 步骤 2：编辑硬件配置文件
  - 步骤 3：修改配置
  - 步骤 4：应用更改
- 删除硬件配置文件
  - 步骤 1：定位硬件配置文件
  - 步骤 2：删除
  - 步骤 3：确认删除
- 将硬件配置文件用于推理服务

步骤 1：启动部署表单

步骤 2：选择硬件配置文件

步骤 3：查看并自定义资源分配

步骤 4：部署

创建硬件配置文件

前提条件

- 您已使用具有管理员权限的用户登录平台。
- 您已确认底层 Kubernetes 集群中可用的目标计算资源，包括 CPU、内存以及任何专用加速器（例如 GPU 型号）。
- 您熟悉 Kubernetes 调度概念，例如 Node Selectors、Taints 和 Tolerations。

操作步骤

1 步骤 1：导航到硬件配置文件

在主导航菜单中，进入 **Hardware Profile**。将打开 Hardware Profiles 页面，显示系统中现有的硬件配置文件。

2 步骤 2：开始创建硬件配置文件

在右上角单击 **Create hardware profile**。将打开 Create hardware profile 配置页面。

3 步骤 3：配置基本详情

在 Basic Details 部分，为该配置文件提供标识信息：

- **Name**：为硬件配置文件输入一个唯一且具有描述性的名称（例如，`gpu-high-performance-profile`）。
- **Description**：（可选）输入清晰的硬件配置文件描述，以帮助其他用户理解其预期用途。

4 步骤 4：配置资源标识符（requests 和 limits）

您可以为计算资源定义约束，例如 CPU、内存或特定加速器（例如 `nvidia.com/gpu`）。单击 **Add Identifier** 或修改已存在的资源字段。您可以添加两类标识符：

- **Built-in Identifiers**：从平台配置的标准资源类型下拉列表中选择（例如 `cpu`、`memory`、`nvidia.com/gpu`）。对于这些内置类型，**Identifier**、**Display Name** 和 **Resource Type** 由平台严格预定义，且不可更改。
- **Custom Identifiers**：输入您自己的唯一资源参数。您必须手动定义：
  - **Identifier**：精确的 Kubernetes 资源键（例如 `nvidia.com/a100` 或某个自定义厂商 ASIC）。
  - **Display Name**：将在 UI 中显示的、便于阅读的资源名称（例如 `NVIDIA A100 GPU`）。
  - **Resource Type**：为集群准确地对资源分类：
    - **`CPU`** / **`Memory`**：选择此项以定义标准计算边界。
    - **`Accelerator`**：主要用于任何专用 AI 芯片（例如 NVIDIA GPU、AMD GPU 或 Intel Gaudi 加速器），这些芯片用于模型训练或高负载推理任务。将类型设置为 Accelerator 后，平台会明确将该依赖识别为核心 AI 计算引擎。
    - **`Other`**：用于附加到节点上的非 AI 辅助设备（例如用于 RDMA 的高速网络接口、infiniband 或独特的存储参数）。

对于内置标识符和自定义标识符，您都必须配置精确的分配边界：

- **Default**：设置要分配的该资源的基础数量。当用户选择该配置文件时，这一值会最初注入到其工作负载中。
- **Minimum allowed**：定义可接受的最小请求量。它作为硬性下限，用于防止用户为关键模型请求不足的资源。
- **Maximum allowed**：（可选）指定绝对最大限制。这将严格防止用户预留超出已定义容量阈值的过多集群资源。

5 步骤 5：配置节点调度规则

为了严格控制推理工作负载可以调度到哪些节点，请设置 Node Selectors 和 Tolerations。这可确保高性能工作负载落到物理上正确的节点池中。

- **Node Selectors**：在 Node Selectors 部分，单击 **Add Node Selector**。输入 **Key** 和 **Value** 约束。平台会自动注入这些键值对，以将工作负载限制到仅匹配标签的节点上。
- **Tolerations**：在 Tolerations 部分，单击 **Add Toleration**，以显式允许工作负载调度到具有匹配 taint 的节点上。定义 **Key**、**Operator**（例如 `Equal`、`Exists`）、**Value**、**Effect**（例如 `NoSchedule`、`NoExecute`）以及可选的 **Toleration Seconds**。与原生 Kubernetes tolerations 一样，您可以为单个硬件配置文件添加多个 toleration。

6 步骤 6：完成创建

检查您输入的配置，确保准确无误。单击 **Create**，完成硬件配置文件的创建。

更新硬件配置文件

您可以更新部署中的现有硬件配置文件，以适应新的基础设施变化、硬件升级或逐步修订的资源策略。您可以可靠地更改重要的标识信息、最小和最大资源约束，或者通过节点选择器和容忍度调整集群节点放置。

1 步骤 1：定位硬件配置文件

在导航菜单中，单击 **Hardware Profile**。从列表中找到要更新的硬件配置文件。

2 步骤 2：编辑硬件配置文件

在相关硬件配置文件所在行的右侧，单击操作菜单（⋮），然后选择 **Update**。

3 步骤 3：修改配置

对硬件配置文件配置进行必要的修改：

- 安全地调整 **Description**。

- 更新特定资源标识符的 **Default**、**Minimum** 或 **Maximum allowed** 阈值，使其与当前集群容量严格匹配。
- 修改 **Node Selectors** 以匹配不同的节点标签，或更新 **Tolerations** 以适配新设置 taint 的工作节点。

4 步骤 4：应用更改

单击 **Update**，永久应用您的更改。

注意：更新硬件配置文件通常只会影响后续新配置的工作负载。之前使用此硬件配置文件创建的正在运行的部署将严格保留最初注入的约束。若要将新的硬件配置文件设置强制应用到已运行的工作负载上，您必须显式编辑或重新部署对应的推理服务。

删除硬件配置文件

当某个特定硬件配置已过时或涉及已废弃的 Kubernetes 节点时，您可以安全地删除其硬件配置文件。这样可确保未来没有数据科学家错误地选择过时的节点配置或无法管理的限制。

1 步骤 1：定位硬件配置文件

在主导航菜单中，单击 **Hardware Profile**。找到要删除的硬件配置文件。

2 步骤 2：删除

单击相关硬件配置文件行最右侧的操作菜单（⋮），然后谨慎选择 **Delete**。

3 步骤 3：确认删除

将出现一个警告对话框，要求您确认删除操作。单击 **Delete**。

注意：删除硬件配置文件不会删除或主动中断之前使用该配置文件部署的正在运行的推理服务。它们将继续在平台 webhook 最初注入的资源限制和拓扑约束下正常运行。不过，已删除的硬件配置文件将立即从所有新建部署的配置文件选择下拉列表中消失。

将硬件配置文件用于推理服务

当用户（例如数据科学家、AI 工程师和开发者）动态创建或配置模型推理服务（包括 `InferenceService` 和 `LLMInferenceService`）时，他们可以高效利用预定义的硬件配置文件。

硬件配置文件可无缝简化手动配置复杂节点调度规则和设置显式资源限制的繁琐任务。根据您的工作负载具体需求，您既可以采用严格的默认配置，也可以在所选配置文件正式授权的边界内精细自定义限制。

1 步骤 1：启动部署表单

在导航菜单中，进入 **Service Manage**。单击 **Create**，启动一个全新模型推理服务的部署表单。

2 步骤 2：选择硬件配置文件

在部署表单中，向下滚动并进入 **Deployment Resources** 部分。在这里，您可以先选择一个 **Config Type** 来定义资源限制：

- 默认设置为 **Hardware Profile**。随后，您可以单击 **Profile** 下拉菜单，选择一个当前由平台管理员为您目标计算环境启用的特定硬件配置文件。
- 或者，如果您希望绕过预定义配置文件并手动提供原始 Kubernetes 资源限制，也可以选择 **Custom**。

3 步骤 3：查看并自定义资源分配

选定硬件配置文件后，表单会安全地锁定管理员整理的对应基础定义。不过，您仍然可以优化精确的资源限制：

- 要查看管理员指定的边界，请单击配置文件下拉菜单旁的 **View Detail** 按钮。这将打开一个信息抽屉或模态框，明确展示硬件配置文件的详细信息，包括已配置的节点规则以及 CPU、Memory 和 GPUs 的绝对限制。
- 根据您的具体工作负载需求，单击硬件配置文件部分下方动态显示的 **Custom Configuration** 按钮。自定义 requests 和 limits 必须在概念上严格保持在硬件配置文件定义的最小和最大约束范围内。

- 通过执行此自定义操作，您将能够直接修改推理服务最终的 **Requests** 和 **Limits** 配置。如果您提交了无效的 request 参数，校验引擎会优雅地捕获该偏差并向您显示验证错误。

4 步骤 **4**：部署

填写服务的其余参数，然后单击 **Deploy**。

使用 CLI 创建 Hardware Profile

本文档介绍如何使用命令行创建 `HardwareProfile` 资源，并提供一个示例 YAML。

目录

前提条件

创建 HardwareProfile

检查 HardwareProfile 状态

前提条件

- 你可以访问已安装该平台的 Kubernetes 集群。
- 你已配置 `kubectl` 以便与集群通信。
- 你在管理 `HardwareProfile` 资源的命名空间中具有权限（例如，像 `kube-public` 这样的管理员命名空间）。

创建 HardwareProfile

创建一个名为 `gpu-high-performance-profile.yaml` 的 YAML 文件，内容如下：

```yaml
apiVersion: infrastructure.opendatahub.io/v1alpha1
kind: HardwareProfile
metadata:
  name: gpu-high-performance-profile
  namespace: kube-public
spec:
  # Define resource limitations and defaults
  identifiers:
    - identifier: "nvidia.com/gpu"
      displayName: "GPU"
      minCount: "1"
      maxCount: "8"
      defaultCount: "1"
      resourceType: Accelerator
    - identifier: "cpu"
      displayName: "CPU"
      minCount: "4"
      maxCount: "32"
      defaultCount: "8"
      resourceType: CPU
    - identifier: "memory"
      displayName: "Memory"
      minCount: "16Gi"
      maxCount: "128Gi"
      defaultCount: "32Gi"
      resourceType: Memory
  # Configure Node Selectors and Tolerations for scheduling
  scheduling:
    type: Node
    node:
      nodeSelector:
        accelerator: nvidia-a100
        node-role.kubernetes.io/worker: "true"
      tolerations:
        - key: "nvidia.com/gpu"
          operator: "Exists"
          effect: "NoSchedule"
```

然后使用 `kubectl` 将该 YAML 文件应用到集群：

```bash
kubectl apply -f gpu-high-performance-profile.yaml -n kube-public
```

检查 **HardwareProfile** 状态

你可以使用以下命令检查 `HardwareProfile` 是否已成功创建：

```
kubectl get hardwareprofile gpu-high-performance-profile -n kube-public
```

输出应类似于以下内容：

```
NAME                           AGE
gpu-high-performance-profile   2m
```

成功应用后，数据科学家在通过 UI 部署其 Inference Services 时，就可以选择 **GPU High Performance**，并且配置文件中指定的约束会自动验证并注入到已部署的工作负载中。

将工作负载调度到特定的 GPU 节点

在定义 Hardware Profile 时，通常需要确保 AI 推理工作负载被严格调度到具有特定类型 GPU（例如 NVIDIA A100 或 H100）的节点上，并且该工作负载能够容忍这些专用节点上的污点，以避免常规 CPU 工作负载占用 GPU 节点。

本指南演示如何在 Hardware Profile 中配置这些约束，以便你的数据科学家无需手动配置它们。

目录

使用节点选择器

使用污点和容忍

组合配置

使用节点选择器

节点选择器允许你基于节点标签将 Pod 引导到特定节点。

1. 找到集群中 GPU 节点的准确 Kubernetes 标签。例如：
   - `accelerator: nvidia-a100`
   - `nvidia.com/gpu.present: "true"`
2. 编辑或创建你的 Hardware Profile。

3. 在 **Node Selectors** 部分，添加与该标签对应的键值对：

- **Key**: `accelerator`
- **Value**: `nvidia-a100`

保存后，任何尝试使用此 Hardware Profile 的推理服务都会自动获得该节点选择器，确保它只会被调度到具有 A100 GPU 的节点上。

使用污点和容忍

集群管理员经常会对 GPU 节点设置“污点”，以便标准 Pod（例如 Web 服务器或通用数据库）不会被调度到这些节点上，从而将 GPU 计算能力保留给 AI 工作负载。

如果你的 GPU 节点带有类似 `nvidia.com/gpu:NoSchedule` 的污点，那么你的 Hardware Profile 必须包含对应的容忍。

1. 在 Hardware Profile 的 **Tolerations** 部分添加一个新的容忍。
2. 将其配置为匹配 GPU 节点上的污点：

- **Key**: `nvidia.com/gpu`
- **Operator**: `Exists`（这表示容忍 `nvidia.com/gpu` 键的任意值。或者，也可以使用 `Equal` 并显式设置 **Value**。）
- **Effect**: `NoSchedule`（匹配该污点的限制性效果）

通过向 Hardware Profile 添加此容忍，部署的推理服务将被明确授予调度到专用 GPU 节点上的“权限”。

组合配置

通过将 **Node Selector**（指示调度器要去哪里）和 **Toleration**（允许调度器将其放置在那里）结合起来，你的 Hardware Profile 实际上就成为异构节点架构的可靠蓝图。

创建仅 CPU 和 GPU 加速的 Profile

在生产级 AI 平台中，您通常需要为不同类型的机器学习工作负载提供服务。例如，传统机器学习模型（如 scikit-learn 或 XGBoost）或简单的数据处理任务只需要 CPU 资源，而大语言模型（LLMs）或复杂的深度学习模型则需要 GPU 加速。

通过为仅 CPU 和 GPU 加速工作负载创建不同的 Hardware Profile，您可以有效隔离这两类服务，并防止轻量级 CPU 模型无意中消耗昂贵的 GPU 资源。

目录

示例 1：仅 CPU 的 Hardware Profile

示例 2：GPU 加速的 Hardware Profile

示例 1：仅 CPU 的 Hardware Profile

仅 CPU 的 profile 不包含任何加速器标识符（例如 `nvidia.com/gpu`），并且严格依赖 `cpu` 和 `memory` 标识符。

创建仅 CPU 的 profile 时，请确保：

1. 完全排除 **Accelerator** 资源类型。
2. Node Selector 不要选择任何特定于 GPU 的节点。
3. 名称和描述清楚表明此 profile 面向标准 ML 推理或轻量级模型。

以下是仅 CPU 的 hardware profile 示例：

```yaml
apiVersion: infrastructure.opendatahub.io/v1alpha1
kind: HardwareProfile
metadata:
  name: standard-cpu-profile
  namespace: kube-public
spec:
  # Do not include nvidia.com/gpu
  identifiers:
    - identifier: "cpu"
      displayName: "CPU"
      minCount: "1"
      maxCount: "8"
      defaultCount: "2"
      resourceType: CPU
    - identifier: "memory"
      displayName: "Memory"
      minCount: "2Gi"
      maxCount: "16Gi"
      defaultCount: "4Gi"
      resourceType: Memory
  # Standard CPU nodes
  scheduling:
    type: Node
    node:
      nodeSelector:
        node-role.kubernetes.io/worker: "true"
```

示例 2：GPU 加速的 Hardware Profile

GPU 加速的 profile 会显式要求 `nvidia.com/gpu` 标识符，从而确保任何选择此 profile 的工作负载都会分配到物理 GPU 资源。

创建 GPU 加速的 profile 时：

1. 包含特定加速器的标识符（例如 `nvidia.com/gpu`）。
2. 如果您的 GPU 节点带有污点，请添加相应的 Tolerations（例如 `nvidia.com/gpu:NoSchedule`）。

3. 可选地添加 Node Selector，以选择特定的 GPU 架构（例如 `accelerator: nvidia-t4`）。

以下是 GPU 加速的 hardware profile 示例：

```yaml
apiVersion: infrastructure.opendatahub.io/v1alpha1
kind: HardwareProfile
metadata:
  name: gpu-t4-profile
  namespace: kube-public
spec:
  identifiers:
    # Crucially include the GPU resource
    - identifier: "nvidia.com/gpu"
      displayName: "GPU"
      minCount: "1"
      maxCount: "4"
      defaultCount: "1"
      resourceType: Accelerator
    - identifier: "cpu"
      displayName: "CPU"
      minCount: "4"
      maxCount: "16"
      defaultCount: "8"
      resourceType: CPU
    - identifier: "memory"
      displayName: "Memory"
      minCount: "16Gi"
      maxCount: "64Gi"
      defaultCount: "32Gi"
      resourceType: Memory
  scheduling:
    type: Node
    node:
      nodeSelector:
        accelerator: nvidia-t4
      tolerations:
        - key: "nvidia.com/gpu"
          operator: "Exists"
          effect: "NoSchedule"
```

通过提供这两种明显不同的 profile，平台管理员可以确保 Data Scientist 获得其所需的准确环境，而不会将高价值的计算资源浪费在简单任务上。

多租户

Namespace 管理

MLflow 工作区与访问控制

将 namespace 暴露为工作区

授予用户访问权限

从客户端选择工作区

故障排查

Namespace 管理

Alauda AI 支持的租户是 Kubernetes 命名空间，在作为 Alauda AI 的租户使用之前，需要先在 **Namespace Management** 中进行管理。

创建命名空间并向该命名空间添加用户的操作步骤如下：

1. 以管理员身份登录 Web Console
2. 创建命名空间
    1. 切换到 **Projects** 视图
    2. 创建或选择一个现有项目，然后单击进入当前项目详情页。
    3. 单击 **Namespace** > **Namespace**
    4. 单击 **Create Namespace**
    5. 在 **Create Namespace** 页面中，在 **Pod Security Policies** 部分下，针对 `enforce`、`audit` 和 `warn` 这三种安全模式，均选择 **privileged** 作为安全标准。

        > **WARNING**
        >
        > 安全标准应为空白或 **privileged**，否则 AI Inference 工作负载可能拒绝运行。

3. 为用户分配命名空间权限
    1. 切换到 **Administrator** 视图
    2. 在导航中选择 **User Role Management** > **User Management**
    3. 创建一个新用户，或选择要添加到此命名空间的用户
    4. 单击 **Configure Role** 选项卡，然后单击 **Add Role** 按钮
    5. 为用户配置与 Alauda AI 相关的角色，并选择之前创建的命名空间

- Editor：用户可以在该命名空间中创建和删除工作负载，并使用 CPU 和 GPU 等资源配额完成工作
- Owner：用户有权限管理此 Alauda AI 命名空间
- Viewer：用户只能查看该命名空间的资源，不能创建或删除工作负载

4. 返回 Alauda AI，在 **Admin** > **Management Namespace** 中单击 **Namespace Manage**，然后选择之前创建的命名空间，完成管理

MLflow 工作区与访问控制

在 Alauda AI 上，MLflow 是多租户的：每个 工作区 都对应一个 Kubernetes **namespace**，对工作区中的 experiments、runs、datasets 和已注册模型的访问，均通过 Kubernetes RBAC 授权。本指南将展示如何将一个 namespace 暴露为工作区，并授予用户对其的访问权限。

关于底层模型，请参见 简介 → 多租户模型。

目录

将 namespace 暴露为工作区

授予用户访问权限

从客户端选择工作区

故障排查

将 **namespace** 暴露为工作区

当一个 namespace 带有 MLflow 配置所选定的标签时（默认是 `mlflow-enabled=true`），它就会成为一个 MLflow 工作区。只有匹配的 namespace 才会显示为工作区。

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: team-a
  labels:
    mlflow-enabled: "true"
  annotations:
    mlflow.kubeflow.org/workspace-description: "Team A MLflow workspace"
```

应用该标签，或者为现有 namespace 添加标签：

```
kubectl label namespace team-a mlflow-enabled=true
```

> **INFO**
>
> 跟踪服务器的默认工作区 namespace 必须在服务器首次启动之前就已存在并完成标记，否则服务器将无法启动（参见 安装 → 前提条件）。

授予用户访问权限

MLflow 会使用 `mlflow.kubeflow.org` API group，根据调用者在目标 namespace 中的 Kubernetes 权限对每个请求进行授权。请在工作区 namespace 中使用 `Role` 和 `RoleBinding` 授予访问权限：

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: mlflow-manager
  namespace: team-a
rules:
  - apiGroups: ["mlflow.kubeflow.org"]
    resources: ["experiments", "datasets", "registeredmodels"]
    verbs: ["get", "list", "create", "update", "delete"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: mlflow-manager
  namespace: team-a
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: mlflow-manager
subjects:
  - kind: Group
    name: team-a-mlflow-users
    apiGroup: rbac.authorization.k8s.io
```

绑定中的 group 或 user name 必须与 OAuth proxy 转发的令牌中的 identity claim 匹配（即平台的 OIDC groups/user）。

从客户端选择工作区

将 MLflow tracking URI 设置为集群内的 Service（或平台 route），然后选择工作区：

```
import mlflow

mlflow.set_tracking_uri("http://mlflow-tracking-server.kubeflow:5000")  # in-cluster Service, via the OAuth proxy
mlflow.set_workspace("team-a")

with mlflow.start_run():
    mlflow.log_param("tenant", "team-a")
```

对于原始 HTTP 客户端，请传递工作区 header：

```
curl \
  -H "X-MLFLOW-WORKSPACE: team-a" \
  "https://<platform>/clusters/<cluster-name>/mlflow/api/2.0/mlflow/experiments/search"
```

你只能使用账户有权访问的工作区。有关完整的身份验证细节——无需浏览器获取 identity token 并连接 SDK——请参见 在身份验证和 RBAC 下使用 MLflow Python SDK。

故障排查

- 工作区不可见。 验证其 namespace 是否与配置的 `workspaceLabelSelector` 匹配（默认是 `mlflow-enabled=true`）。
- **`403 PERMISSION_DENIED`**。 该账户没有该工作区 namespace 的访问权限。请为该 namespace 中的用户或 group 添加 `RoleBinding`。
- 某个 **run** 显示了错误的 **owner** 或 **workspace**。 owner 是经过身份验证的 identity；workspace 是 `set_workspace()` / `MLFLOW_WORKSPACE` 所选择的工作区（否则为服务器默认值）。请同时检查两者。

安全配置文件

安装安全配置文件的依赖项，并在集群上启用它。

安装 **Secure Profile** 依赖项

通用安装模式

SPIRE

Keycloak

Istio（ambient 模式）

下一步

启用安全 **Profile**

前提条件

1. 在 operand 上启用安全 profile

2. 将命名空间纳入安全 profile

3. 部署受保护的 agent

4. 验证

安装 Secure Profile 依赖项

secure profile 需要在目标集群上安装 **cert-manager**、**SPIRE**、**Keycloak** 和 **Istio**（**ambient** 模式）。cert-manager 随平台一起提供；本指南将安装其余三个组件。请在启用 secure profile 之前完成安装。

目录

通用安装模式

SPIRE

Keycloak

1. 安装 operator
2. 部署 Keycloak 实例

何时创建 `KeycloakRealmImport`

Istio（ambient 模式）

下一步

通用安装模式

每个组件都以平台包的形式交付——即 OLM operator 或 Cluster Plugin。有关通用的软件包下载、`violet` 配置和上传操作步骤，请参见 上架软件包。所有组件的模式都相同：

1. 从 灵雀云 Customer Portal 下载组件包。

2. 按照 上架软件包 将其上传到平台。`violet` 会将软件包中的镜像重新定位到平台镜像仓库，因此这些组件可以安装到 **air-gapped** 集群。
3. 从 Web 控制台进行安装——进入 **Administrator** → **Marketplace / OperatorHub**，选择目标集群——或者使用 `kubectl`（OLM operator 使用 `Subscription`，Cluster Plugin 使用 `ModuleInfo`）。

以下各节介绍每个组件的单独配置。

SPIRE

SPIRE 会签发 AuthBridge 用于 mTLS 的工作负载 SVID。它以 **Cluster Plugin** 的形式交付。

按照 上架软件包 上传 SPIRE 软件包，并将其安装到 **workload cluster**（在 Marketplace 中安装到集群，或者在 global cluster 中创建 `ModuleInfo`）。安装时请进行如下配置：

设置	说明
`trustDomain`	SPIFFE trust domain，例如 `kagenti.local`。请记录此值——它必须与 `Kagenti` operand 上的 `defaults.spiffe.trustDomain` 保持一致。
`clusterName`	此集群的名称。
`persistence.storageClass`	SPIRE server datastore 使用的真实 `StorageClass`（请替换默认占位符）。
`caSubject`	SPIRE trust bundle 的 CA subject（国家 / 组织 / common name）。

验证：

```
# server, agent, and CSI driver running
kubectl get pods -n spire-server
kubectl get pods -n spire-system

# the trust bundle ConfigMap (key bundle.spiffe) and the ClusterSPIFFEID
CRD
kubectl get configmap spire-bundle -n spire-system
kubectl get crd clusterspiffeids.spire.spiffe.io
```

SPIRE 会通过 `csi.spiffe.io` CSI driver 自动为每个 pod 注册（通过默认 `ClusterSPIFFEID`），格式为 `spiffe://<trustDomain>/ns/<namespace>/sa/<serviceaccount>`。

Keycloak

Keycloak 会签发 AuthBridge 验证的 OAuth2/JWT token。它以 **OLM operator** 加上您创建的 `Keycloak` 实例形式交付。

1. 安装 operator

按照 上架软件包 上传 Keycloak operator 软件包，然后从 **OperatorHub** 安装。平台 operator 需要 **`Manual`** 升级批准——请批准生成的 `InstallPlan` 以完成安装。

2. 部署 Keycloak 实例

创建一个 `Keycloak` 自定义资源。请将其命名为 **`keycloak`**，这样 operator 才会生成 Kagenti operator 会读取的 `keycloak-initial-admin` secret：

```yaml
apiVersion: k8s.keycloak.org/v2beta1
kind: Keycloak
metadata:
  name: keycloak
  namespace: keycloak
spec:
  instances: 1
  image: <relocated-keycloak-image> (1)
  startOptimized: false
  db:
    vendor: dev-file (2)
  http:
    httpEnabled: true
  hostname:
    strict: false (3)
  features:
    enabled:
      - token-exchange (4)
      - admin-fine-grained-authz
```

(1) `image` —— 由 `violet` 重新定位后的 Keycloak server 镜像（集群无法直接拉取 `quay.io`）。

(2) `db.vendor: dev-file` 会在 pod 内运行 H2 数据库——适合 dev/test（重启后数据不会持久化）。在 production 中，请设置 `vendor: postgres`，并将 `db.host` / `db.usernameSecret` / `db.passwordSecret` 指向外部 PostgreSQL。

(3) `hostname.strict: false` 允许 Keycloak 从请求中解析其 hostname，这适合集群内访问。除非您同时设置了 `hostname.hostname`，否则不要将 `hostname.backchannelDynamic: true` 设为 `true`，否则 server 将无法启动。

(4) `token-exchange`（配合 `admin-fine-grained-authz`）是 Kagenti audience / token-exchange 流程所必需的。

> **WARNING**
>
> 不要预先创建 `keycloak-initial-admin` secret。Keycloak operator 会创建并拥有它；如果该名称的 secret 已存在，会阻止 operator 的 reconcile（`Keycloak` 实例将永远不会启动）。

operator 会在集群内将该实例暴露为 `http://keycloak-service.<namespace>.svc:8080` —— 请将此 URL 作为 `Kagenti` operand 上的 `keycloak.publicUrl`。

> **WARNING**
>
> 您设置为 `keycloak.publicUrl` 的值会成为 AuthBridge 用于验证的 JWT issuer（`iss`）claim。请使用与获取 **token** 时相同的 **hostname** 形式——通常是较短的集群内 service URL `http://keycloak-service.keycloak.svc:8080`（上游示例约定）。如果您选择 FQDN 形式（`…svc.cluster.local…`），那么获取 token 的客户端也必须使用该 FQDN；混用这两种形式会导致原本有效的 token 因 `iss` 不匹配而返回 `401`。

何时创建 `KeycloakRealmImport`

`KeycloakRealmImport` 是 Keycloak operator 声明式导入 realm 的方式（`spec.keycloakCRName` + `spec.realm`）。secure profile 需要一个带有平台 client 和 audience scope 的 `kagenti` realm。

- 通常无需手动创建。 当 `authbridgeConfig.enabled` 为 `true` 且已经存在 `Keycloak` 实例时，Kagenti operator 会自动导入 `kagenti` realm——它会向您的 Keycloak 应用名为 `kagenti-realm-import` 的 `KeycloakRealmImport`（包含平台 client 和 `kagenti-platform-audience` scope）。
- 仅当以下情况时才手动创建： 您不使用自动导入（例如 `authbridgeConfig` 关闭，但仍希望使用该 realm），或者需要自定义 realm。请应用一个引用您的实例（`spec.keycloakCRName: keycloak`）且在 `spec.realm` 下包含 realm 表示的 `KeycloakRealmImport`。

Istio（ambient 模式）

secure profile 使用 Istio **ambient** mesh（ztunnel）实现命名空间级 mTLS，作为 灵雀云 **Service Mesh v2**（`servicemesh-operator2` / Sail operator）交付。

从 Marketplace 安装 **Alauda Service Mesh v2** operator（下载 → 通过 上架软件包 上传 → 安装），版本需为 **2.1.1** 或更高（支持 ambient）。ACP 上的 ambient 还要求 **Multus** CNI plugin 和 `kube-ovn` ≥ v4.1.5。

然后通过创建 `IstioCNI`、`Istio` 和 `ZTunnel` 资源并将 `profile` 设为 `ambient` 来启用 ambient。请先创建 `istio-cni`、`istio-system` 和 `ztunnel` 命名空间，并分别为其添加 `istio-discovery=enabled` 标签。完整且权威的操作步骤（先决条件、命名空间标签、验证）请参见 **Alauda Service Mesh v2** 文档——*Installing Istio ambient mode*（`alauda/servicemesh2-docs` ↗）。简而言之：

```yaml
apiVersion: sailoperator.io/v1
kind: IstioCNI
metadata: { name: default }
spec:
  namespace: istio-cni
  profile: ambient
  values:
    cni:
      cniConfDir: /etc/cni/multus/net.d
      ambient: { reconcileIptablesOnStartup: true }
---
apiVersion: sailoperator.io/v1
kind: Istio
metadata: { name: default }
spec:
  namespace: istio-system
  profile: ambient
  values:
    pilot: { trustedZtunnelNamespace: ztunnel }
---
apiVersion: sailoperator.io/v1
kind: ZTunnel
metadata: { name: default }
spec:
  namespace: ztunnel   # must match pilot.trustedZtunnelNamespace
```

NOTE

如果集群中已经运行了共享 Istio（例如用于平台 dashboard），请以增量方式启用 ambient——在现有的 `Istio` 和 `IstioCNI` 资源上设置 `profile: ambient`（保留其现有的 `spec.values`，例如自定义 `meshConfig`），并添加一个 `ZTunnel`。现有的 sidecar 模式工作负载会继续运行，`istiod` 也会在原地更新。

Kagenti operator 会自动将每个已接入的 agent 命名空间标记为 `istio.io/dataplane-mode: ambient` 和 `istio-discovery: enabled`——您无需手动为 agent 命名空间添加这些标签。

下一步

在完成 cert-manager、SPIRE、Keycloak 和 Istio ambient 的安装后，请继续执行 启用 Secure Profile。

启用安全 **Profile**

Kagenti Operator 默认提供 **core profile**——具备 agent 注册和发现能力，但不包含身份 sidecar。**secure profile** 增加了零信任 workload identity（SPIRE mTLS）、OAuth2/JWT 认证（Keycloak）以及 Istio ambient mesh，并通过注入到每个已注册 workload 中的 **AuthBridge** sidecar 来强制执行这些能力。本指南将指导你启用它。关于架构和设计原因，请参见 Security Architecture。

目录

前提条件

1. 在 operand 上启用安全 profile
2. 将命名空间纳入安全 profile
3. 部署受保护的 agent
4. 验证

前提条件

secure profile 依赖于核心安装中不包含的集群组件。请按以下顺序先安装它们：

1. **cert-manager** — webhook 和 CA 证书。
2. **SPIRE** — 签发 workload SVID。提供 `csi.spiffe.io` CSI driver 和信任 bundle ConfigMap（`spire-bundle`）。请记下其**trust domain**（例如 `kagenti.local`）。

3. **Keycloak** — 一个可在集群内访问的实例，在其命名空间中包含 `keycloak-initial-admin` secret，并已启用 `token-exchange` 功能。
4. **Istio (ambient)** — `ztunnel` + `istio-cni`，用于命名空间级别的 mTLS。

> **INFO**
>
> 在 灵雀云 AI 上，这些组件对应为：cert-manager（内置）、**SPIRE** 集群插件、**Keycloak** operator 加上一个 `Keycloak` 实例，以及启用了 ambient 的 **Service Mesh v2**（`Istio` CR `profile: ambient` 再加上一个 `ZTunnel`）。关于各组件的逐步安装和配置，请参见 Install the Secure-Profile Dependencies。

1. 在 operand 上启用安全 profile

Patch `Kagenti` 自定义资源，以开启 sidecar 注入，并将其指向你的 Keycloak 和 SPIRE trust domain：

```yaml
apiVersion: kagenti.alauda.io/v1alpha1
kind: Kagenti
metadata:
  name: kagenti
  namespace: kagenti-system
spec:
  featureGates:
    globalEnabled: true (1)
    injectTools: true
  authbridgeConfig:
    enabled: true (2)
  keycloak:
    publicUrl: http://keycloak-service.keycloak.svc.cluster.local:8080 (3)
  defaults:
    spiffe:
      trustDomain: kagenti.local (4)
    images: (5)
      envoyProxy: <registry>/authbridge-envoy:<tag>
      authbridge: <registry>/authbridge:<tag>
      authbridgeLite: <registry>/authbridge-lite:<tag>
      proxyInit: <registry>/proxy-init:<tag>
```

1 `featureGates.globalEnabled` — AuthBridge sidecar 注入的总开关。`injectTools: true` 会将注入范围扩展到 `tool` workloads，从而也保护 agent↔tool 流量。

2 `authbridgeConfig.enabled` — 开启 operator 托管的 Keycloak client 注册、按命名空间生成的 `authbridge-config` ConfigMap（issuer / audience），以及 realm 引导。

3 `keycloak.publicUrl` — 集群内 Keycloak 的 URL；它会成为 AuthBridge 验证的 token **issuer** 和预期 **audience**。

4 `defaults.spiffe.trustDomain` — 必须与你的 SPIRE trust domain 匹配。

5 `defaults.images` — AuthBridge sidecar 镜像（其中已内置 `spiffe-helper`）。请将它们迁移到你的集群可拉取的镜像仓库；operator bundle 也会将它们记录在 `relatedImages` 中。

当 `authbridgeConfig.enabled` 为 `true`，并且在已配置的命名空间中已经存在 Keycloak 实例时，operator 会自动将 **`kagenti` realm** 导入其中（该 `KeycloakRealmImport` 包含平台 client 和 `kagenti-platform-audience` scope）——无需手动设置 realm。

2. 将命名空间纳入安全 profile

operator 仅管理那些明确加入的命名空间中的 identity。为每个运行受保护 **agent** 或 **tool** 的命名空间添加标签：

```
kubectl label namespace <agent-namespace> kagenti-enabled=true
```

> **WARNING**
>
> 此标签是必需的。如果没有 `kagenti-enabled=true`，operator 不会为该命名空间创建 `authbridge-config` ConfigMap，而其 workloads 的 Keycloak client 注册也不会启动（workloads 会因缺少 credentials secret 而保持 pending）。

3. 部署受保护的 agent

关于在安全 profile 下完整演示 weather-agent，请继续查看 Demo with Secure Profile。部署 YAML 与 core-profile quick start 相同——`Deployment` 或 `AgentRuntime` 中不需要任何

secure-profile 特定字段，因为 AuthBridge webhook 会在 admission 时基于 `Kagenti` operand 和命名空间标签完成所有接线。它会为每个已注册 pod 注入 `authbridge-proxy` sidecar 和 SPIRE SVID，而 operator 会为每个 workload 注册一个 Keycloak client。

如果你需要更严格的变体，让 tool 也运行 AuthBridge，并在 agent 与 tool 之间交换 token，请参见 Demo Advanced: AuthBridge on the Tool with Token Exchange。

> **Agent 镜像必须遵循**
>
> `$PORT` 在 proxy-sidecar 模式下，AuthBridge reverse-proxy 会绑定 workload 的服务端口，而 operator 会将应用重映射为 `PORT` 环境变量值（例如 `8001`）。agent 容器必须绑定 **`$PORT`**，而不是硬编码端口，否则会与 reverse-proxy 冲突（`address already in use`）。上游 weather 示例从 `v0.1.0` 起已遵循 `PORT`。

4. 验证

```
# AuthBridge sidecar injected (app container + authbridge-proxy)
kubectl get pod -n <ns> -l kagenti.io/type=agent \
  -o jsonpath='{.items[0].spec.containers[*].name}{"\n"}'

# SPIRE SVID delivered to the sidecar
kubectl exec -n <ns> <agent-pod> -c authbridge-proxy -- ls -l /opt/svid.p
em

# Keycloak client registered (credentials secret created + mounted)
kubectl get secret -n <ns> | grep kagenti-keycloak-client
```

现在 mesh 会强制执行身份认证。未认证请求会被拒绝：

```
{"error":"auth.unauthorized","message":"missing Authorization header","pl
ugin":"jwt-validation"}
```

成功的请求必须携带有效的 Keycloak access token，其 audience 必须与 realm issuer 匹配——使用 workload 已注册的 client（`<namespace>/<workload>`，secret 来自 `kagenti-`

`keycloak-client-credentials-*`），通过 `client_credentials` grant 获取该 token，并将其作为 `Authorization: Bearer <token>` 发送。

开发

本章面向数据科学家和原生应用开发者，涵盖 workbench 和 Kubeflow notebooks、连接、pipeline 编排、模型注册表、实验跟踪以及分布式计算。

本章所依赖的组件列在下方的 Components 部分。Agentic MLOps 内容还使用了 Deploy 章节中的 Envoy AI Gateway；关于从 Workbench 上传训练模型的内容，请参阅 Upload Models Using a Workbench；关于注册模型，请参阅 Model Registry。

工作台

介绍

- Workbench
- WorkspaceKind

创建工作台

- 前提条件
- 使用 Web 控制台创建工作台
- 连接到工作台
- 在 JupyterLab 中上传文件
- 可用的工作台镜像
- Docker Hub 镜像同步脚本指南
- 环境变量配置

Kubeflow Notebooks

使用 Kubeflow Notebooks

使用 Kubeflow 存储卷

使用 Kubefl

概念
创建 Notebook Server
连接到 Notebook
环境管理
管理配置（PodDefaults）
访问数据
最佳实践
创建存储卷
管理存储卷
在 Notebook 中使用存储卷
下一步
前提条件
使用 PyTorch 生
创建 TensorBoa
访问监控面板
使用场景
清理

连接

介绍
核心概念
支持的 Connection 类型
使用 Connections
前提条件
创建连接
更新连接
删除连接
管理连接类型
在模型部署中使用连接
资源模型

Pipelines

创建 Data Science Pipelines Ap
前提条件
1. 选择数据库和对象存储
2. 创建 DSPA
3. 验证 DSPA 是否就绪
使用 Kubeflow Pipelines
前提条件
快速开始示例
在 UI 中管理流水线
使用 Elyra 在
前提条件
验证 namespac
配置 Elyra 元数
可用的 Elyra Pi

4. 使用 pipeline 栈（KFP v2 SDK）

清理

使用可复用的 Kubeflow Pipeline 组件

前提条件

创建输入存储 Secret

运行托管 tabular pipeline

Kubeflow Pipelines 执行和存储行为

功能概览

安全地缓存任务

使用 `ParallelFor` 运行任务

持久化并传递类型化 artifact

运行验证 pipeline

在 arm64 集群上部署

平台兼容性和数据库后端

使用 Kueue

前提条件

操作步骤

分布式工作负载

CodeFlare SDK 教程

前提条件

演示 Notebook

操作步骤

运行 Spark Application

前提条件

1. 创建命名空间和 Spark RBAC

2. 提交 SparkApplication

3. 监控应用

4. 验证结果

5. 清理

调度周期性作业

常见的 SparkApplication 字段

使用 Kubeflow Model Registry

使用 Kubeflow Model Registry

访问 Model Registry

注册模型

部署已注册模型

实验跟踪

使用带认证和 RBAC 的 MLflow I

认证如何工作

前提条件

验证 token 方式（管理员）

从命令行获取 token（无需浏览器）

连接 SDK

选择 workspace

注册模型

替代方案：session cookie（无需平台更改

故障排查

Kubeflow Pipeline + MLflow 集成

适用范围

前提条件

组件如何访问 MLflow

完整示例：使用 MLflow 的训练 pipeline

上传并运行

在 Trainer v2 pipeline 中使用 MLflow

最佳实践

故障排查

使用 MLflow

前提条件

跟踪 LangChain

跟踪 OGX 应用

导出 OGX Open

查看 traces

故障排查

Agentic MLOps

将编码智能体与本地部署的推理服
简介
前提条件
各组件如何协同工作
步骤 1：部署并进行端点冒烟测试
步骤 2：在运行时启用 tool calling
步骤 2b（可选）：配置 reasoning models
步骤 3：连接你的编码智能体
最佳实践
故障排查
参考资料

使用编码智能体与本地部署 LLM 开展 MLOps
简介
设置智能体的工作环境
管理 InferenceServices 和 LLMInferenceServices
管理网关：身份验证和速率限制
调优服务性能以适配硬件
规划微调并生成报告
每日 MLOps 循环
最佳实践与防护措施
参考

组件

Kubeflow 数据科学流水线 **KubeRay**

Spark Operator **MLflow** **Label Studi**

Feast

工作台

介绍

Workbench

WorkspaceKind

创建工作台

前提条件

使用 Web 控制台创建工作台

连接到工作台

在 JupyterLab 中上传文件

可用的工作台镜像

Docker Hub 镜像同步脚本指南

环境变量配置

介绍

目录

Workbench

WorkspaceKind

Workbench

Workbench 管理 Jupyter Notebooks、code-server 或 RStudio 等 web IDE 实例。AI/ML 开发者和数据科学家可以使用集群资源（如 GPU）并连接到集群内服务，以创建通用的 ML 作业和流水线。

Workbench 作为 **OLM Helm operator** 安装。`Workbench` 自定义资源（`workbench.alauda.io/v1alpha1`）会部署并管理 `workspace-controller` 以及内置的 IDE 模板。

Workbench 使用 “Kubeflow Notebook 2.0” 作为后端来创建云端 IDE 容器。如果你熟悉 “Kubeflow Notebook”，那么开始使用 Workbench 会很容易。

一些主要功能包括：

- 原生支持 JupyterLab、Visual Studio Code（code-server）以及 RStudio（未来支持）。
- 用户可以直接在集群中创建 notebook 容器，而不是在本地工作站上创建。
- 管理员可以为其组织提供预装了所需包的标准 notebook 镜像。

- 访问控制由 Kubeflow 的 RBAC 管理，从而更便于在整个组织内共享 notebook。

WorkspaceKind

- 这是集群管理员创建的资源
- 它指定 Workspace 的模板（例如，“JupyterLab”、“VSCode”、“RStudio”）
- 最初，我们只会支持 “podTemplate” 类型，这与现有的 Notebook CRD 非常相似，但将来我们可能会支持其他类型的模板（例如，“helmTemplate”）

创建工作台

目录

前提条件

使用 Web 控制台创建工作台

操作步骤

登录，进入 **Alauda AI** 页面。

点击 **Workbench** 进入 **Workbench** 列表页面。

找到 **Create** 按钮，点击 **Create**，你将进入创建表单，填写信息后即可创建工作台。

连接到工作台

在 JupyterLab 中上传文件

离线安装 Python Wheel 文件

可用的工作台镜像

内置镜像

多架构镜像（`x86_64` 和 `arm64`）

其他镜像

`x86_64` 镜像

`arm64` 镜像

使用 Qwen3.5 notebook 验证 ModelSlim 镜像

Docker Hub 镜像同步脚本指南

脚本前提条件

环境变量配置

必填参数（目标私有仓库配置）

可选参数（源 DockerHub 配置）

示例 1：基本用法（最常见）

示例 2：单行命令执行（适用于 CI 环境）

示例 3：带 DockerHub 认证的完整执行（防止速率限制）

故障排查与注意事项

为 Ascend vNPU 工作台配置补充组

前提条件

- 确保已配置 `kubectl` 并连接到集群。
- 确保已创建 `PVC`。

> 创建 **PVC**
>
> 1. 登录，进入 **Alauda Container Platform** 页面。
> 2. 点击 **Storage** > **PersistentVolumeClaims** 进入 **PVC** 列表页面。
> 3. 找到 **Create PVC** 按钮，点击 **Create**，然后输入信息。

使用 **Web** 控制台创建工作台

操作步骤

1. 登录，进入 **Alauda AI** 页面。
2. 点击 **Workbench** 进入 **Workbench** 列表页面。
3. 找到 **Create** 按钮，点击 **Create**，你将进入创建表单，填写信息后即可创建工作台。

连接到工作台

创建工作台实例后，在左侧导航栏中点击 `Workbench`；你的工作台实例应显示在列表中。当状态变为 `Running` 时，点击 `Connect` 按钮进入工作台。

在 JupyterLab 中上传文件

如果你使用的是基于 JupyterLab 的工作台，可以通过文件浏览器中的 **Upload Files** 按钮从本地机器上传文件。当工作台无法访问公网或 PyPI mirror，而你需要从本地 wheel 文件安装 Python 包时，这会很有用。

离线安装 Python Wheel 文件

1. 连接到工作台并打开 JupyterLab。
2. 在左侧文件浏览器中，点击 **Upload Files** 按钮，并从本地机器选择一个或多个 `.whl` 文件。
3. 在 JupyterLab 中打开终端，并进入包含已上传文件的目录。
4. 安装该包：

```
pip install ./your_package-1.0.0-py3-none-any.whl
```

如果该包依赖其他 wheel 文件，请将所需的所有 `.whl` 文件上传到同一目录，并在不访问外部包索引的情况下安装它们：

```
pip install --no-index --find-links . your-package
```

> **INFO**
>
> 直接安装到容器中的包适合临时使用或个人使用。如果你重新创建工作台，仅安装在容器内部的包可能会丢失。对于可重复的环境，建议使用自定义工作台镜像，或使用存储在持久化存储上的虚拟环境。

可用的工作台镜像

该平台提供了一组可直接使用的 `WorkspaceKind` 镜像，这些镜像会直接出现在工作台创建表单中。此外，平台还发布了其他镜像到 Docker Hub，但默认不会同步到平台中。

下表采用与 Red Hat OpenShift AI 文档相同的通用风格：每个镜像都会说明其预期用途，并列出已预装的关键包，便于快速参考。包列表为代表性示例，并非穷尽。版本取自构建仓库中对应的镜像目录及其相应的 lock 文件。

内置镜像

以下镜像开箱即用：

多架构镜像（`x86_64` 和 `arm64`）

镜像名称	描述	主要包

Minimal Python
alauda-workbench-jupyter-minimal-cpu-py312-ubi9↗
	如果你需要轻量级 Jupyter 工作台，并计划自行安装项目特定的包，请使用此镜像。	`Python 3.12`
		`JupyterLab 4.5.6`
		`Jupyter Server 2.17.0`
		`JupyterLab Git 0.52.0`

镜像名称	描述	主要包
		`nbdime 4.0.4` `nbgitpuller 1.2.2`
Standard Data Science alauda-workbench-jupyter-datascience-cpu-py312-ubi9 ↗	适用于不需要特定框架 GPU 镜像的通用数据科学工作。	`Python 3.12` `JupyterLab 4.5.6` `Jupyter Server 2.17.0` `NumPy 2.4.3` `pandas 2.3.3` `SciPy 1.16.3` `scikit-learn 1.8.0` `Matplotlib 3.10.8` `Plotly 6.5.2` `KFP 2.15.2` `Kubeflow Training 1.9.3` `Feast 0.60.0` `CodeFlare SDK 0.35.0` `ODH Elyra 4.3.2`
code-server alauda-workbench-codeserver-datascience-cpu-py312-ubi9 ↗	如果你更喜欢类似 VS Code 的 IDE 进行数据科学开发，请使用此镜像。该镜像不提供基于 Elyra 的 pipeline。	`Python 3.12` `code-server 4.106.3` `Python extension 2026.0.0` `Jupyter extension 2025.9.1` `ipykernel 7.2.0` `debugpy 1.8.20` `NumPy 2.4.3` `pandas 2.3.3` `scikit-learn 1.8.0` `SciPy 1.16.3` `KFP 2.15.2`

镜像名称	描述	主要包
		`Feast 0.60.0` `virtualenv 21.1.0` `ripgrep 15.0.0`

其他镜像

以下镜像可在 Docker Hub 上获取，但默认不会构建到平台中：

`x86_64` 镜像

这些镜像面向支持 NVIDIA GPU 的 `x86_64` 节点。

镜像名称	描述	主要包
TensorFlow alaudadockerhub/odh-workbench-jupyter-tensorflow-cuda-py312-ubi9 ↗	适用于在 NVIDIA GPU 上进行 TensorFlow 模型开发和训练。	`Python 3.12` `CUDA base image 12.9` `TensorFlow 2.20.0+redhat` `TensorBoard 2.20.0` `JupyterLab 4.5.6` `Jupyter Server 2.17.0` `NumPy 2.4.3` `pandas 2.3.3`
PyTorch LLM Compressor alaudadockerhub/odh-workbench-jupyter-pytorch-llmcompressor-cuda-py312-ubi9 ↗	适用于在 NVIDIA GPU 上进行基于 PyTorch 的 LLM 压缩和优化。	`Python 3.12` `CUDA base image 12.9` `PyTorch 2.9.1` `torchvision 0.24.1` `TensorBoard 2.20.0` `llmcompressor 0.9.0.2`

镜像名称	描述	主要包
		`transformers 4.57.3` `datasets 4.4.1` `accelerate 1.12.0` `compressed-tensors 0.13.0` `nvidia-ml-py 13.590.44` `lm-eval 0.4.11`
PyTorch alaudadockerhub/odh-workbench-jupyter-pytorch-cuda-py312-ubi9 ↗	适用于在 NVIDIA GPU 上进行 PyTorch 模型开发和训练。	`Python 3.12` `CUDA base image 12.9` `PyTorch 2.9.1` `torchvision 0.24.1` `TensorBoard 2.20.0` `JupyterLab 4.5.6` `Jupyter Server 2.17.0` `onnxscript 0.6.2`
CUDA Minimal Python alaudadockerhub/odh-workbench-jupyter-minimal-cuda-py312-ubi9 ↗	如果你需要带有 NVIDIA CUDA 支持的轻量级 Jupyter 基础镜像，请使用此镜像。	`Python 3.12` `CUDA base image 13.0` `JupyterLab 4.5.6` `Jupyter Server 2.17.0` `JupyterLab Git 0.52.0` `nbdime 4.0.4` `nbgitpuller 1.2.2`

`arm64` 镜像

这些镜像面向支持 Ascend NPU 的 `arm64` 节点。

镜像名称	描述	主要包

CANN Minimal Python
alauda-workbench-jupyter-minimal-cann-py312-ubi9 ↗
	如果你需要带有 Ascend CANN 支持的轻量级 Jupyter 基础镜像，请使用此镜像。	`Python 3.12`
		`CANN 8.5.0`
		`JupyterLab 4.5.6`
		`Jupyter Server 2.17.0`
		`JupyterLab Git 0.51.4`
		`nbdime 4.0.4`
		`nbgitpuller 1.2.2`

PyTorch CANN
alauda-workbench-jupyter-pytorch-cann-py312-ubi9 ↗
	适用于在 Ascend NPU 上进行 PyTorch 模型开发和训练。	`Python 3.12`
		`CANN 8.5.0`
		`PyTorch 2.9.0`
		`torch_npu 2.9.0` (Ascend release `7.3.0`)
		`JupyterLab 4.5.6`
		`Jupyter Server 2.17.0`
		`TensorBoard 2.20.0`
		`Ray 2.54.0`
		`onnxscript 0.6.2`
		`NumPy 2.4.3`
		`pandas 2.3.3`
		`scikit-learn 1.8.0`
		`SciPy 1.16.3`
		`KFP 2.15.2`
		`Feast 0.60.0`

镜像名称	描述	主要包
MindSpore CANN docker.io/alaudadockerhub/alauda-workbench-jupyter-mindspore-cann-py312-ubi9:v0 .1.7	适用于在 Ascend NPU 上进行 MindSpore 模型开发、checkpoint 转换和训练。	`Python 3.12` `CANN 8.5.0` `MindSpore 2.8.0` `JupyterLab 4.5.6` `Jupyter Server 2.17.0` `TensorBoard 2.20.0` `ODH Elyra 4.3.2` `onnxscript 0.6.2` `KFP 2.15.2` `Kubeflow Training 1.9.3` `pandas 2.3.3` `scikit-learn 1.8.0` `SciPy 1.16.3`
ModelSlim CANN docker.io/alaudadockerhub/alauda-workbench-jupyter-modelslim-cann-py311-ubi9:v0 .1.7	适用于基于 `msmodelslim` 的 Ascend NPU 模型压缩和量化工作流，包括官方 `Qwen3.5` 验证路径。	`Python 3.11` `CANN 8.5.0` `PyTorch 2.9.0` `torch_npu 2.9.0` (Ascend release `7.3.0`) `msmodelslim 26.0.0a2` `transformers 5.2.0` `huggingface-hub 1.10.2` `torchvision 0.24.0` `mistral-common 1.11.0` `easydict 1.13` `wcmatch 10.1` `TensorBoard`

镜像名称	描述	主要包
		`2.20.0` `JupyterLab 4.5.6` `Jupyter Server 2.17.0`

要使用其他镜像，首先需要将其同步到你自己的镜像仓库中。你可以使用 `skopeo` 等工具完成此操作，或者使用下一节中描述的脚本。

使用 Qwen3.5 notebook 验证 ModelSlim 镜像

如果你使用 `docker.io/alaudadockerhub/alauda-workbench-jupyter-modelslim-cann-py311-ubi9:v0.1.7`，可以通过 下载 `qwen35_modelslim_quant_verify.ipynb` 来验证环境。

该 notebook 基于官方 Ascend `msmodelslim` `Qwen3.5` 示例，设计为预检验证 notebook，而不是完整的量化运行。默认情况下，它会：

- 检查已验证软件栈所需的运行时导入和固定版本的包，包括 `msmodelslim 26.0.0a2` 和 `transformers 5.2.0`
- 验证 `msmodelslim` CLI 以及模型目录和输出目录的权限要求
- 准备官方的 `msmodelslim quant --device npu ...` 命令，并且仅在 `RUN_QUANT = True` 时运行量化

在运行 notebook 之前，请将基础模型上传到工作台，并在第一个参数单元中检查 `MODEL_PATH`、`SAVE_PATH`、`MODEL_TYPE` 和 `QUANT_TYPE`。该镜像使用 `Python 3.11`，因为构建所使用的已验证公共 `msmodelslim` wheel 版本面向 CPython `3.11`。上游 `Qwen3.5` 指南目前将 Atlas A2 和 Atlas A3 训练与推理产品列为此量化流程支持的设备族。

Docker Hub 镜像同步脚本指南

sync-from-dockerhub.sh 是一个自动化工具，用于将选定的 Docker Hub 镜像，尤其是超大镜像，同步到私有镜像仓库，例如 Harbor。
由于网络波动，超大镜像在直接传输时更容易出现 OOM 或超时失败。为提高可靠性，该脚本

使用中转工作流：本地拉取 → 导出为 **tar** 归档 → 将 **tar** 归档推送到目标仓库。它还会在任务完成或异常退出时自动清理临时文件。

脚本前提条件

运行此脚本前，请确保执行机器上已安装并可访问以下工具：

- `bash`（执行环境）
- `nerdctl`（用于拉取镜像并将层导出为 tar 归档）
- `skopeo`（用于将 tar 镜像归档推送到目标私有仓库）

环境变量配置

脚本通过读取环境变量来执行同步，无需修改代码即可灵活使用。

必填参数（目标私有仓库配置）

环境变量	描述	示例值
`TARGET_REGISTRY`	目标私有镜像仓库地址	`build-harbor.alauda.cn`
`TARGET_PROJECT`	用于存放镜像的目标仓库中的具体项目/命名空间	`mlops/workbench-images`
`TARGET_USER`	登录目标仓库的用户名	`admin`
`TARGET_PASSWORD`	登录目标仓库的密码	`YourSecretPassword`

可选参数（源 **DockerHub** 配置）

为避免在拉取大量镜像时触发 DockerHub 的速率限制，你可以提供 DockerHub 凭据，在拉取前先登录。如果不需要，请留空。

环境变量	描述	示例值
`DOCKERHUB_USER`	DockerHub 账号用户名	`your_dockerhub_account`
`DOCKERHUB_PASSWORD`	DockerHub 密码或 Access Token	`dckr_pat_xxxxxx...`

示例 1：基本用法（最常见）

如果你只需要将脚本中定义的镜像同步到你的私有 Harbor：

```
# 1. Export environment variables for the target registry
export TARGET_REGISTRY="build-harbor.alauda.cn"
export TARGET_PROJECT="mlops/workbench-images"
export TARGET_USER="admin"
export TARGET_PASSWORD="YourHarborPassword"

# 2. Grant execution permissions to the script (if not already done)
chmod +x ./sync-from-dockerhub.sh

# 3. Execute the synchronization
./sync-from-dockerhub.sh
```

示例 2：单行命令执行（适用于 CI 环境）

你可以在同一行中声明环境变量并运行脚本。这种方式可以避免污染当前 Shell 环境变量：

```
TARGET_REGISTRY="build-harbor.alauda.cn" \
TARGET_PROJECT="mlops/workbench-images" \
TARGET_USER="admin" \
TARGET_PASSWORD="YourHarborPassword" \
./sync-from-dockerhub.sh
```

示例 3：带 DockerHub 认证的完整执行（防止速率限制）

当你从同一台机器频繁拉取镜像时，DockerHub 可能会拒绝你的请求。此时，请包含 DockerHub 凭据：

```
export TARGET_REGISTRY="build-harbor.alauda.cn"
export TARGET_PROJECT="mlops/workbench-images"
export TARGET_USER="admin"
export TARGET_PASSWORD="YourHarborPassword"

export DOCKERHUB_USER="alaudadockerhub"
export DOCKERHUB_PASSWORD="dckr_pat_xxx_your_token_xxx"

./sync-from-dockerhub.sh
```

故障排查与注意事项

1. 磁盘空间：由于脚本需要临时将超大镜像（例如 13GB）存储为 `tar` 归档，请确保系统的 `/tmp` 目录（或其所在的根分区）有充足的可用空间（建议至少 30GB）。脚本默认的暂存目录为 `/tmp/workbench-images-export-from-hub`。
2. 传输超时：当前脚本为推送大文件设置了 120 分钟超时（`SKOPEO_TIMEOUT="120m"`）。如果由于网络速度极慢导致过程失败，你可以使用任意文本编辑器在脚本顶部调整该参数值。
3. 修改镜像列表：如果你不再希望同步某些镜像，只需打开 `sync-from-dockerhub.sh`，并在 `WORKBENCH_IMAGES` 数组中用 `#` 注释掉对应行（类似于在 `sync.sh` 中过滤掉 minimal 镜像的方式）。

镜像在你的仓库中可用后，你还需要将相应配置添加到你计划使用的 **WorkspaceKind** 资源的 **`imageConfig`** 字段中。下面是一个示例 patch YAML，它会向现有 WorkspaceKind 添加一个新的镜像配置：

add-llmcompressor-image-patch.json

```
[
  {
    "op": "add",
    "path": "/spec/podTemplate/options/imageConfig/values/-",
    "value": {
      "id": "jupyter-pytorch-llmcompressor-cuda-py312",
      "spawner": {
        "displayName": "Jupyter | PyTorch LLM Compressor | CUDA | Python
3.12",
        "description": "JupyterLab with PyTorch and LLM Compressor for CU
DA",
        "labels": [
          {
            "key": "python_version",
            "value": "3.12"
          },
          {
            "key": "framework",
            "value": "pytorch"
          },
          {
            "key": "accelerator",
            "value": "cuda"
          }
        ]
      },
      "spec": {
        "image": "build-harbor.alauda.cn/mlops/workbench-images/odh-workb
ench-jupyter-pytorch-llmcompressor-cuda-py312-ubi9:3.4_ea1-v1.41",
        "imagePullPolicy": "IfNotPresent",
        "ports": [
          {
            "id": "jupyterlab",
            "displayName": "JupyterLab",
            "port": 8888,
            "protocol": "HTTP"
          }
        ]
      }
    }
  }
]
```

你可以使用类似下面的命令，将 patch 应用到你正在使用的 WorkspaceKind 上：

```
kubectl patch workspacekind jupyterlab-internal-3-4-ea1-v1-41 \
  --type=json \
  --patch-file add-llmcompressor-image-patch.json \
  -o yaml
```

该命令会将 JSON patch 文件应用到指定的 WorkspaceKind，并更新其 `imageConfig`，从而使新的工作台镜像可以在工作台创建 UI 中使用。

在实际使用中，你可以根据已同步的镜像以及集群中的命名规范，调整 `name`、`image` 和 `description` 字段。

为 Ascend vNPU 工作台配置补充组

如果你使用 `huawei.com/Ascend910B4` 之类的 Ascend vNPU 资源选项，请确认目标 `WorkspaceKind` pod template 在 `supplementalGroups` 中包含 Ascend 设备组。某些 vNPU 环境会将 `/dev/davinci*` 设备文件挂载为按组所有的字符设备，例如 `1000:1000`，权限模式为 `crw-rw----`。在这种情况下，单独的 `fsGroup` 无法授予对设备文件的访问权限，`npu-smi info` 等命令可能会失败，并提示 `dcmi module initialize failed. ret is -8005`。

请为提供 vNPU 工作台选项的 `WorkspaceKind` 打补丁：

```
kubectl patch workspacekind jupyterlab-internal-3-4-ea1-v1-41 \
  --type=merge \
  -p '{"spec":{"podTemplate":{"securityContext":{"fsGroup":0,"supplementa
lGroups":[1000]}}}}'
```

请使用你集群中拥有 Ascend 设备文件的组 ID。基于该 `WorkspaceKind` 创建的新工作台 Pod 将继承更新后的安全上下文。现有的工作台 Pod 必须重启后才能生效。

> **INFO**
>
> 我们还内置了一些资源选项，你可以在下拉菜单中看到它们。

Kubeflow Notebooks

来自 Kubeflow 中央监控面板的 Notebook 服务器、卷和 TensorBoard。

Kubeflow Notebooks 是 Workbench 的替代方案，而不是它的一部分。Workbench 是 Alauda AI 上的一体化体验；当您需要轻量级的上游安装，或者需要仅上游项目才具备的功能时，请使用 Kubeflow Notebooks。这些页面需要 Kubeflow operators。

使用 **Kubeflow Notebooks**

概念
创建 Notebook Server
连接到 Notebook
环境管理
管理配置（PodDefaults）
访问数据
最佳实践

使用 **Kubeflow** 存储卷

创建存储卷
管理存储卷
在 Notebook 中使用存储卷
下一步

使用 **Kubefl**

前提条件
使用 PyTorch 生
创建 TensorBoa
访问监控面板
使用场景
清理

使用 Kubeflow Notebooks

Kubeflow Notebooks 为数据科学家提供了一个 Kubernetes 原生的 Jupyter 环境，用于开发、训练和部署机器学习模型。每个 notebook 服务器都作为 namespace 中一个独立的 Pod 运行，从而确保隔离性和专用资源。

> 注意：我们建议使用 Alauda AI Workbench，以获得更集成的体验，并享受资源类型、配置以及与其他组件更好的集成等附加功能。不过，如果你更偏好轻量级的设置，或者需要上游项目中的特定功能，也可以使用原生的 Kubeflow Notebooks。

目录

概念
创建 Notebook Server
连接到 Notebook
环境管理
  安装 Python 包
  使用自定义镜像
管理配置（PodDefaults）
  创建 PodDefault
  应用配置
访问数据
  使用挂载卷
  使用对象存储
最佳实践

概念

- **Notebook Server**：在容器中运行的 JupyterLab 实例。
- **Custom Image**：你可以使用标准的预构建镜像（例如包含 TensorFlow、PyTorch），也可以提供带有特定库的自定义 Docker 镜像。
- **Persistent Storage**：默认情况下，notebook 服务器会挂载到 Persistent Volume Claims (PVCs)，用于存储工作区目录（通常是 `/home/jovyan`）。这可以确保即使服务器重启或更新，你的 notebooks 和数据也会被保存。

创建 **Notebook Server**

1. 访问监控面板： 导航到 Kubeflow 监控面板中的 **Notebooks** 部分。
2. 新建 **Notebook**： 点击 **New Notebook**。请确保在监控面板顶部选择正确的 namespace，即你希望创建 notebook 服务器的位置。
3. 配置服务器：
   - **Name**：为你的 notebook 服务器输入一个唯一名称。
   - **Image**：
     - **Select Type**：选择镜像类型，包括 JupyterLab、Visual Studio Code 或 RStudio。
     - **Select Image**：从预构建镜像列表中选择，或者通过提供 Docker 镜像 URL 指定自定义镜像。
   - **CPU / RAM**：根据工作负载分配 CPU 和 Memory 资源。可以先从较小的配置开始（例如 1 CPU、2GB RAM），如有需要再增加。
   - **GPUs**：如果你计划运行需要加速的深度学习训练或推理任务，请申请 GPUs（例如 NVIDIA）。
   - **Workspace Volume**：此卷会挂载到你的主目录（`/home/jovyan`）。你可以创建一个新卷（默认），或者挂载现有卷以访问之前的工作。
   - **Data Volumes**：（可选）附加其他现有 PVC，以便访问大型数据集，而无需将其复制到工作区。

- **Configurations**：（可选）选择 PodDefaults（如果可用），以注入诸如 S3 凭据、Git 配置或环境变量等通用配置。

4. 启动： 点击 **Launch**。系统将为该服务器进行 provisioning。等待状态显示变为 **Running**（绿色）。

连接到 **Notebook**

当服务器状态为 **Running** 后：

1. 点击 **Connect**。
2. 这将会在新的浏览器标签页中打开 **JupyterLab/VS Code/RStudio** 界面。
3. 现在你可以创建 Python 3 notebooks、打开终端，或者管理文件。

环境管理

安装 **Python** 包

虽然你可以将包安装到主目录中以便持久保存，但为了保证可重复性，最佳实践是使用自定义镜像。

在主目录中创建一个 "venv" 目录，并将包安装到其中：

```
python -m venv ~/venv
source ~/venv/bin/activate
python -m pip install transformers datasets
```

当你启动新的终端会话时，请记得激活虚拟环境，以访问已安装的包。

要在 Jupyter notebooks 中使用虚拟环境，可以安装 `ipykernel` 并创建一个新的 kernel：

```
source ~/venv/bin/activate
python -m pip install ipykernel
python -m ipykernel install --user --name=venv --display-name "Python (ve
nv)"
```

然后，在你的 Jupyter notebook 中，可以选择 "Python (venv)" kernel 来使用安装在虚拟环境中的包。

虚拟环境会保存在你的主目录中，因此即使你停止并重新启动 notebook 服务器，它们仍然可用。不过，如果你需要在多个 notebook 服务器之间共享该环境，或者希望获得更好的可重复性，可以考虑构建一个预装所需包的自定义 Docker 镜像。

使用自定义镜像

对于生产环境或复杂依赖（例如系统库），请构建一个包含所有必需库的 Docker 镜像，并在创建 notebook 时将其用作你的 **Custom Image**。这样可以确保完全可重复。

管理配置（**PodDefaults**）

Kubeflow 使用 `PodDefault` 资源（在 UI 中通常标记为 **Configurations**）向 Notebooks 注入常见配置，例如环境变量、卷和卷挂载。这是安全提供 Object Storage 凭据且无需在 notebooks 中硬编码的标准方式。

创建 **PodDefault**

你可以通过应用 YAML manifest 来创建 PodDefault。

定义一个 `PodDefault`，用于选择带有特定标签的 pods。

```yaml
apiVersion: kubeflow.org/v1alpha1
kind: PodDefault
metadata:
  name: add-gcp-secret
  namespace: MY_PROFILE_NAMESPACE
spec:
 selector:
  matchLabels:
    add-gcp-secret: "true"
 desc: "add gcp credential"
 volumeMounts:
 - name: secret-volume
   mountPath: /secret/gcp
 volumes:
 - name: secret-volume
   secret:
    secretName: gcp-secret
```

应用配置

创建新的 Notebook Server 时：

1. 滚动到 **Configurations** 部分。
2. 你将看到可用 PodDefaults 列表（例如 `s3-access`）。
3. 勾选复选框以应用它。

这将自动把指定的环境变量或卷注入到你的 Notebook 容器中。

访问数据

使用挂载卷

如果你在创建过程中附加了数据卷（PVC），它将可在指定的挂载点中使用。

```
import pandas as pd

# Assuming you mounted a data volume at /home/jovyan/data
df = pd.read_csv('/home/jovyan/data/dataset.csv')
print(df.head())
```

使用对象存储

要访问 Object Storage 中的数据，请使用 `boto3` 或 `s3fs` 等库。许多提供商都提供与 S3 兼容的 API，因此如果你的管理员已为凭据配置了 PodDefaults，那么诸如 `AWS_ACCESS_KEY_ID` 之类的环境变量通常会被预先填充。

```
import os
import s3fs
import pandas as pd

# Check if credentials are injected
print(os.getenv("AWS_S3_ENDPOINT"))

# Read directly from Object Storage
fs = s3fs.S3FileSystem(
    client_kwargs={'endpoint_url': os.getenv('AWS_S3_ENDPOINT')},
    key=os.getenv('AWS_ACCESS_KEY_ID'),
    secret=os.getenv('AWS_SECRET_ACCESS_KEY')
)

with fs.open('s3://my-bucket/data/train.csv') as f:
    df = pd.read_csv(f)
```

最佳实践

- 停止未使用的服务器：即使处于空闲状态，notebook 服务器也会消耗集群资源（尤其是 GPUs）。当你不在主动工作时，请停止它们。
- **Git** 集成：使用 JupyterLab 中的 Git 扩展（或终端）来对 notebooks 进行版本控制。避免将大型数据集存储在 Git 中。

- 资源监控：监控你的资源使用情况。如果你的 kernel 经常崩溃（OOM），你可能需要停止服务器，并使用更高的 Memory 限制重新启动它。
- 清理：如果旧 notebook 服务器及其关联的 PVC 不再需要，请定期删除它们。

使用 Kubeflow 存储卷

Kubeflow 中的存储卷作为 Kubernetes Persistent Volume Claims (PVCs) 进行管理。它们为你的数据、工作区和模型提供持久化存储，不受 Notebook servers 或其他工作负载生命周期的影响。

目录

创建存储卷

管理存储卷

在 Notebook 中使用存储卷

下一步

创建存储卷

1. 访问监控面板： 在 Kubeflow 中央监控面板侧边栏中点击 **Volumes**。
2. 新建存储卷： 点击 **New Volume**。
3. 配置：

- **Name**：为存储卷输入一个唯一名称。
- **Storage Class**：如果可用的 Storage Class 有多个，请选择一个（例如，topolvm、nfs）。
- **Size**：以 `Gi` 指定存储卷大小（例如，`10`）。

- **Access Mode**：
  - **ReadWriteOnce (RWO)**：由单个节点挂载（通常用于块存储）。
  - **ReadWriteMany (RWX)**：由多个节点挂载（通常用于 NFS/File 存储）。

4. 创建： 点击 **Create**。资源完成配置后，存储卷状态将变为 **Bound**。

管理存储卷

- 打开 **PVC Viewer**：点击存储卷旁边的 "Folder" 图标，创建一个临时 Pod 来挂载该存储卷并打开文件浏览器。这样你可以直接在存储卷中查看、上传或下载文件。完成后点击 "Close" 删除临时 Pod。
- 删除：点击存储卷旁边的删除图标（垃圾桶）将其移除。注意：这将永久删除数据。
- 筛选：使用搜索栏按名称、状态或 Storage Class 筛选存储卷。

在 **Notebook** 中使用存储卷

要在 Notebook Server 中使用存储卷：

1. 在创建新的 **New Notebook** 时，创建一个标准的 **Workspace Volume**（挂载到 `/home/jovyan`），或者……
2. 滚动到 **Data Volumes** 以附加其他现有存储卷。
3. 点击 **Attach Existing Volume** 并选择你的存储卷。
4. 指定 **Mount Path**（例如，`/home/jovyan/data`）。

下一步

要从同一监控面板中提供存储在这些存储卷之一上的模型服务，请参阅 从 Kubeflow 监控面板部署推理服务。

使用 Kubeflow Tensorboards

TensorFlow 的可视化工具 TensorBoard 是一个强大的监控面板，用于可视化机器学习实验。它允许你跟踪 loss 和 accuracy 等指标，查看 model graph，查看 weights 和 biases 的直方图，以及更多内容。

Kubeflow 提供了一种原生方式，可以直接在你的 Kubernetes 集群内创建 TensorBoard 实例，并将其指向存储在 Persistent Volume Claims (PVCs) 或 Object Storage 中的现有日志。

目录

前提条件
使用 PyTorch 生成日志
创建 TensorBoard 实例
访问监控面板
使用场景
可视化 Training 指标
比较运行
调试 Model Architecture
清理

前提条件

在创建 TensorBoard 实例之前，请确保你的 training 作业正在将日志写入集群可访问的位置。

- **PVC**：如果你的 training 作业将日志写入 Persistent Volume，请记录 PVC 名称以及其中的路径。
- **Object Storage**：如果你的 training 作业将日志写入 Object Storage，请确保你已具备必要的凭据（通常通过 PodDefaults 配置）以及 bucket URI（例如，`s3://my-bucket/logs/experiment-1`）。

使用 PyTorch 生成日志

要可视化你的 training 指标，PyTorch 代码必须将事件写入日志目录。`SummaryWriter` 类是将数据记录下来供 TensorBoard 使用的主要入口点。

```
import torch
import torchvision
from torch.utils.tensorboard import SummaryWriter
from torchvision import datasets, transforms

# Writer will output to ./runs/ directory by default
writer = SummaryWriter()

transform = transforms.Compose([transforms.ToTensor(), transforms.Normali
ze((0.5,), (0.5,))])
trainset = datasets.MNIST('mnist_train', train=True, download=True, trans
form=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffl
e=True)
model = torchvision.models.resnet50(False)
# Have ResNet model take in grayscale rather than RGB
model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=7, stride=2, padding=3,
bias=False)
images, labels = next(iter(trainloader))

grid = torchvision.utils.make_grid(images)
writer.add_image('images', grid, 0)
writer.add_graph(model, images)
writer.close()
```

创建 TensorBoard 实例

1. 访问 **Kubeflow** 监控面板： 导航到 Kubeflow 中央监控面板中的 **TensorBoards** 部分。
2. 新建 **TensorBoard**： 单击 **New TensorBoard** 按钮。
3. 配置实例：
   - **Name**：为你的 TensorBoard 实例输入一个唯一名称，例如 `experiment-1-viz`。
   - **PVC Source**：
     - 如果你的日志位于 PVC 上，请勾选此框。
     - **PVC Name**：从下拉列表中选择 PVC。
     - **Mount Path**：指定 PVC 内存储日志的路径，例如 `/logs/run1`。
   - **Object Storage Source**：
     - 如果你的日志位于云存储中，请勾选此框。
     - **Object Store Link**：提供日志目录的完整 URI，例如 `s3://my-bucket/my-model/logs/`。
     - **Configuration**：如果你的 bucket 需要凭据，请选择一个配置（PodDefault）。
4. 创建： 单击 **Create**。TensorBoard 实例将作为你命名空间中的一个 Pod 进行部署。

访问监控面板

一旦你的 TensorBoard 实例状态变为 **Running**：

1. 单击实例名称旁边的 **Connect**。
2. TensorBoard UI 将在新选项卡中打开。
3. 现在你可以浏览 training 运行生成的 scalars、graphs、distributions 和其他可视化内容。

使用场景

可视化 **Training** 指标

使用 **Scalars** 选项卡查看 accuracy、loss 和 learning rate 随时间变化的图表。这有助于诊断你的 model 是否发生过拟合，或者 learning rate 是否需要调整。

比较运行

如果你将 TensorBoard 指向一个包含多个运行子目录的父目录（例如 `run1`、`run2`），TensorBoard 会自动叠加这些运行的指标，从而让你比较不同 hyperparameters 下的性能。

调试 **Model Architecture**

使用 **Graphs** 选项卡可视化 model 的计算图。这可确保 model 按预期构建，并有助于识别结构性问题。

清理

TensorBoard 实例会消耗集群资源（CPU/Memory）。当你完成对实验的分析后：

1. 返回 **TensorBoards** 列表。
2. 单击实例旁边的 **Delete**（垃圾桶图标）按钮。
3. 确认删除。此操作会移除可视化服务器，但不会删除存储在 PVC 或 Object Storage 中的 training 日志或 model。

连接

介绍

核心概念

支持的 Connection 类型

使用 Connections

前提条件

创建连接

更新连接

删除连接

管理连接类型

在模型部署中使用连接

资源模型

介绍

Connection 使您能够在项目级别存储可复用的外部模型源和数据服务访问设置。在 Alauda AI 中，connection 会作为 Kubernetes `Secret` 存储在项目命名空间中，而 connection type 会作为 Kubernetes `ConfigMap` 存储在 `kube-public` 命名空间中。

目录

核心概念

Connection

Connection Type

支持的 Connection 类型

核心概念

Connection

connection 存储工作负载使用的实际访问参数，例如：

- S3 access key、secret key、endpoint、region 和 bucket
- OCI registry host 和 registry authentication
- 直接的远程 URI

Connection Type

connection type 定义了用户在创建或编辑 connection 时看到的表单。它控制：

- 显示哪些字段
- 哪些字段是必填项
- 默认值和只读设置
- 最终生成的 connection 是否与 URI、OCI 或 S3 model serving 兼容

支持的 **Connection** 类型

类型	**connection** 存储的内容	**model deployment** 如何使用它
URI	`URI` 字段中的完整远程 URI	deployment 表单直接从 connection 中读取 URI
OCI-compatible registry	registry host、registry credentials 和访问模式	deployment 表单将 connection Secret 用作 image pull secret，并要求您提供相对模型路径，例如 `repository:tag`
S3-compatible object storage	access key、secret key、endpoint、可选的地域和 bucket	deployment 表单使用 connection 中的 bucket，并要求您提供该 bucket 下的 object path

有关分步操作，请参见 Using Connections。

使用 Connections

Connections 可让你在项目级别存储可复用的外部模型源和数据服务访问设置。

有关从 S3、PVC 或 OCI 源进行模型部署的直接 YAML 示例，请参见 Model Storage。

> 注意： 对于基于 OCI 的模型部署，模型部署表单中仅提供 `ACCESS_TYPE` 包含 `Pull` 的连接。

目录

- 前提条件
- 创建连接
  - 内置字段集
  - 验证
- 更新连接
- 删除连接
- 管理连接类型
  - 连接类型包含的内容
  - 内置模板
  - 支持的字段类型
  - 创建连接类型
  - 复制、编辑、启用和删除
- 在模型部署中使用连接
  - 运行时行为

资源模型

连接 Secret 示例

连接类型 ConfigMap 示例

前提条件

- 你有权访问目标项目命名空间。
- 已存在合适的连接类型。
- 你拥有要使用的外部存储或 registry 的凭据和终端节点信息。

创建连接

1. 在 **User View** 中，打开目标项目。
2. 点击 **Connections**。
3. 点击 **Create connection**。
4. 选择一个连接类型。
5. 输入该连接的 Kubernetes 资源名称。
6. 可选：输入显示名称。
7. 完成所选连接类型定义的字段。
8. 点击 **Create**。

内置字段集

- **URI**：在 `URI` 字段中提供完整 URI。
- **OCI-compliant registry**：提供 `ACCESS_TYPE`，上传或粘贴 `.dockerconfigjson`，并输入 `OCI_HOST`。
- **S3-compatible object storage**：提供 `AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY`、`AWS_S3_ENDPOINT`，并可选地提供 `AWS_DEFAULT_REGION` 和 `AWS_S3_BUCKET`。

验证

- 新建项会出现在 **Connections** 页面上。
- 表格会显示连接名称、连接类型、兼容性标签和创建者。

> 注意： 连接资源名称和所选连接类型在创建后无法更改。

更新连接

1. 在 **Connections** 页面上，打开目标连接的操作菜单。
2. 点击 **Edit**。
3. 更新显示名称或连接字段。
4. 点击 **Save**。

在你更新凭据或终端节点后，请重新部署或重启依赖的工作负载，以便它们获取最新值。

删除连接

1. 在 **Connections** 页面上，打开目标连接的操作菜单。
2. 点击 **Delete**。
3. 确认该操作。

删除连接只会移除项目 Secret。任何仍依赖该 Secret 的工作负载都必须单独更新。

管理连接类型

连接类型按集群范围进行管理，并存储在 `kube-public` 命名空间中。

连接类型包含的内容

- 基本元数据：名称、显示名称、类别和启用状态
- 按顺序排列的分区标题和字段

- 可选的默认值和只读设置
- 用于 URI、OCI 或 S3 模型服务的兼容性标记

内置模板

连接类型编辑器可追加以下预定义字段集：

- **URI**
- **OCI compliant registry**
- **S3 compatible object storage**

支持的字段类型

字段类型	作用
Section	仅用于视觉分组
Text - Short / Text - Long	纯文本输入
Text - Hidden	密码或密钥等敏感值
URI	URI 输入
Boolean	切换输入
Numeric	数值输入，可选最小值、最大值和单位
Dropdown	单选或多选选项
File	上传或粘贴文件内容，例如 `.dockerconfigjson`

> 注意： 当前 UI 中的 File 类型字段支持粘贴内容或上传最大 1 MiB 的文件。

创建连接类型

1. 打开目标集群，转到 **Settings** > **Connection types**。

2. 点击 **Create connection type**。
3. 输入连接类型名称。平台会自动为其添加 `ct-` 前缀。
4. 可选：输入显示名称。
5. 选择一个或多个类别。
6. 根据需要设置 **Enabled**。
7. 添加分区标题和字段，或应用内置的模型服务兼容模板。
8. 点击 **Create**。

复制、编辑、启用和删除

- 当你希望基于现有连接类型创建新的连接类型时，请使用 **Duplicate**。
- 自定义连接类型可以被编辑或删除。
- 预装连接类型可以复制，但当前 UI 不允许直接编辑或删除它们。
- 列表页提供 **Enabled** 开关，用于将连接类型标记为启用或禁用。

> 注意： 编辑连接类型不会重写之前创建的现有连接 Secret。

在模型部署中使用连接

当你创建或更新推理服务时，可以将 **Model location** 设置为 **Connection**。

1. 打开推理服务创建或更新表单。
2. 将 **Model location** 设置为 **Connection**。
3. 选择该连接。
4. 根据连接类型完成模型路径字段：
   - **URI**：不需要额外路径；平台会从连接中读取完整 URI。
   - **OCI**：输入 `oci://<OCI_HOST>/` 下的相对路径，例如 `my-model:latest`。
   - **S3**：输入 `s3://<AWS_S3_BUCKET>/` 下的对象路径，例如 `llm/qwen2.5`。

运行时行为

- 对于 **OCI** 连接，连接 Secret 会作为 `imagePullSecret` 挂载到工作负载。
- 对于 **S3** 连接，平台会创建一个专用的 `ServiceAccount`，并引用该连接 Secret。
- 推理服务元数据会将所选连接名称存储在 `aml-model-source-connection` 注解中。

资源模型

当前实现对连接使用项目 Secret，对连接类型使用集群级别的 ConfigMap。

连接 **Secret** 示例

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: minio-models
  namespace: demo-project
  labels:
    cpaas.io/dashboard: "true"
  annotations:
    cpaas.io/display-name: MinIO models
    cpaas.io/connection-type-ref: ct-s3-storage
    cpaas.io/connection-type-protocol: s3
    serving.kserve.io/s3-endpoint: minio.minio.svc:9000
    serving.kserve.io/s3-region: us-east-1
type: Opaque
stringData:
  AWS_ACCESS_KEY_ID: <access-key>
  AWS_SECRET_ACCESS_KEY: <secret-key>
  AWS_S3_ENDPOINT: http://minio.minio.svc:9000
  AWS_DEFAULT_REGION: us-east-1
  AWS_S3_BUCKET: models
```

连接类型 **ConfigMap** 示例

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: ct-uri-repository
  namespace: kube-public
  labels:
    cpaas.io/dashboard: "true"
    cpaas.io/connection-type: "true"
  annotations:
    cpaas.io/display_name: URI repository
    cpaas.io/disabled: "false"
data:
  category: '["URI"]'
  fields: >-
    [
      {
        "type": "section",
        "name": "URI",
        "description": "Fields required for URI model serving."
      },
      {
        "type": "uri",
        "name": "URI",
        "envVar": "URI",
        "required": true,
        "properties": {
          "defaultReadOnly": false
        }
      }
    ]
```

> 注意： 与最近的 OpenShift AI 文档相比，当前 Alauda AI 前端仍通过 `cpaas.io/connection-type-ref` 将连接绑定到其模板，并且还会将模型服务兼容性写入 `cpaas.io/connection-type-protocol`。

Pipelines

创建 pipeline server，构建并运行 Kubeflow Pipelines，并从 Alauda DevOps 触发它们。

创建 Data Science Pipelines Ap

前提条件
1. 选择数据库和对象存储
2. 创建 DSPA
3. 验证 DSPA 是否就绪
4. 使用 pipeline 栈（KFP v2 SDK）
清理

使用 Kubeflow Pipelines

前提条件
快速开始示例
在 UI 中管理流水线

使用 Elyra 在

前提条件
验证 namespac
配置 Elyra 元数
可用的 Elyra Pi

使用可复用的 Kubeflow Pipeline 组件

前提条件
创建输入存储 Secret
运行托管 tabular pipeline

Kubeflow Pipelines 执行和存储行为

功能概览
安全地缓存任务
使用 `ParallelFor` 运行任务
持久化并传递类型化 artifact
运行验证 pipeline
在 arm64 集群上部署
平台兼容性和数据库后端

使用 Kueue

前提条件
操作步骤

创建 Data Science Pipelines Application

本指南介绍如何创建一个 `DataSciencePipelinesApplication`（DSPA）——即命名空间中的一个隔离的 Kubeflow Pipelines v2 栈——然后使用 `kfp` Python SDK 进行访问。

你可以通过两种方式创建 DSPA：

- **`kubectl`** —— 应用 `DataSciencePipelinesApplication` 清单。
- **Alauda Console UI** —— 依次进入 **Administrator** → **Marketplace** → **OperatorHub** → **Data Science Pipelines Operator** → **All instances** → **Create**。

这两种方式创建的是同一个资源；请选择最适合你工作流的方式。

目录

前提条件

1. 选择数据库和对象存储
2. 创建 DSPA

   方式 A — 使用 `kubectl`

   方式 B — 使用 Alauda Console UI
3. 验证 DSPA 是否就绪
4. 使用 pipeline 栈（KFP v2 SDK）

清理

前提条件

- 已安装 Data Science Pipelines Operator，且其 CSV 报告为 `Succeeded` —— 参见 安装。
- 具备目标集群的 `kubectl` 访问权限（用于 CLI 方式和验证）。
- 为 pipeline 栈准备一个目标项目 **namespace**（本指南使用 `data-science-project`）。

1. 选择数据库和对象存储

每个 DSPA 都需要一个数据库（用于 pipeline metadata）和一个兼容 S3 的对象存储（用于 pipeline artifacts）。你有两种选择：

- 托管方式（最简单） —— 让 operator 在 DSPA namespace 中为你部署 MariaDB 和 MinIO。适合快速开始或自包含项目。

> **WARNING**
>
> 托管的 MariaDB 和 MinIO 仅适用于开发和测试（它们是单实例，没有备份或 HA）。生产环境请使用外部后端。

- 外部方式 —— 将 DSPA 指向你自己的 MySQL 和兼容 S3 的存储。推荐用于生产环境（你可以自行控制备份、HA 和规格）。

对于外部部署，请先创建凭证 secrets：

```
kubectl create namespace data-science-project

# database password
kubectl -n data-science-project create secret generic dspa-db \
  --from-literal=password='<db-password>'

# object-storage access + secret keys
kubectl -n data-science-project create secret generic dspa-s3 \
  --from-literal=accesskey='<s3-access-key>' \
  --from-literal=secretkey='<s3-secret-key>'
```

2. 创建 DSPA

方式 A — 使用 `kubectl`

托管后端（快速开始）： operator 会部署 MariaDB + MinIO。

```yaml
apiVersion: datasciencepipelinesapplications.opendatahub.io/v1
kind: DataSciencePipelinesApplication
metadata:
  name: sample
  namespace: data-science-project
spec:
  dspVersion: v2
  apiServer:
    enableSamplePipeline: false
  objectStorage:
    minio:
      deploy: true
  # database omitted → the operator deploys a managed MariaDB
```

外部后端（生产环境）： 指向你自己的 MySQL 和 S3。

```yaml
apiVersion: datasciencepipelinesapplications.opendatahub.io/v1
kind: DataSciencePipelinesApplication
metadata:
  name: sample
  namespace: data-science-project
spec:
  dspVersion: v2
  apiServer:
    enableOauth: false
    enableSamplePipeline: false
  workflowController:
    deploy: true
  mlmd:
    deploy: true
  database:
    externalDB:
      host: mysql.example.com
      port: "3306"
      username: root
      pipelineDBName: mlpipeline
      passwordSecret:
        name: dspa-db
        key: password
  objectStorage:
    externalStorage:
      host: s3.example.com
      port: "8333"
      scheme: http
      bucket: mlpipeline
      s3CredentialsSecret:
        secretName: dspa-s3
        accessKey: accesskey
        secretKey: secretkey
```

保存为 `dspa.yaml` 并应用：

```bash
kubectl apply -f dspa.yaml
```

方式 B — 使用 Alauda Console UI

1. 在 **Administrator** 视图中，进入 **Marketplace** → **OperatorHub**。
2. 在顶部的 **Cluster** 下拉列表中，选择目标集群。
3. 打开已安装的 **Data Science Pipelines Operator**。
4. 切换到 **All instances** 选项卡并点击 **Create**（如有提示，选择 **DataSciencePipelinesApplication**）。
5. 填写表单，或切换到 **YAML** 视图并粘贴方式 A 中的某个清单。将 **Namespace** 设置为你的项目 namespace（例如 `data-science-project`）。
6. 点击 **Create**。

此时该实例会出现在 **All instances** 下；其状态会反映 DSPA 的就绪情况（请参见下一步）。

3. 验证 DSPA 是否就绪

```
# Ready=True means all components reconciled
kubectl -n data-science-project get dspa sample \
  -o jsonpath='{range .status.conditions[*]}{.type}={.status}{"\n"}{end}'

# the component pods are Running
kubectl -n data-science-project get pods -l component=data-science-pipelines
```

你应该会看到 `Ready=True`，并且 `ds-pipeline-*` pods（APIServer、persistence agent、scheduled-workflow、workflow controller、MLMD）处于 `Running` 状态。

4. 使用 pipeline 栈（KFP v2 SDK）

DSPA 的 APIServer 可在集群内通过 `http://ds-pipeline-<name>.<namespace>.svc:8888` 访问——这里是 `http://ds-pipeline-sample.data-science-project.svc:8888`。使用 `kfp` Python SDK 编译并上传 pipeline（可从 pod 中执行，或从能够访问该 Service 的机器上执行）：

```
import kfp
from kfp import dsl

@dsl.component(base_image="python:3.12-slim")
def hello(msg: str) -> str:
    return msg

@dsl.pipeline(name="hello-pipeline")
def hello_pipeline(msg: str = "hello"):
    hello(msg=msg)

client = kfp.Client(
    host="http://ds-pipeline-sample.data-science-project.svc:8888",
    namespace="data-science-project",
)

# compile + upload the pipeline
kfp.compiler.Compiler().compile(hello_pipeline, package_path="hello.yam
l")
uploaded = client.upload_pipeline("hello.yaml", pipeline_name="hello-pipe
line")
print("uploaded pipeline:", uploaded.pipeline_id)

# confirm it is listed
for p in client.list_pipelines(page_size=10).pipelines or []:
    print(" -", p.display_name, p.pipeline_id)
```

如果你安装 operator 时设置了 `EXTERNAL_ROUTE_PROVIDER=virtualservice`，那么 DSPA 的 `status.components.apiServer.externalUrl` 也会提供一个通过共享 Istio gateway 访问的外部路径。

清理

```
kubectl -n data-science-project delete dspa sample
```

删除 DSPA 会移除其创建的 pipeline 栈。在使用托管后端时，托管的 MariaDB/MinIO（及其数据）也会一并删除；外部后端则不会受到影响。

使用 Kubeflow Pipelines

Kubeflow Pipelines (KFP) 是一个基于 Docker 容器，用于构建和部署可移植、可扩展的机器学习（ML）工作流的平台。KFP SDK 允许你使用 Python 定义和操作流水线及组件。

目录

- 前提条件
  - 安装 KFP SDK
  - 配置 KFP 以使用你的对象存储运行
- 快速开始示例
- 在 UI 中管理流水线
  - 访问 Pipelines 监控面板
  - 上传流水线
  - 创建运行
  - 查看运行详情
  - 定期运行

前提条件

安装 KFP SDK

在你的命名空间中启动一个 Jupyter Notebook（或 Workbench），并安装 KFP SDK：

```
python -m pip install kfp
```

配置 KFP 以使用你的对象存储运行

在安装了外部 Object Storage 的 Kubeflow 时，请添加一个 `KFP Launcher` ConfigMap，用于配置当前命名空间或用户所使用的存储。你可以参考 Kubeflow 文档 https://www.kubeflow.org/docs/components/pipelines/operator-guides/configure-object-store/#s3-and-s3-compatible-provider ↗ 获取更多详情。如果未设置任何配置，pipeline 运行仍可能尝试访问默认的集群内对象存储端点，而这可能与你的部署不匹配。

从 Kubeflow 1.11.0 开始，默认捆绑的对象存储将更改为 SeaweedFS。为了让你的环境在不同版本之间保持可移植性，请将 Kubeflow Pipelines 配置为使用你自己的 Object Storage 端点，而不是依赖内置的存储服务名称。

下面是一个供你开始使用的简单示例：

```
apiVersion: v1
data:
  defaultPipelineRoot: s3://pipeline-artifacts
  providers: |-
    s3:
      default:
        endpoint: object-storage.storage.svc:80
        disableSSL: true
        region: us-east-2
        forcePathStyle: true
        credentials:
          fromEnv: false
          secretRef:
            secretName: mlpipeline-object-storage-artifact
            accessKeyKey: accesskey
            secretKeyKey: secretkey
kind: ConfigMap
metadata:
  name: kfp-launcher
  namespace: wy-testns
```

例如，在此 ConfigMap 中设置以下值，以指向你自己的 Object Storage：

defaultPipelineRoot：用于存储流水线中间数据和制品的位置
endpoint：Object Storage 服务端点。它不应以 `http` 或 `https` 开头
disableSSL：是否禁用对该端点的 HTTPS 访问
region：Object Storage 提供商所需的地域
credentials：存储在所引用的 Secret 中的访问密钥和密钥

添加此 ConfigMap 后，新启动的 Kubeflow Pipeline 运行会自动读取该配置，并将流水线数据存储到已配置的 Object Storage 中。

快速开始示例

流水线是 ML 工作流的描述，包括工作流中的所有组件，以及它们如何以图的形式组合在一起。

下面是一个使用 KFP SDK 定义一个打印 "Hello, World!" 的流水线的简单示例。

```python
from kfp import dsl
from kfp import compiler
from kfp.client import Client

@dsl.component
def say_hello(name: str) -> str:
    hello_text = f'Hello, {name}!'
    print(hello_text)
    return hello_text

@dsl.pipeline
def hello_pipeline(recipient: str) -> str:
    hello_task = say_hello(name=recipient)
    return hello_task.output


# Compile the pipeline to a YAML file
compiler.Compiler().compile(hello_pipeline, 'pipeline.yaml')

# Create a KFP client and submit the pipeline run
client = Client(host='<MY-KFP-ENDPOINT>')
run = client.create_run_from_pipeline_package(
    'pipeline.yaml',
    arguments={
        'recipient': 'World',
    },
)
```

有关如何定义和运行流水线的更多详情，请参阅官方 KFP 文档：
https://www.kubeflow.org/docs/components/pipelines/user-guides/

在 UI 中管理流水线

你也可以直接从 Kubeflow 监控面板中管理流水线、实验和运行。

访问 Pipelines 监控面板

1. 登录 Kubeflow 中央监控面板。

2. 在侧边栏菜单中点击 **Pipelines**。

上传流水线

如果你已将流水线编译为一个 YAML 文件（例如上面示例中的 `pipeline.yaml`），则可以上传它：

1. 点击 **Pipelines** -> **Upload Pipeline**。
2. **Upload a file**：选择你的 `pipeline.yaml`。
3. **Pipeline Name**：为其指定一个名称（例如 `Hello World Pipeline`）。
4. 点击 **Create**。

创建运行

要执行你刚上传的流水线：

1. 点击流水线名称以打开其详情。
2. 点击 **Create Run**。
3. **Run Name**：输入一个具有描述性的名称。
4. **Experiment**：选择一个现有实验或创建一个新的实验。实验有助于对相关运行进行分组。
5. **Run Parameters**：为任何流水线参数输入值（例如，`recipient`：`World`）。
6. 点击 **Start**。

查看运行详情

运行启动后，你将被重定向到 **Run Details** 页面。

- **Graph**：可视化你的流水线步骤（组件）及其状态（Running、Succeeded、Failed）。
- **Logs**：在图中点击某个特定步骤，以在侧边面板中查看其容器日志。这对于调试至关重要。
- **Inputs/Outputs**：查看在步骤之间传递或作为最终输出生成的制品。
- **Visualizations**：如果你的流水线生成指标或图表，它们将显示在 **Run Output** 或 **Visualizations** 选项卡中。

定期运行

你可以安排流水线按特定间隔自动运行：

1. 在 **Pipelines** 列表中找到你的流水线。
2. 点击 **Create Run**，但将运行类型选择为 **Recurring Run**（或导航到 **Experiments (KFP)** -> **Create Recurring Run**）。
3. **Trigger**：设置计划（例如，Periodic、Cron）。
4. **Parameters**：配置每次计划执行时将使用的输入。
5. 点击 **Start**。

使用 Elyra 在 JupyterLab 中运行 Kubeflow Pipelines

Elyra 允许你在 JupyterLab 中通过在画布上连接 notebook 来构建可视化 pipeline，然后将该 pipeline 提交到 Kubeflow Pipelines (KFP)。本指南将通过一个由两个 notebook 组成的“hello world” pipeline 演示整个过程，并展示如何在 KFP UI 中验证运行结果。

此工作流使用 KFP v2。每个 notebook 节点都会作为容器化的 pipeline 步骤执行，KFP 会将运行产物存储在与 S3 兼容的对象存储中，例如 Ceph Object Storage。

目录

- 前提条件
- 验证 namespace 的 KFP 运行时配置
- 配置 Elyra 元数据挂载
  - 挂载到 `/opt/app-root/runtimes` 的 KFP 运行时配置
  - 挂载到 `/opt/app-root/pipeline-runtimes` 的 Pipeline 运行时镜像元数据
- 可用的 Elyra Pipeline 运行时镜像
- 打开 JupyterLab
- 创建两个演示 notebook
- 在 Elyra 中创建 pipeline
  - 打开 Pipeline Editor
  - 添加 Notebook 节点
  - 配置节点属性
- 将 pipeline 提交到 KFP

在 Kubeflow Pipelines 中验证运行结果

故障排查

namespace 在 Kubeflow 中不可见

Elyra 未显示 KFP 运行时配置

Elyra 中没有可用的运行时镜像

在 notebook 代码开始执行前运行失败

Elyra 的成功对话框已关闭

前提条件

开始之前，请确保以下平台前提条件已就绪：

- 已安装 Alauda AI Workbench，并且你可以创建或打开一个 JupyterLab workbench。有关 workbench 创建步骤，请参见 创建 Workbench。
- 已部署 Kubeflow Base (`kfbase`) 和 Kubeflow Pipelines (`kfp`)。有关部署步骤，请参见 安装 Kubeflow Plugins。
- 你的 namespace 在 Kubeflow 中可见。如果在登录 Kubeflow 后 namespace 仍未显示，请按照 安装 Kubeflow Plugins 中的 namespace 绑定说明进行操作。
- 已配置 KFP 对象存储。有关 namespace 侧 `kfp-launcher` 配置，请参见 使用 Kubeflow Pipelines。
- 你使用的 JupyterLab workbench 镜像包含 Elyra 和 KFP SDK 2.x，例如 创建 Workbench 中列出的 **Standard Data Science** Jupyter 镜像。
- Elyra 运行时元数据已挂载到 workbench 中。运行时配置会告诉 Elyra 如何提交到 KFP，而运行时镜像元数据会告诉 Elyra 每个 pipeline 节点可以使用哪个镜像运行。

> **NOTE**
>
> code-server workbench 镜像不提供 Elyra 可视化 pipeline 编辑器。若要创建 Elyra pipeline，请使用 JupyterLab 镜像。

验证 namespace 的 KFP 运行时配置

在使用 Elyra 之前，请让平台管理员确认 workbench 所在的 namespace 已准备好用于 KFP v2 运行：

- 该 namespace 在 Kubeflow 中可见，并且已绑定到用户。请参见 安装 Kubeflow Plugins。
- KFP 已配置为将产物存储到对象存储中。请参见 使用 Kubeflow Pipelines。

本指南不会重复介绍 KFP 安装和对象存储 manifest，而是重点介绍 JupyterLab 中的 Elyra 配置。

配置 Elyra 元数据挂载

本节通常由平台管理员完成。JupyterLab workbench 启动时，Elyra 会读取两个不同的元数据目录：

挂载路径	用途	推荐来源
`/opt/app-root/runtimes`	KFP 运行时配置。该配置会向 Elyra 提供 KFP API endpoint、namespace、认证类型以及对象存储设置。	在当前 Workbench 版本中临时使用 PVC。
`/opt/app-root/pipeline-runtimes`	Pipeline 运行时镜像元数据。该配置会告诉 Elyra 每个 pipeline 节点可用的容器镜像。	通过 `WorkspaceKind` 挂载 namespace 级别的 ConfigMap。

需要使用 `..data` 路径。Kubernetes 会为 ConfigMap 和 Secret 卷自动创建这种符号链接布局，JupyterLab 镜像也会按设计遵循该布局。当前针对 `/opt/app-root/runtimes` 的 PVC 方案也需要相同布局，因此在准备 PVC 时需要自行创建 `..data` 目录。

在未来的 Workbench 版本中，Workspace 级别的 Secret 挂载将允许直接从 Secret 挂载 `/opt/app-root/runtimes`。等到该功能可用后，你就可以停止使用 PVC 方案来存放敏感的 Elyra 运行时配置。

挂载到 `/opt/app-root/runtimes` 的 KFP 运行时配置

当前 Workbench 版本不支持在 Workspace 中直接挂载 Secret。由于 Elyra 的 KFP 运行时配置可能包含对象存储凭据，请使用专用 PVC 作为临时方案，并将该 PVC 视为敏感数据。

PVC 中必须包含如下文件：

```
/opt/app-root/runtimes/
  ..data/
    mlops-kfp.json
  mlops-kfp.json -> ..data/mlops-kfp.json
```

请使用以下 `mlops-kfp.json` 作为模板。将 namespace、endpoint、bucket 和凭据替换为你自己的值：

mlops-kfp.json

```
{
  "display_name": "MLOps KFP",
  "metadata": {
    "runtime_type": "KUBEFLOW_PIPELINES",
    "description": "Kubeflow Pipelines runtime for the workbench namespac
e",
    "api_endpoint": "http://ml-pipeline.kubeflow.svc:8888",
    "user_namespace": "<your-namespace>",
    "engine": "Argo",
    "auth_type": "KUBERNETES_SERVICE_ACCOUNT_TOKEN",
    "cos_endpoint": "http://<ceph-rgw-service>.<ceph-namespace>.svc:748
0",
    "cos_bucket": "<your-kfp-artifact-bucket>",
    "cos_auth_type": "KUBERNETES_SECRET",
    "cos_secret": "elyra-cos-credentials",
    "cos_username": "<object-storage-access-key>",
    "cos_password": "<object-storage-secret-key>",
    "tags": [
      "kfp",
      "mlops",
      "<your-namespace>"
    ],
    "public_api_endpoint": "https://<kubeflow-domain>/_/pipeline"
  },
  "schema_name": "kfp"
}
```

如果该 PVC 已经挂载到 workbench 中，你可以在 JupyterLab 终端中创建该文件：

```
mkdir -p /opt/app-root/runtimes/..data
cat > /opt/app-root/runtimes/..data/mlops-kfp.json <<'EOF'
{
  "display_name": "MLOps KFP",
  "metadata": {
    "runtime_type": "KUBEFLOW_PIPELINES",
    "description": "Kubeflow Pipelines runtime for the workbench namespac
e",
    "api_endpoint": "http://ml-pipeline.kubeflow.svc:8888",
    "user_namespace": "<your-namespace>",
    "engine": "Argo",
    "auth_type": "KUBERNETES_SERVICE_ACCOUNT_TOKEN",
    "cos_endpoint": "http://<ceph-rgw-service>.<ceph-namespace>.svc:748
0",
    "cos_bucket": "<your-kfp-artifact-bucket>",
    "cos_auth_type": "KUBERNETES_SECRET",
    "cos_secret": "elyra-cos-credentials",
    "cos_username": "<object-storage-access-key>",
    "cos_password": "<object-storage-secret-key>",
    "tags": ["kfp", "mlops", "<your-namespace>"],
    "public_api_endpoint": "https://<kubeflow-domain>/_/pipeline"
  },
  "schema_name": "kfp"
}
EOF
ln -sf ..data/mlops-kfp.json /opt/app-root/runtimes/mlops-kfp.json
```

将 PVC 挂载到现有 Workspace 中。以下片段是一个 patch，而不是完整的 manifest：`spec.kind` 和 `spec.podTemplate.options` 是 `Workspace` 上的必填字段，因此单独应用此 YAML 会被拒绝。请改为 patch 现有对象：

```
kubectl patch workspace <workspace-name> -n <your-namespace> --type=merge
-p '
spec:
  podTemplate:
    volumes:
      data:
        - mountPath: /opt/app-root/runtimes
          pvcName: <elyra-runtime-config-pvc>
          readOnly: false
'
```

请将 `<workspace-name>`、`<your-namespace>` 和 `<elyra-runtime-config-pvc>` 替换为环境中的实际值。

创建或更新文件后，请重启 workbench，以便 Elyra 加载运行时配置。

挂载到 `/opt/app-root/pipeline-runtimes` 的 Pipeline 运行时镜像元数据

Pipeline 运行时镜像元数据不是用户特定的。namespace 级别的 ConfigMap 更适合这一用途，并且可以通过 `WorkspaceKind` 挂载，这样该类型下的所有 JupyterLab workspaces 都会获得相同的运行时镜像列表。

在每个运行 Elyra pipeline 的 namespace 中创建一个名为 `pipeline-runtime-images` 的 ConfigMap：

pipeline-runtime-images.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: pipeline-runtime-images
  namespace: <your-namespace>
data:
  odh-pipeline-runtime-minimal-cpu-py312-ubi9.json: |
    {
      "schema_name": "runtime-image",
      "display_name": "Runtime | Minimal | CPU | Python 3.12",
      "metadata": {
        "description": "Minimal runtime image for Elyra pipeline nodes.",
        "image_name": "docker.io/alaudadockerhub/odh-pipeline-runtime-min
imal-cpu-py312-ubi9:<tag>",
        "pull_policy": "IfNotPresent",
        "tags": ["kfp", "minimal", "cpu", "python-3.12", "ubi9"]
      }
    }
  odh-pipeline-runtime-datascience-cpu-py312-ubi9.json: |
    {
      "schema_name": "runtime-image",
      "display_name": "Runtime | Data Science | CPU | Python 3.12",
      "metadata": {
        "description": "Data science runtime image for Elyra pipeline nod
es.",
        "image_name": "docker.io/alaudadockerhub/odh-pipeline-runtime-dat
ascience-cpu-py312-ubi9:<tag>",
        "pull_policy": "IfNotPresent",
        "tags": ["kfp", "datascience", "cpu", "python-3.12", "ubi9"]
      }
    }
  odh-pipeline-runtime-tensorflow-cuda-py312-ubi9.json: |
    {
      "schema_name": "runtime-image",
      "display_name": "Runtime | TensorFlow | CUDA | Python 3.12",
      "metadata": {
        "description": "TensorFlow CUDA runtime image for Elyra pipeline
nodes.",
        "image_name": "docker.io/alaudadockerhub/odh-pipeline-runtime-ten
sorflow-cuda-py312-ubi9:<tag>",
        "pull_policy": "IfNotPresent",
        "tags": ["kfp", "tensorflow", "cuda", "python-3.12", "ubi9"]
      }
```

```
    }
  odh-pipeline-runtime-pytorch-cuda-py312-ubi9.json: |
    {
      "schema_name": "runtime-image",
      "display_name": "Runtime | PyTorch | CUDA | Python 3.12",
      "metadata": {
        "description": "PyTorch CUDA runtime image for Elyra pipeline nod
es.",
        "image_name": "docker.io/alaudadockerhub/odh-pipeline-runtime-pyt
orch-cuda-py312-ubi9:<tag>",
        "pull_policy": "IfNotPresent",
        "tags": ["kfp", "pytorch", "cuda", "python-3.12", "ubi9"]
      }
    }
  odh-pipeline-runtime-pytorch-llmcompressor-cuda-py312-ubi9.json: |
    {
      "schema_name": "runtime-image",
      "display_name": "Runtime | PyTorch LLM Compressor | CUDA | Python
3.12",
      "metadata": {
        "description": "PyTorch and LLM Compressor CUDA runtime image for
Elyra pipeline nodes.",
        "image_name": "docker.io/alaudadockerhub/odh-pipeline-runtime-pyt
orch-llmcompressor-cuda-py312-ubi9:<tag>",
        "pull_policy": "IfNotPresent",
        "tags": ["kfp", "pytorch", "llmcompressor", "cuda", "python-3.1
2", "ubi9"]
      }
    }
```

如果你的集群从私有镜像仓库镜像源拉取镜像，请将 `docker.io/alaudadockerhub/...:<tag>` 替换为镜像仓库中的地址。

通过 JupyterLab `WorkspaceKind` 挂载该 ConfigMap：

```
apiVersion: kubeflow.org/v1beta1
kind: WorkspaceKind
metadata:
  name: <jupyterlab-workspacekind-name>
spec:
  podTemplate:
    extraVolumeMounts:
      - name: pipeline-runtime-images
        mountPath: /opt/app-root/pipeline-runtimes
    extraVolumes:
      - name: pipeline-runtime-images
        configMap:
          name: pipeline-runtime-images
          optional: true
```

ConfigMap 挂载会自动创建 Kubernetes 原子卷布局。在 pod 内部，Elyra 会看到如下文件：

```
/opt/app-root/pipeline-runtimes/
  ..data -> ..2026_...
  odh-pipeline-runtime-minimal-cpu-py312-ubi9.json -> ..data/odh-pipeline
-runtime-minimal-cpu-py312-ubi9.json
```

可用的 Elyra Pipeline 运行时镜像

Elyra pipeline 运行时镜像发布在 Docker Hub 上的 alaudadockerhub↗ 下。它们与 JupyterLab workbench 镜像不同：workbench 镜像运行编写界面，而这些运行时镜像用于运行单个 KFP pipeline 节点。

与 创建 Workbench 中介绍的其他 workbench 镜像一样，这些 Docker Hub 地址属于公开源镜像。在私有环境或离线环境中，请先将所需的 pipeline 运行时镜像同步到你内部的镜像仓库，然后更新 `pipeline-runtime-images` ConfigMap 中的 `metadata.image_name` 字段，使其指向内部镜像仓库地址。

下表介绍了五个常用的运行时镜像。包列表仅作参考，基于镜像构建仓库中 `runtimes/` 下对应的源目录。

运行时镜像	描述	主要包

Minimal CPU
alaudadockerhub/odh-pipeline-runtime-minimal-cpu-py312-ubi9 ↗
	适用于轻量级 Python notebook 节点和简单的控制流步骤。	`Python 3.12`
		Elyra notebook 执行依赖，例如 `papermill`、`nbclient`、`nbconvert`、`nbformat`、`ipykernel`
		`minio` 客户端
		`requests`

Data Science CPU
alaudadockerhub/odh-pipeline-runtime-datascience-cpu-py312-ubi9 ↗
	适用于通用的基于 CPU 的数据处理和 ML pipeline 节点。	`Python 3.12`
		`NumPy`
		`pandas 2.3.3`
		`SciPy 1.16.x`
		`scikit-learn 1.8.0`
		`Matplotlib 3.10.x`
		`Plotly 6.5.2`
		`CodeFlare SDK 0.35.x`
		`Feast 0.60.x`

TensorFlow CUDA
alaudadockerhub/odh-pipeline-runtime-tensorflow-cuda-py312-ubi9 ↗
	适用于 NVIDIA GPU 节点上的 TensorFlow pipeline 节点。	`Python 3.12`
		CUDA 基础镜像
		`TensorFlow 2.20.x`
		`TensorBoard 2.20.x`
		数据科学运行时依赖

PyTorch CUDA
alaudadockerhub/odh-pipeline-runtime-pytorch-cuda-py312-ubi9 ↗
	适用于 NVIDIA GPU 节点上的 PyTorch pipeline 节点。	`Python 3.12`
		CUDA 基础镜像
		`PyTorch 2.9.1`
		`torchvision 0.24.1`
		`TensorBoard 2.20.x`
		数据科学运行时依赖

运行时镜像	描述	主要包

PyTorch LLM Compressor CUDA
alaudadockerhub/odh-pipeline-runtime-pytorch-llmcompressor-cuda-py312-ubi9 ↗
	适用于 NVIDIA GPU 节点上的 LLM 压缩和评估 pipeline 节点。	`Python 3.12`
		CUDA 基础镜像
		`PyTorch 2.9.1`
		`torchvision 0.24.1`
		`LLM Compressor 0.9.0.2`
		`transformers 4.57.3`
		`datasets 4.4.1`
		`accelerate 1.12.0`
		`compressed-tensors 0.13.0`
		`lm-eval 0.4.x`

打开 **JupyterLab**

1. 登录 Alauda AI。
2. 进入 **Workbench**。
3. 打开一个现有的 JupyterLab workbench，或使用包含 Elyra 的 JupyterLab 镜像创建一个新的 workbench。
4. 等待 workbench 状态变为 `Running`。
5. 单击 **Connect** 打开 JupyterLab。

JupyterLab 打开后，你可以选择在 JupyterLab 终端中验证 KFP SDK 版本：

```
python -c "import kfp; print(kfp.__version__)"
```

版本应为 `2.x`。

创建两个演示 **notebook**

这个 hello world pipeline 使用你直接在 JupyterLab 中创建的三个文件：

文件	用途
`01-hello.ipynb`	第一个 notebook 节点。它会打印一条消息并完成执行。
`02-world.ipynb`	第二个 notebook 节点。只有在第一个 notebook 成功后才会运行。
`hello-two-nodes.pipeline`	连接这两个 notebook 的 Elyra pipeline 画布。

在 JupyterLab 文件浏览器中，创建一个名为 `hello-two-nodes` 的文件夹。

在该文件夹中，创建第一个名为 `01-hello.ipynb` 的 notebook，并添加以下单元：

```
print("hello from the first Elyra node")
message = "first notebook completed"
print(message)
```

创建第二个名为 `02-world.ipynb` 的 notebook，并添加以下单元：

```
print("hello from the second Elyra node")
print("this notebook runs after the first notebook succeeds")
```

在 JupyterLab 中分别运行这两个 notebook 一次，以确认它们可以在本地执行且没有语法错误。

在 **Elyra** 中创建 **pipeline**

JupyterLab 打开后，请等待主启动器页面和 Elyra 面板加载完成。在基于浏览器的验证中，JupyterLab 有时需要几秒钟之后，**Pipeline Editor** 和 **Runtime Images** 面板才可用。

打开 **Pipeline Editor**

1. 在 JupyterLab 文件浏览器中打开 `hello-two-nodes` 文件夹。
2. 打开 **Launcher** 选项卡。如果 launcher 不可见，请单击 **File** > **New Launcher**。

3. 单击 **Pipeline Editor**。JupyterLab 会打开一个新的未命名 Elyra pipeline 画布。
4. 将空 pipeline 另存为 `hello-two-nodes.pipeline`，并保存在 `hello-two-nodes` 文件夹中。

添加 **Notebook** 节点

1. 将 `01-hello.ipynb` 从 JupyterLab 文件浏览器拖到 pipeline 画布上。
2. 将 `02-world.ipynb` 从文件浏览器拖到同一画布上。
3. 将节点从左到右排列，便于阅读执行顺序。
4. 将 `01-hello.ipynb` 的输出端口连接到 `02-world.ipynb` 的输入端口。该边表示依赖关系，使第二个 notebook 只会在第一个 notebook 成功后启动。

配置节点属性

1. 单击 `01-hello.ipynb` 节点。
2. 打开节点属性面板。根据你的 JupyterLab 布局，属性面板可能显示在画布右侧，或者通过节点上下文菜单打开。
3. 确认节点文件指向 `01-hello.ipynb`。
4. 将 **Runtime Image** 设置为适用于你平台的 Elyra pipeline 运行时镜像之一，例如：

```
Runtime | Minimal | CPU | Python 3.12
```

5. 如果该节点有 CPU、memory、GPU、环境变量、输入文件或输出文件字段，请在此 hello world 示例中保留默认值。
6. 单击 `02-world.ipynb` 节点，并重复相同的 **Runtime Image** 设置。
7. 再次保存 pipeline 文件。

> **TIP**
>
> 对于这个 hello world 示例，连接仅表示执行依赖。如果你的第二个 notebook 必须读取第一个 notebook 创建的文件，请在 Elyra 中配置节点文件依赖和输出，以便文件通过 pipeline 产物存储进行传递。

将 pipeline 提交到 KFP

1. 在 Elyra pipeline editor 中，单击 **Run Pipeline**。
2. 在运行对话框中，将 **Runtime Platform** 设置为 **Kubeflow Pipelines**。
3. 选择管理员提供的 KFP v2 运行时配置，例如 `MLOps KFP`。
4. 输入 pipeline 名称，例如 `hello-two-nodes`。
5. 输入运行名称，或保留自动生成的运行名称。
6. 选择一个现有 experiment，或创建一个新的 experiment，例如 `elyra-demo`。
7. 检查两个 notebook 节点的运行时镜像值。
8. 单击 **OK** 或 **Submit**。

提交成功后，Elyra 会显示类似 `Job submission to Pipelines succeeded` 的对话框。

> **NOTE**
>
> Elyra 是用于 pipeline 编写和提交的 UI。提交之后，它不会在 JupyterLab 画布中提供持久的运行历史视图。请使用 Kubeflow Pipelines 查看执行状态。

在 Kubeflow Pipelines 中验证运行结果

1. 打开 Kubeflow UI。
2. 选择你的 workbench 所在的相同 namespace。
3. 进入 **Pipelines** > **Runs**。
4. 打开 `hello-two-nodes` 的最新运行记录。
5. 在 **Graph** 选项卡中，确认两个 notebook 节点按顺序显示。
6. 单击每个节点并查看日志。你应该会看到：

```
hello from the first Elyra node
hello from the second Elyra node
```

7. 等待运行状态变为 `Succeeded`。

根据你的 Kubeflow route，运行详情 URL 可能类似以下其中一种：

```
https://<kubeflow-domain>/_/pipeline/#/runs/details/<run-id>
https://<kubeflow-domain>/_/pipeline/?ns=<your-namespace>#/runs/details/<
run-id>
```

你也可以检查该 namespace 中的 Kubernetes 资源：

```
kubectl get pod -n <your-namespace>
kubectl get workflow -n <your-namespace>
```

对于 KFP v2，运行中出现 driver 和 implementation pods 是正常的。用户 notebook 代码在 implementation 容器中运行。

故障排查

namespace 在 Kubeflow 中不可见

该 namespace 必须关联到 Kubeflow `Profile`，并且用户必须绑定到该 namespace。请按照安装 Kubeflow Plugins 中的 namespace 绑定步骤进行操作。

Elyra 未显示 KFP 运行时配置

请使用包含 Elyra 的 JupyterLab workbench 镜像。然后请管理员确认 Elyra 运行时元数据已挂载到 workbench 中，并且该运行时指向你的 KFP endpoint 和对象存储。

Elyra 中没有可用的运行时镜像

请管理员创建或更新 `pipeline-runtime-images` ConfigMap，并通过 `WorkspaceKind` 将其挂载到 `/opt/app-root/pipeline-runtimes`。修改 ConfigMap 后请重启 workbench。

在 notebook 代码开始执行前运行失败

检查 namespace 侧的 KFP 对象：

```
kubectl get secret -n <your-namespace> mlpipeline-minio-artifact
kubectl get configmap -n <your-namespace> kfp-launcher
kubectl get configmap -n <your-namespace> metadata-grpc-configmap
```

同时检查该 namespace 中失败 pod 的日志。与存储相关的失败通常指向对象存储 endpoint、bucket、凭据或 `kfp-launcher` 配置。与元数据上报相关的失败通常指向 `metadata-grpc-configmap` 或 metadata gRPC 服务。

Elyra 的成功对话框已关闭

直接打开 Kubeflow Pipelines，并检查同一 namespace 下的 **Runs**。Elyra 在成功对话框关闭后不会保留持久的运行链接。

使用可复用的 Kubeflow Pipeline 组件

下一版 Alauda AI 将为 Kubeflow Pipelines 2.16.1 打包来自镜像 `pipelines-components` 项目的可复用组件。该 KFP 版本将由 Alauda `kfp-operator` v26.3.1 和 DSPO v2.15.2 提供。任一安装都会预加载两个 beta 版 AutoGluon pipeline：

Pipeline	用途	输入数据
`autogluon-tabular-training-pipeline`	回归、二分类和多分类	带有目标列的 CSV
`autogluon-timeseries-training-pipeline`	预测一个或多个时间序列	带有 ID、timestamp 和数值目标列的 CSV 或 Parquet

托管 pipeline 是完成训练运行的最短路径。你也可以克隆该镜像，并在需要自定义 pipeline DAG 时导入单独的组件。

> **NOTE**
>
> KFP 2.16.1 服务和可复用组件镜像支持 `linux/amd64` 和 `linux/arm64`。在 arm64 上，DSPO 不需要额外设置，因为它不会部署可选的、仅限 amd64 的 metadata writer。对于 Alauda `kfp-operator`，请设置 `kfpMetadataWriter.enabled: false`。请参见 arm64 部署说明。

目录

前提条件

创建输入存储 Secret

运行托管 tabular pipeline

运行托管 time-series pipeline

基于镜像组合自定义 pipeline

故障排查

前提条件

要求	详细信息
Kubeflow Pipelines 2.16.1	将在下一版 Alauda AI 中随 Alauda `kfp-operator` v26.3.1 或 DSPO v2.15.2 一起提供。集群上只能安装一个 pipeline operator。
支持的架构	`linux/amd64` 或 `linux/arm64`。在 arm64 上，请遵循上面链接的 operator 专用配置。
S3 兼容的输入存储	数据加载器会直接从 S3 读取输入对象。
S3 凭据 Secret	在执行 pipeline 运行的命名空间中创建它。
Workspace 存储	托管 pipeline 会申请一个 12 GiB 的 `ReadWriteOnce` workspace PVC。安装时必须配置可用的默认存储类。
计算资源	数据加载器申请 2 CPU 和 8 GiB。使用 `preset=speed` 训练时申请 4 CPU 和 16 GiB；使用 `preset=balanced` 时申请 8 CPU 和 32 GiB。

输入 bucket 可以不同于为 KFP artifact store 配置的对象存储。下面的 Secret 用于对数据加载器进行输入 bucket 身份验证；KFP 会单独为其 launcher 提供 artifact-store 凭据。

创建输入存储 **Secret**

在 pipeline 运行命名空间中创建该 Secret：

```
NS=my-pipeline-namespace

kubectl -n "$NS" create secret generic pipeline-input-s3 \
  --from-literal=AWS_ACCESS_KEY_ID='<access-key>' \
  --from-literal=AWS_SECRET_ACCESS_KEY='<secret-key>' \
  --from-literal=AWS_S3_ENDPOINT='https://s3.example.com' \
  --from-literal=AWS_DEFAULT_REGION='us-east-1'
```

请保留 `AWS_S3_ENDPOINT` 中的 URI scheme。该组件会将此值作为 endpoint URL 传递给 S3 client。

在开始运行之前，将训练文件上传到 bucket。例如，下面的 tabular 输入包含两个特征和一个回归标签：

```
feature_a,feature_b,price
1.0,2.0,10.5
2.0,3.0,20.5
3.0,4.0,30.5
```

在去除重复行、无效标签和其他不可用值之后，tabular 数据必须至少包含 100 行有效数据。

运行托管 **tabular pipeline**

1. 打开 **Kubeflow Pipelines**，然后选择 **Pipelines**。
2. 搜索 `autogluon-tabular-training-pipeline` 并选择其最新版本。
3. 选择 **Create run**，选择包含该 Secret 的命名空间，并设置以下输入：

输入	示例	说明
`train_data_secret_name`	`pipeline-input-s3`	S3 凭据 Secret
`train_data_bucket_name`	`training-data`	输入 bucket
`train_data_file_key`	`tabular/housing.csv`	CSV 对象 key
`label_column`	`price`	目标列

输入	示例	说明
`task_type`	`regression`	`regression`、`binary` 或 `multiclass`
`top_n`	`3`	保留的模型数量，范围为 1 到 10
`positive_class`	empty	二分类的可选正样本标签
`eval_metric`	empty	为空时，回归选择 `r2`，分类选择 `accuracy`
`preset`	`speed`	`speed` 或资源更高的 `balanced` 等级

4. 启动运行，并在运行图中跟踪数据加载器和模型训练任务。

该 pipeline 会对源数据进行采样和拆分，训练并排序 AutoGluon 模型，然后重新拟合表现最好的 `top_n` 个模型。训练拆分数据会通过运行的 workspace PVC 共享。测试拆分、leaderboard、模型 predictor、指标以及生成的推理 notebook 都是持久化到已配置 artifact store 中的 KFP artifact。

运行托管 **time-series pipeline**

选择 `autogluon-timeseries-training-pipeline`，并提供上文所述的通用存储输入。其 time-series 专用输入如下：

输入	示例	说明
`target`	`sales`	要预测的数值
`id_column`	`product_id`	标识每个独立序列
`timestamp_column`	`date`	可解析的观测时间戳
`known_covariates_names`	`["is_holiday", "promo"]`	预测区间内已知的可选列
`prediction_length`	`14`	要预测的时间步数

输入	示例	说明
`top_n`	`3`	保留的模型数量
`eval_metric`	`mean_absolute_scaled_error`	AutoGluon time-series 排名指标
`preset`	`speed`	`speed` 或 `balanced`

加载器会对 ID/timestamp 对进行去重，并为每个序列执行时间拆分。它会将工作中的训练拆分写入 workspace PVC，并将测试拆分和已训练模型写入 artifact store。

基于镜像组合自定义 **pipeline**

`kfp-components` 包未发布到 PyPI。请克隆 Alauda 镜像，并在 Python 3.11 或更高版本的虚拟环境中安装它。将 SDK 锁定到 Alauda AI 发布的版本。将 `<pipelines-components-repo-url>` 替换为 Alauda 为该环境提供的仓库地址：

```
git clone <pipelines-components-repo-url>
cd pipelines-components

python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install 'kfp==2.16.1' 'kfp-kubernetes==2.16.1'
python -m pip install --no-deps -e .
```

此示例导入 tabular 数据加载器，并将其用于一个更小的自定义 pipeline。请将镜像替换为镜像到你集群中的 AutoML runtime：

```
from kfp import compiler, dsl, kubernetes
from kfp_components.components.data_processing.automl.tabular_data_loader
import (
    automl_data_loader,
)

AUTOML_IMAGE = "<registry>/mlops/kubeflow/odh-automl:v1.11.0.post.1"


@dsl.pipeline(
    name="custom-tabular-loader",
    pipeline_config=dsl.PipelineConfig(
        workspace=dsl.WorkspaceConfig(
            size="12Gi",
            kubernetes=dsl.KubernetesWorkspaceConfig(
                pvcSpecPatch={"accessModes": ["ReadWriteOnce"]}
            ),
        )
    ),
)
def custom_tabular_loader(
    secret_name: str,
    bucket: str,
    object_key: str,
    label: str,
    task_type: str = "regression",
):
    loader = automl_data_loader(
        bucket_name=bucket,
        file_key=object_key,
        workspace_path=dsl.WORKSPACE_PATH_PLACEHOLDER,
        label_column=label,
        task_type=task_type,
    )
    loader.set_container_image(AUTOML_IMAGE)
    kubernetes.use_secret_as_env(
        loader,
        secret_name=secret_name,
        secret_key_to_env={
            "AWS_ACCESS_KEY_ID": "AWS_ACCESS_KEY_ID",
            "AWS_SECRET_ACCESS_KEY": "AWS_SECRET_ACCESS_KEY",
            "AWS_S3_ENDPOINT": "AWS_S3_ENDPOINT",
            "AWS_DEFAULT_REGION": "AWS_DEFAULT_REGION",
```

```
        },
    )


compiler.Compiler().compile(
    pipeline_func=custom_tabular_loader,
    package_path="custom-tabular-loader.yaml",
)
```

从 Kubeflow Pipelines UI 上传 `custom-tabular-loader.yaml`，或者使用 KFP SDK 提交它。在使用其他资产之前，请先查看镜像中的组件 `README.md`、`metadata.yaml` 和测试；该仓库包含具有不同外部服务要求的 alpha 和 beta 资产。

故障排查

- 找不到 **Secret**： Secret 是按命名空间作用域生效的。请在与运行相同的命名空间中创建输入 Secret。
- **Workspace PVC** 一直处于 **Pending**： 配置一个支持 `ReadWriteOnce` 的默认存储类，或者请平台管理员配置 KFP workspace 存储类。
- 运行一直处于 **Pending**： 将集群可用资源与 前提条件 中的 CPU 和内存请求进行比较。请从 `preset=speed` 开始。
- 无法读取输入对象： 检查 endpoint scheme、bucket 和对象 key、凭据、region、网络策略以及 S3 server certificate。
- **Pod** 完成后缺少 **artifact**： pipeline 输入 Secret 未配置 KFP 持久化。请与平台管理员一起检查 KFP 安装的 artifact-store endpoint 和凭据。

Kubeflow Pipelines 执行和存储行为

由 Alauda `kfp-operator` v26.3.1 或 Data Science Pipelines Operator (DSPO) v2.15.2 安装的 Kubeflow Pipelines 2.16.1 支持任务缓存、`dsl.ParallelFor`、类型化 KFP artifacts，以及在 amd64 和 arm64 集群上的 S3 兼容 artifact 持久化。这些机制都是 KFP 的功能；operator 负责配置 API server、workflow controller、cache server 和 artifact store 来实现它们。

目录

功能概览

安全地缓存任务

使用 `ParallelFor` 运行任务

持久化并传递类型化 artifact

运行验证 pipeline

在 arm64 集群上部署

`kfp-operator`

Data Science Pipelines Operator (DSPO)

平台兼容性和数据库后端

功能概览

机制	KFP 2.16.1	行为
任务缓存	支持	相同的任务执行可以复用，并在重复运行中报告为 `SKIPPED`。
`dsl.ParallelFor`	支持	每个循环项都会成为一个独立任务；`parallelism` 用于限制并发迭代数。
类型化 artifacts	支持	`dsl.Output[dsl.Dataset]` 和 `dsl.Input[dsl.Dataset]` 会创建显式的生产者到消费者依赖关系。
artifact 持久化	支持	launcher 会将 artifact 文件和元数据上传到已配置的 S3 兼容存储，并在消费者侧下载它们。

下面链接的验证 pipeline 是在 x86 集群上使用 DSPO 运行的。它生成了一个 `Dataset`，由三个 `ParallelFor` 任务读取，然后再次提交完全相同的 pipeline。三个消费者都成功了，第二次运行中的四个 executor task 因为缓存结果被复用而被报告为 `SKIPPED`。在 workflow pod 完成后，该 dataset 对象仍然存在于 S3 中。

> **NOTE**
>
> 受管的 AutoGluon pipelines 会刻意为其主任务禁用缓存。它们的源对象 key 可能引用可变数据，并且每次运行都必须发布新的进度和 model artifacts。缓存对你自己的 pipelines 仍然可用。

安全地缓存任务

默认情况下，运行会启用缓存。提交编译后的 package 时也可以进行设置：

```
client.create_run_from_pipeline_package(
    pipeline_file="pipeline.yaml",
    arguments={"dataset_version": "2026-07-23"},
    enable_caching=True,
)
```

对于具有外部副作用或可变输入的任务，请禁用缓存：

```
task = publish_report(...)
task.set_caching_options(False)
```

KFP 会根据 component 及其声明的输入计算 cache key。它无法检测未更改的 S3 object key、URL、database query 或 image tag 背后的内容是否发生了变化。当需要新的外部内容使 cache 失效时，请传入不可变的 object version、digest 或显式的 version 参数。

使用 `ParallelFor` 运行任务

当同一个 component 可以处理彼此独立的项时，请使用 `dsl.ParallelFor`：

```
from kfp import dsl


@dsl.pipeline(name="parallel-evaluation")
def evaluation_pipeline():
    dataset = produce_dataset()
    with dsl.ParallelFor(
        items=["accuracy", "precision", "recall"],
        parallelism=3,
    ) as metric:
        evaluate(dataset=dataset.outputs["dataset"], metric=metric)
```

controller 会为每个项创建一个 `evaluate` 任务。当 namespace quota、node 容量或外部 API 无法同时承载所有迭代时，请将 `parallelism` 设为低于项数的值。

持久化并传递类型化 artifact

将文件写入输出 artifact 提供的路径。不要自行构造本地路径并期望它能在 producer pod 中保留下来：

```python
from kfp import dsl


@dsl.component(base_image="python:3.12-slim")
def produce_dataset(dataset: dsl.Output[dsl.Dataset]):
    from pathlib import Path

    output = Path(dataset.path)
    output.parent.mkdir(parents=True, exist_ok=True)
    output.write_text("value\n42\n", encoding="utf-8")
    dataset.metadata["rows"] = 1


@dsl.component(base_image="python:3.12-slim")
def consume_dataset(dataset: dsl.Input[dsl.Dataset]):
    from pathlib import Path

    print(Path(dataset.path).read_text(encoding="utf-8"))
```

将 `producer.outputs["dataset"]` 传递给 consumer 会建立依赖关系。KFP launcher 会把 producer 的文件上传到 artifact store，并在 consumer pod 中将其 materialize 到 `dataset.path`。

这与 KFP workspace PVC 不同：

存储	最适合	生命周期
类型化 KFP artifact	声明的 component 输入和输出、lineage、以及运行结束后仍需保留的结果	根据其保留策略持久化到已配置的 object store 中
Workspace PVC	在一次运行中由多个任务共享的大型中间文件	为运行创建；生命周期遵循 KFP workspace 处理方式
Container filesystem	单个任务内部的临时 scratch 数据	pod 移除后丢失

运行验证 **pipeline**

下载 `verify-kfp-features.py` ↗ ，并在可以访问 KFP API 的 Workbench 或其他环境中运行它：

```
python -m pip install 'kfp==2.16.1'

export KFP_ENDPOINT='http://<kfp-api-service>:8888'
export KFP_NAMESPACE='my-pipeline-namespace'
python verify-kfp-features.py
```

该脚本会编译一个 producer 和三个并行 consumer，使用相同参数提交两次，并在以下任一条件不满足时失败：

- 第一次运行成功；
- 恰好有三个 `verify-dataset` 任务成功；
- 每个 consumer 都读取了 producer 的 Dataset 内容；并且
- 第二次运行将缓存任务报告为 `SKIPPED`。

最终输出中，每种机制都会有一行 `PASS`。管理员可以在 pod 结束后，通过列出 KFP artifact bucket 中的 run prefix 来独立确认持久化情况。

在 arm64 集群上部署

Alauda KFP 2.16.1 版本为 `linux/amd64` 和 `linux/arm64` 都打包了 pipeline service 和 runtime images。可选的 `kfp-metadata-writer` image 仍然仅支持 amd64。DSPO 不部署此组件；Alauda `kfp-operator` 必须在 arm64 上将其禁用。

`kfp-operator`

使用 Alauda `kfp-operator` v26.3.1 时，请先从 OperatorHub 正常安装 operator，并在 reconciliation 之前将 `KubeflowPipelines` custom resource 中可选的 metadata writer 设为 `false`：

```yaml
apiVersion: operator.alauda.io/v1
kind: KubeflowPipelines
metadata:
  name: kubeflowpipelines
spec:
  global:
    images:
      kfpMetadataWriter:
        enabled: false
```

保留 release sample 中其余的 `spec.global.images` 值不变。`support_arm` image 字段仅控制打包/image 迁移；它不会禁用 Deployment。请在 reconciliation 后验证 writer 不存在：

```
kubectl -n kubeflow get deployment metadata-writer
# Error from server (NotFound) is expected
```

Data Science Pipelines Operator (DSPO)

DSPO v2.15.2 中的 V2 reconciler 不会部署 `kfp-metadata-writer`。无需 arm64 覆盖。使用常规 V2 设置创建 DSPA，并且不要添加 MLMD writer 字段：

```yaml
apiVersion: datasciencepipelinesapplications.opendatahub.io/v1
kind: DataSciencePipelinesApplication
metadata:
  name: sample
  namespace: data-science-project
spec:
  dspVersion: v2
  mlmd:
    deploy: true
```

验证该 namespace 包含 MLMD gRPC/Envoy 资源，但没有 `metadata-writer` Deployment。DSPO 和 `kfp-operator` 彼此互斥；每个集群只能选择一种安装模型。

平台兼容性和数据库后端

以下结果适用于与 KFP 2.16.1 对齐的 Alauda 构建：

安装	amd64	arm64	MySQL	Postg
DSPO v2.15.2	支持；smoke 和机制测试通过	支持；DSPA readiness 和 KFP v2 SDK smoke 通过	支持	不支持
Alauda `kfp-operator` v26.3.1	支持	在 `kfpMetadataWriter.enabled` 为 `false` 时支持；已验证 KFP API 和 Argo workflow 执行	支持	不支持

这些 release 中所有必需的 runtime images 都有 amd64 和 arm64 manifest，`kfp-metadata-writer` 除外。只有 arm64 的 Alauda `kfp-operator` 安装需要禁用该组件；它不是 DSPO deployment 的一部分。

当前这两个 operator 都只暴露 MySQL 连接设置。DSPO 使用 MySQL driver 执行外部数据库健康检查，并渲染 `DB_DRIVER_NAME=mysql`。`kfp-operator` chart 会渲染 `dbType=mysql`、`DBDriverName=mysql` 和 `DBCONFIG_MYSQLCONFIG_*`。将任一 operator 指向 PostgreSQL endpoint 都不会切换其 driver，也不会生成可用的 KFP 安装。在 operator 接口中接入 PostgreSQL driver 选择以及上游 PostgreSQL deployment 补丁之前，请继续使用 MySQL。

使用 Kueue 调度 Alauda DevOps Pipelines

本页面介绍在运行 Alauda DevOps Pipelines（Tekton Pipelines）时，如何利用 Alauda Build of Kueue 的调度和资源管理能力。

目录

前提条件

操作步骤

前提条件

- 已安装 Alauda DevOps Pipelines。
- 已安装 Alauda Build of Kueue。
- 已安装 Alauda Build of Hami（用于演示 vGPU）。
- Alauda Container Platform Web CLI 已与您的集群建立通信。

操作步骤

1. 在 Alauda Container Platform 中创建一个项目和命名空间，例如项目名为 `test`，命名空间名为 `test-1`。
2. 运行以下命令创建资源：

```
cat <<EOF| kubectl create -f -
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: cluster-queue
spec:
  namespaceSelector: {}
  resourceGroups:
  - coveredResources: ["cpu", "memory", "pods", "nvidia.com/gpualloc",
"nvidia.com/total-gpucores", "nvidia.com/total-gpumem"]
    flavors:
    - name: "default-flavor"
      resources:
      - name: "cpu"
        nominalQuota: 9
      - name: "memory"
        nominalQuota: 36Gi
      - name: "pods"
        nominalQuota: 5
      - name: "nvidia.com/gpualloc"
        nominalQuota: "2"
      - name: "nvidia.com/total-gpucores"
        nominalQuota: "50"
      - name: "nvidia.com/total-gpumem"
        nominalQuota: "20000"
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: default-flavor
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  namespace: test-1
  name: test
spec:
  clusterQueue: cluster-queue
EOF
```

3. 通过 Web CLI 或 UI 在 Alauda Container Platform 中创建一个 `Pipeline` 资源：

```yaml
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: test
  namespace: test-1
spec:
  tasks:
    - name: run-script
      taskSpec:
        description: test
        metadata: {}
        spec: null
        steps:
          - computeResources:
              limits:
                cpu: "2"
                memory: 2Gi
                nvidia.com/gpualloc: "2"
                nvidia.com/gpucores: "50"
                nvidia.com/gpumem: 8k
              requests:
                cpu: "1"
                memory: 1Gi
            image: nvidia/cuda:11.0-base
            imagePullPolicy: IfNotPresent
            name: run-script
            script: |
              #!/bin/sh
              nvidia-smi
            securityContext:
              allowPrivilegeEscalation: false
              capabilities:
                drop:
                  - ALL
              runAsNonRoot: true
              runAsUser: 65532
              seccompProfile:
                type: RuntimeDefault
      timeout: 30m0s
```

4. 通过 Web CLI 或 UI 在 Alauda Container Platform 中创建一个 `PipelineRun` 资源：

```yaml
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  annotations:
    chains.tekton.dev/signed: "true"
    integrations.tekton.dev/integrations: |
      []
  generateName: test-
  labels:
    tekton.dev/pipeline: test
    kueue.x-k8s.io/queue-name: test (1)
  namespace: test-1
spec:
  pipelineRef:
    name: test (2)
  taskRunTemplate:
    podTemplate:
      securityContext:
        fsGroup: 65532
        fsGroupChangePolicy: OnRootMismatch
    serviceAccountName: default
  timeouts:
    pipeline: 1h0m0s
```

(1) `kueue.x-k8s.io/queue-name: test` 标签：指定管理 `PipelineRun` 的所有 pod 的 `LocalQueue`。

(2) `spec.pipelineRef.name`：指定 `PipelineRun` 引用的 `Pipeline` 资源。

5. 查看 `PipelineRun` 的 pod：

```bash
kubectl -n test-1 get pod | grep test
```

您会看到该 pod 处于 `SchedulingGated` 状态：

```
test-dw4q7-run-script-pod   0/1     SchedulingGated   0          13s
<none>   <none>   <none>           <none>
```

6. 更新 `nvidia.com/total-gpucores` 配额：

```
cat <<EOF| kubectl apply -f -
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: cluster-queue
spec:
  namespaceSelector: {}
  resourceGroups:
  - coveredResources: ["cpu", "memory", "pods", "nvidia.com/gpualloc",
"nvidia.com/total-gpucores", "nvidia.com/total-gpumem"]
    flavors:
    - name: "default-flavor"
      resources:
      - name: "cpu"
        nominalQuota: 9
      - name: "memory"
        nominalQuota: 36Gi
      - name: "pods"
        nominalQuota: 5
      - name: "nvidia.com/gpualloc"
        nominalQuota: "2"
      - name: "nvidia.com/total-gpucores"
        nominalQuota: "100"
      - name: "nvidia.com/total-gpumem"
        nominalQuota: "20000"
EOF
```

您会看到该 pod 处于 `Running` 状态：

```
test-dw4q7-run-script-pod   1/1     Running   0          13s   <none>
<none>   <none>           <none>
```

分布式工作负载

可从工作台或 CLI 运行分布式 Ray 和 Spark 工作负载。

CodeFlare SDK 教程

前提条件

演示 Notebook

操作步骤

运行 Spark Application

前提条件

1. 创建命名空间和 Spark RBAC

2. 提交 SparkApplication

3. 监控应用

4. 验证结果

5. 清理

调度周期性作业

常见的 SparkApplication 字段

CodeFlare SDK 教程

本教程演示如何使用 CodeFlare SDK 向现有 RayCluster 提交 RayJob。您将学习如何启动 Ray 集群、验证其状态、提交 RayJob，以及管理集群生命周期。

目录

前提条件

演示 Notebook

操作步骤

步骤 1：创建 RayCluster

步骤 2：验证集群状态

步骤 3：提交 RayJob

步骤 4：监控 RayJob

步骤 5：清理

前提条件

- 您已在数据科学集群中安装了 `Alauda Build of KubeRay Operator` 集群插件，参见 安装 Alauda Build of KubeRay Operator。
- 您可以访问 Alauda AI 中的一个命名空间，创建一个 workbench，并且该 workbench 正在运行一个包含 CodeFlare SDK 的默认 workbench 镜像，例如 **Standard Data Science** notebook。有关创建 workbench 的信息，请参见 创建 Workbench。

- 您已登录 Alauda AI，启动了您的 workbench，并登录到 JupyterLab。

演示 Notebook

下载演示 Jupyter notebook，以便跟随本教程进行操作：

- **Notebook**: CodeFlare SDK RayJob Demo

单击 JupyterLab 页面上的向上箭头按钮，上传已下载的 notebook 文件。

操作步骤

在 JupyterLab 中打开演示 notebook，并按照以下步骤操作。每个步骤都对应 notebook 中的一个部分。

步骤 1：创建 RayCluster

运行前两个代码单元，导入 CodeFlare SDK，并使用 `ClusterConfiguration` API 创建一个 Ray 集群。在运行之前，请将 `image` 参数更新为与您的硬件架构兼容的 Ray 集群镜像。如果您的集群无法直接访问互联网，请使用内部镜像仓库中的镜像。

`cluster.apply()` 调用会提交集群配置，并等待其就绪。您可以根据需要调整 `timeout` 值。

步骤 2：验证集群状态

运行 `cluster.status()` 单元。如果集群没有立即启动，请再运行该单元几次，直到看到其处于 `Ready` 状态。

步骤 3：提交 RayJob

运行 `RayJob` 单元，向正在运行的集群创建并提交一个作业。请注意以下参数：

- `job_name`：您的 RayJob 的唯一名称。

- `cluster_name`：必须与现有 RayCluster 的名称一致。
- `entrypoint`：要执行的命令。在标准实践中，这里通常应指向一个 Python 训练脚本，而不是演示中使用的内联命令。

步骤 4：监控 RayJob

运行 `rayjob.status()` 单元。此函数会根据 RayJob 的当前状态输出不同的表格。您可以多次重新运行该单元，以观察状态变化。

步骤 5：清理

作业完成后，运行 `cluster.down()` 关闭集群。

> **WARNING**
>
> 为了实现最佳资源管理，当 Ray 集群不再使用时，您应始终将其删除。

运行 Spark Application

本指南演示如何使用 `SparkApplication` 自定义资源提交 Apache Spark 作业、监控其直至完成，并检查结果。示例使用 Spark 镜像中自带的内置 **Spark Pi** 示例。

目录

前提条件

1. 创建命名空间和 Spark RBAC
2. 提交 SparkApplication
3. 监控应用
4. 验证结果
5. 清理

调度周期性作业

常见的 SparkApplication 字段

前提条件

- 已安装 Alauda Build of Spark Operator，且其 CSV 报告为 `Succeeded` — 参见 安装。
- 已拥有目标集群的 `kubectl` 访问权限。

1. 创建命名空间和 **Spark RBAC**

在 cluster 模式下，Spark **driver** 会创建并管理其 **executor** pod，因此 driver 的 `ServiceAccount` 需要具备在其命名空间中管理 pod（以及 Spark 使用的 services / configmaps）的权限。

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: spark-demo
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: spark
  namespace: spark-demo
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: spark-driver
  namespace: spark-demo
rules:
  - apiGroups: [""]
    resources: ["pods", "services", "configmaps", "persistentvolumeclaim
s"]
    verbs: ["create", "get", "list", "watch", "delete", "deletecollectio
n", "patch", "update"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: spark-driver
  namespace: spark-demo
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: spark-driver
subjects:
  - kind: ServiceAccount
    name: spark
    namespace: spark-demo
```

保存为 `spark-rbac.yaml` 并应用：

```bash
kubectl apply -f spark-rbac.yaml
```

2. 提交 SparkApplication

```yaml
apiVersion: sparkoperator.k8s.io/v1beta2
kind: SparkApplication
metadata:
  name: spark-pi
  namespace: spark-demo
spec:
  type: Scala
  mode: cluster
  image: docker.io/library/spark:4.0.1 (1)
  imagePullPolicy: IfNotPresent
  mainClass: org.apache.spark.examples.SparkPi
  mainApplicationFile: "local:///opt/spark/examples/jars/spark-examples.j
ar" (2)
  arguments: ["100"] (3)
  sparkVersion: "4.0.1"
  restartPolicy:
    type: Never
  driver:
    coreRequest: "500m"
    coreLimit: "1000m"
    memory: 512m
    serviceAccount: spark (4)
    securityContext: (5)
      runAsUser: 185
      runAsGroup: 185
      runAsNonRoot: true
      allowPrivilegeEscalation: false
      capabilities: { drop: ["ALL"] }
      seccompProfile: { type: RuntimeDefault }
  executor:
    instances: 2 (6)
    coreRequest: "500m"
    coreLimit: "1000m"
    memory: 512m
    securityContext:
      runAsUser: 185
      runAsGroup: 185
      runAsNonRoot: true
      allowPrivilegeEscalation: false
      capabilities: { drop: ["ALL"] }
      seccompProfile: { type: RuntimeDefault }
```

1. `spec.image` — driver 和 executor 使用的 Apache Spark 运行时镜像。在 air-gapped 集群中，请使用平台 registry 中已重新定位的副本（记录在 CSV 的 `relatedImages` 中）或你们内部的镜像仓库，例如 `build-harbor.alauda.cn/mlops/spark:4.0.1`。
2. `spec.mainApplicationFile` — 应用制品的路径或 URL。`local://` 表示镜像内部的路径；Spark 示例 jar 随 Spark 镜像一同提供。
3. `spec.arguments` — 传递给应用的参数；对于 Spark Pi，这里是分区数。
4. `spec.driver.serviceAccount` — 必须引用第 1 步中创建的 `ServiceAccount`。
5. `spec.driver.securityContext` / `spec.executor.securityContext` — 为满足受限的 Pod Security Admission 策略所必需。Spark 镜像以 UID/GID `185` 运行。
6. `spec.executor.instances` — 要运行的 executor pod 数量。

保存为 `spark-pi.yaml` 并应用：

```
kubectl apply -f spark-pi.yaml
```

3. 监控应用

operator 会在 `.status.applicationState.state` 中报告进度，其状态会依次经过 `SUBMITTED` → `RUNNING` → `COMPLETED`。

```
# high-level status
kubectl get sparkapplication spark-pi -n spark-demo

# just the state field
kubectl get sparkapplication spark-pi -n spark-demo \
  -o jsonpath='{.status.applicationState.state}{"\n"}'

# driver and executor pods (driver pod is named <app-name>-driver)
kubectl get pods -n spark-demo
```

4. 验证结果

当应用到达 `COMPLETED` 时，检查 driver 日志中计算出的 Pi 值：

```
kubectl logs spark-pi-driver -n spark-demo | grep "Pi is roughly"
```

你应该会看到类似如下的输出：

```
Pi is roughly 3.1415...
```

5. 清理

```
kubectl delete sparkapplication spark-pi -n spark-demo
# optionally remove the demo namespace
kubectl delete namespace spark-demo
```

调度周期性作业

要按计划运行 Spark 应用，请将相同的应用 `spec` 包装在 `ScheduledSparkApplication` 中：

```yaml
apiVersion: sparkoperator.k8s.io/v1beta2
kind: ScheduledSparkApplication
metadata:
  name: spark-pi-scheduled
  namespace: spark-demo
spec:
  schedule: "@every 5m" (1)
  concurrencyPolicy: Forbid (2)
  successfulRunHistoryLimit: 3
  failedRunHistoryLimit: 1
  template: (3)
    type: Scala
    mode: cluster
    image: docker.io/library/spark:4.0.1
    imagePullPolicy: IfNotPresent
    mainClass: org.apache.spark.examples.SparkPi
    mainApplicationFile: "local:///opt/spark/examples/jars/spark-example
s.jar"
    arguments: ["100"]
    sparkVersion: "4.0.1"
    restartPolicy:
      type: Never
    driver:
      coreRequest: "500m"
      coreLimit: "1000m"
      memory: 512m
      serviceAccount: spark
    executor:
      instances: 2
      coreRequest: "500m"
      coreLimit: "1000m"
      memory: 512m
```

(1) `spec.schedule` — 用于控制运行何时启动的 cron 表达式（或 `@every <duration>`）。

(2) `spec.concurrencyPolicy` — `Allow`、`Forbid` 或 `Replace`；用于控制是否允许重叠运行。

(3) `spec.template` — 每个调度周期要运行的 `SparkApplication` spec。

查看该计划及其生成的运行实例：

```
kubectl get scheduledsparkapplication spark-pi-scheduled -n spark-demo
kubectl get sparkapplication -n spark-demo
```

常见的 SparkApplication 字段

字段	描述
`spec.type`	应用语言：`Scala`、`Java`、`Python` 或 `R`。
`spec.mode`	提交模式；使用 `cluster`。
`spec.image`	driver 和 executor 使用的 Spark 运行时镜像。
`spec.mainClass`	JVM（`Scala` / `Java`）应用的主类。
`spec.mainApplicationFile`	应用制品的路径或 URL（`local://`、`http(s)://`、`s3a://`，等等）。
`spec.arguments`	传递给应用的参数。
`spec.sparkVersion`	镜像中的 Spark 版本（`4.0.1`）。
`spec.sparkConf`	传递给应用的任意 `spark.*` 配置。
`spec.deps`	要获取的额外 `jars`、`files`、`pyFiles` 和 Maven 包坐标。
`spec.driver` / `spec.executor`	pod 设置：`cores` / `coreRequest` / `coreLimit`、`memory`、`serviceAccount`、`instances`（executor）、labels、env、volumes、node selectors、`securityContext`，等等
`spec.restartPolicy`	`Never`、`OnFailure` 或 `Always`，以及重试 / 回退设置。

完整 schema 请参见上游 Spark Operator API reference ↗。

使用 Kubeflow Model Registry

Kubeflow Model Registry 是一个用于管理机器学习模型、其版本以及相关元数据的中心仓库。它允许数据科学家发布模型、跟踪其来源，并协作进行模型开发。

目录

访问 Model Registry

注册模型

选项 1：使用 UI

选项 2：使用 Python Client

部署已注册模型

访问 Model Registry

1. 打开监控面板：登录 Kubeflow 中央监控面板。
2. **Model Registry**：在侧边栏中单击 **Model Registry**。这将带你进入当前 namespace 中已注册模型的列表。

> 注意：如果你看不到 Model Registry，请确保平台管理员已在你的 namespace 中部署了一个 Model Registry 实例。

注册模型

你可以通过用户界面或使用 Python client 以编程方式注册模型。

选项 1：使用 UI

1. 创建已注册模型：

   - 单击 **Register Model**。
   - **Model Name**：输入一个唯一名称（例如，`fraud-detection`）。
   - **Description**：添加描述。
   - **Version details**：可选地添加版本信息、标签和元数据。
   - **Model Location**：提供模型 artifact 的 S3/URI（例如，`s3://my-bucket/models/fraud-detection/v1/`）。
   - 单击 **Create**。

2. 创建版本：

   - 单击 **Registered Model** 旁边下拉菜单，并选择 **Register New Version**。
   - 输入版本名称、描述、元数据和 artifact URI。
   - 单击 **Register new version**。

选项 2：使用 Python Client

你可以直接在 Jupyter Notebook 中使用 `model-registry` Python client 注册模型。

前提条件：

- 安装 client：`python -m pip install model-registry=="0.3.5" kserve=="0.13"`
- 确保你可以访问 Model Registry service。如果在 Kubeflow Notebook 中运行，可以使用内部 service DNS（例如 `http://model-registry-service.<namespace>.svc:8080`）。

示例代码：

以下示例演示如何注册存储在 S3 中的模型。

```python
from model_registry import ModelRegistry

# 1. Connect to the Model Registry
# Replace with your actual Model Registry service host/port
# Inside cluster, typically: "http://model-registry-service.<namespace>.svc.cluster.local:8080"
registry = ModelRegistry(
    server_address="http://model-registry-service.kubeflow.svc.cluster.local",
    port=8080,
    author="your name",
    is_secure=False
)

# 2. Register a new Model
rm = registry.register_model(
    "iris",
    "s3://kfserving-examples/models/sklearn/1.0/model",
    model_format_name="sklearn",
    model_format_version="1",
    version="v1",
    description="Iris scikit-learn model",
    metadata={
        "accuracy": 3.14,
        "license": "BSD 3-Clause License",
    }
)

# 3. Retrieve model information
model = registry.get_registered_model("iris")
print("Registered Model:", model, "with ID", model.id)

version = registry.get_model_version("iris", "v1")
print("Model Version:", version, "with ID", version.id)

art = registry.get_model_artifact("iris", "v1")
print("Model Artifact:", art, "with ID", art.id)
```

部署已注册模型

一旦模型完成注册，你就可以使用 KServe 将其部署为 **InferenceService**。

要进行部署，通常需要模型 artifact 的 URI。你可以从 Registry UI 或通过 Python API 获取该 URI：

```
from kubernetes import client
import kserve

isvc = kserve.V1beta1InferenceService(
    api_version=kserve.constants.KSERVE_GROUP + "/v1beta1",
    kind=kserve.constants.KSERVE_KIND,
    metadata=client.V1ObjectMeta(
        name="iris-model",
        namespace=kserve.utils.get_default_target_namespace(),
        labels={
            "modelregistry/registered-model-id": model.id,
            "modelregistry/model-version-id": version.id,
        },
    ),
    spec=kserve.V1beta1InferenceServiceSpec(
        predictor=kserve.V1beta1PredictorSpec(
            model=kserve.V1beta1ModelSpec(
                storage_uri=art.uri,
                model_format=kserve.V1beta1ModelFormat(
                    name=art.model_format_name, version=art.model_format_
version
                ),
            )
        )
    ),
)
ks_client = kserve.KServeClient()
ks_client.create(isvc)
```

部署完成后，KServe controller 将从指定的 S3 URI 拉取模型，并启动 inference server。

实验跟踪

使用带认证和 RBAC 的 MLflow I

认证如何工作

前提条件

验证 token 方式（管理员）

从命令行获取 token（无需浏览器）

连接 SDK

选择 workspace

注册模型

替代方案：session cookie（无需平台更改

故障排查

Kubeflow Pipeline + MLflow 集成

适用范围

前提条件

组件如何访问 MLflow

完整示例：使用 MLflow 的训练 pipeline

上传并运行

在 Trainer v2 pipeline 中使用 MLflow

最佳实践

故障排查

使用 MLflow

前提条件

跟踪 LangChain

跟踪 OGX 应用

导出 OGX Open

查看 traces

故障排查

使用带认证和 RBAC 的 MLflow Python SDK

在 Alauda AI 中，MLflow Tracking Server 运行在单点登录和多租户之后：OAuth proxy 会对每个调用方进行身份验证，server 会按调用用户记录每次 run，并基于 Kubernetes RBAC 对其进行授权。本指南将标准 **MLflow Python SDK** 通过该 OAuth proxy 以你自己的身份进行访问，且无需浏览器，使用针对平台登录脚本化的 OAuth2 **authorization code** 流程（带 PKCE）——不使用 password grant，也绝不直接访问 MLflow container 端口。

有两种无需浏览器的方式来呈现你的身份；请选择其一：

- **Bearer token**（推荐）。从 CLI 或 Python 获取 Dex **id token**，并将其作为 `MLFLOW_TRACKING_TOKEN` 传入；使用 refresh token 进行续期。operator 默认启用此方式（请在下方验证设置）。
- **Session cookie**（无需平台更改）。驱动 proxy 自身的登录以获取其 `_oauth2_proxy` cookie，并将其附加到请求中。可直接在任何安装环境中使用，无需改动（见下文）。

目录

认证如何工作

前提条件

验证 token 方式（管理员）

从命令行获取 token（无需浏览器）

Python helper

Shell 等价实现（curl + openssl，无 Python 依赖）

连接 SDK

选择 workspace

注册模型

替代方案：session cookie（无需平台更改）

故障排查

认证如何工作

在你的 runs 前面有两层：

1. **OAuth proxy**（`oauth2-proxy`）对请求进行身份验证——可以是作为 `Authorization: Bearer …` 发送的 Dex **id token**（token 方式），也可以是其 `_oauth2_proxy` **session cookie**（cookie 方式）。
2. MLflow server 的 `kubernetes-auth` plugin 会从该凭证中读取你的身份，将其记录为 run 的 **owner**，并基于 workspace 中的 Kubernetes 权限对其进行授权。

client 始终通过 OAuth proxy 访问——切勿直接连接到 MLflow container 端口。

前提条件

- `mlflow` **3.13.0**（`pip install "mlflow==3.13.0"`），这是本次发布验证过的 client 版本。workspace 选择（`mlflow.set_workspace`）在此版本中可用。Python token helper 还会使用 `requests` 和 `cryptography`。
- 可访问目标 workspace 的平台 **username** 和 **password**（参见 MLflow Workspaces and Access Control）。这是一种普通的平台登录——不是 Kubernetes ServiceAccount——普通用户账号即可。对于共享或自动化使用场景（pipeline、headless job），建议使用专用的、非个人的账号，而不是某个个人登录，因为凭证/token 会存储在 `Secret` 中，并且每次 run 都会记录为你所认证的身份。
- 平台的 **OAuth client id** 和 **secret**——MLflow proxy 使用的 client（由管理员提供）。在 Alauda 中，这是平台 auth client，例如 `alauda-auth`；其 secret 存储在 Kubernetes `Secret` 中（例如 `cpaas-oidc-secret`）。

验证 **token** 方式（管理员）

从 MLflow Operator v3.13.0 开始，workload-cluster OAuth proxy 默认接受 Dex id token。请确认 `MLflow` custom resource 保持启用该设置：

```
apiVersion: mlflow.alauda.io/v1alpha1
kind: MLflow
spec:
  auth:
    oauth:
      skipJwtBearerTokens: true
```

这是 workload cluster 上的 MLflow proxy，不是 平台的全局 auth server。不需要对 Dex 或 global-auth 进行任何更改：下面的登录使用的是平台 client 已经允许的 `authorization_code` grant。如果你是从较旧版本升级，或者替换了默认 OAuth 配置，请恢复 `skipJwtBearerTokens: true`。**cookie** 方式不需要任何设置。

从命令行获取 **token**（无需浏览器）

平台登录是一个 SSO 页面，但其 API 支持带 PKCE 的标准 OAuth **authorization code** 流程，因此你可以通过脚本完成登录——无需浏览器重定向。密码会使用 login service 的公钥（`/dex/pubkey`）进行 RSA 加密，和登录页面的处理方式完全一致，然后交换为 **id token**（以及用于 headless 续期的 **refresh token**）。

Python helper

```
import base64, hashlib, json, os, secrets
from urllib.parse import urlparse, parse_qs
import requests
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.serialization import load_pem_public_
key

PLATFORM      = os.environ["PLATFORM_ADDRESS"].rstrip("/")    # https://<
platform>
CLIENT_ID     = os.environ["DEX_CLIENT_ID"]                   # the MLflo
w proxy's client, e.g. alauda-auth
CLIENT_SECRET = os.environ["DEX_CLIENT_SECRET"]
USERNAME      = os.environ["MLFLOW_USERNAME"]
PASSWORD      = os.environ["MLFLOW_PASSWORD"]
REDIRECT_URI  = f"{PLATFORM}/oauth2/callback"                 # any URI t
he client has registered
VERIFY_TLS    = os.environ.get("PLATFORM_CA", False)          # CA bundle
path, or False to skip (lab only)

s = requests.Session(); s.verify = VERIFY_TLS
_b64url = lambda b: base64.urlsafe_b64encode(b).rstrip(b"=").decode()

def get_tokens() -> dict:
    """Run the authorization-code + PKCE flow headlessly. Returns the Dex
token response."""
    verifier  = _b64url(secrets.token_bytes(48))
    challenge = _b64url(hashlib.sha256(verifier.encode()).digest())
    # 1) start the flow -> auth-request id
    req = s.get(f"{PLATFORM}/dex/api/v1/authorize", params={
        "client_id": CLIENT_ID, "redirect_uri": REDIRECT_URI, "response_t
ype": "code",
        "scope": "openid email groups offline_access", "state": "cli",
        "code_challenge": challenge, "code_challenge_method": "S256"}).js
on()["req"]
    # 2) RSA-encrypt the password, then log in via the local connector ->
auth code
    pk  = s.get(f"{PLATFORM}/dex/pubkey").json()              # {"ts":
..., "pubkey": "<PEM>"}
    payload = json.dumps({"ts": pk["ts"], "password": PASSWORD}, separato
rs=(",", ":")).encode()
    enc = base64.b64encode(load_pem_public_key(pk["pubkey"].encode()).enc
rypt(payload, padding.PKCS1v15())).decode()
    redirect = s.post(f"{PLATFORM}/dex/api/v1/authorize/local", params=
```

```
{"req": req},
        json={"account": USERNAME, "password": enc}).json()["redirect_ur
l"]
    code = parse_qs(urlparse(redirect).query)["code"][0]
    # 3) exchange the code (with the PKCE verifier) -> id_token + refresh
_token
    return s.post(f"{PLATFORM}/dex/token", data={
        "grant_type": "authorization_code", "code": code, "redirect_uri":
REDIRECT_URI,
        "code_verifier": verifier, "client_id": CLIENT_ID, "client_secre
t": CLIENT_SECRET}).json()

def refresh(refresh_token: str) -> str:
    """Mint a fresh id token from a refresh token — no login, no browse
r."""
    return s.post(f"{PLATFORM}/dex/token", data={
        "grant_type": "refresh_token", "refresh_token": refresh_token,
        "client_id": CLIENT_ID, "client_secret": CLIENT_SECRET,
        "scope": "openid email groups"}).json()["id_token"]
```

Shell 等价实现（curl + openssl，无 Python 依赖）

```
PLATFORM=https://<platform>; CLIENT_ID=<client>; CLIENT_SECRET=<secret>
USERNAME='<user>'; PASSWORD='<password>'; REDIRECT_URI="$PLATFORM/oauth2/
callback"

V=$(openssl rand -base64 48 | tr '+/' '-_' | tr -d '=' | cut -c1-64)
# PKCE verifier
C=$(printf %s "$V" | openssl dgst -sha256 -binary | openssl base64 -A | t
r '+/' '-_' | tr -d '=')
RU=$(jq -rn --arg u "$REDIRECT_URI" '$u|@uri'); SC=$(jq -rn '"openid emai
l groups offline_access"|@uri')
REQ=$(curl -sk "$PLATFORM/dex/api/v1/authorize?client_id=$CLIENT_ID&redir
ect_uri=$RU&response_type=code&scope=$SC&state=cli&code_challenge=$C&code
_challenge_method=S256" | jq -r .req)
PK=$(curl -sk "$PLATFORM/dex/pubkey"); TS=$(echo "$PK"|jq -r .ts); echo
"$PK"|jq -r .pubkey >/tmp/dex_pub.pem
ENC=$(printf '{"ts":"%s","password":"%s"}' "$TS" "$PASSWORD" | openssl pk
eyutl -encrypt -pubin -inkey /tmp/dex_pub.pem -pkeyopt rsa_padding_mode:p
kcs1 | openssl base64 -A)
CODE=$(curl -sk -X POST "$PLATFORM/dex/api/v1/authorize/local?req=$REQ" -
H 'Content-Type: application/json' \
  --data "$(jq -nc --arg a "$USERNAME" --arg p "$ENC" '{account:$a,passwo
rd:$p}')" | jq -r .redirect_url | sed -E 's/.*code=([^&]+).*/\1/')
curl -sk "$PLATFORM/dex/token" -d grant_type=authorization_code -d code
="$CODE" \
  --data-urlencode redirect_uri="$REDIRECT_URI" -d code_verifier="$V" \
  -d client_id="$CLIENT_ID" --data-urlencode client_secret="$CLIENT_SECRE
T" | jq -r .id_token
```

连接 **SDK**

```python
import os, mlflow

tok = get_tokens()
os.environ["MLFLOW_TRACKING_TOKEN"] = tok["id_token"].strip()           # → Authorization: Bearer
mlflow.set_tracking_uri("http://mlflow-tracking-server.kubeflow:5000")  # in-cluster Service (fronted by the OAuth proxy)
mlflow.set_workspace("team-a")                                          # workspace namespace → X-MLFLOW-WORKSPACE
mlflow.set_experiment("my-experiment")

with mlflow.start_run(run_name="sdk-quickstart") as run:
    mlflow.log_param("learning_rate", 2e-4)
    mlflow.log_metric("loss", 0.123)
    print("run:", run.info.run_id)
```

该 run 会出现在 **Alauda AI** → 工具 → **MLFlow** 下，归属于你认证时使用的用户。（在受保护的安装环境中已端到端验证：run owner 是 token 的用户身份。）

当 client 在集群内部运行时（pipeline component、Workbench notebook），请使用集群内 Service URL `http://mlflow-tracking-server.kubeflow:5000`。在集群外部，则改为指向平台路由 `https://<platform>/clusters/<cluster>/mlflow`——两者都会到达同一个 OAuth proxy（如果平台证书不被你的机器信任，请设置 `MLFLOW_TRACKING_INSECURE_TLS=true`）。

> **WARNING**
>
> 对于共享或 headless 使用场景，建议使用专用的、非个人的账号（普通的平台登录，而不是 Kubernetes ServiceAccount），并将其凭证和 client secret 保存在 Kubernetes `Secret` 中，切勿写入代码。务必对 token 调用 `.strip()`（末尾换行会导致 `Invalid … character(s) in header value: 'Bearer …\n'`）。id token 会过期（默认 24 小时）；对于长时间运行的 job，请使用 `refresh(tok["refresh_token"])` 续期，而不要再次登录。

选择 **workspace**

run 会记录在你所选择的 workspace 中；如果未选择，则使用 server 的默认 workspace。以下任一方式都可以设置它（SDK 会将其转换为 `X-MLFLOW-WORKSPACE` header）：

- 在代码中调用 `mlflow.set_workspace("team-a")`，
- 或设置环境变量 `MLFLOW_WORKSPACE=team-a`。

你只能使用你账号有权限访问的 workspace；参见 MLflow Workspaces and Access Control。

注册模型

model registry 以 workspace 为作用域，并以相同方式进行授权，因此一旦连接成功，常规 SDK 调用即可正常工作：

```
mlflow.set_workspace("team-a")
with mlflow.start_run():
    mlflow.sklearn.log_model(sk_model, name="model", registered_model_nam
e="fraud-detector")
```

然后可在 MLflow UI 中将已注册版本提升为 **Staging** 或 **Production**。

替代方案：**session cookie**（无需平台更改）

如果你无法启用 `spec.auth.oauth.skipJwtBearerTokens`，可以驱动 proxy 自身的登录流程来获取其 `_oauth2_proxy` cookie，并将其附加到请求中——这在任何安装环境中都可直接使用，无需更改。proxy 会为你启动 OAuth 流程（它自己的 PKCE 和 `redirect_uri`）；你只需通过同一个脚本化登录重放该流程，并将 code 交回给 proxy callback：

```
PLATFORM=https://<platform>; CLUSTER=<cluster>
USERNAME='<user>'; PASSWORD='<password>'
JAR=$(mktemp)
# 1) start the MLflow proxy login -> the Dex auth query it wants
LOC=$(curl -sk -c "$JAR" -D - -o /dev/null "$PLATFORM/clusters/$CLUSTER/mlflow/" \
  | awk 'BEGIN{IGNORECASE=1}/^location:/{print $2}' | tr -d '\r')
QS=${LOC#*\?}
# 2) authorize -> req, then 3) scripted local login -> the proxy callback URL
REQ=$(curl -sk -b "$JAR" -c "$JAR" "$PLATFORM/dex/api/v1/authorize?$QS" | jq -r .req)
PK=$(curl -sk "$PLATFORM/dex/pubkey"); TS=$(echo "$PK"|jq -r .ts); echo "$PK"|jq -r .pubkey >/tmp/dex_pub.pem
ENC=$(printf '{"ts":"%s","password":"%s"}' "$TS" "$PASSWORD" | openssl pkeyutl -encrypt -pubin -inkey /tmp/dex_pub.pem -pkeyopt rsa_padding_mode:pkcs1 | openssl base64 -A)
CB=$(curl -sk -b "$JAR" -c "$JAR" -X POST "$PLATFORM/dex/api/v1/authorize/local?req=$REQ" -H 'Content-Type: application/json' \
  --data "$(jq -nc --arg a "$USERNAME" --arg p "$ENC" '{account:$a,password:$p}')" | jq -r .redirect_url)
# 4) the proxy callback exchanges the code and sets the _oauth2_proxy cookie
curl -sk -b "$JAR" -c "$JAR" -o /dev/null "$CB"
COOKIE=$(awk -F'\t' '$6 ~ /^_oauth2_proxy/{printf "%s=%s; ",$6,$7}' "$JAR" | sed 's/; $//')   # includes any _oauth2_proxy_N chunks
echo "$COOKIE"
```

然后使用 header provider 附加 cookie（该 cookie 携带你的身份——没有 token，也不需要平台设置）：

```
import os, mlflow
from mlflow.tracking.request_header.abstract_request_header_provider impo
rt RequestHeaderProvider
from mlflow.tracking.request_header.registry import _request_header_provi
der_registry

class ProxySessionHeader(RequestHeaderProvider):
    def in_context(self):
        return bool(os.environ.get("MLFLOW_PROXY_COOKIE"))     # export M
LFLOW_PROXY_COOKIE='_oauth2_proxy=<value>'
    def request_headers(self):
        return {"Cookie": os.environ["MLFLOW_PROXY_COOKIE"]}

_request_header_provider_registry.register(ProxySessionHeader)
mlflow.set_tracking_uri("https://<platform>/clusters/<cluster>/mlflow")
mlflow.set_workspace("team-a")
```

在集群内部，请改为指向集群内 Service URL `http://mlflow-tracking-server.kubeflow:5000`（不会有 TLS 问题）。在外部的 `https://<platform>/…` 路由上，平台证书是自签名的，因此请设置 `MLFLOW_TRACKING_INSECURE_TLS=true`（或者将 `REQUESTS_CA_BUNDLE` 指向平台 CA）。

你也可以从浏览器会话中复制 `_oauth2_proxy` cookie（DevTools → **Application/Storage** → **Cookies**）。session cookie 会过期——当调用开始返回登录重定向时，请重新生成。

故障排查

症状	检查
`/dex/api/v1/authorize` 返回 `PKCE code_challenge is required`	client 强制使用 PKCE。请发送 `code_challenge` 和 `code_challenge_method=S256`（helper 已自动完成）。
本地登录返回 captcha challenge / `CaptchaError`	最近失败登录过多，触发了 retry-captcha。请等待、修正凭证后重试——首次干净登录不需要 captcha。
`/dex/token` 返回 `invalid_grant`	auth code 或 PKCE verifier 已过期或被重复使用。请从头重新执行流程（`authorize` → login → token）；code 仅可使

症状	检查
	用一次。
调用返回 HTML 或重定向（`302` 到登录页面）	**Token** 方式： proxy 拒绝了 bearer token——请确认 `spec.auth.oauth.skipJwtBearerTokens` 为 `true`，且该 token 是有效的 Dex id token（`aud` = proxy 的 client）。**Cookie** 方式： `_oauth2_proxy` cookie 丢失或已过期。
`Invalid … character(s) in header value: 'Bearer …\n'`	token 末尾带有空白字符。请将 `MLFLOW_TRACKING_TOKEN` 设置为 `.strip()` 后的值。
`Failed to query /api/3.0/mlflow/server-info`，通常伴随 `SSLCertVerificationError: self-signed certificate`	proxy 拒绝了你的凭证，并返回了一个到登录页面的 `302`；随后 SDK 会跟随该重定向访问平台的自签名 HTTPS endpoint，并在 TLS 校验时失败。请修正凭证，而不是 TLS：对于 **token** 方式，请确认 `spec.auth.oauth.skipJwtBearerTokens` 为 `true` 且 token 有效；对于 **cookie** 方式，请重新生成 cookie。只有当你有意使用外部 `https://<platform>/…` 路由时，才需要另外设置 `MLFLOW_TRACKING_INSECURE_TLS=true`（或将 `REQUESTS_CA_BUNDLE` 指向平台 CA）。
`403 PERMISSION_DENIED`	你的账号没有访问该 workspace namespace 的权限。请申请该 workspace 的访问权限（参见 MLflow Workspaces and Access Control）；这里不涉及 ServiceAccount。
run 显示的 owner 或 workspace 错误	owner 是你认证时使用的身份；workspace 由 `set_workspace()` / `MLFLOW_WORKSPACE` 决定（否则使用 server 默认值）。请同时检查这两项。

Kubeflow Pipeline + MLflow 集成

本指南展示了 Kubeflow Pipelines (KFP) 组件如何使用 **MLflow Python client** 将参数、指标和模型记录到 Alauda AI 上的 MLflow。认证以及 workspace/RBAC 遵循 将 MLflow Python SDK 与认证和 RBAC 一起使用 —— 每个组件都会使用用户身份 token 进行认证，server 会以该用户身份记录 run。

目录

- 适用范围
- 前提条件
- 组件如何访问 MLflow
- 完整示例：使用 MLflow 的训练 pipeline
- 上传并运行
  - 通过 KFP UI
  - 通过 KFP SDK
- 在 Trainer v2 pipeline 中使用 MLflow
- 最佳实践
  - 在 MLflow 中使用 pipeline job ID
  - 将凭据保存在 Secret 中并刷新 token
  - 在 run 内记录指标
  - 面向生产的 artifact 存储
- 故障排查

适用范围

- Alauda AI 2.5 及更高版本。
- 已安装 Kubeflow Pipelines 和 **MLflow Operator**，并且 `MLflow` custom resource 正在运行。
- MLflow workspace 是一个标注了 `mlflow-enabled=true` 的 namespace。
- 对于 bearer-token 方法，请保持默认的 `spec.auth.oauth.skipJwtBearerTokens: true` — 请参见 SDK 指南中的 验证 token 方法。无需更改全局认证，cookie 方法也不需要任何额外配置。

前提条件

- `kfp` 和 `kfp-kubernetes` Python SDK（`pip install kfp kfp-kubernetes`）。
- 可以访问 KFP endpoint（请参见 使用 Kubeflow Pipelines）。
- 一个用于专用非个人账户的 **Dex id token**（普通平台登录，不是 Kubernetes ServiceAccount），通过 authorization-code flow 在无浏览器环境下签发（请参见 SDK 指南）。将其存储在 Kubernetes `Secret` 中，并注入到 component 中。
- 该账户可以访问的 MLflow workspace（一个带有 `mlflow-enabled=true` 的 namespace）。

组件如何访问 **MLflow**

pipeline component 运行在集群内部，因此它通过集群内的 Service `http://mlflow-tracking-server.kubeflow:5000` 访问 MLflow（该 Service 前面由 OAuth proxy 提供入口 — 组件不会直接使用 MLflow container 端口）。它的认证方式与其他 MLflow client 完全一致：

- `MLFLOW_TRACKING_TOKEN` — 一个 Dex id token；MLflow client 会将其作为 `Authorization: Bearer …` 发送。
- `mlflow.set_workspace(...)` — 选择 workspace（`X-MLFLOW-WORKSPACE`）。

server 会从 token 中读取身份，并将 run 记录到该用户名下。有关 token 的获取方式以及授权的工作原理，请参见 SDK 指南。

完整示例：使用 **MLflow** 的训练 **pipeline**

该 component 使用 MLflow client，并从通过 `kfp-kubernetes` ↗ 注入的 `Secret` 中读取 `MLFLOW_TRACKING_TOKEN`。KFP v2 会从各自的 source 打包每个 component，因此 `import mlflow` 位于函数内部。

```
from kfp import dsl, compiler
from kfp import kubernetes


@dsl.component(base_image="python:3.11-slim", packages_to_install=["mlflow==3.13.0"])
def train_model(
    workspace: str,
    model_name: str,
    learning_rate: float,
    epochs: int,
    run_id: str,
) -> dict:
    """Simulated training component that logs to MLflow as the calling user."""
    import mlflow   # MLFLOW_TRACKING_TOKEN is injected from a Secret (see the pipeline below)

    mlflow.set_tracking_uri("http://mlflow-tracking-server.kubeflow:5000")  # in-cluster Service, via the OAuth proxy
    mlflow.set_workspace(workspace)
    mlflow.set_experiment("kfp-training-experiment")

    metrics = {}
    with mlflow.start_run(run_name=f"run-{run_id}"):
        mlflow.log_param("model_name", model_name)
        mlflow.log_param("learning_rate", learning_rate)
        mlflow.log_param("epochs", epochs)
        for epoch in range(1, epochs + 1):
            loss = 2.0 * (0.95 ** epoch)
            accuracy = 1.0 - loss
            mlflow.log_metric("loss", loss, step=epoch)
            mlflow.log_metric("accuracy", accuracy, step=epoch)
            metrics = {"final_loss": loss, "final_accuracy": accuracy}

    print("logged run:", mlflow.last_active_run().info.run_id)
    return metrics


@dsl.pipeline(name="mlflow-training-pipeline", description="Train with MLflow tracking")
def training_pipeline(
    workspace: str = "team-a",
```

```
    model_name: str = "qwen3-0.6b",
    learning_rate: float = 2e-4,
    epochs: int = 10,
):
    task = train_model(
        workspace=workspace,
        model_name=model_name,
        learning_rate=learning_rate,
        epochs=epochs,
        # PIPELINE_JOB_ID_PLACEHOLDER resolves to the run's job id at run time;
        # pass it in as an argument (a component cannot reference dsl.* itself).
        run_id=dsl.PIPELINE_JOB_ID_PLACEHOLDER,
    )
    # Inject the Dex id token from a Secret as MLFLOW_TRACKING_TOKEN.
    kubernetes.use_secret_as_env(
        task, secret_name="mlflow-token", secret_key_to_env={"token": "MLFLOW_TRACKING_TOKEN"}
    )


compiler.Compiler().compile(training_pipeline, "pipeline.yaml")
```

使用 Dex id token 创建 `mlflow-token` Secret。按照 SDK 指南中的 authorization-code flow，在无浏览器环境下签发 `ID_TOKEN` — 请参见 从命令行获取 token：

```
# ID_TOKEN: mint it with the curl/Python flow in the SDK guide (browser-free, current grants)
kubectl -n <pipeline-namespace> create secret generic mlflow-token --from-literal=token="$ID_TOKEN"
```

WARNING

id token 会过期（默认 24 小时），因此在提交长时间运行的 pipeline 之前，请刷新 `mlflow-token` Secret — 或者在 component 内部根据保存在 Secret 中的该账户凭据签发 token（参见 SDK 指南中的 token flow），并使用 refresh token 续期，这样每次运行都能获得新的 token。

上传并运行

通过 KFP UI

1. 进入 **Kubeflow Dashboard** → **Pipelines** → **Upload Pipeline** 并选择 `pipeline.yaml`。
2. 点击 **Create Run** 并填写参数（workspace、model name、epochs）。
3. run 启动后，在 **Alauda AI** → **Tools** → **MLFlow** 下检查 MLflow UI — run owner 是 token 的用户。

通过 KFP SDK

```
from kfp.client import Client

client = Client(host="<MY-KFP-ENDPOINT>")
run = client.create_run_from_pipeline_package(
    "pipeline.yaml",
    arguments=dict(workspace="team-a", model_name="qwen3-0.6b", epochs=1
0),
)
print(f"Run ID: {run.run_id}")
```

在 Trainer v2 pipeline 中使用 MLflow

如果你使用 Kubeflow Trainer v2 进行 fine-tune，那么 framework 的 MLflow 集成（例如 LLaMA-Factory 中的 `report_to: mlflow`）采用相同的认证方式。Trainer v2 使用 `apiVersion: trainer.kubeflow.org/v1alpha1`、`kind: TrainJob`，以及 `spec.runtimeRef` + `spec.trainer` 的结构。将其指向集群内的 Service，并从 `Secret` 中注入 id token：

```yaml
apiVersion: trainer.kubeflow.org/v1alpha1
kind: TrainJob
metadata:
  name: mlflow-finetune
spec:
  runtimeRef:
    name: torch-distributed        # a TrainingRuntime / ClusterTrainingR
untime
  trainer:
    image: alaudadockerhub/fine_tune_with_llamafactory:v0.1.1
    env:
      - name: MLFLOW_TRACKING_URI
        value: "http://mlflow-tracking-server.kubeflow:5000"
      - name: MLFLOW_EXPERIMENT_NAME
        value: "trainer-v2-finetune"
      - name: MLFLOW_TRACKING_TOKEN
        valueFrom:
          secretKeyRef:
            name: mlflow-token       # a Secret holding a Dex id token
            key: token
```

完整的 Trainer v2 + MLflow 示例请参见 使用 Workbench 进行 LLM fine-tuning。

如果你需要一个更完整的方案，将此集成与 MLflow Model Registry、TrustyAI `LMEvalJob` 以及每日运行的 KFP Recurring Run 结合起来，请参见 基于 MLflow 和 TrustyAI 的每日 Fine-Tuning Pipeline。

最佳实践

在 **MLflow** 中使用 **pipeline job ID**

KFP v2 提供了 `dsl.PIPELINE_JOB_ID_PLACEHOLDER`（v1 中的 `dsl.RUN_ID_PLACEHOLDER` 已移除）。它是 pipeline 级别的 placeholder，因此请作为参数传入 component — component 不能在自身内部直接引用 `dsl.*`。将接收到的字符串用于 run name，以便使每次 pipeline 执行的 runs 保持唯一。

将凭据保存在 **Secret** 中并刷新 **token**

切勿将 token 或账户凭据硬编码到 `pipeline.yaml` 中 — 编译后的 pipeline 会被存储和共享。请从 `Secret` 中注入它们，并在 id token 过期前刷新它（或在 component 内部签发）。

在 **run** 内记录指标

每个 metric 都属于一个 `mlflow.start_run()` block。如果 component 有多个逻辑阶段，应为每个阶段分别打开一个 run，而不是在 run context 之外记录。

面向生产的 **artifact** 存储

通过代理的 S3-compatible storage，pipeline component 可以在无需获得 S3 凭据的情况下，通过 tracking server 上传。请在记录 artifacts 或注册模型之前配置 `artifacts.s3`（参见 MLflow installation → 高可用性和存储）。

故障排查

症状	检查
component 失败，并返回 HTML/redirect（`302`）响应	OAuth proxy 拒绝了 token。确认 `spec.auth.oauth.skipJwtBearerTokens` 为 `true`，并且 `MLFLOW_TRACKING_TOKEN` 是有效的 Dex id token（请参见 SDK 指南）。
`401 UNAUTHENTICATED`	`MLFLOW_TRACKING_TOKEN` 未设置、为空或已过期 — 请刷新 `mlflow-token` Secret。
`403 PERMISSION_DENIED`	token 的用户没有访问该 workspace namespace 的权限。请授予对 MLflow workspace 的访问权限（请参见 MLflow Workspaces and Access Control）；这里不涉及 ServiceAccount。
run 显示在错误的 owner / workspace 下	owner 是 token 的身份；workspace 是 `set_workspace()` 指定的值（否则使用 server 默认值）。请同时检查两者。
MLflow 指标未出现在 KFP UI 中	KFP 和 MLflow 是两个独立系统。记录到 MLflow 的指标会出现在 MLflow UI（**Alauda AI** → **Tools** → **MLFlow**）中，而不会显示在 KFP run 输出中。

使用 MLflow 跟踪 AI 智能体

MLflow Tracing 会记录 AI 智能体执行的步骤，包括模型请求和响应、链调用和工具调用、检索步骤、延迟、令牌使用量以及错误。在 Alauda AI 上，trace 会存储在所选的 MLflow experiment 中，并受到与 runs 和 models 相同的 workspace 身份验证和 Kubernetes RBAC 保护。

本指南介绍以下场景的推荐 tracing 路径：

- 使用 LangChain 构建的 Python 智能体。
- 通过 OpenAI 兼容 API 调用 Llama Stack，现称 OGX 的应用程序。
- 已经发出 OpenTelemetry traces 的 OGX 服务器或其他应用程序。

目录

前提条件

跟踪 LangChain 智能体

跟踪 OGX 应用程序

导出 OGX OpenTelemetry traces

查看 traces

故障排查

前提条件

- 一个 MLflow tracking server 和一个你可以访问的 workspace。请参阅 MLflow Workspaces 和访问控制。
- `MLFLOW_TRACKING_TOKEN` 中的 Dex id token。请按照 将 MLflow Python SDK 与身份验证和 RBAC 一起使用 获取并续期 token。
- 用于 traces 的 experiment。创建或写入 traces 需要对 workspace 的 `experiments` 资源具有 `create` 和 `update` 权限。
- 智能体到 MLflow tracking server 及其模型 endpoint 的网络访问。

在启动智能体之前，先设置通用的 MLflow 连接变量：

```
export MLFLOW_TRACKING_URI=http://mlflow-tracking-server.kubeflow:5000
export MLFLOW_TRACKING_TOKEN='<dex-id-token>'
export MLFLOW_WORKSPACE=team-a
```

上面的 URI 是集群内 Service，由 OAuth proxy 提供前置代理。对于集群外的客户端，请使用 `https://<platform>/clusters/<cluster>/mlflow`。

> **WARNING**
>
> Trace 可能包含提示词、响应、检索到的内容，以及工具参数或结果。除非你的组织允许将这些数据存储在 MLflow 中，否则不要发送密钥或敏感个人数据。

跟踪 LangChain 智能体

MLflow 的 LangChain 集成会自动为智能体、嵌套链、模型请求、检索和工具调用创建 span。它支持同步、异步、批处理和流式 LangChain 调用。

安装智能体依赖项：

```
pip install "mlflow[genai]" langchain langchain-openai
```

下面的示例使用工具调用智能体和一个与 OpenAI 兼容的模型 endpoint。为你环境中可用的模型服务设置 `MODEL_BASE_URL`、`MODEL_API_KEY` 和 `MODEL_ID`。

```
import os

import mlflow
from langchain.agents import create_agent
from langchain.tools import tool
from langchain_openai import ChatOpenAI


mlflow.set_experiment("agent-tracing")
mlflow.langchain.autolog()


@tool
def get_weather(city: str) -> str:
    """Return the current weather for a city."""
    # Replace this example result with a call to your weather service.
    return f"The weather in {city} is sunny and 24 C."


model = ChatOpenAI(
    model=os.environ["MODEL_ID"],
    base_url=os.environ["MODEL_BASE_URL"],
    api_key=os.environ["MODEL_API_KEY"],
)

agent = create_agent(
    model=model,
    tools=[get_weather],
    system_prompt="You are a concise travel assistant.",
)

result = agent.invoke(
    {"messages": [{"role": "user", "content": "Should I take a coat to Ta
ipei today?"}]}
)
print(result["messages"][-1].content)
```

在应用启动期间、调用智能体之前执行一次 `mlflow.langchain.autolog()`。现有的 LangChain 应用代码无需其他更改。

跟踪 **OGX** 应用程序

OGX 提供与 OpenAI 兼容的 Chat Completions 和 Responses API。对于调用 OGX 的 Python 应用程序，MLflow 的 OpenAI 集成是最简单的选择，并且可以完整跟踪 Responses API 调用。

安装客户端依赖项：

```
pip install "mlflow[genai]" openai
```

将 OpenAI 客户端指向 OGX API。`@mlflow.trace` 包装器会将 `run_agent` 执行的所有模型和工具活动归入一个父 span；`mlflow.openai.autolog()` 会为每个 OGX 请求创建嵌套 span。

```
import os

import mlflow
from openai import OpenAI


mlflow.set_experiment("ogx-agent-tracing")
mlflow.openai.autolog()

client = OpenAI(
    base_url=os.environ.get(
        "OGX_BASE_URL",
        "http://<ogx-service>.<namespace>.svc.cluster.local:8321/v1",
    ),
    api_key=os.environ.get("OGX_API_KEY", "not-used"),
)


@mlflow.trace(name="ogx-agent")
def run_agent(question: str) -> str:
    response = client.responses.create(
        model=os.environ["OGX_MODEL_ID"],
        input=question,
    )
    return response.output_text


print(run_agent("Give me a two-step plan for testing a Kubernetes operato
r."))
```

在多步骤智能体循环中发出的每次调用，都会嵌套在 `ogx-agent` trace 之下。当天请求使用函数工具时，MLflow 还会记录工具定义以及模型的工具调用响应。

导出 OGX OpenTelemetry traces

当你希望跟踪 OGX 服务器本身、为非 Python 应用程序添加埋点，或通过 collector 路由 telemetry 时，请使用 OpenTelemetry。MLflow 通过 **OTLP/HTTP** 在 `/v1/traces` 接收 traces；它不接受 OTLP/gRPC 摄取。

首先，获取数值形式的 experiment ID：

```
import mlflow

experiment = mlflow.set_experiment("ogx-server-tracing")
print(experiment.experiment_id)
```

配置 OpenTelemetry Collector，以接收来自 OGX 的 traces 并将其转发到 Alauda MLflow：

```
receivers:
  otlp:
    protocols:
      http:
        endpoint: 0.0.0.0:4318

processors:
  batch: {}

exporters:
  otlphttp/mlflow:
    traces_endpoint: http://mlflow-tracking-server.kubeflow:5000/v1/trace
s
    headers:
      Authorization: "Bearer ${env:MLFLOW_TRACKING_TOKEN}"
      X-MLflow-Experiment-Id: "${env:MLFLOW_EXPERIMENT_ID}"
      X-MLFLOW-WORKSPACE: "${env:MLFLOW_WORKSPACE}"

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [otlphttp/mlflow]
```

使用环境变量 `MLFLOW_TRACKING_TOKEN`、`MLFLOW_EXPERIMENT_ID` 和 `MLFLOW_WORKSPACE` 启动 collector。然后配置 OGX，将 OTLP/HTTP traces 发送到 collector：

```
export OTEL_EXPORTER_OTLP_ENDPOINT=http://otel-collector:4318
export OTEL_EXPORTER_OTLP_PROTOCOL=http/protobuf

opentelemetry-instrument \
  --traces_exporter otlp \
  --metrics_exporter none \
  --service_name ogx-server \
  -- \
  ogx run starter
```

OGX 使用 OpenTelemetry 自动埋点来采集服务器和受支持的 SDK 活动。MLflow 的 OTLP endpoint 只接收 traces；metrics 请单独路由到 metrics backend。

> **NOTE**
>
> 除非你有意将它们配置为共用同一个 tracer provider，否则不要对同一组调用同时启用 MLflow autologging 和独立的 OpenTelemetry exporter。独立 exporter 可能会创建重复的 traces。对于 OGX Responses API 客户端调用，建议优先使用 `mlflow.openai.autolog()`，因为 OpenTelemetry 自动埋点可能无法捕获所有 Responses API 操作。

查看 **traces**

1. 打开 **Alauda AI** → **Tools** → **MLFlow**。
2. 选择与应用程序或 collector 配置相同的 workspace 和 experiment。
3. 打开该 experiment 的 **Traces** 选项卡。
4. 选择一个 trace，查看其 span 层次结构、输入和输出、延迟、令牌使用量、工具调用以及错误。

故障排查

症状	检查内容
请求重定向到登录页或返回 `401`	确认 `MLFLOW_TRACKING_TOKEN` 包含当前的 Dex id token，并且 MLflow OAuth proxy 接受 bearer token。请参阅 验证

症状	检查内容
	token 方法。
`403 PERMISSION_DENIED`	已认证用户需要能够访问该 workspace，并且对其 `experiments` 资源具有 `create` / `update` 权限。
没有出现 traces	确认应用程序和 UI 使用相同的 tracking URI、workspace、experiment 和时间范围。还要确保在调用智能体之前已启用 autologging。
OTLP 导出返回 `400` 或 `403`	确认 exporter 使用 OTLP/HTTP，并发送 `Authorization`、`X-MLflow-Experiment-Id` 和 `X-MLFLOW-WORKSPACE` 标头。
同一个请求出现了两次	对该请求路径禁用 MLflow autologging 或独立的 OpenTelemetry exporter 其中之一。
OGX Responses API 调用未出现在 OpenTelemetry traces 中	使用 `mlflow.openai.autolog()` 跟踪客户端，它支持 Responses API。

Agentic MLOps

将编码智能体与本地部署的推理服

简介

前提条件

各组件如何协同工作

步骤 1：部署并进行端点冒烟测试

步骤 2：在运行时启用 tool calling

步骤 2b（可选）：配置 reasoning models

步骤 3：连接你的编码智能体

最佳实践

故障排查

参考资料

使用编码智能体与本地部署 **LLM** 开展 **MLOps**

简介

设置智能体的工作环境

管理 InferenceServices 和 LLMInferenceServices

管理网关：身份验证和速率限制

调优服务性能以适配硬件

规划微调并生成报告

每日 MLOps 循环

最佳实践与防护措施

参考

将编码智能体与本地部署的推理服务结合使用

目录

简介
前提条件
各组件如何协同工作
步骤 1：部署并进行端点冒烟测试
步骤 2：在运行时启用 tool calling
步骤 2b（可选）：配置 reasoning models 和 reasoning effort
服务端参数
配置 reasoning effort 和 thinking 行为
服务端默认值
请求时控制
步骤 3：连接你的编码智能体
opencode
Codex CLI
Claude Code
选项 1：LiteLLM 代理
选项 2：claude-code-router
本地部署运行注意事项
最佳实践
推荐用于编码智能体的模型
选择适配你硬件的模型

适配你硬件的量化版本
硬件适配指南
调优 inference service 性能
启用 MTP（multi-token prediction）
开始进行 vibe coding
开始进行 MLOps
故障排查
参考资料

简介

opencode（opencode↗）、Codex CLI↗ 和 Claude Code↗ 这类编码智能体是基于终端的助手，它们会读取你的仓库、规划变更、编辑文件，并代表你运行命令。它们通常通过互联网与托管的模型提供商通信。

本文档展示如何将这些智能体指向你在 Alauda AI 上自托管的模型，从而让你的源代码、提示词和基础设施配置永远不会离开集群。只要暴露的是 **OpenAI-compatible API**，并且启用了 **tool**（**function**）**calling**，你为其他工作负载部署的同一个本地部署 `InferenceService` 也可以为交互式编码智能体提供支持。opencode 和 Codex CLI 可以直接调用该端点；Claude Code 使用 Anthropic Messages API（`/v1/messages`），因此需要一个轻量级转换代理（参见 Claude Code）。

本页直接建立在部署操作指南之上。它不会重复如何创建或暴露 `InferenceService`；而是链接到相关内容，并重点介绍面向智能体的配置与调优。

> **WARNING**
>
> 编码智能体及其配置格式演进很快。下面的配置片段适用于撰写本文时可用的版本，作为起点是正确的。请始终以你所使用智能体的最新上游文档为准，确认字段名称。

前提条件

- 一个正在运行且就绪的 `InferenceService`，提供 OpenAI-compatible API。参见 使用 CLI 创建 Inference Service。
- 运行智能体的机器可访问该服务端点的网络。如果智能体运行在集群外的开发者笔记本上，请参见 为 Inference Services 配置外部访问。
- 一个支持 **tool/function calling** 的模型，并使用匹配的 vLLM parser 提供服务（参见 在运行时启用 tool calling）。如果没有这个能力，智能体可以聊天，但不能编辑文件或运行命令。
- 本地已安装智能体 CLI（`opencode`、`codex` 或 `claude`）。
- 对于 Claude Code，需要一个转换代理（LiteLLM 或 claude-code-router），将 Claude Code 的 Anthropic Messages API 连接到 OpenAI-compatible 端点（参见 Claude Code）。

各组件如何协同工作

```
opencode / Codex CLI
      │  OpenAI Chat Completions API  (POST /v1/chat/completions)
      ▼
External access / Load Balancer  ──►  KServe InferenceService (vLLM)

Claude Code
      │  Anthropic Messages API  (POST /v1/messages)
      ▼
Translation proxy (LiteLLM / claude-code-router)
      │  OpenAI Chat Completions API  (POST /v1/chat/completions)
      ▼
same InferenceService endpoint
```

- **opencode** 和 **Codex CLI** 原生支持 OpenAI Chat Completions API，因此可以直接调用 `InferenceService` 端点。
- **Claude Code** 使用 Anthropic Messages API，而 vLLM 不提供该 API。它需要在 OpenAI-compatible 端点前面放置一个小型转换代理（参见 Claude Code）。

步骤 1：部署并进行端点冒烟测试

按照 使用 CLI 创建 Inference Service 部署你的模型为 `InferenceService`；如果智能体运行在集群外，则按照 为 Inference Services 配置外部访问 暴露该服务。

在接入任何智能体之前，先确认端点可以响应聊天请求。如果 base URL、模型名称或认证配置错误，编码智能体会以令人困惑的方式失败，因此请先用 `curl` 验证：

```
# BASE_URL must end at /v1
BASE_URL="https://your-inference-service-domain.com/v1"
MODEL="qwen-2"        # must match --served-model-name in the InferenceSe
rvice
API_KEY="sk-local"    # any non-empty value if the server does not enforc
e auth

curl -sS ${BASE_URL}/chat/completions \
  -H "Authorization: Bearer ${API_KEY}" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "'"${MODEL}"'",
    "messages": [{"role": "user", "content": "Reply with the single word:
ready"}],
    "max_tokens": 16
  }'
```

一个正常的 JSON completion 表明端点可达且模型名称正确。注意你后续为每个智能体都会复用的三个值：**base URL**（以 `/v1` 结尾）、**model name**（`--served-model-name`）以及 **API key**。

> **TIP**
>
> 对于具备推理能力的模型（在推荐集合中，包括 GLM-5.2 和 Qwen3.6，以及 reasoning-mode Deepseek-v4 构建），还应在 vLLM 启动参数中添加匹配的 `--reasoning-parser`。参见 配置 reasoning models 和 reasoning effort。

步骤 2：在运行时启用 tool calling

编码智能体通过调用工具（读取文件、写入文件、运行 shell）来工作。这要求模型既要输出 tool calls，vLLM 也要能够解析它们。将以下参数添加到 `InferenceService` 中的 vLLM 启动

命令里（在 使用 CLI 创建 Inference Service 的示例中，它们添加在 `python3 -m vllm.entrypoints.openai.api_server` 那一行）：

```
--enable-auto-tool-choice \
--tool-call-parser hermes        # match the parser to your model family
```

- parser 必须与模型匹配。例如，Qwen2.5 和 QwQ-32B 通常使用 `hermes`；Qwen3-Coder 使用 `qwen3_xml`；Llama 3.x 模型使用 `llama3_json`；Mistral 模型使用 `mistral`。请查阅 vLLM tool calling 文档↗ 以获取当前 parser 列表及与你的模型匹配的取值。
- 某些模型需要特定的 chat template 才能正确输出 tool calls；如果模型卡有要求，请传入 `--chat-template`。
- 如果你提供的是 reasoning model，也要启用匹配的 `--reasoning-parser`，这样智能体收到的 assistant content 就会与 reasoning traces 分离并保持干净。

通过让智能体执行一个简单的文件操作来端到端验证 tool calling（例如，“创建一个包含 hi 这个词的 `hello.txt`”）。如果模型返回的是自然语言而不是编辑文件，则说明 tool calling 没有正确接入——请重新检查 parser 和模型。

步骤 2b（可选）：配置 reasoning models 和 reasoning effort

有些模型会先输出 chain-of-thought reasoning，再给出最终答案。在推荐的模型家族中，这包括 **GLM-5.2** 和 **Qwen3.6**（reasoning 默认开启），以及 reasoning-mode 的 **Deepseek-v4** 构建。vLLM 会把 reasoning traces 与 assistant content 分离，这样你的智能体就能收到干净的输出——但你必须启用匹配的参数。

服务端参数

将 `--reasoning-parser` 添加到你的 vLLM 启动命令中。如果同一个模型还需要 agent tool calls，则将其与合适的 `--tool-call-parser` 配对使用：

```
--enable-auto-tool-choice \
--tool-call-parser <parser> \
--reasoning-parser <reasoning-parser>
```

下表展示了 parser 配对供参考。parser 名称与版本相关，并且会随 vLLM 版本变化——请始终以你所运行版本的 vLLM tool calling 文档↗ 为准。

模型家族	`--tool-call-parser`	`--reasoning-parser`	说明
Qwen3 / Qwen3.6 reasoning（`Qwen/Qwen3*`）	请查阅当前 vLLM 文档中对应的 Qwen 变体	`qwen3`	reasoning 默认开启；如有需要，可通过 `chat_template_kwargs` 将其关闭
GLM-5.2（Zhipu AI）	请查阅当前 vLLM 文档中对应的 GLM 变体	请查阅当前 vLLM 文档中对应的 GLM 变体	混合 reasoning；tool-call 和 reasoning parser 共享 GLM 家族取值
Deepseek-v4（`deepseek_v4`）	请查阅当前 vLLM 文档中对应的 DeepSeek 变体	请查阅当前 vLLM 文档中对应的 DeepSeek 变体	仅 reasoning-mode 构建；参见 DeepSeek-V4-Flash (W4A8) 配方

这些家族对应的精确 `--tool-call-parser` / `--reasoning-parser` 取值取决于 vLLM 版本——在确定前请阅读模型卡和 vLLM tool calling 文档↗，并端到端验证 tool calls 是否能够往返正常。

配置 reasoning effort 和 thinking 行为

Reasoning effort 控制模型在回答前会“思考”多少。对于编码智能体，通常希望使用 低 reasoning effort，以保持交互延迟可接受——多次简短、低 reasoning 的轮次，胜过一次漫长、高 reasoning 的轮次。

服务端默认值

vLLM 不提供通用的 `--reasoning-effort` 启动参数。服务端级别的控制是通过模型的 chat template 实现的：你可以提供一个自定义 Jinja template，默认关闭 thinking，然后通过 `--`

`chat-template` 传入。或者，某些模型和 vLLM 版本会暴露按模型配置的 template kwargs；请查阅特定的 vLLM release notes。

请求时控制

不要假设每个基于 vLLM 的 `InferenceService` 都接受 `reasoning_effort`。是否支持取决于 vLLM 版本、OpenAI-compatible 服务端实现、模型以及 chat template。如果服务拒绝未知请求字段，即使模型本身支持 reasoning，`reasoning_effort` 也可能失败。

优先使用你已部署的 vLLM 服务文档中说明的模型特定控制方式。例如，Qwen3 风格的 template 通常使用 `chat_template_kwargs` 来启用或禁用 thinking：

```
{
  "model": "Qwen/Qwen3-8B",
  "messages": [{"role": "user", "content": "..."}],
  "chat_template_kwargs": {
    "enable_thinking": false
  }
}
```

使用 OpenAI Python client 时，通过 `extra_body` 传递 vLLM 特定的请求字段：

```
client.chat.completions.create(
    model="Qwen/Qwen3-8B",
    messages=[{"role": "user", "content": "..."}],
    extra_body={"chat_template_kwargs": {"enable_thinking": False}},
)
```

对于支持显式 thinking budget 的 parser，还可以为每个请求限制 reasoning token 数量：

```
{
  "model": "Qwen/Qwen3-0.6B",
  "messages": [{"role": "user", "content": "..."}],
  "thinking_token_budget": 256
}
```

当使用转换代理（LiteLLM 或 claude-code-router）时，在依赖这些字段之前，请先确认代理版本会透传这些 vLLM/OpenAI 扩展字段。

只有在你确认自己的 vLLM 镜像和模型模板确实支持它之后，才使用 `reasoning_effort`。在受支持的部署中，它可以作为顶层 Chat Completions 字段发送，例如 `"reasoning_effort": "low"`；在不受支持的部署中，请改用 `chat_template_kwargs`、`thinking_token_budget` 或 `max_tokens`。

步骤 3：连接你的编码智能体

opencode

opencode 会从项目根目录下的 `opencode.json` 或 `~/.config/opencode/opencode.json` 读取配置。定义一个指向你端点的自定义 OpenAI-compatible provider：

```
{
  "$schema": "https://opencode.ai/config.json",
  "provider": {
    "onprem": {
      "npm": "@ai-sdk/openai-compatible",
      "name": "On-Prem Alauda AI",
      "options": {
        "baseURL": "https://your-inference-service-domain.com/v1",
        "apiKey": "{env:ONPREM_API_KEY}"
      },
      "models": {
        "qwen-2": {
          "name": "Qwen3.6-35B-A3B (on-prem)"
        }
      }
    }
  }
}
```

- 模型键（`qwen-2`）必须与 `InferenceService` 的 `--served-model-name` 一致。
- 导出配置中引用的密钥，然后选择该模型：`export ONPREM_API_KEY=sk-local`，并在 opencode 中使用 `/models` 命令选择 `onprem/qwen-2`。

Codex CLI

Codex CLI 读取 `~/.codex/config.toml`。将你的端点注册为一个模型提供者并选择它：

```
model = "qwen-2"
model_provider = "onprem"

[model_providers.onprem]
name = "On-Prem Alauda AI"
base_url = "https://your-inference-service-domain.com/v1"
env_key = "ONPREM_API_KEY"
wire_api = "chat"
```

- `base_url` 必须以 `/v1` 结尾；`model` 必须与 `--served-model-name` 一致。
- `env_key` 指定保存 API key 的环境变量名称：`export ONPREM_API_KEY=sk-local`。
- 对于 vLLM 的 OpenAI Chat Completions API，请使用 `wire_api = "chat"`。

Claude Code

Claude Code 通过 Anthropic Messages API（`/v1/messages`）通信，而你的 `InferenceService` 暴露的是 OpenAI-compatible 端点（`/v1/chat/completions`）。通过在端点前运行一个转换代理来桥接二者。有两种常见选择：

- LiteLLM ↗ 代理：暴露 Anthropic-compatible 的 `/v1/messages` 端点，并将请求路由到任意后端模型。
- claude-code-router ↗：专为将 Claude Code 指向 OpenAI-compatible 及其他后端而构建的代理。

这两种方式都会替你处理 API 转换。根据你的工作流选择即可——LiteLLM 更通用，而 claude-code-router 则更贴合 Claude Code 的需求。

选项 1：LiteLLM 代理

启动 LiteLLM 代理，并将其指向你的 `InferenceService` 端点：

```
litellm --model openai/qwen-2 \
  --api_base https://your-inference-service-domain.com/v1 \
  --port 4000
```

这会暴露 `http://localhost:4000/v1/messages`（Anthropic 格式），并将请求转发到你的 OpenAI-compatible 后端。

然后将 Claude Code 指向该代理：

```
export ANTHROPIC_BASE_URL="http://127.0.0.1:4000"
export ANTHROPIC_AUTH_TOKEN="not_set"
export ANTHROPIC_API_KEY="not_set_either!"
export ANTHROPIC_MODEL="qwen-2"

export CLAUDE_CODE_DISABLE_NONESSENTIAL_TRAFFIC=1
export CLAUDE_CODE_ATTRIBUTION_HEADER=0
export CLAUDE_CODE_ENABLE_TELEMETRY=0
export CLAUDE_CODE_DISABLE_1M_CONTEXT=1
export CLAUDE_CODE_MAX_OUTPUT_TOKENS=64000

claude
```

选项 2：claude-code-router

在 `~/.claude-code-router/config.json` 中创建一个配置文件，将你的 `InferenceService` 作为 provider：

```
{
  "Providers": [
    {
      "name": "onprem",
      "api_base_url": "https://your-inference-service-domain.com/v1/chat/
completions",
      "api_key": "sk-local",
      "models": ["qwen-2"]
    }
  ],
  "Router": {
    "default": "onprem,qwen-2"
  }
}
```

然后通过 router 启动 Claude Code：

```
ccr code
```

router 会自动设置所需的 `ANTHROPIC_BASE_URL` 和其他环境变量——无需手动 `export`。模型由配置中的 `Router.default` 字段选择（格式：`provider_name,model_name`）。你也可以先在 shell 中通过 `eval "$(ccr activate)"` 启用 router，然后直接运行 `claude`。在运行中的会话内，可使用 `/model provider_name,model_name` 切换模型。

本地部署运行注意事项

- `ANTHROPIC_AUTH_TOKEN` / `ANTHROPIC_API_KEY` 的值（与 LiteLLM 选项一起使用）必须非空，但如果你的代理和端点不会检查它们，其内容并不重要；请在端点或代理层面进行访问控制（参见 管理网关，了解如何通过 Envoy AI Gateway 添加认证）。
- `CLAUDE_CODE_DISABLE_*` 标志才是真正让“本地部署”保持在本地的关键：如果没有它们，Claude Code 仍可能向 Anthropic 托管端点发出非必要请求，并向模型请求本地模型无法满足的功能（1M 上下文、超大输出等）。claude-code-router 会自动设置其中的一些。
- `ANTHROPIC_MODEL` 必须与 `InferenceService` 暴露的模型名称一致（即 `--served-model-name`）。
- 你也可以选择将 `ANTHROPIC_SMALL_FAST_MODEL` 设置为一个本地部署模型，这样后台/低成本请求也能留在本地。

Claude Code 的 agentic 能力在很大程度上取决于所提供模型的 tool-calling fidelity——优先选择 instruction 和 tool 调优能力强的模型，并在依赖它之前先端到端确认 tool calls 能够正常往返。

最佳实践

推荐用于编码智能体的模型

作为最佳实践，应当基于以下五个模型家族构建本地部署的编码智能体——并且为了保持结果可预测，最好将它们作为首选清单，而不是转向其他模型。每个家族都兼具强大的代码生成、可靠的指令遵循以及原生 tool/function calling 能力，并且都提供多种尺寸和量化格式，便于适配你现有的加速器。每种选择都应搭配与硬件匹配的量化方案：GPU 上使用 Unsloth↗ 的

GGUF 构建，或 Ascend NPU 上使用 **ModelSlim (`msModelSlim`)** 的 W8A8/W4A8 构建（参见 适配你硬件的量化版本）。

模型	类型	为什么适合编码智能体	适配性与说明
GLM-5.2（Zhipu AI）	大型 MoE	前沿级 agentic reasoning 和多步 tool 使用；擅长跨多个文件的大范围、有计划的编辑。	该集合中最重——建议使用量化版（GGUF / ModelSlim）或在多个加速器之间进行 tensor parallel。
Qwen3.6-27B-MTP（Qwen）	稠密 `qwen3_5` hybrid + MTP	通用代码能力强；原生 multi-token-prediction head 可加速 decode，使交互轮次更迅捷。	在 Ascend 上已验证 W8A8 → Qwen3.6-27B (W8A8)。量化后约 33–36 GB。
Qwen3.6-35B-A3B（Qwen）	MoE，约 3B active	该集合中 VRAM 性价比最高：每个 token 仅激活约 3B 参数，但保留 35B 的知识库。	在单张 24 GB GPU 上即可轻松运行——这是大多数用户的推荐默认选择。
Gemma4-12B-it / Gemma4-31B-it（Google）	稠密，instruction-tuned	指令遵循干净，tool calls 可靠。12B 适合中端 GPU；31B 可在 40–48 GB 显卡上提升质量。	追求延迟选 **12B**，追求质量选 **31B**。
Deepseek-v4（DeepSeek）	大型 MoE（MLA + DSA + MTP）	前沿级代码能力和 agentic planning；尽管专家池很大，稀疏 MoE 仍可保持较低的 active parameters。	较重——请使用量化版或多 GPU/多卡部署。在 Ascend 上已验证 W4A8 → DeepSeek-V4-Flash (W4A8)。

请根据你运行的 vLLM 版本确认匹配的 `--tool-call-parser`（对于具备 reasoning 能力的家族，还要确认 `--reasoning-parser`）——参见 在运行时启用 tool calling。

选择适配你硬件的模型

先根据你拥有的 GPU 内存来选择，然后挑选在为 KV cache 保留余量后仍能容纳的最大模型。粗略的权重大小估算公式是 `parameters × bytes-per-parameter`——FP16 约 2 bytes，

FP8/INT8 约 1 byte，INT4 约 0.5 bytes/parameter——除此之外，KV cache 和运行时开销还会占用更多内存。请保留 **15–25%** 的余量。

适配你硬件的量化版本

有两条量化路径覆盖 Alauda AI 运行的加速器。请选择与你硬件匹配的构建，然后应用上面的 15–25% 余量规则。

GPU — Unsloth GGUF。 Unsloth↗ 提供最新模型的 GGUF 构建，并针对使用 vLLM（`--quantization gguf`）的快速加载进行了优化。其 QAT（quantization-aware training）构建在激进位宽下比后量化更能保持质量。以下是推荐模型中最有用的构建：

模型	格式	**Active params**	**VRAM**（约）	说明
`unsloth/gemma-4-12b-it-GGUF`↗	GGUF	12B	~16 GB	适合单张中端 GPU
`unsloth/Qwen3.6-35B-A3B-GGUF`↗	GGUF（MoE）	~3B active	~12 GB	质量/VRAM 比例最佳；适合 24 GB
`unsloth/Qwen3.6-27B-MTP-GGUF`↗	GGUF（MTP）	27B	~36 GB	multi-token prediction，解码更快
`unsloth/gemma-4-31b-it-GGUF`↗	GGUF	31B	~40 GB	最高质量的稠密 Gemma 4
`unsloth/GLM-5.2-GGUF`	GGUF（MoE）	large	multi-GPU	前沿 agentic；使用 `--tensor-parallel-size` 分片
`unsloth/Deepseek-v4-GGUF`	GGUF（MoE）	large	multi-GPU	前沿代码能力；跨 GPU 分片

> 注意： Unsloth 仓库名称遵循 `unsloth/<model>-GGUF` 约定——拉取前请先在 Unsloth 组织页面↗确认准确路径，尤其是大型 `GLM-5.2` 和 `Deepseek-v4` 构建，它们通常会拆分为

> 多个文件。如果你需要 AWQ 或 GPTQ INT4 版本，则可以在 huggingface.co/models↗ 中搜索，例如 `qwen3.6 AWQ` 或 `gemma-4 GPTQ`。

Ascend NPU — ModelSlim (`msModelSlim`)。 在华为 Ascend NPU 上，请使用 Ascend 的 `msModelSlim` 工具链进行量化，它会生成可通过 Alauda AI 的 InferNex surface 在 `vLLM-Ascend` engine 上提供服务的 **W8A8 / W4A8** 权重。推荐模型中已有两个以经过验证、基准测试的 recipe 形式提供：

模型	**ModelSlim** 格式	拓扑	**Recipe**
Qwen3.6-27B	W8A8（~33 GB）	Ascend 910B4 ×8（2 × TP=4）	Qwen3.6-27B (W8A8)
DeepSeek-V4-Flash	W4A8（~151 GB）	Ascend 910B4 ×8（1 × TP=8）	DeepSeek-V4-Flash (W4A8)

如果要在 NPU 上量化你自己的权重，请使用 ModelSlim workbench 镜像及其验证 notebook（参见 创建 Workbench）。

硬件适配指南

加速器内存	示例硬件	推荐模型
12–16 GB	L4, A10, RTX 4070	`Qwen3.6-35B-A3B`（MoE，约 3B active）或 `Gemma4-12B-it`
24 GB	A30（24G），RTX 4090	`Qwen3.6-35B-A3B`（MoE）——推荐默认选择
40–48 GB	A40, L40S, A6000	`Qwen3.6-27B-MTP` 或 `Gemma4-31B-it`（GGUF）
80 GB	A100-80G, H100, H800	`Qwen3.6-27B-MTP` 或 `Gemma4-31B-it`，FP16
Ascend NPU（8 × 32 GB）	Ascend 910B4 ×8	`Qwen3.6-27B`（W8A8）或 `Deepseek-v4`（W4A8），ModelSlim —— 参见 已验证模型

加速器内存	示例硬件	推荐模型
Multi-GPU / multi-card（2–8×）	2–8 × 80 GB	`GLM-5.2` 或 `Deepseek-v4`（大型 MoE，tensor parallel）

其他选择建议：

- 保持在这五个推荐家族之内。 这些模型都经过 instruction-tuned，并原生支持 tool/function calling——这是智能体能够可靠编辑文件的前提。偏离到模型卡未提及 tool calling 的模型，通常意味着智能体可以聊天，但不能执行操作。
- 在最终确定前，确认你所选模型存在匹配的 **vLLM parser**。 参见 在运行时启用 tool calling。请在你所用版本的 vLLM 文档中验证 `--tool-call-parser`（以及适用于 GLM-5.2、Qwen3.6 和 Deepseek-v4 的 `--reasoning-parser`）。
- 为上下文长度预留预算。 编码智能体会发送很大的提示词（system prompt + 文件和仓库上下文）。请选择其上下文窗口能够覆盖你预期最大 prompt 的模型，并记住更长的 `--max-model-len` 会为每次请求消耗更多 KV cache，从而降低并发能力。
- 量化在本地部署中是放大器。 GGUF（GPU，经由 Unsloth）或 W8A8/W4A8（Ascend NPU，经由 ModelSlim）能够让你在相同内存中容纳明显更强的模型，而这通常比原始 FP16 精度更影响智能体质量。
- **MoE** 模型尤其高效。 `Qwen3.6-35B-A3B`、`GLM-5.2` 和 `Deepseek-v4` 每个 token 只激活其参数的一小部分，同时保留更大的知识库，以远低于稠密模型的 VRAM 成本提供接近稠密模型的质量。

调优 inference service 性能

编码智能体的流量形态很有特点：长且高度重复的提示词（同一个 system prompt 和仓库上下文会在每轮重复发送）、短交互请求的突发流量，以及对首 token 延迟的敏感性。请针对这些特点进行调优：

- 启用 **prefix caching**（`--enable-prefix-caching`）。这是对编码智能体影响最大的单项参数：共享的提示前缀会在多轮之间复用，而不是每次重新计算，从而显著降低 prefill 成本和延迟。参见 Automatic Prefix Caching — vLLM ↗。
- 将 `--gpu-memory-utilization` 提高到 `0.90–0.95` 左右，以增大 KV cache，从而提升并发能力和可支持的上下文长度。

- 合理设置 `--max-model-len`。 将其设为智能体实际需要的最大上下文，而不是模型的理论上限——每增加一个 token 的容量都会消耗 KV-cache 内存。
- 当长提示在并发下引发延迟尖峰时，启用 **chunked prefill**（`--enable-chunked-prefill`），以避免 decode 步骤被大规模 prefill 饿死。注意 CLI 示例 默认关闭了它。
- 允许 **CUDA graphs** 以获得稳定状态下更低的延迟：CLI 示例设置了 `ENFORCE_EAGER=True`（eager mode，启动更快但运行更慢）。服务稳定后，切换到 non-eager 以捕获 CUDA graphs，但启动时间会更长。
- 使用 `--max-num-seqs` 和 `--max-num-batched-tokens` 调整 **batching**，在你的并发水平下平衡吞吐量与单请求延迟。
- 使用 **FP8 KV cache**（`--kv-cache-dtype fp8`）在内存紧张时延长上下文长度并提高并发。
- 当模型放不进单卡时，通过 `--tensor-parallel-size` 在多 **GPU** 间分片。
- 为更低的交互延迟启用 **MTP / speculative decoding**——推荐的 `Qwen3.6-27B-MTP` 和 `Deepseek-v4` 自带原生 multi-token-prediction head。参见下文 启用 MTP（multi-token prediction） 以及 vLLM Inference Services 的 Speculative Decoding。
- 注意 **autoscaling** 和 **cold starts**。 对于交互式单用户智能体使用，请保持 `minReplicas: 1`——从零扩缩容会带来数分钟的 cold start，在任务进行中会非常痛苦。对于突发性的多开发人员使用，请有意识地配置 autoscaling；参见 为 Inference Services 配置 Scaling 和 使用 KEDA 为 Inference Services 设置 Autoscaling。
- 允许长请求。 智能体轮次可能运行很久；请相应设置 Knative 的 `serving.knative.dev/progress-deadline` 注解和客户端超时时间。如果请求被截断，请参见 Inference timeout troubleshooting。

启用 MTP（multi-token prediction）

推荐模型中的两个——`Qwen3.6-27B-MTP` 和 `Deepseek-v4`——都带有原生 **multi-token-prediction**（**MTP**） head，可以在每个 decode 步骤中提议多个 token，从而降低智能体循环中的交互延迟，而无需额外管理 draft model。MTP 是一种 speculative decoding，可通过运行时的 speculative-decoding 配置启用。

vLLM。 将模型的 MTP method 传给 `--speculative-config`。 `method` 字符串和 token 数量因模型和版本而异；以下是已验证 recipe 中使用的值：

```
# Qwen3.6-27B-MTP (qwen3_5 architecture)
--speculative-config '{"method":"qwen3_5_mtp","num_speculative_tokens":
3}'

# Deepseek-v4
--speculative-config '{"method":"mtp","num_speculative_tokens":1}'
```

请根据你的 vLLM 版本确认准确的 `method`，并将完整 recipe——包括这些在 Ascend 上需要的 MTP guardrails（关闭 prefix caching、限制 `--max-num-seqs`、`enforce_eager`）——作为事实来源： Qwen3.6-27B (W8A8) 和 DeepSeek-V4-Flash (W4A8)。关于其他 speculative 方法（EAGLE-3、N-gram）以及端到端验证与回滚，请参见 vLLM Inference Services 的 Speculative Decoding。

llama.cpp（**GGUF**）。 在提供支持 MTP 的 GGUF 构建时（例如 Unsloth 的 `-MTP` 权重），请使用 `--spec-type draft-mtp` 打开 draft-MTP 路径。请根据你的 llama.cpp 构建验证该参数——不同版本中的 MTP 支持和参数名会变化。

> **NOTE**
>
> MTP 加速的是 **decode**，而不是 prefill，而且其收益取决于你流量的 proposal acceptance rate。在确定采用之前，请先在具有代表性的编码智能体工作负载上进行有无对比基准测试，并测量真实的每 token 延迟（ITL）：MTP 每个 streamed chunk 会输出多个 token，因此按 chunk 统计的延迟会低估每 token 的收益。在上面的 Ascend recipe 中，W8A8/W4A8 量化为更大的 KV cache 释放了 HBM，但本身并不会加速 MTP decode 路径。

开始进行 **vibe coding**

“Vibe coding”——通过描述意图并让智能体编写代码来快速迭代——在基础设置正确后，与自托管模型配合得很好：

1. 从一个与你的 GPU 余量匹配的推荐模型开始—— `Qwen3.6-35B-A3B` 或 `Gemma4-12B-it` 都是不错的首选；一个响应更快的小模型，通常比一个迟缓的大模型更适合交互式流程。对于 24 GB GPU，`Qwen3.6-35B-A3B`（MoE，约 3B active）是极佳的起点。
2. 为代码生成设置一个 低 **temperature**（大约 `0–0.2`），以保持编辑确定性并减少反复试错。

3. 在尝试任何实际任务之前，先用一个简单任务验证 tool calling（例如“创建一个文件并运行它”）。
4. 保持提示词聚焦——只打开或引用相关文件，让智能体的上下文保持在主题上，同时降低 prefill 成本。
5. 采用小而可审查的步骤工作，并在接受每个 diff 之前先阅读它。频繁提交，这样你就可以干净地回滚不合适的建议。

开始进行 **MLOps**

由于模型运行在你的集群内部，基于本地部署 `InferenceService` 的编码智能体非常适合用来运维平台本身——你的 manifests、配置和专有代码都不会离开环境，这在受监管场景中尤为重要。适合作为起点的任务包括：

- 生成或修改 `InferenceService` YAML——例如，“为模型 X 编写一个面向 24 GB GPU、启用了 prefix caching 和 tool calling 的 `InferenceService`。”
- 添加 autoscaling、调度或资源配置——KEDA/KPA autoscaling、考虑 CUDA 版本的调度，或 Kueue/Volcano 队列。
- 为模型生命周期编写并调整流水线和监控。
- 闭环操作：使用智能体部署一个模型，然后再用同一个本地部署模型推动后续的平台运维。

关于更详细的 MLOps 工作流——管理 InferenceServices、配置网关、迭代调优性能以及规划 fine-tuning 运行——请参见 使用编码智能体和本地部署 LLM 运行 MLOps。

故障排查

- 智能体可以聊天，但从不编辑文件或运行命令。 Tool calling 未启用，或者 parser 与模型不匹配——参见 在运行时启用 tool calling。
- **`model not found` / 404**。 智能体配置中的模型名称与 `--served-model-name` 不匹配，或者 base URL 没有以 `/v1` 结尾。
- **401 / 403**。 智能体发送的 API key 与端点或网关期望的不一致（或未发送）。
- 长任务请求超时。 增大 Knative 的 `progress-deadline` 注解以及客户端超时——参见 Inference timeout troubleshooting。

- 空闲后的第一次请求非常慢。 服务已缩容到零并正在 cold-start；对于交互式使用，请设置 `minReplicas: 1`。

参考资料

- 使用编码智能体和本地部署 LLM 运行 MLOps
- 使用 CLI 创建 Inference Service
- 为 Inference Services 配置外部访问
- 为 Inference Services 配置 Scaling
- 使用 KEDA 为 Inference Services 设置 Autoscaling
- vLLM Inference Services 的 Speculative Decoding
- 扩展 Inference Runtimes
- Tool Calling — vLLM ↗
- Reasoning Outputs — vLLM ↗
- Automatic Prefix Caching — vLLM ↗
- opencode 文档 ↗
- Codex CLI ↗
- Claude Code 文档 ↗
- LiteLLM ↗
- claude-code-router ↗

使用编码智能体与本地部署 LLM 开展 MLOps

目录

简介

设置智能体的工作环境

管理 InferenceServices 和 LLMInferenceServices

管理网关：身份验证和速率限制

调优服务性能以适配硬件

1. 定义服务级目标
2. 生成可复现的基准测试
3. 让智能体一次提出一项变更
4. 应用、测量并记录
5. 在达到 SLO 或硬件上限时停止

规划微调并生成报告

为任务选择正确工具

可复用的微调计划模板

可复用的微调报告模板

每日 MLOps 循环

最佳实践与防护措施

参考

简介

将编码智能体连接到 Alauda AI 上的自托管模型后（请参阅将编码智能体与本地部署推理服务结合使用），同一个智能体就可以在平台上处理日常 MLOps 工作。由于模型和操作目标位于同一集群中，提示词、清单、训练数据引用和基准测试结果都不会离开您的环境——这正是自托管智能体适用于受监管工作的原因。

本文介绍编码智能体最有用的四种工作流：

- 编写和管理 `InferenceService` 与 `LLMInferenceService` 资源。
- 配置推理流量网关——通过 Alauda Build of Envoy AI Gateway 实现身份验证和速率限制。
- 反复调优推理服务的性能，使其适配特定硬件。
- 规划微调运行，并根据结果生成结构化报告。

本文假设您已经在运行智能体，并且它可以访问启用了工具调用的本地 OpenAI 兼容端点。如果尚未满足，请先阅读上面的先决条件文档。

> **WARNING**
>
> 能够针对真实集群运行 `kubectl` 的编码智能体也可以删除资源。请将其 kubeconfig 限制在单个命名空间，探索期间进行任何应用操作时优先使用 `--dry-run=server`，并要求人工审查每项变更后再将其部署到生产环境。应将智能体视为拥有集群访问权限的初级工程师，而不是自主 operator。

设置智能体的工作环境

在委派 MLOps 工作之前，请为智能体提供一个小型、可靠的操作上下文。每个项目通常都值得一次性完成以下三件事：

1. 限制集群访问范围。 创建专用命名空间（例如平台示例中使用的 `mlops-demo-ai-test`），以及权限仅限于智能体应接触资源的 `ServiceAccount` / kubeconfig——通常包括 `InferenceService`、`LLMInferenceService`、`TrainJob`、`TrainingRuntime`、`AIGatewayRoute`、`AIServiceBackend`、`BackendSecurityPolicy`、`SecurityPolicy`、`BackendTrafficPolicy` 以及它们引用的 secrets/configmaps。避免授予集群范围的写入权限。

2. 固定默认硬件配置。 Platform Hardware Profiles 会为您的节点群编码 GPU 类型、污点、容忍度和节点选择器。预先选择正确的配置并告知智能体使用它——这样可以避免智能体自行编造 affinity 块。请参阅Hardware Profiles。
3. 提交智能体上下文文件。 大多数编码智能体都会读取项目级指令文件（例如 `AGENTS.md`、`CLAUDE.md` 或 `opencode.md`）。使用它记录集群名称、目标命名空间、本地模型端点、命名约定、“始终先运行 `kubectl apply --dry-run=server`”，以及智能体应遵循的内部链接。该文件创建后，后续每个提示词都会更短、更准确。

管理 InferenceServices 和 LLMInferenceServices

平台支持两种用于提供模型服务的相关资源：

- **`InferenceService`**（`serving.kserve.io/v1beta1`）——使用 CLI 创建 Inference Service 中的标准 KServe predictor。适用于单容器模型服务器（vLLM、Triton、自定义运行时）。
- **`LLMInferenceService`**——KServe 的更高层 LLM 资源，用于多组件 LLM 服务（编排 predictor、可选的 prefill/decode 分离，以及网关/推理扩展集成）。Platform Hardware Profiles 等功能会识别它，并将其与 `InferenceService` 并列提及（请参阅Hardware Profiles）。当单容器 `InferenceService` 已无法满足需求时使用它。

对于任一资源，良好的智能体循环都是相同的：

```
draft YAML → kubectl apply --dry-run=server → apply → poll status
→ smoke test → iterate
```

可作为起点的实用提示词：

- “为模型 `Qwen3.6-35B-A3B` 生成一个 `InferenceService`，使用 `aml-vllm` 运行时、硬件配置 `single-a30-24g` 和命名空间 `mlops-demo-ai-test`。启用 prefix caching 和工具调用，并使用与 Qwen3.6 匹配的 parser。运行 `kubectl apply --dry-run=server`，在应用之前向我展示与现有对象的差异。”
- “将此 `InferenceService` 转换为支持 prefill/decode 分离的 `LLMInferenceService`；保留相同的模型、硬件配置和 served-model 名称。展示变更内容及原因。”
- “列出 `mlops-demo-ai-test` 中所有 `InferenceService` 和 `LLMInferenceService` 对象、它们的 `READY` 状态以及各自提供服务的模型。标记任何已 `NotReady` 超过 10 分钟的

对象，并总结最新的 predictor pod 事件。”

对于智能体需要复现的 YAML 字段以及平台特定的标签/注解，请将使用 CLI 创建 Inference Service 作为规范示例。要对外暴露新服务，请将其指向配置 Inference Services 的外部访问。

管理网关：身份验证和速率限制

Alauda Build of Envoy AI Gateway 是 Alauda Build of KServe 的必需依赖，通过 OpenAI 兼容的 API 表面、AI 感知路由和按模型配置来处理推理流量（请参阅 Envoy AI Gateway 简介）。Envoy AI Gateway 由 Alauda AI 通过 `envoyAIGateway` 组件在 `default` `AmlCluster` 中进行管理；请参阅安装 Alauda AI。智能体非常适合编写其 CRD，因为这些 CRD 通常较为冗长：

关注点	**CRD /** 资源	来源
将请求路由到一个或多个模型后端	`AIGatewayRoute`、`AIServiceBackend`	Envoy AI Gateway
对客户端（下游）进行身份验证：API key、JWT、OIDC	`SecurityPolicy`	Envoy Gateway
对上游模型进行身份验证（连接托管提供商时）	`BackendSecurityPolicy`	Envoy AI Gateway
按路由或按模型进行速率限制和令牌预算强制执行	`BackendTrafficPolicy`（全局速率限制）或 `AIGatewayRoute` token-rate-limit 设置	Envoy Gateway / Envoy AI Gateway
TLS 终止、可观测性	标准 `Gateway` / `HTTPRoute` 和 Envoy Gateway 功能	Envoy Gateway

实际的智能体工作流如下：

1. 用业务术语告知智能体您的意图。 例如：“在 `https://ai.example.internal` 暴露 `qwen-2` 和 `deepseek-v4`，使其位于同一个 OpenAI 兼容端点之后。要求使用来自 Kubernetes `Secret`、名称为 `ai-gateway-keys` 的 `Authorization: Bearer` API key。将每个 key 限制为每分钟 60 个请求、每小时 200k 个令牌。将 `qwen-2` 流量发送到

`mlops-demo-ai-test` 中的 `qwen-2` `InferenceService`，并将 `deepseek-v4` 发送到同名的 `LLMInferenceService`。”

2. 让智能体起草 **CRD**。 在基础设施仓库下的目录中为每个资源创建一个文件，并添加注释说明每项策略决策。
3. 应用前进行验证。 要求智能体运行 `kubectl apply --dry-run=server -f ./gateway/` 并总结将发生的变更。仅在审查后应用。
4. 对新策略执行冒烟测试。 让智能体发送有效请求、未经身份验证的请求以及超出速率限制的请求，并确认预期的 200 / 401 / 429 响应。将测试作为小型脚本与清单放在一起，以便今后的变更能够重新验证。

关于每个 CRD 的确切字段结构，请参考下面链接的上游文档——版本会发生变化，智能体应读取实时规范，而不是自行编造字段。

调优服务性能以适配硬件

本节重点介绍如何让智能体驱动调优，而不是由您手动完成。服务器端旋钮因运行时而异——对于 vLLM，请参阅最佳实践：调优推理服务性能；对于 llama.cpp（GGUF），相关标志包括 `--cache-type-k` / `--cache-type-v`（KV cache 量化）、`--ctx-size`（上下文窗口）、`--parallel`（并发）和 `--reasoning` / `--reasoning-budget`（推理控制）。

高效的循环如下：

1. 定义服务级目标

在调优前固定数值。告诉智能体“足够好”是什么样：

- 预期并发下的最大首 token 延迟（TTFT）。
- 代表性提示词的最大 P95 token 间延迟或总响应时间。
- 最低可持续吞吐量（请求数/分钟或 token/秒）。
- 智能体流量将发送的最大上下文长度。
- GPU 内存余量目标（为 KV cache 增长留出 ≥10% 的空闲空间）。

2. 生成可复现的基准测试

要求智能体编写一个模拟真实流量的基准测试脚本——包括典型提示词大小、系统提示词和并发量。可用的起点包括 `vllm bench serve`、`genai-perf`，或直接驱动 `/v1/chat/completions` 的 `k6`/Python 脚本。

智能体应针对流式和非流式请求记录 **TTFT**（首 token 时间）、**ITL**（token 间延迟，单位为 ms/token）和 **TPS**（每秒 token 数）。如果模型启用了推理，智能体还必须测量推理开销比：`reasoning_tokens / content_tokens`，因为它可能在不增加用户可见输出的情况下同时提高总延迟和 TPS。

确保服务器暴露指标。对于 vLLM，请抓取 OpenAI 兼容服务器内置的 `/metrics` 端点。对于 llama.cpp，请在抓取 `/metrics` 之前，通过 `--metrics` 启用指标端点。如果指标端点不可用，则通过 API 退回到按请求测量延迟。

3 3. 让智能体一次提出一项变更

将基准测试输出和当前 YAML 提供给智能体。要求它提出一项变更并说明预期效果，例如：

- “添加 `--enable-prefix-caching` 并重新运行；预期：降低重复系统提示词前缀的 TTFT（vLLM）。”
- “将 KV cache 切换为 `--cache-type-k q8_0 --cache-type-v q8_0` 并重新运行；预期：在 GPU 内存有限时容纳更长的上下文（llama.cpp / GGUF）。”
- “设置 `--reasoning-budget 2048`，而不是 `-1`；预期：限制推理开销，为内容输出提供更多 token（llama.cpp）。”
- “增加 `--max-num-seqs`；预期：以提高 P95 延迟为代价提升吞吐量（vLLM）。”

每次迭代只进行一项变更，才能明确归因。

4 4. 应用、测量并记录

智能体更新 `InferenceService` YAML，应用它，等待 `READY`，然后重新运行基准测试。它还应检查 GPU 级指标：利用率（持续推理预期为 70–90%）、内存使用量（超过 95% 时标记）和功耗。每次运行都在 markdown 表格中追加一行，记录配置差异、首 token 延迟、ITL、TPS 和 GPU 指标。

5 5. 在达到 SLO 或硬件上限时停止

> 当满足 SLO，或下一个合理旋钮变成“不同硬件”或“不同模型”时，循环结束——此时智能体应明确说明，而不是继续无谓尝试。常见上限包括：目标上下文长度下 KV cache 已饱和、tensor-parallel 扩展不再线性、单请求延迟受 decode 限制，或 GPU 内存余量低于 5%。

关于模型大小与 GPU 内存的选择，请参阅前一篇文档中的选择适合硬件的模型部分。关于自动扩缩容和冷启动的权衡，请参阅配置 Inference Services 的扩缩容。关于交互式延迟优化，请参阅 vLLM Inference Services 的推测解码。

规划微调并生成报告

微调有两种失败模式，编码智能体尤其擅长防止它们：跳过规划步骤（“直接运行 SFT”）以及跳过报告步骤（“loss 看起来没问题”）。智能体的任务是让这两步都明确化。

为任务选择正确工具

情况	推荐工具	参考
交互式探索、小型数据集、一到两块 GPU	Workbench Notebook	使用 Notebooks 进行微调
生产级 SFT / OSFT，自动内存管理	Training Hub	使用 Training Hub 微调 LLM
可复用模板、许多运行、在 Kueue 上计划运行/批处理	Kubeflow Trainer v2 + LlamaFactory	使用 Kubeflow Trainer v2 进行微调
已调优模型需要在提供服务前适配更小的 GPU	LLM Compressor	使用 Alauda AI 的 LLM Compressor

可复用的微调计划模板

在提交任何作业之前，让智能体填写此模板，并将结果与训练代码一同提交。这样可以将“我们打算做什么”与“我们实际运行了什么”分开，而这正是报告稍后所需的比较。

```
# Fine-tuning plan: <run-id>

## Objective

- Business goal:
- Success metric (what improves; how it's measured):
- Acceptance threshold (minimum acceptable score on the metric):

## Base model

- Model and revision:
- Why this base (capability, license, context window, tool-calling suppor
t):

## Dataset

- Source(s) and license:
- Size (examples / tokens):
- Format (e.g., JSONL chat messages):
- Splits (train / eval / held-out):
- Known biases or contamination risks:

## Method

- Approach (SFT / LoRA / QLoRA / OSFT / continued pre-train):
- Justification vs. the alternatives:
- Tool (Training Hub / Kubeflow Trainer v2 / Notebook / LlamaFactory):

## Compute budget

- Hardware (GPU type, count, hours):
- Hardware Profile to use:
- Estimated cost / wall-clock:

## Hyperparameters

- Effective batch size, max_tokens_per_gpu, lr, epochs, scheduler, seed:
- Checkpoint cadence and retention:

## Evaluation plan

- Benchmarks (public + internal):
- Eval harness and seed:
```

```
- Comparison baselines (the base model, prior runs):

## Risks and rollback

- What could go wrong (catastrophic forgetting, tool-calling regression,
license conflict):
- How we'll detect it:
- Rollback (which model artifact to revert to):
```

实用提示词：“读取 `plan.md`。起草一个 Kubeflow Trainer v2 `TrainingRuntime` 和 `TrainJob`（或 Training Hub notebook），在命名空间 `mlops-demo-ai-test` 中严格执行此计划。突出显示计划中含糊的字段，并在猜测前询问我。”

可复用的微调报告模板

作业完成后，让智能体读取训练日志、评估输出和资源指标，并填写此报告。将报告提交到计划旁边。

```
# Fine-tuning report: <run-id>

## Provenance

- Plan: link to plan.md and its commit SHA
- TrainJob / Notebook: name, namespace, start/end time
- Hardware actually used (vs. planned):
- Model artifact location (S3 URI / PVC URI / OCI image URI):

## Training summary

- Steps / epochs completed:
- Final training loss; loss trend (link to TensorBoard / MLflow run):
- Throughput (tokens/sec, samples/sec):
- Wall-clock and GPU-hours:
- Anomalies (loss spikes, restarts, OOMs):

## Evaluation results

- Headline metric vs. baseline and acceptance threshold:
- Per-benchmark scores table (this run, base model, prior best):
- Tool-calling sanity check (pass/fail with example):
- Qualitative samples (3–5 prompts; this run vs. base, side by side):

## Cost

- GPU-hours, $ (if applicable), $/percentage-point of improvement:

## Decision

- Promote / re-run / abandon:
- If promote: which `InferenceService` to update and how (image, storageU
ri, runtime flags):
- If re-run: what to change in the next plan.md:

## Next actions

- Owner / date:
```

实用提示词："为 `mlops-demo-ai-test` 中的 TrainJob `qwen-coder-sft-2026-05-29` 生成 `report.md`。从 MLflow 运行 `<id>` 获取指标，从 pod 获取训练日志，并从 `s3://aml-`

`evals/<run-id>/` 获取评估结果。与之前的运行 `qwen-coder-sft-2026-05-15` 进行比较。如果某个部分无法根据现有数据填写，请标记为 `TODO`，不要编造数字。”

对于实验跟踪和运行元数据，Kubeflow 上的 MLflow 是平台原生选项；告诉智能体在训练代码内部将日志记录到那里，使报告拥有真实的数据来源。

每日 MLOps 循环

在完成上述设置后，智能体可以驱动以下端到端流程：

1. 分类处理。 “列出我的命名空间中的推理服务，找出任何 `NotReady` 或意外缩放到零的服务，并总结近期网关 4xx/5xx 比率。”
2. 调优。 “`qwen-2` 的 P95 超出预算。提出一项变更，应用、重新进行基准测试并报告。”
3. 更新。 “`qwen-coder-sft-2026-05-29` 有新的模型构件。起草 YAML，将其换入 `qwen-2` `InferenceService`，先将发布限制到一个 Replica，并编写冒烟测试。”
4. 规划。 “起草一个微调计划，修复我们在上周评估中发现的工具调用回归问题。说明方法选择的理由。”
5. 报告。 “昨晚的作业已完成。生成报告，并告诉我是否应晋级。”

每一步都是一个独立提示词，并有自己的差异供审查。智能体是打字员；您仍然是记录在案的工程师。

最佳实践与防护措施

- 先只读，后写入。 每项新任务都先要求智能体读取状态（`get`、`describe`、日志、指标），并在进行变更前*描述它将采取的操作*。
- 始终执行 `--dry-run=server`。 将其作为智能体上下文文件中的长期规则；在涉及 `kubectl apply` 的每个提示词中都提及它。
- 每次迭代只进行一项变更。 尤其是性能调优时，同时进行两项变更会掩盖究竟是哪一项发挥了作用。
- 绝不允许智能体编造指标。 要求它引用获取每个数字所依据的文件、日志或运行 ID，并在数据缺失时标记 `TODO`。

- 考虑推理开销。 对启用推理的模型进行基准测试时，同时报告 token 总数以及推理与内容的细分。一个模型在 50 个内容 token 之前输出 8,000 个推理 token，即产生 160:1 的开销——这会主导延迟和 TPS。对于 llama.cpp，使用服务器端 `--reasoning-budget` 对其进行限制；对于其他运行时，仅使用已部署服务接受且有文档说明的请求时控制项。
- 让循环保持在本地。 确认任何智能体配置中的备用模型都没有指向托管提供商（有关需要检查的每个智能体设置，请参阅连接编码智能体）。
- 提交所有内容。 计划、报告、生成的 YAML 和基准测试脚本全部提交到 Git，以便下一位人员——或下一个智能体——从当前进度继续工作。

参考

- 将编码智能体与本地部署推理服务结合使用
- 使用 CLI 创建 Inference Service
- 配置 Inference Services 的外部访问
- 配置 Inference Services 的扩缩容
- vLLM Inference Services 的推测解码
- 扩展 Inference Runtime
- Envoy AI Gateway——简介
- Hardware Profiles
- 使用 Kubeflow Trainer v2 进行微调
- 使用 Training Hub 微调 LLM
- 使用 Notebooks 进行微调
- 使用 Alauda AI 的 LLM Compressor
- Kubeflow 上的 MLflow
- Envoy AI Gateway 上游文档↗
- Envoy Gateway 上游文档↗
- KServe LLMInferenceService↗
- vLLM 基准测试↗

组件

本章所依赖的组件的介绍和安装指南。

Kubeflow

Alauda 对 Kubeflow 的支持

安装 Kubeflow Operators

环境准备

组件上架

常见问题

在 Kubernetes

如何配置 Kubef

部 S

ow

败：

升级 Kubeflow Operators

从 `v1.x`（Cluster Plugin）迁移到 `v26.3.0`（OLM Operator）

`kfbase` 的升级说明（`v1.x` 历史版本）

数据科学流水线

Data Science Pipelines operato

它部署的内容

安装 Data Science Pipelines

前提条件

关键概念
与共享 Kubeflow Pipelines 安装的互斥关系
何时使用
上架 Operator Package
安装 Operator

KubeRay

Alauda Build of KubeRay Opera

概述
关键特性
使用场景

安装 KubeRay Operator

前提条件

Spark Operator

Alauda Build 的 Spark operator

概述
主要特性
使用场景

安装 Spark Operator

前提条件
上传 Operator 软件包
安装 Operator

MLflow

MLflow

- 它会部署什么
- 关键概念
- 多租户模型
- 何时使用它

安装 MLflow

- 前提条件
- 上传 Operator 软件包
- 启用 MLflow
- 创建 MLflow tracking server
- 验证
- 高可用性和存储
- Bearer-token 客户端
- 从 v3.10.x 升级到 v3.13.0
- 故障排查

Label Studio

Label Studio

- Label Studio
- Core Concepts
- Documentation

安装 Label Studio

- 下载
- 部署
- 配置
- 访问地址
- 用户管理

快速入门

- S3 集成
- 其他资源

主要功能

- 多用户标注
- 多类型数据支持
- 灵活的标注配置
- 数据导入/导出
- Machine Learni

Feast

Alauda Build of Feast

- 常见需求
- 托管资源
- 核心概念
- 典型工作流
- Operator 职责
- 延伸阅读

安装 Feast

- 上传 Operator 软件包
- 启用 Feast
- 验证安装
- 后续步骤

快速入门

- 先决条件
- FeatureStore C
- 常见服务配置
- 常见持久化模式
- 关于 Secret 的
- 特征仓库初始化
- 部署 FeatureSt
- 服务与访问
- 使用 Feast SDK
- 配置授权
- 延伸阅读

Alauda 对 Kubeflow 的支持

Alauda 对 Kubeflow 的支持

安装 Kubeflow Operators

环境准备

组件上架

常见问题

在 Kubernetes

如何配置 Kubef

部 S

ow

败：

升级 Kubeflow Operators

从 `v1.x`（Cluster Plugin）迁移到 `v26.3.0`（OLM Operator）

`kfbase` 的升级说明（`v1.x` 历史版本）

Alauda 对 Kubeflow 的支持

Alauda 对 Kubeflow 的支持提供了一个 Kubernetes 原生机器学习平台，使用户能够大规模构建、部署和管理机器学习模型。该平台集成了 Kubeflow 生态系统中的各种组件，例如用于工作流编排的 Kubeflow Pipelines、用于训练作业管理的 Kubeflow Trainer v2，以及用于模型版本控制和管理的 Kubeflow Model Registry。

从 **v26.3.0**（Alauda AI v2.3）开始，Alauda Kubeflow 组件以 **OLM Helm Operators** 的形式提供，并从 ACP OperatorHub 安装：`kfbase-operator`、`kfp-operator` 和 `kubeflow-trainer-operator`。早期的集群插件形式（`v1.x`）已停用；有关迁移路径，请参阅 升级 Kubeflow Operators。对应的上游版本为 `kubeflow/manifests` release 26.03↗。

有关 Kubeflow 组件和功能的更多详细信息，请参阅 Kubeflow 文档↗。

> 注意： 要使用 Kubeflow 组件，您需要将命名空间 PSA 设置为 privileged。如果使用 Kubeflow 组件时遇到权限问题，请联系您的集群管理员，将命名空间 PSA 设置为 privileged。

安装 Kubeflow Operators

本文介绍如何在 Alauda AI 2.3 及更高版本中部署与 Kubeflow 相关的 operator。

从 **v26.3.0**（Alauda AI v2.3）开始，Kubeflow 以 OLM Helm Operators 的形式发布，而不是早期的 Cluster Plugin 形态。安装通过 ACP **OperatorHub** 完成，而不是通过 **Cluster Plugins**。这些 operator 封装了上游 `kubeflow/manifests` 26.03 charts。

支持的 operator：

- `kfbase-operator`：Kubeflow 基础组件，包括身份验证和授权、中心监控面板、Notebooks、PVC Viewer、TensorBoards、Volumes、Model Registry UI、KServe Endpoints UI，以及 Model Catalog API service。负责管理 `KubeflowBase` CR。架构：`amd64`、`arm64`。
- `kfp-operator`：Kubeflow Pipelines（KFP runtime 2.16.0）。负责管理 `KubeflowPipelines` CR。架构：仅 **`amd64`**（根据 Alauda AI v2.3 支持的配置，Kubeflow Pipelines 仅支持 x86_64）。
- `kubeflow-trainer-operator`：Kubeflow Trainer v2（controller-manager 2.1.0 + JobSet 0.10.1）。负责管理 `KubeflowTrainer` CR。架构：`amd64`、`arm64`。替代已弃用的 `kftraining` 插件。
- `model-registry-operator`：Kubeflow Model Registry operator（形态不变）。

> 注意：`kftraining` Cluster Plugin（Kubeflow Training Operator v1）在早期版本中已弃用，并已在 v26.3.0 中退役。请改用 `kubeflow-trainer-operator`（Trainer v2）。

目录

环境准备
配置 Dex 重定向
配置 `oauth2-proxy` 插件
配置 ASM v1（已弃用）
配置 ASM v2
组件上架
部署步骤
1. 部署 `kfbase-operator`（Kubeflow Base）
2. 创建 Kubeflow 用户命名空间并绑定用户
3. 将用户绑定到现有命名空间
4. 部署 `kfp-operator`（Kubeflow Pipelines）
5. 部署 Kubeflow Model Registry
6. 部署 `kubeflow-trainer-operator`（Kubeflow Trainer v2）

环境准备

开始之前，请确保满足以下前提条件：

1. ACP 环境可用且正在运行。
2. Alauda AI 已部署。v26.3.0 operator 集需要 Alauda AI 2.3 或更高版本。
3. 已安装 Alauda Build of KServe。
4. 业务集群中已部署 ASM，Kubeflow 将运行在该集群中。如果尚未安装 ASM，请先部署后再继续。ASM v1 已弃用。请尽可能使用 ASM v2。
5. LeaderWorkerSet 已安装并就绪。LWS 通常由 Alauda AI 通过 `default` `AmlCluster` 中的 `lws` 组件进行管理；参见安装 Alauda AI。仅当你计划部署 `kubeflow-trainer-operator` 时才需要它。
6. 已按照下文配置 `oauth2-proxy` 插件。

配置 **Dex** 重定向

> 注意： 在安装 `kfbase-operator` 并创建其 `KubeflowBase` CR 之前，请先配置用于 Dex 重定向的平台访问 URL。此步骤可能会更新平台 CA 证书。如果你在配置 `oauth2-proxy` 后证书发生变化，`oauth2-proxy` 配置可能会失败。

在 管理员 > **System Settings** > **Platform Parameters** 中，单击 **Platform Access URLs** 旁边的 **Edit**，并添加格式为 `https://<your-kubeflow-domain>` 的重定向 URL，例如 `https://kubeflow.example.com`。

- `<your-kubeflow-domain>` 必须与 `KubeflowBase` CR 中的 `spec.global.kubeflowHost` 一致。

配置 `oauth2-proxy` 插件

获取平台 Dex CA 证书，供后续在 **Global** 集群中使用：

```
crt=$(kubectl get secret -n cpaas-system dex.tls -o jsonpath='{.data.tls
\.crt}')
echo -n $crt | base64 -d
```

配置 **ASM v1**（已弃用）

在 global 集群中，或在 ACP **Platform Management** > **Resource Management** 中，更新 `ServiceMesh` 资源，并在 `spec` 下添加以下内容。

> 注意： 如果已经配置了 `spec.values.pilot.jwksResolverExtraRootCA`，则只更新 `spec.meshConfig.extensionProviders`。添加新条目时不要删除现有条目。

```
spec:
  overlays:
    - kind: IstioOperator
      patches:
        - path: spec.values.pilot.env.PILOT_JWT_PUB_KEY_REFRESH_INTERVAL
          value: 1m
        - path: spec.values.pilot.jwksResolverExtraRootCA
          value: |
            -----BEGIN CERTIFICATE-----
            <YOUR_DEX_CA_CERTIFICATE_BASE64_HERE>
            -----END CERTIFICATE-----
        - path: spec.meshConfig.extensionProviders
          value:
            - envoyExtAuthzHttp:
                headersToDownstreamOnDeny:
                  - content-type
                  - set-cookie
                headersToUpstreamOnAllow:
                  - authorization
                  - path
                  - x-auth-request-user
                  - x-auth-request-email
                  - x-auth-request-access-token
                includeAdditionalHeadersInCheck:
                  X-Auth-Request-Redirect: http://%REQ(Host)%%REQ(:PATH)%
                includeRequestHeadersInCheck:
                  - authorization
                  - cookie
                  - accept
                port: 80
                service: oauth2-proxy.kubeflow-oauth2-proxy.svc.cluster.l
ocal
              name: oauth2-proxy-kubeflow
```

配置 **ASM v2**

> 注意： 如果仍然存在任何 ASM v1 webhooks，请先将其删除。否则 Kubeflow 身份验证可能会失败。

```
kubectl delete validatingwebhookconfigurations istiod-default-validator
kubectl delete mutatingwebhookconfigurations istio-sidecar-injector-1-22
kubectl delete mutatingwebhookconfigurations istio-revision-tag-default
```

在 ACP 中，进入 管理员 > **MarketPlace** > **OperatorHub**，找到 **Alauda Service Mesh v2**，打开 **All Instances** 选项卡，定位类型为 **Istio** 的实例，例如 `default`，单击 **Update**，并在 `spec` 下添加以下内容：

```
spec:
  values:
    pilot:
      env:
        PILOT_JWT_PUB_KEY_REFRESH_INTERVAL: 1m
      jwksResolverExtraRootCA: |
        -----BEGIN CERTIFICATE-----
        <YOUR_DEX_CA_CERTIFICATE_BASE64_HERE>
        -----END CERTIFICATE-----
    meshConfig:
      extensionProviders:
        - envoyExtAuthzHttp:
            headersToDownstreamOnDeny:
              - content-type
              - set-cookie
            headersToUpstreamOnAllow:
              - authorization
              - path
              - x-auth-request-user
              - x-auth-request-email
              - x-auth-request-access-token
            includeAdditionalHeadersInCheck:
              X-Auth-Request-Redirect: http://%REQ(Host)%%REQ(:PATH)%
            includeRequestHeadersInCheck:
              - authorization
              - cookie
              - accept
            port: 80
            service: oauth2-proxy.kubeflow-oauth2-proxy.svc.cluster.local
          name: oauth2-proxy-kubeflow
```

组件上架

下载以下 operator 的 bundle 软件包，然后按照上架软件包进行上传。这些 bundle 会将 operator 注册到 ACP OperatorHub。

- `kfbase-operator`：Kubeflow 基础功能（负责管理 `KubeflowBase` CR）。
- `kfp-operator`：Kubeflow Pipelines（负责管理 `KubeflowPipelines` CR）。仅支持 amd64。
- `kubeflow-trainer-operator`：Kubeflow Trainer v2（负责管理 `KubeflowTrainer` CR）。替代已弃用的 `kftraining`。
- `model-registry-operator`：Kubeflow Model Registry operator。

部署步骤

1. 部署 `kfbase-operator`（Kubeflow Base）

在 管理员 > **MarketPlace** > **OperatorHub** 中找到 `kfbase-operator`，然后单击 **Install**。接着打开 **All Instances** 选项卡，并创建一个 `KubeflowBase` CR，将 `spec.global.kubeflowHost` 设置为你的 Kubeflow 域名。等待部署完成。

部署完成后：

- 在 管理员 > **System Settings** > **Platform Parameters** 中，确认 **Platform Access URLs** 包含格式为 `https://<your-kubeflow-domain>` 的地址，其中 `<your-kubeflow-domain>` 与 `KubeflowBase` CR 中的 `spec.global.kubeflowHost` 一致。
- 配置 DNS 解析，或添加本地 hosts 条目，使 `<your-kubeflow-domain>` 解析到 `kubectl -n istio-system get gateway kubeflow-external-gateway` 分配的 IP 地址。

部署完成后，Kubeflow 入口会出现在 Alauda AI 的 **Tools** 下。

有关升级相关操作，请参见升级 Kubeflow Operators。

2. 创建 Kubeflow 用户命名空间并绑定用户

在用户首次登录 Kubeflow 之前，需要先将 ACP 用户绑定到某个命名空间。以下示例将创建命名空间 `kubeflow-admin-cpaas-io`，并将 `admin@cpaas.io` 设为所有者。

> 注意： 如果在 Alauda AI 部署期间已经创建了该 `Profile` 资源，则可以跳过此步骤。
> 注意： 在创建 Notebook 实例及类似工作负载之前，你可能需要降低用户命名空间的 Pod Security Admission 级别。

```yaml
apiVersion: kubeflow.org/v1beta1
kind: Profile
metadata:
  name: kubeflow-admin-cpaas-io
spec:
  owner:
    kind: User
    name: "admin@cpaas.io"
```

3. 将用户绑定到现有命名空间

如果 Alauda AI 已经部署，且命名空间 `kubeflow-admin-cpaas-io` 已存在，那么 `Profile` 也可能已经存在。如果该命名空间仍未出现在 Kubeflow 中，请创建以下资源，将该账户绑定到该命名空间：

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: default-editor
  namespace: kubeflow-admin-cpaas-io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: default-editor
  namespace: kubeflow-admin-cpaas-io
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kubeflow-edit
subjects:
  - kind: ServiceAccount
    name: default-editor
    namespace: kubeflow-admin-cpaas-io
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: user-admin-cpaas-io-clusterrole-admin
  namespace: kubeflow-admin-cpaas-io
  annotations:
    role: admin
    user: "admin@cpaas.io"
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kubeflow-admin
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: User
    name: "admin@cpaas.io"
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: user-admin-cpaas-io-clusterrole-admin
  namespace: kubeflow-admin-cpaas-io
  annotations:
```

```
    role: admin
    user: "admin@cpaas.io"
spec:
  rules:
    - from:
        - source:
            ## for more information see the KFAM code:
            ## https://github.com/kubeflow/kubeflow/blob/v1.8.0/components/access-management/kfam/bindings.go#L79-L110
            principals:
              ## required for Kubeflow notebooks
              ## TEMPLATE: "cluster.local/ns/<ISTIO_GATEWAY_NAMESPACE>/sa/<ISTIO_GATEWAY_SERVICE_ACCOUNT>"
              - "cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"

              ## required for Kubeflow pipelines
              ## TEMPLATE: "cluster.local/ns/<KUBEFLOW_NAMESPACE>/sa/<KFP_UI_SERVICE_ACCOUNT>"
              - "cluster.local/ns/kubeflow/sa/ml-pipeline-ui"
      when:
        - key: request.headers[kubeflow-userid]
          values:
            - "admin@cpaas.io"
```

4. 部署 `kfp-operator`（Kubeflow Pipelines）

在 管理员 > **MarketPlace** > **OperatorHub** 中找到 `kfp-operator`，然后单击 **Install**。接着在 operator 实例命名空间中创建一个 `KubeflowPipelines` CR。待 CR 完成调谐后，KFP runtime 组件将被部署，且与 pipeline 相关的功能将在 Kubeflow UI 中可用。

> 注意： `kfp-operator` 仅支持 amd64。请勿在仅支持 arm64 的集群上安装。
> 注意： 只有在 `KubeflowPipelines` 完成调谐后，Kubeflow UI 中才会显示 pipeline 相关功能。

5. 部署 Kubeflow Model Registry

在 管理员 > **MarketPlace** > **OperatorHub** 中找到 **Model Registry Operator**，然后单击 **Install**。

安装 operator 后，打开 **All Instances** 选项卡，并在用户的命名空间中创建一个 `ModelRegistry` 实例。

> 注意： 请在已经绑定到 Kubeflow `Profile` 的命名空间中创建该实例。否则不会显示 Model Registry UI。

创建实例时，请按需配置以下字段：

- 名称：Model Registry 实例的名称。
- 命名空间：实例运行所在的命名空间。该命名空间必须已绑定到 Kubeflow `Profile`。
- **MySQL Storage Class**：用于 Model Registry 元数据的存储类，例如 `standard`。
- **MySQL Storage Size**：元数据数据库的存储大小。默认值为 `10Gi`。
- **DisplayName**：Model Registry 实例的显示名称。
- **Description**：实例的简短描述。

> 注意： 实例启动后，请刷新 Kubeflow 左侧导航中的 **Model Registry** 入口，以查看新实例。在创建第一个实例之前，Model Registry 页面为空。
> 注意： Model Registry 实例会限制来自其他命名空间的网络请求。若要允许其他命名空间访问，请编辑该实例的 `authorizationpolicy`，例如执行 `kubectl -n <your-namespace> edit authorizationpolicy <model-registry-name>`，并根据 Istio 文档更新策略。
> 注意： 你可以在不同的命名空间中部署多个 Model Registry 实例。每个实例都是独立的。

6. 部署 `kubeflow-trainer-operator`（Kubeflow Trainer v2）

> 注意： 如果仍然安装着已弃用的 `kftraining` Cluster Plugin（来自 v26.3.0 之前的集群），请先卸载它，然后再安装 `kubeflow-trainer-operator`。
> 注意： LeaderWorkerSet 是 `kubeflow-trainer-operator` 的依赖项。在部署 `kubeflow-trainer-operator` 之前，请通过 `default` `AmlCluster` 中的 `lws` 组件确认 LWS 已启用并就绪；参见安装 Alauda AI。
> 注意： `kubeflow-trainer-operator` 的 v26.3.0 版本与上游 `kubeflow/manifests` 26.03 对齐，并随附 **Trainer v2.1.0 + JobSet v0.10.1**。对于某些 OLM CatalogSource 已经声明更高 trainer 版本（`>=2.2.0`）的集群，请使用 `installPlanApproval: Manual` 和 `startingCSV: kubeflow-trainer-operator.v2.1.0` 进行安装，以防止 OLM 自动升级并超过 26.03 的版本锁定。

在 管理员 > **MarketPlace** > **OperatorHub** 中找到 `kubeflow-trainer-operator`，单击 **Install**，如果需要锁定版本，请选择 `Manual` install-plan approval，然后打开 **All Instances** 选项卡，并创建一个启用了 `JobSet` 的 `KubeflowTrainer` CR。

常见问题

目录

在 Kubernetes 中设置 PSA=restricted 时如何使用 Kubeflow 插件

如何配置 Kubeflow 在登录时使用替代平台地址？

v26.3.0 及更高版本（OLM operator 安装）

v1.x（集群插件安装 — 旧版）

如何使用外部 S3/MinIO 存储启动 Kubeflow Pipeline Run

配置 Kubeflow Notebook 使用自定义 GPU 资源

Pod 启动失败：Probe 超时（kube-ovn 环境）

在 Kubernetes 中设置 PSA=restricted 时如何使用 Kubeflow 插件

如果您的 namespace 设置了 PSA=restricted，在使用 Kubeflow 组件时可能会遇到错误，例如创建 notebook、kubeflow pipeline run 等。为了解决这个问题，您需要将当前 namespace 的默认 PSA 修改为 `baseline`：

```
kubectl label --overwrite ns [NAMESPACE_NAME] pod-security.kubernetes.io/audit=baseline
kubectl label --overwrite ns [NAMESPACE_NAME] pod-security.kubernetes.io/enforce=baseline
kubectl label --overwrite ns [NAMESPACE_NAME] pod-security.kubernetes.io/warn=baseline
```

> 注意：您可能需要咨询集群管理员，以确认更改 PSA 是被允许的。

如何配置 **Kubeflow** 在登录时使用替代平台地址？

在某些环境中，平台访问地址被配置为内部网络地址，用户需要通过“替代平台地址”进行登录。在这种情况下，虽然 OIDC issuer 仍然基于原始平台地址，但登录页面 URL 必须更新为替代地址。

v26.3.0 及更高版本（**OLM operator** 安装）

步骤：

- 在 **operator-instance namespace** 中定位用于 **kfbase-operator** 安装的 `KubeflowBase` **CR**：

  ```
  kubectl get kubeflowbase -A
  ```

- 编辑 `KubeflowBase` **CR**。在 `spec.global` 下添加 `oidcAuthURL`。将 `<Alternative-Platform-Address>` 替换为实际的替代地址。

  ```
  spec:
    global:
      oidcAuthURL: https://<Alternative-Platform-Address>/dex/auth
  ```

  kfbase-operator 的 helm-operator reconcile loop 会将该更改应用到底层 chart release。

- 重启目标集群中的 **OAuth2 Proxy**：

```
kubectl rollout restart deploy -n kubeflow-oauth2-proxy oauth2-proxy
```

v1.x（集群插件安装 — 旧版）

步骤：

- 在 global 集群中定位 ModuleInfo Resource：

```
kubectl get moduleinfoes -l cpaas.io/module-name=kfbase,cpaas.io/cluste
r-name=<deployed-cluster-name>
```

- 编辑 ModuleInfo Resource。在 `spec` 下添加 `valuesOverride`：

```
spec:
  valuesOverride:
    mlops/kfbase:
      oidcAuthURL: https://<Alternative-Platform-Address>/dex/auth
```

- 重启 OAuth2 Proxy：

```
kubectl rollout restart deploy -n kubeflow-oauth2-proxy oauth2-proxy
```

如何使用外部 S3/MinIO 存储启动 Kubeflow Pipeline Run

当您使用外部 S3/MinIO 存储服务安装 Kubeflow 时，需要添加一个 “KFP Launcher” configmap，以设置当前 namespace 或用户使用的存储。您可以查看 Kubeflow 文档 https://www.kubeflow.org/docs/components/pipelines/operator-guides/configure-object-store/#s3-and-s3-compatible-provider↗ 了解更多详情。如果未设置任何配置，pipeline run 仍可能访问默认服务地址，例如 "minio-service.kubeflow:9000 "，这并不正确。

下面是一个简单示例，您可以从这里开始：

```
apiVersion: v1
data:
  defaultPipelineRoot: s3://mlpipeline
  providers: |-
    s3:
      default:
        endpoint: minio.minio-system.svc:80
        disableSSL: true
        region: us-east-2
        forcePathStyle: true
        credentials:
          fromEnv: false
          secretRef:
            secretName: mlpipeline-minio-artifact
            accessKeyKey: accesskey
            secretKeyKey: secretkey
kind: ConfigMap
metadata:
  name: kfp-launcher
  namespace: wy-testns
```

例如，您应在此 configmap 中设置以下值，以指向您自己的 S3/MinIO 存储：

- defaultPipelineRoot：用于存储 pipeline 中间数据的位置
- endpoint：S3/MinIO 服务端点。注意，不应以 "http" 或 "https" 开头
- disableSSL：是否禁用到该端点的 "https" 访问
- region：S3 region。如果使用 MinIO，任意值均可
- credentials：secrets 中的 AK/SK

添加此 configmap 后，新启动的 Kubeflow Pipeline Runs 将自动读取此配置，并保存 Kubeflow Pipeline 使用的数据。

配置 **Kubeflow Notebook** 使用自定义 **GPU** 资源

您可以添加其他 GPU 资源类型，以便 Kubeflow Notebook 网页能够创建利用这些硬件的实例，例如在使用 Ascend GPU 时。

通过运行以下命令编辑 configmap：

```
kubectl -n kubeflow get configmap | grep jupyter-web-app-config
kubectl -n kubeflow edit configmap jupyter-web-app-config-<actual-cm-suff
ix>
```

找到下面的部分，并添加您的 GPU 资源类型，例如 "your-custom.com/gpu"。

> 注意：您只能添加使用整数值的资源类型，例如 1、2、4、8。此外，您不能像使用 HAMi 时那样，同时通过 “Cores” 和 “Memory” 添加 “Virtual” 或 “Shared” GPU 资源。

```
################################################################
# GPU/Device-Plugin Resources
################################################################
gpus:
  readOnly: false

  # configs for gpu/device-plugin limits of the container
  # https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/#using-d
evice-plugins
  value:
    # the `limitKey` of the default vendor
    # (to have no default, set as "")
    vendor: ""

    # the list of available vendors in the dropdown
    #  `limitsKey` - what will be set as the actual limit
    #  `uiName` - what will be displayed in the dropdown UI
    vendors:
    - limitsKey: "nvidia.com/gpu"
      uiName: "NVIDIA"
    - limitsKey: "amd.com/gpu"
      uiName: "AMD"
    - limitsKey: "habana.ai/gaudi"
      uiName: "Intel Gaudi"
    - limitsKey: "your-custom.com/gpu"
      uiName: "Your Custom Vendor"
    # the default value of the limit
    # (possible values: "none", "1", "2", "4", "8")
    num: "none"
```

Pod 启动失败：Probe 超时（kube-ovn 环境）

症状： `kubeflow` namespace 中大量 Pod 卡在 `CrashLoopBackOff` 或 `Init:1/2` 状态，Pod Event 显示如下错误：

```
Startup probe failed: Get "http://<pod-ip>:<port>/healthz": context deadl
ine exceeded (Client.Timeout exceeded while awaiting headers)
Liveness probe failed: ...context deadline exceeded...
```

原因： kfbase 部署的 `default-allow-same-namespace` NetworkPolicy 只允许来自同一 namespace 中的 Pod 以及少量系统 namespace 的入站流量。在使用 **kube-ovn** 作为 CNI 的集群中，kubelet 发送的健康检查流量会通过 kube-ovn 的 **join subnet**（默认 `100.64.0.0/16`）到达 Pod。该流量的源 IP 不匹配任何现有的 NetworkPolicy 规则，因此会被 OVN ACL 丢弃，导致所有 probe 超时。

修复： 创建一个允许来自 kube-ovn join subnet 入站流量的 NetworkPolicy：

```
# 1. Check the CIDR of the kube-ovn join subnet
kubectl get subnet join -o jsonpath='{.spec.cidrBlock}'
# Example output: 100.64.0.0/16

# 2. Check the IP of each node on the join subnet
kubectl get nodes -o custom-columns='NAME:.metadata.name,JOIN_IP:.metadat
a.annotations.ovn\.kubernetes\.io/ip_address'

# 3. Verify whether the probe timeout is related to the NetworkPolicy (te
mporary test)
# Change the ingress of default-allow-same-namespace to [{}] (allow all i
nbound traffic),
# then observe whether the Pods recover. Be sure to revert the change aft
er confirmation.
```

```
    # First get the join subnet CIDR
JOIN_CIDR=$(kubectl get subnet join -o jsonpath='{.spec.cidrBlock}')

kubectl apply -f - <<EOF
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-kubelet-probes
  namespace: kubeflow
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: ${JOIN_CIDR}
EOF
```

注意： 不同集群的 join subnet CIDR 可能不同。请始终通过运行 `kubectl get subnet join` 获取实际值。常见默认值为 `100.64.0.0/16`。

升级 Kubeflow Operators

本页面描述在升级 Kubeflow operators 之后可能需要执行的手动操作。

有关安装步骤，请参阅 安装 Kubeflow Operators。

目录

从 `v1.x`（Cluster Plugin）迁移到 `v26.3.0`（OLM Operator）
推荐的迁移操作步骤
`v1.x` 与 `v26.3.0` 之间的变化
`kfbase` 的升级说明（`v1.x` 历史版本）
从 `v1.10.13` 或更早版本升级
从 `v1.10.9` 或更早版本升级

从 `v1.x`（Cluster Plugin）迁移到 `v26.3.0`（OLM Operator）

从 **v26.3.0**（Alauda AI v2.3）开始，Kubeflow 组件以 OLM Helm Operators（`kfbase-operator`、`kfp-operator`、`kubeflow-trainer-operator`）的形式发布，而不再使用 Cluster Plugins（`kfbase`、`kfp`、`kftraining`、`kubeflow-trainer`）。这两种形态之间不支持就地升级路径——Cluster Plugin 安装描述符（`ModuleInfo`）与 OLM `Subscription` 彼此不兼容。

推荐的迁移操作步骤

1. 备份用户数据：
   - 在每个用户命名空间中快照 Notebook PVC。
   - 导出 `Profile` CR，以及你为 Kubeflow 用户创建的任何自定义 `RoleBinding` / `AuthorizationPolicy`。
   - 通过 `kfp` CLI 导出你的 KFP pipelines、experiments 和 scheduled runs。
   - 导出任何 `TrainingRuntime` 和处于活动状态的 `TrainJob` CR。
2. 从 **AC UI**（`Cluster Plugins`）中卸载 **v1.x Cluster Plugin** 安装：按顺序移除 `kubeflow-trainer`、`kfp`、`kftraining`（如果存在）以及 `kfbase`。匹配的 `ModuleInfo` / `ModuleConfig` 资源会自动移除。

   > 警告： 在确认步骤 1 中的备份完成之前，请勿继续。如果卸载会移除 `kubeflow.org` CRD，则所有 `Profile` CR（以及它们所属的用户命名空间 / Notebook PVC）都可能被级联删除。在依赖下面的恢复步骤之前，请先验证你的用户命名空间和 PVC 在卸载后是否仍然保留。

3. 从 管理员 > **MarketPlace** > **OperatorHub** 安装 **v26.3.0 operator** 集合：
   - 先安装 `kfbase-operator`（其他 operator 依赖基础组件）。
   - 如果你需要 Kubeflow Pipelines，则安装 `kfp-operator`（仅限 amd64 集群）。
   - 如果你需要 Trainer v2，则安装 `kubeflow-trainer-operator`。
4. 创建对应的 **CR** 实例（`KubeflowBase`、`KubeflowPipelines`、`KubeflowTrainer`），并复用你在 v1.x 安装中的配置。之前在 Cluster Plugin 安装表单中设置的 chart values 现在通过 operator 的 CSV `specDescriptors` 暴露出来——大多数字段名称保持不变。
5. 恢复用户数据： 如果 `Profile` CR 及其 PVC 在卸载过程中被保留，它们会通过 Notebook controller reconcile 自动重新关联。如果它们已被移除，请先重新应用你在步骤 1 中导出的 `Profile` CR，并从快照中恢复 PVC。无论哪种情况，都要重新导入 KFP pipelines 和 TrainingRuntimes。

`v1.x` 与 `v26.3.0` 之间的变化

- 形态： Cluster Plugin → OLM Helm Operator。

- 安装描述符： `ModuleInfo` → OLM `Subscription` + `ClusterServiceVersion` + operator 自有 CR。
- **Trainer**： 移除了 `kftraining`（Training Operator v1，已弃用）；替换为 `kubeflow-trainer-operator`（Trainer v2）。
- 上游对齐： 所有 chart 重新固定到 `kubeflow/manifests` 26.03。
- 架构： `kfp-operator` 现在仅支持 amd64； `kfbase-operator` 和 `kubeflow-trainer-operator` 仍支持 amd64 + arm64。

`kfbase` 的升级说明（`v1.x` 历史版本）

从 `v1.10.13` 或更早版本升级

直到 `v1.10.13` 的版本都通过 `NodePort` 暴露 Kubeflow dashboard。升级后，推荐改为通过 gateway endpoint 访问。

升级后：

- 检查 `kfbase` plugin 配置中的 `kubeflowDomain` 字段，以获取 `<your-kubeflow-domain>`。
- 运行 `kubectl -n istio-system get gateway kubeflow-external-gateway` 以获取 gateway IP 地址。
- 更新 DNS 解析，或者更新本地 hosts 文件，使 `<your-kubeflow-domain>` 解析到该 gateway IP 地址。

如果你仍然需要使用 `NodePort`，请手动将 `istio-system/kubeflow-istio-ingressgateway` service 修改为 `NodePort` 类型，然后运行以下命令获取分配给 `443` 的端口：

```
kubectl -n istio-system get service kubeflow-istio-ingressgateway
```

随后，你可以通过以下地址访问 dashboard：

```
https://<ip-of-master-node-of-the-cluster>:<NodePort>/
```

从 `v1.10.9` 或更早版本升级

在升级之前，请在集群中设置一个默认 storage class，它将用于 `kfbase` plugin 配置中的 `pgStorageClass` 参数。如果未设置默认 storage class，升级可能会因缺少必需参数而失败。这些参数是在 `v1.10.10` 中引入的。

数据科学流水线 Operator

Data Science Pipelines operato

它部署的内容

关键概念

与共享 Kubeflow Pipelines 安装的互斥关系

何时使用

安装 Data Science Pipelines

前提条件

上架 Operator Package

安装 Operator

Data Science Pipelines operator

Data Science Pipelines operator（DSPO）是一个 Kubernetes 原生 operator，用于在 Alauda AI 上部署和管理 **Kubeflow Pipelines**（**KFP**）**v2** 栈。它基于开源的 OpenDataHub Data Science Pipelines Operator↗，使每个数据科学项目都能运行其各自独立的 **pipeline** 运行时，并通过单个 `DataSciencePipelinesApplication`（DSPA）Custom Resource 进行声明——无需争用共享的、集群级别的 pipeline 服务。

目录

它部署的内容

关键概念

与共享 Kubeflow Pipelines 安装的互斥关系

何时使用

它部署的内容

当你在某个 namespace 中创建 `DataSciencePipelinesApplication` 时，operator 会在该 namespace 中协调出一个自包含的 KFP v2 栈：

- **APIServer** — Kubeflow Pipelines API server（用于编译、上传、运行和跟踪 pipelines；也是 `kfp` Python SDK 的目标端）。
- **Persistence Agent** — 将 workflow engine 中的运行状态同步回 API server / database。

- **Scheduled Workflow controller** — 驱动周期性运行。
- **Argo Workflow controller** — 将 pipeline DAG 作为 Argo `Workflows` 执行。
- **ML Metadata（MLMD）** — 存储 pipeline/run/artifact 的 lineage，并由 Envoy gRPC proxy 作为前端。
- **Database + object storage** — 可以是 operator 部署的托管 MariaDB 和 MinIO，也可以是你自己的外部 MySQL 和 S3-compatible 存储。

关键概念

概念	说明
`DataSciencePipelinesApplication`（DSPA）	声明一个 KFP v2 栈的 namespace 级别 CR。一个 DSPA = 一个独立的 pipelines 运行时。API group `datasciencepipelinesapplications.opendatahub.io/v1`
每个 namespace 隔离	每个项目 namespace 都拥有自己的 APIServer、database 和 object storage。团队之间不会共享同一个 pipeline 服务。
外部后端 vs. 托管后端	`spec.database` / `spec.objectStorage` 可以指向外部的 MySQL / S3，也可以留给 operator 去部署托管的 MariaDB / MinIO。
外部 endpoint provider	DSPA 的 APIServer 如何暴露到集群外部是可插拔的，可通过 operator 的 `EXTERNAL_ROUTE_PROVIDER` 设置进行配置（`none`、`virtualservice`）。参见 Installation。

与共享 **Kubeflow Pipelines** 安装的互斥关系

DSPO 与 Kubeflow plugin（`kfp-operator`）创建的共享、全局作用域 Kubeflow Pipelines 安装是互斥的。若集群中存在该共享安装，operator 将拒绝协调 DSPA（`KUBEFLOW_PIPELINES_GLOBAL_CHECK=enforce`）。每个集群请选择一种模型：

- **DSPO** — 按 namespace 隔离、面向项目的 pipeline 栈（本 operator）。
- **Kubeflow plugin** `kfp-operator` — 为整个集群提供单个共享的 Kubeflow Pipelines 实例。

何时使用

当你希望每个项目或团队都拥有独立的 Kubeflow Pipelines v2 运行时——并且具备各自的 database、object storage 和访问边界——而不是共享一个集群级别的 pipelines 服务时，请使用 Data Science Pipelines operator。

安装 Data Science Pipelines

Data Science Pipelines Operator 以 **OLM Operator** 形式交付，并从平台 **OperatorHub** 安装。

目录

前提条件

上架 Operator Package

安装 Operator

外部端点提供程序（可选）

验证

前提条件

- **ACP version: v4.0 or later**（已在 v4.3 / Kubernetes 1.34 上验证）。
- 目标集群架构为 `linux/amd64`（operator 及其镜像也提供 `linux/arm64`）。
- 目标集群上可用 Operator Lifecycle Manager (OLM)（由 ACP 提供）。
- 目标集群中不得存在共享的 **Kubeflow Pipelines** 安装（`kfp-operator`）——DSPO 与其互斥（参见 Introduction）。
- **Argo Workflow CRD**。 每个 DSPA 都会运行一个 Argo Workflow controller 和 KFP v2 driver/launcher，它们需要 `workflows.argoproj.io` CRD。DSPO 的 bundle 中不包含这

些 CRD（以避免在已运行 Argo 的集群上产生 OLM CRD 所有权冲突）。请在集群安装时从 operator 源码树一次性安装它们：

```
kubectl apply -k config/argo
```

如果集群中已经存在 `workflows.argoproj.io`，则可跳过此步骤（例如，已安装现有的 Argo）。

- **Alauda ServiceMesh v2 (Istio)**——仅在你计划通过 Istio `VirtualService` 公开 DSPA APIServer 时才需要（`EXTERNAL_ROUTE_PROVIDER=virtualservice`）。`virtualservices.networking.istio.io` 必须处于 Established 状态。

上架 Operator Package

从 灵雀云 customer portal / Marketplace 下载 `Data Science Pipelines Operator` bundle，然后按照 Upload Packages 上架软件包。

> **INFO**
>
> operator bundle 会在 CSV 的 `relatedImages` 中记录所有运行时镜像（operator、KFP APIServer / driver / launcher / persistence-agent / scheduled-workflow、Argo workflow-controller 和 argoexec、MLMD、MariaDB，以及 pipeline runtime image），因此 `violet` 版本会将它们重新定位到平台 registry 中。这使得 operator 可以在离线集群中安装，而无需访问 `quay.io` / `docker.io`。

安装 Operator

在 **Administrator** 视图中：

1. 单击 **Marketplace** / **OperatorHub**。
2. 在控制台顶部，从 **Cluster** 下拉列表中选择目标集群。
3. 搜索并选择 **Data Science Pipelines Operator**，然后单击 **Install**。
4. 保持 **Channel** 不变（`stable`）。

5. 确认 **Version** 与你要安装的版本一致（例如 `v2.15.1`）。
6. 保持 **Installation Location** 不变——默认是 `data-science-pipelines-operator` namespace。
7. 选择 **Upgrade Strategy**（生产环境建议使用 `Manual`）。
8. 单击 **Install**。

外部端点提供程序（可选）

operator 会根据各 DSPA 的 `EXTERNAL_ROUTE_PROVIDER` 设置公开其 APIServer，该设置配置在 Subscription 的 `spec.config.env` 中：

- `none`（默认）——operator 不创建外部端点。APIServer 仅能在集群内访问（`ds-pipeline-<name>.<namespace>.svc:8888`）。当入口流量由其他方式处理时使用此选项。
- `virtualservice`——operator 为每个 DSPA 创建一个 Istio `VirtualService`，并绑定到共享的 Istio Gateway（默认 `kubeflow/kubeflow-gateway`），将 `/pipelines/<namespace>/<name>/...` 路由到该 DSPA 的 APIServer 和 MLMD 服务。需要 ServiceMesh v2（参见前提条件）。

要使用 `virtualservice` 提供程序，请在创建 Subscription 时设置该环境变量（或在之后编辑它）：

```
spec:
  config:
    env:
      - name: EXTERNAL_ROUTE_PROVIDER
        value: virtualservice
      - name: ISTIO_GATEWAY
        value: kubeflow/kubeflow-gateway
      - name: ISTIO_HOSTS
        value: "*"
```

验证

确认 **Data Science Pipelines Operator** 磁贴显示为 `Installed`，然后在集群上验证：

```
# the CSV reports Succeeded
kubectl get csv -n data-science-pipelines-operator | grep data-science-pi
pelines-operator

# the operator controller-manager pod is Running
kubectl get pods -n data-science-pipelines-operator

# the CRDs are registered
kubectl get crd | grep -E 'datasciencepipelinesapplications|pipelines.kub
eflow.org|scheduledworkflows'
```

operator 默认监视所有 **namespace**，因此你可以在任何 project namespace 中创建 `DataSciencePipelinesApplication` 资源。

INFO

`Succeeded` 的 CSV 表示 operator controller 正在运行。KFP pipeline stack 本身只有在你创建 `DataSciencePipelinesApplication` 之后才会出现——参见 Create a Data Science Pipelines Application。

Alauda 构建的 KubeRay operator

Alauda Build of KubeRay Opera

概述

关键特性

使用场景

安装 KubeRay Operator

前提条件

Alauda Build of KubeRay Operator

Alauda Build of KubeRay Operator 是一个 Kubernetes 原生 operator，提供在 Kubernetes 上运行 Ray 应用程序的完整解决方案。它基于开源 KubeRay 项目构建，使用 Kubernetes Custom Resource Definitions (CRDs) 简化了 Ray 集群、作业和服务的部署与管理。

目录

概述

关键特性

使用场景

概述

Alauda Build of KubeRay Operator 提供三个核心 CRD：

- **RayCluster**：完全管理 Ray 集群的生命周期，包括集群创建/删除、自动扩缩容和故障容错。
- **RayJob**：在集群就绪时自动创建 RayCluster 并提交作业。支持作业完成后的自动清理。
- **RayService**：管理 Ray Serve 部署，支持零停机升级以及用于生产环境 ML 模型服务的高可用性。

关键特性

- 自动扩缩容：根据工作负载需求自动调整 worker 节点数量。
- 异构计算：支持 GPU 和其他加速器资源，用于分布式训练和推理。
- 多 **Ray** 版本：在同一个 Kubernetes 集群中运行不同版本的 Ray。
- 故障容错：提供用于处理节点故障和作业重试的内置机制。
- **Kubernetes** 集成：与现有 Kubernetes 工具和工作流无缝集成。
- 生态系统支持：可与可观测性工具（Prometheus、Grafana）、队列系统（Kueue、Volcano）以及 ingress controller 配合使用。

使用场景

- 分布式机器学习：在多个节点之间扩展 ML 训练工作负载。
- 模型服务：使用 Ray Serve 以大规模方式部署和提供 ML 模型服务。
- 批量推理：使用并行推理工作负载处理大型数据集。
- 超参数调优：使用 Ray Tune 运行分布式超参数优化。
- **LLM** 推理：部署大语言模型以进行在线推理。

更多详情，请参阅 Ray on Kubernetes↗。

安装 KubeRay Operator

目录

前提条件

下载集群插件

安装 Alauda Build of KubeRay Operator

前提条件

- **ACP** 版本：**v4.0** 或更高

下载集群插件

从 灵雀云 Customer Portal 下载 `Alauda Build of KubeRay Operator` 软件包，或联系支持，然后按照 上架软件包 上传软件包。

安装 Alauda Build of KubeRay Operator

1. 转到 `管理员` -> `Marketplace` -> `集群插件` 页面，切换到目标集群，然后部署 `Alauda Build of KubeRay Operator` 集群插件。
2. 验证安装。你可以检查 pod 状态：

```
kubectl get pods -n cpaas-system | grep "kuberay-operator"
```

你应该会看到 KubeRay operator pods 正在运行。

Alauda 构建版 Spark Operator

Alauda Build 的 Spark operator

概述

主要特性

使用场景

安装 Spark Operator

前提条件

上传 Operator 软件包

安装 Operator

Alauda Build 的 Spark operator

Alauda Build 的 Spark operator 是一个 Kubernetes 原生的 operator，用于在 Kubernetes 上运行和管理 Apache Spark 应用。它基于开源的 Kubeflow Spark Operator 构建，使你能够通过 Kubernetes Custom Resource Definitions (CRDs) 声明式地提交、调度、监控和清理 Spark 工作负载——而无需自行运行 `spark-submit` 客户端或独立的 Spark 集群。

目录

概述

主要特性

使用场景

概述

该 operator 提供以下 CRD（API group `sparkoperator.k8s.io`）：

- **SparkApplication**：定义一个单独的 Spark 应用（一个 driver 及其 executors）。该 operator 以 cluster mode 提交它，跟踪其生命周期直至完成，并回收其资源。
- **ScheduledSparkApplication**：按照 cron 调度运行一个 `SparkApplication`，并提供并发与运行历史控制。
- **SparkConnect**：管理一个长期运行的 Spark Connect server，用于交互式 / 远程 Spark 会话。

主要特性

- 声明式提交：将 Spark 作业作为 Kubernetes 资源提交；operator 会替你以 cluster mode 运行 `spark-submit`。
- 生命周期管理：跟踪 driver/executor 状态，暴露 `applicationState`，遵循配置的重启策略，并清理已完成的应用。
- 调度：通过 `ScheduledSparkApplication` 实现 cron 风格的周期性作业。
- **Admission webhook**：对 Spark 资源进行变更与校验（volumes、affinity、security context 等）。
- 批调度器集成：可选通过 Volcano 或 Yunikorn 实现 gang scheduling。
- **Metrics**：为应用、executors 和提交延迟暴露 Prometheus metrics。

使用场景

- 批量数据处理：在 Kubernetes 上运行 ETL 和分析作业。
- 定时流水线：无需外部调度器即可运行周期性 Spark 作业。
- 分布式数据准备 **/ ML**：大规模特征工程与训练数据准备。
- 交互式 **Spark**：通过 Spark Connect 进行远程 Spark 会话。

此版本包含 **Apache Spark 4.0.1**。有关 Spark 概念和 Kubernetes 相关细节，请参阅 Running Spark on Kubernetes↗。

安装 Spark Operator

Alauda Build of Spark Operator 以 **OLM Operator** 的形式交付，并从平台 **OperatorHub** 安装。

目录

前提条件

上传 Operator 软件包

安装 Operator

验证

前提条件

- **ACP** 版本：**v4.0** 或更高版本（已在 v4.3 / Kubernetes 1.34 上验证）。
- 目标集群架构为 `linux/amd64` 或 `linux/arm64`（该 operator 提供多架构镜像）。
- 目标集群上可用 Operator Lifecycle Manager (OLM)（由 ACP 提供）。

上传 Operator 软件包

从 灵雀云 Customer Portal / Marketplace 下载 `Alauda Build of Spark Operator` bundle，然后按照 上架软件包 上传软件包。

INFO

该 operator bundle 会在 CSV 的 `relatedImages` 中记录其运行时镜像（operator 镜像和 Apache Spark 运行时镜像），因此 `violet` 版本会将这些镜像迁移到平台镜像仓库中。这样一来，operator 即可在 **air-gapped** 集群上安装，而无需访问 `docker.io`。

安装 Operator

在 管理员 视图中：

1. 单击 **Marketplace / OperatorHub**。
2. 在控制台顶部，从 集群 下拉列表中选择目标集群。
3. 搜索并选择 **Alauda Build of Spark Operator**，然后单击 **Install**。
4. 保持 **Channel** 不变（`stable`）。
5. 检查 **Version** 是否与要安装的版本匹配（例如 `v2.5.1`）。
6. 保持 **Installation Location** 不变——默认是 `spark-operator` 命名空间。
7. 选择 **Upgrade Strategy**（生产环境建议使用 `Manual`）。
8. 单击 **Install**。

验证

确认 **Alauda Build of Spark Operator** 磁贴显示为 `Installed`，然后在集群上验证：

```
# the CSV reports Succeeded
kubectl get csv -n spark-operator | grep spark-operator

# the operator (controller + webhook) pods are Running
kubectl get pods -n spark-operator

# the CRDs are registered
kubectl get crd | grep sparkoperator.k8s.io
```

该 operator 默认监视所有命名空间，因此你可以在任意命名空间中创建 `SparkApplication` 资源。

INFO

`Succeeded` CSV 表示 operator 控制器正在运行。只有在你创建 `SparkApplication`（或 `ScheduledSparkApplication`）之后，Spark 工作负载才会出现——请参见 运行 Spark Application。

MLflow

MLflow

它会部署什么

关键概念

多租户模型

何时使用它

安装 MLflow

前提条件

上传 Operator 软件包

启用 MLflow

创建 MLflow tracking server

验证

高可用性和存储

Bearer-token 客户端

从 v3.10.x 升级到 v3.13.0

故障排查

MLflow

MLflow operator 在 Alauda AI 上部署并管理一个多租户的 **MLflow Tracking Server**。它基于开源 MLflow↗ 构建，为数据科学和 ML 团队提供一个共享的实验跟踪和 model registry 服务，并与平台的单点登录和 Kubernetes RBAC 集成——因此每个团队只能查看和管理自己的 runs、experiments 和 registered models。

MLflow 以 基于 **OLM Helm** 的 **operator** 形式交付。从平台 **OperatorHub** 安装该 operator 并创建一个 `MLflow` 自定义资源后，就会启动 tracking server，通过平台 ingress 对外暴露，并在 **Alauda AI** → **Tools** 菜单中添加一个 **MLFlow** 条目。

目录

它会部署什么
关键概念
多租户模型
何时使用它

它会部署什么

当你创建一个 `MLflow` 自定义资源时，operator 会在配置的 namespace 中协调一套 MLflow 栈（默认是 `kubeflow`）：

- **MLflow Tracking Server** — 实验跟踪 API 和 Web UI：记录参数、指标、artifacts 和 AI-agent traces；浏览 runs 和 traces；以及管理 model registry（`mlflow` Python SDK 的目标）。
- **OAuth proxy** (`oauth2-proxy`) — 在请求到达 MLflow 之前，先通过平台 OIDC provider 对每个请求进行认证。客户端始终通过 proxy 访问；它们不会直接连接到 MLflow 容器端口。
- **Kubernetes auth plugin** — 在 tracking server 内部，从转发的令牌中读取调用者身份，将每个 run 记录到该用户名下，并基于 Kubernetes RBAC 对请求进行授权。
- **Tools** 菜单条目和 **ingress** 路由 — 一个 `ConfigMap`，用于将 **MLFlow** 添加到 Alauda AI **Tools** 菜单，以及平台 load balancer 上的一条 ingress 路由。

MLflow 元数据存储在你提供的 外部 **PostgreSQL** 数据库中。run artifacts 会写入 tracking server 配置的 artifact root（请参见 Installation → High availability and storage）。

关键概念

概念	描述
`MLflow` 自定义资源	声明一个 MLflow tracking server 的 CR（`mlflow.alauda.io/v1alpha1`）。一个 CR = 集群中的一个 tracking server。
Workspace	一个暴露给 MLflow 作为租户边界的 **Kubernetes namespace**。Experiments、runs、datasets 和 registered models 都属于某个 workspace。
多租户	只有与已配置 label selector 匹配的 namespace（默认是 `mlflow-enabled=true`）才会作为 workspace 可见；访问权限会基于 Kubernetes RBAC 按 namespace 进行授权。
OAuth proxy	`oauth2-proxy` 位于 tracking server 前端：它先通过平台 SSO 对调用者进行认证，然后将身份令牌转发给 MLflow。

多租户模型

在 Alauda AI 上，MLflow 通过将 **workspaces** 映射到 **Kubernetes namespaces** 来实现多租户：

- 当一个 namespace 带有已配置的 label（默认是 `mlflow-enabled=true`）时，它就会成为 MLflow **workspace**。
- 每一次 MLflow API 调用都会先由 OAuth proxy 进行认证，然后使用 `mlflow.kubeflow.org` API group，基于目标 namespace 中调用者的 Kubernetes RBAC 进行授权。
- run 会记录在发起调用的用户身份下。

因此，workspace 的成员关系和权限由普通的 Kubernetes `Role` / `RoleBinding` 对象管理——请参见 MLflow Workspaces and Access Control。

何时使用它

当团队需要一个共享的、受 SSO 保护的实验跟踪、AI-agent tracing 和 model registry 服务，并且每个团队的 runs 和 models 都通过 namespace 隔离并由 Kubernetes RBAC 管理时，请使用 MLflow Operator——而不是使用一个未认证的、单租户的 MLflow server。

安装 MLflow

在 Alauda AI 2.8.x 中，**MLflow** 由 Alauda AI 安装和管理。集群作用域的 `default` `AmlCluster` 中的 `mlflow` 组件默认值为 `Unmanaged`；将其设置为 `Managed` 后，Alauda AI 会自动安装 **MLflow Operator**，你无需手动从 OperatorHub 安装该 Operator。启用该组件只会安装 Operator；你仍然需要创建一个 `MLflow` 自定义资源来部署 tracking-server 堆栈，如 创建 MLflow tracking server 中所述。

目录

前提条件

上传 Operator 软件包

启用 MLflow

创建 MLflow tracking server

方法 A — 使用 `kubectl`

方法 B — 使用 Alauda Console UI

验证

高可用性和存储

Bearer-token 客户端

从 v3.10.x 升级到 v3.13.0

故障排查

前提条件

- 目标集群上已安装 **Alauda AI**。该 operator 及其镜像同时提供 `linux/amd64` 和 `linux/arm64` 版本。
- 目标集群上可用 Operator Lifecycle Manager (OLM)（由平台提供）。
- **PostgreSQL 12** 或更高版本数据库 可从集群访问，并包含一个名为 `mlflow` 的数据库用于 MLflow 元数据。请准备好主机、端口、用户名和密码，并在安装前创建数据库：

  ```
  CREATE DATABASE mlflow;
  ```

  生产环境请使用高可用 PostgreSQL 服务（参见 高可用性和存储）。
- 用于生产 **artifact** 存储的现有 **S3** 兼容 **bucket**，可位于 SeaweedFS、MinIO、Ceph RGW 或 AWS S3 中。operator 可以代理 artifact 访问，因此客户端无需 object-storage 凭据。pod 内文件系统默认值仅适用于 smoke test。
- 平台 **OAuth / OIDC provider**（由 Alauda AI 提供）—— tracking server 前面的 OAuth proxy 会使用它进行单点登录。
- **对于多租户场景，默认 workspace namespace 必须已存在，并且在 tracking server 首次启动之前就已带有标签 `mlflow-enabled=true`。**否则 server 会陷入 crash-loop，并报错 `Workspace '<name>' not found in the Kubernetes cluster`。

  ```
  kubectl create namespace finetune
  kubectl label namespace finetune mlflow-enabled=true
  ```

上传 **Operator** 软件包

从 灵雀云 Customer Portal / Marketplace 下载 `MLflow Operator` bundle，然后按照 上传软件包 上传该软件包。

> **INFO**

operator bundle 会将所有运行时镜像（operator、MLflow tracking server 以及 `oauth2-proxy`）记录在 CSV 的 `relatedImages` 中，因此 `violet` 版本会把它们迁移到平台 registry 中。这使得该 operator 可以在 **air-gapped** 集群上安装，而无需访问 `build-harbor.alauda.cn` / `docker.io`。

启用 MLflow

在 **Administrator** 视图中：

1. 打开 **Marketplace** > **OperatorHub**。
2. 选择目标集群并打开 **Alauda AI**。
3. 打开 **All Instances** 选项卡，并编辑 `default` `AmlCluster` 实例。
4. 将 `spec.components.mlflow.managementState` 设置为 `Managed`。
5. 保存 `AmlCluster` 实例，并等待 MLflow Operator 就绪。

你也可以在 `AmlCluster` YAML 中应用该组件设置：

```yaml
spec:
  components:
    mlflow:
      managementState: Managed
```

继续之前，请确认 operator 已就绪：

```bash
kubectl get amlcluster default

# the CSV reports Succeeded
kubectl get csv -n mlflow-operator | grep mlflow-operator

# the operator controller-manager pod is Running
kubectl get pods -n mlflow-operator
```

`AmlCluster` 应报告 `Phase=Ready`。启用该组件只会安装 **MLflow Operator**；只有在你按照下文创建 `MLflow` 自定义资源后，tracking server 才会被部署。

创建 MLflow tracking server

在创建 `MLflow` 自定义资源（`mlflow.alauda.io/v1alpha1`）之前，operator 不会执行任何操作。你可以通过以下两种方式创建它：

- `kubectl` — 应用一个 `MLflow` manifest。
- **Alauda Console UI** — 依次进入 **Administrator** → **Marketplace** → **OperatorHub** → **MLflow Operator** → **All instances** → **Create**。

两种方式创建的是同一个资源；请选择最适合你工作流的方式。

方法 A — 使用 `kubectl`

设置数据库连接和多租户字段，然后应用：

```yaml
apiVersion: mlflow.alauda.io/v1alpha1
kind: MLflow
metadata:
  name: mlflow
spec:
  # Namespace the tracking server is deployed into (chart-mandated: kubeflow).
  namespace: kubeflow

  # External PostgreSQL for MLflow metadata — the `mlflow` database must already exist.
  pgHost: postgresql.database.svc
  pgPort: 5432
  pgUsername: mlflow
  pgPassword: <db-password>

  # Multi-tenancy: map Kubernetes namespaces to MLflow workspaces.
  multitenancy:
    enabled: true
    defaultWorkspace: finetune                # must exist and be labelled before first boot
    workspaceLabelSelector: mlflow-enabled=true

  # Run-artifact location (see High availability and storage below).
  artifacts:
    defaultArtifactRoot: /mlflow/artifacts
    # Recommended for production. The bucket and Secret must already exist.
    s3:
      enabled: true
      endpointUrl: https://object-storage.example.com
      bucket: mlflow
      prefix: production
      existingSecret: mlflow-s3-credentials
      accessKeyIdKey: AWS_ACCESS_KEY_ID
      secretAccessKeyKey: AWS_SECRET_ACCESS_KEY
      proxied: true
```

将其保存为 `mlflow.yaml` 并应用：

```bash
kubectl apply -f mlflow.yaml
```

INFO

在当前版本中，operator 会从 `kube-public/global-info` `ConfigMap` 中解析集群特定值——ingress/ALB 名称、集群名称和镜像 registry，因此通常无需设置 `spec.global` 和 `spec.platformAddr`。`auth` 设置（`oauth2-proxy` 配置）也有合理的默认值。Console 中的 **Create** 表单会预填一个完整示例（operator 的 `alm-examples`），你可以直接编辑；完整字段集记录在 operator 的 `config/samples/mlflow_v1alpha1_mlflow.yaml` 中。

方法 B — 使用 Alauda Console UI

1. 在 **Administrator** 视图中，进入 **Marketplace** → **OperatorHub**。
2. 在顶部的 **Cluster** 下拉列表中选择目标集群。
3. 打开已安装的 **MLflow Operator**。
4. 切换到 **All instances** 选项卡并点击 **Create**（如果提示，请选择 **MLflow**）。
5. 填写表单，或者切换到 **YAML** 视图并粘贴方法 A 中的 manifest。为生产环境设置 PostgreSQL 连接、多租户 workspace 字段以及 S3 artifact 存储字段。
6. 点击 **Create**。

验证

验证 tracking server 是否已在配置的命名空间（`kubeflow`）中完成 reconcile：

```
# the operand exists
kubectl get mlflow

# the tracking-server Deployment is Available (two containers: the server
+ oauth2-proxy)
kubectl -n kubeflow get deploy mlflow-tracking-server

# its pod is Running
kubectl -n kubeflow get pods -l app.kubernetes.io/name=mlflow-tracking-se
rver
```

然后打开 **Alauda AI** → **Tools** → **MLFlow**。run owner 和可见的 workspaces 会反映你登录所使用的身份。

高可用性和存储

- 数据库。 MLflow 会将所有 experiment/run/registry 元数据存储在你配置的外部 PostgreSQL 中。生产环境请使用高可用 PostgreSQL 服务；tracking server 本身在元数据方面是无状态的。
- **Artifacts**。 默认的 `artifacts.defaultArtifactRoot: /mlflow/artifacts` 是 tracking-server pod 本地的 `emptyDir`。它不具备持久性，外部客户端也无法可靠地使用 server 本地路径。从 v3.13.0 开始，请在用户记录 artifact 或注册模型之前，为生产环境启用 `artifacts.s3`。在默认的 `proxied: true` 下，tracking server 会存储 S3 凭据，客户端则通过经过身份验证的 MLflow endpoint 上传。
- **Replicas**。 默认部署为单副本。除非你的 release notes 另有说明，否则这不是多副本高可用部署。

在应用 `MLflow` 资源之前，请先在 tracking-server namespace 中创建所引用的 credentials Secret：

```
kubectl -n kubeflow create secret generic mlflow-s3-credentials \
  --from-literal=AWS_ACCESS_KEY_ID='<access-key>' \
  --from-literal=AWS_SECRET_ACCESS_KEY='<secret-key>'
```

bucket 必须已存在。对于 AWS S3，请将 `endpointUrl` 留空。对于 S3 兼容服务，请将其设置为服务 endpoint。必要时，你还可以在 `artifacts.s3` 下设置 `region`、`ignoreTls` 和 `prefix`。

Bearer-token 客户端

从 v3.13.0 开始，operator 默认启用 `spec.auth.oauth.skipJwtBearerTokens: true`，允许 **MLflow Python SDK** 通过 `MLFLOW_TRACKING_TOKEN` 使用 Dex **id token** 进行身份验证。如果现有安装覆盖了 OAuth 设置，请确认该字段仍保持启用。请参见 SDK 指南中的 验证 token 方法。浏览器 session-cookie 方法也仍然可用。

从 **v3.10.x** 升级到 **v3.13.0**

`stable` OLM channel 支持此升级。在批准之前，请先备份 PostgreSQL 后端数据库和 S3 artifact bucket。backend-store schema 迁移会在 tracking server 启动时自动运行。

升级后，请保留 `spec.auth.kubernetes.authorizationMode: user_identity_token` 和 `spec.auth.oauth.skipJwtBearerTokens: true`。这些默认值会保留本文档所述的 Dex bearer-token 流程。

故障排查

- 如果 **MLFlow** Tools 菜单项缺失，请确认 tracking-server namespace 中存在 `aml-mlflow-menu-config` `ConfigMap`，并且它带有标签 `aml.cpaas.io/centralMenuItem: "true"`。
- 如果 server 无法启动，并且 pod 日志中出现 `Workspace '<name>' not found`，则 `defaultWorkspace` namespace 不存在，或者没有标记为 `mlflow-enabled=true`。请创建并为其添加标签，然后重启 pod。
- 如果 server 因其他原因无法启动，请验证 PostgreSQL 连通性、`mlflow` 数据库是否存在，以及 `spec.pg*` 中的凭据是否正确。
- 如果 artifact 上传或模型注册失败，请验证 `artifacts.s3.enabled` 是否为 `true`、bucket 是否存在，以及所引用的 credentials Secret 和 key 名称是否正确。
- 如果某个 workspace 不可见，请验证其 namespace 是否与已配置的 `workspaceLabelSelector` 匹配。
- 如果请求被拒绝（`403 PERMISSION_DENIED`），请检查 workspace namespace 中用户的 Kubernetes `RoleBinding`——参见 MLflow Workspaces and Access Control。

Label Studio

Label Studio

Label Studio

Core Concepts

Documentation

安装 Label Studio

下载

部署

配置

访问地址

用户管理

快速入门

S3 集成

其他资源

主要功能

多用户标注

多类型数据支持

灵活的标注配置

数据导入/导出

Machine Learni

Label Studio

目录

Label Studio
Core Concepts
Project
Labeling Interface
Data Manager
Annotations
Machine Learning Integration
Core Concept Relationships
Documentation
Official Documentation

Label Studio

Label Studio 是一款开源的多类型数据标注与注释工具，提供标准化输出格式。它支持多种数据类型的标注，包括图像、音频、文本、时间序列和视频。

它包含以下主要组件：

- **Backend Service**：基于 Django 的 Python Web 服务，提供 REST API、Python SDK 和机器学习集成

- **Frontend Interface**：基于 React 的 Web UI，提供完整的注释界面，包括项目管理、数据管理、注释工具和结果导出
- **Database**：支持 PostgreSQL 13+ 数据库，用于存储项目数据和注释结果
- **Cache System**：用于缓存和任务队列管理的 Redis（可选）

Label Studio 帮助团队构建并维护高质量的数据标注工作流：从简单的图像分类到复杂的多模态数据注释任务。

Core Concepts

Project

Project 是 Label Studio 中数据标注的基本组织单元，包括：

- **Project Settings**：注释配置、数据导入设置、用户权限等
- **Data Management**：数据导入、存储和版本控制
- **Annotation Interface**：可配置的注释工具和界面
- **Annotation Results**：注释数据的存储和管理

每个 Project 都具有独立的配置和数据空间，支持多用户协作标注。

Labeling Interface

Labeling Interface 是用户执行数据注释的核心工具，支持：

- **Multiple Annotation Types**：图像分类、目标检测、文本分类、命名实体识别等
- **Configurable Interface**：通过配置语言自定义注释界面
- **Template Support**：提供多种预定义注释模板
- **Shortcut Support**：提供快捷键功能以提升标注效率

Labeling Interface 使用一种专门设计的配置语言，可灵活适配各种注释需求。

Data Manager

Data Manager 是项目数据的核心管理工具，提供：

- **Data Import**：支持从文件、云存储（AWS S3、Google Cloud Storage）导入数据
- **Data Formats**：支持 JSON、CSV、TSV 等格式
- **Data Preview**：查看和预览待注释数据
- **Data Filtering**：按状态、标注者、标签及其他条件筛选数据

Data Manager 支持批量操作和高级搜索功能。

Annotations

Annotations 是用户为数据添加的标签和评论，包括：

- **Annotation Data**：用户添加的标签、边界框、分割区域
- **Annotation Metadata**：注释时间、标注者、置信度及其他信息
- **Annotation Status**：草稿、已完成、已跳过等状态
- **Annotation Quality**：注释质量评分和验证

注释数据以标准化 JSON 格式存储，便于后续处理和分析。

Machine Learning Integration

Label Studio 提供强大的机器学习集成能力：

- **Pre-annotation**：使用机器学习模型进行预标注，以提升效率
- **Online Learning**：在标注过程中进行实时训练和模型更新
- **Active Learning**：智能选择需要标注的复杂样本
- **Model Comparison**：比较不同模型的预测结果

支持多种机器学习框架和模型格式。

Core Concept Relationships

- **Projects** 是组织标注任务和数据的基本容器
- **Labeling Interfaces** 定义用户如何与数据交互并执行标注

- **Data Managers** 负责项目内的数据导入、存储和组织
- **Annotations** 存储实际的标注结果和元数据
- **Machine Learning Integration** 连接外部模型，用于预标注和主动学习

Documentation

Label Studio 提供全面的官方文档和 API 参考，帮助用户深入理解和使用平台功能：

Official Documentation

- **Main Documentation**: https://labelstud.io/guide/ ↗
  - 详细介绍 Label Studio 的核心概念和工作流
  - 包括安装指南、快速开始和最佳实践
  - 提供常见使用场景、示例代码、教程和 API 参考

安装 Label Studio

本文档提供了将 Label Studio 部署到 Kubernetes 集群以及常用配置参数的详细说明。

目录

下载

部署

准备存储

准备数据库

准备 Redis（可选）

创建应用

配置

1. 配置存储

1.1 配置 Storage Class 和存储大小

2. 配置数据库

2.1 配置 PostgreSQL

2.2 配置 Redis

3. 配置访问方式

3.1 修改 Service 类型

3.2 启用 Ingress

3.3 配置 OAuth2 Proxy（可选）

4. 配置用户管理

4.1 禁用用户注册

访问地址

1. 通过 Service 访问
2. 通过 Ingress 访问

用户管理

下载

下载 Label Studio 安装包，然后按照 上架软件包 上传该软件包。

部署

准备存储

Label Studio 将数据存储在数据库中，并且需要持久化存储。集群需要预先安装 CSI，或者预先准备好 `PersistentVolume`。

准备数据库

Label Studio 支持以下数据库：

- **PostgreSQL**：版本 13 或更高

Data Services 提供的 `PostgreSQL operator` 可用于创建 `PostgreSQL cluster`。

可在 `Data Services` 中 `PostgreSQL` 实例详情里查看访问地址和密码。

准备 **Redis**（可选）

Redis 不是必需的，但建议在生产环境中使用。

可使用 `Data Services` 创建 `Redis` 实例。

注意：Label Studio 仅支持以 `standalone` 模式访问 Redis。

- 创建 `standalone` 模式的 `Redis`：

  1. 创建 `Redis` 实例时，在 `Architecture` 中选择 `Redis Sentinel`。
  2. 设置完所有参数后，切换到 `YAML` 模式，将 `spec.arch` 改为 `standalone`，然后点击 `Create` 按钮。
  3. 创建完成后，切换到 `Alauda Container Platform` 视图，找到名为 `rfr-<Redis instance name>-read-write` 的 `Service`，这就是该 Redis 实例的访问地址。

创建应用

1. 进入 `Alauda Container Platform` 视图，选择要部署 Label Studio 的 namespace。
2. 在左侧导航中，选择 `Applications` / `Applications`，然后点击右侧页面上的 `Create` 按钮。
3. 在弹窗中选择 `Create from Catalog`，页面将跳转到 `Catalog` 视图。
4. 找到 `3rdparty/chart-label-studio`，点击 `Create` 创建该应用。
5. 在 `Catalog` / `Create label-studio` 表单中，填写 `Name`（建议为 `label-studio`）以及 `Values` 中的 `Custom` 配置，然后点击 `Create` 按钮完成创建。`Custom` 内容将在下文说明。也可以在创建后通过 `Update` 应用方式进行修改。

配置

用户可以修改应用的 `Custom Values` 来调整配置。主要需要关注的配置如下：

1. 配置存储

1.1 配置 Storage Class 和存储大小

可以通过添加以下配置来指定 storage class：

```yaml
label-studio:
  persistence:
    storageClass: storage-class-name
    size: 20Gi                                  # Replace with the actual re
quired space size
```

2. 配置数据库

2.1 配置 PostgreSQL

可以通过设置以下字段来配置 PostgreSQL 访问信息：

```yaml
global:
  pgConfig:
    host: localhost                             # PostgreSQL access address
    port: 5432                                  # PostgreSQL access port, de
fault: 5432
    dbName: labelstudio                         # Database name, note: datab
ase will be created automatically
    userName: postgres                          # Database username
    password:
      secretName: postgre-secret                # Secret name storing databa
se access password
      secretKey: password                       # Secret key storing databas
e access password
```

2.2 配置 Redis

可以通过设置以下字段来配置 Redis 访问信息：

```yaml
global:
  redisConfig:
    host: "redis://your-redis-host:6379/1"    # Redis connection address, format: redis://[:password]@host:port/db
    password:                                 # Optional, password can be included in host or provided separately via Secret
      secretName: "redis-secret"              # Secret name storing Redis access password
      secretKey: "password"                   # Secret key storing Redis password
    ssl:                                      # Optional
      redisSslCertReqs: "optional"            # SSL certificate requirements: "" means not required, "optional", "required"
      redisSslSecretName: "redis-ssl-secret"  # SSL certificate Secret name
      redisSslCaCertsSecretKey: "ca.crt"      # CA certificate Secret key
      redisSslCertFileSecretKey: "tls.crt"    # Client certificate Secret key
      redisSslKeyFileSecretKey: "tls.key"     # Client private key Secret key
```

3. 配置访问方式

默认使用 `LoadBalancer` 提供访问地址。

3.1 修改 Service 类型

可以通过设置以下字段来修改 `Service` 类型：

```yaml
label-studio:
  app:
    service:
      type: LoadBalancer                      # Can be changed to NodePort or ClusterIP
```

3.2 启用 Ingress

可以通过设置以下字段来配置 Ingress。启用 Ingress 后，Service 类型通常会改为 ClusterIP：

```yaml
label-studio:
  app:
    ingress:
      enabled: true                          # Enable Ingress functionali
ty
      host: localhost                        # Access domain (must be DNS
name, not IP address)
      tls:
        - secretName: certificate-secret     # Secret name storing TLS ce
rtificate
global:
  extraEnvironmentVars:
    LABEL_STUDIO_HOST: https://x.x.x.x       # Web access URL for fronten
d resource loading
```

3.3 配置 **OAuth2 Proxy**（可选）

```yaml
oauth2_proxy:
  enabled: true
  oidcIssuer: "https://x.x.x.com/dex"          # OIDC Issuer address
  oidcClientID: "your-client-id"               # OIDC client ID
  oidcClientSecret: "your-client-secret"       # OIDC client secret (reco
mmended to use Secret)

  ingress:                                     # Optional, use ingress to
access OAuth2 proxy
    enabled: true                              # Default is false
    host: localhost                            # Access domain, must be s
et, cannot use IP address, only domain name
    tls:
      - secretName: certificate-secret         # Secret name storing TLS
certificate
```

如果要将 `Alauda Container Platform` 配置为 OIDC Provider，请按以下方式配置：

- `oauth2_proxy.oidcIssuer` 为平台访问地址加上 `/dex`
- `oauth2_proxy.oidcClientID` 固定为 `label-studio`
- `oauth2_proxy.oidcClientSecret` 固定为 `ZXhhbXBsZS1hcHAtc2VjcmV0`

还需要在 global 集群中创建一个 OAuth2Client 资源，以配置 Label Studio 的客户端信息：

```
apiVersion: dex.coreos.com/v1
kind: OAuth2Client
metadata:
  name: nrqwezlmfvzxi5lenfx4x4u44scceizf
  namespace: cpaas-system
id: label-studio                                # Must match oauth2_prox
y.oidcClientID in values
name: Label Studio
secret: ZXhhbXBsZS1hcHAtc2VjcmV0               # Must match oauth2_prox
y.oidcClientSecret in values
redirectURIs:
- http://xxx.xxx.xxxx.xxx:xxxxx/*               # OAuth2-Proxy access add
ress, see below for how to obtain
                                                # If multiple Label Studi
o instances are deployed, add all access addresses here
```

注意：OAuth2 proxy 的访问地址可从 `<Application Name>-oauth2-proxy` 这个 Service 获取，请根据 Service 类型使用相应的访问方式。

4. 配置用户管理

4.1 禁用用户注册

可以通过设置以下字段来禁用用户注册：

```
global:
  extraEnvironmentVars:
    LABEL_STUDIO_DISABLE_SIGNUP_WITHOUT_LINK: true
```

访问地址

1. 通过 **Service** 访问

`Label Studio` 通过 Service 提供外部访问。请查看其 Service 以获取访问地址。

- 如果未启用 OAuth2 proxy，则 Service 名称为：`<Application Name>-ls-app`
- 如果启用 OAuth2 proxy，则 Service 名称为：`<Application Name>-oauth2-proxy`

如果 `Service` 类型为 `LoadBalancer`，且环境中的负载均衡器控制器已经分配了访问地址，请通过该地址访问。

对于 `LoadBalancer` 或 `NodePort` 服务类型，也可以通过节点 IP 和其 `NodePort` 访问。

2. 通过 Ingress 访问

如果启用了 Ingress，请通过配置好的 LABEL_STUDIO_HOST 访问。

用户管理

Label Studio 默认没有用户名和密码。用户可以在登录页面填写邮箱和密码完成新用户注册。

注意：

- 默认配置允许任何人注册新用户
- 所有用户具有相同的功能权限，并且可以访问所有项目
- 如需限制用户注册，请配置环境变量 `LABEL_STUDIO_DISABLE_SIGNUP_WITHOUT_LINK=true`，请参见：4.1 禁用用户注册

快速入门

目录

S3 集成
前提条件
在 Label Studio 中使用 S3
存储结构建议
其他资源

S3 集成

Label Studio 支持与 S3 兼容存储集成，用于导入数据和导出标注。常见选项包括 Amazon S3、Ceph RGW 以及其他 S3 兼容对象存储服务。

前提条件

- 具有相应权限的 S3 兼容存储桶
- 访问凭据（Access Key ID 和 Secret Access Key）

存储桶和凭据必须来自受支持的 S3 兼容服务；平台不要求特定的对象存储产品。

在 Label Studio 中使用 S3

1. 访问存储设置

- 打开 Label Studio 项目
- 转到 **Settings** > **Cloud Storage**

2. 添加源存储

- 单击 **Add Source Storage**
- 选择 **AWS S3** 作为存储类型
- 填写所需信息：
  - **Storage Title**：存储连接的名称
  - **Bucket Name**：S3 存储桶名称
  - **Region Name**：存储地域（例如，AWS S3 使用 us-east-1；如果提供商不要求，也可以留空）
  - **S3 Endpoint**：可选的自定义 S3 endpoint（AWS S3 默认值留空；使用非默认或 S3 兼容 endpoint 时进行设置）
  - **Access Key ID**：访问密钥
  - **Secret Access Key**：密钥
  - **Session Token**：临时凭据的可选 session token
  - **Bucket Prefix**：存储桶中的可选路径前缀（例如，`data/`、`input/`）
  - **File Filter Regex**：用于筛选文件的可选 regex（例如，`.*csv` 或 `.*(jpe?g|png|tiff)`）
- 配置可选设置：
  - **Treat every bucket object as a source file**：针对媒体文件勾选；针对 JSON 任务文件取消勾选
  - **Recursive scan**：启用后递归扫描子目录
  - **Use pre-signed URLs**：启用后可通过浏览器直接访问 S3（推荐）
  - **Expiration minutes**：在启用 **Use pre-signed URLs** 时的 URL 过期时间（默认：15 分钟）
- 单击 **Check Connection** 测试连通性
- 单击 **Add Storage** 创建存储连接

3. 添加目标存储（可选）

- 单击 **Add Target Storage** 将标注导出到 S3
- 填写与源存储类似的 S3 参数
- 目标存储的其他参数：
  - **SSE KMS Key ID**：用于服务端加密的可选 KMS key
- 配置可选设置：
  - **Can delete objects from storage**：启用后允许从存储中删除标注
- 单击 **Check Connection** 测试连通性
- 单击 **Add Storage** 创建存储连接

4. 将数据上传到 **S3**

- 将数据文件上传到已配置的 S3 存储桶和前缀路径
- 确保可使用已配置的访问凭据访问这些数据文件
- 使用 AWS CLI 或其他 S3 兼容客户端进行批量上传

5. 导入数据

- 在 `Source Cloud Storage` 下单击 **Sync Storage**，从 S3 导入数据
- 每当 S3 存储桶中新增数据时，都可使用同步操作

6. 执行标注

- 在 Label Studio 界面中访问已导入的数据
- 使用已配置的标注界面完成标注

7. 导出标注

- 单击 **Export** 按钮，以多种格式（JSON、CSV 等）下载标注结果
- 或在 `Target Cloud Storage` 中单击 **Sync Storage**，将标注推送到 S3
- 注意：目标存储仅以 JSON 格式导出标注。使用 Label Studio SDK 可将 JSON 标注转换为其他格式（CSV、COCO、Pascal VOC、YOLO 等）。详情请参见 SDK converter ↗ 。

8. 将数据和标注应用于模型训练/验证

- 使用 AWS CLI、boto3 或其他 S3 客户端从 S3 下载训练数据和标注。实现细节请参见 S3 examples↗。
- 如有需要，使用 Label Studio SDK 转换标注格式。
- 将数据集成到机器学习流水线中。
- 使用标注进行模型训练或验证。

存储结构建议

- 为不同项目使用不同的存储桶或不同的路径前缀，以避免数据冲突。
- 目标存储和源存储可以使用同一个 S3 存储桶，但采用不同的路径前缀（例如，源使用 `input/`，目标使用 `output/`），也可以使用不同的存储桶，以获得更好的数据隔离和访问控制。

其他资源

有关 Label Studio 快速入门指南，请参阅官方文档：Getting Started With Label Studio: A Step-By-Step Guide↗

主要功能

目录

多用户标注

多类型数据支持

灵活的标注配置

数据导入/导出

Machine Learning 集成

多用户标注

- 用户管理：支持用户注册、登录和基本权限管理
- 协作标注：多个用户可同时对同一项目进行标注
- 标注分配：灵活的任务分配和进度跟踪
- 质量控制：标注质量评估和一致性检查

多类型数据支持

- 图像数据：图像分类、目标检测、语义分割
- 文本数据：文本分类、命名实体识别、情感分析

- 音频数据：音频分类、语音识别、音频转写
- 视频数据：视频分类、目标跟踪、动作识别
- 时序数据：时序分类、事件识别
- 多模态数据：支持图像+文本、视频+音频等组合标注

灵活的标注配置

- 配置语言：使用 XML 配置语言定义标注界面
- 模板库：提供多种预定义标注模板
- 标注工具：支持多种内置标注工具
- 界面定制：支持基础界面配置和布局调整

数据导入/导出

- 多种格式：支持 JSON、CSV 等常见格式
- 数据导入：支持从本地文件和 URL 导入数据
- 数据导出：支持以多种格式导出标注结果
- 批量操作：支持数据的批量导入和导出

Machine Learning 集成

- **ML** 后端：支持基础 Machine Learning 后端集成
- 预标注：支持使用模型预测结果进行预标注
- **API** 集成：提供 REST API 和 Python SDK 用于集成

Alauda 构建版 Feast

Alauda Build of Feast

常见需求
托管资源
核心概念
典型工作流
Operator 职责
延伸阅读

安装 Feast

上传 Operator 软件包
启用 Feast
验证安装
后续步骤

快速入门

先决条件
FeatureStore C
常见服务配置
常见持久化模式
关于 Secret 的
特征仓库初始化
部署 FeatureSt
服务与访问
使用 Feast SDK
配置授权
延伸阅读

Alauda Build of Feast

Feast 是一个用于机器学习的开源 feature store。它帮助团队在模型训练、批量评分和在线推理中使用相同的 feature 定义，从而减少 training-serving skew 和重复的 feature engineering 工作。

在 Alauda Build of Feast 中，Feast 通常通过 Feast Operator 部署到 Kubernetes 上。每个部署都由一个 `FeatureStore` 自定义资源（CR）管理。单个 `FeatureStore` CR 可以创建主要的 Feast 服务，例如 online feature server、offline feature server、registry server 和 UI。

目录

- 常见需求
- 托管资源
- 核心概念
  - Project
  - Entity
  - Data Source
  - Feature View
  - Feature Service
  - Registry
  - Materialization
  - Push Source
  - Permission
- 典型工作流

Operator 职责

延伸阅读

常见需求

当平台需要一种更一致的方式来执行以下操作时，通常会使用 Feast：

- 在共享目录中管理 feature 定义
- 在训练、批量评分和在线推理中使用相同的 feature 定义
- 在一个位置控制 freshness、materialization 和访问策略

Feast 通过共享 registry、online 和 offline store，以及用于注册、materialization、服务和访问控制的标准工作流，提供这些能力。

托管资源

在由 Feast Operator 管理的 Kubernetes 部署中，`FeatureStore` 通常会创建或管理以下资源：

- **Offline store**：存储历史 feature 数据，并支持训练数据集生成或回填。
- **Online store**：存储最新的 feature 值，以便在推理期间进行低延迟在线读取。
- **Registry**：存储 entities、feature views、feature services 和权限等元数据。
- **UI**：提供用于浏览 feature 元数据和服务状态的 Web 界面。
- 客户端配置：Operator 还会创建一个包含客户端 `feature_store.yaml` 的 ConfigMap，以便客户端应用和作业可以连接到已部署的服务。

核心概念

以下概念是在开始使用 Feast 时最重要的内容：

Project

Feast project 是一个隔离的逻辑工作区。feature 定义、权限和 registry 对象都按 project 组织。在 Alauda Build of Feast 中，project 名称通过 `FeatureStore` CR 中的 `spec.feastProject` 设置。

Entity

Entity 是用于检索 feature 的业务键，例如 `user_id`、`driver_id` 或 `merchant_id`。

Data Source

Data source 描述原始 feature 数据的来源。示例包括文件、data warehouse 或 push source。

Feature View

feature view 定义以下内容：

- feature 属于哪个 entity
- 源数据来自哪里
- 哪些字段作为 feature 暴露
- 是否应在线提供这些 feature

在实践中，feature view 通常是用户注册和 materialize 的主要对象。

Feature Service

feature service 将一个或多个 feature 组合成一个可复用集合，供某个模型或应用版本使用。这有助于在训练和服务之间保持 feature 选择的一致性。

Registry

registry 存储的是 Feast 元数据，而不是 feature 值。它是已注册对象的权威来源，例如 entities、feature views、feature services 和权限。

Materialization

materialization 会将 feature 值从 offline source 复制到 online store，以便在推理时以低延迟获取。

Push Source

push source 允许应用或流式作业将最新的 feature 值直接写入 Feast，写入目标可以是 online store、offline store，或者两者同时。

Permission

权限用于定义谁可以读取或修改 Feast 对象和数据。在 Alauda Build of Feast 中，常见配置结合了：

- Kubernetes `Role` 和 `RoleBinding`
- 使用 `RoleBasedPolicy` 的 Feast `Permission` 对象

这使得为不同用户或工作负载授予只读和读写访问权限成为可能。

典型工作流

典型的 Feast 工作流如下：

1. 安装 Feast Operator。
2. 创建一个 `FeatureStore` CR 来部署 Feast 服务。
3. 在 feature repository 中定义 entities、feature views、feature services 和权限。
4. 运行 `feast apply` 将元数据注册到 registry 中。
5. 将 feature 数据 materialize 或 push 到 online store。
6. 从客户端应用、批处理作业或在线服务读取 feature。

Operator 职责

与手动部署 Feast 相比，Feast Operator 会管理 Feast 所需的 Kubernetes 资源。示例如下：

- 创建并协调 Feast Deployments 和 Services
- 初始化 feature repository，或从 Git 克隆一个 repository
- 为 offline、online 和 registry 数据提供基于 PVC 的持久化
- 创建 Feast 授权使用的可选 Kubernetes `Role` 对象
- 通过 CR status 和客户端 ConfigMaps 发布连接信息

延伸阅读

在本文档集中：

1. 安装 Feast
2. 快速开始

关于 Feast 的官方背景资料，请参阅：

- Feast concepts overview ↗
- Feast quickstart ↗
- Feast on Kubernetes ↗

安装 Feast

在 Alauda AI 2.8.x 中，**Alauda Build of Feast** 由 Alauda AI 安装并管理。将集群范围的 `default` `AmlCluster` 中的 `feast` 组件设置为 `Managed`；你无需从 OperatorHub 安装 Operator，也无需手动创建自定义资源。组件完成协调后，即可创建 `FeatureStore` 自定义资源来部署 Feast 服务。

目录

上传 Operator 软件包

启用 Feast

验证安装

后续步骤

上传 Operator 软件包

下载 Feast Operator 软件包，然后按照 上架软件包 上传该软件包。

启用 Feast

在 **Administrator** 视图中：

1. 打开 **Marketplace** > **OperatorHub**。

2. 选择目标集群并打开 **Alauda AI**。
3. 打开 **All Instances** 选项卡，并编辑 `default` `AmlCluster` 实例。
4. 将 `spec.components.feast.managementState` 设置为 `Managed`。
5. 保存 `AmlCluster` 实例，并等待 Feast Operator 就绪。

你可以在 `AmlCluster` YAML 中应用该组件设置：

```yaml
spec:
  components:
    feast:
      managementState: Managed
```

验证安装

检查 `AmlCluster` 状态以及 Feast Operator：

```
kubectl get amlcluster default

kubectl get pods -n feast-operator-system
```

`AmlCluster` 应报告 `Phase=Ready`，且 Feast Operator controller pod 应处于 `Running` 状态。Operator 就绪后，会注册 `FeatureStore` CRD：

```
kubectl get crd featurestores.feast.dev
```

后续步骤

启用该组件会创建 Operator、controller 和 CRD。应用 `FeatureStore` 自定义资源后，将创建 Feast 服务。

继续阅读 Quickstart，了解 `FeatureStore` 配置和部署示例。

快速入门

本页重点介绍 `FeatureStore` 自定义资源，它是 operator 安装完成后用于部署和配置 Feast 的主要接口。

目录

- 先决条件
- FeatureStore CR 概览
- 常见服务配置
  - 离线存储
  - 在线存储
  - registry
  - UI
- 常见持久化模式
  - 基于 PVC 的配置
  - Redis Online Store + SQL Registry
  - PostgreSQL Online Store + SQL Registry
  - 对象存储中的 registry
- 关于 Secret 的说明
- 特征仓库初始化
  - 默认初始化
  - Init 模板
  - Git 仓库
- 部署 FeatureStore

服务与访问
使用 Feast SDK 和 CLI
仓库布局
特征定义示例
CLI
配置授权
SDK token 配置
延伸阅读

先决条件

- 已安装 Feast Operator。请参见 安装 Feast。
- 对于基于 PVC 的示例，集群应具有默认的 `StorageClass`，或者应提前 জানা目标 `storageClassName`。
- 对于使用外部后端而不是基于 PVC 的文件存储，请提前准备所需服务：
  - 建议使用 PostgreSQL 16。PostgreSQL 用于 SQL registry，也可选用于 PostgreSQL online store。
  - 建议使用 Redis 6.0 或更高版本。Redis 用于 Redis online store。Feast 支持单个 Redis 实例、Redis Cluster 和 Redis Sentinel。
  - 在 `FeatureStore` CR 引用这些后端之前，请先准备包含后端连接参数的 Kubernetes Secret。

对于本页中的 PostgreSQL 和 Redis 示例，默认假定后端服务、数据库或 schema、凭据以及 Kubernetes Secret 已经存在。仍应在目标集群中验证连通性和驱动兼容性。

FeatureStore CR 概览

`FeatureStore` CR 中最常用的字段如下：

字段	用途	常见用法
`spec.feastProject`	Feast 项目名称	必需。用于组织特征定义和元数据。
`spec.feastProjectDir`	特征仓库的创建方式	使用默认的 `feast init`、init 模板或 Git 仓库。
`spec.services.offlineStore`	离线存储配置	历史数据存储和可选的远程离线 server。
`spec.services.onlineStore`	在线存储配置	低延迟的在线特征读取。
`spec.services.registry`	registry 配置	用于实体、feature view、feature service 和权限的元数据存储。
`spec.services.ui`	UI 服务	启用 Feast UI。
`spec.authz`	授权配置	通常是基于 Kubernetes 的角色或基于 OIDC 的访问控制。
`spec.replicas`	Pod 副本数	简单部署时保持为 `1`。对于多个副本，请使用由持久化存储支持的 durable store。

常见服务配置

离线存储

`offlineStore` 控制历史特征数据的存储位置，以及是否暴露远程 offline server。

常见选项：

- `persistence.file`：基于文件的存储，通常是在 PVC 上使用 DuckDB，适用于本地开发和评估
- `persistence.store`：外部离线后端，例如 BigQuery 或其他受支持的 Feast offline store
- `server: {}`：创建用于网络访问的远程 offline server Service

基于 PVC 的 DuckDB 示例：

```yaml
services:
  offlineStore:
    persistence:
      file:
        type: duckdb
        pvc:
          create:
            storageClassName: fast
            resources:
              requests:
                storage: 10Gi
          mountPath: /data/offline
    server: {}
```

基于 store 的示例：

```yaml
services:
  offlineStore:
    persistence:
      store:
        type: bigquery
        secretRef:
          name: feast-data-stores
    server: {}
```

在线存储

`onlineStore` 用于低延迟特征服务。

常见选项：

- `persistence.file`：基于文件的本地存储，通常仅用于开发
- `persistence.store`：Redis、PostgreSQL 或其他受支持的 Feast online store 后端
- `server`：可选的额外 server 设置，例如 `env`、`envFrom`、`resources`、`tls` 或 `volumeMounts`

本页示例推荐的后端版本：

- Redis 6.0 或更高版本
- PostgreSQL 16

基于 PVC 的文件示例：

```
services:
  onlineStore:
    persistence:
      file:
        path: online_store.db
        pvc:
          create:
            resources:
              requests:
                storage: 5Gi
          mountPath: /data/online
```

基于 Redis 的示例：

```
services:
  onlineStore:
    persistence:
      store:
        type: redis
        secretRef:
          name: feast-data-stores
```

基于 PostgreSQL 的示例：

```
services:
  onlineStore:
    persistence:
      store:
        type: postgres
        secretRef:
          name: feast-data-stores
    server:
      envFrom:
        - secretRef:
            name: postgres-secret
```

registry

`registry` 用于存储 Feast 元数据。它可以是当前 `FeatureStore` 的本地 registry，也可以复用远程 registry。

常见选项：

- `registry.local.persistence.file`：位于 PVC 或对象存储上的基于文件的 registry
- `registry.local.persistence.store`：基于 SQL 的 registry
- `registry.local.server: {}`：暴露 registry server 以供远程访问
- `registry.remote`：复用另一个 `FeatureStore` 或外部主机名中的 registry

对于本页中的 SQL-backed registry 示例，建议使用 PostgreSQL 16。

基于 PVC 的 registry 示例：

```yaml
services:
  registry:
    local:
      persistence:
        file:
          path: registry.db
          pvc:
            create:
              resources:
                requests:
                  storage: 5Gi
            mountPath: /data/registry
      server: {}
```

基于 SQL 的 registry 示例：

```yaml
services:
  registry:
    local:
      persistence:
        store:
          type: sql
          secretRef:
            name: feast-data-stores
      server: {}
```

远程 registry 示例：

```yaml
services:
  registry:
    remote:
      feastRef:
        name: <shared-featurestore-name>
        namespace: <shared-namespace>
```

UI

要启用 Feast UI，请设置：

```
services:
  ui: {}
```

如果不设置 `ui: {}`，则不会创建 UI Service。

常见持久化模式

基于 **PVC** 的配置

这是用于评估、本地测试和单副本部署的最简单模式：

```yaml
apiVersion: feast.dev/v1
kind: FeatureStore
metadata:
  name: <featurestore-name>
  namespace: <namespace>
spec:
  feastProject: <feast-project>
  replicas: 1
  services:
    offlineStore:
      persistence:
        file:
          type: duckdb
          pvc:
            create:
              resources:
                requests:
                  storage: 10Gi
            mountPath: /data/offline
      server: {}
    onlineStore:
      persistence:
        file:
          path: online_store.db
          pvc:
            create:
              resources:
                requests:
                  storage: 5Gi
            mountPath: /data/online
    registry:
      local:
        persistence:
          file:
            path: registry.db
            pvc:
              create:
                resources:
                  requests:
                    storage: 5Gi
              mountPath: /data/registry
        server: {}
    ui: {}
```

Redis Online Store + SQL Registry

这是用于持久化 online store 和基于数据库的 registry 的常见模式。

先创建 Secret：

```
apiVersion: v1
kind: Secret
metadata:
  name: feast-data-stores
  namespace: <namespace>
type: Opaque
stringData:
  redis: |
    connection_string: redis.<namespace>.svc.cluster.local:6379,password=
<redis-password>
  sql: |
    path: postgresql+psycopg://<user>:<password>@postgres.<namespace>.sv
c.cluster.local:5432/feast
    cache_ttl_seconds: 60
    sqlalchemy_config_kwargs:
      pool_pre_ping: true
      echo: false
```

然后在 `FeatureStore` 中引用它：

```yaml
apiVersion: feast.dev/v1
kind: FeatureStore
metadata:
  name: <featurestore-name>
  namespace: <namespace>
spec:
  feastProject: <feast-project>
  services:
    offlineStore:
      persistence:
        file:
          type: duckdb
          pvc:
            create: {}
            mountPath: /data/offline
      server: {}
    onlineStore:
      persistence:
        store:
          type: redis
          secretRef:
            name: feast-data-stores
    registry:
      local:
        persistence:
          store:
            type: sql
            secretRef:
              name: feast-data-stores
        server: {}
    ui: {}
```

PostgreSQL Online Store + SQL Registry

如果将 PostgreSQL 作为 online store 后端，Secret 可以同时包含 `postgres` 和 `sql` 条目：

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: feast-data-stores
  namespace: <namespace>
type: Opaque
stringData:
  postgres: |
    host: postgres.<namespace>.svc.cluster.local
    port: 5432
    database: feast
    db_schema: public
    user: <user>
    password: <password>
  sql: |
    path: postgresql+psycopg://<user>:<password>@postgres.<namespace>.sv
c.cluster.local:5432/feast
    cache_ttl_seconds: 60
    sqlalchemy_config_kwargs:
      pool_pre_ping: true
      echo: false
```

然后使用：

```yaml
services:
  onlineStore:
    persistence:
      store:
        type: postgres
        secretRef:
          name: feast-data-stores
  registry:
    local:
      persistence:
        store:
          type: sql
          secretRef:
            name: feast-data-stores
      server: {}
```

对象存储中的 registry

registry 文件也可以放在 S3 或 GCS 中：

```
services:
  registry:
    local:
      persistence:
        file:
          path: s3://bucket/registry.db
          cache_ttl_seconds: 60
          cache_mode: sync
```

关于 Secret 的说明

当使用 `persistence.store` 时，operator 会从 Kubernetes Secret 中读取后端参数。

重要说明：

- 默认情况下，Secret 键名与配置的 `type` 匹配
- 对于 `type: redis`，operator 读取 `redis` 键
- 对于 `type: postgres`，operator 读取 `postgres` 键
- 对于 `type: sql`，operator 读取 `sql` 键
- 如有需要，可使用 `secretKeyName` 覆盖键名

Secret 内容应遵循 `feature_store.yaml` 中使用的后端配置样式，但需要省略后端自身的 `type` 字段。

特征仓库初始化

operator 可以通过三种常见方式初始化特征仓库。

默认初始化

如果未设置 `spec.feastProjectDir`，operator 会执行默认的 `feast init <project>`，并在以下路径下创建仓库：

```
<offline mount path>/<feastProject>/feature_repo
```

Init 模板

当 operator 需要引导特定 Feast 模板时，可以使用 init 模板：

```
spec:
  feastProject: <feast-project>
  feastProjectDir:
    init:
      template: spark
```

Git 仓库

当特征定义已经在源码管理中维护时，可以使用 Git：

```
spec:
  feastProject: <feast-project>
  feastProjectDir:
    git:
      url: https://github.com/feast-dev/feast-credit-score-local-tutorial
      ref: 598a270
```

如果特征仓库不位于默认的 `feature_repo` 子目录中，请设置 `featureRepoPath`。

部署 FeatureStore

应用基于 PVC 的示例：

```
kubectl apply -f featurestore.yaml
```

主要的就绪字段是 `status.phase`。当部署就绪时，此字段会变为 `Ready`。

可通过以下命令直接检查：

```
kubectl get featurestore <featurestore-name> -n <namespace> -o jsonpath
='{.status.phase}'
```

服务与访问

operator 会将连接信息写入 `FeatureStore` 的 status：

```
kubectl get featurestore <featurestore-name> -n <namespace> -o jsonpath
='{.status.serviceHostnames}'
kubectl get featurestore <featurestore-name> -n <namespace> -o jsonpath
='{.status.clientConfigMap}'
```

operator 使用以下模式创建 Service 名称：`feast-<featurestore-name>-<component>`。常见示例如下：

- `feast-<featurestore-name>-online`
- `feast-<featurestore-name>-offline`
- `feast-<featurestore-name>-registry`
- `feast-<featurestore-name>-ui`

生成的 `clientConfigMap` 包含当前 `FeatureStore` 部署对应的客户端 `feature_store.yaml`。配置存储在 `feature_store.yaml` 键下，Feast 客户端可以直接加载，包括 Python 应用程序和集群内作业。

常见访问模式：

- 在集群内使用生成的 Service 名称
- 使用生成的 client ConfigMap 作为 Feast 客户端配置
- 通过目标环境中使用的访问方式暴露这些 Service

如果需要访问 offline server 或 registry server，则应在 CR 中设置 `offlineStore.server: {}` 或 `registry.local.server: {}`。

使用 Feast SDK 和 CLI

特征定义通常维护在 `FeatureStore` 工作负载之外，例如本地仓库、Git 仓库、CI 作业或其他基于 SDK 的工作流中。生成的客户端 `feature_store.yaml` 用于将 Feast CLI 或 Python SDK 连接到已部署的服务。

当从客户端环境运行 `feast apply` 时，registry service 必须可被 Feast 客户端访问。对于本页使用的本地 registry 模式，请设置：

```
services:
  registry:
    local:
      server: {}
```

如果还需要远程 materialization 或远程 historical retrieval，也请暴露 offline server：

```
services:
  offlineStore:
    server: {}
```

仓库布局

在 Feast 客户端环境中准备一个特征仓库，并将生成的客户端配置作为 `feature_store.yaml` 放入该仓库。配置可以从 `.status.clientConfigMap` 中指定的 `ConfigMap` 读取。

示例仓库布局：

```
<feature-repo>/
  feature_store.yaml
  driver_repo.py
  data/
```

仓库根目录是包含 `feature_store.yaml` 的目录。

Feast CLI 和 Python SDK 都使用同一个仓库根目录。对于 Python 代码，可通过以下方式加载：

```
from feast import FeatureStore

store = FeatureStore(repo_path=".")
```

特征定义示例

下面的示例定义了一个 entity、一个基于文件的 batch source、两个 feature view、一个 push source 和一个 feature service。这些对象足以在单个仓库中演示注册、在线服务、materialization 和基于 push 的更新。

`driver_stats_source` 从 `data/driver_stats.parquet` 读取批量数据。`driver_hourly_stats` 暴露这些字段，用于常规的 batch-to-online materialization。`driver_stats_push_source` 和 `driver_hourly_stats_fresh` 展示同一 entity 如何通过 push 接收新鲜值。`driver_activity_v1` 将该 feature view 组合为可复用的服务定义。

示例特征定义文件：

```
from datetime import timedelta

from feast import Entity, FeatureService, FeatureView, Field, FileSource,
PushSource
from feast.data_format import ParquetFormat
from feast.types import Float32, Int64
from feast.value_type import ValueType

driver = Entity(name="driver", join_keys=["driver_id"], value_type=ValueT
ype.INT64)

driver_stats_source = FileSource(
    name="driver_stats_source",
    path="data/driver_stats.parquet",
    file_format=ParquetFormat(),
    timestamp_field="event_timestamp",
    created_timestamp_column="created",
)

driver_hourly_stats = FeatureView(
    name="driver_hourly_stats",
    entities=[driver],
    ttl=timedelta(days=365),
    schema=[
        Field(name="conv_rate", dtype=Float32),
        Field(name="acc_rate", dtype=Float32),
        Field(name="avg_daily_trips", dtype=Int64),
    ],
    online=True,
    source=driver_stats_source,
)

driver_stats_push_source = PushSource(
    name="driver_stats_push_source",
    batch_source=driver_stats_source,
)

driver_hourly_stats_fresh = FeatureView(
    name="driver_hourly_stats_fresh",
    entities=[driver],
    ttl=timedelta(days=365),
    schema=[
        Field(name="conv_rate", dtype=Float32),
```

```
        Field(name="acc_rate", dtype=Float32),
        Field(name="avg_daily_trips", dtype=Int64),
    ],
    online=True,
    source=driver_stats_push_source,
)

driver_activity_v1 = FeatureService(
    name="driver_activity_v1",
    features=[driver_hourly_stats],
)
```

CLI

从特征仓库的根目录运行 `feast apply`。在上面的示例中，该命令是在 `<feature-repo>` 中运行的，而不是在 `data/` 子目录中运行。

`feast apply` 会从仓库根目录读取 `feature_store.yaml`，并递归扫描该仓库中的 Python 文件。如果特征定义存储在子目录中，只要这些文件位于仓库内且未被 `.feastignore` 排除，命令仍然应从仓库根目录运行。

如果从其他工作目录运行该命令，请显式指定仓库：

```
feast --chdir /path/to/<feature-repo> apply
```

默认形式为：

```
feast apply
```

配置授权

Feast 中基于 Kubernetes 的授权由三部分组成：

1. 在 `FeatureStore` CR 中声明角色名称。
2. 使用相同的角色名称在特征仓库中定义 Feast `Permission` 对象。
3. 通过 `RoleBinding` 将 Kubernetes 用户或 ServiceAccount 绑定到这些角色。

仅在 `FeatureStore` CR 中声明 `feast-reader` 和 `feast-writer` 并将主体绑定到这些角色，还不足以完成授权。Feast 的授权决策还需要特征仓库中匹配的 `Permission` 定义。

`FeatureStore` 片段示例：

```
spec:
  authz:
    kubernetes:
      roles:
        - feast-reader
        - feast-writer
```

特征仓库中的 `Permission` 定义示例：

```
from feast import Entity, FeatureService, FeatureView
from feast.permissions.action import ALL_ACTIONS, READ
from feast.permissions.permission import Permission
from feast.permissions.policy import RoleBasedPolicy

reader_perm = Permission(
    name="reader_perm",
    types=[Entity, FeatureView, FeatureService],
    policy=RoleBasedPolicy(roles=["feast-reader"]),
    actions=READ,
)

writer_perm = Permission(
    name="writer_perm",
    types=[Entity, FeatureView, FeatureService],
    policy=RoleBasedPolicy(roles=["feast-writer"]),
    actions=ALL_ACTIONS,
)
```

添加或更新 `Permission` 对象后，请运行 `feast apply`，以便将授权元数据注册到 Feast 中。

`RoleBinding` 配置示例：

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: feast-reader-sa
  namespace: <namespace>
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: feast-writer-sa
  namespace: <namespace>
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: feast-reader-binding
  namespace: <namespace>
subjects:
  - kind: ServiceAccount
    name: feast-reader-sa
    namespace: <namespace>
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: feast-reader
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: feast-writer-binding
  namespace: <namespace>
subjects:
  - kind: ServiceAccount
    name: feast-writer-sa
    namespace: <namespace>
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: feast-writer
```

创建示例 ServiceAccount token：

```
READER_TOKEN=$(kubectl create token feast-reader-sa -n <namespace>)
WRITER_TOKEN=$(kubectl create token feast-writer-sa -n <namespace>)
```

如果需要自定义 token 生命周期，请在命令中添加 `--duration`。

SDK token 配置

启用 `authz.kubernetes` 后，Feast Python 客户端会自动将 token 作为 bearer token 发送。对于运行在同一集群内且挂载了 ServiceAccount token 的客户端，通常不需要额外的 SDK 设置。

对于外部客户端，可通过以下方式之一提供 token：

- 在 `feature_store.yaml` 中设置 `authz_config.user_token`
- 设置 `LOCAL_K8S_TOKEN` 环境变量

`feature_store.yaml` 片段示例：

```
authz_config:
  type: kubernetes
  user_token: <kubernetes-token>
```

环境变量示例：

```
export LOCAL_K8S_TOKEN=<kubernetes-token>
```

延伸阅读

有关端到端 Feast 工作流和更广泛的产品用法，请继续阅读 Feast 官方文档：

- 概念↗：Feast 核心对象和数据模型
- 架构↗：部署模型、检索路径和服务模式
- 快速入门↗：包含 SDK 用法的端到端本地工作流

- 示例用例教程↗：面向常见 ML 工作流的场景化教程
- 在生产环境中运行 Feast↗：生产模式和运维指导

训练

本章涵盖微调和预训练、训练运行时镜像、检查点与恢复、GPU 切片，以及训练作业的配额和调度治理。

本章依赖的组件列在下方的 Components 中：Kueue 负责队列、配额和抢占，而 Volcano 和 JobSet 负责批处理编排。训练期间的实验跟踪和模型注册使用 Develop 章节中的 MLflow。

训练指南

Kubeflow Trainer 快速入门

运行时镜像

ClusterTrainingRuntime

运行示例 notebook

使用 Kubeflow Trainer v2 进行 F

前提条件

构建或使用预构建镜像

运行示例 notebook

使用 Kueue 进行调度

Checkpoint

前置条件

checkpoint 必须

选择能在 Pod 之

让 runtime 感知

恢复

何恢

使用 Kueue、Checkpointing 和 Inference 共存的可抢占 TrainJobs

前提条件

cohort：一个 CQ 保留配额，另一个借用配

让 TrainJob 具备抢占安全性

提交工作负载

与在线 InferenceService 安全共存

保留并共享：用于 namespace 级保留的对

验证配置

使用 Dynamic Resource Allocation (DRA) 进行

device plugin 与 DRA 的对比

前提条件

一个切片请求中的三个对象

第 1 步 —— 确认 driver 正在发布切片

第 2 步 —— 对切片做冒烟测试

第 3 步 —— 使用 Kubeflow Trainer v2 在切

选择切片大小 / 选择 profile

Training Ru

可用运行时

没有 MIG？通过 time-slicing 共享整张 GPU
将切片与 Kueue 配额结合使用
将同一个 claim 接入其他 orchestrator
将其转变为真实微调
选择运行时
应用 TrainingRu
提交 TrainJob
设备资源请求
项

使用 **Training Hub** 对 **LLM** 进行微调

要求
数据格式
运行示例 notebooks
关键参数
QLoRA（4-bit LoRA）
持续预训练 (CPT)
多节点

使用 **MLflow**

该 pipeline 的作
前置条件
步骤 1 — 创建 I
步骤 2 — pipeli

使用 **Workbench** 微调 **LLM**

在 **Ascend NPU** 上微调和预训练 **LLM**

开始之前
准备基础模型
准备数据集
运行 notebook
需要查看的参数
输出路径
注意事项

9. 发布微调后的模型
在非 NVIDIA GPU 上运行

配额与调度

配置 RBAC

1. 为批处理管理员授予权限
2. 为批处理用户授予权限

配置配额

1. 配置集群队列
2. 配置资源风味
3. 配置本地队列

使用 cohort

在 cluster queu

共享

Gang 调度

配置 Gang 调度

使用 Kueue 管理 Ascend NPU 配额

前提条件

操作步骤

注意事项

组件

Kueue

Volcano

JobSet

训练指南

在 Alauda AI 上对 LLMs 进行微调和预训练的端到端配方。

目录

选择路径

选择路径

你想要的场景	使用	指南
可复用模板、可重复运行、可选的 Kueue 配额	Kubeflow Trainer v2 + LlamaFactory	使用 Kubeflow Trainer v2 进行微调
在崩溃、抢占或暂停后恢复，而不会损失数小时的训练时间	Checkpoint PVC + `resume_from_checkpoint` + `maxRestarts`	检查点保存与恢复 TrainJobs
将训练与在线推理结合，在需要时归还 GPU	Kueue cohort + 抢占 + 检查点恢复	使用 Kueue 的可抢占 TrainJobs
在 GPU 的一个 *切片* 上运行微调，而不是整卡	Dynamic Resource Allocation (DRA) MIG slice + Trainer v2	使用 Dynamic Resource Allocation (DRA) 进行 GPU 切片

你想要的场景	使用	指南
一组 Curated 的 `TrainingRuntime` 镜像（CUDA / CANN）	Trainer v2 runtime catalog	Training Runtime 镜像
在 Trainer v2 上一键快速开始分布式 PyTorch	`ClusterTrainingRuntime` + MNIST	Kubeflow Trainer 快速开始
具有自动内存管理的生产级 SFT / OSFT	`training_hub`	使用 Training Hub 微调 LLMs
交互式探索、自定义脚本、VolcanoJob 提交	Workbench Notebook	使用 Workbench 进行 LLMs 微调
在 Ascend NPU 上进行全参数 SFT / 预训练	Workbench `PyTorch CANN` / `MindSpore CANN`	在 Ascend NPU 上微调和预训练
在 MLflow 中跟踪 KFP 运行的参数和指标	KFP component + MLflow SDK	Kubeflow Pipeline + MLflow 集成
使用可复用的 AutoGluon assets 训练表格或时间序列模型	Managed KFP pipelines or composable components	使用可复用的 Kubeflow Pipeline 组件
使用 KFP 缓存、并行循环和持久化的 typed artifacts	KFP 2.16.1 execution mechanisms	Kubeflow Pipelines 执行与存储行为
通过 MLflow + TrustyAI 实现每日微调 → 评估 → 比较循环	KFP Recurring Run + MLflow Model Registry + `LMEvalJob`	结合 MLflow 和 TrustyAI 的每日微调流水线

Kubeflow Trainer 快速入门

在 Alauda AI 上使用 Kubeflow Trainer v2 进行最小化分布式 PyTorch 训练的配置：自定义运行时镜像、`ClusterTrainingRuntime` 和一个 MNIST 示例 notebook。

目录

运行时镜像

ClusterTrainingRuntime

运行示例 notebook

运行时镜像

使用预构建镜像 `alaudadockerhub/torch-distributed:v2.9.1-aml2`，或者基于此 `torch_distributed.Containerfile` 构建你自己的镜像：

```
FROM python:3.13-trixie
ARG USERNAME=appuser
ARG USER_UID=1000
ARG USER_GID=$USER_UID

RUN sed -i 's/deb.debian.org/mirrors.ustc.edu.cn/g' /etc/apt/sources.lis
t.d/debian.sources && \
    apt-get update && apt-get install -y build-essential

RUN pip install --no-cache-dir -i https://pypi.tuna.tsinghua.edu.cn/simpl
e -U pip && \
    pip install --no-cache-dir --index-url https://download.pytorch.org/w
hl/cu126 \
        "torch==2.9.1" "torchvision==0.24.1"

RUN groupadd --gid $USER_GID $USERNAME \
    && useradd --uid $USER_UID --gid $USER_GID -m $USERNAME

WORKDIR /workspace
RUN chown $USERNAME:$USER_GID /workspace
```

ClusterTrainingRuntime

请以集群管理员身份应用此 `kf-torch-distributed.yaml`。该 Pod spec 针对 Alauda AI 的默认 PSA 进行了收紧。

```yaml
apiVersion: trainer.kubeflow.org/v1alpha1
kind: ClusterTrainingRuntime
metadata:
  name: torch-distributed
  labels:
    trainer.kubeflow.org/framework: torch
spec:
  mlPolicy:
    numNodes: 1
    torch:
      numProcPerNode: auto
  template:
    spec:
      replicatedJobs:
        - name: node
          template:
            metadata:
              labels:
                trainer.kubeflow.org/trainjob-ancestor-step: trainer
            spec:
              template:
                spec:
                  securityContext:
                    runAsNonRoot: true
                    runAsUser: 1000
                    runAsGroup: 1000
                    fsGroup: 1000
                  volumes:
                    - name: dshm
                      emptyDir:
                        medium: Memory
                        sizeLimit: 2Gi
                    - name: workspace
                      emptyDir: {}
                  containers:
                    - name: node
                      image: alaudadockerhub/torch-distributed:v2.9.1-aml
2
                      env:
                        - { name: TORCH_HOME,           value: /tmp/torch
_cache }
                        - { name: TORCH_EXTENSIONS_DIR, value: /tmp/torch
_extensions }
```

```
                    - { name: TRITON_CACHE_DIR,     value: /tmp/trito
n_cache }
                  volumeMounts:
                    - { name: workspace, mountPath: /workspace }
                    - { name: dshm,      mountPath: /dev/shm }
                  securityContext:
                    allowPrivilegeEscalation: false
                    capabilities: { drop: [ALL] }
                    runAsNonRoot: true
                    seccompProfile: { type: RuntimeDefault }
```

运行示例 **notebook**

该 notebook 会安装 Python 包并下载 MNIST，因此工作台需要具备出站网络访问能力。

下载 `kubeflow-trainer-mnist.ipynb` ↗ 并上传到你的工作台，然后按照其中的步骤提交 `TrainJob`。

有关 Trainer v2 功能的背景信息，请参阅上游 Kubeflow Trainer 文档 ↗。

使用 Kubeflow Trainer v2 进行 Fine-Tuning

使用 Kubeflow Trainer v2 在 Kubernetes 上通过 LlamaFactory 运行 supervised fine-tuning。

Trainer v2 将作业拆分为可复用的 `TrainingRuntime`（镜像 + pipeline 步骤 + LlamaFactory 配置）和按实验运行的 `TrainJob`，后者只覆盖发生变化的部分（模型、数据集、超参数、GPU 资源）。

目录

前提条件

构建或使用预构建镜像

运行示例 notebook

使用 Kueue 进行调度

前提条件

要求	详情
Kubeflow Trainer v2	可用的 `trainer.kubeflow.org` API group
Kueue	可选；用于作业排队和配额

要求	详情
Shared PVC	在所有 training pod 上均可用的 RWX，或已正确配置的 RWO
Git 凭据	`Secret` `aml-image-builder-secret`，包含 `MODEL_REPO_GIT_USER` 和 `MODEL_REPO_GIT_TOKEN`
GPU 节点	NVIDIA GPU；根据你的节点调整 `nodeSelector`
`kubectl` 访问	在你的 namespace 中管理 `trainingruntimes` 和 `trainjobs` 的权限

如果遇到 RBAC 错误，请让集群管理员为你的 workbench ServiceAccount 授予在目标 namespace 中对 `trainjobs` 和 `trainingruntimes` 的读写权限（示例 role：`apiGroups: ["trainer.kubeflow.org"]`，`resources: ["trainjobs","trainingruntimes"]`）。

构建或使用预构建镜像

可以使用 `alaudadockerhub/fine_tune_with_llamafactory:v0.1.11`，也可以根据 `assets/build-train-image/` ↗ 下的 `Containerfile` 自行构建。

运行示例 **notebook**

将 `fine-tune-with-trainer-v2.ipynb` ↗ 下载到你的 workbench 中，并按单元格执行。该 notebook 会创建一个 `TrainingRuntime`，然后提交一个挂载共享 PVC 并使用 `aml-image-builder-secret` 的 `TrainJob`。

对于华为 Ascend NPU，请改用 `fine-tune-with-trainer-v2-mindspeed-npu.ipynb` ↗ —— 它会在 `huawei.com/Ascend910B4` 资源上使用 `runtimeClassName: ascend` 运行 MindSpeed-LLM SFT pipeline（HF → MCore checkpoint、preprocess、train）。

使用 **Kueue** 进行调度

当安装了 Kueue ↗ 时，TrainJob 会保持 **suspended**，直到 Kueue 基于已配置的 `ClusterQueue` quota 接纳它们。可直接应用的 YAML 位于 `assets/kueue/` ↗：

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/train/guides/assets/kueue
NS=my-namespace  # edit to the namespace where you submit jobs
# 1. Cluster admin — one ResourceFlavor + one ClusterQueue (edit nominalQuota to taste)
kubectl apply -f $base/cluster-queue.yaml
# 2. Namespace admin — LocalQueue pointing at the ClusterQueue
curl -fsSL $base/local-queue.yaml | sed "s/<your-namespace>/$NS/" | kubectl apply -f -
# 3. Submit a TrainJob labelled with the queue name; Kueue admits it
curl -fsSL $base/trainjob-kueue-example.yaml | sed "s/<your-namespace>/$NS/" | kubectl create -f -
```

这三个文件分别是：

- `cluster-queue.yaml` ↗ — 一个 `ResourceFlavor` 加一个覆盖 cpu / memory / `nvidia.com/gpu` 的 `ClusterQueue`。集群管理员需要在每个 quota pool 上应用一次。
- `local-queue.yaml` ↗ — 一个命名空间级别的 `LocalQueue`，引用 `cluster-queue`。namespace 管理员需要在每个 namespace 上应用一次。
- `trainjob-kueue-example.yaml` ↗ — 一个带有标签 `kueue.x-k8s.io/queue-name: local-queue` 的 `TrainJob`。该 TrainJob 会保持 Suspended，直到 Kueue 接纳它；一旦被接纳，JobSet 就会启动 trainer pod。

完整配置请参见 Kueue docs。

> **NOTE**
>
> 当 Kueue 的 **PodsReady timeout** 较短且训练镜像较大时，首次尝试可能会因 image-pull timeout 而被驱逐。重新提交通常会成功，因为镜像已经缓存在节点上。

Checkpointing 和 TrainJob 恢复

一个运行六小时的微调，最终总会遇到节点重启、Pod 被驱逐、抢占，或者在第 5 小时发生 OOM。没有 checkpoint，这些事件中的任何一个都会让你付出整个训练过程的代价。

Kubeflow Trainer v2 没有自己的 **checkpoint API**。`TrainJob` 或 `TrainingRuntime` 规范中没有任何内容会写入、查找或恢复 checkpoint。职责划分如下：

Trainer v2 / JobSet 提供的能力	你的训练脚本必须完成的事情
持久化到 Pod 之外的存储（你挂载的 PVC）	定期将 checkpoint 写入其中
失败后重启 Pod（`failurePolicy.maxRestarts`）	在启动时检测最新的 checkpoint，并从中恢复
按需暂停和恢复（`spec.suspend`）	在 SIGTERM 下优雅退出，并刷新最后一个 checkpoint

如果任意一列做错，失败模式都是一样的，而且是静默的：Job 重启后找不到任何 checkpoint，然后愉快地从 step 0 重新训练。

本指南介绍其工作机制。对于故意中断训练、把 GPU 还给 `InferenceService` 的特定场景，请参见 使用 Kueue 的可抢占 TrainJob，它建立在本文内容之上。

目录

前置条件

checkpoint 必须包含什么

选择能在 Pod 之外存活的存储
让 runtime 感知 checkpoint
提交一个可恢复的 TrainJob
各类中断如何恢复
崩溃或节点故障
抢占
有意暂停
各框架的参数开关
多节点运行
验证它确实能恢复
故障排查

前置条件

要求	详情
Kubeflow Trainer v2	`trainer.kubeflow.org/v1alpha1`；参见 使用 Kubeflow Trainer v2 进行微调
一个 PVC 和一个 storage class	`kubectl get storageclass`。访问模式请参见 选择能在 Pod 之外存活的存储
一个训练运行时镜像	来自 runtime 目录 的任意镜像
`kubectl` 访问权限	允许在你的 namespace 中管理 `trainjobs`、`trainingruntimes` 和 `persistentvolumeclaims`

checkpoint 必须包含什么

“恢复”并不等于“重新加载权重”。只保存 model weights 的 checkpoint 会让 optimizer 从零开始——momentum 缓冲区、learning-rate 调度位置以及 step 计数器都会丢失，loss 曲线会在恢复点处出现明显折角。

一个可恢复的 checkpoint 应包含：

- **Model weights**
- **Optimizer state** — Adam 的 moment 估计。通常约为模型大小的 ~2 倍（fp32），也是 checkpoint 中最大的一部分。
- **LR scheduler state** — 这样学习率调度才能继续，而不是重新 warm up。
- **Step / epoch counter** — 用于从哪里继续。
- **RNG state** 和 **data-loader** 位置 — 这样恢复后的运行不会以相同顺序再次看到同一批样本。

HuggingFace Trainer 的 `checkpoint-N/` 目录已经包含了以上所有内容。如果你是手写训练循环，只保存 `model.state_dict()` 是最常见的“恢复”方式失效的原因，它会悄悄破坏训练过程。

选择能在 **Pod** 之外存活的存储

checkpoint 目录必须位于 PVC 上。`emptyDir` 会随着 Pod 一起消失，而 `hostPath` 或本地存储 class 会把 checkpoint 留在某个节点上，而重启后的 Pod 可能永远不会再调度到那个节点。

访问模式只取决于有多少个 trainer Pod 会同时写入：

`numNodes`	访问模式	原因
`1`	`ReadWriteOnce` 就够了	一次只有一个 Pod 挂载它
`> 1`	需要 `ReadWriteMany`	每个 trainer Pod 同时挂载同一个 PVC

> **NOTE**
>
> `ReadWriteOnce` 的含义是“一次一个节点”，而不是“永远只有一个节点”。一个通过网络挂载的 RWO volume（如 Ceph RBD、EBS 及类似存储）会从旧节点卸载，并在重启后的 Pod 被调度到哪里时重新挂载，因此单节点运行即使恢复到不同节点也能正常继续。只有并发的多 Pod 访问才需要 RWX。

这不适用于 `local-storage` / `topolvm` / `hostPath`，它们在物理上固定在某一个节点上。

应用该 PVC（`checkpoint-pvc.yaml` ↗）：

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/train/guides/assets/checkpointing
NS=my-namespace   # edit to your namespace

curl -fsSL $base/checkpoint-pvc.yaml | sed "s/<your-namespace>/$NS/" | kubectl apply -f -
```

容量按 `checkpoint size x save_total_limit` 再加上余量来规划。完整权重 checkpoint 通常比人们预期的大——一个带 optimizer state 的 7B 模型大约每个 checkpoint 80 GiB。LoRA/QLoRA adapter 只有几 MB，因此同一个 PVC 可以保存更多此类 checkpoint。

让 runtime 感知 checkpoint

`checkpoint-trainingruntime.yaml` ↗ 是一个可直接运行的 `TrainingRuntime`，完整模式已经配置好。关键部分如下：

```yaml
spec:
  template:
    spec:
      failurePolicy:
        maxRestarts: 3                          # recreate trainer pods on
failure
      replicatedJobs:
        - name: node
          template:
            metadata:
              labels:
                trainer.kubeflow.org/trainjob-ancestor-step: trainer   #
see warning below
            spec:
              backoffLimit: 0                   # let failurePolicy own res
tarts
              template:
                spec:
                  terminationGracePeriodSeconds: 60   # room to flush on
SIGTERM
                  volumes:
                    - name: ckpt
                      persistentVolumeClaim: { claimName: train-ckpt }
                  containers:
                    - name: node
                      env:
                        - { name: CKPT_DIR, value: /mnt/ckpt/run }
                      volumeMounts:
                        - { mountPath: /mnt/ckpt, name: ckpt }
```

以及脚本中真正执行恢复的两行：

```python
# Sort numerically: a lexical sort puts checkpoint-9 after checkpoint-100
# and silently rewinds your run by 91 steps.
ckpts = sorted(glob.glob(f"{ckpt_dir}/checkpoint-*"),
               key=lambda p: int(p.rsplit("-", 1)[1]))
resume = ckpts[-1] if ckpts else None    # None on the first run = start
clean

trainer.train(resume_from_checkpoint=resume)
```

这种自动检测机制使同一个 manifest 既能用于首次提交，也能用于之后的每次重启。硬编码 `resume_from_checkpoint=True` 会让第一次运行失败（因为那时还没有 checkpoint）；硬编码路径则会把你永远绑定到某一个 checkpoint。

> **WARNING**
>
> trainer `replicatedJob` 上的 `trainer.kubeflow.org/trainjob-ancestor-step: trainer` label 是必需的。没有它，controller 不会识别该 replicatedJob 是 trainer step，而你的 TrainJob 中所有 `spec.trainer.*` 字段——`env`、`resourcesPerNode`、`image`、`command`、`numNodes`——都会被静默丢弃：没有错误、没有事件、没有警告。TrainJob 会改用 runtime 的默认值运行，因此你以为分配了 GPU 和 `CKPT_DIR` 的 Job，实际上两者都没有。
>
> 请验证这些覆盖项是否真正生效，而不要想当然地相信它们：
>
> ```
> kubectl -n "$NS" get pod <trainer-pod> \
>   -o jsonpath='{.spec.containers[0].env}{"\n"}{.spec.containers[0].resources}{"\n"}'
> ```

应用该 runtime：

```
curl -fsSL $base/checkpoint-trainingruntime.yaml | sed "s/<your-namespace>/$NS/" | kubectl apply -f -
```

提交一个可恢复的 TrainJob

`trainjob-resume.yaml` ↗ 通过 `podTemplateOverrides` 挂载 checkpoint PVC，因此一个共享 runtime 可以服务多个运行，而每个运行都把 checkpoint 写到不同位置：

```
spec:
  trainer:
    env:
      - { name: CKPT_DIR, value: /mnt/ckpt/sft-run-1 }   # per-run direct
ory
  podTemplateOverrides:
    - targetJobs:
        - name: node
      spec:
        volumes:
          - name: ckpt
            persistentVolumeClaim: { claimName: train-ckpt }
        containers:
          - name: node
            volumeMounts:
              - { name: ckpt, mountPath: /mnt/ckpt }
```

```
curl -fsSL $base/trainjob-resume.yaml | sed "s/<your-namespace>/$NS/" | k
ubectl create -f -
```

NOTE

`podTemplateOverrides` 可以添加 **volumes**、**volumeMounts**、**nodeSelector**、**tolerations** 和 **affinity**，但它不能为 trainer 或 initializer 容器设置 `env`——validating webhook 会直接拒绝该 TrainJob：

```
admission webhook "validator.trainjob.trainer.kubeflow.org" denied
the request:
spec.podTemplateOverrides: must not have envs for the dataset-initi
alizer,
model-initializer, node containers
```

环境变量应放在 `spec.trainer.env` 中。

各类中断如何恢复

崩溃或节点故障

有两层相互独立的机制会重启失败的 trainer，而具备 checkpoint 感知能力的脚本在这两种情况下都能正确恢复：

```
  trainer exits non-zero
        │
        ├─ restartPolicy: OnFailure ──▶ kubelet restarts the container
        │                                in place, same pod, same node
        │
        └─ backoffLimit: 0 ──▶ Job fails ──▶ JobSet failurePolicy.maxRest
arts
                                              recreates the pod (possibly
on
                                              another node), PVC re-attac
hes
```

无论哪种情况，新进程都会执行相同的自动检测，并接着使用最新的 `checkpoint-N`。观察 JobSet 上的 `restarts` 递增：

```
kubectl -n "$NS" get jobset <trainjob-name> -o jsonpath='{.status.restart
s}{"\n"}'
```

一旦 `maxRestarts` 用尽，TrainJob 就会变为 `Failed`——因此它的值要足够高，以吸收临时性的节点波动，但也不要高到让真正损坏的脚本无限循环。

抢占

Kueue 会驱逐 Workload，而 Trainer v2 会在 quota 重新释放后重建 JobSet；trainer 会从 PVC 恢复。cohort/quota 的配置本身是一个独立话题——请参见 使用 Kueue 的可抢占 TrainJob。

有意暂停

`spec.suspend` 可以在不丢失训练状态的情况下暂停运行——例如临时把 GPU 借给更紧急的任务一小时：

```
# Pause: pods are terminated, the TrainJob object stays.
kubectl -n "$NS" patch trainjob sft-resumable --type=merge -p '{"spec":
{"suspend":true}}'

# Resume: a new pod starts and picks up the newest checkpoint.
kubectl -n "$NS" patch trainjob sft-resumable --type=merge -p '{"spec":
{"suspend":false}}'
```

恢复后的 Pod 是一个新的 Pod，具有新的名称，并且可能被调度到不同的节点。它的第一行日志应该是你的 `[checkpoint] resuming from ...`。

> **NOTE**
>
> 当一个被暂停的 TrainJob 恢复时，`activeDeadlineSeconds` 会重新开始计时——也就是说，每次暂停后，Job 都会重新获得完整的截止时间，而不是从创建时刻开始累计。

各框架的参数开关

所有基于 HuggingFace-Trainer 的栈都暴露相同的三个开关，因此上面的模式可以原样迁移：

Stack	保存频率	恢复方式
HuggingFace `Trainer`	`save_strategy="steps"`, `save_steps=N`, `save_total_limit=2`	`trainer.train(resume_from_checkpoint=path)`
LlamaFactory	YAML 配置中的 `save_steps`	自动——它会替你调用 `get_last_checkpoint(output_dir)`
`training_hub`	`checkpoint_at_epoch: true`	`accelerate_full_state_at_epoch: true`

Stack	保存频率	恢复方式
MindSpeed / Megatron (NPU)	`--save-interval`	自动从 `latest_checkpointed_iteration.txt` 恢复
手写 PyTorch	你自己的 `torch.save`	你自己的 `torch.load`

根据你能够容忍的中断时间来选择 `save_steps`：风险最大的工作量最多就是 `save_steps x seconds_per_step`。如果每步 5 s，`save_steps: 100` 的风险大约是 8 分钟。保存过于频繁本身也有成本——完整权重 checkpoint 可能会让 GPU 停顿几十秒，因此在稳定集群上可以把值设大一些，在可抢占队列上则应设小一些。务必同时设置 `save_total_limit`，否则 PVC 会无限增长。

多节点运行

当 `numNodes: > 1` 时，PVC 必须是 RWX，并且会有两点变化：

- 对于标准的数据并行训练，只有 **rank 0** 应写入 checkpoint，否则每个 rank 都会争抢写同一路径。HF Trainer 已经处理了这一点；手写循环必须用 `if rank == 0` 进行保护。
- 所有 **rank** 必须从同一个 **checkpoint** 恢复。 如果不同 rank 恢复到不同 checkpoint，会静默分叉，而不是崩溃。

对于分片策略（FSDP、DeepSpeed ZeRO），每个 rank 持有一部分 weights 和 optimizer state，把所有内容汇总到 rank 0 往往对大模型来说根本不可行。这正是 `torch.distributed.checkpoint` ↗ 的用途：每个 rank 并行写入自己的 shard，结果可以在不同的 **world size** 下重新加载——因此一个在 8 张 GPU 上 checkpoint 的运行可以恢复到 4 张 GPU 上。保存为单个聚合文件的 checkpoint 则无法以这种方式重新分片。

验证它确实能恢复

不要等到真正的故障发生后才发现问题。每个新的 runtime 都先执行一次这个演练：

```
# 1. Submit, and wait for a first checkpoint to appear on the PVC.
kubectl -n "$NS" exec <trainer-pod> -- ls /mnt/ckpt/sft-run-1

# 2. Force an interruption.
kubectl -n "$NS" patch trainjob sft-resumable --type=merge -p '{"spec":
{"suspend":true}}'
kubectl -n "$NS" patch trainjob sft-resumable --type=merge -p '{"spec":
{"suspend":false}}'

# 3. The new pod's first lines must say it resumed -- not that it started
fresh.
kubectl -n "$NS" logs <new-trainer-pod> | head -3
```

```
[checkpoint] resuming from /mnt/ckpt/sft-run-1/checkpoint-150
```

如果第 3 步打印的是 `no prior checkpoint, starting fresh`，说明恢复有问题——真实抢占发生时也一样会有问题。

> **NOTE**
>
> 通过 grep trainer 日志中的 HF Trainer “Saving model checkpoint to” 这一行并不可靠：tqdm 进度条会覆盖它。应当检查 PVC 上是否存在 `checkpoint-*` 目录，上面的日志命令正是在做这件事。

故障排查

症状	原因
重启后的 Job 从 step 0 开始训练	checkpoint 目录位于 `emptyDir` 上，或不同运行之间 `CKPT_DIR` 不一致，或自动检测的 glob 没找到任何内容
从一个更早的 checkpoint 恢复	`checkpoint-*` 的字典序排序问题（`checkpoint-9` 会排在 `checkpoint-100` 后面）——请按数字排序
`spec.trainer.env` / `resourcesPerNode` 被忽略，且没有报错	runtime 的 trainer replicatedJob 上缺少 `trainer.kubeflow.org/trainjob-ancestor-step: trainer` label

症状	原因
TrainJob 被拒绝：`must not have envs for the ... node containers`	在 `podTemplateOverrides` 下设置了 `env`；请将其移到 `spec.trainer.env`
恢复后 loss 曲线向上折起	checkpoint 只有 weights，没有 optimizer/scheduler state
恢复时 Pod 处于 `Pending`，volume 无法挂载	RWO PVC 仍挂载在旧节点上，或者在非 RWX class 上使用了 `numNodes > 1`
运行中 PVC 已满	未设置 `save_total_limit`
抢占后缺少最终 checkpoint	`terminationGracePeriodSeconds` 太短，来不及刷新

使用 Kueue、Checkpointing 和 Inference 共存的可抢占 TrainJobs

当训练作业与在线 `InferenceService` 工作负载共享一个集群时，你希望同时满足两点：

- **Inference** 受到保护。 它始终拥有所需的 GPU；队列准入是恒定时间的，并且不会被训练作业阻塞。
- **Training** 充分利用空闲资源。 只要 inference 低于峰值，训练就可以借用空闲 GPU 并继续推进；一旦 inference 重新收回配额，训练便立即让出资源。

本指南通过一组精简的 asset YAML，将 Kubeflow Trainer v2 + Kueue + HuggingFace Trainer checkpointing 组合起来，实现上述行为。这里的所有内容都已在仓库 e2e harness 中的 `c12_kueue_preemption.sh` ↗ 用例中完成端到端验证。

目录

前提条件

cohort：一个 CQ 保留配额，另一个借用配额

让 TrainJob 具备抢占安全性

提交工作负载

与在线 InferenceService 安全共存

保留并共享：用于 namespace 级保留的对称 cohort

实际操作中的调节项

什么时候选择哪种布局

验证配置

前提条件

要求	详情
Kubeflow Trainer v2	`trainer.kubeflow.org` API group；参见 使用 Kubeflow Trainer v2 进行 Fine-Tuning
Kueue（`v1beta2` API 需要 v0.13+）	参见 安装 Kueue
共享 RWX 存储	checkpoint PVC 必须能从训练器在重新准入后可能调度到的任意节点访问
GPU device plugin	示例使用 Alauda Build of HAMI↗ 的 vGPU 资源（`nvidia.com/gpualloc`、`gpucores`、`gpumem`）；如果你使用上游 NVIDIA device plugin，则替换为 `nvidia.com/gpu`
训练运行时镜像	Trainer v2 runtime catalog 中任意包含你所用框架的镜像

cohort：一个 **CQ** 保留配额，另一个借用配额

核心思路是一个 双 **ClusterQueue cohort**。Inference 持有 GPU 的名义配额；training 持有 0，但当 inference 空闲时，允许训练借用相同数量的配额。当 inference 工作负载收回其配额时，Kueue 会驱逐借用资源的训练 Workload，而 Trainer v2 会在配额释放后立即从头重建 JobSet。

```
                    ┌──────────────────────────────┐
                    │  Cohort: c12-shared          │
                    │                              │
   inference label ──┤  c12-inference-cq            │
                    │    nominalQuota: 1 GPU       │
                    │    borrowingLimit: 0         │ ← inference never b
orrows
                    │    reclaimWithinCohort: Any  │
                    │                              │
   training label  ──┤  c12-training-cq             │
                    │    nominalQuota: 0 GPU       │ ← owns nothing
                    │    borrowingLimit: 1 GPU     │ ← borrows when idle
                    │                              │
                    └──────────────────────────────┘
```

应用 cohort 和按 namespace 划分的 LocalQueue：

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/tra
in/guides/assets/kueue/preemption
NS=my-namespace  # edit to the namespace where you submit jobs

# 1. Cluster admin — one ResourceFlavor + two cohort ClusterQueues.
kubectl apply -f $base/cluster-queues.yaml

# 2. Cluster admin — Kueue WorkloadPriorityClasses for inference / traini
ng.
kubectl apply -f $base/workload-priorities.yaml

# 3. Namespace admin — LocalQueues pointing at the cohort.
curl -fsSL $base/local-queues.yaml | sed "s/<your-namespace>/$NS/" | kube
ctl apply -f -
```

这些 asset 文件分别是：

- `cluster-queues.yaml` ↗ — 包含 cohort、两个 ClusterQueue 以及 `ResourceFlavor`。修改 `nominalQuota` 和 `borrowingLimit`，以匹配你希望借出的 GPU 数量。
- `workload-priorities.yaml` ↗ — 两个 `WorkloadPriorityClass` 值：`c12-inference-prio=1000`、`c12-training-prio=10`。如果没有这些配置，cohort 的回收规则仍会生效，但队列内不会有优先级顺序。

- `local-queues.yaml` ↗ — `c12-inference-lq` 和 `c12-training-lq`，每个 ClusterQueue 对应一个。

让 TrainJob 具备抢占安全性

被抢占的 TrainJob 的 Pod 会被终止（先 SIGTERM，宽限期结束后再 SIGKILL）。要在这种情况下继续运行而不是从头开始，你需要：

1. 在 **RWX PVC** 上放置 **checkpoint** 目录。 抢占后的 Pod 可能会落到不同节点上——本地存储不够用。
2. 频繁保存 **checkpoint**。 `save_strategy: steps` 加上较小的 `save_steps`。因抢占而丢失的最大工作量由该时间间隔决定。
3. 在下一次启动时恢复。 HuggingFace Trainer 的 `.train(resume_from_checkpoint=<path>)` 会自动从 `output_dir` 中取回 `checkpoint-N/`。LlamaFactory、training_hub、mini_trainer 以及其他任何基于 Trainer 的 recipe 都能自动继承这一能力——它们都暴露相同的 `output_dir` / `save_strategy` / `resume_from_checkpoint` 配置项。
4. 优雅退出。 将 `terminationGracePeriodSeconds` 设得足够大，让训练器的信号处理器可以在 SIGKILL 之前刷出最后一个 checkpoint。

`training-runtime.yaml` ↗ 这个 asset 将上述四项打包成一个可运行的 `TrainingRuntime`。trainer-script 的核心如下：

```yaml
spec:
  template:
    spec:
      replicatedJobs:
        - name: node
          template:
            spec:
              template:
                spec:
                  terminationGracePeriodSeconds: 60   # let final checkpo
int flush
                  volumes:
                    - name: ckpt
                      persistentVolumeClaim: { claimName: c12-ckpt }
                  containers:
                    - name: node
                      env:
                        - { name: CKPT_DIR, value: /mnt/ckpt/run }
                      volumeMounts:
                        - { mountPath: /mnt/ckpt, name: ckpt }
                      command: [bash, -ec]
                      args:
                        - |
                          python - <<'PY'
                          import os, glob
                          from transformers import Trainer, TrainingArgum
ents
                          ckpt_dir = os.environ["CKPT_DIR"]
                          # Auto-detect latest checkpoint so the first ru
n starts clean
                          # and every subsequent (re-admitted) run resume
s from it.
                          ckpts = sorted(glob.glob(f"{ckpt_dir}/checkpoin
t-*"),
                                         key=lambda p: int(p.rsplit("-",
1)[1]))
                          resume = ckpts[-1] if ckpts else None
                          args = TrainingArguments(
                              output_dir=ckpt_dir,
                              save_strategy="steps", save_steps=4, save_t
otal_limit=2,
                              # ... rest of your training args
                          )
```

```
                    trainer = Trainer(model=..., args=args, train_d
ataset=...)

                    trainer.train(resume_from_checkpoint=resume)
                    PY
```

同样的结构也适用于 LlamaFactory（在 `lf-sft.yaml` 中设置 `resume_from_checkpoint: true`），以及任何其他基于 Trainer 的 recipe——它们最终都可以归结为“将 `output_dir` 指向 PVC，设置 `save_steps`，并把最新 checkpoint 传给 `.train()`”。

`save_steps` 应根据你可接受的最坏抢占情况来选择：如果每步耗时五秒，`save_steps: 100` 可将损失工作量上限定为约 10 分钟。再配合 `save_total_limit`，避免 PVC 无限制增长。

创建 PVC 并部署 runtime：

```
curl -fsSL $base/checkpoint-pvc.yaml    | sed "s/<your-namespace>/$NS/" |
kubectl apply -f -
curl -fsSL $base/training-runtime.yaml  | sed "s/<your-namespace>/$NS/" |
kubectl apply -f -
```

提交工作负载

一个训练 TrainJob，带有标签，确保以 training 优先级进入 training 队列：

```
curl -fsSL $base/trainjob-low-priority.yaml | sed "s/<your-namespace>/$N
S/" | kubectl create -f -
```

一个参与同一 cohort、且处于 inference 优先级的 InferenceService：

```
curl -fsSL $base/inference-service.yaml | sed "s/<your-namespace>/$NS/" |
kubectl create -f -
```

你应该观察到以下现象：

1. **Training** 先启动 — training Workload 在 `c12-training-cq` 上达到 `Admitted=True`（从 cohort 中的 inference CQ 借用了 GPU 配额）。

2. **Inference** 到达。 它的 Workload 需要一个当前借给 training 的 GPU。Kueue 的经典抢占会将 training Workload 选为目标并驱逐它：

```
status:
  conditions:
    - type: Preempted
      status: "True"
      reason: InCohortReclamation
      message: "Preempted to accommodate a workload ... due to reclamat
ion within the cohort"
    - type: Requeued
      status: "True"
```

3. **Training Pod** 终止。 JobSet 发送 SIGTERM；训练器刷出最后一个 checkpoint 并退出。
4. **Inference** 启动 并且不受阻塞地运行。
5. **Inference** 完成（或缩容）。Kueue 重新准入 training；Trainer v2 重建 JobSet；训练容器在 PVC 上看到 `checkpoint-N/`，并从那里继续。

实时查看这一往返过程：

```
kubectl -n "$NS" get workload -w
kubectl -n "$NS" get trainjob,pods
kubectl -n "$NS" get workload -o jsonpath='{range .items[*]}{.metadata.na
me}: {range .status.conditions[*]}{.type}={.status} {end}{"\n"}{end}'
```

与在线 InferenceService 安全共存

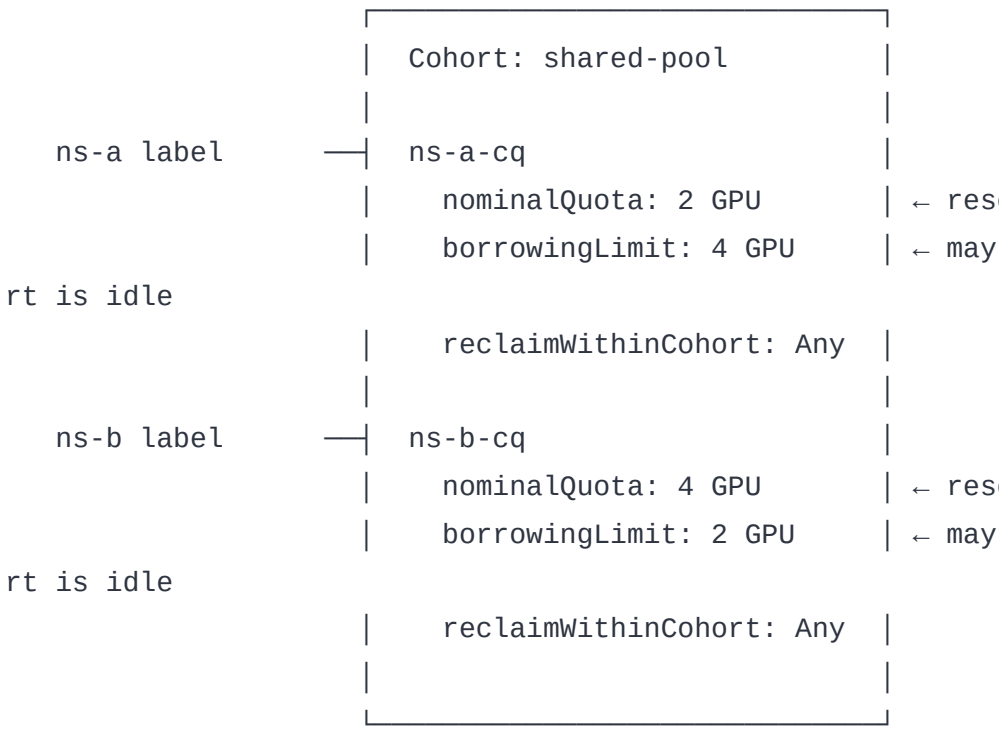
这个双 CQ cohort 是承重部分。再配合几个参数，可以让日常运行更加平稳：

- 按峰值而不是平均值来为 **inference CQ** 设定容量。 如果按平均值来配额，第一次流量尖峰就会侵蚀训练已经开始消耗的容量——每次抢占都会让训练器暂停。应适当增加 `nominalQuota`，确保稳定状态下的 inference 准入不会触碰借用配额。
- 在 **inference** 资源上保持 **`borrowingLimit: 0`**。 `borrowingLimit` 是借用方侧的限制：这样可以防止 inference 工作负载消耗另一个 CQ 的名义配额。它不会阻止 training 借用 inference 的空闲名义配额；如果你需要限制某个 CQ 向 cohort 借出多少资源，请使用 Kueue 的 `lendingLimit`。

- 在 **inference CQ** 上使用 `reclaimWithinCohort: Any`，不要使用 `LowerPriority`。 使用 `LowerPriority` 时，只有严格低于 inference 优先级类的工作负载才会被抢占；`Any` 则允许 inference 无论 training 侧如何配置优先级，都能抢回资源。
- 在 **Kueue** 配置中为 **training** 设置 `PodsReady` 超时↗。 如果一个被抢占后重新准入的 training Pod 遇到镜像拉取很慢的情况，你不希望它永远占着借来的配额；超时后它会回到队列，并让其他工作负载继续前进。
- 为你发布的每个 **InferenceService** 都设置 `WorkloadPriorityClass`，而不只是 **cohort** 中的那些。 如果缺少该标签，Workload 的优先级会停留在 0，抢占规则就无法提升它。
- 不要在 **Kueue** 配置中设置 `manageJobsWithoutQueueName: true`。 一旦开启，所有受限 namespace 中的 Pod/Deployment 都必须带有队列标签，这对集群组件来说非常容易踩坑。
- 让 **inference predictor** 的资源请求保持为单个工作负载。 如果某个 InferenceService 申请的资源超过了 cohort 的名义 inference 配额，再多的抢占也无法满足它。应改为拆分成多个副本。

保留并共享：用于 namespace 级保留的对称 cohort

上面的双 CQ 布局是有意设计成 非对称 的——inference 拥有一切，training 负责借用。使用同样原语的另一种形态，可以让每个租户都 保留一个下限，同时在邻居空闲时 借用 **cohort** 中剩余的资源：



此时每个 ClusterQueue 看起来如下——注意 `nominalQuota > 0` 且 `borrowingLimit > 0`，并且设置 `reclaimWithinCohort: Any`，这样即使邻居借走了资源，所有者也能把自己的保留配额拿回来：

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata: { name: ns-a-cq }
spec:
  cohortName: shared-pool
  namespaceSelector:
    matchLabels: { kueue.x-k8s.io/queue: ns-a }
  resourceGroups:
    - coveredResources: ["nvidia.com/gpualloc"]
      flavors:
        - name: c12-default
          resources:
            - name: nvidia.com/gpualloc
              nominalQuota: 2
              borrowingLimit: 4
  preemption:
    reclaimWithinCohort: Any
    withinClusterQueue: LowerPriority
```

其行为如下：

- 两个 **namespace** 都空闲。 Cohort 持有 6 个 GPU 的名义容量，但都未被使用。
- 只有 **ns-a** 有待处理工作。 ns-a 最多可准入 6 个 GPU（自身 2 个名义配额 + 从 ns-b 的空闲名义配额中借来的 4 个）。
- 随后 **ns-b** 也有待处理工作。 当前有 4 个 GPU 的保留配额被借给了 ns-a。ns-b 的 Workload 触发 `InCohortReclamation`；Kueue 驱逐 ns-a 的 Workload，直到 ns-b 能以其保留配额准入。ns-a 的前 2 个 GPU（其自身名义配额）不会被触碰。
- 两个 **namespace** 都满载。 每个都只会准入到自己的 `nominalQuota`。由于没有空闲配额可借，因此不会发生借用。

实际操作中的调节项

- **`nominalQuota`** 的总和必须 ≤ 物理容量。 保留配额是保证。如果 cohort 的名义总量超过物理 GPU，那么两个 namespace 可能同时达到各自的保留值，而其中一个会在等待 device plugin，而不是等待 Kueue。
- 根据你希望获得的弹性来设置 **`borrowingLimit`**。 `borrowingLimit + nominalQuota` 是单个 CQ 可准入足迹的上限。如果你想要最大的突发能力，就把它设为整个 cohort 减去你的保留值；如果你想为后到的邻居留出余量，则设小一些。
- 借用的工作负载是可抢占的——要对它做 **checkpoint**。 任何准入量高于 `nominalQuota` 的工作负载都运行在借用配额上，所有者一旦收回配额就可能被驱逐。 让 TrainJob 具备抢占安全性 一节中的形态可直接复用：共享 PVC、频繁的 `save_steps`、`terminationGracePeriodSeconds: 60`。否则，每次回收都会丢失已经消耗的墙钟时间。
- 使用 **`borrowWithinCohort`** 来控制准入时的抢占。 当 `borrowWithinCohort.policy: LowerPriority` 时，借用准入可以在借出方侧抢占严格更低优先级的工作负载。若不启用，则借用只会发生在真正空闲的配额上——行为更安静，但高优先级作业如果遇到忙碌的邻居 CQ，仍需等待自然空闲容量。
- 不要轻易在同一个 **cohort** 中混合非对称和对称模式。 带有 `borrowingLimit: 0` 的 CQ（上面的 inference 模式）仍然可以借出空闲名义配额，但它不会从 cohort 中借回配额。在对称模式中，每个 CQ 都会同时借入和借出自己的名义配额。把这两种形态混在一个 cohort 中是可行的，但心智模型会更复杂；如果你两者都需要，最好使用两个 cohort。

什么时候选择哪种布局

目标	布局
保护在线 inference；让 training 按机会运行	非对称 — inference 保留配额，training 借用（即上面的原始方案）
为每个租户提供保底；允许其突发使用共享容量	对称 — 每个 CQ 都有 `nominalQuota` > 0 且 `borrowingLimit` > 0
一个 namespace 中存在不同 SLO 的混合作业	一个 CQ + 多个 `WorkloadPriorityClass` 值 + `withinClusterQueue: LowerPriority` — 不需要 cohort

验证配置

被抢占的 Workload 上的 condition payload 是你的事实依据：

```
kubectl -n "$NS" get workload \
  -o jsonpath='{range .items[?(@.status.conditions[?(@.type=="Preempte
d")].status=="True")]}{.metadata.name}{"\n"}{end}'
```

一个至少被抢占过一次的 training Workload 会显示 `reason: InCohortReclamation`。它的替代对象（在 inference 完成后）将是一个新的 Workload，具有相同的 JobSet 血缘关系，但 UID 不同——Trainer v2 会根据 TrainJob 确定性地命名它们，因此 TrainJob 名称在重启之间保持稳定。

如果你想在 HAMI 集群上对整条流程进行可重复的端到端覆盖，可以使用仓库 `e2e/` harness 中的 `c12_kueue_preemption.sh` ↗ 用例：它会建立 cohort、提交 TrainJob、触发高优先级抢占者，并对 `InCohortReclamation` condition 以及 checkpoint 恢复进行断言。

> **NOTE**
>
> 抢占是有状态的——它会与 SIGTERM 到达时训练器正在执行的操作相互作用。在生产环境中依赖该机制之前，请务必先使用代表性的 TrainingRuntime + 数据集至少完整运行一次抢占-恢复循环。该机制本身是稳健的；最坏的情况只是在最后一个 checkpoint 和 SIGTERM 之间产生少量重复工作。

有关完整的 Kueue 配置，请参见 Kueue 文档；有关底层算法，请参见 Preemption 概念页 ↗。

使用 Dynamic Resource Allocation (DRA) 进行 GPU 切片

传统的 device-plugin 模型会分配整张 **GPU**：Pod 请求 `nvidia.com/gpu: 1`，即使只是一个 LoRA 微调、从未使用超过几 GiB 显存，也会得到整块卡。在一张 24 GB 的 A30 上，这相当于作业实际所需内存的最多 20 倍，而这些资源都处于闲置状态。

Dynamic Resource Allocation (DRA) ↗ —— 一个 `resource.k8s.io/v1` API，并已在 **Kubernetes 1.34** 中达到 **GA** —— 允许 Pod 改为请求 GPU 的一个*切片*。借助 **Alauda Build of NVIDIA DRA Driver for GPUs**，这个切片可以是：

- 一个 **MIG** 分区 —— 由硬件隔离的卡片一部分（拥有独立的 SM 和 VRAM，由 GPU 强制隔离），或者
- 一个 **time-sliced / MPS-shared** 的完整 GPU —— 多个 Pod 共享一张卡，但没有显存隔离。

本指南通过 `ResourceClaimTemplate` 请求一个 MIG 切片，然后在其中使用 Kubeflow Trainer v2 运行一次监督微调。所有资源都位于 `assets/dra/` ↗，并且仓库中的 e2e harness 通过 `c15_dra_gpu_slice.sh` ↗ 用例对整条流程进行了端到端验证。

目录

device plugin 与 DRA 的对比

前提条件

一个切片请求中的三个对象

第 1 步 —— 确认 driver 正在发布切片

第 2 步 —— 对切片做冒烟测试

第 3 步 —— 使用 Kubeflow Trainer v2 在切片内进行微调

选择切片大小 / 选择 profile

没有 MIG？通过 time-slicing 共享整张 GPU

将切片与 Kueue 配额结合使用

将同一个 claim 接入其他 orchestrator

将其转变为真实微调

准备 GPU 节点和 DRA driver（管理员）

在节点上准备 MIG profile

故障排查

device plugin 与 DRA 的对比

	传统 **device plugin**	**Dynamic Resource Allocation**
请求形式	`resources.limits."nvidia.com/gpu": 1`	`resources.claims: [{name: gpu-slice}]` + 一个 `ResourceClaim`
粒度	整张 GPU（或像 HAMi 的 `gpualloc` 这样的供应商固定 vGPU）	按 claim 切片：在请求时选择 MIG profile 或共享模式
形状由谁定义	集群范围的 device-plugin 配置 / 节点重新标记 + 重启	用户按工作负载在 `ResourceClaimTemplate` 中定义
隔离	整卡，或供应商特定隔离	使用 MIG 时为硬件隔离；使用 TimeSlicing/MPS 时为时间共享
选择逻辑	节点标签 + 资源名称	针对设备属性（`profile`、`memory`、`productName` 等）的 CEL 表达式

DRA 不会替代调度器或 Kueue —— 它改变的是*设备如何被描述和声明*。配额、优先级和抢占机制仍然有效（参见 将切片与 Kueue 配额结合使用）。

前提条件

要求	详情
Kubernetes 1.34+	已提供 `resource.k8s.io/v1` —— 可通过 `kubectl api-resources --api-group=resource.k8s.io` 检查
Alauda Build of NVIDIA DRA Driver for GPUs	已安装到集群中，并且其 kubelet-plugin 在 GPU 节点上已发布 `ResourceSlice`（参见 准备 GPU 节点和 DRA driver）
支持 MIG 且已创建 profile 的 GPU	对于 MIG 切片：A30 / A100 / H100 / H200，且已启用 **MIG** 模式并创建 **MIG** 实例（参见 在节点上准备 MIG profile）。没有 MIG 的 GPU 仍可使用 time-slicing 路径
Kubeflow Trainer v2	`trainer.kubeflow.org` API group（参见 使用 Kubeflow Trainer v2 进行微调）。对于 KFP、Volcano 或普通 Jobs，DRA 的接入方式完全相同——参见 将同一个 claim 接入其他 orchestrator
`kubectl` 访问权限	能在你的命名空间中管理 `resourceclaimtemplates`，并读取 `deviceclasses` / `resourceslices`（集群作用域）

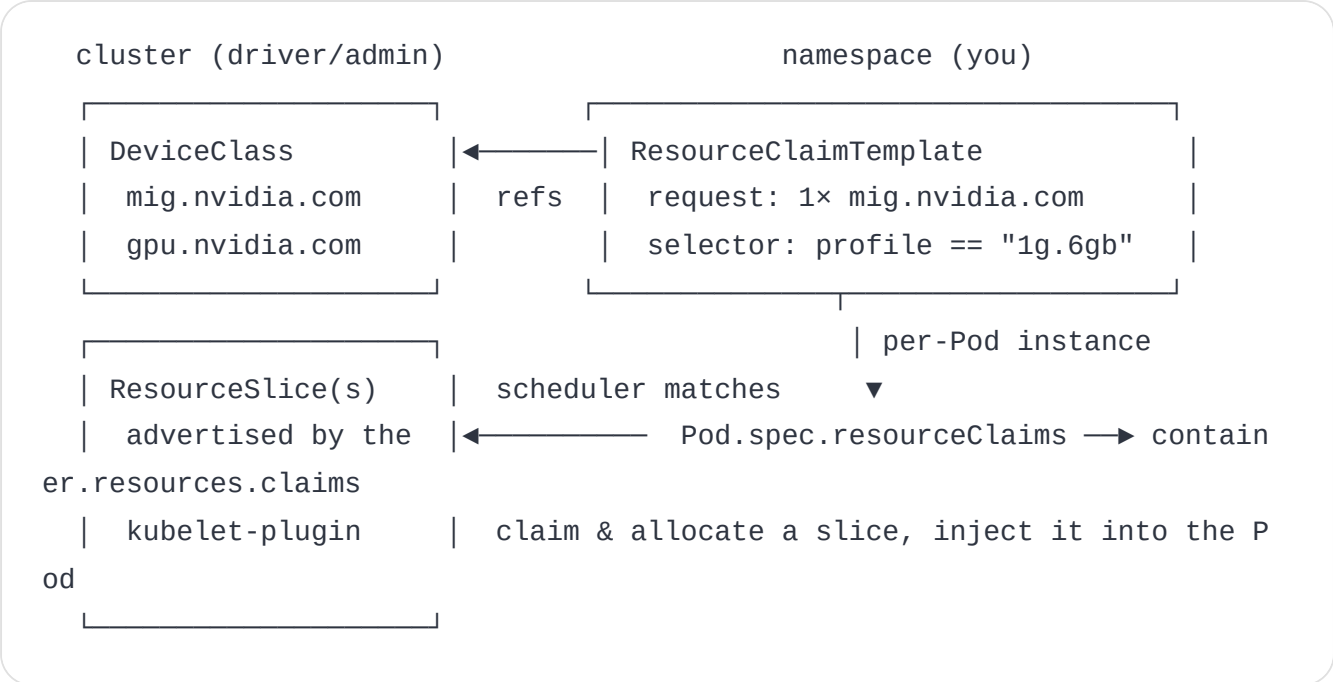
NOTE

`DeviceClass` 和 `ResourceSlice` 是集群作用域资源，由 driver/admin 拥有。

`ResourceClaimTemplate` 和 `ResourceClaim` 是命名空间作用域资源，由你创建。

`ResourceClaimTemplate` 必须与引用它的 Pod 位于同一个命名空间。

一个切片请求中的三个对象



- `DeviceClass`（由 driver 创建）定义一种*设备类型*和一个基础选择器。**Alauda Build of NVIDIA DRA Driver for GPUs** 提供 `gpu.nvidia.com`（完整 GPU）和 `mig.nvidia.com`（MIG 切片）。
- `ResourceClaimTemplate`（由你创建）表示“给每个 Pod 分配一个来自该类、匹配此 CEL 选择器的设备”。调度器会基于它为每个 Pod 生成一个 `ResourceClaim`。
- **Pod** 在 `spec.resourceClaims` 中引用该模板，而 **container** 通过 `resources.claims` 选择使用它。完全不会使用 `nvidia.com/*` 限制。

第 1 步 —— 确认 driver 正在发布切片

只要 driver 安装完成，DeviceClass 就会存在；但只有当 **kubelet-plugin** 为节点的 **GPU** 发布 `ResourceSlice` 后，Pod 才能被调度：

```
kubectl get deviceclasses
# NAME              ...
# gpu.nvidia.com
# mig.nvidia.com

kubectl get resourceslices
# NAME                        NODE              DRIVER           ...
# <node>-gpu.nvidia.com-...   192.168.143.59    gpu.nvidia.com   ...
```

如果 `get resourceslices` 结果为空，说明 driver 已安装，但尚未在任何节点上发布设备——请先跳到 准备 GPU 节点和 DRA driver，然后再继续。

检查已发布的设备，以了解你的 CEL 选择器可以匹配哪些确切的属性键和值——不要假设 **profile** 字符串；请从你的集群中读取它们：

```
kubectl get resourceslices -o yaml | grep -A20 'attributes:'
# ...
#   attributes:
#     gpu.nvidia.com/profile: {string: "1g.6gb"}
#     gpu.nvidia.com/productName: {string: "NVIDIA A30"}
#     gpu.nvidia.com/type: {string: "mig"}
```

第 2 步 —— 对切片做冒烟测试

在把 DRA 接入训练作业之前，先使用 `dra-smoke-pod.yaml` ↗ 证明 driver 确实会分配一个切片。它会请求一个 MIG 切片并打印 container 看到的内容：

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/train/guides/assets/dra
NS=my-namespace   # namespace where your Pods run

# 1. A ResourceClaimTemplate for one 1g.6gb MIG slice.
curl -fsSL $base/mig-slice-resourceclaimtemplate.yaml | sed "s/kubeflow-admin-cpaas-io/$NS/" | kubectl apply -f -

# 2. A Pod that claims it and runs nvidia-smi + a matmul on the slice.
curl -fsSL $base/dra-smoke-pod.yaml | sed "s/kubeflow-admin-cpaas-io/$NS/" | kubectl create -f -

# 3. Watch it, then read the logs.
kubectl -n "$NS" get pods -l app=dra-smoke -w
kubectl -n "$NS" logs -l app=dra-smoke --tail=40
```

你应该能看到该切片降低后的显存上限——对于 `1g.6gb`，大约是 6 GiB，而不是整张卡的 24 GiB——这正是切片的意义所在：

```
===== nvidia-smi -L =====
GPU 0: NVIDIA A30 (UUID: GPU-...)
  MIG 1g.6gb  Device  0: (UUID: MIG-...)
cuda_available: True
device: NVIDIA A30 MIG 1g.6gb
slice_total_mem_GiB: 5.75
matmul_on_slice: OK
```

从 control plane 检查分配情况：

```
kubectl -n "$NS" get resourceclaims
# NAME                 STATE                ...
# dra-smoke-...-gpu-slice   allocated,reserved
```

第 3 步 —— 使用 Kubeflow Trainer v2 在切片内进行微调

`dra-sft-trainingruntime.yaml` ↗ 中的 `TrainingRuntime` 是一个普通的 Trainer v2 runtime，恰好包含两个 **DRA** 钩子，并且没有 `nvidia.com/*` 限制：

```
spec:
  template:
    spec:
      replicatedJobs:
        - name: node
          template:
            spec:
              template:
                spec:
                  resourceClaims:                         # (1) Pod-leve
l: bind a per-Pod claim
                    - name: gpu-slice
                      resourceClaimTemplateName: mig-1g-6gb
                  containers:
                    - name: node
                      resources:
                        claims:                           # (2) Container
-level: consume it
                          - name: gpu-slice
                        # NOTE: no nvidia.com/gpu here — the GPU comes fr
om the claim
```

它的训练脚本是自包含的——会构建一个小型 causal-LM，为其包装一个 LoRA adapter，并在合成数据集上训练，因此这次运行不需要下载模型或数据集，并且大约一分钟即可完成。这使它非常适合在离线 GPU 节点上运行，也非常适合验证切片。你可以替换为真实的基础模型和数据集，将其变成生产级微调（参见 将其转变为真实微调）；DRA 相关的接入方式不变。

应用 runtime 并提交作业：

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/train/guides/assets/dra
NS=my-namespace

# ResourceClaimTemplate (skip if you already applied it in Step 2).
curl -fsSL $base/mig-slice-resourceclaimtemplate.yaml | sed "s/kubeflow-admin-cpaas-io/$NS/" | kubectl apply -f -
# TrainingRuntime that consumes the slice.
curl -fsSL $base/dra-sft-trainingruntime.yaml         | sed "s/kubeflow-admin-cpaas-io/$NS/" | kubectl apply -f -
# TrainJob.
curl -fsSL $base/dra-sft-trainjob.yaml                | sed "s/kubeflow-admin-cpaas-io/$NS/" | kubectl create -f -
```

跟踪它直到完成：

```
kubectl -n "$NS" get trainjob,pods
pod=$(kubectl -n "$NS" get pods -l jobset.sigs.k8s.io/replicatedjob-name=node -o jsonpath='{.items[0].metadata.name}')
kubectl -n "$NS" logs -f "$pod"
```

成功时如下所示：

```
[slice] device=NVIDIA A30 MIG 1g.6gb total_mem=5.75 GiB
trainable params: 147,456 || all params: 4,...  || trainable%: ...
{'loss': ..., 'step': 10}
[slice] peak_mem=1.83 GiB
DRA LoRA SFT on GPU slice: OK
```

`peak_mem` 明显低于切片的 `total_mem`，这证明微调完全运行在其分区内。再启动第二个 `TrainJob`，并让第一个继续运行：如果一张卡被划分为四个 `1g.6gb` 切片，则最多可以在一张物理 GPU 上并发且隔离运行四个微调——每个作业都看不到其他作业的显存。

选择切片大小 / 选择 profile

修改 `mig-slice-resourceclaimtemplate.yaml` ↗ 中 CEL 选择器里的 MIG profile：

```
selectors:
  - cel:
      expression: device.attributes["gpu.nvidia.com"].profile == "2g.12g
b"
```

常见 profile（请根据你的 `ResourceSlice` 确认确切字符串——几何布局因 GPU 而异）：

GPU	Profiles	每卡最多切片数
A30 (24 GB)	`1g.6gb`, `2g.12gb`, `4g.24gb`	4
A100 / H100 (80 GB)	`1g.10gb`, `2g.20gb`, `3g.40gb`, `7g.80gb`	7

选择能够轻松容纳你的模型 + LoRA + optimizer state + activations 的最小 profile。0.5–1.5B 模型的 LoRA 适合 `1g.6gb`；7B QLoRA 通常需要 `2g.12gb` 及以上。

没有 **MIG**？通过 **time-slicing** 共享整张 **GPU**

如果 GPU 不支持 MIG，或者你不想对其分区，可以通过一个不透明的 `GpuConfig` 使用 `shared-gpu-resourceclaimtemplate.yaml` ↗ 请求一个通过 **time-slicing** 共享的完整 **GPU**：

```
spec:
  spec:
    devices:
      requests:
        - name: gpu-slice
          exactly:
            deviceClassName: gpu.nvidia.com
            allocationMode: ExactCount
            count: 1
      config:
        - requests: ["gpu-slice"]
          opaque:
            driver: gpu.nvidia.com
            parameters:
              apiVersion: resource.nvidia.com/v1beta1
              kind: GpuConfig
              sharing:
                strategy: TimeSlicing   # or MPS for concurrent sharing
```

将 runtime 指向这个模板即可（`resourceClaimTemplateName: shared-gpu-timeslice`）——其余部分无需更改。e2e 用例会使用 `DRA_SLICE_MODE=shared` 来运行这条路径。

	MIG 切片	**Time-slicing / MPS**
显存隔离	有 —— 硬件强制隔离	无 —— 每个使用者都能看到完整 VRAM
故障隔离	有	无 —— 一个作业可能导致其他作业 OOM
需要 MIG 模式	是	否
最适合	有保证、隔离的训练切片	将大量轻量 / 突发型作业塞进一张卡

将切片与 Kueue 配额结合使用

DRA 关注的是设备的*形态*；Kueue 关注的是*谁可以使用多少*。Kueue 能理解 DRA `DeviceClass`，因此 `ClusterQueue` 可以像配额 `nvidia.com/gpu` 一样，为 `mig.nvidia.com` 设备设置配额——按切片预算接纳 `TrainJob`，并在更高优先级作业需要切片时抢占借用者。`Preemptible TrainJobs with Kueue` 指南中的 cohort 模式可原样使

用；只需将 `ResourceFlavor` 中受管的资源替换为 DRA device class 即可。（Kueue 中的 DRA 支持仍在演进——在生产环境依赖之前，请先根据你的 Kueue 版本进行验证。）

将同一个 claim 接入其他 orchestrator

DRA 钩子位于 **Pod spec** 中，因此任何能够构造 Pod 的系统都可以请求一个切片——Trainer v2 只是其中一个调用者：

- **Kubeflow Pipelines (KFP v2)**：通过 `kfp-kubernetes` ↗ 的平台配置（或 pod-spec patch），为某个 task 的 Pod 附加 `resourceClaims` / `resources.claims`，让 pipeline component 运行在切片上。
- **Volcano / 普通 `Job` / `Pod`**：这两个钩子与 冒烟测试 Pod 在字节级别上完全一致——设置 `spec.resourceClaims` 和 container 的 `resources.claims` 即可。

冒烟测试 Pod ↗ 就是这些场景的最小模板。

将其转变为真实微调

随附脚本训练的是一个合成模型，因此可以离线验证切片。若要在切片上微调真实模型，请保留这两个 DRA 钩子，并将 container 替换为真实 recipe——例如 使用 Kubeflow Trainer v2 进行微调 中的 LlamaFactory runtime，其中 `dataset-initializer` / `model-initializer` 步骤会将基础模型和数据集下载到共享 PVC 上。该 runtime 唯一需要修改的地方，就是删除 `nvidia.com/gpualloc` / `gpucores` / `gpumem` 限制，并添加 `spec.resourceClaims` + `resources.claims`。根据模型大小来配置 MIG profile（0.6B LoRA 适合 `1g.6gb`；7B QLoRA 需要 `2g.12gb` 及以上）。

准备 GPU 节点和 DRA driver（管理员）

Alauda Build of NVIDIA DRA Driver for GPUs 安装了一个 controller，以及一个受节点标签控制的 kubelet-plugin `DaemonSet`。只要 GPU 节点还没有带上该标签，plugin 就不会发布任何 `ResourceSlice`，而 DRA Pod 也会一直处于 `Pending` 状态。

```
# The kubelet-plugin DaemonSet is scheduled by a node label (check its no
deSelector):
kubectl -n kube-system get ds nvidia-dra-driver-gpu-kubelet-plugin \
  -o jsonpath='{.spec.template.spec.nodeSelector}{"\n"}'
# {"nvidia-device-enable":"pgpu-dra"}

# Label the GPU node so the plugin lands and starts advertising devices:
kubectl label node <gpu-node> nvidia-device-enable=pgpu-dra
```

对于完整 **GPU / time-sliced** 方式的 claim，这就是全部需要做的事情——此时 plugin 会为每块卡发布一个 `gpu.nvidia.com` 设备，你可以直接跳到 对切片做冒烟测试。MIG 切片则需要下面的额外准备。

在节点上准备 MIG profile

在 **driver** 能够分配之前，卡上必须先存在 **MIG** 切片。 Alauda Build of NVIDIA DRA Driver for GPUs 不会按需创建 MIG 分区——它只会暴露已经存在的 MIG 设备。即使 MIG 模式已启用但尚未创建任何实例，kubelet-plugin 也会进入 crash-loop，并报 `invalid CDI Spec: no devices`。因此，管理员需要在每个 **GPU** 节点上只创建一次 MIG profile（所有 `nvidia-smi` 命令都在节点宿主机上执行——可以通过特权 Pod 或 SSH）：

```
# 0. The GPU must be IDLE. MIG-mode changes reset GPU state and are refus
ed while any
#    CUDA workload or monitoring client (e.g. dcgm-exporter) holds the ca
rd. Stop GPU
#    workloads and move dcgm-exporter off this node first.

# 1. Enable MIG mode on the GPU (index 0 here) and confirm it took effec
t.
nvidia-smi -i 0 -mig 1
nvidia-smi -i 0 --query-gpu=mig.mode.current,mig.mode.pending --format=cs
v
#    If mode stays "pending", a client is still attached — free it, then
finalise with
#    a GPU reset (`nvidia-smi -i 0 -r`) or a node reboot.

# 2. List the GPU-instance profiles this card supports, with their IDs an
d how many fit.
nvidia-smi mig -lgip
#   GPU  Name          ID    Free/Total   Memory
#     0  MIG 1g.6gb    19     4/4         ~6 GiB     <- four small slices
#     0  MIG 2g.12gb   14     2/2         ~12 GiB
#     0  MIG 4g.24gb    0     1/1         ~24 GiB    <- whole card as a si
ngle MIG device

# 3. CREATE the GPU instances and their compute instances (-C). Repeat a
profile ID to
#    make several slices of it. This carves the card into FOUR 1g slices
(ID 19 x4):
nvidia-smi mig -cgi 19,19,19,19 -C
#    ...or two medium slices:  nvidia-smi mig -cgi 14,14 -C

# 4. Confirm the MIG devices now exist.
nvidia-smi -L
nvidia-smi mig -lgi          # list the created GPU instances
```

WARNING

profile 的**ID**、名称以及每个实例的显存都与硬件相关，而且 `nvidia-smi mig -lgip` 显示的名称可能与 driver 发布的 `profile` 属性不同——例如，A30 的一个 GPU-instance profile 在列表中显示为 `2g.6gb`，但在 DRA 中会暴露为 `1g.6gb`。不要直接照抄这个示例。 请先在第 2 步读取你卡

的真实 profile，然后（在下面）读取发布出来的 `profile` 字符串，并让你的 `ResourceClaimTemplate` 的 CEL 选择器与之匹配。

使用 `nvidia-smi mig -cgi` 创建的 MIG 实例并不总是能在 GPU 重置或节点重启后保留下来。请在重启后重新创建，或者通过 driver 的 MIG-config 工具 / 启动时 unit 自动化创建流程。

最后，重启 kubelet-plugin Pod，使其重新枚举该卡并发布 MIG `ResourceSlice`，然后读取 CEL 选择器必须匹配的确切 `profile` 字符串：

```
kubectl -n kube-system delete pod -l app.kubernetes.io/name=nvidia-dra-dr
iver-gpu --field-selector spec.nodeName=<gpu-node>

kubectl get resourceslices -o json \
  | jq -r '.items[].spec.devices[].attributes."gpu.nvidia.com".profile.st
ring' | sort -u
# 1g.6gb          <- use this exact string in mig-slice-resourceclaimtemp
late.yaml
```

WARNING

一块物理 **GPU** 只能对应一个 **device plugin**。 NVIDIA DRA kubelet-plugin 和传统 device plugin（例如 HAMi 的 `nvidia.com/gpualloc`，或默认的 `nvidia.com/gpu` plugin）不能同时管理同一张卡——否则会造成重复分配。请为 DRA 专门保留一台节点（或特定 GPU），并确保其他 plugin 不会选择它。在另一 plugin 正在提供服务的卡上启用 MIG 模式，会干扰该 plugin 的工作负载。

故障排查

症状	原因 / 修复
Pod 卡在 `Pending`，事件为 `cannot allocate all claims`	没有 `ResourceSlice` 匹配该选择器。执行 `kubectl get resourceslices -o yaml`，检查 profile / 属性字符串；确认节点已打标签且 kubelet-plugin Pod 处于 `Running` 状态。

症状	原因 / 修复
`ResourceClaim` 一直保持 `pending`（从未变为 `allocated`）	GPU 没有处于 MIG 模式（针对 `mig.nvidia.com`），或者该 profile 的切片已经全部被预留。
kubelet-plugin `CrashLoopBackOff`，日志为 `invalid CDI Spec: no devices`	MIG 模式已开启，但没有创建 MIG 实例。请在节点上执行 `nvidia-smi mig -cgi <ids> -C`（见管理员部分），然后重启 plugin Pod。
`nvidia-smi -i 0 -mig 1` 显示 `pending`，模式始终无法变成 `Enabled`	有其他进程占用了 GPU（CUDA workload，或 `dcgm-exporter`）。请停止它们 / 将监控 DaemonSet 移出该节点，然后重试；执行 GPU 重置（`nvidia-smi -i 0 -r`）或重启节点可最终完成卡住的切换。
`TrainingRuntime` 被拒绝：`numProcPerNode ... no such overload`	`numProcPerNode` 必须是 `auto`、`cpu`、`gpu`，或者一个不加引号的整数——请使用 `numProcPerNode: 1`，而不是 `"1"`。
`must be no more than ...` / 应用时出现 schema 错误	API 版本不匹配——本指南面向 `resource.k8s.io/v1`（K8s 1.34 GA）。在 1.31–1.33 中，该 group 使用 `v1beta1` / `v1beta2`，其 `requests` 结构略有不同。
两个 GPU 资源相互争用 / 随机 CUDA 错误	传统 device plugin 和 DRA plugin 都在管理这张卡。请参见上面的警告。
Trainer container 看到的是完整的 24 GiB，而不是切片	你应用的是 `shared-gpu-timeslice`（完整 GPU）模板，或者 MIG 模式未开启。要获得真正的显存隔离切片，请在启用了 MIG 模式的 GPU 上使用 MIG 模板。

为了对整个流程进行可重复覆盖，`c15_dra_gpu_slice.sh` 用例会应用模板、runtime 和 TrainJob，并断言微调能够在切片内完成。如果集群中不存在 `gpu.nvidia.com` `ResourceSlice`，它会自动跳过，因此在仅使用 device-plugin 的集群上也会保持通过。

Training Runtime 镜像

适用于 Kubeflow Trainer v2 的 Curated `TrainingRuntime` 镜像。每个镜像都打包了特定的 PyTorch + accelerator 栈，因此用户无需重新构建即可提交 `TrainJob`。

目录

可用运行时
选择运行时
应用 TrainingRuntime
提交 TrainJob
设备资源请求
NVIDIA GPU
Huawei Ascend NPU
镜像注意事项
自行构建

可用运行时

镜像	设备	框架	拉取
torch (CUDA)	NVIDIA GPU (CUDA 12.6)	PyTorch 2.6, transformers, accelerate, datasets, mlflow	`alaudadockerhub`

镜像	设备	框架	拉取
torch (CANN)	Huawei Ascend NPU (CANN 8.5)	PyTorch 2.6 + torch_npu 2.6.0.post5	`alaudadockerhub`
LLaMA-Factory (CUDA)	NVIDIA GPU (CUDA 12.6)	LLaMA-Factory 0.9.4 (`metrics,awq,modelscope`)	`alaudadockerhub`
LLaMA-Factory (CANN)	Huawei Ascend NPU (CANN 8.5)	LLaMA-Factory 0.9.4 (`metrics,modelscope`)	`alaudadockerhub`
TrainingHub (CUDA)	NVIDIA GPU (CUDA 12.6)	trl, peft, bitsandbytes, deepspeed (SFT/OSFT/DPO)	`alaudadockerhub`
MindSpeed-LLM (CANN)	Huawei Ascend NPU (CANN 8.5)	MindSpeed + MindSpeed-LLM (Megatron core 0.8.0)	`alaudadockerhub`
Fine-tune (LlamaFactory CUDA)	NVIDIA GPU (CUDA 12.6)	LLaMA-Factory 0.9.4 + `git-lfs`, MLflow, MySQL/Postgres clients — runtime for `fine-tune-with-trainer-v2.ipynb` ↗	`alaudadockerhub`
Fine-tune (LlamaFactory CANN)	Huawei Ascend NPU (CANN 8.5)	LLaMA-Factory 0.9.4 + `torch_npu` 2.6 + `git-lfs` — NPU counterpart of the LlamaFactory fine-tune image	`alaudadockerhub` `cann_8.5.0-torch`
Workbench PyTorch CANN	Huawei Ascend NPU (CANN 8.5)	Jupyter + PyTorch 2.9 + `torch_npu` 2.9 + MindSpeed-LLM — 用于 Ascend NPU fine-tune / pretrain notebooks 和 `fine-tune-with-trainer-v2-mindspeed-npu.ipynb` ↗	`alaudadockerhub` `py312-ubi9:v0.1`
Workbench MindSpore CANN	Huawei Ascend NPU (CANN 8.5)	Jupyter + MindSpore 2.8 + `msadapter` + bundled MindSpeed-Core-MS (MindSpeed + MindSpeed-LLM + Megatron-LM + MSAdapter source tree at `/opt/app-root/share/MindSpeed-Core-MS/`) — used by `qwen3_0.6b_finetune_verify.ipynb`	`alaudadockerhub` `cann-py312-ubi9`

镜像	设备	框架	拉取
torch-distributed	NVIDIA GPU (CUDA 12.6)	PyTorch 2.9.1 + torchvision distributed-training base — `ClusterTrainingRuntime` referenced by Kubeflow Trainer Quick Start	`alaudadockerhub`

CUDA 镜像仅支持 amd64；CANN 镜像仅支持 arm64。

```
docker pull alaudadockerhub/torch2.6-cu126-amd64:v0.1.0
```

选择运行时

- 在 **GPU** 上使用 **torchrun** → `torch2.6-cu126-amd64`
- 在 **NPU** 上使用 **torchrun** → `torch2.6-cann8.5-arm64`（设置 `runtimeClassName: ascend`）
- 使用 **LLaMA-Factory** 进行 **LLM SFT / LoRA** → `llamafactory0.9-cu126-amd64`（GPU）或 `llamafactory0.9-cann8.5-arm64`（NPU）
- **TRL / PEFT SFT / OSFT / DPO** → `traininghub0.1-cu126-amd64`
- 在 **Ascend** 上进行 **Megatron** 风格训练 → `mindspeed-llm-cann8.5-arm64`

应用 TrainingRuntime

可直接应用的 YAML 位于 `assets/training-runtimes/`↗。每个 YAML 都固定使用 `:v0.1.0`；如需跟踪其他发布版本，请覆盖该标签。YAML 默认使用 Kubeflow `Profile` 命名空间 — 请将 `metadata.namespace` 修改为你提交作业的命名空间。

```
base=https://raw.githubusercontent.com/alauda/aml-docs/master/docs/en/train/guides/assets/training-runtimes
# NVIDIA GPU
kubectl apply -f $base/torch2.6-cu126-amd64-trainingruntime.yaml
kubectl apply -f $base/llamafactory0.9-cu126-amd64-trainingruntime.yaml
kubectl apply -f $base/traininghub0.1-cu126-amd64-trainingruntime.yaml
# Huawei Ascend NPU
kubectl apply -f $base/torch2.6-cann8.5-arm64-trainingruntime.yaml
kubectl apply -f $base/llamafactory0.9-cann8.5-arm64-trainingruntime.yaml
kubectl apply -f $base/mindspeed-llm-cann8.5-arm64-trainingruntime.yaml
```

提交 TrainJob

共享 smoke 模板适用于任意运行时 — 将 `spec.runtimeRef.name` 设置为你要验证的运行时：

```
kubectl apply -f $base/trainjob-smoke.yaml
kubectl -n <your-namespace> get trainjobs
trainjob=$(kubectl -n <your-namespace> get trainjobs -o name | tail -1)
kubectl -n <your-namespace> logs -f -l jobset.sigs.k8s.io/jobset-name=${trainjob##*/}-node
```

设备资源请求

NVIDIA GPU

整卡请求：

```
resources:
  limits:
    nvidia.com/gpu: 1
```

HAMI vGPU 切片：

```yaml
resources:
  limits:
    nvidia.com/gpualloc: 1    # virtual GPU slot
    nvidia.com/gpucores: 50   # 50% of one physical GPU
    nvidia.com/gpumem: "8192" # 8 GiB
```

Huawei Ascend NPU

务必设置 `runtimeClassName: ascend`，以便注入宿主机驱动库和 DCMI sockets。

标准 Huawei device-plugin：

```yaml
spec:
  runtimeClassName: ascend
  containers:
    - resources:
        limits:
          huawei.com/Ascend910: "1"
```

HAMI vNPU（每个 910B4 切分为 20 cores / 32 GiB）：

```yaml
spec:
  schedulerName: hami-scheduler
  runtimeClassName: ascend
  containers:
    - resources:
        limits:
          huawei.com/Ascend910B4: "1"
          huawei.com/Ascend910B4-memory: "8192"
```

使用 HAMI 时，`allocatable.huawei.com/Ascend910B4` 显示为 `0`，因为 HAMI 通过其 scheduler extender 进行分配。如果 Pod 一直处于 `Pending` 且提示 `hami-scheduler: 1 node unregistered`，请确认宿主机驱动已加载（`/sys/bus/pci/drivers/davinci`、`npu-smi info` 正常），并且节点已标记为 `ascend=on`。

镜像注意事项

- 所有 **CANN** 镜像 — `runtimeClassName: ascend` 会 bind-mount 宿主机的 `/usr/local/Ascend`，但不会导出 CANN 环境变量（`LD_LIBRARY_PATH`、`ASCEND_HOME_PATH` 等）。任何导入 `torch_npu` 的入口程序都必须先 `source /usr/local/Ascend/ascend-toolkit/set_env.sh`（以及可选的 `/usr/local/Ascend/nnal/atb/set_env.sh`）；否则导入会因 `libhccl.so: cannot open shared object file` 而失败。已发布的运行时 YAML 已经包含这些操作 — 请在任何派生运行时中保留这些 `source` 行。
- **`traininghub0.1-cu126-amd64`** — 提供 CUDA *runtime*，但不包含 toolkit。DeepSpeed JIT op 编译需要 `nvcc`；如果使用这些 ops，请挂载或安装 `nvidia-cuda-toolkit`，并设置 `CUDA_HOME`。
- **`mindspeed-llm-cann8.5-arm64`** — `megatron.core` 需要 `pkg_resources`，因此请在作业入口程序中安装 `setuptools<81`。目前 `import mindspeed_llm` 会因为 `mindspeed_llm` master / `core_r0.8.0` 不匹配而失败；在没有该 adapter shim 的情况下，底层的 torch + torch_npu + megatron.core + mindspeed 栈可正常训练。

自行构建

Containerfiles、多架构 buildkitd helper、e2e harness，以及 post-fix 扫描证据位于 `kubeflow-plugin/training-runtimes` ↗。每个框架镜像都只是 `torch2.6-cu126-amd64` 或 `torch2.6-cann8.5-arm64` 上的一层薄封装，因此派生一个新运行时基本上就是在 `FROM docker.io/alaudadockerhub/torch2.6-cu126-amd64:v0.1.0` 的基础上再安装对应框架。

使用 Training Hub 对 LLM 进行微调

`training_hub` 是一个 Python 库，它将多种 LLM 后训练算法——Supervised Fine-Tuning (SFT)、Orthogonal Subspace Fine-Tuning (OSFT)、LoRA / QLoRA，以及 continued pre-training (CPT)——封装在单个函数调用（`sft(...)`、`osft(...)`、`lora_sft(...)`）之后，以统一方式处理单 GPU、多 GPU 和多节点训练。

- 自动内存管理 — `max_tokens_per_gpu` 会限制 GPU 内存占用，并自动计算 micro-batch size 和 gradient accumulation，以满足你的目标 `effective_batch_size`。
- **OSFT** 实现了 Nayak et al., 2025 (arXiv:2504 .07097) ↗ —— 将权重更新限制在正交子空间中，可在没有 replay data 的情况下防止 catastrophic forgetting。
- **QLoRA** 以 4-bit 方式加载冻结的 base model，并且只训练 LoRA adapters，使大型模型能够适配单个较小的 GPU 切片（Dettmers et al., 2023, arXiv:2305 .14314 ↗）。
- **Continued pre-training (CPT)** 在原始文本语料上执行 next-token prediction，以便在 instruction tuning 之前注入领域知识。
- 内置 checkpointing、experiment tracking 和 Liger kernel 支持。

> **设计背景**
>
> `training_hub` 是 RHOAI / Open Data Hub 用于通过单一 API 暴露后训练 算法的上游库，而 Alauda AI 则在 Kubeflow Trainer v2 （`TrainJob` / `ClusterTrainingRuntime`）上运行同一套代码。*算法*（此库） 与 *distributed runtime*（Kubeflow Trainer）之间的拆分遵循 Open Data Hub 的 architecture decision records ↗ （参见 `distributed-workload` 和 `workbenches` 组件文档）——因此，这里的示例 在 workbench notebook 中运行，或作为集群 `TrainJob` 运行时，都会平滑映射到 相同的 `sft` / `osft` / `lora_sft` entrypoint。

方面	SFT	OSFT	QLoRA	CPT
用例	初始 instruction tuning	已微调模型的持续领域适配	节省内存的适配	注入领域
数据	Chat JSONL	Chat JSONL	Chat JSONL	原始文本
可训练权重	全部（bf16）	正交子空间	LoRA adapters（base 为 4-bit）	全部（bf1
关键参数	标准 hyperparameters	`unfreeze_rank_ratio`（0.0–1.0）	`load_in_4bit` + `lora_r`	`is_pret` `block_s`
入口函数	`sft(...)`	`osft(...)`	`lora_sft(...)`	`sft(...` `is_pretr`
后端	instructlab-training	mini-trainer	unsloth / peft + bitsandbytes	instructla

目录

要求

数据格式

运行示例 notebooks

关键参数

QLoRA（4-bit LoRA）

持续预训练 (CPT)

多节点

要求

- 你的 cluster 中已安装 Alauda AI Workbench。

- 一个具有互联网访问能力（或内部 PyPI 镜像）、至少一个 NVIDIA GPU，以及用于 checkpoint 的持久化存储的 workbench。
- HuggingFace 模型名称或本地路径。
- JSONL 格式的训练数据（见下文）。

数据格式

每一行都是一个对话：

```
{"messages": [{"role": "system", "content": "You are a helpful assistan
t."}, {"role": "user", "content": "What is machine learning?"}, {"role":
"assistant", "content": "Machine learning is a subset of AI..."}]}
```

角色：`system`、`user`、`assistant`、`pretraining`。Masking：

- **SFT**（默认）——只有 `assistant` 内容会对 loss 产生贡献。为某个 sample 添加 `"unmask": true` 可将所有非 system 内容都纳入训练。
- **OSFT**——由 `unmask_messages` 控制（默认 `False`）。

如果数据集已预先 tokenized，并包含 `input_ids` / `labels`，则可以通过 `use_processed_dataset=True` 支持。

运行示例 **notebooks**

下载到你的 workbench 中，并逐 cell 执行：

Notebook	Algorithm	下载
SFT comprehensive tutorial	SFT	`sft-comprehensive-tutorial.ipynb` ↗
OSFT comprehensive tutorial	OSFT	`osft-comprehensive-tutorial.ipynb` ↗
QLoRA comprehensive tutorial	QLoRA	`qlora-comprehensive-tutorial.ipynb` ↗

Notebook	Algorithm	下载
CPT comprehensive tutorial	CPT	`cpt-comprehensive-tutorial.ipynb` ↗

安装并配置：

```
pip install training-hub
# `training-hub` pulls a stray `attr` package that shadows `attrs.attr` a
nd
# breaks `aiohttp`; uninstall it.
pip uninstall -y attr
```

NOTE

在预构建的 `traininghub0.1-cu126-amd64:v0.1.0` runtime image 上，请在全新的 venv 中安装 training-hub ——因为 `pip install --user training-hub` 会将 `transformers` 升级到与随附 `peft` 不兼容的版本：

```
python -m venv /workspace/venv
source /workspace/venv/bin/activate
pip install training-hub
pip uninstall -y attr
```

编辑参数 cells：

```
model_path = "Qwen/Qwen2.5-7B-Instruct"        # HF name or local path
data_path       = "/path/to/your/training_data.jsonl"
ckpt_output_dir = "/path/to/checkpoints/my_experiment"
selected_distributed = "single_node_8gpu"      # or single_gpu_dev, multi
_node_master, ...
# OSFT only:
unfreeze_rank_ratio = 0.25                      # 0.1–0.3 conservative,
0.3–0.5 balanced
```

内置的模型预设覆盖 Qwen 2.5 7B、Llama 3.1 8B、Phi 4 Mini，以及通用的 7B / 小型模型。

运行所有 cells。最后的训练 cell 会调用：

```
from training_hub import sft, osft

result = sft(model_path=model_path, data_path=data_path, ckpt_output_dir=
ckpt_output_dir,
             effective_batch_size=128, max_tokens_per_gpu=20000, max_seq_
len=16384,
             learning_rate=1e-5, num_epochs=3, nproc_per_node=8)

result = osft(model_path=model_path, data_path=data_path, ckpt_output_dir
=ckpt_output_dir,
              unfreeze_rank_ratio=0.25,
              effective_batch_size=128, max_tokens_per_gpu=10000, max_seq
_len=8192,
              learning_rate=5e-6, num_epochs=1, nproc_per_node=8)
```

checkpoint 会在每个 epoch 写入 `ckpt_output_dir`（由 `checkpoint_at_epoch` 控制）。

关键参数

通用（SFT 和 OSFT）：

参数	必需	描述
`model_path`	是	HF 名称或本地路径
`data_path`	是	JSONL 训练数据
`ckpt_output_dir`	是	checkpoint 目录
`effective_batch_size`	是	全局有效 batch size
`max_tokens_per_gpu`	是	每个 GPU 的 token 预算；自动计算 micro-batch size
`max_seq_len`	是	最大序列长度
`learning_rate`	是	optimizer LR

参数	必需	描述
`num_epochs`	否	默认为 `1`
`lr_scheduler`, `warmup_steps`	否	LR 调度
`use_liger`	否	Liger kernels（OSFT 默认 `True`）
`seed`	否	默认为 `42`
`data_output_dir`	否	处理后的数据缓存；RAM-disk 使用 `"/dev/shm"`
`use_processed_dataset`	否	如果数据已包含 `input_ids` / `labels`，则跳过 tokenization
`checkpoint_at_epoch`, `save_final_checkpoint`	否	默认为 `True`
`nproc_per_node`, `nnodes`, `node_rank`	否	分布式拓扑
`rdzv_id`, `rdzv_endpoint`	否	多节点 rendezvous

仅 OSFT：

参数	必需	描述
`unfreeze_rank_ratio`	是	每个 weight matrix 中可更新的比例（0.0–1.0）。越低表示保留越多。
`unmask_messages`	否	如果为 `True`，则在所有非 system 内容上训练
`target_patterns`	否	用于将 OSFT 限定到特定层的子字符串模式

QLoRA（4-bit LoRA）

QLoRA 以 **4-bit NormalFloat (NF4)** 精度冻结 base model，并且只训练其上的小型 **LoRA adapter** 矩阵。对于完整 SFT 需要约 ~60 GiB 的 7B 模型，它可以在单个 16–24 GiB GPU（或 HAMI vGPU slice）上运行。只要你受限于 GPU 内存，就应当使用它； 代价是会有轻微的量化质量差距，以及稍慢的 step。

`training_hub` 通过 `lora_sft(...)` 提供 QLoRA——即 LoRA 加上 bitsandbytes 的 4-bit 选项：

```
from training_hub import lora_sft

result = lora_sft(
    model_path="Qwen/Qwen2.5-7B-Instruct",   # HF name or local path
    data_path="/path/to/training_data.jsonl", # same chat JSONL as SFT
    ckpt_output_dir="/path/to/checkpoints/qlora_run",
    # LoRA adapter
    lora_r=16, lora_alpha=32, lora_dropout=0.05,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],  # or "all-linear"
    # 4-bit quantization — this is what makes it QLoRA
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype="bfloat16",
    bnb_4bit_use_double_quant=True,
    # core training (LoRA tolerates higher LRs than full SFT)
    num_epochs=3, effective_batch_size=16, learning_rate=2e-4, max_seq_len=4096,
    nproc_per_node=1,
)
```

QLoRA 专用参数：

参数	必需	描述
`load_in_4bit`	是（用于 QLoRA）	以 4-bit 加载冻结的 base model。若不设置，则 `lora_sft` 只是普通（16-bit）LoRA。
`bnb_4bit_quant_type`	否	`"nf4"`（推荐）或 `"fp4"`

参数	必需	描述
`bnb_4bit_compute_dtype`	否	反量化计算 dtype，例如 `"bfloat16"`
`bnb_4bit_use_double_quant`	否	对量化常量进行嵌套量化（额外节省内存）
`lora_r`, `lora_alpha`, `lora_dropout`	是	LoRA adapter 的 rank / scaling / dropout
`target_modules`	否	要适配的 modules（attention projections，或 `"all-linear"`）
`load_in_8bit`	否	8-bit base（LoRA-8bit，而不是 QLoRA）

输出是一个 **LoRA adapter**，而不是完整 checkpoint。部署时，可使用 `peft` 加载 base + adapter， 或者调用一次 `merge_and_unload()` 将其合并并导出为独立模型。

> **NOTE**
>
> 通过 bitsandbytes 实现的 4-bit QLoRA 需要具备计算能力 **sm_75** 或更高 （Turing / Ampere / Hopper）的 NVIDIA GPU。`traininghub0.1-cu126-amd64` runtime 已经捆绑了 `trl`、`peft` 和 `bitsandbytes`。默认的 `lora_sft` 后端是 `unsloth`； 如果需要完全控制，也可以在同一个 runtime image 上直接驱动 `peft` + `bitsandbytes`。
>
> **NPU**： bitsandbytes 4-bit 在 Huawei Ascend 上不可用。请在 `llamafactory0.9-cann8.5-arm64` runtime 上使用不含 4-bit 的 LoRA（`finetuning_type: lora`） 作为参数高效路径——参见 Fine-tune and Pretrain on Ascend NPU。

持续预训练 (CPT)

持续预训练 (CPT) 保持原始的 **next-token prediction** 目标，但将其运行在来自你的领域 （医疗、法律、代码、新语言）的 原始文本语料 上。它注入的是 *知识和词汇*——与 通过 chat 数据教授 *行为* 的 SFT/OSFT 不同。常见流水线是 **CPT → SFT → alignment**。

`training_hub` 通过相同的 `sft(...)` entrypoint 运行 CPT，并设置 `is_pretraining=True`：loss 会覆盖 *所有* token（不使用仅 assistant masking），并且文档会被打包到固定大小的 `block_size` window 中。

```
from training_hub import sft

result = sft(
    model_path="Qwen/Qwen2.5-7B",              # a BASE checkpoint
    data_path="/path/to/corpus.jsonl",          # raw text, one document p
er line
    ckpt_output_dir="/path/to/checkpoints/cpt_run",
    # continued pre-training
    is_pretraining=True,
    block_size=4096,                            # packed context-window le
ngth
    document_column_name="text",                # JSONL field holding the
raw text
    # core training — keep the LR low to limit forgetting
    num_epochs=1, effective_batch_size=128, learning_rate=5e-6,
    max_seq_len=4096, max_tokens_per_gpu=20000,
    nproc_per_node=8,
)
```

CPT 数据是原始文本，不是 chat turns——在 `document_column_name` 下每行一个 document：

```
{"text": "A paragraph or document of raw domain text ..."}
{"text": "Another document ..."}
```

CPT 专用参数：

参数	必需	描述
`is_pretraining`	是	切换到原始文本预训练（无 chat masking）
`block_size`	否	打包后的上下文窗口长度（例如 1024–4096）
`document_column_name`	否	保存原始文本的 JSONL 字段（默认 `"text"`）

CPT 会更新 所有 权重并写出完整 checkpoint。它可能导致 general ability 的 catastrophic forgetting——可通过较低的 learning rate、混入一部分通用领域文本，以及随后执行一次 SFT/OSFT 来缓解（将 `model_path` 指向 CPT checkpoint）。如果 forgetting 是主要顾虑，请优先选择 **OSFT**。

> **NOTE**
>
> **NPU**： 全参数持续预训练也可以通过 `MindSpeed-LLM` runtime（`pretrain_gpt.py`）在 Huawei Ascend 上运行。 参见 Fine-tune and Pretrain on Ascend NPU 和 `qwen25_pretrain_verify.ipynb` 示例。

多节点

在每个节点上运行该 notebook（或脚本），并使用相同的 `rdzv_id` / `rdzv_endpoint`，以及不同的 `node_rank`：

```
nproc_per_node = 8
nnodes         = 2
rdzv_id        = 42
rdzv_endpoint  = "10.0.0.1:29500"
node_rank      = 0   # 1 on the worker
```

在训练开始之前，所有节点都需要能够访问 `rdzv_endpoint`。

使用 MLflow Tracking 和 TrustyAI Evaluation 的每日 Fine-Tuning Pipeline

一个完整的 Kubeflow Pipeline，用于 fine-tune 一个 LLM（此示例中为 Qwen3-0.6B），在 MLflow 中跟踪运行，将生成的模型注册到 **MLflow Model Registry**，将其部署为 KServe `InferenceService`，使用 TrustyAI `LMEvalJob` 进行评估，将评估结果写回同一个 MLflow experiment，并清理临时 serving 资源。随后，该 pipeline 会连接到 **KFP Recurring Run**，从而每天触发一次——为你提供一条不断增长的 `train → eval` 运行历史，MLflow 的 compare 视图会将其转化为回归信号。

本指南将已经分别记录的各个部分整合到一起——Use Kubeflow Pipelines、Kubeflow Pipeline + MLflow Integration、Evaluate LLM——形成一个可运行的方案。如果某一步不熟悉，请先阅读这些文档。

目录

该 pipeline 的作用

前置条件

步骤 1 — 创建 MLflow token Secret 和共享 PVC

步骤 2 — pipeline

在离线隔离集群上使用真实数据集

1. 在可出站访问的机器上下载数据集

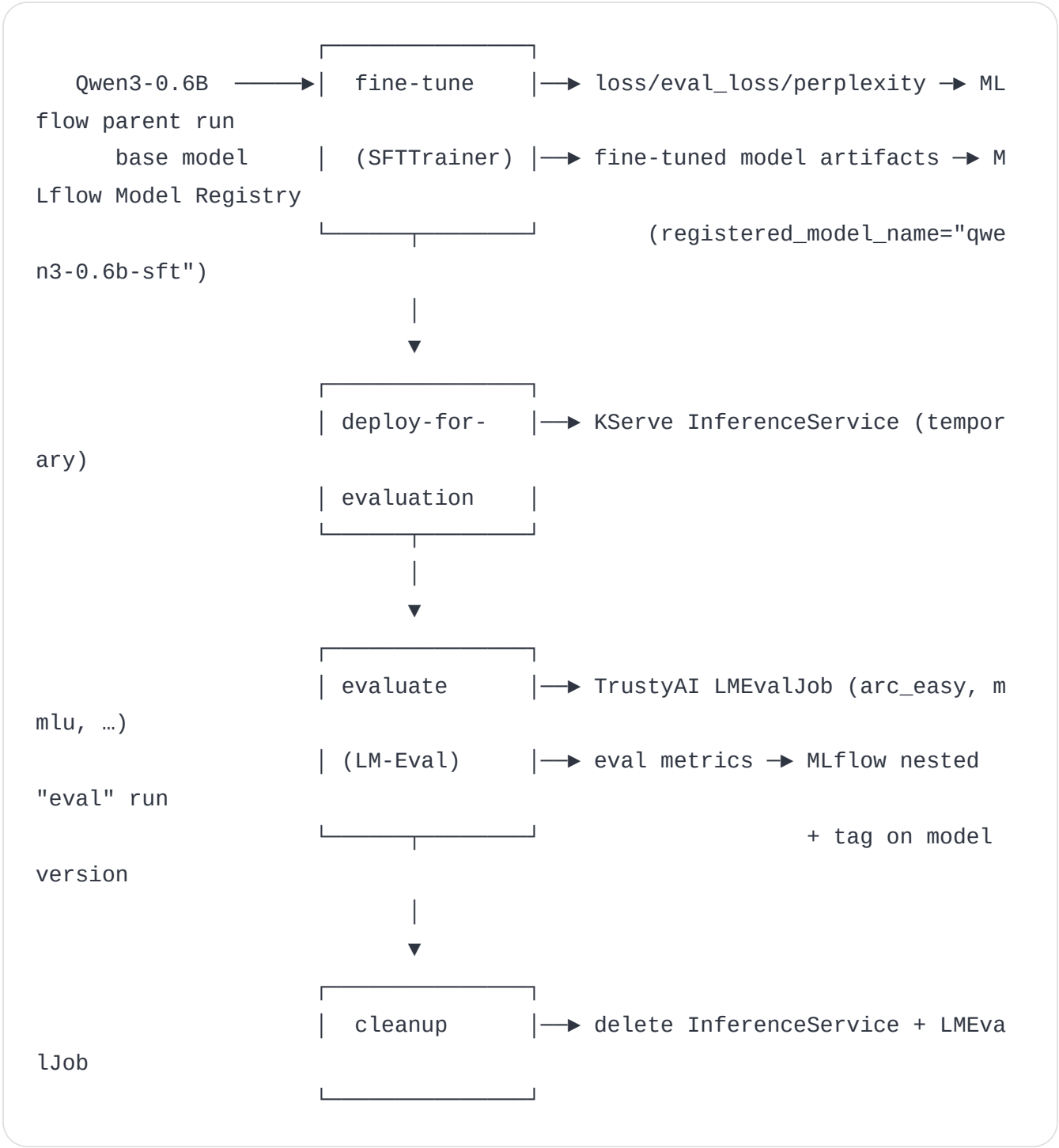
2. 转换为带有 `text` 列的 JSONL 文件

3a. 上传到共享 PVC（最简单）

3b. 上传到 S3 / seaweedfs（多个 pipeline 共享）

4. 关于基础模型的说明
步骤 3 — 提交一次性运行
步骤 4 — 在 MLflow 中可视化训练指标
步骤 5 — 将其安排为每日 Recurring Run
通过 UI
通过 SDK
步骤 6 — 在 MLflow 中比较每日评估结果
故障排查
构建自定义 image
相关指南

该 **pipeline** 的作用



每天的运行都会向 MLflow experiment 追加一行。由于训练和评估指标都位于同一个父运行下，MLflow 的 **Compare** 和 **Chart** 视图可以直接绘制按日变化的 loss 和评估准确率，而无需额外的胶水代码。

前置条件

需求	详情
Alauda AI 2.5 或更高版本	已安装 Kubeflow Pipelines、MLflow（已配置 **Model Registry** artifact store）、TrustyAI、KServe。
预构建的 pipeline runtime image	下方的四个 KFP component 使用 `docker.io/alaudadockerhub/finetune-pipeline-runtime-cu126-amd64:v0.1.0`（linux/amd64）。它内置了完整的 CUDA / HF / trl / MLflow / KServe / kubernetes 技术栈，因此 pod 启动时无需执行 `pip install`——在离线隔离集群中，只要将该镜像镜像到内部 registry，同样可以工作。请参见下方的 Bake a custom image 以推导你自己的镜像，以及 Training Runtime Images 获取源 Containerfile 目录。
MLflow artifact store	在 MLflow 插件中配置了 S3 兼容的对象存储。`mlflow.transformers.log_model()` 需要它来持久化模型文件，以便 KServe 后续拉取。请参见 MLflow install 中的 **High availability and storage** 部分。
名为 `finetune` 的 namespace，并带有 MLflow label	在 namespace 上设置 `mlflow-enabled=true`，使其显示为 workspace。
共享 PVC `finetune-shared`（RWX）	用于在 component 之间缓存基础模型和 fine-tuned checkpoint。任何支持 RWX 的 StorageClass 都可以（CephFS、NFS、JuiceFS）。
一个 GPU 节点	对于小型数据集，1 × NVIDIA GPU 且显存 ≥ 16 GiB 即可对 Qwen3-0.6B 进行全精度 SFT；对于小型 LoRA 运行，8 GiB 也足够。
MLflow token `Secret`	名为 `mlflow-token` 的 `Secret`，其中 key 为 `token`，保存一个 Dex id token，用于访问 `finetune` MLflow workspace 的 service account。请参见 Get a token from the command line。
`finetune` namespace 中的 RBAC	pipeline 的 ServiceAccount 可以对 `inferenceservices.serving.kserve.io` 和 `lmevaljobs.trustyai.opendatahub.io` 执行 `get` / `create` / `delete`。
训练数据集（可选）	`fine_tune` component 默认使用一个小型合成的进程内语料库——冒烟运行无需准备数据集。对于真实工作负载，请先在共享 PVC、seaweedfs 或 S3 上预先放置一个 JSONL 文件，并在 `dataset_path` 中传入其路径；请参见下方的 Using a real dataset。

需求	详情
到 Hugging Face 的出站访问（或离线 PVC）	默认有两个 component 会从 Hugging Face 拉取内容：（a）`fine_tune` 通过 `AutoModelForCausalLM.from_pretrained(...)` 下载基础模型 `Qwen/Qwen3-0.6B`；（b）`evaluate` component 的 `LMEvalJob` 会获取 tokenizer 和 dataset——即使 `tokenized_requests` 为 `False`，`AutoTokenizer.from_pretrained(...)` 也会执行。对于（a），请先将基础模型镜像到共享 PVC 一次（见 Using a real dataset → 基础模型说明）。对于（b），请将 eval job 切换到离线模式（见 Evaluate LLM）：预先在 PVC 中放入 tokenizer + dataset cache，设置 `spec.offline.storage.pvcName`，并将 `tokenizer` 的 `modelArgs` 指向挂载路径。

步骤 1 — 创建 MLflow token Secret 和共享 PVC

```
NS=finetune

# 1. Mint a Dex id token (browser-free) — see the SDK guide for ID_TOKEN.
kubectl -n $NS create secret generic mlflow-token --from-literal=token
="$ID_TOKEN"

# 2. Shared PVC used by fine-tune (write) and deploy (read).
kubectl -n $NS apply -f - <<'EOF'
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: finetune-shared
spec:
  accessModes: ["ReadWriteMany"]
  resources:
    requests:
      storage: 50Gi
  storageClassName: cephfs   # any RWX-capable StorageClass
EOF
```

下方的 pipeline component 假定这两个对象都存在于 pipeline 运行所在的 namespace 中。

步骤 2 — pipeline

将以下内容保存为 `finetune_pipeline.py`。该 pipeline 被拆分为四个 component，因此失败可以更精确地定位，而且重跑成本较低；component 通过 PVC 以及 模型版本上的 **MLflow tags** 共享状态。

```
# finetune_pipeline.py
from kfp import dsl, compiler, kubernetes

BASE_MODEL      = "Qwen/Qwen3-0.6B"
REGISTERED_NAME = "qwen3-0.6b-sft"
WORKSPACE       = "finetune"                        # MLflow workspace (=
namespace)
EXPERIMENT      = "qwen3-0.6b-daily-sft"
PVC_NAME        = "finetune-shared"
PVC_MOUNT       = "/mnt/shared"

MLFLOW_URI      = "http://mlflow-tracking-server.kubeflow:5000"

# Pre-built runtime image with torch (CUDA 12.6) + HF + trl + mlflow + ks
erve
# + kubernetes baked in. No pip install runs at pod start, so the same ta
g
# works on air-gapped clusters once it's mirrored into an internal regist
ry.
RUNTIME_IMAGE   = "docker.io/alaudadockerhub/finetune-pipeline-runtime-cu
126-amd64:v0.1.0"



# ----------------------------------------------------------------------
----
# 1. Fine-tune with the HF Trainer, autolog to MLflow, register the mode
l.
# ----------------------------------------------------------------------
----
@dsl.component(base_image=RUNTIME_IMAGE)
def fine_tune(
    workspace: str,
    experiment: str,
    base_model: str,
    registered_name: str,
    pvc_mount: str,
    run_id: str,
    # Empty = generate a tiny synthetic instruction corpus in-process (no
    # network egress; smoke-shape only, not a meaningful fine-tune). Non-
empty
    # = a local JSONL path (e.g. "/mnt/shared/datasets/alpaca.jsonl") or
an
    # "s3://…" URI to a JSONL file with a `text` column. See the "Using a
```

```
    # real dataset" section for how to prepare the file.
    dataset_path: str = "",
    num_train_epochs: int = 1,
    learning_rate: float = 2e-4,
    per_device_train_batch_size: int = 2,
    max_samples: int = 512,
) -> str:
    """Full SFT of `base_model` on a small synthetic corpus (or a JSONL f
ile
    at `dataset_path`). Logs metrics + registers the fine-tuned model.
    Returns the model version as a string."""
    import operator, os, random, mlflow, mlflow.transformers
    from datasets import Dataset, load_dataset
    from transformers import AutoModelForCausalLM, AutoTokenizer, Trainin
gArguments
    from trl import SFTTrainer

    # 1) MLflow (auth via MLFLOW_TRACKING_TOKEN injected from the Secre
t).
    mlflow.set_tracking_uri(MLFLOW_URI)
    mlflow.set_workspace(workspace)
    mlflow.set_experiment(experiment)
    mlflow.transformers.autolog(log_models=False)          # models are l
ogged manually below

    # 2) Data. Synthetic by default so the pipeline's smoke run needs no
    # network egress and no dataset pre-staging. Swap in a real dataset b
y
    # passing `dataset_path` (JSONL file with a `text` column) — either a
    # PVC path or an S3 URI. See "Using a real dataset" below.
    if dataset_path.startswith("s3://"):
        ds = load_dataset(
            "json", data_files=dataset_path, split=f"train[:{max_sample
s}]",
            # storage_options / credentials come from AWS_* env; see the
docs.
        )
    elif dataset_path:
        ds = load_dataset(
            "json", data_files=dataset_path, split=f"train[:{max_sample
s}]",
        )
    else:
        rng = random.Random(42)
```

```
        ops = [("+", operator.add), ("-", operator.sub), ("*", operator.m
ul)]
        rows = []
        for _ in range(max_samples):
            a, b = rng.randint(0, 99), rng.randint(0, 99)
            sym, op = rng.choice(ops)
            rows.append({"text":
                f"### Instruction:\nWhat is {a} {sym} {b}?\n"
                f"### Response:\nThe answer is {op(a, b)}."
            })
        ds = Dataset.from_list(rows)

    # 3) Model & tokenizer.
    tokenizer = AutoTokenizer.from_pretrained(base_model)
    model     = AutoModelForCausalLM.from_pretrained(base_model)

    output_dir = f"{pvc_mount}/models/{registered_name}/{run_id}"

    # 4) Train. `report_to="mlflow"` streams loss / eval_loss / lr per st
ep.
    args = TrainingArguments(
        output_dir=output_dir,
        num_train_epochs=num_train_epochs,
        learning_rate=learning_rate,
        per_device_train_batch_size=per_device_train_batch_size,
        logging_steps=10,
        save_strategy="no",
        report_to="mlflow",
        run_name=f"train-{run_id}",
    )
    with mlflow.start_run(run_name=f"pipeline-{run_id}") as parent:
        mlflow.set_tag("pipeline_run_id", run_id)
        mlflow.set_tag("base_model", base_model)
        with mlflow.start_run(run_name=f"train-{run_id}", nested=True):
            # TRL 0.12+ renamed the `tokenizer` kwarg to `processing_clas
s`.
            trainer = SFTTrainer(model=model, args=args, train_dataset=d
s,
                                 processing_class=tokenizer)
            trainer.train()
            trainer.save_model(output_dir)

        # 5) Register the fine-tuned model.
        # The artifact upload uses the MLflow plugin's artifact store (S
```

```
3).
        # KServe then pulls the model back from that S3 URI in the deploy
step.
        # `task="text-generation"` is required because MLflow cannot infe
r the
        # task from a locally-loaded (non-Hub) model.
        mv = mlflow.transformers.log_model(
            transformers_model={"model": trainer.model, "tokenizer": toke
nizer},
            artifact_path="model",
            registered_model_name=registered_name,
            task="text-generation",
        )

        # 6) Record the PVC path on the model version so downstream steps
        #    can serve it from the shared PVC without another S3 round-tr
ip.
        client = mlflow.MlflowClient()
        version = client.get_latest_versions(registered_name, stages=["No
ne"])[0].version
        client.set_model_version_tag(registered_name, version, "pvc_pat
h", output_dir)
        client.set_model_version_tag(registered_name, version, "pipeline_
run_id", run_id)

    print(f"registered {registered_name} version {version} @ {output_di
r}")
    return version


# ---------------------------------------------------------------------------
----
# 2. Deploy the just-registered model as a temporary KServe InferenceServ
ice.
# ---------------------------------------------------------------------------
----
@dsl.component(base_image=RUNTIME_IMAGE)
def deploy_for_evaluation(
    workspace: str,
    registered_name: str,
    model_version: str,
    pvc_name: str,
    pvc_mount: str,
    run_id: str,
```

```
) -> str:
    """Create an InferenceService that serves the model straight off the
PVC.
    Returns the InferenceService name."""
    import time, mlflow
    from kubernetes import client, config
    from kserve import KServeClient, constants
    from kserve import (V1beta1InferenceService, V1beta1InferenceServiceS
pec,
                        V1beta1PredictorSpec, V1beta1ModelSpec, V1beta1Mo
delFormat)

    # Resolve the PVC path from the tag set by fine_tune().
    mlflow.set_tracking_uri(MLFLOW_URI); mlflow.set_workspace(workspace)
    tags = mlflow.MlflowClient().get_model_version(registered_name, model
_version).tags
    pvc_path = tags["pvc_path"]

    isvc_name = f"{registered_name}-eval-{run_id[-8:]}"
    config.load_incluster_config()
    ns = open("/var/run/secrets/kubernetes.io/serviceaccount/namespace").
read().strip()

    isvc = V1beta1InferenceService(
        api_version=constants.KSERVE_GROUP + "/v1beta1", kind=constants.K
SERVE_KIND,
        metadata=client.V1ObjectMeta(name=isvc_name, namespace=ns, labels
={
            "app.kubernetes.io/part-of": "finetune-pipeline",
            "pipeline-run-id": run_id,
        }),
        spec=V1beta1InferenceServiceSpec(predictor=V1beta1PredictorSpec(
            model=V1beta1ModelSpec(
                model_format=V1beta1ModelFormat(name="huggingface"),
                runtime="kserve-huggingfaceserver",
                storage_uri=f"pvc://{pvc_name}{pvc_path.removeprefix(pvc_
mount)}",
                resources=client.V1ResourceRequirements(
                    limits={"nvidia.com/gpu": "1", "memory": "16Gi"},
                    requests={"nvidia.com/gpu": "1", "memory": "8Gi"},
                ),
            ),
        )),
    )
```

```
    KServeClient().create(isvc)

    # Wait for READY.
    deadline = time.time() + 15 * 60
    while time.time() < deadline:
        got = KServeClient().get(isvc_name, namespace=ns)
        cond = {c["type"]: c["status"] for c in (got.get("status", {}).ge
t("conditions") or [])}
        if cond.get("Ready") == "True":
            print(f"InferenceService {isvc_name} is Ready")
            return isvc_name
        time.sleep(10)
    raise TimeoutError(f"{isvc_name} did not become Ready in 15 min")


# ----------------------------------------------------------------------
----
# 3. Evaluate the InferenceService with a TrustyAI LMEvalJob, log to MLfl
ow.
# ----------------------------------------------------------------------
----
@dsl.component(base_image=RUNTIME_IMAGE)
def evaluate(
    workspace: str,
    experiment: str,
    registered_name: str,
    model_version: str,
    isvc_name: str,
    run_id: str,
    tasks: list = ["arc_easy", "hellaswag"],
    limit: str = "50",
) -> dict:
    """Create an LMEvalJob against the InferenceService, wait for it to
    Complete, log the metrics into the parent MLflow run + as a nested ru
n,
    and tag the model version with the primary accuracy."""
    import json, time, mlflow
    from kubernetes import client, config

    config.load_incluster_config()
    ns = open("/var/run/secrets/kubernetes.io/serviceaccount/namespace").
read().strip()
    api = client.CustomObjectsApi()
```

```
    job_name = f"eval-{isvc_name}"
    body = dict(
        apiVersion="trustyai.opendatahub.io/v1alpha1", kind="LMEvalJob",
        metadata=dict(name=job_name, labels={"pipeline-run-id": run_id}),
        spec=dict(
            model="local-completions",
            modelArgs=[
                dict(name="model", value=isvc_name),
                dict(name="base_url",
                     value=f"http://{isvc_name}-predictor.{ns}.svc/v1/com
pletions"),
                dict(name="num_concurrent", value="1"),
                dict(name="max_retries", value="3"),
                dict(name="tokenized_requests", value="True"),
                dict(name="tokenizer", value="Qwen/Qwen3-0.6B"),
            ],
            taskList=dict(taskNames=list(tasks)),
            allowOnline=True, allowCodeExecution=False,
            batchSize="1", limit=limit, logSamples=False,
            chatTemplate=dict(enabled=False),
            outputs=dict(pvcManaged=dict(size="100Mi")),
            # The ta-lmes-job image runs as UID 65532; a fresh PVC is roo
t-owned,
            # so the driver's `open(stdout.log)` fails with permission de
nied
            # unless fsGroup gives the pod's group write access to the mo
unt.
            pod=dict(securityContext=dict(fsGroup=65532)),
        ),
    )
    api.create_namespaced_custom_object(
        "trustyai.opendatahub.io", "v1alpha1", ns, "lmevaljobs", body)

    # Wait for a terminal state. `state` reaches `Complete` on both succe
ss
    # and failure — the reason field is what actually distinguishes them.
    deadline = time.time() + 60 * 60
    while time.time() < deadline:
        got = api.get_namespaced_custom_object(
            "trustyai.opendatahub.io", "v1alpha1", ns, "lmevaljobs", job_
name)
        status = got.get("status") or {}
        state, reason = status.get("state"), status.get("reason")
        if state == "Complete" and reason == "Succeeded":
```

```
        if state == "Complete" and reason == "Succeeded":
            break
        if state in {"Cancelled", "Failed"} or (state == "Complete" and r
eason in {"Failed", "Cancelled"}):
            raise RuntimeError(
                f"LMEvalJob {job_name} ended: state={state} reason={reaso
n} "
                f"message={status.get('message')}")
        time.sleep(20)
    else:
        raise TimeoutError(f"LMEvalJob {job_name} did not finish in 1 h")

    results = json.loads(got["status"]["results"])["results"]

    # Log each task's metrics back to MLflow.
    mlflow.set_tracking_uri(MLFLOW_URI)
    mlflow.set_workspace(workspace)
    mlflow.set_experiment(experiment)
    parent = mlflow.search_runs(filter_string=f"tags.pipeline_run_id =
'{run_id}'",
                                order_by=["attributes.start_time DESC"],
max_results=1)
    parent_run_id = parent.iloc[0]["run_id"]

    flat = {}
    with mlflow.start_run(run_id=parent_run_id):
        with mlflow.start_run(run_name=f"eval-{run_id}", nested=True):
            for task, metrics in results.items():
                for k, v in metrics.items():
                    if isinstance(v, (int, float)) and "stderr" not in k:
                        name = f"{task}/{k.replace(',', '_')}"
                        mlflow.log_metric(name, float(v))
                        flat[name] = float(v)
            mlflow.set_tag("model_version", model_version)
            mlflow.set_tag("isvc_name", isvc_name)

    # Tag the primary accuracy on the model version so it shows up on the
    # Model Registry page next to the version number.
    primary = next((k for k in flat if k.endswith("/acc_none")), None)
    if primary is not None:
        mlflow.MlflowClient().set_model_version_tag(
            registered_name, model_version, "eval_acc", f"{flat[primar
y]:.4f}")
    return flat
```

```
# ------------------------------------------------------------------------
----
# 4. Cleanup: delete the temporary InferenceService and LMEvalJob.
# ------------------------------------------------------------------------
----
@dsl.component(base_image=RUNTIME_IMAGE)
def cleanup(isvc_name: str):
    from kubernetes import client, config
    config.load_incluster_config()
    ns = open("/var/run/secrets/kubernetes.io/serviceaccount/namespace").
read().strip()
    api = client.CustomObjectsApi()
    for gv, plural, name in [
        ("serving.kserve.io/v1beta1", "inferenceservices", isvc_name),
        ("trustyai.opendatahub.io/v1alpha1", "lmevaljobs", f"eval-{isvc_n
ame}"),
    ]:
        group, version = gv.split("/")
        try:
            api.delete_namespaced_custom_object(group, version, ns, plura
l, name)
        except Exception as exc:
            print(f"delete {plural}/{name} skipped: {exc}")


# ------------------------------------------------------------------------
----
# Pipeline: wire the four components together.
# ------------------------------------------------------------------------
----
@dsl.pipeline(name="qwen3-sft-mlflow-trustyai",
              description="Daily SFT + MLflow tracking + TrustyAI evaluat
ion")
def sft_mlflow_trustyai(
    workspace: str = WORKSPACE,
    experiment: str = EXPERIMENT,
    base_model: str = BASE_MODEL,
    registered_name: str = REGISTERED_NAME,
    # Default = synthetic in-process data (offline smoke). Override with
a
    # JSONL path on the shared PVC or an s3:// URI to fine-tune on a real
    # dataset — see the "Using a real dataset" section.
```

```
    dataset_path: str = "",
    num_train_epochs: int = 1,
    learning_rate: float = 2e-4,
    max_samples: int = 512,
    eval_limit: str = "50",
):
    train = fine_tune(
        workspace=workspace, experiment=experiment,
        base_model=base_model, registered_name=registered_name,
        pvc_mount=PVC_MOUNT, run_id=dsl.PIPELINE_JOB_ID_PLACEHOLDER,
        dataset_path=dataset_path, num_train_epochs=num_train_epochs,
        learning_rate=learning_rate, max_samples=max_samples,
    ).set_accelerator_type("nvidia.com/gpu").set_accelerator_limit(1)
    train.set_memory_limit("32Gi").set_cpu_limit("8")

    deploy = deploy_for_evaluation(
        workspace=workspace, registered_name=registered_name,
        model_version=train.output, pvc_name=PVC_NAME, pvc_mount=PVC_MOUN
T,
        run_id=dsl.PIPELINE_JOB_ID_PLACEHOLDER,
    )

    ev = evaluate(
        workspace=workspace, experiment=experiment,
        registered_name=registered_name, model_version=train.output,
        isvc_name=deploy.output, run_id=dsl.PIPELINE_JOB_ID_PLACEHOLDER,
    )

    with dsl.ExitHandler(cleanup(isvc_name=deploy.output)):
        ev  # cleanup runs whether evaluate() succeeds or not

    # Mount the shared PVC on training + deploy.
    for task in (train, deploy):
        kubernetes.mount_pvc(task, pvc_name=PVC_NAME, mount_path=PVC_MOUN
T)

    # Inject the Dex id token into every component that talks to MLflow.
    for task in (train, deploy, ev):
        kubernetes.use_secret_as_env(
            task, secret_name="mlflow-token",
            secret_key_to_env={"token": "MLFLOW_TRACKING_TOKEN"})


compiler.Compiler().compile(sft_mlflow_trustyai, "pipeline.yaml")
```

编译：

```
pip install "kfp>=2.7" "kfp-kubernetes>=1.3"
python finetune_pipeline.py
# → pipeline.yaml
```

在离线隔离集群上使用真实数据集

`fine_tune` component 默认使用一个合成的进程内 instruction 语料库（几百条算术问答对）。这可以让冒烟运行完全离线——不需要 HuggingFace、ModelScope、S3——但它不会产出有意义的 fine-tune 结果。对于真实工作负载，请先将数据集一次性预置到集群中，然后在每次运行时通过 `dataset_path` 传入其路径。`fine_tune` component 可以接受本地 JSONL 路径（最简单）或 `s3://…` URI（可供多个 pipeline 共享）。

1. 在可出站访问的机器上下载数据集

任何可以访问 Hugging Face 的机器都可以——笔记本电脑、跳板机、或者某个 namespace 中可出站访问的 Workbench：

```
pip install huggingface_hub datasets
huggingface-cli download tatsu-lab/alpaca --repo-type dataset --local-dir
./alpaca-raw
```

如果 Hugging Face 被阻断，ModelScope 会镜像大多数常用数据集，并且在中国大陆网络环境下可用：

```
pip install modelscope
modelscope download --dataset AI-ModelScope/alpaca-gpt4-data-en --local_d
ir ./alpaca-raw
```

2. 转换为带有 `text` 列的 JSONL 文件

TRL 的 `SFTTrainer` 要求每一行只有一个文本字段。请将原始数据集转换为 pipeline 所消费的格式：

```
from datasets import load_dataset

raw = load_dataset("./alpaca-raw")["train"]

def format_alpaca(row):
    prompt = row["instruction"]
    if row.get("input"):
        prompt += "\n\n" + row["input"]
    return {"text": f"### Instruction:\n{prompt}\n### Response:\n{row['output']}"}

raw.map(format_alpaca).to_json("alpaca.jsonl", orient="records", lines=True)
```

3a. 上传到共享 PVC（最简单）

如果你的集群中有一个挂载了 `finetune-shared` PVC 的 jump-pod，可以使用 `kubectl cp` 将 JSONL 复制进去：

```
kubectl -n finetune apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata: { name: dataset-uploader, namespace: finetune }
spec:
  restartPolicy: Never
  containers:
  - name: sh
    image: busybox
    command: ["sh", "-c", "mkdir -p /mnt/shared/datasets && sleep 3600"]
    volumeMounts: [{ name: shared, mountPath: /mnt/shared }]
  volumes:
  - name: shared
    persistentVolumeClaim: { claimName: finetune-shared }
EOF
kubectl -n finetune wait --for=condition=Ready pod/dataset-uploader
kubectl -n finetune cp alpaca.jsonl dataset-uploader:/mnt/shared/dataset
s/alpaca.jsonl
kubectl -n finetune delete pod dataset-uploader
```

然后在提交 pipeline 时使用 `dataset_path="/mnt/shared/datasets/alpaca.jsonl"`（见下方步骤 3）。

3b. 上传到 S3 / seaweedfs（多个 pipeline 共享）

对于运行大量 pipeline 的团队，建议将数据集托管在与 MLflow 插件 artifact store 相同的 S3 兼容 bucket 中——集群内部的 seaweedfs endpoint 可被每个 pipeline pod 访问，而客户端只需要 `AWS_ACCESS_KEY_ID` / `AWS_SECRET_ACCESS_KEY` 环境变量。

请从能够访问 S3 endpoint 的机器上传——通常是同一集群中的 Workbench。Alauda AI Workbench 镜像已经自带 `mc`（MinIO Client）和 `boto3`，因此无需额外安装。

使用 `mc`（交互式 / 单行命令）：

```
mc alias set seaweedfs http://seaweedfs.mlops-demo-e2e:8333 <ACCESS_KEY>
<SECRET_KEY>
mc mb --ignore-existing seaweedfs/mlops-datasets
mc cp alpaca.jsonl seaweedfs/mlops-datasets/alpaca/alpaca.jsonl
```

使用 `boto3`（脚本化 / 在 **notebook** 中）：

```
import boto3

s3 = boto3.client(
    "s3",
    endpoint_url="http://seaweedfs.mlops-demo-e2e:8333",
    aws_access_key_id="<ACCESS_KEY>",
    aws_secret_access_key="<SECRET_KEY>",
)
# Idempotent bucket create.
try:
    s3.create_bucket(Bucket="mlops-datasets")
except s3.exceptions.BucketAlreadyOwnedByYou:
    pass
s3.upload_file("alpaca.jsonl", "mlops-datasets", "alpaca/alpaca.jsonl")
```

（请根据你的 seaweedfs / MinIO Service 调整 endpoint。凭据来自 MLflow 插件的 artifact-store Secret，或来自你的本地 S3 提供商。）

然后创建一个包含 S3 凭据的 Secret，并将其与 `mlflow-token` 一起注入到 `fine_tune` 中：

```
kubectl -n finetune create secret generic mlflow-s3 \
  --from-literal=AWS_ACCESS_KEY_ID=<key> \
  --from-literal=AWS_SECRET_ACCESS_KEY=<secret> \
  --from-literal=MLFLOW_S3_ENDPOINT_URL=http://seaweedfs.mlops-demo-e2e:8
333
```

将以下内容添加到 pipeline function 中（紧接在现有的、挂载 MLflow token 的 `use_secret_as_env` block 旁边）：

```
kubernetes.use_secret_as_env(train, secret_name="mlflow-s3", secret_key_t
o_env={
    "AWS_ACCESS_KEY_ID":       "AWS_ACCESS_KEY_ID",
    "AWS_SECRET_ACCESS_KEY":   "AWS_SECRET_ACCESS_KEY",
    "MLFLOW_S3_ENDPOINT_URL":  "AWS_ENDPOINT_URL",   # what s3fs / fsspec
read
})
```

然后使用 `dataset_path="s3://mlops-datasets/alpaca/alpaca.jsonl"` 提交 pipeline。`s3fs` 已经包含在 runtime image 中，`load_dataset("json", data_files="s3://…")` 会在底层使用 `fsspec` —— `AWS_*` 环境变量会自动被拾取。

4. 关于基础模型的说明

`AutoModelForCausalLM.from_pretrained("Qwen/Qwen3-0.6B")` 在首次运行时也会访问 Hugging Face。对于完全离线隔离的 fine-tune，请先镜像一次模型（同样的模式：`huggingface-cli download Qwen/Qwen3-0.6B --local-dir ./qwen3-0.6b`，然后复制到 PVC 或 S3），并设置 `BASE_MODEL = "/mnt/shared/models/qwen3-0.6b"`。随后 MLflow model registry 会像之前一样将该 fine-tune 注册为 `qwen3-0.6b-sft`。

步骤 3 — 提交一次性运行

在 **Kubeflow Dashboard** → **Pipelines** → **Upload Pipeline** 中上传 `pipeline.yaml`，然后点击 **Create Run**——或者从 Workbench 中执行：

```
from kfp.client import Client

client = Client(host="<MY-KFP-ENDPOINT>")
run = client.create_run_from_pipeline_package(
    "pipeline.yaml",
    arguments=dict(
        # Omit for the synthetic-smoke default; set to a JSONL path or an
        # s3:// URI to fine-tune on a real dataset (see the section abov
e).
        # dataset_path="/mnt/shared/datasets/alpaca.jsonl",
        num_train_epochs=1, eval_limit="50",
    ),
)
print("Run:", run.run_id)
```

当运行进行中时：

- 打开 **Alauda AI** → **Tools** → **MLFlow**，并选择 `finetune` workspace。

- `qwen3-0.6b-daily-sft` experiment 会显示一个名为 `pipeline-<pipeline-run-id>` 的父运行，并带有两个嵌套运行：`train-<...>`（由 HF Trainer callback 流式写入）以及稍后的 `eval-<...>`（由 `evaluate` 步骤写入）。
- 在 **Models** 下，`qwen3-0.6b-sft` registered model 会新增一个 version。

步骤 4 — 在 MLflow 中可视化训练指标

1. 在 MLflow UI 中，打开父运行 → **Metrics** tab。
2. 选择 `loss`、`eval_loss` 以及你添加的任何自定义指标（`perplexity`、`learning_rate`、`grad_norm`）→ 点击 **Chart**。
3. 若要实时查看 loss 曲线，请在图表面板中将 **Refresh** 间隔设置为 `10s` ——随着 Trainer 写入新的 step，MLflow 会重新获取指标流。

对于分类类目标，如果 accuracy 很重要，可以在 `TrainingArguments` 中添加一个 `compute_metrics` callback，返回 `{"accuracy": ...}`；`report_to="mlflow"` 会把 `compute_metrics` 返回的内容直接流式写入同一张图表。

步骤 5 — 将其安排为每日 Recurring Run

KFP 原生支持 cron 调度。将同一个 `pipeline.yaml` 连接到 **Recurring Run**，使其每天触发一次。

通过 UI

1. **Pipelines** → 选择 `qwen3-sft-mlflow-trustyai` → **Create Run**。
2. 将运行类型选择为 **Recurring Run**。
3. **Trigger**：`Cron`，表达式 `0 2 * * *`（每天 UTC 02:00）。
4. **Max concurrent runs**：`1` ——该 pipeline 会占用 1 个 GPU；重叠运行会排队。
5. **Catchup**：`off` ——跳过错过的时间窗口，而不是在周一早上补跑积压任务。
6. **Parameters**：对于每日任务，设置 `num_train_epochs=1`、`eval_limit=200`（比一次性默认值更具信号）。

7. **Start**。

通过 SDK

```
from kfp.client import Client

client = Client(host="<MY-KFP-ENDPOINT>")

# One-time: upload the pipeline as a versioned resource.
uploaded = client.upload_pipeline(
    "pipeline.yaml", pipeline_name="qwen3-sft-mlflow-trustyai")

# Attach a daily schedule.
client.create_recurring_run(
    experiment_id=client.create_experiment("qwen3-daily").experiment_id,
    job_name="qwen3-sft-daily-02utc",
    cron_expression="0 0 2 * * *",      # KFP uses 6-field cron (with seco
nds)
    max_concurrency=1,
    no_catchup=True,
    enabled=True,
    pipeline_id=uploaded.pipeline_id,
    version_id=uploaded.pipeline_version_id,
    params=dict(num_train_epochs=1, eval_limit="200"),
)
```

WARNING

`Secret/mlflow-token` 中的 MLflow token 会过期（Dex id token 默认有效期为 24 小时）。对于运行时间超过 token 生命周期的计划任务，可以选择：在 `fine_tune` component 内使用 service-account 凭据生成 token 并在其中刷新它（请参见 SDK token flow），或者运行一个小型 CronJob，以略短于 token TTL 的周期轮换 `mlflow-token` Secret。

步骤 6 — 在 MLflow 中比较每日评估结果

运行几天之后，experiment 中就会出现每天一行的记录。MLflow 可以让你一键完成比较：

1. 在 MLflow UI 中，打开 `qwen3-0.6b-daily-sft` experiment。
2. 在 runs 表中，筛选嵌套的 eval runs：`tags.mlflow.runName LIKE 'eval-%'`。
3. 按 `attributes.start_time DESC` 排序。
4. 勾选你要比较的 runs（建议先看最近 7 天）→ 点击 **Compare**。
5. 在 **Chart** tab 中，按 `tags.mlflow.runName` 分组绘制 `arc_easy/acc_none` 和 `hellaswag/acc_none`。稳定的曲线表示 fine-tune 是稳定的；明显下跌通常意味着训练数据集或基础 checkpoint 发生了变化。
6. **Parallel Coordinates** 视图会把训练超参数和评估指标分组显示，便于你肉眼判断哪些超参数与最大的评估收益相关。

对于程序化比较——例如将结果发送到 Slack 告警，或者在 accuracy 日环比下降超过 2% 时让 CI gate 失败——可以使用 search API：

```
import mlflow, pandas as pd

mlflow.set_tracking_uri("http://mlflow-tracking-server.kubeflow:5000")
mlflow.set_workspace("finetune")

df = mlflow.search_runs(
    experiment_names=["qwen3-0.6b-daily-sft"],
    filter_string="tags.mlflow.runName LIKE 'eval-%'",
    order_by=["attributes.start_time DESC"],
    max_results=14,
)[["run_id", "start_time", "metrics.arc_easy/acc_none",
   "metrics.hellaswag/acc_none"]]

df.sort_values("start_time")\
  .assign(delta=lambda d: d["metrics.arc_easy/acc_none"].diff())\
  .to_string(index=False)
```

若要将每日得分最高的版本提升到下游 serving 栈，可以使用 Model Registry 的别名 API——pipeline 已经为每个 version 标记了 `eval_acc`：

```
client = mlflow.MlflowClient()
best = max(client.search_model_versions("name='qwen3-0.6b-sft'"),
           key=lambda v: float(v.tags.get("eval_acc", "0")))
client.set_registered_model_alias("qwen3-0.6b-sft", "champion", best.vers
ion)
# Downstream InferenceService references models:/qwen3-0.6b-sft@champion.
```

故障排查

症状	检查项
`log_model` 因 `NoCredentialsError` / `Endpoint URL error` 失败	MLflow artifact store 未配置为 S3。请参见 MLflow install 中 **availability and storage** 部分，并在 MLflow 插件中设置 ar bucket。然后重新运行。
`deploy_for_evaluation` 一直看不到 `Ready=True`	检查 InferenceService：`kubectl -n finetune describe <name>`。最常见的原因是 `pvc_path` tag 中的 PVC 路径不 集群中没有可用的 KServe HF runtime image。
`LMEvalJob` 一直停留在 `Scheduled`	eval job pod 处于 Pending。请检查 pod 的 node selector 和 `kubectl -n finetune get pods -l app=<job-name>`，对 pod 执行 `kubectl describe`。
`LMEvalJob` 无法访问 InferenceService	`modelArgs` 中的 `base_url` 必须是 `-predictor` Servic 层 InferenceService。示例使用的是 `http://<isvc>-predi <ns>.svc/v1/completions` —— `-predictor` 后缀和 `/v1/completions` 路径对于 OpenAI-compatible endpoint
`LMEvalJob` 结束为 `state=Complete, reason=Failed, message="open …/output/stdout.log: permission denied"`	由 operator 管理的 outputs PVC 属于 root 用户，但 `ta-lm` 以 UID 65532 运行，因此 driver 无法创建 `stdout.log`。请 `spec.pod.securityContext.fsGroup: 65532`（示例中已 便挂载对 pod 的 supplemental group 可写。
`state=Complete`，但 pipeline 将其视为成功，即使运行失败了	对于任何终态，`state` 都会转为 `Complete`；真正的结果 `status.reason` 中（`Succeeded` / `Failed` / `Cancelle` 检查二者，就像上面的 `evaluate` component 一样。
`local-completions` 失败并报 `OSError: We couldn't connect to`	即使 `tokenized_requests` 为 `"False"`，lm-evaluation- 端仍然会调用 `AutoTokenizer.from_pretrained(...)` —

症状	检查项
`'https://huggingface.co' to load this file … it looks like <name> is not the path to a directory containing a file named config.json.`	`tokenizer` 的 `modelArgs` 条目必须是 eval pod 可以访问 或者是离线 PVC 下的本地路径。在离线隔离集群上，请参照 LLM 中的 **Optional: offline storage and PVC** 部分——在 `spec.offline.storage.pvcName` 挂载一个 PVC，预先填 和 dataset cache，并将 `tokenizer` 设置为挂载路径。
嵌套 MLflow runs 在父运行下缺失	`set_experiment` 在 `search_runs` 解析父运行之前就被调 实际上没有任何内容被记录，因为父 tag 是从另一个 compon 写入的。请在父运行内部使用 `set_tag("pipeline_run_id run_id)` ，并按该 tag 搜索，就像 `evaluate` component 内
Recurring Run 已触发，但运行一开始就因 `401 UNAUTHENTICATED` 失败	`Secret/mlflow-token` 中的 token 已在上一次成功的 pipe 运行之间过期。请轮换它（或者把 token 的生成移到 compo 参见步骤 5 中的警告。
KFP v2 pod 失败，并提示 `container has runAsNonRoot and image will run as root`（发生在 argoexec `init` init-container 和 `kfp-launcher` init-container 上）	KFP-v2 pod template 在 pod 级别设置了 `runAsNonRoot: t` argoexec / kfp-launcher image 没有设置非 root 的 `USER` 。 题的后端（例如 DSPO 基于 Argo 的 pipeline 栈）上，请在这 `spec.podSpecPatch` 修补 `Workflow` ，在 pod 级别以及每 container（`init` + `kfp-launcher` ）上都设置 `runAsUse`
fine-tune component 失败，并报 `PermissionError: [Errno 13] Permission denied: '/mnt/shared/...'`	共享 PVC 的根目录属于 root 用户。你可以在同一个 `podSp` 添加 `fsGroup: 1001` ，或者先使用一个以 root 运行的临时 次 PVC，并执行 `chown -R 1001:1001 /mnt` 。
fine-tune component 失败，并报 `modelscope_hub.errors.CacheError: [E1022] Failed to create SDK directories: [Errno 13] Permission denied: '/.modelscope'`	`python:3.12-slim` container 将 `HOME=/` 保持不变，而 M 和 Hugging Face 默认将缓存放在 `~/` 。非 root pod 无法在 fine-tune component 中，请在第一次执行 `from modelscop` / `AutoModel.from_pretrained` 之前，将 `MODELSCOPE_CA` `HF_HOME` 和 `HOME` 设置为共享 PVC 上的路径（或 `/tmp`
runtime image tag 在 worker node 上无法访问	离线隔离或受限出站的集群无法直接从 `docker.io` 拉取。 `skopeo copy` 或 `crane copy` 将 `docker.io/alaudadockerhub/finetune-pipeline-runtim amd64:v0.1.0` 镜像到你的内部 registry，然后将 `RUNTIME_` 镜像后的引用，并将相应的 `imagePullSecrets` 添加到 pip ServiceAccount。
在 KFP API server 上提交 pipeline 失败，报 `unknown component`	后端（例如 DSPO 基于 Argo 的 pipeline 栈）不接受 KFP v2 `dsl.ExitHandler` 子图。请将 `with`

症状	检查项
`implementation: comp-exit-handler-1`	`dsl.ExitHandler(cleanup(...)): ev` 替换为普通的 `cleanup(...).after(ev)`，并使 `cleanup` 幂等化（通过 selector 删除，使其在 `evaluate` 已经在成功时删除资源的工作）。
尽管 PVC 中存在该文件，KServe pod 仍然崩溃，并报 `FileNotFoundError: No such file or directory: /mnt/models/model.safetensors`	`KServe` 容器以 UID 1000 运行（在 `ClusterServingRunt` 义），而 `fine_tune` component 使用默认 umask 以 UID 这会生成模式 `0660` ——对 group 可写，但对其他用户不可 `save_pretrained` 之后增加一个 `chmod` 处理（如示例所 目录设置为 `0755`，每个文件设置为 `0644`。仅在 fine-tun `fsGroup: 1001` 不够——它只会更改挂载根目录的 group 文件的权限模式。
`evaluate` component 内部出现 `mlflow.exceptions.RestException: RESOURCE_DOES_NOT_EXIST: No Experiment with id=0 exists`	`start_run(run_id=parent)` 会附加到现有运行，但嵌套的 `start_run(run_name=...)` 需要一个 experiment 上下文。 何 run 之前，于 component 顶部一次性调用 `mlflow.set_experiment(experiment)`。否则 MLflow 会 experiment `0`，而多租户 server 不允许这样做。
`mlflow.create_model_version` 失败，并报 `INVALID_PARAMETER_VALUE: Invalid model version source: '/mnt/…'. To use a local path as a model version source, the run_id request parameter has to be specified and the local path has to be contained within the artifact directory of the run specified by the run_id.`，随后切换到 `mlflow.log_artifacts` 又触发 `PermissionError: [Errno 13] Permission denied: '/mlflow'`	Alauda MLflow 插件默认将 artifact root 设为 `/mlflow/arti` 这是 tracking server pod 上 `emptyDir` 中的本地路径。客户 `log_artifacts` / `log_model` 无法写入该位置，而 `run` 录之外的 `file://` version source 会被拒绝。你有两个选择 MLflow 部署重新配置为 S3 artifact backend（参见 MLflow i **High availability and storage** 部分）；或者（b）跳过 mod 通过 父运行上的 **tags** 协调下游步骤——pipeline 仍然可以做 在 MLflow 中只有一行记录，并且可以通过 `mlflow.get_run(...).data.tags` 访问 `pvc_path`、`fi` `eval_acc`。
`TypeError: SFTTrainer.__init__() got an unexpected keyword argument 'tokenizer'`	TRL 0.12+ 已移除了 `tokenizer` 关键字——请改为将 toke `processing_class=tokenizer` 传入。参考 runtime image LLaMA-Factory 0.9.4 间接包含了 post-0.12 版本的 TRL。上 用的是 `processing_class=`；仍然传递 `tokenizer=` 的 同样的重命名。
`mlflow.transformers.log_model` 失败，并报 `MlflowException: The`	MLflow 可以从 Hub 模型推断 pipeline task，但不能从本地加 像）的模型推断——fine-tune pod 的 `trainer.model` 在

症状	检查项
`task could not be inferred from the model. If you are saving a custom local model that is not available in the Hugging Face hub, please provide the 'task' argument to the log_model or save_model function.`	`AutoModelForCausalLM.from_pretrained("/mnt/shared/` 之后属于后者。请在 `log_model` 中传入 `task="text-generation"`（或者针对其他模型类型传入相应的 task 字符
MLflow 客户端调用失败，并报 `UNAUTHENTICATED: Authentication with the Kubernetes API failed. The provided token may be invalid or expired.`	mlflow-tracking-server 的 `oauth2-proxy` sidecar 没有转发——插件配置界面没有暴露该项，必须在 Deployment 上打补 `oauth2-proxy` 容器的 env 中添加 `OAUTH2_PROXY_SKIP_JWT_BEARER_TOKENS=true`，等待 rol 后重新运行。请注意，mlflow-plugin controller 会协调该 Dep 可能回滚此 env；每次 operator 升级或手动重新安装后都需

构建自定义 image

参考 image 基于 `llamafactory0.9-cu126-amd64` 构建，仅额外添加了 `mlflow>=3.10`、`kserve>=0.13` 和 `kubernetes>=29`。如果你想自行构建——例如切换 CUDA 版本、添加专有数据处理库，或者使基础镜像符合 FIPS——可以编写一个单层 `Containerfile`：

```
# syntax=docker/dockerfile:1.7
ARG BASE_IMAGE=docker.io/alaudadockerhub/llamafactory0.9-cu126-amd64:v0.1.0
FROM ${BASE_IMAGE}

USER 0
RUN uv pip install --no-cache-dir \
        "mlflow>=3.10" \
        "kubernetes>=29" \
        "kserve>=0.13" \
        # add anything your pipeline imports:
        # "your-private-etl-lib==1.2.3" \
        ;
USER 1000
WORKDIR /workspace
```

使用 `buildctl`（或 `docker buildx build`）通过你的 buildkitd 进行构建，推送到内部 registry，然后相应地设置 `RUNTIME_IMAGE`。pipeline 的其余代码保持不变。

如果你的集群已经在运行 Kubeflow Trainer v2，你可以复用已发布的某个 `TrainingRuntime` image——`llamafactory0.9-cu126-amd64:v0.1.0` 已经自带 torch / HF / trl / mlflow；对于 `deploy_for_evaluation` + `evaluate` + `cleanup` component，只需要再添加 `kserve` 和 `kubernetes`。参考 image 正是这样做的。

相关指南

- Kubeflow Pipeline + MLflow Integration — 本指南扩展的更小入门文档。
- 使用带认证和 RBAC 的 MLflow Python SDK — `MLFLOW_TRACKING_TOKEN` 如何获取与刷新。
- Evaluate LLM — `LMEvalJob` 的完整参考，包括离线 / 离线隔离模式。
- Use Kubeflow Pipelines — Recurring Runs UI 和对象存储配置。
- Training Runtime Images — 参考 image 继承而来的预构建 framework image 目录（torch、LLaMA-Factory、TrainingHub、MindSpeed-LLM）。
- 使用 Kubeflow Trainer v2 进行 Fine-Tuning — 上述 SFT component 的 `TrainJob` 替代方案。

使用 Workbench 微调 LLM

本指南介绍如何使用从 Alauda AI Workbench 启动的 LLaMA-Factory↗ 微调 LLM（示例：`Qwen3-0.6B`）。Notebook 会向集群提交一个 VolcanoJob↗，因此 GPU 任务在集群节点上运行，而你可以继续在 JupyterLab 中迭代。

当你需要交互式控制、自定义训练脚本以及针对每次实验调整 YAML 时，可以使用此方法。对于可复用的模板和配额，建议改用 Kubeflow Trainer v2。

目录

适用范围

前提条件

1. 创建 Notebook / VSCode 实例
2. 准备基础模型
3. 准备输出模型位置
4. 准备数据集

   HuggingFace `datasets` 格式

   LLaMA-Factory 格式
5. 运行时镜像
6. 提交微调 VolcanoJob

   NFS workspace PVC 注意事项
7. 管理作业
8. 实验跟踪
9. 发布微调后的模型

在非 NVIDIA GPU 上运行

在其他设备上进行实验跟踪

适用范围

- Alauda AI 1.3 及更高版本。
- 在 x86_64 + NVIDIA GPU 上微调 LLM。其他模型系列（例如 YOLOv5）需要使用各自的镜像、脚本和数据集格式。
- NPU 集群需要与供应商软件栈兼容的运行时镜像 — 请参阅下方的在非 NVIDIA GPU 上运行或 Ascend NPU recipes。

前提条件

- 已安装 Alauda AI Workbench 插件（或 Kubeflow Base + Notebook）。
- 已安装 MLflow 插件，用于实验跟踪。

1. 创建 Notebook / VSCode 实例

在 **Alauda AI** → **Workbench**（或 **Advanced** → **Kubeflow** → **Notebook**）中创建 workbench。workbench 本身只应请求 CPU — GPU 由它提交的 VolcanoJob 请求。请参阅创建 Workbench。

2. 准备基础模型

将 `Qwen/Qwen3-0.6B`（或任何 Hugging Face 模型）下载到共享 PVC 或训练作业可访问的 S3 兼容对象存储中。如果希望通过控制台部署或复用基础模型，请在 Model Registry 中注册生成的 Model URI。请参阅使用 Workbench 上传模型和 Model Registry。

3. 准备输出模型位置

使用共享 PVC 目录保存微调输出，或为作业配置一个 S3 位置以写入输出。作业完成后，使用其 Model URI 将输出 artifact 注册为新的 Model Registry 版本。无需预先创建占位模型记录。

4. 准备数据集

使用示例身份数据集训练模型回答“你是谁？”。将解压后的文件上传到训练作业可访问的 PVC 或 S3 位置，并将 `DATASET_URI` 设置为该位置。数据集布局必须与微调框架一致。

> 数据集格式必须符合微调框架的要求。

HuggingFace `datasets` 格式

```
import datasets
print(datasets.get_dataset_infos("<dataset directory>"))
print(datasets.load_dataset("<dataset directory>"))
```

LLaMA-Factory 格式

如果使用 LLaMA-Factory，请采用其要求的布局 — 请参阅 data_preparation↗。

5. 运行时镜像

使用预构建的 `alaudadockerhub/fine_tune_with_llamafactory:v0.1.1`，或自行构建镜像。镜像必须包含读取所选模型和数据集存储所需的工具。不需要包含大型文件模型仓库客户端。

▶ Containerfile

6. 提交微调 VolcanoJob

创建 YAML，并在 Notebook 终端中使用 `kubectl create -f vcjob_sft.yaml` 提交。（使用 JupyterLab 上传器将 `kubectl` 二进制文件放入 workbench — 镜像中不包含该文件。）

▶ VolcanoJob YAML

提交前需要修改以下内容：

- 将 `BASE_MODEL_URI`、`DATASET_URI` 和 `OUTPUT_MODEL_URI` 修改为环境所使用的受支持存储 URI。将基础模型和数据集挂载或暂存到示例所用的路径，并在作业完成后将输出目录上传到 `OUTPUT_MODEL_URI`。
- `models-cache` PVC — 提前创建。跨实验复用它，以避免重复下载基础模型。
- 共享内存 `dshm` — 多 GPU 场景至少需要 4 GiB。
- CPU / memory / GPU `requests` 和 `limits` — 与集群的 device-plugin 匹配（例如 `nvidia.com/gpu`、`nvidia.com/gpualloc`）。
- 超参数 — LLaMA-Factory YAML 内联在脚本中。将经常调整的参数提取为环境变量。

NFS workspace PVC 注意事项

如果 PVC 后端是 NFS：

- 每个可能挂载 PVC 的节点都需要 `nfs-utils`（`yum install -y nfs-utils`）。
- 在 StorageClass 上设置 `mountPermissions: "0757"`：

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ai-nfs
provisioner: nfs.csi.k8s.io
parameters:
  mountPermissions: '0757'
  server: <nfs-server-address>
  share: <nfs-export-path>
reclaimPolicy: Delete
volumeBindingMode: Immediate
mountOptions: [hard, nfsvers=4.1]
```

7. 管理作业

```
kubectl get vcjob
kubectl get vcjob <name> -o yaml
kubectl get pod && kubectl logs <pod>
kubectl describe vcjob <name>      # if pods aren't scheduling
kubectl get podgroups              # Volcano scheduling view
kubectl delete vcjob <name>
```

成功后，将合并后的目录上传到 `OUTPUT_MODEL_URI`，并将其注册为新的 Model Registry 版本。发布时选择该活动版本。

8. 实验跟踪

在 LLaMA-Factory 配置中设置 `report_to: mlflow`，并配置 `MLFLOW_TRACKING_URI` / `MLFLOW_EXPERIMENT_NAME` 环境变量，即可将指标路由到 MLflow。在 **Alauda AI** → **Tools** → **MLFlow** 中查找运行记录、比较损失曲线，并固定表现最佳的运行记录。

在安全（SSO + 多租户）的 MLflow 安装中，作业还必须进行身份验证 — 提供一个 `MLFLOW_TRACKING_TOKEN` 并选择一个 workspace。请参阅使用带身份验证和 RBAC 的 MLflow Python SDK，了解如何获取令牌并配置客户端。

9. 发布微调后的模型

> 示例使用 LoRA，并在上传输出 artifact 前将 adapter 合并到基础模型中。当前尚不支持由基础模型 + adapter 组合构成的推理服务。

1. 打开 **AI Hub > Model Registry**，将微调后的输出 URI 注册为新的模型版本，并在其元数据中设置 Task Type 和 Framework（例如 `text-generation` 和 `transformers`）。
2. 打开活动版本，然后选择 **Create Inference API**。
3. 选择与集群 CUDA 匹配的 vLLM 运行时，填写存储/资源和 GPU 设置，然后点击 **Publish**。
4. 运行后，点击 **Experience** 与模型聊天（仅当模型包含 `chat_template` 时可用）。

在非 **NVIDIA GPU** 上运行

适用于 Huawei Ascend NPU、Intel Gaudi、AMD 等设备。关于使用 `PyTorch CANN` + `MindSpeed-LLM` 的 Ascend NPU 配方，请参阅在 Ascend NPU 上微调和预训练 LLM。

一般步骤：

1. 前提条件：已部署供应商驱动程序和 Kubernetes device plugin，并且 Pod 可以看到设备。记录资源名称（例如 `huawei.com/Ascend910: "1"`）。
2. 收集供应商解决方案 — 文档、微调镜像、支持的模型、示例数据以及启动命令/参数。
3. （可选）先端到端验证供应商解决方案，以排除解决方案本身的问题。
4. （可选）将其封装为基本 **Kubernetes Job**，在加入 Volcano 之前确认 device plugin 在 K8s 下正常工作。
5. 作为 **VolcanoJob** 运行，以下 YAML 可作为起点。

▶ VolcanoJob YAML（供应商模板）

在其他设备上进行实验跟踪

LLaMA-Factory 和 Transformers 可直接与 MLflow / wandb 集成。在框架配置中设置目标位置（例如 LLaMA-Factory 的 `report_to: mlflow`），并提供 `MLFLOW_TRACKING_URI` 和 `MLFLOW_EXPERIMENT_NAME` 环境变量（在安全安装中还需提供 `MLFLOW_TRACKING_TOKEN` — 请参阅使用带身份验证和 RBAC 的 MLflow Python SDK）。在 **Alauda AI** → **Tools** → **MLFlow** 下查看结果。

在 Ascend NPU 上微调和预训练 LLM

使用基于 Workbench 的配方，在带有 Huawei Ascend NPU 的 `arm64` 节点上运行全参数 SFT 和预训练。这些 notebook 直接在附加了多个 NPU 的单个 Workbench 容器中运行训练（不使用 VolcanoJob）。

Notebook	Workbench 镜像	工作负载
`qwen3_finetune_verify.ipynb`	`PyTorch CANN`	`Qwen3-8B` 的全参数 SFT（MindSpeed-LLM）
`qwen25_pretrain_verify.ipynb`	`PyTorch CANN`	`Qwen2.5-7B` 的预训练（MindSpeed-LLM）
`qwen3_0.6b_finetune_verify.ipynb`	`MindSpore CANN`	`Qwen3-0.6B` 的全参数 SFT（捆绑 MindSpeed-Core-MS 和 MindSpeed-LLM）

三个 notebook 都采用验证优先的方式：默认使用较短的序列和较少的迭代次数，以便在扩展规模前确认运行时、模型加载、预处理和分布式启动路径。`PyTorch CANN` 镜像捆绑了 Python 3.12、CANN 8.5.0、PyTorch 2.9.0 和 `torch_npu 2.9.0`。

目录

开始之前

准备基础模型

准备数据集

微调数据（Alpaca 风格 JSONL）

预训练数据（原始文本）

运行 notebook

需要查看的参数

输出路径

注意事项

开始之前

- 已安装 Ascend 驱动、CANN 运行时和 Kubernetes device plugin；你的 Workbench 可以调度到 `arm64` Ascend 节点。
- PyTorch 示例需要至少 4 个 NPU；MindSpore notebook 针对 2× Ascend 910B 32G 进行了调优，使用 `TP=1`、`PP=1`、`MBS=2`。
- 工作区使用持久化存储，并为 HuggingFace 模型和转换后的 Megatron / MCore 权重预留足够空间。
- 网络：PyTorch notebook 运行时从 `https://gitcode.com/ascend/MindSpeed-LLM.git` 克隆 `MindSpeed-LLM`。如果 Workbench 无法访问该仓库，请将本地副本放入工作区，并更新第一个单元格中的路径。MindSpore notebook 使用 `/opt/app-root/share/MindSpeed-Core-MS` 下捆绑的目录树，不会进行克隆。

按照创建 Workbench 创建 Workbench（`PyTorch CANN` 或 `MindSpore CANN`），然后通过 JupyterLab 文件浏览器上传 notebook。

准备基础模型

每个 notebook 都需要在以下位置提供 HuggingFace 格式的模型：

Notebook	`HF_MODEL_DIR` 默认值
微调	`/opt/app-root/src/models/Qwen3-8B`

Notebook	`HF_MODEL_DIR` 默认值
预训练	`/opt/app-root/src/models/Qwen2.5-7B`
MindSpore SFT	`/opt/app-root/src/models/Qwen3-0.6B`

将模型文件放到该位置，或在第一个参数单元格中更改 `HF_MODEL_DIR`。对于有版本控制且可复用的模型，请将 artifact 存储在 S3、PVC 或 OCI registry 中。请参阅使用 Workbench 上传模型；然后在 Model Registry 中单独注册生成的 URI。

> 三个 notebook 都会在训练前将 HF 转换为 Megatron / MCore；请预留磁盘空间以存放第二个大型权重目录。

准备数据集

微调数据（**Alpaca 风格 JSONL**）

两个微调 notebook 都需要：

- `ALPACA_PARQUET = /opt/app-root/src/datasets/alpaca/train-00000-of-00001-a09b74b3ef9c3b56.parquet`
- `RAW_DATA_FILE = .../finetune_dataset/alpaca_sample.jsonl`（位于 notebook 的工作目录下）

如果 parquet 文件存在，notebook 会自动将其转换为 JSONL。如果已有 JSONL，请将其放置在 `RAW_DATA_FILE`，或更新该变量。记录格式如下：

```
{ "instruction": "...", "input": "...", "output": "..." }
```

通过更改参数单元格中的 handler，可以替换为 ShareGPT 和 Pairwise 格式。

预训练数据（原始文本）

`MindSpeed-LLM` 预处理接受 `.parquet`、`.json`、`.jsonl` 和 `.txt`。结构化格式需要 `text` 字段；纯文本则需要每行一个片段。默认路径为同一个 `ALPACA_PARQUET`；如果该路径不存在，notebook 会回退到内置示例，因此你仍可验证整个流程。

对于更大的数据集，请挂载 PVC 或从平台数据集仓库拉取——请参阅使用 Workbench 微调 LLM。

运行 notebook

每个 notebook 的流程结构相同：

1. 环境检查——确认 `torch_npu`（或 `mindspore` + `msadapter`）、`MindSpeed` 和 `MindSpeed-LLM` 可导入，并确认可用 NPU 数量与 `TP × PP` 匹配。
2. 数据集准备——将 parquet 转换为 JSONL（或在预训练中回退到内置示例）。
3. **HF → MCore** 权重转换——将权重写入特定于 `TP/PP` 的输出目录。
4. 数据预处理——生成 MindSpeed 所需的 `.bin` / `.idx` 文件。
5. 训练——SFT 使用 `posttrain_gpt.py`，预训练使用 `pretrain_gpt.py`。MindSpore notebook 使用 `msrun` 和 `--ai-framework mindspore --ckpt-format msadapter`，并将日志写入 `.../logs`。
6. 验证——检查 `latest_checkpointed_iteration.txt`，列出 `iter_*` 目录，并（对于 PyTorch SFT）运行快速推理冒烟测试。

MindSpore SFT notebook 与上游 Qwen3 路径保持一致：

- `examples/mindspore/qwen3/ckpt_convert_qwen3_hf2mcore.sh`
- `examples/mindspore/qwen3/data_convert_qwen3_instruction.sh`
- `examples/mindspore/qwen3/tune_qwen3_0point6b_4K_full_ms.sh`

默认 MindSpore 参数：`TP=1`、`PP=1`、`MBS=2`、`SEQ_LENGTH=2048`、`TRAIN_ITERS=100`、`ENABLE_THINKING=true`。

需要查看的参数

各 notebook 通用：

- `HF_MODEL_DIR`、`OUTPUT_DIR`
- `ALPACA_PARQUET` 或 `RAW_DATA_FILE`
- `TP`、`PP`（MindSpore 还包括 `MBS`）
- `SEQ_LENGTH`、`TRAIN_ITERS`
- `ENABLE_THINKING`（Qwen3）
- `MASTER_ADDR`、`MASTER_PORT`、`NNODES`、`NODE_RANK`（MindSpore 多节点）

对于实际运行，请将 `SEQ_LENGTH` 提高到模型的上下文窗口，将 `TRAIN_ITERS` 提高到生产值，并调整并行度 / batch size 以适应可用 NPU 数量和数据集大小。如果更改 `TP` 或 `PP`，请重新运行权重转换，使 checkpoint 布局保持匹配。

输出路径

Notebook	默认输出
`qwen3_finetune_verify.ipynb`	`/opt/app-root/src/Qwen3-8B-work-dir/output/qwen3_8b_finetuned`
`qwen25_pretrain_verify.ipynb`	`/opt/app-root/src/Qwen2.5-7B-work-dir/output/qwen25_7b_pretrained`
`qwen3_0.6b_finetune_verify.ipynb`	`/opt/app-root/src/Qwen3-0.6B-work-dir/output/qwen3_0.6b_finetuned`

请将这些路径保存在持久化存储上。训练后，使用使用 Workbench 上传模型将结果上传到 S3、PVC 或 OCI registry，然后使用 Model Registry 将其 Model URI 注册为新的 Model Registry 版本。

注意事项

- notebook 运行的是全参数 **SFT**，而非 LoRA。请将默认值视为冒烟测试配置，在正式运行前进行调优。

- MindSpore notebook 仅验证 checkpoint 生成，不包含稳定的推理步骤。
- 对于离线或受限集群：请在工作区预先准备 `MindSpeed-LLM` 仓库以及模型 / 数据集文件（PyTorch notebook）。MindSpore notebook 仅使用捆绑的源代码树。

配额与调度

配置 RBAC

1. 为批处理管理员授予权限
2. 为批处理用户授予权限

配置配额

1. 配置集群队列
2. 配置资源风味
3. 配置本地队列

使用 cohort

在 cluster queu

Gang 调度

配置 Gang 调度

共享

使用 Kueue 管理 Ascend NPU 配额

前提条件

操作步骤

注意事项

配置 RBAC

当你安装 Alauda Build of Kueue 时，会为我们假定会与 Kueue 交互的两个主要角色创建以下一组 ClusterRole：

- **kueue-batch-admin-role** 包含管理 ClusterQueues、Queues、Workloads 和 ResourceFlavors 的权限。
- **kueue-batch-user-role** 包含管理 Jobs 以及查看 Queues 和 Workloads 的权限。

目录

1. 为批处理管理员授予权限
2. 为批处理用户授予权限

1. 为批处理管理员授予权限

批处理管理员通常需要在所有命名空间中拥有 `kueue-batch-admin-role` ClusterRole。

要将 `kueue-batch-admin-role` 角色绑定到批处理管理员（表示为用户 `admin@cpaas.com`），请创建一个 ClusterRoleBinding，其清单类似于以下内容：

```yaml
# batch-admin-role-binding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kueue-admins
subjects:
- kind: User
  name: admin@cpaas.com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: kueue-batch-admin-role
  apiGroup: rbac.authorization.k8s.io
```

要创建 ClusterRoleBinding，请保存前面的清单并运行以下命令：

```bash
kubectl apply -f batch-admin-role-binding.yaml
```

2. 为批处理用户授予权限

批处理用户通常需要以下权限：

- 在其命名空间中创建和查看 Jobs。
- 查看其命名空间中可用的队列。
- 查看其命名空间中 Workloads 的状态。

要为命名空间 `team-a` 中的用户 `team-a-owner@cpaas.com` 授予这些权限，请创建一个 RoleBinding，其清单类似于以下内容：

```yaml
# team-a-batch-user-role-binding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: kueue-users
  namespace: team-a
subjects:
- kind: User
  name: team-a-owner@cpaas.com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: kueue-batch-user-role
  apiGroup: rbac.authorization.k8s.io
```

要创建 RoleBinding，请保存前面的清单并运行以下命令：

```
kubectl apply -f team-a-batch-user-role-binding.yaml
```

配置配额

作为管理员，你可以使用 Alauda Build of Kueue 配置配额，以优化用户工作负载的资源分配和系统吞吐量。你可以为 CPU、内存、pod 和 GPU 等计算资源配置配额。

你可以通过完成以下步骤在 Alauda Build of Kueue 中配置配额：

1. 配置集群队列。
2. 配置资源风味。
3. 配置本地队列。
4. 然后，用户可以将其工作负载提交到本地队列。

目录

1. 配置集群队列
   1.1. 前提条件
   1.2. 操作步骤
2. 配置资源风味
   2.1. 前提条件
   2.2. 操作步骤
3. 配置本地队列
   3.1. 前提条件
   3.2. 操作步骤
4. 配置默认本地队列
   4.1. 前提条件

4.2. 操作步骤

4.3. 验证

---

1. 配置集群队列

集群队列是一种集群范围资源，由 ClusterQueue 对象表示，用于管理 GPU、CPU、内存和 pod 等资源池。集群队列可用于定义使用限制、资源风味的配额、消费顺序以及公平共享规则。

> **INFO**
>
> 注意：在配置 **ResourceFlavor** 对象之前，集群队列不能用于使用。

1.1. 前提条件

- Alauda Container Platform Web CLI 已与集群通信。
- 你拥有集群管理员权限或 `kueue-batch-admin-role` 角色。

1.2. 操作步骤

1. 将 ClusterQueue 对象创建为 YAML 文件：

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: cluster-queue
spec:
  namespaceSelector: {} (1)
  #matchLabels:
  #  kubernetes.io/metadata.name: team-a
  resourceGroups:
  - coveredResources: ["cpu", "memory", "pods"] (2)
    flavors:
    - name: "default-flavor" (3)
      resources: (4)
      - name: "cpu"
        nominalQuota: 9
      - name: "memory"
        nominalQuota: 36Gi
      - name: "pods"
        nominalQuota: 5
  - coveredResources: ["nvidia.com/gpualloc", "nvidia.com/total-gpucore
s", "nvidia.com/total-gpumem"] (5)
    flavors:
    - name: "t4-flavor" (6)
      resources:
      - name: "nvidia.com/gpualloc"
        nominalQuota: "20"
      - name: "nvidia.com/total-gpucores"
        nominalQuota: "300"
      - name: "nvidia.com/total-gpumem"
        nominalQuota: "20480"
  - coveredResources: ["nvidia.com/gpu"] (7)
    flavors:
    - name: "a30-flavor" (8)
      resources:
      - name: "nvidia.com/gpu"
        nominalQuota: 100
```

(1) `namespaceSelector`：定义哪些命名空间可以使用由该集群队列管理的资源。示例中的 `namespaceSelector` 为空，这表示所有命名空间都可以使用这些资源。

(2) `default-flavor` 的 `coveredResources`：定义由集群队列管理的资源类型。此示例中的 `ClusterQueue` 对象管理 CPU、内存、pod 和 GPU 资源。

3 `default-flavor` 的 `flavors.name`：定义应用于所列资源类型的资源风味。在此示例中，`default-flavor` 资源风味应用于 CPU、内存、pod 和 GPU 资源。

4 `default-flavor` 的 `resources`：定义接纳作业所需的资源要求。此示例中的集群队列仅在满足以下条件时接纳作业：

- CPU 请求总和小于或等于 9。
- 内存请求总和小于或等于 36Gi。
- pod 总数小于或等于 5。
- GPU task 总和小于或等于 20（如果使用 Alauda Build of Hami，参见注释 5）。
- GPU 核心请求总和小于或等于 300（如果使用 Alauda Build of Hami）。
- GPU 内存请求总和小于或等于 20480。
- GPU 请求总和小于或等于 100（如果使用 Alauda Build of NVIDIA GPU Device Plugin，参见注释 7）。

5 `t4-flavor` 的 `coveredResources`：定义 Alauda Build of Hami 的资源要求。如果你不使用 Alauda Build of Hami，请将其删除。

6 `t4-flavor` 的 `flavors.name`：定义应用于所列资源类型的资源风味。在此示例中，`t4-flavor` 资源风味应用于 Nvidia T4 GPU 卡。如果你不想为特定卡类型配置配额，可以填入 `default-flavor`。

7 `a30-flavor` 的 `coveredResources`：定义 Alauda Build of NVIDIA GPU Device Plugin 的资源要求。如果你不使用 Alauda Build of NVIDIA GPU Device Plugin，请将其删除。

8 `a30-flavor` 的 `flavors.name`：定义应用于所列资源类型的资源风味。在此示例中，`a30-flavor` 资源风味应用于 Nvidia A30 GPU 卡。如果你不想为特定卡类型配置配额，可以填入 `default-flavor`。

2. 运行以下命令来应用 `ClusterQueue` 对象：

```
kubectl  apply -f <filename>.yaml
```

2. 配置资源风味

配置好 ClusterQueue 对象后，你可以配置 ResourceFlavor 对象。

集群中的资源通常并不同质。如果集群中的资源是同质的，你可以使用空的 ResourceFlavor，而不必向自定义资源风味添加标签。

你可以使用自定义 ResourceFlavor 对象，通过标签、taint 和 toleration 来表示与集群节点关联的不同资源变体。然后，你可以将工作负载与特定节点类型关联，以实现细粒度的资源管理。

2.1. 前提条件

- Alauda Container Platform Web CLI 已与集群通信。
- 你拥有集群管理员权限或 `kueue-batch-admin-role` 角色。

2.2. 操作步骤

1. 将 ResourceFlavor 对象创建为 YAML 文件：
   空 **ResourceFlavor** 对象示例

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: default-flavor
```

   Nvidia Tesla T4 GPU 的自定义 **ResourceFlavor** 对象示例

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: "t4-flavor"
spec:
  nodeLabels:
    nvidia.com/gpu.product: Tesla-T4
```

   Nvidia A30 GPU 的自定义 **ResourceFlavor** 对象示例

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: "a30-flavor"
spec:
  nodeLabels:
    nvidia.com/gpu.product: NVIDIA-A30
```

2. 运行以下命令来应用 ResourceFlavor 对象：

```
kubectl apply -f <filename>.yaml
```

3. 配置本地队列

本地队列是一个命名空间级对象，由 LocalQueue 对象表示，用于将属于同一命名空间的紧密相关工作负载分组。

作为管理员，你可以配置 LocalQueue 对象指向一个集群队列。这样会将集群队列中的资源分配给 LocalQueue 对象中指定命名空间内的工作负载。

3.1. 前提条件

- Alauda Container Platform Web CLI 已与集群通信。
- 你拥有集群管理员权限或 `kueue-batch-admin-role` 角色。
- 你已创建 `ClusterQueue` 对象。

3.2. 操作步骤

1. 将 LocalQueue 对象创建为 YAML 文件：
   基本 **LocalQueue** 对象示例

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  namespace: team-namespace
  name: user-queue
spec:
  clusterQueue: cluster-queue
```

2. 运行以下命令来应用 LocalQueue 对象：

```
kubectl apply -f <filename>.yaml
```

4. 配置默认本地队列

作为集群管理员，你可以通过管理选定命名空间中的所有作业来改进集群中的配额强制执行，而无需显式为每个作业添加标签。你可以通过创建默认本地队列来实现这一点。

默认本地队列将作为新建且未带有 `kueue.x-k8s.io/queue-name` 标签的作业的本地队列。创建默认本地队列后，在该命名空间中创建的任何不带 `kueue.x-k8s.io/queue-name` 标签的新作业，都会自动更新为带有 `kueue.x-k8s.io/queue-name: default` 标签。

4.1. 前提条件

- Alauda Container Platform Web CLI 已与集群通信。
- 你拥有集群管理员权限或 `kueue-batch-admin-role` 角色。
- 你已创建 ClusterQueue 对象。

4.2. 操作步骤

1. 将名为 default 的 LocalQueue 对象创建为 YAML 文件：
   默认 **LocalQueue** 对象示例

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  namespace: team-namespace
  name: default
spec:
  clusterQueue: cluster-queue
```

2. 运行以下命令来应用 `LocalQueue` 对象：

```
kubectl apply -f <filename>.yaml
```

4.3. 验证

1. 在与默认本地队列相同的命名空间中创建一个作业。
2. 观察该作业已更新为带有 `kueue.x-k8s.io/queue-name: default` 标签。

使用 cohort

你可以使用 cohort 对集群队列进行分组，并确定哪些集群队列能够彼此共享可借用资源。可借用资源定义为 cohort 中所有集群队列的未使用标称配额。

使用 cohort 可以通过防止资源利用不足并启用公平共享配置，帮助优化资源利用率。cohort 还可以帮助简化团队之间的资源管理和分配，因为你可以将集群队列按相关工作负载或按团队进行分组。你也可以使用 cohort 在组级别设置资源配额，以定义一组集群队列可消耗的资源上限。

目录

在 cluster queue spec 中配置 cohort

在 cluster queue spec 中配置 cohort

你可以通过在 `ClusterQueue` 对象的 `.spec.cohortName` 字段中指定 cohort 名称，将 cluster queue 添加到某个 cohort 中，如下例所示：

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: cluster-queue
spec:
# ...
  cohortName: example-cohort
# ...
```

所有 `spec.cohortName` 匹配的 cluster queue 都属于同一个 cohort。

如果省略 `spec.cohortName` 字段，则该 cluster queue 不属于任何 cohort，且无法访问可借用资源。

配置公平共享

公平共享是一种抢占策略，用于在一个 cohort 的各个租户之间实现可借用资源的均等或加权份额。可借用资源是一个 cohort 中所有 ClusterQueue 未使用的名义配额。

目录

ClusterQueue 权重

ClusterQueue 权重

份额值在 ClusterQueue 对象中以权重值表示。
份额值非常重要，因为它允许管理员对特定作业类型或团队进行优先级排序。可以为关键应用或高优先级团队配置加权值，使其获得按比例更大的可用资源份额。配置权重可确保未使用的资源根据定义好的组织或项目优先级进行分配，而不是按先到先服务的方式分配。

权重值，也称份额值，定义了 ClusterQueue 在竞争可借用资源时的相对优势。一般来说，Alauda Build of Kueue 会优先接纳份额值较低的作业。份额值较高的作业比份额值较低的作业更有可能先被抢占。

配置了公平共享权重的示例 **ClusterQueue**

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: cluster-queue
spec:
  namespaceSelector: {}
  resourceGroups:
  - coveredResources: ["cpu"]
    flavors:
    - name: default-flavor
      resources:
      - name: cpu
        nominalQuota: 9
  cohortName: example-cohort
  fairSharing:
    weight: 2
```

如果不设置权重，默认值为 1。

权重值为 0 表示无限份额值。这意味着与其他队列相比，该 ClusterQueue 始终处于劣势，并且在启用公平共享时，其工作负载总是最先被抢占。

Gang 调度

Gang 调度是一种基于超时的 All-or-nothing 调度实现，适用于 Alauda Build of Kueue。

Gang 调度可确保一组相关作业只有在所有必需资源都可用时才会启动。Alauda Build of Kueue 会通过挂起作业来启用 Gang 调度，直到 Alauda Container Platform 集群能够保证有足够的容量，同时启动并执行该 gang 中所有相关作业。

如果你正在使用昂贵且有限的资源，例如 GPU，Gang 调度就非常重要。Gang 调度可以防止作业申请了 GPU 但未实际使用，从而提高 GPU 利用率并降低运行成本。Gang 调度还可以帮助避免资源碎片化和死锁等问题。

目录

配置 Gang 调度

配置 Gang 调度

Gang 调度默认已启用。作为集群管理员，你可以通过修改 `Alauda Build of Kueue` 集群插件的部署表单参数来更新超时时间或禁用 Gang 调度。

使用 Kueue 管理 Ascend NPU 配额

本页介绍如何使用 Alauda Build of Kueue 来管理由 Ascend Device Plugin（作为 NPU Operator 的一部分提供）暴露的 Ascend NPU 资源配额。Kueue 将 `huawei.com/Ascend910` 以及类似的 device-plugin 资源视为普通的可计数资源，因此用于 CPU、内存和 Nvidia GPU 的相同 `ClusterQueue` / `ResourceFlavor` / `LocalQueue` 模型可以直接沿用。

目录

前提条件

操作步骤

1. 识别 NPU 节点标签
2. 创建 Kueue 对象
3. 通过队列提交 NPU Job
4. 观察准入

注意事项

前提条件

- 你已经安装了 `Alauda Build of Kueue` 集群插件。
- 你已经安装了 `Alauda Build of NPU Operator` 集群插件，并在部署表单中启用了 **Driver**、**Ascend Device Plugin** 和 **Ascend Docker Runtime** 组件（默认情况下它们是启用的）。这三个开关决定了集群是否会将 `huawei.com/Ascend910`（或

`huawei.com/Ascend310P`）作为 Ascend 节点上的可调度扩展资源暴露出来。有关完整操作步骤以及各部署表单开关的含义，请参见 安装 NPU Operator↗ 。

- Ascend Device Plugin 正在报告节点上的 NPU 容量，例如 `kubectl describe node <ascend-node>` 在 `Capacity` 和 `Allocatable` 下都显示 `huawei.com/Ascend910: <N>`。
- Alauda Container Platform Web CLI 已与集群建立通信。

操作步骤

1. 识别 NPU 节点标签

`ResourceFlavor` 通过 `spec.nodeLabels` 选择配额适用的节点池。NPU Operator 会为其管理的每个节点添加以下标签——请选择最适合你的集群的标签：

标签	示例	适用场景
`accelerator`	`huawei-Ascend910`	所有 Ascend 910 节点共享同一个 flavor
`accelerator-type`	`module-910b-8`	需要区分不同外形规格时
`node.kubernetes.io/npu.chip.name`	`910B4`	需要区分不同芯片代际时

本页后续示例使用 `accelerator: huawei-Ascend910`。

2. 创建 Kueue 对象

创建一个绑定到 NPU 节点池的 `ResourceFlavor`、一个声明 NPU 配额的 `ClusterQueue`，以及一个供用户提交 Job 的命名空间中的 `LocalQueue`。

```yaml
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: ascend910-flavor
spec:
  nodeLabels:
    accelerator: huawei-Ascend910 (1)
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: ascend910-cq
spec:
  namespaceSelector: {}
  resourceGroups:
  - coveredResources: ["huawei.com/Ascend910", "cpu", "memory"] (2)
    flavors:
    - name: ascend910-flavor
      resources:
      - name: "huawei.com/Ascend910"
        nominalQuota: 2 (3)
      - name: "cpu"
        nominalQuota: 8
      - name: "memory"
        nominalQuota: 16Gi
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  namespace: team-ascend
  name: ascend910-lq
spec:
  clusterQueue: ascend910-cq
```

(1) `nodeLabels`：将该 flavor 限制为由 NPU Operator 标记为 Ascend 910 的节点。请替换为与你的集群相匹配的标签（参见上表）。

(2) `coveredResources`：将 device-plugin 资源与 CPU 和内存一起列出，以便在准入决策中对三者进行统一计量。

(3) `nominalQuota`：队列可在任意时刻发放的 NPU 卡总数。请将其设置为等于或低于该 flavor 所选择节点上的可分配 NPU 之和。

使用 `kubectl apply -f <filename>.yaml` 应用该清单。

3. 通过队列提交 NPU Job

添加 `kueue.x-k8s.io/queue-name` 标签，并在容器的 `resources` 块中请求 NPU。Kueue 内置的 `batch/job` 集成会以挂起状态创建 Job，并且只会在配额检查通过后解除挂起。

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: ascend-train
  namespace: team-ascend
  labels:
    kueue.x-k8s.io/queue-name: ascend910-lq (1)
spec:
  parallelism: 1
  completions: 1
  suspend: true (2)
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: worker
        image: <your-ascend-image>
        command: ["sh", "-c", "echo training; sleep 600"]
        resources:
          requests: { cpu: 100m, memory: 256Mi, "huawei.com/Ascend910": 2
} (3)
          limits:   { cpu: "1",  memory: 1Gi,   "huawei.com/Ascend910": 2
}
```

(1) `queue-name` 标签：将 Job 路由到你上面创建的 `LocalQueue`。

(2) `suspend: true`：这是 Kueue 暂存 Job 直到配额可用所必需的。准入后，Kueue 会将其改回 `false`。

(3) NPU 请求：使用 `huawei.com/Ascend910`（或 `huawei.com/Ascend910B4` 等），以匹配节点上的 Ascend Device Plugin 报告的资源名称。request 和 limit 必须相等——这是一个扩展资源。

4. 观察准入

向一个 `nominalQuota` 为 2 的队列提交两个这样的 Job，每个 Job 请求 2 个 NPU。

```
kubectl -n team-ascend get jobs,workloads
```

第一个 Job 会被准入，并以 Pod 的形式运行；该 Pod 会从 Ascend Device Plugin 获得 `/dev/davinci*` 设备和 `ASCEND_VISIBLE_DEVICES`。第二个 Job 会保持 `Suspended`，其 `Workload` 会报告配额不足：

```
NAME                STATUS      COMPLETIONS   DURATION   AGE
job/ascend-train    Running     0/1           5s         12s
job/ascend-train-2  Suspended   0/1                      12s

NAME                                    QUEUE          RESERVED IN
ADMITTED
workload/job-ascend-train-xxxxx         ascend910-lq   ascend910-cq
True
workload/job-ascend-train-2-yyyyy       ascend910-lq
```

查看被保留的 Workload 可以更明确地看到原因：

```
kubectl -n team-ascend get workload job-ascend-train-2-yyyyy \
  -o yaml
```

`status.conditions` 字段会报告配额不足：

```
status:
  conditions:
  - type: QuotaReserved
    status: "False"
    reason: Pending
    message: "couldn't assign flavors to pod set main: insufficient unuse
d quota for huawei.com/Ascend910 in flavor ascend910-flavor, 2 more neede
d"
```

当第一个 Job 完成或被删除后，Kueue 会自动准入被保留的 Workload，解除第二个 Job 的挂起状态，并由 Ascend Device Plugin 将释放的 NPU 设备分配给其 Pod。

注意事项

- `ClusterQueue` 中针对 `huawei.com/Ascend910` 的 `nominalQuota` 是独立于节点 `Allocatable` 进行强制限制的。如果 `Allocatable` 低于该配额（例如因为 device plugin 将某些芯片标记为不健康），即使 Pod 已被准入，也仍可能在 kube-scheduler 层处于 `Pending`，并显示 `Insufficient huawei.com/Ascend910`。请确保 `nominalQuota` ≤ 所选节点上的 `Allocatable` 之和。
- 相同模式也适用于 device plugin 暴露的其他 Ascend 资源名称（`huawei.com/Ascend910B4`、`huawei.com/Ascend310P` 等）。按需将它们添加到 `coveredResources` 和容器的 `resources` 块中。
- Kueue 的其他功能——公平共享、cohort、gang scheduling、preemption——都可以直接用于 NPU 资源，无需额外配置。

组件

本章所依赖组件的介绍和安装指南。

Kueue

Alauda Build of Kueue

安装 Kueue

下载集群插件

上传集群插件

安装 Alauda Build of Kueue

升级 Alauda Build of Kueue

Volcano

Alauda 对 Volcano 的支持

Volcano

文档

安装 Volcano

下载集群插件

上传集群插件

安装 Volcano

升级 Volcano

卸载 Volcano

JobSet

JobSet 的 Alauda Build

JobSet

文档

安装 JobSet

上架软件包

安装 operator

创建 JobSetOperatorCtl 实例

后续步骤

快速开始

前提条件

JobSet CR 概览

运行一个简单演

检查资源

验证完成

清理

后续步骤

Alauda 的 Kueue 构建版

Alauda Build of Kueue

安装 Kueue

下载集群插件

上传集群插件

安装 Alauda Build of Kueue

升级 Alauda Build of Kueue

Alauda Build of Kueue

Alauda Build of Kueue 是一个 Kubernetes 原生系统，用于管理配额以及作业如何消耗这些配额。Alauda Build of Kueue 决定作业何时应等待、何时应被准入以启动（即可以创建 Pod），以及何时应被抢占（即应删除活动 Pod）。

Alauda Build of Kueue 不会替换 Kubernetes 集群中的任何现有组件，而是与现有的 Kubernetes API server、scheduler 和 cluster autoscaler 组件集成。

Alauda Build of Kueue 支持 all-or-nothing 语义。这意味着，要么一个作业及其所有组件整体被准入到集群中，要么如果它无法适配集群，则整个作业被拒绝。

安装 Kueue

目录

下载集群插件

上传集群插件

安装 Alauda Build of Kueue

升级 Alauda Build of Kueue

下载集群插件

INFO

可从 customer portal 获取 `Alauda Build of Kueue` 集群插件。

如需更多信息，请联系 Consumer Support。

上传集群插件

有关上传集群插件的更多信息，请参阅 上传集群插件↗

安装 Alauda Build of Kueue

1. 转到 `管理员` -> `Marketplace` -> `Cluster Plugin` 页面，切换到目标集群，然后部署 `Alauda Build of Kueue` 集群插件。

   > **INFO**
   >
   > 注意：部署表单参数可以保持默认值，也可以在了解其用法后进行修改。

2. 验证结果。您可以在 UI 中看到状态为 "Installed"，或者检查 pod 状态：

   ```
   kubectl get pods -n cpaas-system | grep "kueue"
   ```

升级 Alauda Build of Kueue

1. 将 **Alauda Build of Kueue** 插件的新版本包上传到 ACP。
2. 转到 `管理员` -> `Clusters` -> `Target Cluster` -> `Functional Components` 页面，然后单击 `Upgrade` 按钮，您将看到 `Alauda Build of Kueue` 可以升级。

Alauda 对 Volcano 的支持

Alauda 对 Volcano 的支持

Volcano

文档

安装 Volcano

下载集群插件

上传集群插件

安装 Volcano

升级 Volcano

卸载 Volcano

Alauda 对 Volcano 的支持

目录

Volcano

文档

Volcano

Alauda 对 **Volcano** 的支持 将 Volcano↗ CNCF 项目打包为 Alauda Container Platform 的集群插件。Volcano 是面向 Kubernetes 的云原生批处理系统，为机器学习、大数据和 HPC 工作负载提供调度和资源管理能力。

主要组件和功能包括：

- **Volcano scheduler**：一种面向批处理的 scheduler，用于补充默认的 Kubernetes scheduler。它支持 gang scheduling、fair-share、binpack 以及分布式训练和批处理工作负载所需的其他策略。
- **VolcanoJob (`batch.volcano.sh/v1alpha1`)**：一种 job API，用于将具有不同角色（例如 master 和 worker）的任务分组，并将它们作为单个单元进行调度，从而使多 pod 训练任务要么同时启动，要么完全不启动。
- **PodGroup (`scheduling.volcano.sh/v1beta1`)**：一种分组原语，scheduler 使用它来跟踪属于同一逻辑 job 的 pods 之间的 quota、queue 和 gang-scheduling 约束。

- **Queue (`scheduling.volcano.sh/v1beta1`)**：一种资源 queue，用于按优先级、权重和容量组织 jobs，从而实现租户感知的集群资源共享。
- **JobFlow (`flow.volcano.sh/v1alpha1`)**：对多个具有依赖关系的 VolcanoJobs 进行声明式编排，适用于包含预处理和训练等多个阶段的流水线。

有关在平台上安装的信息，请参见 安装 Volcano。

文档

Volcano 上游文档和相关资源：

- **Volcano** 文档：https://volcano.sh/↗ — 官方文档，涵盖概念、scheduler 插件和使用指南。
- **Volcano GitHub**：https://github.com/volcano-sh/volcano↗ — Volcano 项目的源代码、API 参考和示例。
- **Kubeflow** 集成：当将 **Kubeflow Trainer** 与 Volcano scheduler 一起使用时，请在部署训练组件之前安装 **Alauda** 对 **Volcano** 的支持，并在训练组件的配置中启用 Volcano scheduler 选项。

安装 Volcano

目录

下载集群插件
上传集群插件
安装 Volcano
升级 Volcano
卸载 Volcano

下载集群插件

INFO

`Volcano` 集群插件可从 customer portal 获取。

如需更多信息，请联系 Consumer Support。

上传集群插件

有关上传集群插件的更多信息，请参阅 上传集群插件↗

安装 Volcano

1. 前往 `管理员` -> `Marketplace` -> `集群插件` 页面，切换到目标集群，然后部署 `Volcano` 集群插件。

> **INFO**
>
> 注意：部署表单参数可以保持默认值，也可以在了解其用法后进行修改。

2. 验证结果。您可以在 UI 中看到 "Installed" 状态，也可以检查 pod 状态：

```
kubectl get pods -n kubeflow | grep volcano
```

升级 Volcano

1. 将 **Volcano** 插件包的新版本上传到 ACP。
2. 前往 `管理员` -> `Clusters` -> `目标集群` -> `功能组件` 页面，然后点击 `升级` 按钮，您会看到 `Volcano` 可升级。

卸载 Volcano

在移除 `Volcano` 集群插件之前，请手动删除在安装该插件期间创建的任何 `VolcanoJob`、`PodGroup`、`Queue` 和 `JobFlow` 资源。移除插件不会为您清理这些自定义资源。

Alauda 的 JobSet 构建

JobSet 的 Alauda Build

JobSet

文档

安装 JobSet

上架软件包

安装 operator

创建 JobSetOperatorCtl 实例

后续步骤

快速开始

前提条件

JobSet CR 概览

运行一个简单演

检查资源

验证完成

清理

后续步骤

JobSet 的 Alauda Build

目录

JobSet

文档

JobSet

JobSet 的 **Alauda Build** 基于 JobSet↗ Kubernetes SIG 项目。
JobSet 是一个面向 Kubernetes 的原生 API，用于将一组 Kubernetes Jobs↗ 作为单个单元进行管理。它为在 Kubernetes 上部署 HPC（例如 MPI）和 AI/ML 训练工作负载（PyTorch、JAX、TensorFlow 等）提供统一的 API。

主要组件和能力包括：

- **JobSet CRD**：核心 API 资源（`jobset.x-k8s.io/v1alpha2`、`kind: JobSet`），用于定义一组 `ReplicatedJobs`。每个 `ReplicatedJob` 都是一个 Job 模板，控制器会将其实例化为一个或多个 Kubernetes Jobs，从而允许不同的 Pod 模板（leader、workers、parameter servers 等）在同一个工作负载中共存。
- 多模板 **Jobs**：可以在单个资源中建模不同的 Pod 组，因此像 leader/worker、driver/worker 或 parameter-server/worker 这类工作负载不再需要通过手动协调多个顶层 Jobs 来实现。
- 自动 **Headless Service** 和稳定主机名：JobSet 会自动配置 headless Service，并使用 IndexedJobs 为每个 Pod 提供稳定的 DNS 主机名，为分布式训练框架（PyTorch DDP、

Horovod、JAX、TensorFlow、MPI 等）提供可预测的网络标识。

- 可配置的失败与成功策略：失败策略控制 JobSet 重启的次数以及不同失败类型的处理方式；当目标子集的 `ReplicatedJobs` 成功时，成功策略会将 JobSet 声明为完成，从而在有意义的工作完成后即可释放资源。
- 启动顺序：可以配置启动策略，使 worker Jobs 在 leader（driver）Job 就绪后再启动，支持 leader/worker 模式，其中主进程必须先初始化，worker 才能连接。
- 独占拓扑放置：通过注解，JobSet 可以强制子 Job 与拓扑域（例如机架或可用区）之间建立 1:1 映射，使子 Job 在该域内独占计算资源，以实现性能隔离。
- 快速故障恢复：发生故障时，JobSet 可以快速重建子 Job。如果工作负载支持 checkpointing，则可在重启后从上次保存的 checkpoint 继续执行。reconciler 经过优化，即使在大规模场景下也能尽量减少对调度吞吐量的影响。
- 与 **Kueue** 集成：JobSet 可与 Kueue 集成，用于批处理 AI/ML 工作负载的排队、多租户和资源共享。

在平台上安装，请参见 安装 JobSet。

文档

JobSet 上游文档和相关资源：

- **JobSet** 文档：https://jobset.sigs.k8s.io/↗ — 涵盖概念、API 参考和使用指南的官方文档。
- **JobSet GitHub**：https://github.com/kubernetes-sigs/jobset↗ — JobSet Kubernetes SIG 项目的源代码、API 参考和示例。
- **Kueue**（**Alauda Build**）：../kueue/intro — JobSet 与 Kueue 集成，用于批处理工作负载的排队和资源管理。

安装 JobSet

本页面介绍如何通过 Operator Hub 在 Alauda Container Platform 上安装 **Alauda Build of JobSet**。JobSet 以基于 Helm 的 operator 形式提供：从 Operator Hub 安装后会部署 `jobset-operator` controller；随后创建 `JobSetOperatorCtl` 实例会部署 JobSet controller 和 webhook，用于协调 `JobSet` 自定义资源。

目录

上架软件包

安装 operator

创建 JobSetOperatorCtl 实例

后续步骤

上架软件包

下载 JobSet Operator 软件包，然后按照 上架软件包 中的说明上传该软件包。

安装 operator

在 管理员 视图中：

1. 转到 `Marketplace` / `Operator Hub`。

2. 选择目标集群。
3. 搜索 **Alauda Build of JobSet**（Marketplace 中的软件包名称为 `jobset-operator`）。
4. 单击 **Install**。
5. 保持默认安装设置。operator 默认安装到 `jobset-system` 命名空间。
6. 完成安装。

确认 operator controller 正在运行：

```
kubectl get pods -n jobset-system
```

`jobset-operator` pod 应处于 `Running` 状态，例如：

```
NAME                                READY   STATUS    RESTARTS   AGE
jobset-operator-xxxxxxxxxx-xxxxx    1/1     Running   0          1m
```

创建 JobSetOperatorCtl 实例

仅安装 operator 并不会部署 JobSet controller——operator 只会协调 `JobSetOperatorCtl` 资源，并且只有在该资源存在时才会部署 JobSet controller 和 webhook。

将以下内容保存为 `jobsetoperatorctl.yaml`：

```yaml
apiVersion: jobset-operator.alauda.io/v1
kind: JobSetOperatorCtl
metadata:
  name: jobset
  namespace: jobset-system
spec:
  controller:
    replicas: 1
    leaderElection:
      enable: true
  certManager:
    enable: true
  prometheus:
    enable: false
```

应用该文件：

```bash
kubectl apply -f jobsetoperatorctl.yaml
```

在 operator 协调该资源后，JobSet controller pod 会部署到同一个命名空间中：

```bash
kubectl get pods -n jobset-system
```

预期输出如下（operator pod 和 JobSet controller pod 并排显示）：

```
NAME                                  READY   STATUS    RESTARTS   AGE
jobset-operator-xxxxxxxxxx-xxxxx      1/1     Running   0          5m
jobset-controller-xxxxxxxxxx-xxxxx    1/1     Running   0          1m
```

`JobSet` CRD 由 operator 注册，可通过以下命令进行验证：

```bash
kubectl get crd jobsets.jobset.x-k8s.io
```

后续步骤

当 JobSet controller 运行后，即可提交 `JobSet` 自定义资源来运行分布式 AI/ML 和 HPC 工作负载。

继续阅读 快速开始，运行一个简单的 JobSet 演示。

快速开始

本页将通过一个最小的 JobSet 演示来帮助你确认 operator 是否正常工作，并介绍 `JobSet` 自定义资源中最常用的字段。

目录

前提条件

JobSet CR 概览

运行一个简单演示

检查资源

验证完成

清理

后续步骤

前提条件

- 已安装 JobSet Operator，且 `jobset-system` 中的 controller pod 处于 `Running` 状态。请参阅 安装 JobSet。
- 可使用 `kubectl` 访问目标集群。
- 一个用于运行演示的 namespace。以下示例使用 `default` namespace；如有需要请替换。

JobSet CR 概览

`JobSet` CR 中最常用的字段如下：

字段	作用	常见用途
`spec.replicatedJobs`	用于实例化的 Job 模板列表	必填。每一项定义一个带名称的 Job 组（例如 `leader`、`workers`），并拥有各自的 pod template。
`replicatedJobs[].name`	组名称	用于 DNS 主机名，并作为成功/失败策略的目标。
`replicatedJobs[].replicas`	要为该组创建多少个 Job	默认值为 `1`。每个副本都是一个独立的 Kubernetes Job。
`replicatedJobs[].template`	内嵌的 `batchv1.JobSpec`	标准 Kubernetes Job spec，包括 `parallelism`、`completions`、`backoffLimit` 以及 pod template。
`spec.successPolicy`	何时将 JobSet 视为成功	`operator: All` 或 `Any`，并通过 `targetReplicatedJobs` 将策略限定到特定组。
`spec.failurePolicy`	失败时的重启行为	`maxRestarts` 控制在标记为失败之前，JobSet 可被重新创建的次数。
`spec.network`	用于稳定 pod 主机名的无头 Service 配置	通常默认值已足够；也可以通过 subdomain 进行自定义，或禁用默认 Service。
`spec.startupPolicy`	`replicatedJobs` 之间可选的启动顺序	使用 `InOrder` 依次启动各组，例如先等待 leader 再启动 workers。

运行一个简单演示

下面的示例定义了一个包含两个组的 JobSet：

- `leader`：一个 Job，包含一个 pod，用于模拟 driver 进程。
- `workers`：一个 Job，包含四个并行 pod，用于模拟 workers。

当 `leader` 和 `workers` 都完成后，JobSet 即被标记为成功。

将清单保存为 `jobset-quickstart.yaml`：

```yaml
apiVersion: jobset.x-k8s.io/v1alpha2
kind: JobSet
metadata:
  name: jobset-quickstart
  namespace: default
spec:
  successPolicy:
    operator: All
  replicatedJobs:
  - name: leader
    replicas: 1
    template:
      spec:
        parallelism: 1
        completions: 1
        backoffLimit: 0
        template:
          spec:
            restartPolicy: Never
            containers:
            - name: leader
              image: busybox:1.36
              command:
              - sh
              - -c
              - |
                echo "leader $(hostname) starting"
                sleep 30
                echo "leader done"
  - name: workers
    replicas: 1
    template:
      spec:
        parallelism: 4
        completions: 4
        backoffLimit: 0
        template:
          spec:
            restartPolicy: Never
            containers:
            - name: worker
              image: busybox:1.36
              command:
```

```
            - sh
            - -c
            - |
              echo "worker $(hostname) starting"
              sleep 30
              echo "worker done"
```

应用它：

```
kubectl apply -f jobset-quickstart.yaml
```

检查资源

确认 JobSet 已被接受，并且已创建子 Job：

```
kubectl get jobset jobset-quickstart -n default
kubectl get jobs -n default -l jobset.sigs.k8s.io/jobset-name=jobset-quic
kstart
kubectl get pods -n default -l jobset.sigs.k8s.io/jobset-name=jobset-quic
kstart
```

预期结果：

- 一个名为 `jobset-quickstart-leader-0` 的 Job，包含一个 pod。
- 一个名为 `jobset-quickstart-workers-0` 的 Job，包含四个 pod。
- 一个名为 `jobset-quickstart` 的无头 Service，用于为这些 pod 提供稳定的 DNS 主机名。

例如，查看某个 pod 的日志：

```
kubectl logs -n default -l jobset.sigs.k8s.io/replicatedjob-name=leader -
f
```

验证完成

大约 30 秒后，所有 pod 都会完成。确认 JobSet 状态：

```
kubectl get jobset jobset-quickstart -n default -o jsonpath='{.status.conditions}'
```

成功的 JobSet 会报告一个 `type: Completed` 且 `status: "True"` 的 condition。子 Job 会显示其 `COMPLETIONS` 达到配置值（`leader` 为 `1/1`，`workers` 为 `4/4`）。

清理

删除演示资源：

```
kubectl delete -f jobset-quickstart.yaml
```

删除 `JobSet` 会级联删除其子 Job、pod，以及为该 workload 创建的无头 Service。

后续步骤

如需了解更高级的模式，请参考上游示例和文档：

- **JobSet** 概念：https://jobset.sigs.k8s.io/docs/concepts/↗
- 上游示例：https://github.com/kubernetes-sigs/jobset/tree/main/site/static/examples↗（上游的 `examples/` 是指向该路径的 symlink，因此 GitHub 的 tree 视图仅在实际位置下可用）
- 成功和失败策略：可配置的成功条件（`Any` / `All`）以及按错误类型进行的失败处理。
- 启动顺序：`spec.startupPolicy.startupPolicyOrder: InOrder` 强制执行先 leader 后 workers 的启动顺序。
- 独占放置：在 JobSet 上设置 `alpha.jobset.sigs.k8s.io/exclusive-topology`，可将每个子 Job 的 pod 共同放置在同一个 topology domain 中。
- **Kueue** 集成：将 JobSet 与 Alauda Build of Kueue 结合使用，以管理 batch workload 的队列和资源配额。

部署

本章介绍推理服务的创建与扩缩容、模型管理与存储、模型即服务（Model as a Service, MaaS）、推理网关中的配额和计量，以及模型压缩。

本章所依赖的组件列在下方的 Components 中。

模型管理

介绍

- Model Catalog
- Model Registry
- 推荐工作流

模型目录和模型注册表

- 模型目录
- 模型注册表
- 从模型管理进行部署

使用 Workb

- 选择上传路径
- 前提条件
- 创建并连接到 W
- 将模型下载或复

模型存储

- 选择存储后端
- 使用 S3 对象存储
- 使用 OCI 容器存储模型
- 使用 PVC 存储模型
- Model URI 要求

共享模型

- 概述
- 通过 Model Catalog 共享模型
- 通过 Model Registry 共享模型
- 推荐的共享模式

推理服务

介绍

故障排查

管理推理服务

优势

推理服务模板管理

更新推理服务

调用已发布的推理服务

操作指南

推理网关

对消费者进行身份验证

简介

使用场景

前提条件

步骤

验证

了解更多

下一步

配置令牌配额

简介

使用场景

前提条件

步骤

验证

了解更多

后续步骤

令牌使用量

简介

用例

前提条件

步骤

验证

了解更多

下一步

路由到 **LLM** 提供商

简介

使用场景

前提条件

步骤

验证

了解更多

Token 用量分摊计费

简介

使用场景

前提条件

步骤

验证

故障排查

了解更多

下一步

LLM 压缩器

介绍

前言

LLM Compressor 支持多种压缩技术：

LLM Compressor 支持多种压缩算法：

在 Alauda AI 中使用 LLM Compressor

支持的模型压缩工作流

Model as a Service

介绍

为什么使用 MaaS？

请求流程

主要概念

选择指南

相关主题

MaaS 管理员指南

发布平台推理服务

导入外部模型服务

创建订阅

配置 guardrails

查看使用情况和报告

MaaS 用户指

打开我的订阅

查找你的模型和

创建 API key

调用模型

理解配额行为

查看个人使用情

理解请求错误

MaaS 组件扩缩容

Gateway

Authorino

rate limiter

Guardrail engine 和 shim

在哪里配置副本和资源

常见问题

转发到使用非默认端口的外部模型服务时，请求可能失败

组件

KServe

Envoy AI Gateway

Alauda Buil

InferNex Bridge

模型管理

介绍

- Model Catalog
- Model Registry
- 推荐工作流

模型目录和模型注册表

- 模型目录
- 模型注册表
- 从模型管理进行部署

使用 Workb

- 选择上传路径
- 前提条件
- 创建并连接到 W
- 将模型下载或复

模型存储

- 选择存储后端
- 使用 S3 对象存储
- 使用 OCI 容器存储模型
- 使用 PVC 存储模型
- Model URI 要求

共享模型

- 概述
- 通过 Model Catalog 共享模型
- 通过 Model Registry 共享模型
- 推荐的共享模式

介绍

Alauda AI 通过两项互补的能力管理模型：**Model Catalog** 和 **Model Registry**。两者用途不同，不应混淆。

目录

Model Catalog

Model Registry

推荐工作流

Model Catalog

Model Catalog 是由已配置目录源提供的模型可搜索清单。目录源可以是 Hugging Face 组织、YAML 目录或平台提供的 OCI 模型目录。目录条目是外部引用；目录存储其描述性信息和 artifact URI，但不会自行存储模型文件。

从 **User View > AI Hub > Model Catalog** 中，您可以：

- 以网格或列表视图浏览模型；
- 按源、提供商、任务类型、框架和标签筛选模型；
- 打开模型详细信息，包括其描述、模型卡片、许可证和 artifact URI；
- 直接从目录 artifact 创建推理服务；或

- 将目录模型及其某个 artifact 注册到 Model Registry 中。

管理员在 **Admin View > Settings > Model Catalog Sources** 中配置源。对于用户，目录列表为只读：对目录模型的更改应在上游源中进行，或通过将模型注册到 Model Registry 中进行。

Model Registry

Model Registry 是用于管理团队在命名空间中注册和使用的模型生命周期的系统。它存储模型元数据、已注册模型、模型版本和模型 artifact。模型 artifact 由 **Model URI** 标识（例如 S3 URI、PVC URI 或 OCI image URI）；注册表存储元数据和 artifact 引用，而不是模型文件工作树。

从 **User View > AI Hub > Model Registry** 中，您可以：

- 注册模型及其第一个版本；
- 向现有注册模型添加其他版本；
- 编辑标签、描述和自定义属性；
- 查看某个版本的 artifact URI 和模型框架；
- 从活动版本创建推理服务；以及
- 归档或恢复模型和版本。

管理员在 **Admin View > Settings > Model Registries** 中创建注册表后端。注册表使用选定的 Storage Class 和存储大小来运行其元数据服务。在底层部署和存储就绪之前，注册表可能会一直处于 provisioning 状态。

推荐工作流

1. 在受支持的存储位置准备模型，例如 S3、PVC 或 OCI registry。请参阅 Model Storage。
2. 如果模型来自已配置的源，请在 Model Catalog 中找到它并查看其详细信息。否则，打开 Model Registry 并直接注册模型。
3. 选择一个 Model Registry，输入模型名称和版本，并提供 Model URI。
4. 验证模型元数据和活动版本。
5. 从目录或活动的 Model Registry 版本部署模型。

6. 归档不再使用的版本。带有已发布推理服务的版本必须先完成清理，然后才能归档。

以下页面将详细介绍目录、注册表、存储、上传和共享工作流。

模型目录和模型注册表

Alauda AI 在 **AI Hub** 中提供两种模型管理体验：

功能	用途	管理位置
模型目录	浏览由已配置的外部源或平台源提供的模型和元数据。	User View > AI Hub > Model Catalog
模型注册表	注册供命名空间使用的模型、版本、元数据和构件位置。	User View > AI Hub > Model Registry

模型通过 **Model URI** 引用，而元数据和生命周期状态 由目录或注册表 API 管理。

目录

- 模型目录
  - 浏览目录模型
  - 部署或注册目录模型
  - 配置目录源（管理员）
- 模型注册表
  - 创建注册表（管理员）
  - 注册模型和版本
  - 添加版本
  - 更新元数据
  - 归档和恢复

从模型管理进行部署

模型目录

浏览目录模型

1. 切换到 **User View** 并选择一个命名空间。
2. 在左侧导航中展开 **AI Hub**，然后选择 **Model Catalog**。
3. 可选：选择一个目录 **Source**，并按任务类型、提供商、 框架或标签进行筛选。
4. 选择模型卡片或模型名称以打开其详细信息。
5. 查看模型描述、模型卡片、任务类型、标签、框架、 张量类型、大小、许可证和 Model URI。

用户视图中的目录条目为只读。它是外部引用，因此 不能为该条目上传本地文件或执行删除操作。

部署或注册目录模型

在目录模型的详细信息页面上：

- 选择 **Create Inference API**，打开推理服务表单，其中会预先选择 目录构件；或者
- 选择 **Register Model**，将目录模型复制到模型注册表中。 选择一个注册表，查看模型名称和描述，接受或编辑建议的版本名称， 并确认构件 URI。目录元数据（例如任务和框架）可能会复制到已注册模型或版本。

注册目录模型会在所选注册表中创建一个已注册模型、一个模型版本和一个 模型构件。该构件仍通过其 URI 寻址。

配置目录源（管理员）

管理员可以在 **Admin View > Settings > Model Catalog Sources** 中配置源。

1. 选择 **Add Source**。

2. 输入源名称。使用小写字母、数字和下划线；不要使用连字符。
3. 选择源类型：
   - **Hugging Face**：提供必需的 Organization，以及在需要时提供 Access Token。
   - **YAML**：提供 YAML 目录内容。
4. 可选：设置 Display Name、Allowed Models、Excluded Models 和 Enabled。
5. 选择 **Add**。源状态会指示其是否可用、已禁用或存在错误。
6. 需要时，从源所在行的操作菜单中编辑或删除源。

源配置控制 Model Catalog 中显示的内容。它不会 创建私有已注册模型或上传模型文件。

模型注册表

创建注册表（管理员）

用户注册模型前，管理员必须在 **Admin View > Settings > Model Registries** 中创建后端：

1. 选择 **Create Model Registry**。
2. 输入注册表名称，以及可选的显示名称。
3. 选择 Storage Class。
4. 选择 Access Mode（默认通常为 `ReadWriteOnce`）。
5. 设置元数据存储大小（GiB）。
6. 选择 **Create**。

注册表列表显示注册表状态、Storage Class、存储大小和 创建时间。创建操作会配置注册表底层的部署和持久化存储，因此状态 最初可能为 **Deploying** 或尚不可用。行操作菜单还提供 **Manage permissions**、**Edit** 和 **Delete**。

注册模型和版本

1. 切换到 **User View**，选择一个命名空间，然后打开 **AI Hub > Model Registry**。
2. 选择 **Register Model**。

3. 选择一个 **Model Registry**。如果表单已选择第一个可用注册表， 请确认它是预期的注册表。
4. 输入模型 **Name** 和可选的 Description。
5. 输入 **Version** 名称。如果模型名称已存在，控制台会建议下一个可用名称， 例如 `Version 2`；提交前请检查该名称。
6. 输入 **Model URI**。URI 必须包含 scheme，例如： `s3://bucket/path`、`pvc://claim/path` 或 `oci://registry.example.com/project/model:tag`。
7. 选择 **Register**。

控制台会创建或复用已注册模型，并添加提交的版本和模型构件。模型详细信息页面 包含 **Overview** 和 **Versions** 标签页。Overview 显示描述、所有者、ID、标签、 自定义属性、最新版本和相关推理服务。版本详细信息页面显示版本状态、构件 URI、 框架、时间戳和相关推理服务。

添加版本

要向现有已注册模型添加新构件：

1. 打开已注册模型的详细信息页面。
2. 选择 **Versions**。
3. 选择 **Register New Version**。
4. 输入版本名称、可选描述、Model URI 和模型框架。
5. 选择 **Register New Version**。

版本在归档前保持 **LIVE** 状态。需要再次使用已归档版本时， 使用版本的操作菜单将其恢复。

更新元数据

在已注册模型或版本的详细信息页面上，拥有编辑权限的用户可以 更新标签、描述和自定义属性。元数据存储在 Model Registry 中， 与任何上游目录源相互独立。已归档的模型和版本在恢复前无法编辑。

归档和恢复

归档是用于替代删除已注册模型或版本的生命周期操作：

1. 打开模型或版本的操作菜单。
2. 选择 **Archive**。
3. 在确认字段中输入准确的模型或版本名称。
4. 选择 **Archive**。
5. 要再次使用它，切换到已归档视图并选择 **Restore**。

归档已注册模型也会归档其所有版本。已发布推理服务的模型或版本无法归档； 请先删除相关推理服务。已归档条目仍保留在注册表中，可以查看或恢复。

从模型管理进行部署

两种模型管理视图都可以启动推理部署：

- **Model Catalog**：打开目录条目并选择 **Create Inference API**。
- **Model Registry**：打开活动模型版本并选择 **Create Inference API**，或在最新版本旁选择 **Deploy**。

控制台会将所选构件的 Model URI、模型名称、版本、任务 和框架传递到推理服务表单。在其中选择合适的服务运行时和部署资源。 有关存储 URI 格式和直接存储工作流，请参阅 Model Storage。

使用 Workbench 上传模型

Workbench 提供了一个便捷的环境，用于下载模型文件、从本地计算机复制文件，以及将文件上传到模型存储中。本页仅介绍工作流中的 Workbench 部分：在本地准备模型，然后将其复制到 S3、PVC 或 OCI registry。

Model Catalog 不需要此页面：catalog 模型已由配置的源提供。在 Model Registry 中注册已上传的 artifact 是一个独立的控制台操作。上传完成后，请按照注册模型和版本创建一个已注册的模型版本，并使用生成的 Model URI。

目录

选择上传路径

前提条件

创建并连接到 Workbench

将模型下载或复制到 Workbench

将模型从 Workbench 上传到 S3

1. 在 Workbench 中准备 S3 访问
2. 安装或验证 AWS CLI
3. 上传并验证模型目录

从 Workbench 将模型上传到 PVC

将模型作为 OCI artifact 上传

下一步：注册已上传的 artifact

选择上传路径

需求	**Workbench** 操作	生成的 **Model URI**
将模型保留在共享文件系统上	挂载 PVC，并将模型复制到稳定目录	`pvc://<pvc-name>/<path>`
将模型存储为对象	在 Workbench 中配置 S3 访问并上传目录	`s3://<bucket>/<prefix>`
将模型作为镜像分发	构建并推送 OCI 模型镜像	`oci://<registry>/<repository>:<tag>`

前提条件

- 具有 Alauda AI 监控面板的访问权限以及目标命名空间。
- 一个正在运行的 Workbench，并具有足够的本地存储或已挂载的 PVC 存储来保存模型。
- S3 对象存储、PVC 或 OCI registry 中的目标位置。
- 目标存储的写入凭据。
- 对于 S3，需要已有的 bucket、endpoint、access key、secret key，以及可选的 region。上传脚本不会创建 bucket。
- 对于 PVC，需要一个可由 Workbench 和后续推理服务挂载的 claim。Workbench 和推理服务可以依次使用 PVC，因此在不需要并发挂载时不需要 `ReadWriteMany`。

有关存储后端要求，请参阅 Model Storage。

创建并连接到 **Workbench**

1. 从左侧导航中的 **Workbench** 创建 Workbench。在较旧的安装中，对应入口可能位于 **Advanced > Notebook** 下。
2. 当模型过大，无法放入 Workbench 的本地工作区，或文件必须在 Workbench 重启后仍然可用时，挂载 PVC。
3. 等待 Workbench 状态变为 **Running**，然后选择 **Connect**。

4. 在 JupyterLab 或 code-server 中打开终端。

将模型下载或复制到 Workbench

从获准的上游源下载模型，或将模型从本地计算机复制到 Workbench。例如，使用 Hugging Face mirror：

```
export HF_ENDPOINT=https://hf-mirror.com
pip install huggingface_hub
huggingface-cli download \
  deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B \
  --local-dir ./DeepSeek-R1-Distill-Qwen-1.5B
```

其他常见源包括：

- Hugging Face↗
- hf-mirror↗
- ModelScope↗

如果 Workbench 无法访问 Internet，请在能够连接上游源的计算机上下载模型，然后通过 Workbench 文件浏览器上传：

- 在 JupyterLab 中，打开 **Files**，选择 **Upload Files**，然后选择模型文件或 archive。
- 在 code-server 中，打开 **Explorer**，将文件拖到目标文件夹中。

必要时，在 Workbench 终端中解压 archive。上传前，请确认模型目录包含其 serving framework 所需的文件，例如 configuration、tokenizer 和 weight 文件。不要包含凭据或无关的训练数据。

将模型从 Workbench 上传到 S3

以下操作步骤会将 Workbench 本地目录递归上传到兼容 S3 的 bucket。它适用于 AWS S3、Ceph、MinIO 以及其他提供兼容 S3 API 的服务。

1. 在 Workbench 中准备 S3 访问

如果项目使用托管的 connection 设置，请在 **User View > Connections** 中创建或获取 S3 Connection。选择 **S3-compatible object storage**，并提供 access key、secret key、endpoint、region 和 bucket。S3 Connection 会记录诸如 `AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY`、`AWS_S3_ENDPOINT`、`AWS_DEFAULT_REGION` 和 `AWS_S3_BUCKET` 等值。

为推理服务选择的 Connection 不会自动将其值注入已在运行的 Workbench。Workbench 需要拥有自己的 S3 访问权限。请使用管理员批准的方法配置相同的值，例如 PodDefault、已挂载的 Secret 或临时终端环境变量。以下上传命令使用环境变量方式。

对于临时终端会话，请导出这些值，不要将它们放入 notebook 或提交到文件中。对于可重复的工作负载，优先使用 Secret 或 PodDefault，并在上传完成后清除变量：

```
export AWS_ACCESS_KEY_ID='<access-key>'
export AWS_SECRET_ACCESS_KEY='<secret-key>'
export AWS_S3_ENDPOINT='https://s3.example.com'
export AWS_DEFAULT_REGION='us-east-1'
export AWS_S3_BUCKET='models'
```

对于 AWS S3，使用默认 AWS endpoint 时可以省略 `AWS_S3_ENDPOINT`。对于兼容 S3 的服务，请将其设置为服务 endpoint。如果 endpoint 使用自签名证书，请使用组织批准的证书配置；不要在生产环境中禁用 TLS 验证。

2. 安装或验证 AWS CLI

AWS CLI 是 Amazon S3 的标准命令行客户端。当传入存储 endpoint 时，它还支持 Ceph、MinIO 和 SeaweedFS 等兼容 S3 的对象存储。在 Workbench 终端中验证它是否可用：

```
aws --version
```

如果 AWS CLI 不可用，请在 home 目录中安装官方 AWS CLI v2 package。以下命令会根据 Workbench 的 CPU 架构选择 package：

```
mkdir -p "$HOME/bin" "$HOME/.local/aws-cli"
case "$(uname -m)" in
  x86_64) AWSCLI_ARCH=x86_64 ;;
  aarch64|arm64) AWSCLI_ARCH=aarch64 ;;
  *) echo "Unsupported architecture: $(uname -m)"; exit 1 ;;
esac
curl -fL "https://awscli.amazonaws.com/awscli-exe-linux-${AWSCLI_ARCH}.zip" \
  -o /tmp/awscliv2.zip
unzip -q /tmp/awscliv2.zip -d /tmp
/tmp/aws/install -i "$HOME/.local/aws-cli" -b "$HOME/bin"
export PATH="$HOME/bin:$PATH"
aws --version
```

3. 上传并验证模型目录

Bucket 必须已存在。AWS CLI 使用环境中的 `AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY` 和 `AWS_DEFAULT_REGION`。对于 Amazon S3，请将 `AWS_S3_ENDPOINT` 留空。对于兼容 S3 的服务，以下命令将 `AWS_S3_ENDPOINT` 作为 endpoint URL 传入：

```
AWS_ARGS=()
if [ -n "${AWS_S3_ENDPOINT:-}" ]; then
  AWS_ARGS+=(--endpoint-url "$AWS_S3_ENDPOINT")
fi

aws "${AWS_ARGS[@]}" s3 cp \
  "${LOCAL_MODEL_DIR:-./DeepSeek-R1-Distill-Qwen-1.5B}/" \
  "s3://${AWS_S3_BUCKET}/${S3_PREFIX:-DeepSeek-R1-Distill-Qwen-1.5B}/" \
  --recursive

aws "${AWS_ARGS[@]}" s3 ls \
  "s3://${AWS_S3_BUCKET}/${S3_PREFIX:-DeepSeek-R1-Distill-Qwen-1.5B}/" \
  --recursive
```

该命令会保留 S3 prefix 下的模型目录布局。例如，当 `S3_PREFIX=qwen` 时，`./model/config.json` 会上传为 `s3://models/qwen/config.json`。

生成的 Model URI 为：

```
s3://<bucket>/<prefix>
```

例如，使用 `AWS_S3_BUCKET=models` 和 `S3_PREFIX=DeepSeek-R1-Distill-Qwen-1.5B` 时，请使用：

```
s3://models/DeepSeek-R1-Distill-Qwen-1.5B
```

`s3://` URI 不包含 endpoint。endpoint 和凭据会通过 S3 Connection，或通过 ServiceAccount 和 Secret 单独为推理服务配置。请参阅使用 S3 对象存储进行模型存储。

上传后，尽可能从终端会话中删除临时凭据：

```
unset AWS_ACCESS_KEY_ID AWS_SECRET_ACCESS_KEY AWS_S3_ENDPOINT
unset AWS_DEFAULT_REGION AWS_S3_BUCKET
```

从 Workbench 将模型上传到 PVC

1. 在目标命名空间中创建或确定一个 PVC。如果 Workbench 和推理服务在不同时间使用 PVC，则 PVC 不需要 `ReadWriteMany`。
2. 将 PVC 挂载到 Workbench。
3. 在 Workbench 文件浏览器中打开已挂载的 PVC 目录。
4. 创建一个稳定的模型目录，例如 `models/DeepSeek-R1-Distill-Qwen-1.5B`。
5. 从本地计算机上传模型文件，或将文件从 Workbench 工作区复制到该目录。
6. 确认 configuration、tokenizer 和 weight 文件出现在该目录中。
7. 在启动需要独占 claim 访问权限的推理服务之前，停止 Workbench，或以其他方式从 Workbench 卸载 PVC。这适用于 `ReadWriteOnce` PVC。如果 Workbench 必须在推理服务运行期间保持挂载，请使用支持并发挂载的 storage class 和 access mode，例如 `ReadWriteMany`。

生成的 Model URI 为：

```
pvc://<pvc-name>/models/DeepSeek-R1-Distill-Qwen-1.5B
```

请参阅将模型文件上传到 PVC。

将模型作为 **OCI artifact** 上传

要使用 OCI 模型存储，请将模型打包为 OCI image，并将其推送到 Harbor 等兼容 OCI 的 registry。Workbench 必须具有获准的 OCI client 和 registry 凭据。推送后记录 image reference，例如：

```
oci://registry.example.com/models/deepseek-r1:1.0
```

推理集群必须能够拉取该 image。有关打包说明，请参阅使用 KServe Modelcar 进行模型存储。

下一步：注册已上传的 **artifact**

上传到 S3、PVC 或 OCI registry 后，记录 Model URI 并打开 **User View > AI Hub > Model Registry**。选择 **Register Model**，选择一个 Model Registry，输入模型名称和版本，并提供 Model URI。此注册步骤会存储模型元数据和 artifact 引用；它不会移动或复制模型文件。在已注册版本处于使用状态期间，请保持 S3 bucket、prefix 和访问配置不变。

模型存储

Model Catalog 或 Model Registry 存储模型元数据和模型构件的位置。模型文件本身保留在推理服务可以访问的存储后端中。支持的存储后端包括：

- **S3** 对象存储：Storage Initializer 在主容器启动前下载模型。
- **Persistent Volume Claim (PVC)**：Storage Initializer 从已挂载的持久卷加载模型。
- **OCI** 容器：在 KServe 中也称为 modelcar；容器运行时使用镜像层和缓存高效加载模型。

注册模型版本时，将构件位置作为 **Model URI** 输入。有关在 UI 中管理的项目级 S3、URI 或 OCI 端点，请参阅使用连接。

目录

选择存储后端
使用 S3 对象存储
身份验证配置
部署 S3 支持的模型
使用 OCI 容器存储模型
使用 PVC 存储模型
将模型文件上传到 PVC
前提条件
操作步骤
部署 PVC 支持的模型
Model URI 要求

选择存储后端

后端	**Model URI** 示例	凭据或访问配置
S3	`s3://models/Qwen2.5-0.5B-Instruct`	S3 Connection，或 ServiceAccount 和 Secret
PVC	`pvc://model-pvc/models/Qwen2.5-0.5B-Instruct`	目标命名空间中可用的 PVC
OCI	`oci://registry.example.com/models/qwen:1.0`	注册表需要身份验证时使用 OCI Connection

将模型放入存储后，打开 **AI Hub > Model Registry > Register Model** 并输入 URI。然后，可以从 Model Registry 部署已注册的活动版本。目录条目也可以提供构件 URI，并可直接从 **AI Hub > Model Catalog** 部署。

使用 **S3** 对象存储

S3 通常用于模型存储。推理服务需要通过附加到 ServiceAccount 的 Secret，或通过在控制台中选择的 S3 Connection 获取端点和凭据。

身份验证配置

建议为每个项目创建单独的 ServiceAccount 和 Secret。以下示例展示 KServe S3 Secret 格式：

```yaml
apiVersion: v1
stringData:
  AWS_ACCESS_KEY_ID: YOUR_ACCESS_KEY
  AWS_SECRET_ACCESS_KEY: YOUR_SECRET_KEY
kind: Secret
metadata:
  annotations:
    serving.kserve.io/s3-endpoint: your_s3_service_ip:your_s3_port
    serving.kserve.io/s3-usehttps: '0'
  name: s3-creds
  namespace: demo-space
type: Opaque
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sa-models
  namespace: demo-space
secrets:
  - name: s3-creds
```

1 将 `YOUR_ACCESS_KEY` 和 `YOUR_SECRET_KEY` 替换为 S3 服务的凭据。2. 将 `your_s3_service_ip:your_s3_port` 替换为 S3 端点。3. 当端点使用 HTTPS 时，将 `serving.kserve.io/s3-usehttps` 设置为 `"1"`。4. 将 Secret 和 ServiceAccount 保留在推理服务运行所在的命名空间中。

部署 S3 支持的模型

以下是直接使用 KServe 的示例。在控制台中，建议将 `storageUri` 注册为模型构件 URI，并从 Model Registry 的活动版本创建服务。

```yaml
kind: InferenceService
apiVersion: serving.kserve.io/v1beta1
metadata:
  annotations:
    serving.kserve.io/deploymentMode: Standard
  labels:
    aml.cpaas.io/runtime-type: vllm
  name: s3-demo
  namespace: demo-space
spec:
  predictor:
    maxReplicas: 1
    minReplicas: 1
    model:
      modelFormat:
        name: transformers
      protocolVersion: v2
      resources:
        limits:
          cpu: '2'
          ephemeral-storage: 10Gi
          memory: 8Gi
        requests:
          cpu: '2'
          memory: 4Gi
      runtime: aml-vllm-0.11.2-cpu
      storageUri: s3://models/Qwen2.5-0.5B-Instruct
    securityContext:
      seccompProfile:
        type: RuntimeDefault
    serviceAccountName: sa-models
```

将运行时替换为已安装的 `ClusterServingRuntime`。`storageUri` 必须指向 S3 bucket 中的模型目录，并且 ServiceAccount 必须能够读取 Secret。

使用 **OCI** 容器存储模型

你可以将模型打包为 OCI 镜像，并将其推送到 Harbor 或 Quay 等兼容 OCI 的注册表。对于离线安装，以及从具有节点侧镜像缓存的注册表快速启动，modelcar 存储非常有用。

将镜像引用记录为 Model URI，例如：

```
oci://registry.example.com/models/qwen:1.0
```

推理集群必须能够拉取该镜像。注册表为私有注册表时，请配置 OCI Connection。有关打包和部署步骤，请参阅使用 KServe Modelcar 存储模型。

使用 PVC 存储模型

将模型文件上传到 PVC

你可以从现有 PVC 提供模型服务。从已附加 PVC 的 Workbench 上传文件：

前提条件

- 你可以访问 Alauda AI 监控面板和目标命名空间。
- 你有一个正在运行的 Workbench 和一个 PVC。
- Workbench 已附加到 PVC。
- 模型文件位于本地计算机或 Workbench 中。

有关创建 Workbench 和附加 PVC 的信息，请参阅创建 Workbench。

操作步骤

1. 在 Alauda AI 监控面板中，选择 **Workbench**。
2. 找到正在运行的 Workbench，然后选择 **Connect**。
3. 打开 IDE 文件浏览器。在 JupyterLab 中使用 **Files**；在 code-server 中使用 **Explorer**。
4. 导航到已附加 PVC 的根目录，通常是 Workbench 主目录。在此处创建的文件会持久保存在 PVC 中。
5. 为模型创建目录，例如 `models/Qwen2.5-0.5B-Instruct`。
6. 上传模型文件。在 JupyterLab 中选择 **Upload**；在 code-server 中，将文件拖入目标目录。
7. 确认模型文件显示在目标目录中。

Model URI 为：

```
pvc://<pvc-name>/models/Qwen2.5-0.5B-Instruct
```

将此 URI 注册为 Model Registry 版本的构件位置。PVC 和推理服务必须位于服务可以挂载该 claim 的命名空间中。

部署 **PVC** 支持的模型

```yaml
kind: InferenceService
apiVersion: serving.kserve.io/v1beta1
metadata:
  annotations:
    serving.kserve.io/deploymentMode: Standard
  labels:
    aml.cpaas.io/runtime-type: vllm
  name: pvc-demo-1
  namespace: demo-space
spec:
  predictor:
    maxReplicas: 1
    minReplicas: 1
    model:
      modelFormat:
        name: transformers
      protocolVersion: v2
      resources:
        limits:
          cpu: '2'
          ephemeral-storage: 10Gi
          memory: 8Gi
        requests:
          cpu: '2'
          memory: 4Gi
      runtime: aml-vllm-0.11.2-cpu
      storageUri: pvc://model-pvc/models/Qwen2.5-0.5B-Instruct
    securityContext:
      seccompProfile:
        type: RuntimeDefault
```

将 `model-pvc` 和路径替换为包含模型的 PVC 和目录。

Model URI 要求

控制台会验证 Model URI 是否包含 scheme 以及非空的位置。以下形式在部署配置支持时有效：

- `s3://<bucket>/<path>`
- `pvc://<pvc-name>/<optional-path>`
- `oci://<registry>/<repository>:<tag>`
- 存储初始化器或服务运行时支持的其他已配置 URI scheme

诸如 `/models/qwen` 的本地路径或不带 scheme 的仓库名称不是完整的 Model URI。确认所选服务运行时支持该模型格式，并且运行时可以访问所引用的存储。

共享模型

目录

概述
通过 Model Catalog 共享模型
配置 catalog 源
浏览和使用 catalog 模型
通过 Model Registry 共享模型
授予 Model Registry 访问权限
管理已注册模型
归档和恢复
推荐的共享模式

概述

模型共享通过 **Model Catalog** 源和 **Model Registry** 权限进行管理。

当管理员希望让用户发现来自上游源的模型时，请使用 Model Catalog。当团队需要与其他用户或命名空间共享已注册的模型、版本、元数据和构件位置时，请使用 Model Registry。

共享模型仍需要一个可访问的构件位置。Model Catalog 或 Model Registry 存储模型元数据和 Model URI；模型文件本身保留在 S3、PVC 或 OCI registry 中。配置相应的 Connection、Secret 或 service account，以便推理服务访问构件。

通过 **Model Catalog** 共享模型

配置 **catalog** 源

管理员可以从 **Admin View > Settings > Model Catalog Sources** 添加源：

1. 选择 **Add Source**。
2. 使用小写字母、数字和下划线输入源名称。
3. 选择源类型：
   - **Hugging Face**：输入必需的 Organization，以及（如有要求）Access Token。
   - **YAML**：输入 catalog YAML 内容。
4. 可选设置 Display Name、Allowed Models、Excluded Models 和 Enabled。
5. 选择 **Add**，等待源状态变为可用。

Allowed Models 和 Excluded Models 可以限制该源公开的模型。请勿在 YAML 内容或模型元数据中包含凭据。不再需要某个源时，可从其行操作菜单编辑或删除该源。

浏览和使用 **catalog** 模型

用户可以打开 **User View > AI Hub > Model Catalog**，选择一个源，然后打开模型条目。详情页可以显示 provider、description、model card、task type、labels、framework、license、size 和 artifact URI。

在详情页中，用户可以：

- 选择 **Create Inference API**，直接部署 catalog 构件；或
- 选择 **Register Model**，选择可用的 Model Registry，并创建引用 catalog 构件的本地已注册模型和版本。

Catalog 条目本身为只读。要编辑本地元数据或控制其生命周期，请注册模型并管理生成的 Model Registry 记录。

通过 **Model Registry** 共享模型

授予 **Model Registry** 访问权限

管理员可以从 **Admin View > Settings > Model Registries** 管理 registry 访问权限：

1. 找到目标 registry。
2. 打开其行操作菜单并选择 **Manage permissions**。
3. 添加所需的用户、组或命名空间，并分配相应权限。
4. 保存权限更改。

添加命名空间后，该命名空间中的 service account 可以根据配置的权限访问 registry。仅授予团队或工作负载所需的访问权限。

管理已注册模型

具有所需 registry 权限的用户可以打开 **User View > AI Hub > Model Registry**，然后：

1. 选择包含该模型的 Model Registry。
2. 打开已注册模型，查看其 Overview 和 Versions 标签页。
3. 根据需要更新 labels、descriptions 或 custom properties。
4. 打开一个版本，检查其 Model URI、framework、state 以及相关的 inference services。
5. 对于 LIVE 版本，选择 **Create Inference API** 或 **Deploy**。

模型和版本元数据由 Model Registry 维护。它们独立于 catalog 源元数据。

归档和恢复

归档会将模型或版本从活动视图中移除，同时保留其元数据和构件引用：

1. 打开模型或版本操作菜单，然后选择 **Archive**。
2. 在确认字段中输入准确的模型或版本名称。
3. 选择 **Archive**。
4. 使用已归档视图和 **Restore** 操作再次将其激活。

正在被推理服务使用的模型或版本无法归档。请先删除相关推理服务。归档已注册模型也会归档其版本。在恢复之前，已归档条目不能用于新的部署。

推荐的共享模式

需求	推荐方法
让用户发现经过批准的上游模型	配置 Model Catalog Source
部署 catalog 模型而不管理本地副本	直接从 Model Catalog 部署
跟踪团队拥有的版本和元数据	在 Model Registry 中注册模型
与另一个命名空间共享已注册模型	管理 Model Registry 权限并授予命名空间访问权限
让模型可供推理服务使用，但从活动选择中隐藏它	仅在移除依赖的推理服务后归档模型/版本
与工作负载共享模型文件	将构件保留在 S3、PVC 或 OCI 存储中，并配置访问凭据

推理服务

介绍

介绍

管理推理服务

管理推理服务

优势

核心功能

创建推理服务

推理服务模板管理

更新推理服务

调用已发布的推理服务

操作指南

使用 CLI 创建推理服务

从 Kubeflow 监控面板部署推理服

扩展推理运行

前提条件
创建 InferenceService
检查 InferenceService 状态
使用 OpenAI 兼容 API 测试 InferenceServ

简介
前提条件
访问 Endpoints UI
部署新模型
验证

简介
场景
前提条件
标准工作流（示
特定运行时示例
运行时配置对比

使用 KServe Modelcar 进行模型存储

概述
使用 OCI Containers 进行模型存储的优势
先决条件
将模型打包为 OCI Image
从 OCI Image 部署模型
最佳实践
故障排查
结论

配置推理服务外部访问

简介
步骤

配置推理服务

简介
前提条件

使用 KEDA 为推理服务配置自动扩缩容

简介
前提条件
步骤

基于 CUDA

简介
步骤

使用 Kueue 调度推理服务

前提条件
操作步骤

为 vLLM Inference Services 启用 Expert Parall

简介
何时使用 Expert Parallel
前提条件和限制
EP 配置概览
配置单节点 InferenceService
查看已配置的 Spec
关于多节点部署
参考资料

vLLM 推理服

简介
在决定之前
本指南在 Alaud
推荐起始点
前提条件
配置面
在 Alauda AI 上
端到端示例
验证并衡量影响
回滚
故障排查

注意事项和已知
参考资料

故障排查

使用 **MLServer Runtime** 时遇到
问题描述
根本原因分析
解决方案
总结

推理服务无法进入运行状态
问题描述
根因分析
解决方案
总结

介绍

推理服务是 Alauda AI 平台的核心功能，专注于将 LLM 模型高效部署为在线推理服务，支持 HTTP API 和 gRPC 等多种调用方式。通过推理服务，用户可以快速构建 LLM 应用，并对外提供稳定、高性能的 LLM 能力。

WARNING

在容器中运行内置运行时需要 root 权限。请确保在可信环境中使用，并遵循您的安全策略。

管理推理服务

推理服务功能的核心定义是将训练好的机器学习或深度学习模型部署为可在线调用的服务，并使用 HTTP API 或 gRPC 等协议，使原生应用能够实时或批量使用模型的预测、分类、生成及其他功能。此功能主要解决模型训练完成后，如何高效、稳定、便捷地将模型部署到生产环境，并提供可扩展的在线服务。

目录

- 优势
- 核心功能
- 创建推理服务
  - 第 1 步：访问推理服务创建页面
  - 第 2 步：配置模型元数据（如需要）
  - 第 3 步：选择发布模式并配置
  - 第 4 步：监控和管理推理服务
- 推理服务模板管理
  - 第 1 步：创建模板
  - 第 2 步：从现有模板创建新模板
  - 第 3 步：删除模板
- 更新推理服务
- 调用已发布的推理服务
  - 推理体验
  - 通过 HTTP API 调用
  - 推理参数说明

不同任务类型的参数说明

文本生成

图像生成

文本分类

其他参考资料

优势

- 简化模型部署流程，降低部署复杂度。
- 提供高可用、高性能的在线和批量推理服务。
- 支持动态模型更新和版本管理。
- 实现模型推理服务的自动化运维与监控。

核心功能

直接将模型部署为推理服务

- 允许用户部署 **Model Catalog** 中的制品或 **Model Registry** 中选定的活动版本，也可直接从 **PVC** 挂载模型。系统会自动下载、缓存并加载模型，然后启动推理服务。

将原生应用用作推理服务

- 使用 Kubernetes 原生应用作为推理服务。此方式具有更大的灵活性，允许用户根据需要自定义推理环境。

推理服务模板管理

- 支持创建、管理和删除推理服务模板，允许用户基于预定义模板快速部署推理服务。

推理服务批量操作

- 支持对多个推理服务执行批量操作，例如批量启动、停止、更新和删除。
- 支持创建和监控批量推理任务，以及导出任务结果。

- 提供批量资源管理，可批量分配和调整推理服务的资源。

推理体验

- 提供交互式界面，便于用户测试和体验推理服务。
- 支持多种输入和输出格式，满足不同原生应用场景的需求。
- 提供模型性能评估工具，帮助用户优化模型部署。

推理 **Runtime** 支持

- 集成 vLLM、Seldon MLServer 等多种主流推理框架，并支持用户自定义推理 Runtime。

> **TIP**
>
> - vLLM：针对 DeepSeek/Qwen 等大型语言模型 (LLM) 进行了优化，支持高并发处理和更高吞吐量，同时具备出色的资源效率。
> - MLServer：专为传统 ML 模型（XGBoost/图像分类）设计，提供多框架兼容性和简化的调试流程。

访问方式、日志、**Swagger**、监控等

- 提供 HTTP API 和 gRPC 等多种访问方式。
- 支持详细的日志记录与分析，便于用户排查问题。
- 自动生成 Swagger 文档，便于用户集成和调用推理服务。
- 提供实时监控和告警功能，确保服务稳定运行。

创建推理服务

1 第 1 步：访问推理服务创建页面

可通过以下两种方式启动推理服务发布流程：

1. 从 **Model Catalog**：在用户视图中，打开 **AI Hub > Model Catalog**，选择一个目录模型，然后单击 创建推理 **API**。

2. 从 **Model Registry**：在用户视图中，打开 **AI Hub > Model Registry**，选择一个活动的已注册模型版本，然后单击 创建推理 **API** 或 部署。
3. 从推理服务：打开 **AI Hub >** 推理服务，然后选择 创建推理服务。

> **TIP**
>
> 自定义发布推理服务需要手动设置参数。您也可以通过组合输入参数创建“模板”，以便快速发布推理服务。

2 第 2 步：配置模型元数据（如需要）

如果 创建推理 **API** 不可用，请确认目录条目或已注册模型版本具有制品 URI，以及所需的任务和框架元数据。对于已注册模型，可在 Model Registry 详情页面更新标签或自定义属性。模型版本必须处于活动 (LIVE) 状态后才能部署。

3 第 3 步：选择发布模式并配置

进入 发布模式选择 页面。AML 提供 自定义发布 和 模板发布 选项。

1. 模板发布：
   - 选择模型并单击 模板名称
   - 进入模板发布表单，其中已预加载模板中的参数，但可手动编辑
   - 单击 发布 以部署推理服务
2. 自定义发布：
   - 单击 自定义发布
   - 进入自定义发布表单并配置参数，包括 模型来源（Model Catalog、Model Registry、Connection 或 Persistent Volume Claim，具体取决于已安装的版本）。
   - 单击 发布 以部署推理服务

4 第 4 步：监控和管理推理服务

您可以在左侧导航栏的推理服务下查看已发布推理服务的状态、日志及其他详细信息。如果推理服务启动失败或运行资源不足，您可能需要更新或重新发布推理服务，并修改可能导致启动失败的配置。

注意： 推理服务会根据请求流量，在“最小副本数”和“最大副本数”之间自动扩缩容。如果“最小副本数”设置为 0，则在一段时间内没有请求时，推理服务会自动暂停并释放资源。此时，如果有请求到达，推理服务可自动启动并加载缓存在 PVC 中的模型。

AML 基于 kserve↗ InferenceService CRD 完成云原生推理服务的发布和运行。如果您熟悉 kserve 的使用，也可以在“直接从模型发布推理服务”时单击右上角的“YAML”按钮，直接修改 YAML 文件以完成更高级的发布操作。

模型发布参数说明

参数	说明
名称	必填，推理 API 的名称。
描述	推理 API 的详细说明，用于解释其功能和用途。
模型位置	必填，选择模型文件的来源：`Model repository` 或 `PVC`。
模型 PVC	当模型位置为 `PVC` 时必填。选择包含模型文件的现有 PVC。
模型 PVC 路径	当模型位置为 `PVC` 时必填。指定模型在 PVC 中的路径。
模型类型	当模型位置为 `PVC` 时必填。选择 `Generative AI model` 或 `Predictive Model`。此选择决定可用的 Runtime。
模型	当模型位置为 `Model repository` 时必填。用于推理的模型名称。
版本	当模型位置为 `Model repository` 时必填。模型版本。选项包括 Branch 和 Tag。
推理 Runtime	必填，用于推理 Runtime 的引擎。当 `Model Type` 为 `Generative AI model` 时，`llm-d` 可用于分布式推理。
配置来源	必填，选择部署资源的来源：`Hardware profile` 或 `Custom`。
硬件配置	当配置来源为 `Hardware profile` 时必填。选择用于配置所需资源的预定义硬件配置。
请求 CPU	当配置来源为 `Custom` 时必填。推理服务请求的 CPU 资源量。
请求内存	当配置来源为 `Custom` 时必填。推理服务请求的内存资源量。

参数	说明
CPU 限制	当配置来源为 `Custom` 时必填。推理服务可使用的最大 CPU 资源量。
内存限制	当配置来源为 `Custom` 时必填。推理服务可使用的最大内存资源量。
GPU 加速类型	当配置来源为 `Custom` 时可用。GPU 加速的类型。
GPU 加速值	当配置来源为 `Custom` 时可用。GPU 加速的值。
存储容量	当模型位置为 `Model repository` 时必填。用于缓存模型的临时存储容量。
自动扩缩容	启用或禁用自动扩缩容功能。
实例数	必填，运行推理服务的实例数量。
环境变量	注入容器 Runtime 环境的键值对。
添加参数	传递给容器入口点可执行文件的参数。字符串数组（例如 ["--port=8080", "--batch_size=4"]）。
启动命令	覆盖容器镜像中的默认 ENTRYPOINT 指令。可执行文件 + 参数（例如 ["python", "serve.py"]）

推理服务模板管理

AML 引入了 模板发布，用于快速部署推理服务。您可以创建和删除模板（更新模板需要创建新模板）。

1 第 1 步：创建模板

- 在左侧导航栏中，单击 推理服务 > 创建推理服务
- 单击 自定义发布
- 进入表单页面并配置参数
- 单击 创建模板

2 第 2 步：从现有模板创建新模板

- 在左侧导航栏中，单击 推理服务 > 创建推理服务
- 选择模型并单击 模板名称
- 根据需要编辑参数
- 单击 创建模板，另存为新模板

3 第 3 步：删除模板

- 在左侧导航栏中，单击 推理服务 > 创建推理服务
- 在模板卡片上，单击 操作 > 删除
- 确认删除

更新推理服务

1. 在左侧导航栏中，单击 推理服务。
2. 单击 推理服务名称。
3. 在推理服务详情页面，单击右上角的 操作 > 更新，进入更新页面。
4. 修改必要字段，然后单击 更新。系统将执行滚动更新，以免中断现有客户端请求。

调用已发布的推理服务

AML 为常见任务类型提供可视化的 “推理体验” 方式来访问已发布的推理服务；您也可以使用 HTTP API 调用推理服务。

推理体验

AML 支持演示以下任务类型的推理服务推理（任务类型在模型元数据中指定）：

- 文本生成
- 文本分类
- 图像分类

- 图像生成

成功发布上述任务类型的推理服务后，您可以在模型详情页面和推理服务详情页面右侧显示 “推理体验” 对话框。根据推理任务的类型，输入和输出数据类型可能有所不同。以文本生成为例，输入文本后，模型生成的文本会以蓝色字体追加到文本框中输入的文本之后。 推理体验支持选择由同一模型多次发布、部署在不同集群中的不同推理服务。选择推理服务后，将调用该推理服务并返回推理结果。

通过 **HTTP API** 调用

发布推理服务后，您可以在原生应用或其他服务中调用此推理服务。本文档将以 Python 代码为例，说明如何调用已发布的推理 API。

1. 在左侧导航栏中单击 推理服务 **>** 推理服务名称，进入推理服务详情页面。
2. 单击 访问方式 选项卡，获取集群内或集群外访问方式。可以直接从此 K8s 集群中的 Notebook 或其他容器使用集群内访问方式。如果需要从集群外的位置（例如本地笔记本电脑）访问，则需要使用集群外访问方式。
3. 单击调用示例以查看示例代码。
   注意：调用示例中提供的代码仅适用于使用 ***mlserver runtime (Seldon MLServer)*** *发布的推理服务所支持的* ***API*** *调用协议。此外，* ***Swagger*** *选项卡也仅支持访问由* ***mlserver runtime*** *发布的推理服务。*

推理参数说明

调用推理服务时，您可以通过调整模型推理参数来调整模型输出效果。 在 推理体验 界面中，已预置常用参数和默认值，也可以添加任意自定义参数。

不同任务类型的参数说明

文本生成

预置参数

参数	数据类型	说明
`do_sample`	bool	是否使用采样；如果不使用，则采用贪心解码。

参数	数据类型	说明
`max_new_tokens`	int	要生成的最大令牌数，不计提示词中的令牌。
`repetition_penalty`	float	用于控制生成文本中重复内容的重复惩罚；1.0 表示不重复，0 表示重复。
`temperature`	float	模型生成文本时下一个令牌的随机性；1.0 表示高随机性，0 表示低随机性。
`top_k`	int	计算下一个令牌的概率分布时，仅考虑概率最高的前 k 个令牌。
`top_p`	float	控制模型选择下一个令牌时考虑的累积概率分布。
`use_cache`	bool	是否使用模型在生成过程中计算的中间结果。

其他参数

参数	数据类型	说明
`max_length`	int	生成令牌的最大数量。对应输入提示词中的令牌数 + `max_new_tokens`。如果设置了 `max_new_tokens`，其效果将被 `max_new_tokens` 覆盖。
`min_length`	int	生成令牌的最小数量。对应输入提示词中的令牌数 + `min_new_tokens`。如果设置了 `min_new_tokens`，其效果将被 `min_new_tokens` 覆盖。
`min_new_tokens`	int	生成令牌的最小数量，不计提示词中的令牌。
`early_stop`	bool	控制基于 beam 的方法的停止条件。True：出现 `num_beams` 个完整候选项时停止生成。False：应用启发式方法，在不太可能找到更好的候选项时停止生成。
`num_beams`	int	beam search 使用的 beam 数量。1 表示不使用 beam search。

参数	数据类型	说明
`max_time`	int	允许计算运行的最长时间，以秒为单位。
`num_beam_groups`	int	将 `num_beams` 分组，以确保不同 beam 组之间的多样性。
`diversity_penalty`	float	启用 `num_beam_groups` 时生效。此参数在组之间应用多样性惩罚，以确保每组生成的内容尽可能不同。
`penalty_alpha`	float	当 `penalty_alpha` 大于 0 且 `top_k` 大于 1 时启用对比搜索。`penalty_alpha` 值越大，对比惩罚越强，生成的文本越有可能符合预期。如果 `penalty_alpha` 值设置得过大，可能会导致生成的文本过于单一。
`typical_p`	float	局部典型性用于衡量：给定已生成的部分文本，预测下一个目标令牌的条件概率与预测下一个随机令牌的预期条件概率之间的相似度。如果设置为小于 1 的浮点数，将保留概率总和达到或超过 `typical_p` 的最小局部典型令牌集用于生成。
`epsilon_cutoff`	float	如果设置为严格介于 0 和 1 之间的浮点数，则仅采样条件概率大于 `epsilon_cutoff` 的令牌。建议值范围为 3e-4 到 9e-4，具体取决于模型大小。
`eta_cutoff`	float	Eta 采样是局部典型采样和 epsilon 采样的混合方式。如果设置为严格介于 0 和 1 之间的浮点数，则仅当令牌大于 `eta_cutoff` 或 sqrt(`eta_cutoff`) _ exp(-entropy(softmax(next_token_logits))) 时才会考虑该令牌。建议值范围为 3e-4 到 2e-3，具体取决于模型大小。
`repetition_penalty`	float	重复惩罚参数。1.0 表示无惩罚。

有关更多参数，请参阅文本生成参数配置↗。

图像生成

预置参数

参数	数据类型	说明
`num_inference_steps`	int	去噪步数。去噪步数越多，通常图像质量越高，但推理速度越慢。
`use_cache`	bool	是否使用模型在生成过程中计算的中间结果。

其他参数

参数	数据类型	说明
`height`	int	生成图像的高度，以像素为单位。
`width`	int	生成图像的宽度，以像素为单位。
`guidance_scale`	float	用于调整生成图像的质量与多样性之间的平衡。值越大，多样性越高，但质量越低；建议范围为 7 到 8.5。
`negative_prompt`	str or List[str]	用于指示图像生成时不应包含哪些内容。

有关更多参数，请参阅图像生成参数配置↗。

文本分类

预置参数

参数	数据类型	说明
`top_k`	int	得分最高的类型标签数量。如果提供的数量为 None 或高于模型配置中的可用标签数量，则默认返回全部标签。
`use_cache`	bool	是否使用模型在生成过程中计算的中间结果。

有关更多参数，请参阅文本分类参数配置↗

其他参考资料

图像分类参数配置 ↗

对话参数配置 ↗

摘要参数配置 ↗

翻译参数配置 ↗

Text2Text 生成参数配置 ↗

图生图参数配置 ↗

Alauda AI

操作指南

使用 CLI 创建推理服务

前提条件

创建 InferenceService

检查 InferenceService 状态

使用 OpenAI 兼容 API 测试 InferenceServ

从 Kubeflow 监控面板部署推理服

简介

前提条件

访问 Endpoints UI

部署新模型

验证

扩展推理运行

简介

场景

前提条件

标准工作流（示

特定运行时示例

运行时配置对比

使用 KServe ModelCar 进行模型存储

概述

使用 OCI Containers 进行模型存储的优势

先决条件

将模型打包为 OCI Image

从 OCI Image 部署模型

最佳实践

故障排查

结论

配置推理服务外部访问

简介

步骤

配置推理服务

简介

前提条件

使用 KEDA 为推理服务配置自动扩缩容

简介

前提条件

步骤

基于 CUDA

简介

步骤

使用 Kueue 调度推理服务

前提条件

为 vLLM Inference Services 启用 Expert Parall

简介

vLLM 推理服

操作步骤
何时使用 Expert Parallel
前提条件和限制
EP 配置概览
配置单节点 InferenceService
查看已配置的 Spec
关于多节点部署
参考资料
简介
在决定之前
本指南在 Alaud
推荐起始点
前提条件
配置面
在 Alauda AI 上
端到端示例
验证并衡量影响
回滚
故障排查
注意事项和已知
参考资料

使用 CLI 创建推理服务

本文档介绍如何使用命令行创建 kserve 推理服务，并提供示例 YAML。

目录

前提条件

创建 InferenceService

检查 InferenceService 状态

使用 OpenAI 兼容 API 测试 InferenceService

前提条件

- 你可以访问已安装 KServe 的 Kubernetes 集群。
- 你已配置 `kubectl` 以便与集群通信。
- 你拥有一个具备创建 `InferenceService` 资源权限的命名空间。

创建 InferenceService

创建一个名为 `qwen-2-vllm.yaml` 的 YAML 文件，内容如下：

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  annotations:
    # 1. Optional: Model Repo reference annotations
    aml-model-repo: Qwen2.5-0.5B-Instruct
    aml-model-repo-branch: main
    aml-model-repo-id: "5010"
    aml-model-repo-tag-commit: 81e07c4246b054b3f3bd2b2c998fdd6da98b1d7f
    aml-pipeline-tag: text-generation
    cpaas.io/description: ""
    serving.kserve.io/deploymentMode: Standard
  labels:
    # 2. Required: Model Repo related labels
    aml-model-group: kubeflow-admin-cpaas-io
    aml-model-repo: Qwen2.5-0.5B-Instruct
    aml-model-subgroup: amlmodels
    aml-pipeline-tag: text-generation
    aml.cpaas.io/runtime-type: vllm
    service.subdomain: "qwen-2-mlops-demo-ai-test"
    # 3. Required: service name and namespace
  name: qwen-2
  namespace: mlops-demo-ai-test
spec:
  predictor:
    # 4. Affinity settings for GPU
    affinity:
      nodeAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - preference:
            matchExpressions:
            - key: nvidia.com/cuda.runtime.major
              operator: In
              values:
              - "12"
            - key: nvidia.com/cuda.runtime.minor
              operator: Gt
              values:
              - "5"
          weight: 100
    maxReplicas: 1
    minReplicas: 1
    model:
```

```
      command:
      - bash
      - -c
      - |
        set -ex
        # 1. check num GPUs
        GPU_COUNT=$(python3 -c "import torch; print(torch.cuda.device_cou
nt())")
        echo "Starting serving model name: ${MODEL_NAME}, num gpus: ${GPU
_COUNT}"
        if [ ${GPU_COUNT} -lt 1 ]; then
            echo "No GPUs found. Please check if the container have aquir
ed any GPU device"
            exit 1
        fi

        # 2. check model path
        MODEL_DIR="/mnt/models/${MODEL_NAME}"
        # a. using git lfs storage initializer, model will be in /mnt/mod
els/<model_name>
        # b. using hf storage initializer, model will be in /mnt/models
        if [ ! -d "${MODEL_DIR}" ]; then
            MODEL_DIR="/mnt/models"
            echo "[WARNING] Model directory ${MODEL_DIR}/${MODEL_NAME} no
t found, using ${MODEL_DIR} instead"
        fi

        # 3. check if using gguf models
        c=`find "${MODEL_DIR}" -maxdepth 1 -type f -name '*.gguf' | wc -l
`
        echo "find ${c} gguf files"
        if [ "${c}" -gt 1 ]; then
            echo "[ERROR] More than one gguf file found in ${MODEL_DIR}"
            echo "Currently, vllm only supports loading single-file GGUF
models. If you have a multi-files GGUF model, you can use gguf-split tool
to merge them to a single-file model."
            exit 1
        elif [ "${c}" -eq 1 ]; then
            n=`find "${MODEL_DIR}" -maxdepth 1 -type f -name '*.gguf' -pr
int`
            echo "[INFO] Using GGUF model file: ${n}"
            MODEL_PATH="${n}"
        else
            echo "[INFO] Using standard model directory"
```

```
        MODEL_PATH="${MODEL_DIR}"
    fi

    # 4. launch vllm server
    if [ "$ENABLE_CHUNKED_PREFILL" = "False" ]; then
        PARAM_ENABLE_CHUNKED_PREFILL="--no-enable-chunked-prefill"
    else
        PARAM_ENABLE_CHUNKED_PREFILL="--enable-chunked-prefill"
    fi
    if [ "$ENFORCE_EAGER" = "True" ]; then
        PARAM_ENFORCE_EAGER="--enforce-eager"
    else
        PARAM_ENFORCE_EAGER=""
    fi
    python3 -m vllm.entrypoints.openai.api_server \
    --port 8080 \
    --served-model-name {{.Name}} {{.Namespace}}/{{.Name}} \
    --model ${MODEL_PATH} \
    --dtype ${DTYPE} \
    --gpu-memory-utilization ${GPU_MEMORY_UTILIZATION} \
    --tensor-parallel-size ${GPU_COUNT} \
    ${PARAM_ENFORCE_EAGER} ${PARAM_ENABLE_CHUNKED_PREFILL} $@
  - bash
  env:
  - name: DTYPE
    value: half
  - name: ENABLE_CHUNKED_PREFILL
    value: "False"
  - name: ENFORCE_EAGER
    value: "True"
  - name: GPU_MEMORY_UTILIZATION
    value: "0.95"
  - name: MODEL_NAME
    value: '{{ index .Annotations "aml-model-repo" }}'
  modelFormat:
    name: transformers
  name: ""
  protocolVersion: v2
  # 4. Required: set resources used by this service
  resources:
    limits:
      cpu: "4"
      ephemeral-storage: 10Gi
      memory: 16Gi
```

```
          nvidia.com/gpualloc: "1"
          nvidia.com/gpucores: "50"
          nvidia.com/gpumem: "8192"
        requests:
          cpu: "1"
          memory: 2Gi
      # 5. Required: select "ClusterServingRuntime" and the storageUri
      runtime: aml-vllm-0.9.2-cuda-12.6
      storageUri: hf://kubeflow-admin-cpaas-io/Qwen2.5-0.5B-Instruct:81e0
7c4246b054b3f3bd2b2c998fdd6da98b1d7f
    securityContext:
      seccompProfile:
        type: RuntimeDefault
```

然后使用 `kubectl` 将该 YAML 文件应用到集群：

```
kubectl apply -f qwen-2-vllm.yaml -n <your-namespace>
```

检查 InferenceService 状态

你可以使用以下命令检查 InferenceService 的状态：

```
kubectl get inferenceservice qwen-2 -n <your-namespace>
```

输出应类似如下：

```
NAME     URL                                          READY   PREV   LATEST
PREVROLLEDOUTREVISION   LATESTREADYREVISION   AGE
qwen-2   http://qwen-2.<your-namespace>.example.com True
2m
```

revision 列保持为空，因为该服务运行在 `Standard` 部署模式下， 这种模式创建的是普通的 Kubernetes Deployment，而不是 Knative revision。

当 `READY` 列显示 `True` 时，InferenceService 即可接收请求。

使用 OpenAI 兼容 API 测试 InferenceService

当 InferenceService 就绪后，你可以使用 OpenAI 兼容 API 对其进行测试。

1. 获取 InferenceService 的 URL：

```
SERVICE_URL=$(kubectl get inferenceservice qwen-2 -n <your-namespace> -
o jsonpath='{.status.url}')
echo $SERVICE_URL
```

2. 使用 `curl` 向 `/v1/chat/completions` 端点发送请求：

```
curl -X POST ${SERVICE_URL}/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "qwen-2",
    "messages": [
      {"role": "system", "content": "You are a helpful assistant."},
      {"role": "user", "content": "What is Kubernetes?"}
    ],
    "max_tokens": 50,
    "temperature": 0.7
  }'
```

注意：请求体中的 `model` 参数应与 InferenceService YAML 中配置的 `--served-model-name` 参数一致（在本例中为 `qwen-2` 或 `mlops-demo-ai-test/qwen-2`）。

响应应包含模型生成的文本。

从 Kubeflow 监控面板部署推理服务

目录

简介
前提条件
访问 Endpoints UI
部署新模型
验证

简介

Kubeflow 中央监控面板的 **KServe Endpoints** 页面可在不离开 Kubeflow 的情况下部署、管理和监控推理服务。它创建的 `serving.kserve.io/v1beta1` `InferenceService` 对象与通过使用 CLI 创建推理服务 以及 Alauda AI 控制台创建的对象相同，因此在此处创建的服务可在这三处查看。

如果您的团队已经在使用 Kubeflow 监控面板，请使用此页面。有关平台自身的部署体验——Model Catalog 和 Model Registry 集成、推理服务模板以及批量操作——请参阅管理推理服务。

前提条件

- 已安装 Kubeflow operators ，并且您可以访问 Kubeflow 中央监控面板。
- 已安装 Alauda Build of KServe。Alauda AI 通过 `kserve` 组件对其进行管理，该组件位于 `default` `AmlCluster` 中；请参阅安装 Alauda AI。
- 模型可在集群能够读取的 `storageUri` 上访问。有关支持的存储类型，请参阅模型存储。

访问 **Endpoints UI**

1. 在中央监控面板侧边栏中点击 **KServe Endpoints**。
2. 在页面顶部选择您的命名空间。
3. 您将看到已部署 InferenceServices 的列表，以及它们的状态和 URL。

部署新模型

1. 新建 **Endpoint**： 点击 **New Endpoint**。
2. **InferenceService YAML**：
   - 提供 InferenceService 的 YAML 定义。您可以使用下面的示例 YAML 作为模板。
3. 部署： 点击 **Create**。

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: my-model
  namespace: my-namespace
spec:
  predictor:
    model:
      modelFormat:
        name: 'transformers'
      runtime: aml-vllm-0.9.2-cuda-12.6
      storageUri: 'hf://model-repo/model-name'
```

验证

部署后，等待状态变为 **Ready**。

- 检查：点击模型名称，查看 YAML 详细信息和日志。
- 获取 **URL**：复制提供的端点 URL（例如 `http://model-name.namespace.svc.cluster.local/v1/models/model-name:predict` 或外部 URL）。
- 测试：使用 `curl` 或 Python 客户端发送预测请求。

扩展推理运行时

目录

简介
场景
前提条件
标准工作流（示例：Xinference）
创建推理运行时资源
发布 Xinference 推理服务并选择运行时
特定运行时示例
MLServer
Triton Inference Server
vLLM-ascend（Ascend NPU）
Ascend 910 的 Modelcar 权限模式
MindIE（Ascend NPU）
运行时配置对比

简介

本文将引导你了解如何为 Large Language Models (LLMs) 以及其他模型（如图像分类、目标检测和文本分类模型）添加新的推理运行时。

Alauda AI 内置了 `vLLM` 推理引擎。借助自定义推理运行时，你可以引入其他推理引擎，例如 Seldon MLServer↗ 和 Triton Inference Server↗。

通过引入自定义运行时，你可以扩展平台对更多模型类型和 GPU 类型的支持，并针对特定场景优化性能，以满足更广泛的业务需求。

在本节中，我们将演示如何使用自定义 Xinference↗ 服务运行时扩展当前 AI 平台，以部署 LLM 并暴露一个兼容 OpenAI 的 API。

场景

如果你遇到以下任一情况，可以考虑扩展 AI Platform 推理服务运行时：

- 支持新的模型类型：你的模型不被当前默认推理运行时 `vLLM` 原生支持。
- 与其他硬件类型兼容：你需要在 AMD GPU 或 Huawei Ascend NPU 等硬件上执行 LLM 推理。
- 针对特定场景进行性能优化：在某些推理场景下，新的运行时（例如 Xinference）可能比现有运行时提供更好的性能或资源利用率。
- 自定义推理逻辑：你需要引入自定义推理逻辑或依赖库，而这些内容很难在现有默认运行时中实现。

前提条件

开始之前，请确保你满足以下条件：

1. 你的 ACP 集群已部署并且运行正常。
2. 你的 AI Platform 版本为 **1.3** 或更高。
3. 你已准备好所需的推理运行时镜像。例如，对于 Xinference 运行时，镜像可能类似于 `xprobe/xinference:v1.2.2`（GPU）或 `xprobe/xinference:v1.2.2-cpu`（CPU）。
4. 你拥有集群管理员权限（创建 CRD 实例所需）。

标准工作流（示例：**Xinference**）

按照以下步骤扩展平台。我们使用 **Xinference** 作为基线示例来演示标准流程。

1 创建推理运行时资源

你需要根据目标硬件环境（GPU/CPU/NPU）创建相应的推理运行时 `ClusterServingRuntime` 资源。

1. 准备运行时 **YAML** 配置：
   根据你要添加的运行时类型（例如 Xinference）以及目标硬件环境，准备相应的 YAML 配置文件。以下是不同硬件环境下 Xinference 运行时的示例：

- **GPU** 运行时示例

```yaml
# This is a sample YAML for Xinference GPU runtime
apiVersion: serving.kserve.io/v1alpha1
kind: ClusterServingRuntime
metadata:
  name: aml-xinference-cuda-12.1 # Name of the runtime resource
  labels:
    cpaas.io/runtime-class: xinference # required runtime type lab
el
    cpaas.io/accelerator-type: "nvidia"
    cpaas.io/cuda-version: "12.1"
  annotations:
    cpaas.io/display-name: xinference-cuda-12.1 # Display name in
the UI
spec:
  containers:
  - name: kserve-container
    image: xprobe/xinference:v1.2.2  # Replace with your actual GP
U runtime image
    env:
    # Required across all runtimes – path to the model directory
    - name: MODEL_PATH
      value: /mnt/models/{{ index .Annotations "aml-model-repo" }}
    # The MODEL_UID parameter is optional for other runtimes.
    - name: MODEL_UID
      value: '{{ index .Annotations "aml-model-repo" }}'
    # The MODEL_ENGINE parameter is required by the Xinference run
time, while it can be omitted for other runtimes.
    - name: MODEL_ENGINE
      value: "transformers"
    # Required parameter for xinference runtime, please set it bas
ed on your model family, value: "llama" # e.g., "llama", "chatgl
m", etc.
    - name: MODEL_FAMILY
      value: ""
    command:
    - bash
    - -c
    - |
        set +e
        if [ "${MODEL_PATH}" == "" ]; then
            echo "Need to set MODEL_PATH!"
            exit 1
        fi
```

```
        if [ "${MODEL_ENGINE}" == "" ]; then
            echo "Need to set MODEL_ENGINE!"
            exit 1
        fi
        if [ "${MODEL_UID}" == "" ]; then
            echo "Need to set MODEL_UID!"
            exit 1
        fi
        if [ "${MODEL_FAMILY}" == "" ]; then
            echo "Need to set MODEL_FAMILY!"
            exit 1
        fi

        xinference-local --host 0.0.0.0 --port 8080 &
        PID=$!
        while [ true ];
        do
            curl http://127.0.0.1:8080/docs
            if [ $? -eq 0 ]; then
                break
            else
                echo "waiting xinference-local server to become ready..."
                sleep 1
            fi
        done

        set -e
        xinference launch --model_path ${MODEL_PATH} --model-engine ${MODEL_ENGINE} -u ${MODEL_UID} -n ${MODEL_FAMILY} -e http://127.0.0.1:8080 $@
        xinference list -e http://127.0.0.1:8080
        echo "model load succeeded, waiting server process: ${PID}..."
        wait ${PID}
    # Add this line to use $@ in the script:
    # see: https://unix.stackexchange.com/questions/144514/add-arguments-to-bash-c
    - bash
    resources:
      limits:
        cpu: 2 (1)
        memory: 6Gi (2)
      requests:
```

```
        cpu: 2
        memory: 6Gi
    startupProbe:
      httpGet:
        path: /docs
        port: 8080
        scheme: HTTP
      failureThreshold: 60
      periodSeconds: 10
      timeoutSeconds: 10
  supportedModelFormats:
    - name: transformers # The model format supported by the runti
me
      version: "1"
```

1 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 CPU limit。

2 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 memory limit。

- 提示：请务必将 `image` 字段的值替换为你实际准备好的运行时镜像路径。你还可以修改 `annotations.cpaas.io/display-name` 字段，以便在 AI Platform UI 中自定义该运行时的显示名称。

2. 应用 **YAML** 文件以创建资源：
   在具有集群管理员权限的终端中，执行以下命令以应用 YAML 文件并创建推理运行时资源：

```
kubectl apply -f your-xinference-runtime.yaml
```

TIP

- 重要提示：请参考上面的示例，并根据实际环境和推理需求创建/配置运行时。 这些示例仅供参考。你需要调整诸如镜像、资源 `limits` 和 `requests` 等参数，以确保运行时与模型和硬件环境兼容并高效运行。
- 注意：只有在创建运行时资源之后，才能在推理服务发布页面上使用此自定义运行时！

2 发布 Xinference 推理服务并选择运行时

一旦成功创建 Xinference 推理运行时资源，你就可以在 AI Platform 上发布 LLM 推理服务时选择并配置它。

1. 为模型配置推理框架：
   确保在即将发布的模型仓库的模型详情页面中，你已经使用 文件管理 元数据编辑功能选择了合适的 **framework**。此处选择的 framework 值必须与创建推理服务运行时时 `supportedModelFormats` 字段中包含的某个值一致。请确保模型 **framework** 值已列在推理运行时的 **`supportedModelFormats`** 字段中。
2. 进入推理服务发布页面：
   登录 AI Platform 并进入 “Inference Services” 或 “Model Deployment” 模块，然后点击 “Publish Inference Service”。
3. 选择 **Xinference** 运行时：
   在推理服务创建向导中，找到 “Runtime” 或 “Inference Framework” 选项。从下拉菜单或列表中，选择你在步骤 1 中创建的 Xinference 运行时（例如 “Xinference CPU Runtime” 或 “Xinference GPU Runtime (CUDA)”）。
4. 设置环境变量： Xinference 运行时需要特定的环境变量才能正常工作。在推理服务配置页面中，找到 “Environment Variables” 或 “More Settings” 部分，并添加以下环境变量：

   - 环境变量

参数名称	描述
`MODEL_FAMILY`	必填。指定你正在部署的 LLM 模型所属的家族类型。Xinference 使用此参数来识别并加载该模型正确的推理逻辑。例如，如果你部署的是 Llama 3 模型，将其设置为 `llama`；如果是 ChatGLM 模型，则设置为 `chatglm`。请根据模型的实际家族类型进行设置。

   - 示例：
     - 变量名：`MODEL_FAMILY`
     - 变量值：`llama`（如果你使用的是 Llama 系列模型，请查阅 文档↗ 了解更多详情。或者你也可以运行 `xinference registrations -t LLM` 来列出所有受支持的模型家族。）

特定运行时示例

了解标准工作流后，可参考以下示例查看其他运行时的具体配置。

MLServer

MLServer 运行时用途广泛，可同时用于 NVIDIA GPU 和 CPU。

```
kind: ClusterServingRuntime
apiVersion: serving.kserve.io/v1alpha1
metadata:
  annotations:
    cpaas.io/display-name: mlserver-cuda11.6-x86-arm
  creationTimestamp: 2026-01-05T07:02:33Z
  generation: 1
  labels:
    cpaas.io/accelerator-type: nvidia
    cpaas.io/cuda-version: "11.6"
    cpaas.io/runtime-class: mlserver
  name: aml-mlserver-cuda-11.6
spec:
  containers:
    - command:
        - /bin/bash
        - -lc
        - |
          if [ "$MODEL_TYPE" = "text-to-image" ]; then
            MODEL_IMPL="mlserver_diffusers.StableDiffusionRuntime"
          else
            MODEL_IMPL="mlserver_huggingface.HuggingFaceRuntime"
          fi

          MODEL_DIR="${MLSERVER_MODEL_URI}/${MLSERVER_MODEL_NAME}"
          # a. using git lfs storage initializer, model will be in /mnt/m
odels/<model_name>
          # b. using hf storage initializer, model will be in /mnt/models
          if [ ! -d "${MODEL_DIR}" ]; then
              MODEL_DIR="${MLSERVER_MODEL_URI}"
              echo "[WARNING] Model directory ${MODEL_DIR}/${MLSERVER_MOD
EL_NAME} not found, using ${MODEL_DIR} instead"
          fi

          export MLSERVER_MODEL_IMPLEMENTATION=${MODEL_IMPL}
          export MLSERVER_MODEL_EXTRA="{\"task\":\"${MODEL_TYPE}\",\"pret
rained_model\":\"${MODEL_DIR}\"}"

          mlserver start $MLSERVER_MODEL_URI $@
        - bash
      env:
        - name: MLSERVER_MODEL_URI
          value: /mnt/models
```

```
        - name: MLSERVER_MODEL_NAME
          value: '{{ index .Annotations "aml-model-repo" }}'
        - name: MODEL_TYPE
          value: '{{ index .Annotations "aml-pipeline-tag" }}'
      image: alaudadockerhub/seldon-mlserver:1.6.0-cu116-v1.3.1
      name: kserve-container
      resources:
        limits:
          cpu: 2 (1)
          memory: 6Gi (2)
        requests:
          cpu: 2
          memory: 6Gi
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
        privileged: false
        runAsNonRoot: true
        runAsUser: 1000
      startupProbe:
        failureThreshold: 60
        httpGet:
          path: /v2/models/{{ index .Annotations "aml-model-repo" }}/read
y
          port: 8080
          scheme: HTTP
        periodSeconds: 10
        timeoutSeconds: 10
  labels:
    modelClass: mlserver_sklearn.SKLearnModel
  supportedModelFormats:
    - name: mlflow
      version: "1"
    - name: transformers
      version: "1"
```

(1) 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 CPU limit。

(2) 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 memory limit。

Triton Inference Server

Triton Inference Server 运行时专为 NVIDIA GPU 设计，并支持多种模型格式。与 MLServer 类似，你需要先创建 `ClusterServingRuntime` 资源，然后再创建推理服务。

```yaml
apiVersion: serving.kserve.io/v1alpha1
kind: ClusterServingRuntime
metadata:
  annotations:
    cpaas.io/display-name: triton-cuda12-x86
  labels:
    cpaas.io/accelerator-type: nvidia
    cpaas.io/cuda-version: "12.1"
    cpaas.io/runtime-class: triton
  name: aml-triton-cuda-12
spec:
  containers:
    - command:
        - /bin/bash
        - -c
        - >
          tritonserver --log-verbose=1 --http-port=8080
          --model-repository=/mnt/models
      env:
        - name: OMP_NUM_THREADS
          value: "1"
        - name: MODEL_REPO
          value: '{{ index .Annotations "aml-model-repo" }}'
      image: alaudadockerhub/tritonserver:25.02-py3
      name: kserve-container
      resources:
        limits:
          cpu: 2 (1)
          memory: 6Gi (2)
        requests:
          cpu: 2
          memory: 6Gi
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
        privileged: false
        runAsNonRoot: true
        runAsUser: 1000
      startupProbe:
        failureThreshold: 60
        httpGet:
```

```
            path: /v2/models/{{ index .Annotations "aml-model-repo" }}/read
y
            port: 8080
            scheme: HTTP
          periodSeconds: 10
          timeoutSeconds: 10
    supportedModelFormats:
      - name: triton
        version: "1"
```

1 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 CPU limit。

2 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 memory limit。

使用说明：

1. 创建 **ClusterServingRuntime**：使用 `kubectl apply -f triton-runtime.yaml` 应用上述 YAML 配置
2. 准备模型：确保你的模型格式受 Triton 支持（例如 TensorFlow、PyTorch、ONNX）
3. 设置模型框架：在模型仓库中，将 framework 元数据设置为 `triton`，以匹配 `supportedModelFormats` 字段
4. 创建推理服务：在发布推理服务时，从运行时下拉菜单中选择 Triton 运行时

vLLM-ascend（Ascend NPU）

`vLLM-ascend` 运行时适用于 Huawei Ascend NPU。它保留了 vLLM 的 OpenAI 兼容服务风格，同时需要为可写路径和组权限额外配置一些 `InferenceService` 设置。

此示例已在 `Ascend 910B4` 上验证。它也应适用于其他 Ascend NPU 型号，但你应根据实际环境调整资源键、镜像以及相关 版本字段。

Ascend 910 的 Modelcar 权限模式

Alauda AI 默认以非 root UID `1000` 运行 KServe Modelcar。此 默认设置是为平台安全基线设计的，适用于常见 GPU 部署，例如 NVIDIA GPU 推理服务。它也适用于 Community `vLLM-ascend` 镜像在已验证的 Ascend 910 单卡部署中。

Ascend 910 单节点多卡推理有额外要求，因为 `vLLM-ascend` 使用 HCCL 进行分布式初始化。在这种场景下，请根据 运行时镜像和验证结果选择以下某一种 Modelcar 权限模式：

场景	命名空间 **PSA** `Enforce`	**Modelcar UID**	运行时镜像要求	建议
Ascend 910 单卡推理	`restricted`	`1000`	Community `vLLM-ascend` 镜像	保持默认非 root 模式
使用与 UID `1000` 兼容的镜像进行 Ascend 910 单节点多卡推理	`restricted`	`1000`	该镜像允许 UID `1000` 访问 Ascend 设备并支持 HCCL 初始化	在验证后使用非 root 模式
在 UID `1000` 兼容性尚未验证的情况下，使用社区镜像进行 Ascend 910 单节点多卡推理	`baseline`	`0`	Community `vLLM-ascend` 镜像	使用 root 模式

> **WARNING**
>
> 这些模式在同一个集群中是互斥的，因为 Modelcar UID 是 在集群级别的 KServe Modelcar 设置中配置的。对于通用工作负载以及已验证的 Ascend 910 单卡服务，请保持默认的非 root UID `1000`。仅在 Ascend 910 多卡 `vLLM-ascend` 部署确有需要时切换到 root 模式。

非 **root** 模式

非 root 模式保持平台默认的 Modelcar UID `1000`，且命名空间 可以继续将 PSA `Enforce` 设置为 `restricted`。

对于使用 Community 镜像的 Ascend 910 单卡 `vLLM-ascend` 推理，请使用此模式。 对于 Ascend 910 单节点多卡推理，仅在验证该镜像支持 UID `1000` 和 HCCL 初始化后再使用此模式。该镜像必须允许运行时进程访问 Ascend 设备 文件，并且 TP>1 / HCCL 初始化可能需要在 `/etc/passwd` 中存在匹配的 UID `1000` 用户条目。

Root 模式

当所选镜像无法使用默认 Modelcar UID `1000` 稳定运行时，root 模式是 Ascend 910 单节点多卡 `vLLM-ascend` 推理的兼容性选项。

root 模式会将集群级别的 Modelcar UID 更改为 `0`。这会影响集群中的 Modelcar 工作负载，而不仅仅是某个单独的推理服务。除非你的镜像或环境有要求，否则不要在单卡部署中使用 root 模式。

要启用 root 模式：

1. 在平台控制台中，进入 项目 > 命名空间，选择推理服务使用的 命名空间，并将 Pod Security Admission `Enforce` 设置为 `baseline`。
2. 在 `AmlCluster` 配置中将 KServe Modelcar UID 设置为 `0`：

```
spec:
  components:
    kserve:
      values:
        kserve:
          storage:
            uidModelcar: 0
```

对于 root 模式部署，不要复用强制使用 非 root UID 的运行时配置，例如 `runAsNonRoot: true` 或 `runAsUser: 1000`。下面的 `ClusterServingRuntime` 示例适用于非 root 模式。

当不再需要 root 模式时，请将集群级别的 Modelcar UID 恢复为 平台默认的非 root 值 `1000`，并根据你的工作负载安全需求设置命名空间 PSA。

1. ClusterServingRuntime

```
apiVersion: serving.kserve.io/v1alpha1
kind: ClusterServingRuntime
metadata:
  annotations:
    aml.cpaas.io/model-type: '["generative"]'
    aml.cpaas.io/user-create: "true"
    cpaas.io/display-name: vllm-ascend-cann8.5
    helm.sh/resource-policy: keep
  labels:
    cpaas.io/accelerator-type: ascend
    cpaas.io/cann-version: "8.5.1"
    cpaas.io/runtime-class: vllm
  name: aml-vllm-ascend-cann-8.5.1
spec:
  containers:
    - command:
        - bash
        - -c
        - >
          set -ex

          # 1. check model path

          MODEL_DIR="/mnt/models/${MODEL_NAME}"

          # a. using git lfs storage initializer, model will be in
          /mnt/models/<model_name>

          # b. using hf storage initializer, model will be in /mnt/models

          if [ ! -d "${MODEL_DIR}" ]; then
              MODEL_DIR="/mnt/models"
              echo "[WARNING] Model directory ${MODEL_DIR}/${MODEL_NAME}
not found, using ${MODEL_DIR} instead"
          fi


          # 2. check if using gguf models

          c=`find "${MODEL_DIR}" -maxdepth 1 -type f -name '*.gguf' | wc
-l`

          echo "find ${c} gguf files"
```

```
          if [ "${c}" -gt 1 ]; then
              echo "[ERROR] More than one gguf file found in ${MODEL_DI
R}"
              echo "Currently, vllm only supports loading single-file GGU
F models. If you have a multi-files GGUF model, you can use gguf-split to
ol to merge them to a single-file model."
              exit 1
          elif [ "${c}" -eq 1 ]; then
              n=`find "${MODEL_DIR}" -maxdepth 1 -type f -name '*.gguf' -
print`
              echo "[INFO] Using GGUF model file: ${n}"
              MODEL_PATH="${n}"
          else
              echo "[INFO] Using standard model directory"
              MODEL_PATH="${MODEL_DIR}"
          fi


          # 3. initialize Ascend toolkit environment

          if [[ -f "/usr/local/Ascend/ascend-toolkit/set_env.sh" ]]; then
              source /usr/local/Ascend/ascend-toolkit/set_env.sh
          else
              echo "ascend-toolkit package is incomplete please check i
t."
              exit 1
          fi


          # 4. launch vllm server

          python3 -m vllm.entrypoints.openai.api_server \

          --port 8080 \

          --served-model-name {{.Name}} {{.Namespace}}/{{.Name}} \

          --model ${MODEL_PATH} \

          --gpu-memory-utilization ${GPU_MEMORY_UTILIZATION} \

          $@
        - bash
```

```
      env:
        - name: MODEL_NAME
          value: '{{ index .Annotations "aml-model-repo" }}'
        - name: GPU_MEMORY_UTILIZATION
          value: "0.95"
        - name: HOME (1)
          value: /tmp
        - name: USER (2)
          value: vllm
      image: quay.io/ascend/vllm-ascend:v0.18.0rc1
      name: kserve-container
      ports:
        - containerPort: 8080
          name: http1
          protocol: TCP
      resources:
        limits:
          cpu: 2 (3)
          memory: 6Gi (4)
        requests:
          cpu: 2
          memory: 6Gi
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop:
            - ALL
        privileged: false
        runAsNonRoot: true
        runAsUser: 1000 (5)
        seccompProfile:
          type: RuntimeDefault
      startupProbe:
        exec:
          command:
            - sh
            - -c
            - >
              curl -s -o /dev/null -w "%{http_code}" -X POST
              "http://127.0.0.1:8080/v1/completions" -H "Content-Type:
              application/json" -d '{"model": "{{ .Name }}", "prompt": "p
ing"}'
              | grep -q "200"
        failureThreshold: 60
```

```
        periodSeconds: 10
        timeoutSeconds: 180
      volumeMounts:
        - mountPath: /dev/shm
          name: devshm
  protocolVersions:
    - v2
  supportedModelFormats:
    - name: transformers
      version: "1"
  volumes:
    - emptyDir:
        medium: Memory
        sizeLimit: 1Gi
      name: devshm
```

1 `HOME` 将临时文件和缓存指向 `/tmp`，该目录对运行时容器可写。

2 `USER` 可防止 Python 的 `getpass.getuser()` 回退逻辑针对容器 UID 查询 `/etc/passwd`。这可避免在以非 root UID 运行且没有匹配 passwd 条目的镜像中发生启动失败，同时仍允许 `torch_npu` 正常自动加载。对于 TP>1 / HCCL 初始化，仍需验证镜像是否需要在 `/etc/passwd` 中存在真实的 UID `1000` 条目。

3 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 CPU limit。

4 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 memory limit。

5 `runAsUser: 1000` 使运行时容器与平台默认的 Modelcar UID 以及常见的 Ascend 设备权限模型保持一致。

2. InferenceService 示例所需变更

上面的 `ClusterServingRuntime` 中已经设置了 `HOME` 和 `USER` 环境变量，因此所有使用该运行时的服务都会继承它们。发布 `vLLM-ascend` 推理服务时，应根据 Modelcar 权限模式选择相应的 `InferenceService` 安全上下文。

对于非 root 模式，请添加 `fsGroup: 1000` 和 `supplementalGroups: [1000]`，以便 服务能够通过预期的组权限访问 Ascend 设备文件：

```yaml
kind: InferenceService
apiVersion: serving.kserve.io/v1beta1
metadata:
  name: qwen35
  namespace: demo
  annotations:
    aml-model-repo: Qwen3.5-0.8B
    modelFormat: transformers
    serving.kserve.io/deploymentMode: Standard
  labels:
    aml.cpaas.io/runtime-type: vllm
spec:
  predictor:
    model:
      modelFormat:
        name: transformers
      protocolVersion: v2
      resources:
        limits:
          cpu: "4" (1)
          huawei.com/Ascend910B4: "1"
          memory: 16Gi (2)
        requests:
          cpu: "2"
          memory: 8Gi
      runtime: aml-vllm-ascend-0.18.0rc1
      storageUri: pvc://qwen35/Qwen3.5-0.8B
    securityContext:
      fsGroup: 1000 (3)
      seccompProfile:
        type: RuntimeDefault
      supplementalGroups:
        - 1000 (4)
```

(1) 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 CPU limit。

(2) 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 memory limit。

(3) `fsGroup: 1000` 会使挂载文件继承组 `1000`，有助于将文件权限与允许访问 Ascend 设备的组保持一致。

4 `supplementalGroups: [1000]` 会将容器进程添加到组 `1000`，从而可以按预期的组权限访问 Ascend 设备及相关挂载文件。

对于 root 模式的 Modelcar 部署，不需要 `UID 1000` 组设置。 将 `InferenceService` 安全上下文保持为最小配置：

```
spec:
  predictor:
    model:
      runtime: aml-vllm-ascend-0.18.0rc1
      storageUri: oci://<registry>/<repository>:<tag>
    securityContext:
      seccompProfile:
        type: RuntimeDefault
```

MindIE（Ascend NPU）

MindIE 专为 Huawei Ascend 硬件设计。它在资源管理和元数据方面的配置差异较大。

此示例已在 `Ascend 310P` 上验证。它也应适用于其他 Ascend NPU 型号，但你应根据实际环境调整镜像、资源配置 以及相关版本字段。

1.ClusterServingRuntime

```yaml
# This is a sample YAML for Ascend NPU runtime
kind: ClusterServingRuntime
apiVersion: serving.kserve.io/v1alpha1
metadata:
  annotations:
    cpaas.io/display-name: mindie-2.2RC1
  labels:
    cpaas.io/accelerator-type: npu
    cpaas.io/cann-version: 8.3.0
    cpaas.io/runtime-class: mindie
  name: mindie-2.2rc1-310p
spec:
  containers:
    - command:
        - bash
        - -c
        - |
          REAL_SCRIPT=$(echo "$RAW_SCRIPT" | sed 's/__LT__/\x3c/g')
          echo "$REAL_SCRIPT" > /tmp/startup.sh
          chmod +x /tmp/startup.sh

          if [ ! -d "${MODEL_PATH}" ]; then
              MODEL_DIR="/mnt/models"
              echo "[WARNING] Model directory ${MODEL_PATH} not found, us
ing ${MODEL_DIR} instead"
              MODEL_PATH="${MODEL_DIR}"
          fi

          CONFIG_FILE="${MODEL_PATH}/config.json"
          echo "Checking for file: ${CONFIG_FILE}"

          ls -ld "${MODEL_PATH}"
          chmod -R 755 "${MODEL_PATH}"
          echo "Fixing MODEL_PATH permission..."
          ls -ld "${MODEL_PATH}"

          /tmp/startup.sh --model-name "${MODEL_NAME}" --model-path "${MO
DEL_PATH}" --ip "${MY_POD_IP}"
      env:
        - name: RAW_SCRIPT
          value: |
            #!/bin/bash
            #
```

```
            #  Copyright 2024 Huawei Technologies Co., Ltd
            #
            #  Licensed under the Apache License, Version 2.0 (the "Licen
se");
            #  you may not use this file except in compliance with the Li
cense.
            #  You may obtain a copy of the License at
            #
            #  http://www.apache.org/licenses/LICENSE-2.0
            #
            #  Unless required by applicable law or agreed to in writing,
software
            #  distributed under the License is distributed on an "AS IS"
BASIS,
            #  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either expre
ss or implied.
            #  See the License for the specific language governing permis
sions and
            #  limitations under the License.
            #  ============================================================
==================
            #

            ##
            # Script Instruction
            ##
            ### Name:
            ### run_mindie.sh - Use to Start MindIE Service given a speci
fic model
            ###
            ### Usage:
            ###   bash run_mindie.sh --model-name xxx --model-path /path/
to/model
            ###
            ### Required:
            ###   --model-name            :Given a model name to identif
y MindIE Service.
            ###   --model-path            :Given a model path which cont
ain necessary files such as yaml/conf.json/tokenizer/vocab etc.
            ### Options:
            ###   --help                  :Show this message.
            ###   --ip                    :The IP address bound to the M
indIE Server business plane RESTful interface,default value: 127.0.0.1.
            ###   --port                  :The port bound to the MindIE
```

```
Server business plane RESTful interface,default value: 1025.
            ###   --management-ip          :The IP address bound to the M
indIE Server management plane RESTful interface,default value: 127.0.0.2.
            ###   --management-port        :The port bound to the MindIE
Server management plane RESTful interface,default value: 1026.
            ###   --metrics-port           :The port bound to the perform
ance indicator monitoring interface,default value: 1027.
            ###   --max-seq-len            :Maximum sequence length,defau
lt value: 2560.
            ###   --max-iter-times         :The global maximum output len
gth of the model,default value: 512.
            ###   --max-input-token-len    :The maximum length of the tok
en id,default value: 2048.
            ###   --max-prefill-tokens     :Each time prefill occurs, the
total number of input tokens in the current batch,default value: 8192
            ###   --truncation             :Whether to perform parameter
rationalization check interception,default value: false.
            ###   --template-type          :Reasoning type,default value:
"Standard"
            ###   --max-preempt-count      :The upper limit of the maximu
m number of preemptible requests in each batch,default value: 0.
            ###   --support-select-batch   :Batch selection strategy,defa
ult value: false.
            ###   --npu-mem-size           :This can be used to apply for
the upper limit of the KV Cache size in the NPU,default value: 8.
            ###   --max-prefill-batch-size :The maximum prefill batch siz
e,default value: 50.
            ###   --world-size             :Enable several cards for infe
rence.
            ###                             1. If it is not set, the para
llel config in the YAML file is obtained by default. Set worldsize = dp*m
p*pp.
            ###                             2. If set, modify the paralle
l config in the YAML file. set parallel config: dp:1 mp:worldSize pp:1
            ###   --ms-sched-host          :MS Scheduler IP address,defau
lt value: 127.0.0.1.
            ###   --ms-sched-port          :MS Scheduler port,default val
ue: 8119.
            ###   For more details about config description, please check
MindIE homepage: https://www.hiascend.com/document/detail/zh/mindie/10RC
3/mindiellm/llmdev/mindie_llm0004.html
            help() {
              awk -F'### ' '/^###/ { print $2 }' "$0"
            }
```

```
            if [[ $# == 0 ]] || [[ "$1" == "--help" ]]; then
              help
              exit 1
            fi

            ##
            # Get device info
            ##
            total_count=$(npu-smi info -l | grep "Total Count" | awk -F
':' '{print $2}' | xargs)

            if [[ -z "$total_count" ]]; then
                echo "Error: Unable to retrieve device info. Please check
if npu-smi is available for current user (id 1001), or if you are specify
ing an occupied device."
                exit 1
            fi

            echo "$total_count device(s) detected!"

            ##
            # Set toolkit envs
            ##
            echo "Setting toolkit envs..."
            if [[ -f "/usr/local/Ascend/ascend-toolkit/set_env.sh" ]];the
n
                        source /usr/local/Ascend/ascend-toolkit/set_e
nv.sh
                    else
                        echo "ascend-toolkit package is incomplete pl
ease check it."
                        exit 1
                    fi
            echo "Toolkit envs set succeeded!"

            ##
            # Set MindIE envs
            ##
            echo "Setting MindIE envs..."
            if [[ -f "/usr/local/Ascend/mindie/set_env.sh" ]];then
                        source /usr/local/Ascend/mindie/set_env.sh
                    else
                        echo "mindie package is incomplete please che
```

```
ck it."
                        exit 1
                fi
            echo "MindIE envs set succeeded!"

            ##
            # Default MS envs
            ##

            # Set PYTHONPATH
            MF_SCRIPTS_ROOT=$(realpath "$(dirname "$0")")
            export PYTHONPATH=$MF_SCRIPTS_ROOT/../:$PYTHONPATH

            ##
            # Receive args and modify config.json
            ##
            export MIES_INSTALL_PATH=/usr/local/Ascend/mindie/latest/mind
ie-service
            CONFIG_FILE=${MIES_INSTALL_PATH}/conf/config.json
            echo "MindIE Service config path:$CONFIG_FILE"
            #default config
            BACKEND_TYPE="atb"
            MAX_SEQ_LEN=2560
            MAX_PREFILL_TOKENS=8192
            MAX_ITER_TIMES=512
            MAX_INPUT_TOKEN_LEN=2048
            TRUNCATION=false
            HTTPS_ENABLED=false
            MULTI_NODES_INFER_ENABLED=false
            NPU_MEM_SIZE=8
            MAX_PREFILL_BATCH_SIZE=50
            TEMPLATE_TYPE="Standard"
            MAX_PREEMPT_COUNT=0
            SUPPORT_SELECT_BATCH=false
            IP_ADDRESS="127.0.0.1"
            PORT=8080
            MANAGEMENT_IP_ADDRESS="127.0.0.2"
            MANAGEMENT_PORT=1026
            METRICS_PORT=1027

            #modify config
            while [[ "$#" -gt 0 ]]; do
                case $1 in
                    --model-path) MODEL_WEIGHT_PATH="$2"; shift ;;
```

```
                    --model-name) MODEL_NAME="$2"; shift ;;
                    --max-seq-len) MAX_SEQ_LEN="$2"; shift ;;
                    --max-iter-times) MAX_ITER_TIMES="$2"; shift ;;
                    --max-input-token-len) MAX_INPUT_TOKEN_LEN="$2"; shif
t ;;
                    --max-prefill-tokens) MAX_PREFILL_TOKENS="$2"; shift
;;
                    --truncation) TRUNCATION="$2"; shift ;;
                    --world-size) WORLD_SIZE="$2"; shift ;;
                    --template-type) TEMPLATE_TYPE="$2"; shift ;;
                    --max-preempt-count) MAX_PREEMPT_COUNT="$2"; shift ;;
                    --support-select-batch) SUPPORT_SELECT_BATCH="$2"; sh
ift ;;
                    --npu-mem-size) NPU_MEM_SIZE="$2"; shift ;;
                    --max-prefill-batch-size) MAX_PREFILL_BATCH_SIZE
="$2"; shift ;;
                    --ip) IP_ADDRESS="$2"; shift ;;
                    --port) PORT="$2"; shift ;;
                    --management-ip) MANAGEMENT_IP_ADDRESS="$2"; shift ;;
                    --management-port) MANAGEMENT_PORT="$2"; shift ;;
                    --metrics-port) METRICS_PORT="$2"; shift ;;
                    --ms-sched-host) ENV_MS_SCHED_HOST="$2"; shift ;;
                    --ms-sched-port) ENV_MS_SCHED_PORT="$2"; shift ;;
                    *)
                        echo "Unknown parameter: $1"
                        echo "Please check your inputs."
                        exit 1
                        ;;
                esac
                shift
            done

            if [ -z "$MODEL_WEIGHT_PATH" ] || [ -z "$MODEL_NAME" ]; then
                echo "Error: Both --model-path and --model-name are requi
red."
                exit 1
            fi
            MODEL_NAME=${MODEL_NAME:-$(basename "$MODEL_WEIGHT_PATH")}
            echo "MODEL_NAME is set to: $MODEL_NAME"

            WORLD_SIZE=$total_count
            NPU_DEVICE_IDS=$(seq -s, 0 $(($WORLD_SIZE - 1)))

            #validate config
```

```
            #validate config
            if [[ "$BACKEND_TYPE" != "atb" ]]; then
                echo "Error: BACKEND must be 'atb'. Current value: $BACKE
ND_TYPE"
                exit 1
            fi

            if [[ ! "$IP_ADDRESS" =~ ^([0-9]{1,3}\.){3}[0-9]{1,3}$ ]] ||
              [[ ! "$MANAGEMENT_IP_ADDRESS" =~ ^([0-9]{1,3}\.){3}[0-9]{1,
3}$ ]]; then
                echo "Error: IP_ADDRESS and MANAGEMENT_IP_ADDRESS must be
valid IP addresses. Current values: IP_ADDRESS=$IP_ADDRESS, MANAGEMENT_IP
_ADDRESS=$MANAGEMENT_IP_ADDRESS"
                exit 1
            fi

            if [[ ! "$PORT" =~ ^[0-9]+$ ]] || (( PORT __LT__ 1025 || PORT
> 65535 )) ||
              [[ ! "$MANAGEMENT_PORT" =~ ^[0-9]+$ ]] || (( MANAGEMENT_POR
T __LT__ 1025 || MANAGEMENT_PORT > 65535 )); then
                echo "Error: PORT and MANAGEMENT_PORT must be integers be
tween 1025 and 65535. Current values: PORT=$PORT, MANAGEMENT_PORT=$MANAGE
MENT_PORT"
                exit 1
            fi

            if [ "$MAX_PREFILL_TOKENS" -lt "$MAX_SEQ_LEN" ]; then
                MAX_PREFILL_TOKENS=$MAX_SEQ_LEN
                echo "MAX_PREFILL_TOKENS was less than MAX_SEQ_LEN. Setti
ng MAX_PREFILL_TOKENS to $MAX_SEQ_LEN"
            fi

            MODEL_CONFIG_FILE="${MODEL_WEIGHT_PATH}/config.json"
            if [ ! -f "$MODEL_CONFIG_FILE" ]; then
                echo "Error: config.json file not found in $MODEL_WEIGHT_
PATH."
                exit 1
            fi
            chmod 600 "$MODEL_CONFIG_FILE"
            #update config file
            chmod u+w ${MIES_INSTALL_PATH}/conf/
            sed -i "s/\"backendType\"\s*:\s*\"[^\"]*\"/\"backendType\":
\"$BACKEND_TYPE\"/" $CONFIG_FILE
            sed -i "s/\"modelName\"\s*:\s*\"[^\"]*\"/\"modelName\": \"$MO
DEL_NAME\"/" $CONFIG_FILE
```

```
DEL_NAME\"/" $CONFIG_FILE
            sed -i "s|\"modelWeightPath\"\s*:\s*\"[^\"]*\"|\"modelWeightPath\": \"$MODEL_WEIGHT_PATH\"|" $CONFIG_FILE
            sed -i "s/\"maxSeqLen\"\s*:\s*[0-9]*/\"maxSeqLen\": $MAX_SEQ_LEN/" "$CONFIG_FILE"
            sed -i "s/\"maxPrefillTokens\"\s*:\s*[0-9]*/\"maxPrefillTokens\": $MAX_PREFILL_TOKENS/" "$CONFIG_FILE"
            sed -i "s/\"maxIterTimes\"\s*:\s*[0-9]*/\"maxIterTimes\": $MAX_ITER_TIMES/" "$CONFIG_FILE"
            sed -i "s/\"maxInputTokenLen\"\s*:\s*[0-9]*/\"maxInputTokenLen\": $MAX_INPUT_TOKEN_LEN/" "$CONFIG_FILE"
            sed -i "s/\"truncation\"\s*:\s*[a-z]*/\"truncation\": $TRUNCATION/" "$CONFIG_FILE"
            sed -i "s|\(\"npuDeviceIds\"\s*:\s*\[\[\)[^]]*\(]]\)|\1$NPU_DEVICE_IDS\2|" "$CONFIG_FILE"
            sed -i "s/\"worldSize\"\s*:\s*[0-9]*/\"worldSize\": $WORLD_SIZE/" "$CONFIG_FILE"
            sed -i "s/\"httpsEnabled\"\s*:\s*[a-z]*/\"httpsEnabled\": $HTTPS_ENABLED/" "$CONFIG_FILE"
            sed -i "s/\"templateType\"\s*:\s*\"[^\"]*\"/\"templateType\": \"$TEMPLATE_TYPE\"/" $CONFIG_FILE
            sed -i "s/\"maxPreemptCount\"\s*:\s*[0-9]*/\"maxPreemptCount\": $MAX_PREEMPT_COUNT/" $CONFIG_FILE
            sed -i "s/\"supportSelectBatch\"\s*:\s*[a-z]*/\"supportSelectBatch\": $SUPPORT_SELECT_BATCH/" $CONFIG_FILE
            sed -i "s/\"multiNodesInferEnabled\"\s*:\s*[a-z]*/\"multiNodesInferEnabled\": $MULTI_NODES_INFER_ENABLED/" "$CONFIG_FILE"
            sed -i "s/\"maxPrefillBatchSize\"\s*:\s*[0-9]*/\"maxPrefillBatchSize\": $MAX_PREFILL_BATCH_SIZE/" "$CONFIG_FILE"
            sed -i "s/\"ipAddress\"\s*:\s*\"[^\"]*\"/\"ipAddress\": \"$IP_ADDRESS\"/" "$CONFIG_FILE"
            sed -i "s/\"port\"\s*:\s*[0-9]*/\"port\": $PORT/" "$CONFIG_FILE"
            sed -i "s/\"managementIpAddress\"\s*:\s*\"[^\"]*\"/\"managementIpAddress\": \"$MANAGEMENT_IP_ADDRESS\"/" "$CONFIG_FILE"
            sed -i "s/\"managementPort\"\s*:\s*[0-9]*/\"managementPort\": $MANAGEMENT_PORT/" "$CONFIG_FILE"
            sed -i "s/\"metricsPort\"\s*:\s*[0-9]*/\"metricsPort\": $METRICS_PORT/" $CONFIG_FILE
            sed -i "s/\"npuMemSize\"\s*:\s*-*[0-9]*/\"npuMemSize\": $NPU_MEM_SIZE/" "$CONFIG_FILE"

            ##
            # Start service
            ""
```

```
            ##
            echo "Current configurations are displayed as follows:"
            cat $CONFIG_FILE
            npu-smi info -m > ~/device_info

            ${MIES_INSTALL_PATH}/bin/mindieservice_daemon
        - name: MODEL_NAME
          value: '{{ index .Annotations "aml-model-repo" }}'
        - name: MODEL_PATH
          value: /mnt/models/{{ index .Annotations "aml-model-repo" }}
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
      image: swr.cn-south-1.myhuaweicloud.com/ascendhub/mindie:2.2.RC1-300I-Duo-py311-openeuler24.03-lts
      name: kserve-container
      resources:
        limits:
          cpu: 2 (1)
          memory: 6Gi (2)
        requests:
          cpu: 2
          memory: 6Gi
      volumeMounts:
        - mountPath: /dev/shm
          name: dshm
      startupProbe:
        failureThreshold: 60
        httpGet:
          path: /v1/models
          port: 8080
          scheme: HTTP
        periodSeconds: 10
        timeoutSeconds: 180
  supportedModelFormats:
    - name: transformers
      version: "1"
  volumes:
    - emptyDir:
        medium: Memory
        sizeLimit: 8Gi
      name: dshm
```

1 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 CPU limit。

2 请根据环境中的实际模型大小、运行时引擎、硬件类型和预期工作负载设置 memory limit。

2.InferenceService 的必需注解

与其他运行时不同，MindIE 在最终发布步骤中必须在 `InferenceService` 元数据中包含以下注解。这样可以确保 平台调度器正确将 NPU 硬件绑定到该服务。

配置键	值	目的
`storage.kserve.io/readonly`	`"false"`	启用对模型存储卷的写入访问。

3. 用户权限（**Root** 访问）

由于 Ascend 驱动和硬件抽象层的要求，MindIE 镜像必须以 **root** 用户运行。请确保你的 `ClusterServingRuntime` 或 `InferenceService` 安全上下文已相应配置：

注意：上面的 MindIE `ClusterServingRuntime` YAML 示例未指定 `securityContext`，这意味着容器将使用镜像的默认设置运行（通常为 root）。与显式设置 `runAsNonRoot: true` 和 `runAsUser: 1000` 的 MLServer 不同，MindIE 需要 root 权限才能访问 NPU 硬件。

运行时配置对比

在继续之前，请参考下表了解不同运行时的具体要求：

运行时	目标硬件	支持的框架	特殊要求
Xinference	CPU / NVIDIA GPU	transformers, pytorch	必须设置 `MODEL_FAMILY` 环境变量
MLServer	CPU / NVIDIA GPU	sklearn, xgboost, mlflow	标准配置
Triton	NVIDIA GPU	triton (TensorFlow,	标准配置

运行时	目标硬件	支持的框架	特殊要求
		PyTorch, ONNX, etc.)	
vLLM-ascend	Huawei Ascend NPU（已在 910B4 上验证）	transformers	必须在 `ClusterServingRuntime` 中设置 `HOME` 和 `USER`，为 Ascend 910 单卡或多卡部署选择合适的 Modelcar 权限模式，并且仅在非 root 模式下添加 UID `1000` 组设置
MindIE	Huawei Ascend NPU（已在 310P 上验证）	mindspore, transformers	必须将所需的 NPU 注解添加到 `InferenceService`

使用 KServe Modelcar 进行模型存储

目录

概述
使用 OCI Containers 进行模型存储的优势
先决条件
将模型打包为 OCI Image
选项 1：使用 Busybox Base Image（Alauda AI 推荐）
选项 2：使用 UBI Micro Base Image（Red Hat 推荐）
构建并推送模型镜像
从 OCI Image 部署模型
部署先决条件
创建 InferenceService
应用 InferenceService
验证部署
最佳实践
故障排查
常见问题
结论

概述

KServe Modelcar，也称为基于 OCI container 的模型存储，是一种在云原生环境中部署模型的强大方式。通过将模型打包为 OCI container 镜像，您可以利用 container runtime 的能力，实现更快的启动时间和更高效的资源利用率。

使用 **OCI Containers** 进行模型存储的优势

- 缩短启动时间：避免多次下载同一个模型
- 降低磁盘空间占用：减少本地下载的模型数量
- 提升模型性能：允许预先拉取镜像以加快加载速度
- 支持离线环境：非常适合互联网访问受限的环境
- 简化模型分发：使用企业内部 registry，如 Quay 或 Harbor

先决条件

- 已安装并运行 Alauda AI platform
- 已准备好可用于打包的模型文件
- 对 container registry（例如 Harbor、Quay）具有写入/推送权限
- 本地机器上已安装 Podman 或 nerdctl

将模型打包为 **OCI Image**

选项 **1**：使用 **Busybox Base Image**（**Alauda AI** 推荐）

创建一个包含以下内容的 Containerfile：

```
# Use lightweight busybox as base image
FROM busybox

# Create directory for model and set permissions
RUN mkdir -p /models && chmod 775 /models

# Note: Harbor has a file size limit per image layer.
# It is recommended to copy large files (e.g. 4GB .safetensors) in chunks of 2 files per layer.
# The last layer should copy the remaining model configuration files.
# Example:
# COPY models/model-00001-of-00004.safetensors models/model-00002-of-00004.safetensors /models/
# COPY models/model-00003-of-00004.safetensors models/model-00004-of-00004.safetensors /models/
# COPY models/*.json models/*.md models/*.txt /models/

# If size is not an issue, you can simply copy everything:
COPY models/ /models/

# According to KServe convention, model loader usually only needs image layers
# No need to keep running, but can add CMD if debugging is needed
```

选项 2：使用 **UBI Micro Base Image（Red Hat 推荐）**

创建一个包含以下内容的 Containerfile：

```
FROM registry.access.redhat.com/ubi9/ubi-micro:latest
COPY --chown=0:0 models /models
RUN chmod -R a=rX /models

# nobody user
USER 65534
```

构建并推送模型镜像

1. 创建一个用于存储模型和支持文件的临时目录：

```
cd $(mktemp -d)
```

2. 创建一个 models 文件夹（对于 OpenVINO 等框架，也可以选择创建版本子目录）：

```
mkdir -p models/1
```

3. 将模型文件复制到相应目录：

- 对于大多数框架：`cp -r your-model-folder/* models/`
- 对于 OpenVINO：`cp -r your-model-folder/* models/1/`

4. 构建 OCI container 镜像：

```
# Using Podman
podman build --format=oci -t <registry>/<repository>:<tag> .

# Using nerdctl
nerdctl build -t <registry>/<repository>:<tag> .
```

5. 将镜像推送到您的 container registry：

```
# Using Podman
podman push <registry>/<repository>:<tag>

# Using nerdctl
nerdctl push <registry>/<repository>:<tag>
```

> 注意 如果您的 repository 是私有的，请确保在上传 container image 之前已通过 registry 身份验证。

从 **OCI Image** 部署模型

部署先决条件

除上述通用先决条件外，无需其他额外先决条件。

创建 InferenceService

创建一个包含以下内容的 InferenceService YAML 文件：

```yaml
kind: InferenceService
apiVersion: serving.kserve.io/v1beta1
metadata:
  annotations:
    aml-model-repo: Qwen2.5-0.5B-Instruct 1
    aml-pipeline-tag: text-generation
    serving.kserve.io/deploymentMode: Standard
  labels:
    aml-pipeline-tag: text-generation
    aml.cpaas.io/runtime-type: vllm 2
  name: oci-demo
  namespace: demo-space 3
spec:
  predictor:
    maxReplicas: 1
    minReplicas: 1
    model:
      modelFormat:
        name: transformers
      protocolVersion: v2
      resources:
        limits:
          cpu: '2'
          ephemeral-storage: 10Gi
          memory: 8Gi
        requests:
          cpu: '2'
          memory: 4Gi
      runtime: aml-vllm-0.11.2-cpu 4
      storageUri: oci://<registry>/<repository>:<tag> 5
    securityContext:
      seccompProfile:
        type: RuntimeDefault
```

1 将 `Qwen2.5-0.5B-Instruct` 替换为您的实际模型名称。

2 `aml.cpaas.io/runtime-type: vllm` 指定了代码运行时类型。有关自定义推理运行时的更多信息，请参见 扩展推理运行时。

3 将 `demo-space` 替换为一个已存在的 namespace，或者先使用 `kubectl create namespace demo-space` 创建它。

4 将 `aml-vllm-0.11.2-cpu` 替换为平台中已安装的运行时名称（对应一个 ClusterServingRuntime CRD instance）。

5 `storageUri` 指定模型存储所在的带标签 OCI image URI。请使用完整的 registry URL。例如：`storageUri: oci://docker.io/alaudapublic/models-qwen2.5:v1`

应用 InferenceService

使用 kubectl 应用 InferenceService 配置：

```
kubectl apply -f oci-inference-service.yaml
```

验证部署

检查 InferenceService 的状态：

```
kubectl get inferenceservices -n demo-space
```

部署成功后，您应看到服务处于 `Ready` 状态。

如果服务启动失败或一直处于未就绪状态，可以通过以下方式进行排查：

1. 检查 InferenceService 事件以获取详细错误信息：

   ```
   kubectl describe inferenceservice oci-demo -n demo-space
   ```

2. 检查 predictor pod 日志：

   ```
   kubectl logs -n demo-space -l serving.kserve.io/inferenceservice=oci-demo
   ```

3. 验证模型镜像是否可以在集群本地成功拉取：

```
# On a node in the cluster
crictl pull <registry>/<repository>:<tag>
```

最佳实践

1. 模型版本管理：在 container image 中使用 tag 对模型进行版本管理
2. 镜像大小优化：使用轻量级 base image，并且仅包含必要的模型文件
3. **Registry** 管理：使用具备适当访问控制的私有 registry
4. 安全性：遵循 container 安全最佳实践，包括定期漏洞扫描
5. 缓存：利用 container registry 缓存来提升拉取速度

故障排查

常见问题

1. 权限错误：确保镜像中的模型文件具有正确的权限
2. **Ascend 910 vLLM-ascend** 权限模式：对于 Huawei Ascend 910 的 `vLLM-ascend` 部署，尤其是单节点多卡服务，请参见 Ascend 910 的 Modelcar 权限模式。
3. **Registry** 身份验证：验证集群是否能够访问 container registry

结论

使用 KServe Modelcar（基于 OCI container 的模型存储）可以在 Alauda AI platform 中提供一种高效的模型部署方式。按照本指南中的步骤，您可以将模型打包为 OCI image，并以更快的启动时间和更高的资源利用率进行部署。

配置推理服务外部访问

目录

简介

步骤

1. 查看推理服务的外部访问地址
2. 创建域名
3. 创建负载均衡器
4. 配置负载均衡器
   4.1 端口配置
   4.2 规则配置
5. 验证通过外部地址访问推理服务

简介

本文档提供了一个逐步指南，说明如何为推理服务配置外部访问，包括检查外部访问地址、创建域名、设置负载均衡器以及验证配置。

步骤

1. 查看推理服务的外部访问地址

你可以：

1. 转到服务详情页，并从 访问方式 卡片中复制地址，或者
2. 在 **YAML** 卡片中查看 `status.url` 下的地址。

2 2. 创建域名

在管理员控制台中，进入 网络 > 域名，然后点击 创建域名。

- 在 域名 字段中，输入推理服务的域名，不包括协议（例如，qwen2-0b5-kubeflow-admin-cpaas-io.my-company.com）。
- 对于 分配到（集群），选择服务所在的集群。
- 对于 分配项目，选择推理服务所在的项目。

3 3. 创建负载均衡器

（*一个负载均衡器可供多个项目共享；仅在必要时创建新的负载均衡器。*）

在管理员控制台中，进入 网络 > 负载均衡器，然后点击 创建负载均衡器。有关详细帮助文档，请参阅 Configure Load Balancer↗。

4 4. 配置负载均衡器

在 Alauda Container Platform 控制台中，进入 网络 > 负载均衡器，然后点击你刚创建的负载均衡器名称，进入其配置页面。

4.1 端口配置

为服务添加监听端口。你可以根据需要添加更多端口。

步骤 **1**：添加端口

1. 在 端口管理 区域中，点击 添加端口 按钮。
2. 在配置页面中，选择协议（**HTTP** 或 **HTTPS**）。
3. 输入相应的端口号。对于 HTTP，标准端口是 `80`。对于 HTTPS，标准端口是 `443`。

步骤 **2**：配置 **HTTPS**（如适用） 如果你选择 **HTTPS** 协议，必须选择默认证书。

1. 确保你已切换到 `istio-system` 命名空间。这对下一步至关重要。
2. 将 `knative-serving-cert` 选为 默认证书。

4.2 规则配置

为上一步中添加的端口配置转发规则。

步骤 **1**：添加规则

1. 在 规则 区域中，点击 添加规则 按钮。
2. 在配置页面中，点击 添加规则指示器 按钮，添加第一条规则。
3. 对于 类型，选择 域名。
4. 从下拉菜单中，选择你为推理服务创建的域名，例如：`qwen2-0b5-kubeflow-admin-cpaas-io.my-company.com`。

步骤 **2**：配置服务组

1. 找到 服务组 配置区域。
2. 确保 命名空间 设置为 `istio-system`。如果不是，请先将项目的命名空间切换到 `istio-system`。
3. 在 服务 下，从下拉列表中选择 `knative-ingressgateway`。
4. 选择端口 **80**。

注意： 配置 HTTPS 协议（端口 443）规则的过程与上述相同。

如需了解更详细的参数配置，请参阅 Configure Load Balancer↗。

5 5. 验证通过外部地址访问推理服务

要验证你的推理服务是否可通过外部访问，请使用下面的 `curl` 命令。请记得将占位符替换为实际的负载均衡器 **IP** 地址、端口和推理服务地址。

```
# For HTTP
curl -v --resolve "your-inference-service-domain.com:your-port:your-load-balancer-ip" \
    http://your-inference-service-domain.com/v1/models

# For HTTPS (skip `-k` if you have a valid certificate)
curl -vk --resolve "your-inference-service-domain.com:443:your-load-balancer-ip" \
    https://your-inference-service-domain.com/v1/models
```

以下是命令中各部分的含义以及你需要替换的内容：

- `your-inference-service-domain.com`：这应该是你为推理服务创建的域名（例如，`qwen2-0b5-kubeflow-admin-cpaas-io.my-company.com`）。
- `your-port`：这是负载均衡器用于监听 HTTP 流量的端口（通常为 `80`）。
- `your-load-balancer-ip`：这是负载均衡器的实际 IP 地址（例如，`192.168.137.21`）。

如果请求成功返回模型列表，则说明配置完成！如果失败，请仔细检查负载均衡器设置，或查看推理服务日志以定位问题。

配置推理服务自动扩缩容

目录

简介
前提条件
步骤
下行扩缩容配置
启用/禁用缩容到零
全局禁用缩容到零
配置缩容到零后的 Pod 保留时间
配置缩容到零的宽限期
上行扩缩容配置
配置并发阈值
配置每秒请求数（RPS）目标

简介

本文档提供了逐步指南，用于为以 **Knative** 部署模式运行的推理服务配置自动向上和向下扩缩容。通过这些设置，您可以优化资源使用、确保服务在高负载期间的可用性，并在低负载期间释放资源。

> **WARNING**
>
> 本页上的每一项设置都仅适用于以 Knative 部署模式运行的推理服务。Alauda AI 默认以 `Standard` 模式部署推理服务， 该模式不使用 Knative——在该模式下，下面的 `config-autoscaler` ConfigMap 和 `autoscaling.knative.dev/*` 注解都不会生效，并且 `minReplicas: 0` 也不会 将服务缩容到零。
>
> 如需在 `Standard` 模式下对服务进行自动扩缩容，请参阅 使用 KEDA 为推理服务设置自动扩缩容。

关于 **Autoscaler**

Knative Serving 支持两种 autoscaler：**Knative Pod Autoscaler (KPA)** 和 Kubernetes 的 **Horizontal Pod Autoscaler (HPA)**。以 Knative 模式部署的服务默认使用 Knative Pod Autoscaler (KPA)。

KPA 面向无服务器工作负载，可以根据并发请求数或 RPS（每秒请求数）快速扩容，并且可以将服务缩容到零个从节点以节省成本。HPA 更通用，通常基于 CPU 或内存使用率等指标进行扩缩容。本指南主要介绍如何通过 Knative Pod Autoscaler (KPA) 配置服务。

前提条件

1. 已安装 Knative Operator，且存在一个 `KnativeServing` 实例。请参阅 安装 Alauda AI。
2. 目标 `InferenceService` 已以 Knative 模式部署。请确认服务上的注解：

   ```
   kubectl get inferenceservice <name> -n <your-namespace> \
     -o jsonpath='{.metadata.annotations.serving\.kserve\.io/deploymentMode}{"\n"}'
   ```

   该命令必须输出 `Knative`。如果输出 `Standard` 或无输出，请改用 KEDA 指南。
3. 您有权限编辑 `knative-serving` 命名空间中的 `config-autoscaler` ConfigMap，以进行平台级设置。

步骤

1 下行扩缩容配置

本节介绍如何配置推理服务在没有流量时自动缩容到零个从节点，或保持最少数量的从节点。

启用/禁用缩容到零

您可以配置是否允许推理服务在没有流量时缩容到零个从节点。默认值为 `true`，即允许缩容到零。

使用 **InferenceService** 资源参数

在 `InferenceService` 的 `spec.predictor` 字段中，设置 `minReplicas` 参数。

- `minReplicas: 0`：允许缩容到零个从节点。
- `minReplicas: 1`：禁止缩容到零个从节点，至少保留一个从节点。

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: demo
  namespace: demo-space
spec:
  predictor:
    minReplicas: 0
    ...
```

全局禁用缩容到零

> **WARNING**
>
> 一旦禁用了平台级功能，所有服务中的 `minReplicas: 0` 配置都将被忽略。

您可以修改全局 `ConfigMap` 来禁用平台的缩容到零功能。此配置具有最高优先级，会覆盖所有单独 `InferenceService` 资源中的设置。

在 `knative-serving` 命名空间中的 `config-autoscaler` ConfigMap 里，将 `enable-scale-to-zero` 的值修改为 `"false"`

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    "helm.sh/resource-policy": keep (1)
  name: config-autoscaler
  namespace: knative-serving
data:
  enable-scale-to-zero: "false"
```

(1) 请确保该注解存在，否则您的配置将恢复为默认值。

配置缩容到零后的 Pod 保留时间

此设置决定了 autoscaler 判定缩容到零后，最后一个 Pod 保持活跃的最短时间。这有助于服务在再次接收流量时快速响应。默认值为 `0s`

您可以选择为单个服务配置此项，或修改全局 ConfigMap 使该设置对所有服务生效。

方法 **1**：使用 `InferenceService` 注解

在 `InferenceService` 的 `spec.predictor.annotations` 中，添加 `autoscaling.knative.dev/scale-to-zero-pod-retention-period` 注解。

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: qwen2
  namespace: fy-1
spec:
  predictor:
    annotations:
      autoscaling.knative.dev/scale-to-zero-pod-retention-period: "1m
5s"
    ...
```

方法 **2**：使用全局 `ConfigMap`

在 `knative-serving` 命名空间中的 `config-autoscaler` ConfigMap 里，将 `scale-to-zero-pod-retention-period` 的值修改为非负的时长字符串，例如 `"1m5s"`。

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    "helm.sh/resource-policy": keep (1)
  name: config-autoscaler
  namespace: knative-serving
data:
  scale-to-zero-pod-retention-period: "1m5s"
```

(1) 请确保该注解存在，否则您的配置将恢复为默认值。

配置缩容到零的宽限期

此设置会在流量停止后延迟移除最后一个从节点，确保 activator/routing 路径已准备就绪，并防止在过渡到零的过程中丢失请求。

> **TIP**
>
> 只有在因服务缩容到零而遇到请求丢失时，才应调整此值。它不会影响在没有流量后的最后一个从节点保留时间，也不能保证在此期间该从节点一定会被保留。

方法：使用全局 `ConfigMap`

在 `knative-serving` 命名空间中的 `config-autoscaler` ConfigMap 里，将 `scale-to-zero-grace-period` 的值修改为一个时长字符串，例如 `"40s"`。

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    "helm.sh/resource-policy": keep (1)
  name: config-autoscaler
  namespace: knative-serving
data:
  scale-to-zero-grace-period: "40s"
```

(1) 请确保该注解存在，否则您的配置将恢复为默认值。

2 上行扩缩容配置

本节介绍如何配置推理服务，以便在流量增加时自动扩容。

配置并发阈值

并发决定了每个应用副本可以同时处理的请求数。您可以使用软限制或硬限制来设置并发。

- 软限制：目标限制，在流量激增期间可以暂时超过，但会触发自动扩缩容以维持目标值。默认值为 `100`。
- 硬限制：严格的上限。当并发达到该值时，超出部分的请求将被缓冲并排队处理。默认值为 `0`，表示无限制。

> **WARNING**
>
> 如果同时指定了软限制和硬限制，将使用两者中较小的值。这可以防止 Autoscaler 的目标值高于硬限制所允许的值。

您可以选择为单个服务配置此项，或修改全局 ConfigMap 使该设置对所有服务生效。

方法 **1**：使用 `InferenceService` 资源参数

- 软限制：在 `spec.predictor` 中设置 `scaleTarget`，并将 `scaleMetric` 设为 `concurrency`。

- 硬限制：在 `spec.predictor` 中设置 `containerConcurrency`

```yaml
# Set soft and hard limits
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: demo
  namespace: demo-space
spec:
  predictor:
    scaleTarget: 200
    scaleMetric: concurrency
    containerConcurrency: 50
    ...
```

方法 **2**：使用全局 **`ConfigMap`**

- 软限制：在 `config-autoscaler` ConfigMap 中设置 `container-concurrency-target-default`。
- 硬限制：没有全局设置可用于硬限制，因为它会影响请求缓冲和排队。

目标利用率百分比

此值指定在 `metric=concurrency` 时 autoscaler 期望达到的目标百分比，从而允许在达到硬限制之前提前扩容。默认值：**`70`**。 使用 RPS 时不适用。

方法 1：使用 `InferenceService` 注解

在 `InferenceService` 的 `spec.predictor.annotations` 中，添加 `autoscaling.knative.dev/target-utilization-percentage` 注解。

```yaml
# Set target utilization by service
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: qwen2
  namespace: fy-1
spec:
  predictor:
    annotations:
      autoscaling.knative.dev/target-utilization-percentage: "80"
```

方法 **2**：使用全局 `ConfigMap`

在 `config-autoscaler` ConfigMap 中，设置 `container-concurrency-target-percentage`。

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    "helm.sh/resource-policy": keep (1)
  name: config-autoscaler
  namespace: knative-serving
data:
  container-concurrency-target-percentage: "80"
```

(1) 请确保该注解存在，否则您的配置将恢复为默认值。

配置每秒请求数（RPS）目标

您可以将扩缩容指标从并发改为每秒请求数（RPS）。默认值为 `200`。 注意：在 RPS 模式下，不会使用并发目标百分比设置。

方法 **1**：使用 `InferenceService` 资源参数

在 `spec.predictor` 中，设置 `scaleTarget` 并将 `scaleMetric` 设为 `rps`。

```yaml
# Set RPS target by service
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: qwen2
  namespace: fy-1
spec:
  predictor:
    scaleTarget: 150
    scaleMetric: rps
    ...
```

方法 **2**：使用全局 **ConfigMap**

在 `config-autoscaler` ConfigMap 中，设置 `requests-per-second-target-default`。

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  annotations:
    "helm.sh/resource-policy": keep (1)
  name: config-autoscaler
  namespace: knative-serving
data:
  requests-per-second-target-default: "200"
```

(1) 请确保该注解存在，否则您的配置将恢复为默认值。

使用 KEDA 为推理服务配置自动扩缩容

目录

简介
前提条件
授予 KServe 访问 KEDA 资源的权限
步骤
停止正在运行的 InferenceService
创建 Prometheus TriggerAuthentication
为 KEDA 配置 InferenceService
用于自动扩缩容的 vLLM 指标
验证配置

简介

在生产环境中部署机器学习模型会带来一些独特的挑战，其中最关键的一点是确保你的推理服务能够高效且可靠地应对不同水平的流量。AI 工作负载的不可预测性——例如流量可能突然激增，并且资源需求会根据输入序列长度、token 生成长度或并发请求数量等因素而变化——往往意味着传统的自动扩缩容方法难以满足需求。

仅依赖 CPU 或内存指标，可能会导致资源过度预留和浪费，或者资源不足并影响用户体验。同样，GPU 利用率既可能表示资源使用高效，也可能表示系统已接近饱和。因此，LLM 自动扩缩容的行业最佳实践已经逐步转向更具工作负载特征的指标。

本指南将通过利用 **KEDA**（Kubernetes Event-driven Autoscaling）以及由 **vLLM** 导出的自定义应用级指标，介绍如何配置 KServe 自动扩缩容。通过这种组合，推理服务可以根据真实的工作负载信号进行扩缩容，而不是依赖通用的基础设施指标。

> **INFO**
>
> KEDA 扩展了标准的 Kubernetes Horizontal Pod Autoscaler (HPA)，使应用能够基于多种事件源——包括 Prometheus 指标——从零扩展到 N 个实例，再缩回去。它引入了一个开放且可扩展的框架，使 KServe 能够基于与你的 AI 模型性能相关的几乎任何信号进行扩缩容。

前提条件

- 已安装 `Alauda AI` Operator plugin。
- 已安装 `KEDA` Operator plugin。
- 一个使用 `Standard` 模式且运行时为 vLLM 的 `InferenceService`。
- 集群中已安装并可访问 `Prometheus`。

授予 KServe 访问 KEDA 资源的权限

在继续之前，请应用以下 RBAC 资源，以允许 `kserve-controller-manager` 管理 KEDA 对象（`ScaledObject`、`TriggerAuthentication` 等）：

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: kserve-keda-manager-role
rules:
- apiGroups:
  - keda.sh
  resources:
  - "*"
  verbs:
  - "*"
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: kserve-keda-manager-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: kserve-keda-manager-role
subjects:
- kind: ServiceAccount
  name: kserve-controller-manager
  namespace: kserve
```

[安装顺序]

如果 KEDA 是在 **Alauda AI** 之后安装的，请在应用上述 RBAC 之后，重启 `kserve-controller-manager` pod（位于 `kserve` namespace 中），以便它能够发现 KEDA CRD：

```
kubectl rollout restart deployment kserve-controller-manager -n kserve
```

步骤

1 停止正在运行的 **InferenceService**

在进行更改之前，请先停止正在运行的 `InferenceService`，以避免现有 HPA 与新的由 KEDA 管理的 scaler 之间发生冲突。添加以下 annotation 以停止它：

```
kubectl annotate inferenceservice <your-isvc-name> -n <your-namespace
> \
  serving.kserve.io/stop='true'
```

> **WARNING**
>
> 如果正在运行的 `InferenceService` 已经拥有 HPA resource，那么在未先停止它的情况下切换到 KEDA 将会导致 resource 冲突。

2 创建 Prometheus TriggerAuthentication

KEDA 需要在与你的 `InferenceService` 相同的 namespace 中创建一个 `TriggerAuthentication` resource，以便通过 Prometheus 进行身份验证。

Prometheus 凭据存储在 `cpaas-system` namespace 中的平台 secret `kube-prometheus-alertmanager-basic-auth` 里。运行以下命令将其复制到你的 namespace 中：

```
kubectl create secret generic prom-basic-auth-secret \
  --namespace=<your-namespace> \
  --from-literal=username=$(kubectl get secret kube-prometheus-alertm
anager-basic-auth \
    -n cpaas-system -o jsonpath='{.data.username}' | base64 -d) \
  --from-literal=password=$(kubectl get secret kube-prometheus-alertm
anager-basic-auth \
    -n cpaas-system -o jsonpath='{.data.password}' | base64 -d)
```

然后创建引用该 secret 的 `TriggerAuthentication`：

```yaml
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: prom-basic-auth
  namespace: <your-namespace>
spec:
  secretTargetRef:
    - parameter: username
      name: prom-basic-auth-secret
      key: username
    - parameter: password
      name: prom-basic-auth-secret
      key: password
```

TIP

以下名称必须在所有资源之间保持一致：

- `prom-basic-auth-secret` — `Secret` 名称，必须与 `TriggerAuthentication` 中的 `secretTargetRef.name` 匹配。
- `prom-basic-auth` — `TriggerAuthentication` 名称，必须与 `InferenceService` spec 中的 `authenticationRef.authenticationRef.name` 匹配。

3 为 KEDA 配置 InferenceService

服务停止后，使用 KEDA 自动扩缩容配置更新 `InferenceService` manifest：

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  name: <your-isvc-name>
  namespace: <your-namespace>
  annotations:
    serving.kserve.io/deploymentMode: Standard
    serving.kserve.io/autoscalerClass: keda (1)
spec:
  predictor:
    maxReplicas: 2 (2)
    autoScaling:
      metrics:
        - type: External
          external:
            authenticationRef:
              authModes: basic
              authenticationRef:
                name: prom-basic-auth (3)
            metric:
              backend: prometheus
              query: > (4)
                sum(vllm:num_requests_running{isvc_name="<your-isvc-name>",namespace="<your-namespace>"})
              serverAddress: http://prometheus-operated.cpaas-system.svc.cluster.local:9090 (5)
            target:
              type: Value
              value: '1' (6)
    # ... rest of your predictor configuration
```

(1) 禁用内置的 KServe HPA，并将扩缩容委托给 KEDA。

(2) 设置最大副本数。要使服务能够根据流量增加自动扩容，请确保该值大于 1（例如，2）。

(3) 引用保存用于与 Prometheus 进行身份验证凭据的 `TriggerAuthentication` resource。请将 `prom-basic-auth` 替换为你实际的 `TriggerAuthentication` 名称。

(4) 返回当前负载单个数值的 PromQL 查询。请将 `<your-model-name>` 和 `<your-namespace>` 替换为你的实际值。

5 Prometheus 实例的内部地址，例如 `http://prometheus-operated.cpaas-system.svc.cluster.local:9090`。

6 每个副本的目标值。KEDA 会根据 `ceil(metricValue / value)` 计算所需的副本数。

4 用于自动扩缩容的 vLLM 指标

平台会自动收集 vLLM 指标。选择合适的指标可以说是整个配置中最关键的部分——Prometheus 查询必须返回一个单一数值，并且能够准确反映模型当前的负载。

以下 vLLM 指标通常用于自动扩缩容：

指标	描述
`vllm:num_requests_running`	当前由模型正在处理的请求数量
`vllm:num_requests_waiting`	已排队并等待处理的请求数量
`vllm:gpu_cache_usage_perc`	当前正在使用的 GPU KV cache 百分比
`vllm:e2e_request_latency_seconds_bucket`	端到端请求延迟 histogram
`vllm:time_per_output_token_seconds_bucket`	token 间延迟（Time Per Output Token，TPOT）

使用 `sum()` 聚合函数，确保查询在你部署的所有 pod 上返回单个值。例如，要根据等待中的请求数进行扩缩容：

```
sum(vllm:num_requests_waiting{isvc_name="<your-isvc-name>", namespace="<your-namespace>"})
```

这样会汇总所有 predictor pod 中的待处理请求，为 KEDA 提供一个单一的聚合信号来进行处理。

5 验证配置

应用更新后的 `InferenceService` 之后，KServe 会自动为你创建一个 KEDA `ScaledObject`。请验证一切是否正常工作：

```
# Check the ScaledObject created by KServe
kubectl get scaledobject -n <your-namespace>

# Check the KEDA-managed HPA
kubectl get hpa -n <your-namespace>

# Watch replica counts in real time
kubectl get hpa -n <your-namespace> -w
```

HPA 输出将显示当前指标值、扩缩容阈值以及当前/期望的副本数。随着推理流量增加，`TARGETS` 值会升高，副本数也会自动扩容。

基于 CUDA 版本调度推理服务

目录

简介

步骤

在节点标签中添加 CUDA 版本

基于 CUDA 版本调度推理服务

简介

在 Kubernetes 集群中，不同 GPU 节点上的 GPU 型号和 CUDA 驱动版本不一致，会导致以下问题：

1. 版本不匹配：应用程序使用的 CUDA Runtime 版本可能与某些节点上的 CUDA Driver 版本不兼容，从而导致失败或性能问题。
2. 调度挑战：原生 Kubernetes 调度器无法感知 CUDA 版本依赖关系，不能保证应用程序被调度到具有兼容版本的 GPU 节点上。
3. 维护开销高：手动管理节点与应用程序之间的 CUDA 版本依赖关系会增加运维复杂度。

本文将逐步介绍如何基于 CUDA Runtime 版本和 Nvidia Driver 版本调度推理服务。 通过这些设置，你可以在 Kubernetes 调度层面解决 CUDA Runtime 和 CUDA Driver 版本不匹配的问题，确保应用程序被调度到兼容的 GPU 节点上。

步骤

1 在节点标签中添加 CUDA 版本

1. 在每个 GPU 节点上，运行以下命令以获取支持的 CUDA Runtime 版本：

   ```
   nvidia-smi | sed -n 's/.*CUDA Version: \([0-9.]\+\).*/\1/p'
   ```

   例如，输出可能为 12.4。

2. 在控制节点上，为该 GPU 节点添加对应的主版本和次版本标签：

   ```
   kubectl label node <node-name> \
   nvidia.com/cuda.runtime.major=12 \
   nvidia.com/cuda.runtime.minor=4
   ```

> **TIP**
>
> 如果你的集群中有很多 GPU 节点，手动为它们添加标签会比较困难。你可以安装 `Node Feature Discovery` 集群插件。 通过部署 Node Feature Discovery（NFD）集群插件并开启 GFD 扩展，GPU 节点会自动添加 CUDA 版本标签。 `Node Feature Discovery` 集群插件可以从 Customer Portal 获取。更多信息请联系 Consumer Support。

基于 CUDA 版本调度推理服务

从 Alauda AI 1.5 开始，产品将根据 CUDA 版本自动调度推理服务的 pod。对于更早的版本，你可以按照以下步骤操作：

1. 在创建推理服务时，确定需要选择哪个 ClusterServingRuntime。
2. 解析 ClusterServingRuntime 标签： 如果 `cpaas.io/accelerator-type` 为 nvidia，则进一步解析 `cpaas.io/cuda-version`（例如 11.8）。
3. 在推理服务中添加 nodeAffinity 字段，例如：

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
spec:
  predictor:
    affinity:
      nodeAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 100
          preference:
            matchExpressions:
            - key: nvidia.com/cuda.runtime.major
              operator: In
              values: ["11"]
            - key: nvidia.com/cuda.runtime.minor
              operator: Gt
              values: ["7"] # Since the k8s operator only supports
Gt, which means greater than but not equal to, we use the rt versi
on minus one to meet the requirements.
```

使用 Kueue 调度推理服务

本页介绍如何在 Alauda AI 中运行 inferenceService 时利用 Alauda Build of Kueue 的调度和资源管理能力。

目录

前提条件

操作步骤

前提条件

- 您已安装 Alauda AI。
- 您已安装 Alauda Build of Kueue。
- 您已安装 Alauda Build of Hami（用于演示 vGPU）。
- Alauda Container Platform Web CLI 已与您的集群建立通信。

操作步骤

1. 在 Alauda Container Platform 中创建一个 project 和 namespace，例如 project 名称为 `test`，namespace 名称为 `test-1`。

2. 切换到 Alauda AI，点击 **Admin** > **Management Namespace** 中的 **Namespace Manage**，并选择前面创建的 namespace，完成管理配置。
3. 运行以下命令创建资源：

```
cat <<EOF| kubectl create -f -
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: cluster-queue
spec:
  namespaceSelector: {}
  resourceGroups:
  - coveredResources: ["cpu", "memory", "pods", "ephemeral-storage", "nvidia.com/gpualloc", "nvidia.com/total-gpucores", "nvidia.com/total-gpumem"]
    flavors:
    - name: "default-flavor"
      resources:
      - name: "cpu"
        nominalQuota: 9
      - name: "memory"
        nominalQuota: 36Gi
      - name: "pods"
        nominalQuota: 5
      - name: "nvidia.com/gpualloc"
        nominalQuota: "2"
      - name: "nvidia.com/total-gpucores"
        nominalQuota: "50"
      - name: "nvidia.com/total-gpumem"
        nominalQuota: "20000"
      - name: "ephemeral-storage"
        nominalQuota: 100Gi
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: default-flavor
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  namespace: test-1
  name: test
spec:
  clusterQueue: cluster-queue
EOF
```

4. 在 Alauda AI UI 中创建一个带有标签 `kueue.x-k8s.io/queue-name: test` 的 `InferenceService` 资源：

```yaml
kind: InferenceService
apiVersion: serving.kserve.io/v1beta1
metadata:
  labels:
    kueue.x-k8s.io/queue-name: test
  name: test
  namespace: test-1
# ...
spec:
  predictor:
    model:
      resources:
        limits:
          cpu: '1'
          ephemeral-storage: 10Gi
          memory: 2Gi
          nvidia.com/gpualloc: '1'
          nvidia.com/gpucores: '80'
          nvidia.com/gpumem: 8k
# ...
```

5. 查看 `InferenceService` 的 pod：

```bash
kubectl -n test-1 get pod | grep test-predictor
```

您会看到该 pod 处于 `SchedulingGated` 状态：

```
test-predictor-8475554f4d-zw7lp   0/1     SchedulingGated   0
13s   <none>   <none>   <none>           <none>
```

6. 更新 `nvidia.com/total-gpucores` 配额：

```
cat <<EOF| kubectl apply -f -
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: cluster-queue
spec:
  namespaceSelector: {}
  resourceGroups:
  - coveredResources: ["cpu", "memory", "pods", "ephemeral-storage", "nvidia.com/gpualloc", "nvidia.com/total-gpucores", "nvidia.com/total-gpumem"]
    flavors:
    - name: "default-flavor"
      resources:
      - name: "cpu"
        nominalQuota: 9
      - name: "memory"
        nominalQuota: 36Gi
      - name: "pods"
        nominalQuota: 5
      - name: "nvidia.com/gpualloc"
        nominalQuota: "2"
      - name: "nvidia.com/total-gpucores"
        nominalQuota: "100"
      - name: "nvidia.com/total-gpumem"
        nominalQuota: "20000"
      - name: "ephemeral-storage"
        nominalQuota: 100Gi
EOF
```

您会看到该 pod 处于 `Running` 状态：

```
test-predictor-8475554f4d-zw7lp   1/1     Running   0          13s   <none>   <none>   <none>           <none>
```

为 vLLM Inference Services 启用 Expert Parallel

目录

简介
何时使用 Expert Parallel
前提条件和限制
EP 配置概览
启用 EP 时的层行为
上游命令与平台映射
配置单节点 InferenceService
这些标志为何重要
查看已配置的 Spec
关于多节点部署
参考资料

简介

本文档展示了一个单节点、以 YAML 为先的起点，用于在 `InferenceService` 中启用 vLLM **Expert Parallel (EP)**。

Expert Parallel 是上游 vLLM 面向 **Mixture-of-Experts (MoE)** 模型的能力。它在 vLLM 中仍然属于 **experimental**，相关参数名称或默认值在未来版本中可能会发生变化。

> **WARNING**
>
> 本页重点介绍一个用于快速入门 Expert Parallel 的单节点配置示例。有关性能调优、容量规划以及分布式部署细节，请参阅官方 vLLM 文档。

何时使用 **Expert Parallel**

当你正在提供一个 MoE 模型服务，并且希望 vLLM 将 expert 层切分到多个 GPU 上，而不是依赖默认的 expert-layer grouping 行为时，Expert Parallel 就很有用。

对于单节点部署，上游 vLLM 的模式如下：

- 使用 `--enable-expert-parallel` 启用 EP。
- 示例保持在单节点上。
- 使用 `--data-parallel-size` 跨越该节点上的 GPU。
- 在本示例中使用 `--tensor-parallel-size 1`，这样 attention 层会在 data parallel ranks 之间复制，而不是通过 tensor parallelism 进行切分。

如果你提供的是 dense 模型，或者当前运行时镜像不包含上游 vLLM 所需的 EP 相关依赖，那么本指南不是合适的起点。

前提条件和限制

- 你可以访问已安装 KServe 的 Kubernetes 集群。
- 你有一个可以创建 `InferenceService` 资源的命名空间。
- 你已经在平台上拥有可用的 vLLM 服务运行时。
- 你使用的运行时镜像已经包含 vLLM EP 所需的上游依赖。
- 你的模型是 MoE 模型，并且已经可以通过配置的 `storageUri` 供服务访问。

- 目标 node 上有多个可见 GPU。本示例使用检测到的 GPU 数量作为单节点的 data parallel size。

如果你当前的 vLLM 镜像尚未包含所需的 EP 依赖，请先扩展或重新构建运行时镜像。有关平台相关的运行时定制，请参阅 Extend Inference Runtimes。有关上游依赖列表和后端指导，请参阅 References 中官方 vLLM EP 部署指南。

EP 配置概览

通过添加 `--enable-expert-parallel` 标志来启用 EP。在上游 vLLM 中，expert parallel size 会自动计算：

```
EP_SIZE = TP_SIZE x DP_SIZE
```

其中：

- `TP_SIZE`：tensor parallel size
- `DP_SIZE`：data parallel size
- `EP_SIZE`：expert parallel size，由 vLLM 自动计算

这意味着你不需要单独设置 EP size 参数。相反，你需要选择 tensor parallel 和 data parallel 的大小，vLLM 会根据这些设置推导出实际的 expert parallel 组大小。

启用 EP 时的层行为

当为 MoE 模型启用 EP 时，不同类型的层会使用不同的并行策略：

层类型	行为	使用的并行方式
Expert (MoE) 层	在所有 EP ranks 上切分	Expert Parallel，大小为 `TP x DP`
Attention 层	行为取决于 `TP_SIZE`	Tensor Parallel 或 Data Parallel，取决于 `TP_SIZE`

对于 attention 层：

- 当 `TP_SIZE = 1` 时，attention 权重会复制到所有 data parallel ranks。
- 当 `TP_SIZE > 1` 时，attention 权重会在每个 data parallel 组内通过 tensor parallelism 进行切分。

例如，如果 `TP_SIZE = 2` 且 `DP_SIZE = 4`，该服务总共使用 8 个 GPU：

- expert 层形成一个大小为 8 的 EP 组，expert 会分布到所有 GPU 上。
- attention 层在 4 个 data parallel 组中的每一个组内都使用大小为 2 的 tensor parallelism。

与常规的 data parallel 部署相比，主要区别在于 MoE 层的分布方式。未启用 `--enable-expert-parallel` 时，MoE 层遵循 tensor parallel grouping 行为。启用 EP 后，expert 层会切换到 expert parallelism，这种方式是专门为 MoE 风格的 expert 切分设计的。

上游命令与平台映射

上游单节点示例使用的命令类似如下：

```
vllm serve deepseek-ai/DeepSeek-V3-0324 \
  --tensor-parallel-size 1 \
  --data-parallel-size 8 \
  --enable-expert-parallel
```

在 Alauda AI 上，这些相同的标志通常通过 `InferenceService` 容器命令传递。换句话说：

- `vllm serve ...` 会变成在 `spec.predictor.model.command` 中启动的命令
- `--tensor-parallel-size`、`--data-parallel-size` 和 `--enable-expert-parallel` 会附加到 vLLM 启动命令中
- 模型位置、运行时名称和 Kubernetes 资源通过 `storageUri`、`runtime` 和 `resources` 表达

因此，下面的示例重点展示如何将 EP 相关标志放入平台的 `InferenceService` YAML 中。

配置单节点 **InferenceService**

创建一个 YAML 文件，例如 `deepseek-v3-ep.yaml`，内容如下：

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  annotations:
    aml-model-repo: DeepSeek-V3-0324
    serving.kserve.io/deploymentMode: Standard
  labels:
    aml.cpaas.io/runtime-type: vllm
  name: deepseek-v3-ep
  namespace: <your-namespace>
spec:
  predictor:
    minReplicas: 1
    maxReplicas: 1
    model:
      command:
        - bash
        - -c
        - |
          set -ex

          GPU_COUNT=$(python3 -c "import torch; print(torch.cuda.device_c
ount())")
          if [ "${GPU_COUNT}" -lt 2 ]; then
            echo "Expert Parallel requires multiple visible GPUs on the s
ame node"
            exit 1
          fi

          MODEL_PATH="/mnt/models/${MODEL_NAME}"
          if [ ! -d "${MODEL_PATH}" ]; then
            MODEL_PATH="/mnt/models"
          fi

          python3 -m vllm.entrypoints.openai.api_server \
            --port 8080 \
            --served-model-name {{.Name}} {{.Namespace}}/{{.Name}} \
            --model "${MODEL_PATH}" \
            --dtype ${DTYPE} \
            --gpu-memory-utilization ${GPU_MEMORY_UTILIZATION} \
            --tensor-parallel-size 1 \
            --data-parallel-size "${GPU_COUNT}" \
            --enable-expert-parallel
```

```
        - bash
      env:
        - name: DTYPE
          value: half
        - name: GPU_MEMORY_UTILIZATION
          value: '0.95'
        - name: MODEL_NAME
          value: '{{ index .Annotations "aml-model-repo" }}'
      modelFormat:
        name: transformers
      protocolVersion: v2
      resources:
        limits:
          cpu: '8'
          memory: 32Gi
          nvidia.com/gpu: '8'
        requests:
          cpu: '4'
          memory: 16Gi
      runtime: <your-vllm-runtime>
      storageUri: hf://<your-model-path>
    securityContext:
      seccompProfile:
        type: RuntimeDefault
```

应用该清单：

```
kubectl apply -f deepseek-v3-ep.yaml -n <your-namespace>
```

这些标志为何重要

标志	作用
`--enable-expert-parallel`	为 MoE 模型启用 Expert Parallel。
`--data-parallel-size "${GPU_COUNT}"`	在此单节点示例中，将节点上所有可见 GPU 作为 data parallel 组使用。
`--tensor-parallel-size 1`	与上游示例一致，并使 attention 层不进行 tensor parallel 切分。

TIP

请调整 GPU 资源字段，以匹配集群中可用的资源键以及目标 node 上的 GPU 数量。此示例的重点在于 vLLM EP 参数在 `InferenceService` command 中的添加位置。如果你需要显式设置 all-to-all backend，请先遵循上游 backend 选择指南，再添加 `--all2all-backend`。

查看已配置的 Spec

应用清单后，检查生成的 `InferenceService` spec，并确认 EP 相关参数已存在：

```
kubectl get inferenceservice deepseek-v3-ep -n <your-namespace> -o yaml
```

重点查看生成的 predictor command，并确认其中仍然包含：

- `--enable-expert-parallel`
- `--data-parallel-size`
- `--tensor-parallel-size 1`

此检查可确认预期的 vLLM 参数已应用到服务配置中。它不验证运行时性能、backend 兼容性或多节点行为。

关于多节点部署

多节点 EP 部署需要额外的分布式运行时和网络配置，包括每个 node 的启动设置、node 角色以及 data-parallel 通信设置。

WARNING

本页重点介绍单节点配置模式。如果你需要多节点 EP，请参考官方 vLLM 指南，并根据你的集群拓扑和运行时环境调整部署模型。

参考资料

- Expert Parallel Deployment - vLLM ↗
- Data Parallel Deployment - vLLM ↗
- Extend Inference Runtimes
- 使用 CLI 创建 Inference Service

vLLM 推理服务的推测解码

目录

简介

在决定之前

本指南在 Alauda AI 上验证过的方法

推荐起始点

内部验证快照 — N-gram

内部验证快照 — EAGLE-3

前提条件

配置面

在 Alauda AI 上提供 Model Artifacts

单 artifact 模式（N-gram）

双 artifact 模式（EAGLE-3 及类似方法）

选项 A — KServe `storageUris`（可用时推荐）

选项 B — 单个 OCI Modelcar，其中包含两个 artifact

选项 C — 预先放置在共享 PVC 上

在 A / B / C 之间选择

端到端示例

示例 1 — N-gram

示例 2 — 在共享 PVC 上同时放置 target + draft 的 EAGLE-3

验证并衡量影响

1. 确认配置已应用

2. 确认推测解码正在实际运行
3. 衡量端到端影响
4. 如何报告或比较数据

回滚

故障排查

注意事项和已知限制

参考资料

简介

推测解码允许 vLLM server 在每个 decode 步骤中提出多个 token，并通过目标 model 的单次 forward pass 对其进行验证，从而在不改变输出分布的情况下，降低交互式工作负载的单 token 延迟。

本页重点介绍如何在 Alauda AI 上为运行中的 `InferenceService` 启用、配置、验证以及回滚推测解码。关于该技术本身以及 vLLM 支持的完整方法列表，请参阅 vLLM speculative decoding 文档↗。

> **WARNING**
>
> 推测解码涉及对 runtime 版本敏感的 flags。下面提到的 `--speculative-config` JSON keys、支持的 `method` 值以及 metric 名称，都取决于你 runtime image 中的 vLLM 版本。本文中的所有示例都应视为起点，并请以你实际发布的 vLLM 版本为准进行确认。

在决定之前

当单个请求的 **decode loop** 主导端到端延迟，并且提议的 token 能够以足够高的接受率被采纳，从而摊薄提议开销时，推测解码才会带来收益。

它通常适用于：

- 交互式 chat / agent 循环，且后续输出相对可预测。

- 摘要、RAG answers 和 code completion，此类场景中输出与 prompt 有较高重叠。

在以下场景中，它可能带来负收益或基本无影响：

- 高温度采样，此时接受率会大幅下降。
- 高 QPS / batch 饱和的服务，此时 decode 容量已不再空闲。vLLM 团队在 2024 年的 V0-engine benchmarks 中报告，在高 QPS 下，同一数据集上出现了 **1.4×–1.8×** 的变慢。V1 engine 的调度方式不同，因此在你的 runtime 上幅度可能不同，但风险方向是相同的。
- 非常小的 target model，此时验证步骤本身已经很便宜。

在将推测解码设为默认值之前，请先在具有代表性的 workload 上进行验证。请参见 验证并衡量影响。

本指南在 **Alauda AI** 上验证过的方法

下面两个方法是本指南覆盖的内容，并且已在 Alauda AI 上完成端到端验证。vLLM upstream 还支持其他方法（例如适用于带有 multi-token-prediction heads 的 model 的 MTP、Medusa、MLP Speculator、Suffix、Draft Model），这些方法也可能通过相同的 `--speculative-config` flag 在 Alauda AI 上使用。它们不在本文范围内，因此请参考 upstream 文档，并在你自己的环境中验证后再推广到生产。

方法	你需要提供的内容	权衡
N-gram	仅 target model	无额外 weights，无需训练。收益取决于 prompt-output token 重叠程度。
EAGLE-3	target model 以及 匹配的 EAGLE-3 draft head	需要一个针对精确 target model 训练的 draft head。额外 GPU memory 占用较小。

说明：

- vLLM upstream 将 N-gram 描述为“适用于摘要和问答等用例，因为 prompt 与 answer 之间存在显著重叠”。
- vLLM upstream 将 EAGLE-3 描述为“当前推测解码算法的 SOTA”（摘自最新 features 页；请按每个 release 重新确认）。

推荐起始点

不存在一种适用于所有 workload 的最佳方法。以下是保守的起始点，用于降低试错成本。在推广到生产之前，请始终结合你自己的流量进行验证。

如果你有 ...	建议从这里开始
一个通用 chat / instruction model，并且有可用的 EAGLE-3 head	EAGLE-3，初始 `num_speculative_tokens: 3`。
prompt-output 重叠很高（RAG、摘要、code completion）且没有 EAGLE-3 head	N-gram，初始 `num_speculative_tokens: 5`。
以上都不满足	暂缓启用推测解码，直到满足上述条件之一。

内部验证快照 — N-gram

上面的起始点是建议，不是保证。下面的测量结果来自 Alauda AI 内部实验室的一个具体数据点，目的是帮助你对类似的单 GPU serving 场景形成预期。你自己的 model、GPU、runtime 版本和流量会产生不同结果——在推广到生产之前务必进行 benchmark。

- 硬件： NVIDIA A30 24 GB × 1
- **Model**： Qwen3-8B（BF16，HuggingFace `Qwen/Qwen3-8B`）
- **Runtime**： vLLM 0.19.1（V1 engine）
- 请求参数： `temperature=0`、`seed=42`、`max_tokens=1024`、`enable_thinking=false`、单个并发请求、1 次 warmup 丢弃 + 3 次计时运行（报告中位数）

基线命令（不使用 **spec decode**）：

```
python3 -m vllm.entrypoints.openai.api_server \
  --port 8080 \
  --served-model-name t-ng \
  --model /mnt/models \
  --gpu-memory-utilization 0.8 \
  --max-model-len 4096 \
  --max-num-seqs 8 \
  --seed 42
```

N-gram 命令（仅 `--speculative-config` 不同）：

```
python3 -m vllm.entrypoints.openai.api_server \
  --port 8080 \
  --served-model-name t-ng \
  --model /mnt/models \
  --gpu-memory-utilization 0.8 \
  --max-model-len 4096 \
  --max-num-seqs 8 \
  --seed 42 \
  --speculative-config '{"method":"ngram","num_speculative_tokens":5,"pro
mpt_lookup_max":4,"prompt_lookup_min":2}'
```

Workloads：

- *code refactor（高 prompt-output 重叠）：* 要求 model 为一个 30 行的 Python class 添加 docstring 和类型注解，并返回完整更新后的 class
- *general chat（无 prompt-output 重叠）：* 要求 model 用不少于 800 个词解释一个概念

结果：

Workload	基线 **tok/s**	**N-gram tok/s**	加速比	墙钟时间差
Code refactor（高重叠）	47.02	45.92	**0.98×**	+524 ms
General chat（无重叠）	47.13	39.94	**0.85×**	+3914 ms

解读：

- 在这个单 GPU 8B 场景中，N-gram 在 code refactor workload 上表现为轻微回退，而在 chat 上则出现了明显的约 15% 回退。提议器的 CPU 工作、每步对 5 个候选 token 的验证，以及 vLLM 在 N-gram 下禁用 async scheduling，这些开销加在一起超过了被接受 token 带来的收益。
- 对高重叠 code workload 而言，接受率是健康的（在更早的非正式探测中，平均接受长度约为 3），但仅有接受率并不能预测端到端加速——每步开销必须与 target model 的实际 decode 时间摊销后才有意义。在单 GPU 的小型 target model 上，decode 本身已经很便宜，可供摊销的空间很小。
- chat 结果验证了 注意事项 中关于缺少 prompt-output 重叠的 workload 的判断。

同样的方法在更大的 target model 上（此时每次 verify step 的成本更高）、在 multi-GPU tensor parallelism 下，或在更高并发下，可能表现得非常不同。请把这个快照当作“需要测量”的提醒，而不是对 N-gram 本身的结论。

内部验证快照 — EAGLE-3

上面的起始点是建议，不是保证。下面的测量结果来自 Alauda AI 内部实验室的一个具体数据点，目的是帮助你对类似的单 GPU EAGLE-3 场景形成预期。你自己的 model、GPU、runtime 版本和流量会产生不同结果——在推广到生产之前务必进行 benchmark。

- 硬件： NVIDIA A30 24 GB × 1
- **Model**： Meta-Llama-3.1-8B-Instruct（BF16，HuggingFace `meta-llama/Meta-Llama-3.1-8B-Instruct`），EAGLE-3 draft 为 `yuhuili/EAGLE3-LLaMA3.1-Instruct-8B`
- **Runtime**： vLLM 0.19.1（V1 engine）
- 请求参数： `temperature=0`、`seed=42`、`max_tokens=1024`、单个并发请求、1 次 warmup 丢弃 + 3 次计时运行（报告中位数）

基线命令（不使用 **spec decode**）：

```
python3 -m vllm.entrypoints.openai.api_server \
  --port 8080 \
  --served-model-name eagle \
  --model /mnt/models/Meta-Llama-3.1-8B-Instruct \
  --dtype auto \
  --gpu-memory-utilization 0.8 \
  --max-model-len 4096 \
  --max-num-seqs 8 \
  --seed 42
```

EAGLE-3 命令（仅 `--speculative-config` 不同）：

```
python3 -m vllm.entrypoints.openai.api_server \
  --port 8080 \
  --served-model-name eagle \
  --model /mnt/models/Meta-Llama-3.1-8B-Instruct \
  --dtype auto \
  --gpu-memory-utilization 0.8 \
  --max-model-len 4096 \
  --max-num-seqs 8 \
  --seed 42 \
  --speculative-config '{"method":"eagle3","model":"/mnt/models/EAGLE3-LLaMA3.1-Instruct-8B","num_speculative_tokens":3}'
```

Workloads：

- *code refactor（高 prompt-output 重叠）：* 要求 model 为一个 30 行的 Python class 添加 docstring 和类型注解，并返回完整更新后的 class
- *general chat（无 prompt-output 重叠）：* 要求 model 用不少于 800 个词解释一个概念

结果：

Workload	基线 **tok/s**	**EAGLE-3 tok/s**	加速比	墙钟时间差（中位数）
Code refactor（高重叠）	47.84	88.25	**1.84×**	−6171 ms

Workload	基线 **tok/s**	**EAGLE-3 tok/s**	加速比	墙钟时间差（中位数）
General chat（无重叠）	47.87	47.45	**0.99×**	+2416 ms

Speedup 是 tok/s 的比值（与 completion length 无关）。Wall delta 直接比较中位数墙钟时间；chat 运行产生的输出数量不同（基线 588 token，而 EAGLE-3 为 709 token），因此在该场景下 Speedup 是更可靠的指标。

推测解码行为（**EAGLE-3** 侧，来自 `SpecDecoding metrics` 日志窗口）：

Workload	平均接受长度	平均 **Draft** 接受率	按位置接受率
Code refactor（高重叠）	≈ 2.54	≈ 51%	0.50 / 0.40 / 0.33
General chat（无重叠）	≈ 1.19	≈ 6%	0.16 / 0.02 / 0.01

平均接受长度和接受率是按各 benchmark 运行所覆盖的 `SpecDecoding metrics` 日志窗口中的 draft 加权结果；按位置的数值来自每次运行中的持续负载窗口。

解读：

- 在这个单 GPU 8B 场景中，EAGLE-3 在 code refactor 上带来了 约 **1.84×** 的加速，而在 general chat 上则基本持平（约 **0.99×**）。两个基线运行的 tok/s 都稳定在约 47.8，这与预期一致——基础 decode 速率是 model 与硬件的属性，不依赖 prompt 内容。所有可观察到的差异都来自 EAGLE-3 一侧。
- 为什么 **code** 有收益而 **chat** 没有——接受率数据直接揭示了机制。在 code 场景中，draft head 在每个 decode step 平均能命中约 2.54 个 token，接受率约 51%，因此大多数 step 会输出多个 token；按位置接受率下降得很慢（0.50 / 0.40 / 0.33），因此即使第 3 个 speculative slot 仍然有约三分之一的时间产生收益。在 chat 场景中，平均接受长度只有约 1.19，接受率约 6%，并且按位置接受率在第 2 个 slot 就迅速崩塌（0.16 / 0.02 / 0.01）——几乎每一步都只输出经过验证的 token，而 draft 出来的 token 都被丢弃。
- 实际效果 **vs** 理论上限。 在没有 proposer 开销的理想情况下，平均接受长度是理论上的最大加速上限。code 场景实际达到 1.84×，而理论上限为 2.54×（约 72% 转化），也就是说 proposer 的 CPU 工作、对被拒绝提议的验证，以及 async scheduling 的成本，吞掉了大约四分之一的可用空间。chat 场景的 1.19× 理论上限则完全被开销吃掉，甚至略微转为回退。

这与 Caveats 中的说明一致：在单 GPU 的小模型上，每步开销几乎没有空闲 decode 容量可供隐藏。

同样的方法在更大的 target model 上（此时每次 verify step 的成本更高）、在 multi-GPU tensor parallelism 下，或在更高并发下，可能表现得非常不同。请把这个快照当作“需要测量”的提醒，而不是对 EAGLE-3 本身的结论。

前提条件

- 一个已安装 KServe 的 Kubernetes 集群，以及可以创建 `InferenceService` 资源的 namespace。
- 平台上已注册一个 vLLM serving runtime，并且其 vLLM 版本支持你计划使用的 speculative method。要检查版本，请进入该 runtime 的运行中 pod：`kubectl exec <pod> -- python3 -c "import vllm; print(vllm.__version__)"`。
- 目标 model 可通过其 storage source（model repository、PVC 或 OCI image）供服务访问。
- 对于 EAGLE-3：一个 draft head，其 architecture、tokenizer 和 base version 与精确的 target model 匹配。若 head 不匹配，通常会悄悄降低接受率，而且不一定会以启动错误的形式暴露。
- 对于 EAGLE-3：一种可以将 target 和 draft 同时送入同一 pod 的 model-artifact 加载机制。请参阅 在 Alauda AI 上提供 Model Artifacts。

配置面

在 vLLM v1 中，推测解码通过一个参数启用：

```
--speculative-config '{"method": "<method>", "num_speculative_tokens": <k>, ...}'
```

常见 keys：

- `method`：要使用的 proposer。本指南使用的值为 `ngram` 和 `eagle3`。upstream 还存在其他值（例如 `medusa`，或模型特定的 MTP 名称如 `deepseek_mtp`）——请在 vLLM

speculative decoding 文档中确认你方法对应的确切值。

- `num_speculative_tokens`：每步提议多少个 token。更高的值可能提升 speedup，但也会在被拒绝的提议上浪费计算。
- `model`：对于加载单独 draft artifact 的方法（例如 EAGLE-3），这是容器内该 artifact 的路径。
- 方法特定的 keys，例如 N-gram 的 `prompt_lookup_max` / `prompt_lookup_min`。这些名称在不同 vLLM release 中发生过变化——请以你发布的版本为准进行验证。

其他所有 vLLM 参数（`--model`、`--tensor-parallel-size`、`--gpu-memory-utilization`，等等）都与非推测解码部署中的用法相同。

在 Alauda AI 上提供 Model Artifacts

不同的方法需要 predictor pod 中的不同文件。

单 artifact 模式（N-gram）

N-gram 只需要 target model。像其他 inference service 一样，直接使用 `storageUri` 即可：

```
spec:
  predictor:
    model:
      storageUri: hf://<your-model-path>
```

model 会落到 `/mnt/models`，并通过 `--model` 传递给 vLLM。

双 artifact 模式（EAGLE-3 及类似方法）

EAGLE-3 需要将 target model 以及 匹配的 draft head 加载到同一个 pod 中。共有三种受支持的交付方式。请根据平台版本、网络访问条件和运维偏好进行选择。

选项 A — KServe `storageUris`（可用时推荐）

`storageUris` 是一个 KServe 字段，支持多个 storage location，并将每个 location 挂载到声明的路径。当你的平台 KServe 版本支持它时，这是最简洁的选择（KServe 0.16 及更高版本）。

```
spec:
  predictor:
    model:
      storageUris:
        - uri: hf://<your-target-model-path>
          mountPath: /mnt/models/target
        - uri: hf://<your-draft-head-path>
          mountPath: /mnt/models/draft
```

然后将 vLLM 指向这两个路径：

```
--model /mnt/models/target \
--speculative-config '{"method":"eagle3","model":"/mnt/models/draft","num_speculative_tokens":3}'
```

需要注意的约束：

- `storageUri`（单数）和 `storageUris`（复数）互斥。
- 所有 `mountPath` 值都必须是绝对路径，并共享一个公共父目录（例如 `/mnt/models/target` 和 `/mnt/models/draft`）。
- 对于私有仓库，请将相应的 credentials secret 绑定到 predictor pod 使用的 service account 上。

如果你平台上的 KServe 版本尚不支持 `storageUris`，请使用选项 B 或选项 C。

选项 B — 单个 OCI Modelcar，其中包含两个 artifact

将 target model 和 draft head 打包进同一个 OCI image 中，并放在可预测的子目录下（例如 `/models/target` 和 `/models/draft`），然后使用 `storageUri: oci://...` 进行部署。打包步骤请参阅 使用 KServe Modelcar 存储模型。可写入镜像的示例磁盘布局如下：

```
/models/
├── target/
│   └── ... target model files ...
└── draft/
    └── ... EAGLE-3 head files ...
```

随后 vLLM 命令引用同样的路径：

```
--model /mnt/models/target \
--speculative-config '{"method":"eagle3","model":"/mnt/models/draft","num_speculative_tokens":3}'
```

对于离线 / air-gapped 集群，此选项非常适合，因为这些 artifact 会一起版本化，并从你自己的 registry 拉取。

选项 **C** — 预先放置在共享 **PVC** 上

将两个 artifact 预先放置到某个 PVC 的已知目录结构中，挂载该 PVC，并在 vLLM 命令中引用本地路径。如果你已经在共享文件系统上管理 model 文件，这是最简单的方案。

在 **A / B / C** 之间选择

约束	选择
在线集群、KServe ≥ 0.16，希望使用声明式 manifests	选项 A
离线 / air-gapped，希望使用单个版本化 artifact	选项 B
已经在共享 PVC 上管理 model 文件	选项 C

端到端示例

下面两个示例覆盖了 Alauda AI 上可用的方法 中列出的方法。请将 `<your-namespace>`、`<your-vllm-runtime>` 和 storage URIs 替换为你环境中的实际值。

示例 **1 — N-gram**

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  annotations:
    aml-model-repo: Qwen2.5-7B-Instruct (1)
    serving.kserve.io/deploymentMode: Standard
  labels:
    aml.cpaas.io/runtime-type: vllm
  name: qwen-ngram-spec
  namespace: <your-namespace>
spec:
  predictor:
    minReplicas: 1
    maxReplicas: 1
    model:
      command:
        - bash
        - -c
        - |
          set -ex

          MODEL_PATH="/mnt/models/${MODEL_NAME}"
          if [ ! -d "${MODEL_PATH}" ]; then
            MODEL_PATH="/mnt/models"
          fi

          python3 -m vllm.entrypoints.openai.api_server \
            --port 8080 \
            --served-model-name {{.Name}} {{.Namespace}}/{{.Name}} \
            --model "${MODEL_PATH}" \
            --dtype ${DTYPE} \
            --gpu-memory-utilization ${GPU_MEMORY_UTILIZATION} \
            --speculative-config '{"method":"ngram","num_speculative_toke
ns":5,"prompt_lookup_max":4,"prompt_lookup_min":2}' (2)
        - bash
      env:
        - name: DTYPE
          value: half
        - name: GPU_MEMORY_UTILIZATION
          value: '0.85'
        - name: MODEL_NAME
          value: '{{ index .Annotations "aml-model-repo" }}'
      modelFormat:
```

```
      name: transformers
    protocolVersion: v2
    resources:
      limits:
        cpu: '8'
        memory: 32Gi
        nvidia.com/gpu: '1'
      requests:
        cpu: '4'
        memory: 16Gi
    runtime: <your-vllm-runtime>
    storageUri: hf://<your-model-path>
  securityContext:
    seccompProfile:
      type: RuntimeDefault
```

1 请替换为你的实际 model 名称；该注解由平台用于展示。

2 `prompt_lookup_*` keys 属于 n-gram proposer。这些名称在不同 vLLM release 之间发生过变化——请以 runtime image 内的版本为准进行验证。

示例 2 — 在共享 PVC 上同时放置 target + draft 的 EAGLE-3

该 manifest 与上文 内部验证快照 — EAGLE-3 中使用的 setup 一致。target model 和 EAGLE-3 draft head 都已预先放置在同一个 PVC 中的可预测子目录下；PVC 通过 `storageUri: pvc://...` 挂载到 `/mnt/models/`，而 vLLM 命令直接引用这两个子目录。

```yaml
apiVersion: serving.kserve.io/v1beta1
kind: InferenceService
metadata:
  annotations:
    aml-model-repo: Meta-Llama-3.1-8B-Instruct
    serving.kserve.io/deploymentMode: Standard
  labels:
    aml.cpaas.io/runtime-type: vllm
  name: llama-eagle3-spec
  namespace: <your-namespace>
spec:
  predictor:
    minReplicas: 1
    maxReplicas: 1
    model:
      command:
        - bash
        - -c
        - |
          set -ex

          python3 -m vllm.entrypoints.openai.api_server \
            --port 8080 \
            --served-model-name {{.Name}} {{.Namespace}}/{{.Name}} \
            --model /mnt/models/Meta-Llama-3.1-8B-Instruct \
            --dtype ${DTYPE} \
            --gpu-memory-utilization ${GPU_MEMORY_UTILIZATION} \
            --max-model-len 4096 \
            --max-num-seqs 8 \
            --seed 42 \
            --speculative-config '{"method":"eagle3","model":"/mnt/model
s/EAGLE3-LLaMA3.1-Instruct-8B","num_speculative_tokens":3}' (1)
        - bash
      env:
        - name: DTYPE
          value: auto
        - name: GPU_MEMORY_UTILIZATION
          value: '0.8' (2)
      modelFormat:
        name: transformers
      protocolVersion: v2
      resources:
        limits:
```

```
          cpu: '8'
          memory: 48Gi
          nvidia.com/gpu: '1'
        requests:
          cpu: '4'
          memory: 24Gi
      runtime: <your-vllm-runtime>
      storageUri: pvc://<your-pvc-name>/ 3
    securityContext:
      seccompProfile:
        type: RuntimeDefault
```

1 vLLM 命令中的两个路径（`--model` 和 `--speculative-config` 内的 `model` key）必须与 PVC 内部 的目录名完全一致。如果你的 PVC 使用了不同的目录名，请同步调整这两个路径。

2 EAGLE-3 head 会占用 `--gpu-memory-utilization` 预算之外的 GPU memory。保留余量（这里用 `0.8` 而不是 `0.9`）可以在两个 artifact 同时加载时降低 OOM 风险。

3 `pvc://<your-pvc-name>/` 期望一个已预先放置目标 **model** 和 **EAGLE-3 draft head** 的 PVC；PVC 根目录会挂载到 `/mnt/models/`，因此两个 artifact 必须分别位于 `/mnt/models/<target-subdir>/` 和 `/mnt/models/<draft-subdir>/`。请参见下方期望布局。如果你更希望使用声明式多 URI 挂载（KServe 0.16+），或者改为将 target + draft 打包进单个 OCI image，请参阅 在提供 Model Artifacts 中的选项 A 或选项 B。

PVC 内部的期望布局（在 pod 中挂载到 `/mnt/models/`）：

```
<PVC root>/
├── Meta-Llama-3.1-8B-Instruct/
│   └── ... target model files ...
└── EAGLE3-LLaMA3.1-Instruct-8B/
    └── ... EAGLE-3 draft head files ...
```

pod 启动后，从 predictor pod 内部验证布局：

```
kubectl exec -n <your-namespace> <pod> -- ls /mnt/models/
# Expected: EAGLE3-LLaMA3.1-Instruct-8B/  Meta-Llama-3.1-8B-Instruct/
```

使用以下命令应用上述任意 manifest：

```
kubectl apply -f <manifest>.yaml -n <your-namespace>
```

验证并衡量影响

确认已配置推测解码是一回事；确认它确实改善了你的 **workload** 则是另一回事。

1. 确认配置已应用

```
kubectl get inferenceservice <name> -n <your-namespace> -o yaml
```

在 predictor command 中查找 `--speculative-config`，并确认 readiness 状态：

```
kubectl get pods -n <your-namespace> -l serving.kserve.io/inferenceservic
e=<name>
```

2. 确认推测解码正在实际运行

启动时的第一个信号是 engine-config 日志行；它会打印 engine 解析出的 `speculative_config`，因此你可以据此验证 method 和 draft path 是否生效：

```
kubectl logs -n <your-namespace> -l serving.kserve.io/inferenceservice=<n
ame> \
  | grep -m1 'Initializing a V1 LLM engine'
# Expected to contain: speculative_config=SpeculativeConfig(method='eagle
3', model='...', num_spec_tokens=3)
```

对于实时 counter，vLLM 会在 `/metrics` 暴露 Prometheus metrics。确切的 metric 名称取决于 vLLM 版本，因此应先进行广泛匹配：

```
kubectl exec <pod> -n <your-namespace> -- curl -s localhost:8080/metrics
| grep -iE 'spec_decode|draft|acceptance'
```

如果没有返回任何内容，说明 pod 还没有处理过任何请求（counter 只会在第一次 generation 完成后才发布），或者你所用 vLLM build 中的 metric 名称不同——此时应回退查看 predictor 日志。

vLLM 会打印一条按窗口汇总的摘要行，这是最直观的实时视图。以下是 vLLM 0.19.1 在 `num_speculative_tokens=3` 时的真实输出格式：

```
SpecDecoding metrics: Mean acceptance length: 2.68, Accepted throughput:
65.69 tokens/s,
Drafted throughput: 116.98 tokens/s, Accepted: 657 tokens, Drafted: 1170
tokens,
Per-position acceptance rate: 0.664, 0.559, 0.462, Avg Draft acceptance r
ate: 56.2%
```

如何阅读它：

- **Mean acceptance length** — 每个 decode step 平均交付的 token 数。基线为 `1`。这是你在该 workload 上可期望获得的实际加速上限。
- **Avg Draft acceptance rate** — 被接受的提议 token 占所有提议 token 的整体比例。它回答的是“proposer 大多在发挥作用，还是大多在浪费计算？”这一问题。
- **Per-position acceptance rate** — 第 `1..num_speculative_tokens` 个 slot 的逐位接受率。你会看到恰好 **`num_speculative_tokens`** 个值——上面的示例之所以有 3 个值，是因为运行时使用了 `num_speculative_tokens=3`；而一个 `num_speculative_tokens=5` 的 `ngram` 运行会打印 5 个值。健康的曲线下降应较缓；如果在第 2 个 slot 附近就降到接近 0，说明该 workload 不适合这个 proposer。

3. 衡量端到端影响

对同一个具有代表性的 workload 运行两次：

1. 移除 `--speculative-config`（基线）。
2. 启用它（其他配置完全相同，包括 `--seed`）。

每次运行记录三个数值：

- 首 token 时间（TTFT）。
- 单 token 延迟（或固定输出长度下的端到端延迟）。

- 在你实际服务的 QPS 下的吞吐量（tokens/second）。

如果在目标 QPS 下这三项都得到改善，那么就值得保留推测解码。常见失败模式是：低 QPS 时有提升，但在生产 QPS 下反而回退——因此务必在你真实运行的负载下进行测量。

4. 如何报告或比较数据

如果没有上下文，性能数据既无法复现，也无法被信任。每当你发布一组对比——无论是内部、客户报告，还是反馈给平台团队——都应包含下面五项字段。缺少任意一项的数据都应被视为轶事，而非证据。

```
**Hardware:** <GPU model and count, e.g. NVIDIA A30 24 GB × 1>
**Model:** <model identifier and dtype, e.g. Qwen3-8B (BF16)>
**Runtime:** <vLLM version and runtime image name, e.g. vLLM 0.19.1 insid
e aml-vllm-x.y.z>
**Request parameters:** <temperature, max_tokens, concurrency, sampling t
oggles, runs per prompt>

**Baseline command (no spec decode):**
```text
python3 -m vllm.entrypoints.openai.api_server \
  --port 8080 \
  --served-model-name <name> \
  --model /mnt/models \
  --gpu-memory-utilization 0.8 \
  --max-model-len 4096 \
  --max-num-seqs 8 \
  --seed 42
```

Spec-decode 命令（仅 `--speculative-config` 不同）：
```

```
python3 -m vllm.entrypoints.openai.api_server \
  --port 8080 \
  --served-model-name <name> \
  --model /mnt/models \
  --gpu-memory-utilization 0.8 \
  --max-model-len 4096 \
  --max-num-seqs 8 \
  --seed 42 \
  --speculative-config '{"method":"ngram","num_speculative_tokens":5,"pro
mpt_lookup_max":4,"prompt_lookup_min":2}'
```

结果：

Workload	基线 **TTFT**	**Spec TTFT**	基线 **tok/s**	**Spec tok/s**	平
chat	…	…	…	…	..
code	…	…	…	…	..
rag	…	…	…	…	..

进行对比时，有两个实用原则：

- 两侧都使用相同的 `--seed` 和 `temperature=0`，并在计时前对每个 service 先进行 3 次丢弃的 warmup 请求——否则采样噪声和 compile-cache 噪声会主导你测到的差异。
- 基线和 spec-decode 必须针对同一组固定 **prompt** 列表，且顺序相同，每个 prompt 至少运行 5–10 次，并比较中位数而不是平均值。

回滚

如果要在不改动其他内容的情况下禁用推测解码，只需从 predictor command 中移除 `--speculative-config` 这一行并重新应用：

```
kubectl edit inferenceservice <name> -n <your-namespace>
# delete the --speculative-config line, save, exit
```

或者重新应用一个不包含该 flag 的 manifest：

```
kubectl apply -f <manifest-without-spec-config>.yaml -n <your-namespace>
```

服务会以不含 speculative proposer 的新 revision 进行滚动更新。对于 N-gram，不需要更改任何 model artifact。对于 EAGLE-3，draft head 会继续挂载但不会被使用——如果你想回收磁盘空间，请在下一次变更时移除 draft-head artifact（删除选项 A 中对应的 `storageUris` entry、在不包含 draft 目录的情况下重新构建选项 B 的 OCI image，或者在选项 C 中删除 PVC 的 draft 子目录）。

故障排查

症状	可能原因	需要检查的内容
Pod 启动失败，并报出包含 `speculative` 或未知 JSON keys 的 vLLM argument error	`--speculative-config` 中的 keys 与 runtime image 内的 vLLM 版本不匹配	`kubectl exec <pod> -- python3 -c "import vllm; print(vllm.__version__)"`，并将 flags 调整为与该版本一致
Pod 启动失败，并提示未知的 `method` 值	`method` 拼写错误，或者该 vLLM 版本不支持该值（例如使用 `eagle` 代替 `eagle3`）	在 upstream speculative decoding 文档中确认你所用 vLLM release 支持的 `method` 值
启用 EAGLE-3 后在 model load 期间发生 OOM	未为 EAGLE-3 head 的 memory 预留预算	将 `--gpu-memory-utilization` 下调 0.05–0.10，或者减少 GPU 上的其他 workload
Service 已 Ready，但接受率接近 0	target 和 draft 之间的 tokenizer / architecture 不匹配，或者采样温度过高	重新确认 draft head 是否与精确的 target model 匹配；在评估时降低采样温度

症状	可能原因	需要检查的内容
在生产 QPS 下 TTFT 或延迟出现回退	提议开销不再被空闲的 decode capacity 所隐藏	在该 service 上禁用，或减小 `num_speculative_tokens`；参见 回滚
`storageUris` 被 API server 拒绝	平台上的 KServe 版本早于 `storageUris` 的引入	改用选项 B（Modelcar）或选项 C（PVC）
rollout 期间 service 一直保持 NotReady，或者 Knative 将 revision 标记为 NotReady 并提示 progress-deadline 超时	带有 draft artifact 的冷启动比不带时更慢——backbone 和 EAGLE head 的 torch.compile + engine profiling 可能会超过 readiness deadline。我们在 A30 + Llama-3.1-8B 上的 EAGLE-3 冷启动约为 **5** 分钟	在 `Standard` mode 下，提高 predictor readiness probe，使其能够容忍 5 分钟启动，例如设置 `initialDelaySeconds: 60`、`periodSeconds: 10` 和 `failureThreshold: 30`。在 `Knative` mode 下，将 `serving.knative.dev/progress-deadline` annotation 设为 `1800s`
客户端在 spec decode 下使用 `min_p` 或 `logit_bias` 时观察到意外的采样行为	在启用推测解码时，这两个参数会被 vLLM 静默忽略（engine init 时会打印 warning）	从请求中移除该参数，或者在依赖这些参数的 service 上禁用推测解码

对于 pod 级别的问题，可以使用标准的 inference-service 故障排查命令：

```
kubectl describe inferenceservice <name> -n <your-namespace>
kubectl logs -n <your-namespace> -l serving.kserve.io/inferenceservice=<name>
```

注意事项和已知限制

- 结果会随 **workload** 形态大幅波动——回退和加速都是真实存在的。 upstream V0 benchmarks 在高 QPS 下报告了 1.4×–1.8× 的变慢。我们自己的 A30 + Qwen3-8B N-gram

测试（参见 内部验证快照 — N-gram）即使在高重叠的 code workload 上也出现了轻微回退。在相同硬件上，Llama-3.1-8B 的 EAGLE-3（参见 内部验证快照 — EAGLE-3）在 code-refactor 上达到了 **1.84×** 加速，但在 **chat** 上基本持平（约 **0.99×**）——同一个 model、同一种方法、同一个 pod，仅仅因为 prompt 形态不同，实际收益就出现了 2 倍的差异。请始终结合你的生产流量 profile 进行验证。

- **N-gram 会禁用 async scheduling**。 在最近的 vLLM 版本中，启用 `ngram` 方法会强制关闭 async scheduling（predictor 日志会显示 `Async scheduling not supported with ngram-based speculative decoding and will be disabled`）。如果你的 service 依赖 async scheduling 来提升吞吐量，优先考虑 EAGLE-3，或者明确测量这种权衡。
- `storageUris` 的可用性。 该字段从 KServe 0.16 开始提供。较旧的平台版本必须使用 Modelcar 或 PVC 方案。
- **Draft head** 不匹配不会报错。 与 target model 不完全匹配的 draft head 通常也能正常启动并提供服务，但接受率会非常低。启用后务必检查接受率。
- 采样参数会影响接受率。 高温度会降低接受率；请使用与生产使用一致的采样配置进行 benchmark。
- `gpu-memory-utilization` 预算。 draft artifact（EAGLE-3 head、MLP speculator、draft model）不计入 `--gpu-memory-utilization` 预算；在添加 draft artifact 时应降低该值。
- 镜像依赖。 runtime image 必须包含所选方法所需的库。如果某个方法初始化失败，请重建或替换 runtime image——参见 扩展 Inference Runtimes。
- `min_p` 和 `logit_bias` 会被静默忽略。 在推测解码下，vLLM 在 engine init 期间会记录警告 `min_p and logit_bias parameters won't work with speculative decoding.`。传入这两个采样参数的请求仍会收到 200 响应，但参数不会被真正执行——如果你的流量依赖它们，请验证这一点是否符合客户端预期。
- 与其他功能的组合。 推测解码可以与 tensor parallelism 和 continuous batching 组合使用，但它与 autoscaling 以及 EP / advanced parallelism 的交互会随 vLLM 版本而变化。带有 draft artifact 时冷启动尤其昂贵：在我们 A30 + Llama-3.1-8B + EAGLE-3 head 的实验环境中，predictor 从 container-ready 到 `Application startup complete` 约耗时 **5** 分钟（weights load 约 45 秒，draft weights 约 5 秒，`torch.compile` backbone 约 48 秒，`torch.compile` EAGLE head 约 17 秒，CUDA-graph capture 和 warmup 约 10 秒，外加约 2 分钟的 engine profiling 和 KV-cache sizing）。readiness deadline 应按这个时间设置，而不是按非推测解码基线来设置：在 `Standard` mode 下，predictor readiness probe 必须能容忍 5 分钟启动；在 `Knative` mode 下，`serving.knative.dev/progress-deadline` annotation 应设置为 `1800s`。任何 autoscaling 的 scale-from-zero SLO 也应按同样的时间来设定。

- 输出等价性。 vLLM 表示，推测解码不会改变输出分布。这是 vLLM 的属性，不是 Alauda AI 的保证——如果你的 runtime image 需要严格的等价性，请将其作为验收测试的一部分进行验证。

参考资料

- Speculative Decoding - vLLM ↗
- How Speculative Decoding Boosts vLLM Performance by up to 2.8x — vLLM Blog ↗
- Speculative Decoding Guide — vllm-ascend ↗
- KServe Multiple Storage URIs ↗
- 使用 KServe Modelcar 存储模型
- 扩展 Inference Runtimes
- 为 vLLM 推理服务启用 Expert Parallel
- 使用 CLI 创建 Inference Service

故障排查

使用 MLServer Runtime 时遇到

- 问题描述
- 根本原因分析
- 解决方案
- 总结

推理服务无法进入运行状态

- 问题描述
- 根因分析
- 解决方案
- 总结

使用 MLServer Runtime 时遇到推理服务超时

目录

问题描述
根本原因分析
解决方案
总结

问题描述

在使用推理服务体验功能并采用 MLServer runtime 时，可能会由于以下两个原因出现超时错误：

计算能力不足或推理输出 **token** 长度过长：

- 症状： 推理服务返回 **502 Bad Gateway** 错误，显示消息 `"Http failure response for \[inference service URL]: 502 OK"`。
- 详细错误信息： 通常包含一个 HTML 格式的错误页面，提示 `"502 Bad Gateway"`。
- 响应时间： 明显超过预期响应时间，甚至可能持续数分钟。

MLServer runtime 非流式返回： 目前 MLServer 的实现会等待整个推理过程完成后再返回结果。这意味着如果推理时间较长，用户将需要等待很久，最终可能触发超时。

根本原因分析

- 计算能力不足： 模型推理所需的计算资源超出了服务器的承载能力。这可能是由于模型本身较大、输入数据复杂，或服务器配置较低所致。
- 推理输出 **token** 长度过长： 模型生成的文本长度超出了服务器的处理能力或预设超时限制。

解决方案

针对上述问题，可以采取以下解决方案：

1. 增加计算资源：
   - 升级服务器配置： 考虑使用性能更高的 CPU、GPU，或增加内存。
2. 限制推理输出 **token** 的长度：
   - 调整模型参数： 在调用推理服务时，设置 `max_new_tokens` 等参数，以限制模型生成的最大 token 数量。
3. 优化模型和输入数据：
   - 模型量化或剪枝： 减小模型规模和计算复杂度，从而降低推理时间。
   - 数据预处理： 对输入数据进行预处理，例如去除冗余信息、简化数据结构，以减少模型需要处理的数据量。

总结

MLServer 超时错误通常由计算资源不足、推理输出 token 长度过长，或 MLServer runtime 的非流式返回机制引起。解决此类问题需要综合考虑硬件资源、模型特性和 runtime 配置等因素，并根据实际情况选择合适的解决方案。

推理服务无法进入运行状态

目录

问题描述
根因分析
解决方案
总结

问题描述

在部署推理服务后，它会长时间停留在非运行状态。对应的 Pod 不会在 **Alauda Container Platform** 的 工作负载 部分创建。但是，关联的 `Deployment` 资源已成功创建。

在该 `Deployment` 的 实时事件 部分，可以观察到类似的错误信息：

> `FailedCreate: Error creating: pods "gpt2-predictor-f677f684f-sjwq7" is forbidden: violates PodSecurity "baseline:latest": host namespaces (hostIPC=true)`

根因分析

此问题由 **Pod Security Admission** 机制引起，该机制已在您的 Kubernetes 集群中启用，并且配置了较为严格的安全策略。当推理服务的 `Pod` 尝试使用特权功能时，例如 host namespaces（比如通过 `hostIPC=true`），该策略会阻止其创建，以防止潜在的安全漏洞。

在这个特定场景中，使用 `hostIPC=true` 违反了 "baseline" Pod Security Standard，因为它明确禁止使用 host namespaces，以确保 Pod 隔离。

解决方案

要解决此问题，我们首先建议检查推理服务配置。如果运行时并不严格需要 `hostIPC: true` 之类的特权模式，最安全的做法是修改工作负载配置并移除这些特权要求。这样可以在不影响安全性的前提下解决问题。

如果您的工作负载确实需要这些特权功能，请按照以下步骤调整安全策略级别：

1. 导航到 **Projects** 视图，并选择包含推理服务的 **Project**。
2. 在 **Namespace** 列表中，找到服务所在的 namespace，然后单击该行右侧的 **"..."** 按钮。
3. 在下拉菜单中，选择 **Update Pod Security Admission**。
4. 在弹出的窗口中，将所有三种安全模式（**Enforce**、**Audit** 和 **Warn**）的 **Security Standard** 设置为 `Privileged`。
5. 单击 **Update** 保存更改。

总结

推理服务 Pod 无法启动，通常是因为其配置违反了 namespace 的 Pod 安全策略。将 **namespace** 的 **Pod Security Admission** 级别调整为 `Privileged` 后，即可允许推理服务 Pod 成功创建并运行。

推理网关

对消费者进行身份验证

简介
使用场景
前提条件
步骤
验证
了解更多
下一步

配置令牌配额

简介
使用场景
前提条件
步骤
验证
了解更多
后续步骤

令牌使用量

简介
用例
前提条件
步骤
验证
了解更多
下一步

路由到 LLM 提供商

简介
使用场景
前提条件
步骤
验证
了解更多

Token 用量分摊计费

简介
使用场景
前提条件
步骤
验证
故障排查
了解更多
下一步

对消费者进行身份验证

目录

简介
使用场景
前提条件
步骤
使用 OIDC 或 JWT 进行身份验证
使用 API key 进行身份验证
验证
了解更多
下一步

简介

Envoy AI Gateway 会在边缘对每个推理请求进行身份验证，并将调用方的身份传递给下游策略。身份验证通过 Envoy Gateway 的 `SecurityPolicy` 资源进行配置，该资源会附加到由 `AIGatewayRoute` 生成的 `HTTPRoute`。在识别出调用方后，会将选定的 claim 复制到请求头中，供 token 配额和使用量计量以每租户键的形式消费。

这会把每个消费者的凭证，例如 SSO (Single Sign-On) token 或 API key，转换为配额和计量可以作用的身份。这是共享网关上多租户模型服务的基础。

使用场景

- 开发人员从平台 Identity Provider 获取 JWT (JSON Web Token) 并使用它调用网关，从而让网关按用户强制执行 token 配额。
- CI 作业提供 service-account token，使自动化流量被归属到某个团队，而不是某个个人。
- 无法运行交互式登录的机器消费者提供静态 API key，并将其映射到已知租户。

前提条件

1. 已安装 Envoy AI Gateway。它由 Alauda AI 通过 `default` `AmlCluster` 中的 `envoyAIGateway` 组件进行管理；请参阅安装 Alauda AI 和 Envoy AI Gateway 简介。
2. 一个 `AIGatewayRoute` 已经将请求路由到一个或多个后端。
3. 对于 OIDC/JWT 路径：需要一个可访问 JWKS 端点的 OIDC issuer。平台内置的身份提供方 Dex 是默认选项；任何其他 OIDC issuer（Keycloak、Auth0、Okta、GitHub OIDC、企业 Entra ID 租户）也都可以，只要网关能够访问其 `/.well-known/openid-configuration` 和 JWKS URL。
4. 对于 API key 路径：集群需要具备在网关命名空间中创建 `Secret` 对象的权限。

> **NOTE**
>
> 请在专用命名空间中创建 `Gateway` 和 `AIGatewayRoute`（例如 `maas-system`），而不是在 Envoy Gateway 控制平面命名空间 `envoy-gateway-system` 中创建。放置在控制平面命名空间中的网关可能不会在其监听器上应用 AI Gateway 请求处理过滤器和 `SecurityPolicy`，这会悄然破坏路由和策略强制执行。请参阅 Envoy AI Gateway。

步骤

1 使用 **OIDC** 或 **JWT** 进行身份验证

验证由 OIDC issuer 签发的 token。平台内置的 Dex 是默认 issuer；它还可以代理外部身份来源，例如 LDAP 或其他 OIDC provider，以便这些用户获取平台 token。这些连接器

在平台 IdP (Identity Provider) 管理中进行配置。有关平台 IdP 配置，请参阅 Identity Providers↗。

任何具有可访问 JWKS 端点的 OIDC issuer 都可以使用。当消费者不是平台用户时，请将下面的 `issuer` 和 `remoteJWKS.uri` 替换为你选择的 issuer，例如企业 Keycloak realm 或 SaaS IdP，这样网关就能接受其 token，而无需平台账户。

将网关指向 OIDC issuer，并将其 claim 映射到身份请求头：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: maas-oidc-auth
  namespace: <your-namespace>
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: <aigatewayroute-name>  # HTTPRoute generated by your AIGa
tewayRoute
  jwt:
    providers:
      - name: platform-idp
        issuer: https://<platform-address>/dex
        audiences:
          - <gateway-client-id>   # reject tokens minted for other cl
ients of the same issuer
        remoteJWKS:
          uri: https://<platform-address>/dex/keys
        claimToHeaders:
          - claim: sub      # caller identity, used as the per-user q
uota and metering key
            header: x-user-id
          - claim: groups    # single-valued group/department claim
(see note: array claims are unsupported)
            header: x-user-group
          - claim: namespace  # custom scalar claim for per-namespace
chargeback; absent by default (see note)
            header: x-user-namespace
          - claim: email
            header: x-user-email
```

- `<platform-address>`：平台访问地址。Dex 将其 issuer 发布在 `/dex`，JWKS 发布在 `/dex/keys`。
- `<aigatewayroute-name>`：由你的 `AIGatewayRoute` 生成的 `HTTPRoute` 名称。
- `audiences`：网关接受的 token audience。对于共享 IdP，如果省略此项，则会接受来自同一 issuer 的任何有效 token——包括为其他客户端签发的 token——这些 token 仍然会解析为 `x-user-id` 并消耗配额。请将其设置为网关 token 签发所针对的 client ID；如果你的 issuer 未设置可区分的 `aud`，请为网关注册一个专用 client。
- `claimToHeaders`：身份与策略之间的桥梁。生成的请求头（`x-user-id`、`x-user-group`、`x-user-namespace`）会成为 token 配额的选择器键，以及使用量计量和 chargeback 的标签值。

> **NOTE**
>
> `claimToHeaders` 仅支持标量 claim（string、int、double、bool）；不支持数组类型的 claim，且不会填充请求头。标准 OIDC `groups` claim 通常是数组——若要将其用作 `x-user-group` / `department`，请从 IdP 连接器中暴露一个单值 claim（例如 primary-group 或专用的 `department` claim）并进行映射。如果 `x-user-group` 仍为空，则按部门的计量、配额和层级会悄然退回为不分组。
>
> `namespace` 不是标准 OIDC claim：必须在上游 IdP 连接器中添加，默认情况下不存在。`x-user-namespace` 是 Metering Token Usage 消费的按命名空间 chargeback 键；仅在按命名空间或租户计费时映射它。对于平台默认不发出的其他任意属性，例如 subscription tier，如需将其作为策略键，请在连接器中添加该 claim，并通过额外的 `claimToHeaders` 条目进行映射。

> **TIP**
>
> 若要在不中断流量的情况下逐步发布，请先设置 `jwt.optional: true` 并进行观测。待所有消费者都提供有效 token 后再将其移除。

2 使用 API key 进行身份验证

> **NOTE**

如果 `Gateway` 创建在 Envoy Gateway 控制平面命名空间 `envoy-gateway-system` 中，则 `apiKeyAuth`（与 model routing 类似）不会被强制执行：`SecurityPolicy` 会报告 `Accepted=True`，但错误的 key 或缺失的 key 仍然会返回 `200`，并且不会注入 `x-user-id`。这与上文前提条件说明中的控制平面 listener-skip 问题相同——不是 Envoy Gateway 版本 bug。解决方法是将 `Gateway` 和 `AIGatewayRoute` 创建在专用命名空间中（推荐，例如 `maas-system`），这样 `apiKeyAuth` 就能原生强制执行，无需补丁。可通过一次无 key 请求进行验证：专用命名空间中的网关会返回 `401`。如果网关必须保留在 `envoy-gateway-system` 中，请参阅本节末尾的受支持修复方案。

对于无法执行 OIDC 流程的机器消费者，请改用静态 API key 进行验证。这里没有签发服务：集群管理员为每个消费者生成一个随机字符串，将其存储在 `Secret` 中，并通过带外方式共享。网关的数据平面通过在该 `Secret` 中查找所提供的值来验证每个请求。

为每个消费者生成一个 key，并将它们存储在同一个 Opaque `Secret` 中。每个 data-map key 都是下游策略看到的客户端标识符；每个 value 都是消费者提供的 **API key**：

```
kubectl -n <your-namespace> create secret generic maas-api-keys \
  --from-literal=alice="$(openssl rand -hex 32)" \
  --from-literal=ci-runner="$(openssl rand -hex 32)"
```

使用 `SecurityPolicy` 将该 `Secret` 绑定到路由：

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: SecurityPolicy
metadata:
  name: maas-apikey-auth
  namespace: <your-namespace>
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: <aigatewayroute-name>  # HTTPRoute generated by your AIGa
tewayRoute
  apiKeyAuth:
    credentialRefs:
      - name: maas-api-keys     # Secret whose data keys are the clie
nt identifiers
    extractFrom:
      - headers:
          - X-API-Key           # dedicated header avoids the "Bearer
" prefix problem of Authorization
    forwardClientIDHeader: x-user-id  # matched client identifier is
injected as this header for downstream policies
    sanitize: true              # strip the raw API key from the requ
est before it reaches the model backend
```

- `credentialRefs`：一个或多个保存凭证的 Opaque `Secret`。每个 data-map key 都是客户端标识符，每个 value 都是字面量 API key。添加消费者只需对一个条目执行 `kubectl patch`；撤销则只需删除一个 key。
- `extractFrom`：Envoy 从哪里读取所提供的 key。该过滤器执行字面字符串比较，因此建议使用专用请求头，例如 `X-API-Key`。复用 `Authorization` 需要将 value 及其 `Bearer` 前缀一并存储，这与同一网关上的 OIDC 路径会产生冲突。
- `forwardClientIDHeader`：将匹配到的客户端标识符传递给上游以及后续过滤器的请求头。请使用与 OIDC `claimToHeaders` 目标相同的名称（`x-user-id`），这样 token 配额和使用量计量在两条认证路径上就能看到一致的 key。
- `sanitize`：防止原始 API key 泄漏到模型后端或在下游日志中被记录。

如果网关必须保留在 **`envoy-gateway-system`** 中，则 `apiKeyAuth` 的强制执行会受到上述相同控制平面 listener-skip 的影响，并且没有任何按路由的 `SecurityPolicy` 更改可以修复它。受支持的修复方案是将 `Gateway` 和 `AIGatewayRoute` 移动到专用命名空

> 间（首选），或者将 Envoy AI Gateway 升级到一个缩小 listener-skip 范围的版本（v0.6.0 / Alauda `release-0.6.0-alauda`）。手工编写的 `EnvoyPatchPolicy` 也许可以作为临时权宜之计，通过修改监听器过滤器链来实现，但它具有版本相关性且较为脆弱——它依赖于运行中的 Envoy 构建的精确过滤器布局——因此不建议用于生产环境。

验证

确认策略已被接受。`SecurityPolicy` 状态是 ancestor-scoped，因此 jsonpath 比大多数资源要多一层：

```
kubectl get securitypolicy <policy-name> -n <your-namespace> \
  -o jsonpath='{.status.ancestors[*].conditions[?(@.type=="Accepted")].st
atus}'
```

当策略已编程时，该命令会返回 `True`。

对于 OIDC 路径，使用有效 token 发送请求，并确认上游服务接收到 `x-user-id`、`x-user-group` 和 `x-user-email` 请求头。

对于 API key 路径，发送匹配的 `X-API-Key`，并确认上游看到的 `x-user-id` 已设置为匹配到的客户端标识符：

```
curl -sS -H "X-API-Key: <alice-key>" \
  https://<gateway-host>/v1/chat/completions \
  -d '{"model":"<model>","messages":[{"role":"user","content":"ping"}]}'
```

错误的 key 或缺失的 key 会在请求到达任何后端之前，由网关返回 `401 Unauthorized`。

了解更多

- 配置 token 配额
- Metering Token Usage

下一步

在身份请求头传播完成后，请配置配置 token 配额以强制执行按租户的 token 预算。

配置令牌配额

目录

简介
使用场景
前提条件
步骤
启用全局限流后端
在路由上捕获令牌用量
按身份定义令牌预算
验证
了解更多
后续步骤

简介

Envoy AI Gateway 可以根据 令牌用量 而不是请求数量进行限流，并基于身份为每个调用方跟踪独立预算。这可以防止单个消费者或失控的智能体耗尽共享的模型预算，并且让你能够在同一个网关上提供按用户、按部门以及按层级划分的令牌配额。

一个令牌配额由三部分组成：

- `AIGatewayRoute.llmRequestCosts` 将每个 LLM 响应中的令牌计数提取到 Envoy 动态元数据中。
- 全局限流后端（Redis）累积该成本。由于令牌成本只有在响应之后才知道，因此不能由按 Pod 的本地限流器来跟踪。
- 类型为 `Global` 的 `BackendTrafficPolicy` 定义预算和身份键。

使用场景

- 为每个用户在昂贵模型上提供每月令牌预算，并为高级层级提供更高预算。
- 限制某个部门通过其所有应用程序可消耗的总令牌数。
- 保护共享模型，防止某个行为异常的自动化账户过度消耗。

前提条件

1. 已安装 Envoy AI Gateway，并且有一个 `AIGatewayRoute` 正在路由到你的模型后端。请确认相关 CRD 已存在：

   ```
   kubectl get crd \
     aigatewayroutes.aigateway.envoyproxy.io \
     backendtrafficpolicies.gateway.envoyproxy.io
   ```

2. 调用方身份通过请求头传递，例如 `x-user-id`。请参见 验证消费者身份。如果没有身份头，下面的预算会退化为所有调用方共享的单一计数器，因此这一步决定了配额是否 *按消费者* 生效。
3. 已有一个 Redis 实例可用。默认提供程序请使用 Redis 缓存服务↗ 中的托管实例，然后记录其访问地址（`host:port`）和凭证。在继续之前，请先验证集群可以访问该实例：

```
# On clusters without docker.io egress (air-gapped / mirror-only), repl
ace
# redis:7-alpine with a Redis image from your registry mirror (it ships
redis-cli).
kubectl run redis-probe --rm -i --restart=Never \
  --image=redis:7-alpine -- \
  redis-cli -h <redis-host> -p <redis-port> PING
# expect: PONG
```

4. 记下 `Gateway` 的名称和命名空间——后续需要它们来重启正确的数据平面代理：

```
kubectl get gateway -A
```

> **NOTE**
>
> 请在专用命名空间中创建 `Gateway` 和 `AIGatewayRoute`（例如 `maas-system`），而不要放在 Envoy Gateway 控制平面命名空间 `envoy-gateway-system` 中。放置在控制平面命名空间中的网关，可能不会将 AI Gateway 请求处理过滤器和 `SecurityPolicy` 应用到其监听器，这会静默破坏路由和策略执行。请参见 Envoy AI Gateway。

步骤

1 启用全局限流后端

本地限流器无法累积来自响应的令牌成本，因此网关必须使用由 Redis 支持的 Global Rate Limit 服务。通过 Redis 缓存服务↗ 创建一个 Redis 实例，并从实例详情页复制其访问地址。然后将该地址设置到 `envoy-gateway-config` `ConfigMap`（命名空间 `envoy-gateway-system`）中的 `data."envoy-gateway.yaml"` 下：

```
kubectl edit configmap envoy-gateway-config -n envoy-gateway-system
```

在顶层配置下添加 `rateLimit` 块（保留现有键，例如 `gateway:` 和 `provider:` 不变）：

```
rateLimit:
  backend:
    type: Redis
    redis:
      url: <redis-host>:<redis-port>  # access address of the Redis i
nstance
      # tls:                          # uncomment if Redis requires T
LS
      #   certificateRef:
      #     name: redis-client-cert
```

- `<redis-host>:<redis-port>`：从 Redis 实例详情页复制的访问地址。
- 对于受密码保护的 Redis，请使用标准 Redis URI 形式将凭证嵌入 URL 中：`url: redis://<username>:<password>@<redis-host>:<redis-port>`（仅密码认证时省略 `<username>:`）。Envoy Gateway v1.5.x 在 `rateLimit.backend.redis` 下只暴露 `url` 和 `tls.certificateRef`，没有单独基于 `Secret` 的认证字段。由于此时凭证会存放在 `envoy-gateway-config` `ConfigMap` 中，请限制其读取权限，或者改用网络隔离 / ACL 受限的 Redis。完整 schema 请参见 Envoy Gateway rate-limit 文档↗。

Envoy Gateway 控制平面只会在启动时读取该引导配置，不会热重载，因此需要重启其 `Deployment` 以应用更改，然后确认专用的 `envoy-ratelimit` `Deployment` 运行正常：

```
kubectl rollout restart deployment envoy-gateway -n envoy-gateway-sys
tem
kubectl rollout status  deployment envoy-gateway   -n envoy-gateway-s
ystem
kubectl rollout status  deployment envoy-ratelimit -n envoy-gateway-s
ystem
# the envoy-ratelimit Deployment is created by the EG controller the
first time
# rateLimit.backend is set; if it never appears, the config above was
not parsed
```

NOTE

每个 Envoy Gateway 使用一个 Redis 实例即可。任何可达的 Redis 都可以工作，但出于可用性和备份考虑，建议使用托管实例。

如果 Gateway 在启用限流后端之前已经运行，还需要重启其数据平面，以便代理加载限流服务：

```
kubectl rollout restart deployment -n envoy-gateway-system \
  -l gateway.envoyproxy.io/owning-gateway-name=<gateway-name>
```

2 在路由上捕获令牌用量

为 `AIGatewayRoute` 添加 `llmRequestCosts`，使网关把令牌计数写入 Envoy 动态元数据（过滤器之间用来通信的每请求临时空间）中，命名空间为 `io.envoy.ai_gateway`。下一步的限流过滤器会从该命名空间中读取。

```
apiVersion: aigateway.envoyproxy.io/v1alpha1
kind: AIGatewayRoute
metadata:
  name: <aigatewayroute-name>
  namespace: <your-namespace>
spec:
  # ... existing parentRefs and rules ...
  llmRequestCosts:
    - metadataKey: llm_input_token
      type: InputToken
    - metadataKey: llm_output_token
      type: OutputToken
    - metadataKey: llm_total_token
      type: TotalToken
```

- `metadataKey`：写入 `io.envoy.ai_gateway` 下计数的键名。名称可自行选择；下面的 `BackendTrafficPolicy` 必须引用同一个字符串。
- `type`：`InputToken` 统计 prompt，`OutputToken` 统计 completion，`TotalToken` 为两者之和。若要使用自定义公式，可选 `CEL`——例如，由于输出令牌更慢，可以将其计费为 3 倍：

```
  - metadataKey: llm_weighted_cost
    type: CEL
    cel: "input_tokens + output_tokens * 3"
```

应用并确认路由仍被接受（新字段不应破坏转换）：

```
kubectl get aigatewayroute <aigatewayroute-name> -n <your-namespace>
\
  -o jsonpath='{.status.conditions[?(@.type=="Accepted")].status}'
# expect: True
```

3 按身份定义令牌预算

绑定一个带有 `Global` 限流的 `BackendTrafficPolicy`。将请求成本设为 `0`，并将响应成本设为捕获到的令牌元数据，这样只有令牌会计入限制。使用 `clientSelectors` 按身份和模型范围限定预算。

```yaml
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: BackendTrafficPolicy
metadata:
  name: maas-token-quota
  namespace: <your-namespace>
spec:
  targetRefs:
    - group: gateway.networking.k8s.io
      kind: HTTPRoute
      name: <aigatewayroute-name>
  rateLimit:
    type: Global
    global:
      rules:
        - clientSelectors:
            - headers:
                - name: x-user-id       # identity header from the SecurityPolicy
                  type: Distinct
                - name: x-ai-eg-model
                  type: Exact
                  value: my-llm
          limit:
            requests: 200000             # 200k tokens per window
            unit: Hour
          cost:
            request:
              from: Number
              number: 0                   # requests do not count
            response:
              from: Metadata              # tokens count
              metadata:
                namespace: io.envoy.ai_gateway
                key: llm_total_token
```

- `x-user-id` 使用 `type: Distinct` 可为每个调用方提供独立计数器，从而实现按用户配额。若要按部门汇总到一个预算，可使用 `x-user-group`；若要实现分层限制，也可以使用 `type: Exact` 匹配特定组（例如 `premium`）。
- 这里的 `limit.requests` 会被解释为 令牌 预算，因为成本来自令牌元数据。使用 `200000` 令牌/小时，并且典型聊天调用大约消耗 1.5–2k 令牌，那么在触发限流之前，每个调用方大约可发出 ~100–130 次调用/小时。

> - `cost.request.number: 0` 表示如果请求未到达上游（例如请求体格式错误），则不会消耗配额。若你也希望按调用次数进行预检限流，请将其设为 `1`。
> - `cost.response.metadata.key` 必须与路由上声明的 `metadataKey` 一致。

验证

使用有效的身份令牌发起一轮短时间的请求，然后再使用另一个身份发起请求，确认只有第一个身份会被限流：

> **NOTE**
>
> 对于上面现实的 `200000` 令牌/小时预算，少量单词级请求消耗不到 1% 的配额，因此它们都会返回 `200`，你将不会看到 `429`。若要在短演示中观察到限流，请临时缩小预算——在 `BackendTrafficPolicy` 中将 `limit.requests: 200` 和 `unit: Minute`，重新应用，然后重启数据平面代理（`kubectl rollout restart deployment -n envoy-gateway-system -l gateway.envoyproxy.io/owning-gateway-name=<gateway-name>`）。演示成功后再恢复为真实预算。或者保留生产预算，发送几百个带大 prompt 和较长 `max_tokens` 的请求。

下面的验证使用 JWT 头 `Authorization: Bearer`。如果你改为配置了 **API-key** 认证（验证消费者身份 → “使用 API key 认证”），网关会从专用的 `X-API-Key` 头中读取凭证——将每个 `-H "Authorization: Bearer $TOKEN_…"` 替换为 `-H "X-API-Key: $TOKEN_…"`。

```
GATEWAY='http://<gateway-address>'
TOKEN_ALICE='<alice-jwt-or-api-key>'
TOKEN_BOB='<bob-jwt-or-api-key>'

# With the demo budget above (200 tokens/minute), expect early 200s then
429 once it is used.
for i in $(seq 1 8); do
  curl -s -o /dev/null -w "alice #$i -> %{http_code}\n" \
    -H "Authorization: Bearer $TOKEN_ALICE" \
    -H 'Content-Type: application/json' \
    -d '{"model":"my-llm","messages":[{"role":"user","content":"hi"}]}' \
    $GATEWAY/v1/chat/completions
done

# Same model, different user — must succeed because the budget is per x-u
ser-id.
curl -s -o /dev/null -w "bob       -> %{http_code}\n" \
  -H "Authorization: Bearer $TOKEN_BOB" \
  -H 'Content-Type: application/json' \
  -d '{"model":"my-llm","messages":[{"role":"user","content":"hi"}]}' \
  $GATEWAY/v1/chat/completions
```

一轮耗尽 alice 配额的运行会输出类似 `alice #1 -> 200 … alice #5 -> 429 … bob -> 200` 的结果。若要直接在 Redis 中检查计数器，可搜索包含该身份值的键——Envoy Gateway 为每个计数器命名为 `<gateway-namespace>/<gateway-name>/<listener>_<route>_..._<x-user-id-value>_..._<window-timestamp>`，因此用户身份是最简单的过滤条件：

```
# <alice-x-user-id-value>: the x-user-id value the SecurityPolicy injects
for alice
# (the JWT claim / API-key mapping value that appears in the counter ke
y), not her token.
# Air-gapped clusters: replace redis:7-alpine with a mirrored Redis imag
e.
kubectl run redis-cli --rm -i --restart=Never \
  --image=redis:7-alpine -- \
  redis-cli -h <redis-host> -p <redis-port> \
  --scan --pattern "*<alice-x-user-id-value>*" | head
```

WARNING

如果 Redis 不可达，Envoy 默认会 **open**：请求会在不计量的情况下通过，直到 Redis 恢复。请监控限流 pod 的日志（`kubectl logs deploy/envoy-ratelimit -n envoy-gateway-system`），并对其 `Ready` 条件进行告警，以免静默的配额失效未被发现。

了解更多

- 验证消费者身份
- 计量令牌用量

后续步骤

配置 计量令牌用量，以按租户报告消耗并进行分摊计费。

令牌使用量计量

目录

简介
用例
前提条件
步骤
为令牌指标添加身份标签
将指标收集到平台中
构建统一的使用量 dashboard
验证
了解更多
下一步

简介

Envoy AI Gateway 会暴露遵循 OpenTelemetry GenAI 语义约定的 Prometheus 指标，其中包括每个请求的令牌使用量。通过将调用方身份作为指标标签添加，并通过平台监控栈收集该指标，你可以按部门、命名空间和模型获得统一的令牌消耗视图。同一份数据还会通过 Alauda Cost Management 用于 chargeback。

该流水线如下：网关发出令牌指标，身份作为标签附加，`PodMonitor` 将指标收集到平台中，`MonitorDashboard` 负责展示。日常查看无需编写原始 PromQL。

用例

- 按模型展示每个部门的令牌消耗，并按项目隔离。
- 跟踪平台中哪些模型带来了最多的令牌使用量。
- 提供 Alauda Cost Management 用于生成 chargeback 报告的使用数据。

前提条件

1. 已配置 `llmRequestCosts` 的 `AIGatewayRoute`。请参见配置令牌配额。如果没有 `llmRequestCosts`，网关仍会发出 `gen_ai_client_token_usage_token`，但每个请求的令牌计数都将为零。
2. 调用方身份已作为请求头传递。请参见认证消费者。
3. 集群已启用平台监控。可通过检查 Prometheus operator CRD 来确认：

   ```
   kubectl get crd podmonitors.monitoring.coreos.com
   ```

4. 在接入监控之前，先确认指标确实已经发出。通过网关发送一个请求，然后读取数据平面 proxy pod 上 ExtProc sidecar 的管理端口：

   ```
   POD=$(kubectl get pod -n envoy-gateway-system \
     -l gateway.envoyproxy.io/owning-gateway-name=<gateway-name> \
     -o jsonpath='{.items[0].metadata.name}')
   kubectl port-forward -n envoy-gateway-system pod/$POD 1064:1064 &
   curl -s http://localhost:1064/metrics | grep gen_ai_client_token_usage_
   token | head
   # expect lines like: gen_ai_client_token_usage_token_sum{gen_ai_operati
   on_name="chat",...} 42
   ```

   如果没有出现任何 `gen_ai_*` 样本，那么下面的任何抓取都不会生效——请先修复路由 / ExtProc 连接。

NOTE

请在专用 namespace 中创建 `Gateway` 和 `AIGatewayRoute`（例如 `maas-system`），不要创建在 Envoy Gateway 控制平面 namespace `envoy-gateway-system` 中。放在控制平面 namespace 中的 gateway 可能不会将 AI Gateway 请求处理 filter 和 `SecurityPolicy` 应用到其 listener，这会静默破坏路由和策略执行。请参见Envoy AI Gateway。

步骤

1 为令牌指标添加身份标签

默认情况下，令牌指标 `gen_ai_client_token_usage_token` 只包含 OpenTelemetry GenAI 标准标签（model、provider、operation、token type）。通过在 Envoy AI Gateway controller 中将身份 header 映射为指标标签，可将其扩展为包含调用方身份维度——计费 namespace 和调用方部门。

controller 从 `ai-gateway-controller` `Deployment` 上的 **CLI flag** `--metricsRequestHeaderAttributes=<header>:<label>[,<header>:<label>...]` 读取映射。如果 controller 是通过 Helm 安装的，则 chart 会从某个 values 键（例如 `controller.metricsRequestHeaderAttributes`）渲染此 flag；请提供你的 release 名称和 chart 引用，然后执行 `helm upgrade --reuse-values`。如果你直接管理 `Deployment`，则需要 patch 其容器参数。

该 flag 是一个单个以逗号连接的 `<header>:<label>` 映射，不是可重复列表：传入两次会导致 controller 只保留最后一份。平台还自带了默认键值对（例如 `x-user-name:user` 和 `x-access-meta:client_id`），数据平面的 sidecar 已经会发出这些标签——因此你必须将这一个 flag 改写为现有键值对与新键值对的并集。省略任何一对，都会在下次重启时从所有 proxy 中移除该指标标签。

先读取数据平面当前实际发出的键值对（proxy sidecar 才是事实来源——controller flag 可能已经被收窄）：

```
PROXY=$(kubectl -n envoy-gateway-system get pod \
  -l gateway.envoyproxy.io/owning-gateway-name=<gateway-name> \
  -o jsonpath='{.items[0].metadata.name}')
kubectl -n envoy-gateway-system get pod "$PROXY" \
  -o jsonpath='{.spec.initContainers[?(@.name=="ai-gateway-extpro
c")].args}' \
  | jq -r 'index("-metricsRequestHeaderAttributes") as $i | .[$i+1]'
# e.g. x-user-name:user,x-user-namespace:user_namespace,x-access-met
a:client_id  (platform defaults)
```

然后用完整集合再加上 `x-user-group:department` 重写 controller 的单个 flag。请将下面左侧替换为上面命令实际输出的内容——不要删除任何键值对：

```
# Locate the flag's argument index on the controller Deployment
IDX=$(kubectl -n envoy-gateway-system get deploy ai-gateway-controlle
r \
  -o jsonpath='{.spec.template.spec.containers[0].args}' \
  | jq 'map(test("^--?metricsRequestHeaderAttributes=")) | index(tru
e)')

# Replace it with the UNION of existing pairs + the new one
kubectl -n envoy-gateway-system patch deploy ai-gateway-controller --
type json -p '[
  {"op":"replace","path":"/spec/template/spec/containers/0/args/'"$ID
X"'",
   "value":"--metricsRequestHeaderAttributes=x-user-name:user,x-user-
namespace:user_namespace,x-access-meta:client_id,x-user-group:departm
ent"}
]'
# If the flag is not present anywhere (IDX is null), add it instead w
ith the same full value:
#   --type json -p '[{"op":"add","path":"/spec/template/spec/containe
rs/0/args/-","value":"--metricsRequestHeaderAttributes=...,x-user-gro
up:department"}]'

kubectl -n envoy-gateway-system rollout status deploy ai-gateway-cont
roller

# The mapping is rendered into each data-plane proxy pod's ExtProc si
decar args
# at pod-creation time. A controller-only rollout does NOT update alr
eady-running
# proxy pods, so recreate them and wait:
kubectl rollout restart deployment -n envoy-gateway-system \
  -l gateway.envoyproxy.io/owning-gateway-name=<gateway-name>
kubectl rollout status deployment -n envoy-gateway-system \
  -l gateway.envoyproxy.io/owning-gateway-name=<gateway-name>
```

- `x-user-namespace` → `user_namespace`：请求所计费的 namespace 或 tenant。这是 通过 Cost Management 进行 chargeback 所按组使用的按 namespace 键，因此只要 Cost Management 在使用，就应保留它。
- `x-user-group` → `department`：用于 dashboard 的低基数身份维度。由 `SecurityPolicy` 根据 IDP `groups` claim 设置。

NOTE

`--metricsRequestHeaderAttributes` 映射会在创建 proxy pod 时被固化到 ExtProc sidecar 参数中。仅 patch controller 并等待其 rollout 不够——必须重新创建 proxy pod（上面的 `rollout restart`）之后，新的标签才会出现。如果下面的验证返回空输出，说明 proxy pod 没有被重新创建。

NOTE

`user_namespace`（计费 namespace）是可靠的默认分组——只要映射了 `x-user-namespace`，它就始终存在。`department`（`x-user-group`）只有在 IDP 发出单值 group claim 时才是有用的低基数维度——标准 OIDC `groups` claim 是数组，不受 `claimToHeaders` 支持，因此在其他情况下 `department` 会保持为空——请参见认证消费者。避免使用按用户标签（`x-user-id`）：它会产生高基数 series（每个用户 × 模型 × 令牌类型一条），因此仅在需要按用户报表且保留窗口能限制 series 数量时才添加它。

rollout 完成后，发送一条新的请求并确认样本上出现了新标签。`user_namespace` 始终存在；只有在配置了标量 group claim 时才会出现 `department`：

```
curl -s http://localhost:1064/metrics | grep 'user_namespace=' | head
```

2 将指标收集到平台中

该指标由 AI Gateway external processor（ExtProc）发出，它作为每个数据平面 proxy pod 的 sidecar 运行（声明为 Kubernetes 原生 sidecar / `initContainer`），并在容器端口 `1064`（名为 `aigw-metrics`）上暴露 Prometheus metrics endpoint。它通过 `PodMonitor` 直接从 proxy pod 抓取。平台工作流请参见 指标管理↗。

在 Alauda 平台上，此 sidecar 的 `PodMonitor` 往往已经预先安装。请先检查——如果在同一个 endpoint 上再创建第二个 `PodMonitor`，会形成重叠的 scrape pool，从而对每个 proxy pod 抓取两次，生成带有不同 `job` 标签的重复 series，并使 dashboard 以及 chargeback 查询中的每个 `increase()` / `sum` 翻倍（两者都不会按 `job` 过滤）：

```
kubectl get podmonitor -A \
  -o jsonpath='{range .items[*]}{.metadata.namespace}/{.metadata.nam
e}{"\t"}{.spec.podMetricsEndpoints[*].port}{"\n"}{end}'
# The Alauda platform ships `ai-gateway-extproc-metrics` targeting th
e `aigw-metrics`
# port. If any PodMonitor already lists `aigw-metrics`, SKIP the appl
y below and go
# straight to the scrape check.
```

只有在没有任何 `PodMonitor` 已经指向 `aigw-metrics` 时，才去发现 Prometheus operator 用来选择 `PodMonitor` 对象的 label（以便下面的资源确实会被拾取），然后创建一个：

```
kubectl get prometheus -A \
  -o jsonpath='{range .items[*]}{.metadata.namespace}/{.metadata.nam
e}: {.spec.podMonitorSelector}{"\n"}{end}'
# On the Alauda platform this is typically: prometheus: kube-promethe
us
```

然后应用该 `PodMonitor`，并将该 label 放在其自身的 `metadata.labels` 中（不是 `selector`——这是两个不同的东西）：

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: ai-gateway-metrics
  namespace: envoy-gateway-system
  labels:
    prometheus: kube-prometheus   # must match your Prometheus's spe
c.podMonitorSelector
spec:
  selector:
    matchLabels:
      app.kubernetes.io/managed-by: envoy-gateway
      app.kubernetes.io/component: proxy
  podMetricsEndpoints:
    - port: aigw-metrics
      path: /metrics
      interval: 30s
```

- `metadata.labels`：Prometheus 通过它发现 `PodMonitor`。没有正确的 label，资源虽然存在，但对 scrape 流程不可见。
- `spec.selector`：匹配集群中每个 Envoy Gateway 数据平面 proxy pod。若要限制到某个 `Gateway`，请将其替换为 `gateway.envoyproxy.io/owning-gateway-name: <gateway-name>` 和 `gateway.envoyproxy.io/owning-gateway-namespace: <gateway-namespace>`。
- `port: aigw-metrics`：ExtProc sidecar 上提供 `/metrics` 的命名端口，对应容器端口 `1064`。

确认 Prometheus 正在抓取该 sidecar——并且恰好只有一个 scrape pool 覆盖它（两个 pool 说明重复的 `PodMonitor` 导致双重计数）。对 Prometheus 做端口转发，并按 pool 对活动目标分组：

```
PROM=$(kubectl -n <prometheus-namespace> get pod -l app.kubernetes.io/name=prometheus \
  -o jsonpath='{.items[0].metadata.name}')
kubectl -n <prometheus-namespace> port-forward $PROM 9090:9090 &

# Wait ~30s after applying a PodMonitor; the operator's config-reload is async.
curl -s 'http://127.0.0.1:9090/api/v1/targets?state=active' \
  | jq '[.data.activeTargets[] | select(.labels.endpoint=="aigw-metrics")]
        | group_by(.scrapePool)
        | map({scrapePool: .[0].scrapePool, targets: length, health: .[0].health})'
# expect: exactly ONE scrape pool covering the sidecar, health="up".
# Two pools => a duplicate PodMonitor; delete the redundant one.
```

3 构建统一的使用量 dashboard

创建一个 `MonitorDashboard` 来展示令牌使用量。使用 namespace、model 和 department 变量，以便消费者可以筛选，并依赖 **User View**，使每个 project 只能看到自己的数据。平台工作流请参见 监控 dashboards↗。

在 ExtProc `/metrics` endpoint 上，该指标以 Prometheus 文本格式暴露，OpenTelemetry 名称会规范化为下划线（如上面的前置检查所示，`gen_ai_client_token_usage_token_sum`）。不过，平台的 scrape 流程会以原始

OpenTelemetry UTF-8 名称存储该 **series**，并保留点号。因此在 Prometheus 中，该 series 为 `gen_ai.client.token.usage_token_sum`（以及 histogram 的 `_sum` / `_count` / `_bucket` 变体），必须使用带引号的 `{__name__="..."}` 形式来选择——裸下划线标识符将匹配不到任何内容并返回空 vector。内置的 GenAI 标签同样使用点号形式（`gen_ai.request.model`、`gen_ai.token.type`），并且需要带引号的 UTF-8 label 语法；而通过 header 映射得到的标签（例如 `department` 和 `user_namespace`）则保持普通标识符形式：

CAUTION

在首页面板中请按 `user_namespace` 分组：只要映射了 `x-user-namespace`，它就始终存在。`department` 标签只有在 `x-user-group` 被填充时才会出现，而这要求 IDP 提供单值 group claim——标准 OIDC `groups` claim 是数组，不受支持，因此否则 `sum by (department)` 会退化为单个空标签 series。只有在确认该标签确实存在（`grep 'department='` 有输出）之后，才按 `department` 分组；否则，请在 IDP connector 中暴露一个标量 claim。

```
# Tokens per billed namespace over the last hour
sum by (user_namespace) (
  increase({__name__="gen_ai.client.token.usage_token_sum"}[1h])
)

# Top 5 models by total tokens in the last 24h
topk(5,
  sum by ("gen_ai.request.model") (
    increase({__name__="gen_ai.client.token.usage_token_sum"}[24h])
  )
)

# Output-token share per namespace (proxy for cost; output is the exp
ensive side)
sum by (user_namespace) (
  increase({__name__="gen_ai.client.token.usage_token_sum", "gen_ai.t
oken.type"="output"}[1h])
)

# Optional: per-department view — only once a scalar group claim is c
onfigured
# (see the caution above), otherwise this returns an empty result:
#   sum by (department) (increase({__name__="gen_ai.client.token.usag
e_token_sum"}[1h]))
```

通过在 Prometheus UI 的 *Status* → *TSDB* 页面列出所有 `gen_ai*` series 名称，或使用以下命令，来确认该指标可查询：

```
curl -s 'http://127.0.0.1:9090/api/v1/label/__name__/values' \
  | jq -r '.data[]' | grep gen_ai
```

验证

发送几条已认证且会解析到不同 namespace 的请求，然后通过对任意 proxy pod 的 ExtProc sidecar metrics 端口做端口转发，确认该指标携带了身份标签：

```
POD=$(kubectl get pod -n envoy-gateway-system \
  -l gateway.envoyproxy.io/owning-gateway-name=<gateway-name> \
  -o jsonpath='{.items[0].metadata.name}')
kubectl port-forward -n envoy-gateway-system pod/$POD 1064:1064 &

# Drive a couple of authenticated requests (substitute valid tokens), the
n read the metric.
for token in '<token-tenant-a>' '<token-tenant-b>'; do
  curl -s -o /dev/null \
    -H "Authorization: Bearer $token" \
    -H 'Content-Type: application/json' \
    -d '{"model":"my-llm","messages":[{"role":"user","content":"hi"}]}' \
    http://<gateway-address>/v1/chat/completions
done

curl -s http://localhost:1064/metrics \
  | grep 'gen_ai_client_token_usage_token_sum{' \
  | grep 'user_namespace='
```

预期每个计费 namespace 至少有一个样本，例如：

```
gen_ai_client_token_usage_token_sum{user_namespace="team-a",gen_ai_reques
t_model="my-llm",gen_ai_token_type="input",...} 184
gen_ai_client_token_usage_token_sum{user_namespace="team-b",gen_ai_reques
t_model="my-llm",gen_ai_token_type="input",...} 91
```

如果你配置了标量 group claim，这些样本还会带有 `department=` 标签（可通过 `grep 'department='` 检查）。打开 dashboard，确认令牌使用量已经显示，并且可以按 namespace、model 和 department 进行筛选。

了解更多

- 认证消费者
- 配置令牌配额

下一步

在收集到令牌使用量之后，配置对令牌使用量进行费用分摊，将该指标通过 Alauda Cost Management 计价并生成按 namespace 计费的账单。

路由到 LLM 提供商

目录

简介
使用场景
前提条件
步骤
存储上游凭证
定义提供商后端
按模型路由并进行故障切换
验证
了解更多

简介

Envoy AI Gateway 可以通过一个兼容 OpenAI 的端点为外部 LLM 提供商提供前置接入。它使用 `BackendSecurityPolicy` 注入每个提供商的上游凭证，按模型名称进行路由，并在提供商之间自动故障切换。消费者只需调用单一的内部地址，且不会持有提供商密钥，因此网关会成为公共云 LLM 流量的受控出口点，并将与自托管模型相同的身份、配额和计量策略应用于外部模型。

使用场景

- 公开托管模型，例如来自 OpenAI、AWS Bedrock、Azure OpenAI、GCP Vertex AI 或 Anthropic 的模型，而无需分发提供商密钥。
- 将不同的模型名称路由到同一端点后的不同提供商。
- 当主提供商不可用时，切换到备用提供商。

前提条件

1. 已安装 Envoy AI Gateway，并且存在一个 `Gateway` 和一个 `AIGatewayRoute`。确认相关 CRD 已存在：

   ```
   kubectl get crd \
     aigatewayroutes.aigateway.envoyproxy.io \
     aiservicebackends.aigateway.envoyproxy.io \
     backendsecuritypolicies.aigateway.envoyproxy.io \
     backends.gateway.envoyproxy.io
   ```

2. 已启用 Envoy Gateway `Backend` API。本文档通过 `kind: Backend`（Envoy Gateway 扩展资源）声明提供商上游，这要求在 EnvoyGateway 配置中设置 `extensionApis.enableBackend: true`——仅仅存在 CRD 还不够。请先确认，如缺失则启用它：

   ```
   kubectl -n envoy-gateway-system get configmap envoy-gateway-config \
     -o jsonpath='{.data.envoy-gateway\.yaml}' | grep -A2 extensionApis
   # expect to see: enableBackend: true
   ```

   如果未设置，请在 `extensionApis` 下添加它并重新启动控制平面：

   ```
   kubectl edit configmap envoy-gateway-config -n envoy-gateway-system
   # extensionApis:
   #   enableBackend: true
   kubectl rollout restart deployment envoy-gateway -n envoy-gateway-syste
   m
   ```

3. 上游提供商凭证（在下一节创建）存储在路由命名空间中的一个 `Secret` 中。

4. 从集群出口可以访问提供商端点。继续之前请先验证：

```
# On clusters without docker.io egress (air-gapped / mirror-only), repl
ace
# curlimages/curl with any small curl/wget image from your registry mir
ror.
kubectl run egress-probe --rm -i --restart=Never \
  --image=curlimages/curl -- \
  curl -s -o /dev/null -w '%{http_code}\n' https://api.openai.com/v1/mo
dels
# expect: 401  (anything other than a connection error means egress wor
ks)
```

NOTE

请在专用命名空间中创建 `Gateway` 和 `AIGatewayRoute`（例如 `maas-system`），不要创建在 Envoy Gateway 控制平面命名空间 `envoy-gateway-system` 中。放置在控制平面命名空间中的网关可能不会将 AI Gateway 请求处理过滤器和 `SecurityPolicy` 应用到其监听器上，这会静默破坏路由和策略执行。另请参阅 Envoy AI Gateway。

步骤

1 存储上游凭证

创建保存提供商 API 密钥的 `Secret`。对于 `type: APIKey`，**data-map key 必须严格为 `apiKey`** —— `BackendSecurityPolicy` 会按名称查找该字段，因此使用 `--from-literal=key=...` 会导致上游调用未通过身份验证，即使策略报告为 `Accepted`：

```
kubectl -n <your-namespace> create secret generic openai-key \
  --from-literal=apiKey="$OPENAI_API_KEY"   # data-map key must be 'a
piKey'
```

使用 `BackendSecurityPolicy` 注入目标为后端的提供商凭证。`type` 字段用于选择提供商的身份验证方案。

```yaml
apiVersion: aigateway.envoyproxy.io/v1alpha1
kind: BackendSecurityPolicy
metadata:
  name: openai-auth
  namespace: <your-namespace>
spec:
  type: APIKey
  apiKey:
    secretRef:
      name: openai-key  # Secret holding the provider API key (data key 'apiKey')
  targetRefs:
    - group: aigateway.envoyproxy.io
      kind: AIServiceBackend
      name: openai-backend
```

`type` 字段接受 `APIKey`、`AWSCredentials`、`AzureAPIKey`、`AzureCredentials`、`GCPCredentials` 和 `AnthropicAPIKey`。每种类型都需要匹配的凭证块和一组匹配的 `Secret` 数据键：

`type`	凭证块	必需的 `Secret`
`APIKey`	`apiKey.secretRef`	`apiKey`
`AWSCredentials`	`awsCredentials.credentialsFile.secretRef`	`credentials`（`region`）
`AzureAPIKey`	`azureAPIKey.secretRef`	`apiKey`
`AzureCredentials`	`azureCredentials.clientSecretRef`	`client-secret`
`GCPCredentials`	`gcpCredentials.credentialsFile.secretRef`	`service_accou` `projectName` / `gcpCredentials` 实现无密钥 WIF）
`AnthropicAPIKey`	`anthropicAPIKey.secretRef`	`apiKey`

当上游身份验证方案错误时，上游通常会返回 `401` / `403`。当 `Secret` 的键名错误时（例如使用 `key:` 而不是 `apiKey:`），故障表现更难以识别：

`BackendSecurityPolicy` 仍会报告 `Accepted=True`，但控制器日志会显示 `failed to get backend auth from backend security policy. Skipping this backend. ... error: secret <name> does not contain key apiKey`，并将该后端从路由中移除，因此发往该后端的请求会超时，而不是返回清晰的 401。引入新凭证时，请跟踪控制器日志：

```
kubectl -n envoy-gateway-system logs deploy/ai-gateway-controller -c
ai-gateway-helm \
  | grep -E 'backend security policy|does not contain key'
```

2 定义提供商后端

有两个资源协同工作：`Backend`（Envoy Gateway）向数据平面说明要连接的网络端点，而 `AIServiceBackend`（Envoy AI Gateway）则向 AI 过滤器说明要转换成的提供商 **schema**。

首先，将上游端点声明为 `Backend`：

```
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: Backend
metadata:
  name: openai-endpoint
  namespace: <your-namespace>
spec:
  endpoints:
    - fqdn:
        hostname: api.openai.com
        port: 443
```

然后引用该 `Backend`，将提供商注册为 `AIServiceBackend`：

```yaml
apiVersion: aigateway.envoyproxy.io/v1alpha1
kind: AIServiceBackend
metadata:
  name: openai-backend
  namespace: <your-namespace>
spec:
  schema:
    name: OpenAI
  backendRef:
    name: openai-endpoint  # Backend pointing at the provider host
    kind: Backend
    group: gateway.envoyproxy.io
```

- `schema.name`：网关必须使用的上游协议。常见值包括：`OpenAI`、`AWSBedrock`、`AzureOpenAI`、`GCPVertexAI`、`Anthropic`。网关会先将传入的兼容 OpenAI 请求转换为该 schema，再转发出去。
- `backendRef`：必须指向 `Backend`（`gateway.envoyproxy.io` 组），而不是 `Service`——AI 过滤器依赖 `Backend` 来处理面向公共端点的 FQDN 和 TLS。

确认这两个资源都已完成协调：

```
kubectl get backend,aiservicebackend -n <your-namespace>
# AIServiceBackend shows ACCEPTED=True immediately;
# the Backend STATUS column stays empty until the next step's AIGatewayRoute
# actually references this AIServiceBackend — Envoy Gateway only reconciles a
# Backend once at least one HTTPRoute targets it.
```

3 按模型路由并进行故障切换

在一个 `AIGatewayRoute` 规则中引用多个后端，并设置 `priority`，以便网关从主后端故障切换到备用后端：

```yaml
apiVersion: aigateway.envoyproxy.io/v1alpha1
kind: AIGatewayRoute
metadata:
  name: <aigatewayroute-name>
  namespace: <your-namespace>
spec:
  # ... parentRefs ...
  rules:
    - matches:
        - headers:
            - name: x-ai-eg-model
              type: Exact
              value: gpt-4o
      backendRefs:
        - name: openai-backend
          priority: 0   # primary
        - name: azure-backend
          priority: 1   # used when the primary is unavailable
```

- `matches.headers[x-ai-eg-model]`：AI 过滤器会从请求体中解析 `model` 字段，并将其写入此 header 以供路由使用。因此，请求中的 `"model":"gpt-4o"` 才会进入该匹配条件——调用方无需手动设置 header。
- `priority`：Envoy 使用优先级级别故障切换。只要 priority-0 组有足够的健康端点，就会承载所有流量；只有当 priority-0 退化时，priority-1 组才会接收流量——这由 Envoy 的 overprovisioning factor 和 panic threshold 决定，而不是严格要求“priority-0 的每个端点都必须宕机”。此 `priority` 字段会覆盖 `Backend` 级别的 `fallback` 设置。故障切换是自动的，但需要几秒钟而不是毫秒级；不要将其用于尾延迟预算。

验证

向网关发送一个兼容 OpenAI 的请求，并确认它能够到达提供商，客户端的 `Authorization` header 已被 BSP 注入的密钥替换，同时返回包含 token 使用量的有效响应：

```
# A deliberately-bogus client Authorization to prove the gateway strips i
t
# and substitutes the upstream key from the BackendSecurityPolicy.
curl -sv http://<gateway-address>/v1/chat/completions \
  -H 'Authorization: Bearer client-token-that-should-be-replaced' \
  -H 'Content-Type: application/json' \
  -d '{"model":"gpt-4o","messages":[{"role":"user","content":"ping"}]}'
```

成功响应（带有 `usage` 对象的 `200 OK`）意味着上游凭证已被注入且路由已解析。要检查实际命中上游的请求，可在 `EnvoyProxy` 资源上启用 Envoy 的访问日志，或者临时路由到一个调试回显后端；上游请求中的 `Authorization` header 应携带来自 `openai-key` `Secret` 的值，而不是上面的伪造客户端值。

要验证故障切换，可以通过将主 `Backend` 指向一个不可达主机（`hostname: invalid.example.invalid`）并维持几秒来模拟主服务中断，然后观察流量切换到备用服务；响应体中的 `model` 字段会反映新的提供商。

了解更多

- 验证消费者身份
- 配置 token 配额

Token 用量分摊计费

目录

简介
使用场景
前提条件
步骤
添加采集配置
添加展示配置
向成本模型添加价格
验证
故障排查
了解更多
下一步

简介

本指南通过遵循平台的 自定义成本模型↗ 工作流，为 AI Gateway token 用量定义一个自定义成本模型。其结构与官方的 vGPU (Hami) 成本模型↗ 指南一致：先在 Cost Management agent 集群上添加采集配置，再在 Cost Management server 集群上添加展示/存储配置，然后在平台控制台中为该成本模型添加价格。

Cost Management 会消费由 Metering Token Usage 收集的 OpenTelemetry GenAI token 指标，并以 `user_namespace` 标签为键，将其转换为按命名空间计费的账单。在多集群部署中，Cost Management server 和 Cost Management agent 可能位于不同集群；下面每一步都会注明其必须执行的集群。

使用场景

- 按命名空间或租户对其消耗的 AI Gateway token 进行计费。
- 按模型设置单价，使更昂贵的模型每个 token 的成本更高。
- 使用与仪表盘相同的指标生成可审计的分摊计费报告。

前提条件

1. 已配置 Metering Token Usage：`PodMonitor` 正在收集 token 指标，且控制器将 `x-user-namespace:user_namespace` 映射到系列上，使其携带 `user_namespace` 标签。请确认可以从 agent 集群的平台 Prometheus（或 Thanos）中查询到该指标：

   ```
   # In Prometheus, this must return a non-empty vector:
   #   sum by (user_namespace) (increase({__name__="gen_ai.client.token.usage_token_sum"}[1h]))
   ```

2. 已安装 Cost Management：server 集群上安装了 `cost-server` 和 `cost-api`，每个需要计费其 AI Gateway 流量的集群上安装了 `cost-agent`。参见 Cost Management 安装↗。
3. 具有访问两个集群的 `kubectl` 权限，并拥有在 `cpaas-system`（agent 集群）和 `kube-public`（server 集群）中写入 `ConfigMap` 的权限。

步骤

1 添加采集配置

集群： **Cost Management agent** 集群——即运行 AI Gateway 且安装了 `cost-agent` 的集群。`cost-agent` 组件以 `slark-agent` 工作负载的形式部署，下面的重启命令会

作用于该工作负载。

创建一个 `ConfigMap`，用于告诉 Cost Management agent 应评估哪个 Prometheus 查询，以及如何将其标签映射到 Cost Management 维度。agent 会发现位于 `cpaas-system` 中、带有 `cpaas.io/slark.collection.config: "true"` 标签的配置。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: slark-agent-aigateway-config
  namespace: cpaas-system
  labels:
    cpaas.io/slark.collection.config: "true"
data:
  config: |
    - kind: AIGateway
      category: AIGatewayToken
      item: AIGatewayTokenUsage
      period: Hourly
      usage:
        query: |
          sum by (user_namespace, "gen_ai.request.model") (
            increase({__name__="gen_ai.client.token.usage_token_sum"}
[5m])
          )
        step: 5m
        mappers:
          name: gen_ai.request.model
          namespace: user_namespace
          cluster: ""
          project: user_namespace
```

字段说明：

- `kind`：Cost Management collector 的类型，同时也是 collector 的名称（在采集配置中必须保持唯一）。`Pod` 保留给 OpenCost 输出的 pod 指标，`Project` 保留给平台内置的 CPU/Memory/Storage 配额 collector。AI Gateway token 用量使用专用的自定义类型 `AIGateway`，与 vGPU/NPU/pGPU 自定义类型模式一致——这是在集群上验证、用于将 `cost.usage` 填充为 `AIGatewayTokenUsage` 行的值。应用此配置后，

请先确认已出现对应行（参见 Troubleshooting 中的 `cost.usage` 检查），再继续下一步。

- `category`、`item`：用于将此采集配置与下一步中的展示/存储配置关联起来的标识符。两个值都必须与展示/存储配置中的相应字段一致。
- `period`：聚合周期。按小时计费请使用 `Hourly`。
- `usage.query`：agent 每个 `step` 执行一次的 PromQL 查询。平台会将该指标以保留点号的 OpenTelemetry UTF-8 名称存储，因此应使用 `{__name__="gen_ai.client.token.usage_token_sum"}` 选择它，并使用带引号的 UTF-8 语法引用带点的 `gen_ai.request.model` 标签。通过 header 映射得到的 `user_namespace` 标签是按命名空间计费的键。
- `usage.step`：查询的评估间隔。
- `usage.mappers`：将 PromQL 标签映射到 Cost Management 的标准维度（`name`、`namespace`、`cluster`、`project`）。将 `cluster` 设为空字符串，这样 agent 会自动填入自己的集群身份；如果将其设置为某个标签名（例如 `cluster`），则只有当源指标暴露了该标签时才会生效，否则每一行都会被丢弃且不会留下日志。

应用 YAML 后，重启 `slark-agent` 工作负载（即 `cost-agent` 组件）以重新加载配置：

```
kubectl -n cpaas-system delete pod \
  -l service_name=slark-agent --grace-period=0 --force
kubectl -n cpaas-system rollout status deploy/slark-agent
# if slark-agent is installed as a DaemonSet in your environment, use
instead:
#   kubectl -n cpaas-system rollout status daemonset/slark-agent
```

2 添加展示配置

集群： **Cost Management server** 集群——即安装了 `cost-server` 的集群（通常是全局控制平面）。`cost-server` 组件以 `slark-server` 工作负载的形式部署，下面的重启命令会作用于该工作负载。

创建一个 `ConfigMap`，在平台控制台中注册计费项及其计费方式。server 会发现位于 `kube-public` 中、带有 `cpaas.io/slark.display.config: "true"` 标签的配置。

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: slark-display-config-for-aigateway
  namespace: kube-public
  labels:
    cpaas.io/slark.display.config: "true"
data:
  config: |
    - name: AIGatewayToken
      displayname:
        zh: "AI Token"
        en: "AI tokens"
      methods:
        - name: Usage
          displayname:
            zh: "使用量"
            en: "Token Usage"
          item: AIGatewayTokenUsage
          unit:
            zh: "tokens"
            en: "tokens"
          divisor: 1
```

字段说明：

- `name`：在成本模型表单中显示的计费项名称。必须与采集配置中的 `category` 值一致。
- `methods[].name`：计费方式，在添加价格时会显示在计费项下方。
- `methods[].item`：必须与采集配置中的 `item` 值一致，以便 server 能将按方式设置的单价与用量行关联起来。
- `divisor`：用于显示用量时的单位换算因子。token 没有单位，因此设置为 `1`；对于字节级项目，使用 `1073741824` 以显示为 `Gi-hours`。

下表总结了此配置注册的计费方式：

计费项	计费方式	来源项	说明
AI tokens	Token Usage	`AIGatewayTokenUsage`	每小时每个命名空间和模型消耗的输出 + 输入 token

> **WARNING**
>
> 请勿修改平台预置的 `slark-server-common-config`（包含 CPU、Memory 和 Storage 的默认展示配置）。向其中添加自定义条目会导致 server 在启动时校验失败。请始终将自定义计费项作为单独的、带有 `cpaas.io/slark.display.config: "true"` 标签的 `ConfigMap` 添加。

应用 YAML 后，重启 `slark-server` 工作负载（即 `cost-server` 组件）以重新加载配置：

```
kubectl -n cpaas-system delete pod \
  -l service_name=slark-server --grace-period=0 --force
kubectl -n cpaas-system rollout status deploy/slark-server
```

3 向成本模型添加价格

集群： 任意集群——通过由 `cost-api` 提供服务的平台控制台进行操作。

在平台控制台中，进入 **Administrator → Metering and Billing → Cost Model**，然后创建或编辑一个成本模型。新注册的 `AI tokens` 计费项现在可以在价格表单中选择。

1. **Cost Model Name**：任意标识符，例如 `aml-cost-model`。
2. **Linked Clusters**：选择该模型应对其 AI Gateway 流量计费的集群。即使留空也能成功保存，但不会匹配任何用量数据，也不会生成账单。
3. **Pricing rows**：
   - **Billing Item**：`AI tokens`
   - **Billing Method**：`Token Usage`
   - **Default Price**：按 token 计费的价格，使用平台币种。

> - **Price By Label**（可选）：按模型覆盖价格，例如对 `gen_ai.request.model="gpt-4o"` 设置更高费率。
>
> 4. 保存。

验证

cost-server worker 每五分钟运行一次。请通过 AI Gateway 发起几次已认证请求，并使用不同的 `x-user-namespace` 值，至少等待一个 worker 周期后刷新平台控制台。

- **Cost Details**（**Administrator** → **Metering and Billing** → **Cost Details**）会显示按命名空间划分的 AI tokens 明细。可按命名空间或日期筛选以进一步查看。
- **Cost Statistics**（**Administrator** → **Metering and Billing** → **Cost Statistics**）会按集群、项目和时间范围汇总相同数据。

如需在打开 UI 之前进行服务端验证，可在 server 集群上查询 ClickHouse：

```
SELECT namespace, project, date, usage, cost
  FROM cost.bills
 WHERE item = 'AIGatewayTokenUsage'
 ORDER BY date DESC, cost DESC
 LIMIT 10;
```

预期结果是每个消耗 token 的（命名空间，小时）组合都有一行。`cost` 列存储的是平台的微货币单位，因此 `usage × default_price × 1_000_000` 应与显示值一致。

故障排查

症状	可能原因与解决方法
一个 agent 周期后，`cost.usage` 中仍没有 `AIGatewayTokenUsage` 行	确认采集 `ConfigMap` 位于 `cpaas-system` 中，并带有 `cpaas.io/slark.collection.config: "true"`。确认在 apply 之后已重启 `cost-agent`。确认 PromQL 在平台 Prometheus 上返回非空向量——使用下划线形式的指标名（`gen_ai_client_token_usage_token_sum`）不会匹配任何内容；请使用

症状	可能原因与解决方法
	`{__name__="gen_ai.client.token.usage_token_sum"}` 选择器。确认 `mappers.cluster: ""` 确实是空字符串。
`AI tokens` 计费项没有出现在成本模型表单中	确认展示/存储 `ConfigMap` 位于 `kube-public`（而不是 `cpaas-system`）中，并带有 `cpaas.io/slark.display.config: "true"`。确认在 apply 之后已重启 `cost-server`。确认 `methods[].item` 与采集配置中的 `item` 完全一致。
`cost.usage` 有行，但 `cost.bills` 中没有 `AIGatewayTokenUsage` 对应记录	在平台控制台中打开成本模型，确认 **Linked Clusters** 下至少选择了一个集群。worker 还会跳过在 `cost.milestones` 中已标记为 `Done` 的分组；如果要在模型变更后重新计算过去时间窗，请删除 `cost.milestones` 中相关行并等待下一个 worker 周期。
每次 tick 的 worker 日志都显示 `Try to skip as no model {"group": "<cluster>/<window>"}`	成本模型的 `Linked Clusters` 不包含 agent 报告的集群。编辑该模型并添加该集群。

了解更多

- Metering Token Usage
- Authenticating Consumers

下一步

在生成账单后，可在 **Price By Label** 下设置按模型的价格覆盖，以反映各模型的相对成本，并定期导出 `cost.bills` 用于财务报表。

LLM 压缩器

介绍

前言

LLM Compressor 支持多种压缩技术：

LLM Compressor 支持多种压缩算法：

在 Alauda AI 中使用 LLM Compressor

支持的模型压缩工作流

介绍

目录

前言

LLM Compressor 支持多种压缩技术：

LLM Compressor 支持多种压缩算法：

前言

LLM Compressor↗，作为 vLLM 项目↗ 中面向高效提供 LLM 服务的一部分，将最新的模型压缩研究整合到一个统一的开源库中，使用户能够以极低的工作量生成高效的压缩模型。

该框架使用户能够应用最新的模型压缩技术研究来提升生成式 AI（gen AI）模型的效率、可扩展性和性能，同时保持准确性。借助对 Hugging Face 和 vLLM 的原生支持，压缩后的模型可以集成到部署流水线中，从而以更低成本在大规模场景下提供更快的推理。

LLM Compressor 允许你执行量化、稀疏化和压缩等模型优化技术，以减少内存占用、模型大小，并在不影响模型响应准确性的前提下提升推理性能。LLM Compressor 支持以下压缩方法：

- 量化：将模型权重和激活值转换为更低比特位格式，例如 int8，从而减少内存使用。
- 稀疏化：将模型权重的一部分置为零，通常采用固定模式，以实现更高效的计算。
- 压缩：缩小保存的模型文件大小，理想情况下对性能的影响最小。

将这些方法结合使用，可以在资源受限的硬件上更高效地部署模型。

LLM Compressor 支持多种压缩技术：

- 仅权重量化（W4A16）将模型权重压缩到 4 位精度，适用于硬件资源有限或对延迟高度敏感的 AI 应用。
- 权重和激活量化（W8A8）将权重和激活值都压缩到 8 位精度，面向整数和浮点格式的通用服务器场景。

LLM Compressor 支持多种压缩算法：

- AWQ：仅权重 `INT4` 量化
- GPTQ：仅权重 `INT4` 量化
- FP8：按 token 动态量化
- SparseGPT：训练后稀疏化
- SmoothQuant：激活量化

有关压缩算法和格式的更多信息，请参阅 文档↗ 以及 llmcompressor↗ 仓库中的示例。 这些压缩方法都会为权重和激活值计算最优的 scale 和 zero-point。优化后的 scale 可以按 tensor、channel、group 或 token 进行设置。最终结果是一个已应用所有量化参数的压缩模型。

在 Alauda AI 中使用 LLM Compressor

本文档介绍如何结合使用 LLM Compressor 与 Alauda AI 平台来执行模型压缩工作流。LLM Compressor 的 Alauda AI 集成提供以下两个示例工作流：

- 一个 Workbench 镜像和 data-free-compressor.ipynb，用于演示如何压缩模型。
- 一个 Workbench 镜像和 calibration-compressor.ipynb，用于演示如何使用校准数据集压缩模型。

notebook

目录

支持的模型压缩工作流

创建 Workbench

准备模型制品

(可选) 准备并上传数据集

(可选) 将数据集上传到 S3 兼容的对象存储

(可选) 使用 S3 兼容对象存储中的数据集

在 JupyterLab 中下载模型和数据集

创建并运行压缩 Notebook

注册压缩后的模型

部署压缩模型并将其用于推理

支持的模型压缩工作流

在 Alauda AI 平台上，您可以使用 Workbench 功能，对存储在 S3、PVC 或 OCI registry 中的模型运行 LLM Compressor。以下工作流概述了压缩模型的典型步骤。

创建 **Workbench**

按照创建 Workbench 中的说明创建新的 Workbench 实例。创建 Workbench 时，请务必选择镜像 `odh-workbench-jupyter-pytorch-llmcompressor-cuda-py312-ubi9`，其中包含所需的 LLM Compressor 环境。目前仅支持在 **JupyterLab** 中进行模型压缩。

如果您的环境中没有此镜像，请参阅链接文章，了解如何同步该镜像并修补 WorkspaceKind 资源，使该镜像显示在 Workbench 创建选项中。

准备模型制品

将模型下载到 Workbench，或将其暂存到 Workbench 可访问的 PVC/S3 位置。压缩后，将输出上传到受支持的存储后端。请参阅 使用 Workbench 上传模型。 使用 Model Registry 将生成的 Model URI 注册为新的 Model Registry 版本。 本指南中的示例 notebook 使用 TinyLlama-1.1B-Chat-v1.0↗ 模型。

data-free-compressor.ipynb

```
from llmcompressor.modifiers.quantization import QuantizationModifier

model_id = "./TinyLlama-1.1B-Chat-v1.0" 1
recipe = QuantizationModifier(targets="Linear", scheme="W4A16", ignore=
["lm_head"]) 2
```

1 要压缩的模型。如果要使用自己的模型，可以修改此行。2. 此 recipe 将量化除 `lm_head` 中的层之外的所有 Linear 层，因为该层通常对量化较为敏感。 `W4A16` 方案 将权重压缩为 `4-bit` 整数，同时保留 `16-bit` 激活值。

(可选) 准备并上传数据集

> **NOTE**
>
> 如果您计划使用 **data-free compressor notebook**，可以跳过此步骤。

要使用 **calibration compressor notebook**，必须准备并上传校准数据集。请按照 *使用 Notebook 上传模型* 中所述的相同流程准备数据集。示例校准 notebook 使用 ultrachat_200k 数据集。

calibration-compressor.ipynb

```
from datasets import load_dataset

dataset_id = "./ultrachat_200k" 1

num_calibration_samples = 512 if use_gpu else 4 2
max_sequence_length = 2048 if use_gpu else 16

ds = load_dataset(dataset_id, split="train_sft") 3
ds = ds.shuffle(seed=42).select(range(num_calibration_samples)) 4

def preprocess(example): 5
    text = tokenizer.apply_chat_template(
        example["messages"],
        tokenize=False,
    )
    return tokenizer(
        text,
        padding=False,
        max_length=max_sequence_length,
        truncation=True,
        add_special_tokens=False,
    )

ds = ds.map(preprocess, remove_columns=ds.column_names)
```

1 使用 Huggingface datasets API 创建校准数据集。如果要使用自己的数据集，可以 修改此行。2. 选择样本数量。建议从 512 个样本开始。增加 样本数量可以提高准确性。3. 加载数据集。4. 将数据集打乱，并仅获取 所需数量的样本。5. 预处理并 tokenization 为模型 使用的格式。

(可选) 将数据集上传到 S3 兼容的对象存储

如果要将数据集上传到 S3 兼容的对象存储，可以在 `JupyterLab` 中运行以下代码。Alauda AI 支持访问 S3 兼容的存储；在典型的产品部署中，存储实现为 Ceph 对象存储，因此可以使用标准 `boto3` 客户端。

```
import os
from boto3.s3.transfer import TransferConfig
import boto3

local_folder = "./ultrachat_200k" 1
bucket_name = "datasets"

config = TransferConfig(
    multipart_threshold=100*1024*1024,
    max_concurrency=10,
    multipart_chunksize=100*1024*1024,
    use_threads=True
) 2

for root, dirs, files in os.walk(local_folder):
    for filename in files:
        local_path = os.path.join(root, filename)
        relative_path = os.path.relpath(local_path, local_folder)
        s3_key = f"ultrachat_200k/{relative_path.replace(os.sep, '/')}"
        s3.upload_file(local_path, bucket_name, s3_key, Config=config)
        print(f"Uploaded {local_path} -> {s3_key}")
```

1 如果要使用自己的数据集，可以修改此行。2. 配置分段上传，分段大小为 100 MB，最大并发 线程数为 10。

(可选) 使用 S3 兼容对象存储中的数据集

如果要使用存储在 S3 兼容对象存储中的数据集，请先安装 `s3fs` 工具，然后按如下所示修改示例中的数据集加载部分。在 Alauda AI 环境中，此 S3 兼容存储通常由 Ceph 对象存储提供支持。

```
pip install s3fs -i https://pypi.tuna.tsinghua.edu.cn/simple
```

calibration-compressor.ipynb

```
import os
from datasets import load_dataset

os.environ["AWS_ACCESS_KEY_ID"] = "<access-key>" (1)
os.environ["AWS_SECRET_ACCESS_KEY"] = "<secret-key>"

storage_options = {
  "key": "<access-key>",
  "secret": "<secret-key>",
  "client_kwargs": {
    "endpoint_url": "https://ceph-obj.example.com" (2)
  }
}

ds = load_dataset(
      'parquet',
      data_files='s3://datasets/ultrachat_200k/data/train_sft-*.parquet',
(3)
      storage_options=storage_options, (4)
      split="train"
)
```

(1) 设置环境变量（作为备用，某些底层组件会使用这些变量）。将 `<access-key>` 和 `<secret-key>` 替换为对象存储账户的凭证。

(2) 定义存储配置；必须为 S3 兼容对象存储服务显式指定 `endpoint_url`，例如 Ceph 对象存储端点。

(3) 如果数据集已拆分，这等同于示例中的 `split="train_sft"`。

在 JupyterLab 中下载模型和数据集

在 JupyterLab 终端中，使用 `hf` 命令行工具或获准使用的 对象存储客户端，将模型和数据集下载到 workspace 中。data-free compressor notebook 不需要数据集。

INFO

如果网络访问条件允许，建议使用 Hugging Face (hf) 命令行工具直接下载模型和数据集。与手动克隆存储库相比，这种方法通常更快速、更简单。

如果您处于网络受限的环境中，可以使用 Hugging Face 镜像来加速访问：

```
export HF_ENDPOINT=https://hf-mirror.com
```

您可以使用 `hf` 命令行工具直接下载模型和数据集。例如，要下载 TinyLlama 模型：

```
hf download TinyLlama/TinyLlama-1.1B-Chat-v1.0 --local-dir TinyLlama-1.1B
-Chat-v1.0
```

对于校准数据集，可用类似方式下载：

```
hf download --repo-type dataset HuggingFaceH4/ultrachat_200k --local-dir
ultrachat_200k
```

创建并运行压缩 **Notebook**

下载适合您使用场景的示例 notebook：如果使用数据集，请下载 **calibration compressor notebook**；否则，请下载 **data-free compressor notebook**。单击 JupyterLab 页面上的向上箭头按钮，上传已下载的 notebook 文件。

注册压缩后的模型

压缩完成后，使用使用 Workbench 上传模型 将压缩后的模型上传到 S3、PVC 或 OCI registry。 然后使用 Model Registry 将其 Model URI 注册为新的 Model Registry 版本。

```
model_dir = "./" + model_id.split("/")[-1] + "-W4A16" (1)
model.save_pretrained(model_dir)
tokenizer.save_pretrained(model_dir);
```

(1) 保存模型和 tokenizer。如果要更改输出的名称，可以修改 此行。

部署压缩模型并将其用于推理

使用 LLM Compressor 创建的量化模型和稀疏模型会通过 `compressed-tensors` 库（Safetensors↗ 的扩展）保存。 压缩格式与模型的量化或稀疏类型相匹配。vLLM 原生支持这些格式，因此可以使用 Alauda AI Inference Server，通过优化的部署 kernel 实现快速推理。

按照创建推理服务中的说明完成此步骤。

Model as a Service

介绍

- 为什么使用 MaaS？
- 请求流程
- 主要概念
- 选择指南
- 相关主题

MaaS 管理员指南

- 发布平台推理服务
- 导入外部模型服务
- 创建订阅
- 配置 guardrails
- 查看使用情况和报告

MaaS 用户指

- 打开我的订阅
- 查找你的模型和
- 创建 API key
- 调用模型
- 理解配额行为
- 查看个人使用情
- 理解请求错误

MaaS 组件扩缩容

- Gateway
- Authorino
- rate limiter
- Guardrail engine 和 shim
- 在哪里配置副本和资源

常见问题

- 转发到使用非默认端口的外部模型服务时，请求可能失败

有关包含 MaaS 的负载测试结果和容量规划指南，请参阅 Deployment Capacity Planning。

介绍

Model as a Service (MaaS) 提供了一个托管网关，用于通过与 OpenAI 兼容的 API 提供大型语言模型服务。它为平台团队提供了一种一致的方式来发布模型、授予访问权限、强制执行 token 配额并了解使用情况；同时，应用开发人员可以使用 API key 调用其已订阅的模型。

MaaS 基于 **Alauda Build of Envoy Gateway**、**Alauda Build of Envoy AI Gateway** 和 **Authorino**。Envoy Gateway 提供 Kubernetes Gateway API 基础，Envoy AI Gateway 提供感知模型的路由和 token 计数，Authorino 负责对请求进行身份验证。Alauda AI 通过 `default` `AmlCluster` 的 `envoyGateway`、`envoyAIGateway` 和 `authorino` 组件安装并管理这些组件；请参见 安装 Alauda AI。

目录

为什么使用 MaaS？

请求流程

主要概念

选择指南

相关主题

为什么使用 **MaaS**？

MaaS 将模型服务与模型消费分离：

- 模型所有者可以发布一个推理服务，而无需暴露其内部路由细节。
- 管理员可以将模型组合成订阅并分配给用户。
- 开发人员使用一个与 OpenAI 兼容的网关，而不必学习每个后端的 URL 和身份验证方式。
- 平台团队可以按模型强制执行 token 配额，并按用户、订阅、模型和 token 类型查看消耗情况。

MaaS 可以通过同一个网关同时暴露平台托管的推理服务和外部与 OpenAI 兼容的模型提供方。

请求流程

对于每个请求，MaaS 会执行以下检查：

1. 调用方提供 API key 并通过身份验证。
2. 调用方通过有效订阅拥有对所请求模型的访问权限。
3. 该订阅对该用户和模型仍有剩余的 token 配额。
4. 网关将请求路由到所选模型，并记录模型服务返回的实际 token 使用量。

未经身份验证或未授权的请求会返回 `401` 或 `403`。超出订阅配额的请求会返回 `429`。

配额按用户强制执行，而不是按 API key 强制执行。属于同一用户的多个 API key 会共享该用户在某个订阅和模型上的额度。

主要概念

概念	作用
模型	已发布的模型，可由订阅选择并通过 MaaS 网关调用。
模型来源	提供一个或多个模型的平台推理服务或外部模型服务。
订阅	访问和配额单位。它将用户绑定到模型，并为每个模型定义 token 限额。
API key	用于调用已订阅模型的开发人员凭证。明文 key 仅在创建时显示一次。

概念	作用
使用情况	管理员和开发人员可见的 token 和请求消耗情况，其中开发人员仅能查看自己的 API key。
Guardrails	可选的内容检查，可附加到订阅中的模型。

选择指南

- MaaS 管理员指南 — 发布和导入模型，创建订阅，配置配额和 Guardrails，并查看报告。
- MaaS 用户指南 — 创建 API key，调用已订阅模型，并查看个人配额和使用情况。
- MaaS 组件扩缩容 — 扩缩容 Gateway、Authorino、限流器以及 Guardrail engine 和 shim。
- MaaS 常见问题 — 排查转发到非默认端口上的外部模型服务的问题。

相关主题

- Inference Service — 创建可发布到 MaaS 的 `LLMInferenceService` 模型。
- Authenticating Consumers — 对调用方进行身份验证，并将身份信息传递给网关策略。
- Configuring Token Quotas — 在推理网关处配置基于 token 的配额强制执行。
- Metering Token Usage — 了解 token 消耗如何被报告。
- Routing to LLM Providers — 通过受控网关路由外部提供方。

MaaS 管理员指南

本指南说明平台管理员如何管理 Model as a Service (MaaS)：发布模型服务、创建订阅、强制执行 token 配额、配置可选的 guardrails，以及查看使用情况。

有关产品概览，请参见 Model as a Service (MaaS)。有关开发人员工作流，请参见 MaaS 用户指南。

目录

- 发布平台推理服务
- 导入外部模型服务
  - 从外部服务发布模型
- 创建订阅
- 配置 guardrails
  - 启用 guardrails
  - 创建 guardrail 配置
  - 将 guardrails 附加到订阅模型
- 查看使用情况和报告
  - 订阅管理使用情况
  - 使用情况报告

发布平台推理服务

模型所有者可以从服务的管理页面发布一个由平台托管的推理服务。该服务必须是 `LLMInferenceService`。

1. 打开包含该推理服务的项目或命名空间。
2. 打开服务管理页面。
3. 在服务操作中，选择 **Publish to MaaS**。
4. 输入可选的显示名称和描述。
5. 单击 **Publish**。

发布后会创建一个 MaaS 模型引用。该模型随后会显示在管理员的 **MaaS → Models → Published Models** 列表中，并可添加到订阅中。

取消发布会将模型从 MaaS 目录中移除。引用该模型的现有订阅将不再能解析到它，因此请在取消发布之前，从这些订阅中移除或替换该模型。

导入外部模型服务

当模型托管在平台外部，或者不是由 `LLMInferenceService` 表示时，请使用外部模型服务。

1. 在 **Administrator** 视图中，打开 **MaaS → Models**。
2. 在 **Model Services** 选项卡上，单击 **Import External Model Service**。
3. 输入该服务的小写名称。
4. 输入提供商的 OpenAI 兼容端点，并在需要时包含其 API base path。例如，当提供商要求 `/v1` 请求时，请使用 `https://host/v1`。
5. 如果提供商需要，请输入上游 API key。若端点不需要凭证，则留空。
6. 单击 **Create**。

API key 会存储在 Kubernetes Secret 中，并由网关在调用提供商时注入。消费者的 MaaS API key 与提供商的上游 key 是分开的凭证：开发人员只能获取 MaaS key。

从外部服务发布模型

导入服务后：

1. 在 **Model Services** 选项卡上，打开该服务的操作菜单。
2. 选择 **Publish Models**。
3. 添加、重命名或移除该服务公开的模型名称。
4. 单击 **Save**。

每个已发布的模型都会成为一个可订阅的独立模型。因此，一个外部服务可以公开多个模型名称，而这些模型可以使用相同的端点和上游凭证。

Published Models 选项卡会显示模型名称、命名空间、引用目标和就绪阶段。使用类型筛选器可区分 **External** 模型和由 **Inference Service** 支持的模型。

创建订阅

订阅是模型访问和 token 配额的单位。用户在调用某个模型之前，必须是包含该模型的订阅的订阅者。

1. 在 **Administrator** 视图中，打开 **MaaS** → **Subscriptions**。
2. 单击 **Create Subscription**。
3. 输入订阅名称。
4. 按用户名添加一个或多个订阅者。
5. 添加一个或多个已发布模型。
6. 为每个模型配置至少一个 token 限制和时间窗口。
7. 可选地为同一模型添加更多配额层级。
8. 单击 **Save**。

> **INFO**
>
> 只有当用户被包含为订阅者时，他们才可以在 **My Subscriptions** 中看到该订阅。将每个用户的用户名添加到 **Subscribers** 中，即可授予该用户访问权限。若要让所有已认证用户都可使用该订阅，请将 `*` 添加为订阅者。`*` 值是通配符，而不是字面量用户名；用户仍然需要有效的 MaaS API key 才能调用这些模型。

对于每个模型，表单接受以百万 token 计的限制以及一个时间窗口，例如 **Minute**、**Hour** 或 **Day**。例如，您可以设置每小时 `100M` token 和每天 `1000M` token。所有已配置的层级都会被强制执行。

一个订阅可以包含多个模型，并且每个模型都可以具有不同的限制。这使您无需为每个模型单独创建网关，就能提供不同的访问层级。

订阅列表会显示订阅名称、模型、订阅者和当前阶段。使用行操作菜单可以查看、编辑或删除订阅。

配置 guardrails

MaaS 可选地使用已配置的内容 guardrails 来检查模型请求和响应。

启用 guardrails

1. 打开 **MaaS** → **Guardrails**。
2. 单击 **Edit settings**。
3. 启用 **guardrails**。
4. 选择失败策略：
   - **End request on guardrails failure** (`FailClosed`) 会在 guardrails 引擎无法完成检查时拒绝或结束请求。
   - **Ignore guardrails on failure** (`FailOpen`) 会在 guardrails 引擎失败时允许请求继续。
5. 保存设置。

禁用 guardrails 会移除数据平面集成。即使订阅上仍保留 guardrail 设置，在全局功能被禁用时也不会生效。

创建 guardrail 配置

1. 在 **Guardrails** 页面上，单击 **Create Config**。
2. 使用小写字母、数字和连字符输入配置 ID。
3. 配置 guardrail 模型，包括模型类型、模型名称和 base URL。

4. 为 guardrail 模型添加可选的 API key。凭证会存储为 Kubernetes Secrets，并通过环境变量暴露给 guardrails 引擎。
5. 编辑 `config.yaml` 内容。
6. 当配置需要 Colang flows 时，编辑 `rails.co` 内容。
7. 单击 **Save**。

一个配置可以包含多个 guardrail 模型，但在同一配置中，每种模型类型必须唯一。**Guardrails** 列表会显示某个配置是否已附加到订阅。

将 **guardrails** 附加到订阅模型

在创建或编辑订阅时，为某个模型启用 guardrails，并选择一个可用的 guardrail 配置。该设置按模型应用，因此同一订阅中的不同模型可以使用不同的配置，或者不使用 guardrails。

查看使用情况和报告

订阅管理使用情况

Subscriptions 页面会显示订阅及其已配置的限制。在可用时，使用订阅的使用情况视图来查看所选订阅和模型的 token 消耗。

使用情况报告

打开 **MaaS** → **Report**，查看管理员级别的使用情况：

1. 选择时间范围：**Last 1 day**、**Last 7 days**、**Last 30 days**、**Last 90 days** 或 **Last 1 year**。
2. 可选地选择一个订阅。
3. 按用户名搜索。
4. 单击刷新以重新加载数据。

报告显示以下内容：

- 用户名
- 订阅

- 总 token 数
- 输入 token
- 缓存 token
- 输出 token
- 推理 token

单击 **Export CSV** 可导出筛选后的报告。导出对话框允许您按订阅、用户、API key 或模型对数据分组，并按天或按月对时间分组。

使用数据旨在提供运行可见性、配额观察和 showback。它不是计费结算记录。

MaaS 用户指南

本指南介绍开发人员和应用用户如何通过 Model as a Service (MaaS) 调用模型。你将了解如何创建 API key、查找可用模型、发送与 OpenAI 兼容的请求，以及查看自己的配额和使用情况。

有关产品概述，请参阅 Model as a Service (MaaS)。有关管理任务，请参阅 MaaS 管理员指南。

目录

打开我的订阅
查找你的模型和端点
创建 API key
调用模型
理解配额行为
查看个人使用情况
理解请求错误
安全建议

打开我的订阅

1. 打开 Alauda AI 控制台右上角的账户菜单。
2. 选择 **My Subscriptions**。

MaaS 会在当前页面上方以侧边面板的形式打开。它关联的是你的用户身份，而不是某个项目命名空间，因此你可以从任何视图中打开它。

该面板包含两个选项卡：

- **API Keys** — 创建和管理你的 MaaS 凭证，并查看访问示例。
- **Usage** — 查看某个订阅的配额状态和个人使用情况。

查找你的模型和端点

在 **API Keys** 选项卡中，**Access** 部分列出了当前可用的模型。每一行显示模型名称和端点。

选择某个模型可更新示例请求。使用端点或示例请求旁边的复制控件将内容复制到剪贴板。

端点通常具有以下形式：

```
https://<maas-gateway>/v1
```

在请求中，将访问列表中显示的模型名称用作 `model` 值。除非底层 Kubernetes 资源名称同时也是已发布的模型名称，否则不要使用它。

创建 API key

1. 在 **API Keys** 选项卡中，单击 **Create API Key**。
2. 输入一个具有描述性的 key 名称，例如 `my-chatbot-dev`。
3. 选择要使用的订阅。
4. 选择有效期。根据平台策略，你可以选择预设时长，例如 30 天或 90 天，自定义时长，或永久 key。
5. 单击 **Create**。
6. 复制明文 key 并妥善保存。

明文 key 仅显示一次，之后无法再检索。该对话框还会提供一个可直接运行的请求，其中已填入新的 key。请像对待密码一样对待该 key：

- 不要将其提交到源代码管理。
- 不要将其包含在浏览器代码或公开 notebook 中。
- 将其存储在 secret manager 或环境变量中。
- 在可能的情况下，为不同应用分别创建独立的 key。
- 如果 key 可能已泄露，请立即吊销。

key 列表会显示 key 名称、状态、订阅、创建时间和最后使用时间。key 可能具有以下状态之一：

- `active` — 可用于调用已订阅的模型。
- `expired` — 有效期已结束。
- `revoked` — 已被禁用，无法使用。

你可以吊销一个处于活动状态的 key。吊销后，你可以从列表中将其删除。

调用模型

MaaS 提供与 OpenAI 兼容的 API。将网关 URL 和 API key 设置为环境变量：

```
export MAAS_ENDPOINT="https://<maas-gateway>/v1"
export MAAS_API_KEY="<your-api-key>"
```

使用 `curl` 发送聊天补全请求：

```
curl "${MAAS_ENDPOINT}/chat/completions" \
  -H "Authorization: Bearer ${MAAS_API_KEY}" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "<published-model-name>",
    "messages": [
      {"role": "user", "content": "Hello"}
    ]
  }'
```

你也可以使用与 OpenAI 兼容的 Python client：

```python
from openai import OpenAI

client = OpenAI(
    base_url="https://<maas-gateway>/v1",
    api_key="<your-api-key>",
)

response = client.chat.completions.create(
    model="<published-model-name>",
    messages=[{"role": "user", "content": "Hello"}],
)

print(response.choices[0].message.content)
```

将 `<published-model-name>` 替换为 **Access** 列表中显示的模型名称。MaaS 网关会选择与该已发布模型名称关联的后端；你的应用无需知道后端服务 URL。

理解配额行为

MaaS 配额由管理员针对你订阅中的每个模型进行配置。一个订阅可以包含多个时间窗口，例如按小时限制和按天限制。

配额消耗按用户、订阅和模型分别跟踪。你的 API key 共享同一个用户配额：

- 创建第二个 API key 不会重置或增加你的配额。
- 使用不同 key 发出的请求会计入同一用户额度。
- 其他用户拥有独立的配额桶。

当某个配额窗口接近上限时，请降低请求量，或请管理员检查该订阅。当配额耗尽时，网关会返回 `429 Too Many Requests`。配额面板会显示已配置的限制、当前使用量（如可用）以及重置时间。

查看个人使用情况

在 **My Subscriptions** 中打开 **Usage** 选项卡：

1. 选择一个订阅。
2. 查看按模型划分的配额条形图。
3. 选择时间范围：**Last 24 hours**、**Last 7 days** 或 **Last 30 days**。
4. 可选：按模型或 API key 过滤图表。
5. 单击刷新以加载最新数据。

使用摘要包括请求数、总 tokens 和缓存 tokens。图表显示随时间变化的 token 序列，包括未缓存输入、缓存输入以及输出 tokens，前提是后端提供这些值。

这是一个个人视图：它只包含使用属于你的 API key 发出的请求。它不会显示其他用户的使用情况，即使你在平台的其他部分是管理员也不例外。

理解请求错误

状态	含义	处理方式
`401 Unauthorized`	API key 缺失、无效、已过期或已吊销。	检查 `Authorization` 请求头，必要时创建新的 key。
`403 Forbidden`	你没有访问所请求模型的权限，或者没有适用的订阅。	确认模型名称并联系你的 MaaS 管理员。
`429 Too Many Requests`	用户、模型或时间窗口的订阅配额已耗尽。	等待配额重置，或请管理员检查该限制。
`5xx`	网关或模型后端发生错误。	在适当情况下重试，并在问题持续存在时联系平台管理员。

当某个模型的 inference service 尚未就绪时，它也可能会在访问列表中可见，但暂时不可用。如果在服务应已就绪后请求仍持续失败，请将模型名称和请求时间提供给管理员。

安全建议

- 将 API key 远离源代码、Git 仓库、shell 历史记录和客户端应用。
- 使用满足需求的最短有效期。

- 为每个应用或环境使用一个 key，这样在某个 key 泄露时，可以将其吊销而不影响无关应用。
- 在 **My Subscriptions** 中吊销未使用或已暴露的 key。
- 切勿将 MaaS API key 与外部提供商共享。MaaS 会在网关侧处理上游提供商凭证。

MaaS 组件扩缩容

本指南说明如何扩展构成 Model as a Service (MaaS) 的各个组件：Gateway、Authorino、rate limiter，以及 Guardrail engine 和 shim。本指南基于长流 SSE 压测以及对组件数据平面行为的观测结果。

目录

Gateway

Authorino

rate limiter

Guardrail engine 和 shim

在哪里配置副本和资源

Gateway

Gateway 是一个受 CPU 约束的数据平面组件。其 CPU 利用率完全可以保持在较高水平：在 SSE 压测中，只要副本数和资源充足，较高的 CPU 利用率本身不会导致请求失败。应当预期并容忍较高的 CPU 使用率，并配置告警，避免产生误报。

当 Gateway 负载很高时，可通过增加副本数或 CPU 进行扩容。一个合理的初始告警点是：持续较高的 CPU 利用率同时伴随 SSE 行为退化，因此在决定扩容前，应将 CPU 与以下信号结合起来判断：

- HTTP 错误和非 `200` 响应；
- 未以 `[DONE]` 结束的 SSE 流；
- TTFT 和总流式时长增加；
- CPU throttling 和 Pod 重启。

Authorino

Authorino 对延迟非常敏感。其瓶颈不在于 CPU 利用率是否达到 100%，而在于 `ext_authz` 检查是否能在超时时间内完成。当 Authorino 排队请求或处理过慢时，即使 Authorino Pod 仍有 CPU 余量，Envoy 也可能因 `ext_authz` 超时返回 `403`。此场景下，典型的 Envoy 响应标志是 `UAEX`。

不要仅根据 CPU 来为 Authorino 设定规模。应监控授权延迟、请求排队情况和 `ext_authz` 超时，并在延迟或超时率上升时通过增加副本数进行横向扩展。

rate limiter

rate limiter 用于强制执行每个订阅的 token 配额。应根据实际请求速率和所使用的限速策略来配置其副本数和资源，并在预期流量模式下进行验证。

Guardrail engine 和 shim

Guardrail engine 使用 Python 实现。单个 engine 实例无法很好地利用多个 CPU 核心：在 SSE 压测中，分配了 1 CPU 的 engine Pod 并未被完全打满，而且为单个实例增加更多核心也没有提升并发吞吐量。经过验证的 Guardrail engine 扩展方式是增加 engine 副本数，同时将每个 engine Pod 保持为 1 CPU。

应根据 `ext_proc` 调用延迟、错误率和并发数来扩展 Guardrail `ext_proc` shim。

在哪里配置副本和资源

大多数 MaaS 组件都在 MaaS `Tenant` 上进行配置：

- Gateway：`spec.gatewayDeployment`
- Authorino：`spec.authorinoDeployment`
- Guardrail engine 和 shim：`spec.guardrails.engine` 和 `spec.guardrails.shim`

全局 rate limiter（`envoy-ratelimit`）是个例外。它在安装 MaaS 的 `AmlCluster` 上进行配置，路径为 `spec.values.maas.gateway.rateLimit`。

`engine.replicas` 和 `shim.replicas` 字段控制 Guardrail engine 和 shim 的副本数。对于 Guardrail engine，请将每个 Pod 保持为 1 CPU，并通过增加 `engine.replicas` 来扩容。

示例 `Tenant`：

```yaml
apiVersion: maas.alauda.io/v1alpha1
kind: Tenant
metadata:
  name: default-tenant
  namespace: aml-maas
spec:
  gatewayDeployment:
    replicas: 2
    resources:
      requests: {cpu: "1", memory: 512Mi}
      limits: {cpu: "1", memory: 512Mi}
  authorinoDeployment:
    replicas: 2
    resources:
      requests: {cpu: "0.5", memory: 256Mi}
      limits: {cpu: "0.5", memory: 256Mi}
  guardrails:
    enabled: true
    engine:
      replicas: 3
      resources:
        requests: {cpu: "1", memory: 1Gi}
        limits: {cpu: "1", memory: 1Gi}
    shim:
      replicas: 2
      resources:
        requests: {cpu: "0.5", memory: 256Mi}
        limits: {cpu: "0.5", memory: 256Mi}
```

示例 `AmlCluster` rate-limit 配置：

```yaml
apiVersion: amlclusters.aml.dev/v1alpha3
kind: AmlCluster
metadata:
  name: default
spec:
  values:
    maas:
      gateway:
        rateLimit:
          replicas: 2
          resources:
            requests:
              cpu: 250m
              memory: 256Mi
            limits:
              cpu: 250m
              memory: 256Mi
```

常见问题

目录

转发到使用非默认端口的外部模型服务时，请求可能失败

症状

根本原因

解决方案

转发到使用非默认端口的外部模型服务时，请求可能失败

症状

当满足以下条件时，可能会出现此问题：

- 通过 Envoy AI Gateway 或 MaaS 访问外部模型服务。
- 上游模型服务监听非默认端口。
- 上游使用主机名和端口的组合进行路由或身份验证。
- 通过网关转发的请求失败。根据上游行为，响应可能是 `401`、`403`、`404` 或其他状态码。

根本原因

Envoy Gateway 的自动 backend hostname 重写可能会以不包含上游非默认端口的 `Host` / `:authority` 值转发请求。因此，基于完整主机名和端口进行路由或身份验证的上游会将该请求视为无效请求。

这是 Envoy Gateway 的 hostname-rewrite 限制。

解决方案

可参考 How to Preserve a Non-Default Backend Port in the Upstream Host Header with Envoy Gateway on ACP↗ 中描述的使用低级别 Envoy 配置的变通方法。

组件

本章所依赖组件的介绍和安装指南。

KServe

Alauda Build of KServe

KServe

文档

Envoy AI Gateway

Envoy AI Gateway 的 Alauda 版本

Envoy AI Gateway

指南

文档

Alauda Build 的 LeaderWorkerSet

LeaderWorkerSet 的 Alauda 版本

LeaderWorkerSet

文档

InferNex Bridge

InferNex Bridge 的 Alauda 构建

InferNex Bridge

部署模式

推理引擎拓扑结构

功能

文档

安装 InferNex Bridge

前提条件

上架 Operator 软件包

安装 operator

社区示例

升级 Alauda Build of InferNex Bridge

Alauda 的 KServe 构建版

Alauda Build of KServe

KServe

文档

Alauda Build of KServe

目录

KServe

Generative AI

Predictive AI

文档

KServe

Alauda Build of KServe 基于 KServe↗ 构建。 KServe 为在 Kubernetes 上大规模提供 machine learning 模型服务提供了标准化的 cloud-native 接口。 它围绕两种主要场景不断演进：用于传统 ML 推理的 **Predictive AI**，以及用于基于 LLM 工作负载的 **Generative AI**。

Generative AI

Generative AI 支持针对 Large Language Model (LLM) 服务进行了优化，并兼容 OpenAI APIs。

- **llm-d (Distributed LLM Inference)**：一个 Kubernetes-native 的分布式推理框架，运行在 KServe control plane 之下。llm-d 使用 Leader/Worker 模式协调多节点 LLM 推理，并基于 KV cache 状态和 GPU 负载做出实时路由决策，从而实现感知 KV-cache 的请求调度、弹性

的 tensor/pipeline parallelism，以及表现得像单机一样的集群级推理。这降低了每 token 成本，并最大化了大模型（例如超出单节点内存的 Llama 3.1 405B）的 GPU 利用率。

- **LLM Inference & Streaming**：原生支持流式响应（SSE / chunked transfer），为聊天和补全工作负载提供实时 token 传递，并提供兼容 OpenAI 的 `/chat/completions` 和 `/completions` APIs。
- **vLLM Runtime**：与 vLLM 的深度集成，将其作为高性能 LLM 服务后端，并支持 continuous batching 和 PagedAttention。
- **Gateway Integration**：与 Envoy Gateway 和 Gateway API Inference Extension (GIE) 原生集成，为 inference services 提供 AI-aware 流量路由、负载均衡以及按模型的速率限制。
- **Autoscaling for LLMs**：面向 LLM 吞吐特征的指标驱动 autoscaling 策略，包括 scale-to-zero，以提升成本效率。

Predictive AI

Predictive AI 覆盖了对高吞吐和低延迟有要求的传统 machine learning 模型服务。

- **InferenceService**：用于部署和管理模型服务端点的核心 CRD。支持金丝雀发布、跨模型版本的流量拆分以及 A/B 测试工作流。
- **Model Serving Runtimes**：针对常见 ML frameworks 的预集成运行时——TensorFlow Serving、TorchServe、Triton Inference Server、SKLearn、XGBoost 等。也支持通过 **ClusterServingRuntime** 和 **ServingRuntime** CRDs 使用自定义运行时。
- **Inference Graph**：**InferenceGraph** CRD 支持将多个模型组合成一个 pipeline，包括前处理/后处理节点、路由逻辑和 ensemble 模式。
- **Autoscaling**：通过 KEDA 或 Kubernetes HPA 提供 scale-to-zero 和 scale-from-zero 支持，策略基于请求速率、队列深度或自定义指标。

KServe 通过 `default` `AmlCluster` 中的 `kserve` 组件由 Alauda AI 安装和管理。有关 KServe 配置，请参阅 Install Alauda AI。

文档

KServe 上游文档和关键依赖项：

- **KServe** 文档: https://kserve.github.io/website/ ↗ — 涵盖概念、模型服务运行时和 API references 的官方文档。
- **KServe GitHub**: https://github.com/kserve/kserve ↗ — 源代码、release notes 和 issues。
- **llm-d**: https://github.com/llm-d/llm-d ↗ — Kubernetes-native 的分布式 LLM inference 框架，支持感知 KV-cache 的调度和弹性并行。
- **LeaderWorkerSet (LWS)**: https://github.com/kubernetes-sigs/lws ↗ — Kubernetes SIG workload controller，用于多节点 Leader/Worker 模式，是多节点 LLM inference 的必需组件。
- **Envoy Gateway**: https://gateway.envoyproxy.io/ ↗ — 基于 Envoy Proxy 构建的 Kubernetes-native gateway，为 KServe inference services 提供底层流量管理能力。
- **Envoy AI Gateway**: https://aigateway.envoyproxy.io/ ↗ — 构建在 Envoy Gateway 之上的 AI-specific gateway 能力，包括 AI-aware 路由和按模型策略。
- **Gateway API Inference Extension (GIE)**: https://gateway-api-inference-extension.sigs.k8s.io/ ↗ — Kubernetes SIG 项目，为 inference services 提供 AI-aware 路由和负载均衡。

Alauda 版 Envoy AI Gateway

Envoy AI Gateway 的 Alauda 版本

Envoy AI Gateway

指南

文档

Envoy AI Gateway 的 Alauda 版本

目录

Envoy AI Gateway
指南
文档

Envoy AI Gateway

Envoy AI Gateway 的 **Alauda** 版本 基于 Envoy AI Gateway↗ 项目构建。
Envoy AI Gateway 是构建在 Envoy Gateway↗ 之上的 Kubernetes 原生、面向 AI 的网关层，为 AI 推理工作负载提供智能流量管理、路由和策略执行能力。

主要组件和功能包括：

- **AI** 感知路由：根据请求内容、模型名称和后端可用性，将推理请求路由到合适的后端模型服务，从而在单一端点后实现透明的多模型服务。
- **OpenAI** 兼容 **API**：为所有下游推理服务提供统一的、与 OpenAI 兼容的 API 接口（`/v1/chat/completions`、`/v1/completions`、`/v1/models`），而不受底层运行时影响。
- 按模型速率限制和策略：在单个模型级别执行细粒度的速率限制、令牌配额和流量策略，防止资源被耗尽，并确保多租户之间公平使用。

- 后端负载均衡：使用可配置的负载均衡策略，将推理请求分配到同一模型的多个副本，并支持健康检查和自动故障转移。
- **Envoy Gateway** 集成：作为 Envoy Gateway 的扩展运行，继承其 Kubernetes Gateway API 原生控制平面、TLS 终止和可观测性功能（指标、访问日志、分布式追踪）。
- **Gateway API Inference Extension (GIE)**：与 Kubernetes SIG Gateway API Inference Extension 集成，基于实时后端状态提供高级、感知推理的调度和负载均衡决策。

Envoy AI Gateway 是 **Alauda Build of KServe** 用于暴露推理服务的必需依赖项。

Envoy AI Gateway 由 Alauda AI 通过 `default` `AmlCluster` 中的 `envoyAIGateway` 组件安装和管理。有关 Alauda AI 的安装和组件配置，请参见 安装 Alauda AI。

指南

以下指南将 Envoy AI Gateway 配置为多租户模型服务控制平面：

- 认证消费者 — 验证 SSO 令牌或 API key，并传播调用方身份。
- 配置令牌配额 — 对每个用户、每个部门和每个层级执行令牌预算限制。
- 令牌用量计量 — 报告每个租户的消耗情况，并提供计费回传数据。
- 路由到 LLM 提供商 — 通过凭证注入和故障转移为外部提供商提供前置接入。
- 对令牌用量计费 — 使用 Cost Management 将令牌用量按价格计入每个命名空间的账单。

> **在专用命名空间中创建网关**
>
> 请在专用命名空间中创建 `Gateway` 及其 `AIGatewayRoute`，例如 `maas-system`，而不是在 Envoy Gateway 控制平面命名空间 `envoy-gateway-system` 中创建。这样可以将租户网关与控制平面隔离，并避免某些版本中的一个问题：如果网关放置在控制平面命名空间中，则不会将 AI Gateway 请求处理（`ext_proc`）过滤器或 `SecurityPolicy` 规则应用到其监听器——这会悄无声息地破坏模型路由、令牌配额和认证。数据平面代理 Pod 无论如何都会在 `envoy-gateway-system` 中创建；这里唯一重要的是 `Gateway` 资源的命名空间。

文档

Envoy AI Gateway 上游文档及相关资源：

- **Envoy AI Gateway** 文档: https://aigateway.envoyproxy.io/↗ — 涵盖架构、配置和 API 参考的官方文档。
- **Envoy AI Gateway GitHub**: https://github.com/envoyproxy/ai-gateway↗ — 源代码、release notes 和 issues。
- **Envoy Gateway**: https://gateway.envoyproxy.io/↗ — Envoy AI Gateway 所扩展的底层网关基础设施。
- **Gateway API Inference Extension (GIE)**: https://gateway-api-inference-extension.sigs.k8s.io/↗ — 与 Envoy AI Gateway 集成的 Kubernetes SIG 项目，用于支持 AI 感知路由。
- **KServe**（**Alauda** 版本）: ../kserve/intro — KServe 使用 Envoy AI Gateway 作为暴露和路由推理服务的必需依赖项。

Alauda Build 的 LeaderWorkerSet

LeaderWorkerSet 的 Alauda 版本

LeaderWorkerSet

文档

LeaderWorkerSet 的 Alauda 版本

目录

LeaderWorkerSet

文档

LeaderWorkerSet

LeaderWorkerSet 的 **Alauda** 版本 基于 LeaderWorkerSet (LWS) Kubernetes SIG 项目。LeaderWorkerSet 提供了一种 Kubernetes 原生的工作负载 API，用于以 **Leader/Worker** 模式部署一组 pod，使多节点分布式工作负载——尤其是大规模 AI 模型训练和推理——能够作为 Kubernetes 上的一等公民运行。

主要组件和功能包括：

- **LeaderWorkerSet CRD**：定义一组复制的 Leader/Worker pod 集合的核心 API 资源。每个副本由一个 leader pod 和若干可配置数量的 worker pod 组成，这些 pod 作为一个单元进行协同调度和管理。
- 协同调度与拓扑感知：组内的 leader 和 worker pod 会一起调度，并支持拓扑分布约束，以便将 pod 共同放置在同一节点、机架或可用区上，从而实现低延迟的节点间通信（例如 NVLink、InfiniBand）。
- 多节点 **LLM** 推理：支持通过 tensor parallelism 或 pipeline parallelism，在多个节点上提供超出单节点 GPU 内存容量的大语言模型（例如 Llama 3.1 405B）。在此场景下，LWS 是

Alauda Build of KServe 的必需依赖。

- 多节点训练：通过提供稳定、共置的 leader/worker pod 组，以及可预测的主机名和网络标识，支持分布式训练框架（PyTorch DDP、DeepSpeed、Megatron-LM）。
- 滚动更新与故障恢复：支持在组级别进行滚动重启和 pod 自动替换，确保在发生故障或更新时，整个 Leader/Worker 组能够一致地回收。
- 启动顺序编排：leader pod 可作为入口点和协调器，worker pod 在 leader 就绪后再启动——从而支持那些要求在 worker 连接之前先初始化 master 进程的框架。

LWS 由 Alauda AI 通过 `default` `AmlCluster` 中的 `lws` 组件进行安装和管理。有关 Alauda AI 的安装和组件配置，请参见 Install Alauda AI。

文档

LeaderWorkerSet 上游文档和相关资源：

- **LeaderWorkerSet** 文档: https://lws.sigs.k8s.io/↗ — 涵盖概念、API 参考和使用指南的官方文档。
- **LeaderWorkerSet GitHub**: https://github.com/kubernetes-sigs/lws↗ — LeaderWorkerSet Kubernetes SIG 项目的源代码、API 参考和示例。
- **KServe (Alauda Build)**: ../kserve/intro — KServe 将 LeaderWorkerSet 作为多节点 LLM 推理工作负载的必需依赖。

Alauda 构建的 InferNex Bridge

InferNex Bridge 的 Alauda 构建

InferNex Bridge
部署模式
推理引擎拓扑结构
功能
文档

安装 InferNex Bridge

前提条件
上架 Operator 软件包
安装 operator
社区示例
升级 Alauda Build of InferNex Bridge

InferNex Bridge 的 Alauda 构建版

目录

InferNex Bridge
部署模式
KServe LLMInferenceService
InferNexService
推理引擎拓扑结构
功能
文档

InferNex Bridge

InferNex Bridge 的 **Alauda** 构建版 基于 openFuyao InferNex↗ 项目。 InferNex Bridge 将 KServe `LLMInferenceService` 工作负载连接到 InferNex 推理加速栈，同时也为不使用 KServe 的环境提供原生 `InferNexService` API。

该 operator 会安装 InferNex Bridge controller、admission webhooks、RBAC 以及以下主要产品 API：

- **InferNexService**：一种托管的 LLM 推理服务，可部署 inference engines、Hermes Router、Mooncake KV cache、cache-indexer、PD-Orchestrator、Eagle-Eye 以及相关资源。

- **InferNexServiceConfig**：一个可复用的配置模板，由 `InferNexService` 通过 `spec.baseRefs` 引用。

OLM bundle 还会安装托管弹性编排组件所使用的 `Tidal`、`ElasticScaler` 和 `ResourceScalingGroup` CRD。

部署模式

InferNex Bridge 支持两个部署入口。每个推理服务请选择其中一个入口，不要通过两条路径同时部署同一个服务。

InferNex Bridge 目前仅支持 NPU 推理工作负载。

KServe LLMInferenceService

当 KServe 已经安装，并且您希望继续使用 KServe `LLMInferenceService` 工作流时，请使用此模式。

将 `infernex.io/runtime: "true"` 标签添加到 `LLMInferenceService`。KServe 继续协调 inference engine、Hermes Router、Gateway、`HTTPRoute` 和 `InferencePool`；InferNex Bridge 负责协调 InferNex 增强组件，例如 Mooncake KV cache、cache-indexer、PD-Orchestrator、Eagle-Eye，以及 KServe runtime 兼容性补丁。

InferNexService

当您希望 InferNex Bridge 在不使用 KServe 作为入口的情况下管理完整推理服务时，请使用此模式。

创建一个引用一个或多个 `InferNexServiceConfig` 模板的 `InferNexService`。InferNex Bridge 会协调 inference engine、Hermes Router、增强组件；当启用智能网关路由时，还会协调 Gateway API 资源。

推理引擎拓扑结构

推理引擎拓扑结构与部署入口无关。请在推理服务或其引用的运行时配置中选择拓扑结构：

拓扑结构	行为	典型用法
Aggregate	同一个引擎工作负载同时处理 prefill 和 decode。仍然可以启用 Mooncake 和 cache-indexer，以复用 KV cache。	一种更简单的部署方式，不需要将 prefill 和 decode 角色分离。
Prefill-decode (PD)	prefill 和 decode 作为独立工作负载运行。Proxy-server 协调这两个角色，配置后 Mooncake 可在它们之间传输 KV cache 数据。	对 prefill 和 decode 角色进行独立扩缩容和调优。

功能

- **KServe** 兼容性：使用现有的 KServe `LLMInferenceService` 工作流，并通过 `infernex.io/runtime: "true"` 标签启用 InferNex 加速。
- 原生 **InferNex API**：直接使用 `InferNexService` 和可复用的 `InferNexServiceConfig` 模板部署推理服务。
- **Prefill-decode** 解耦：通过 proxy-server 协调 prefill 和 decode 工作负载，运行 P/D 推理模式。
- **Mooncake KV cache**：部署 Mooncake KV cache 和 cache-indexer 组件，用于 KV cache 复用与协调。
- 智能网关路由：集成 Hermes Router 和 Gateway API 资源，实现面向模型的请求路由。
- 弹性编排：当推理引擎 replica 字段交由 scaler 管理时，使用 PD-Orchestrator 组件，例如 Elastic-Scaler、Tidal 和 ResourceScalingGroup。
- 可选硬件可观测性：在 NATS 和所需的监控依赖就绪后，启用 Eagle-Eye 硬件监控、硬件诊断和网络性能采集。Alauda OLM 模板中默认禁用 EagleEye，禁用后不会影响核心推理路径。

平台上的安装方法请参见 安装 InferNex Bridge。

文档

InferNex Bridge 上游文档和关键依赖：

- **InferNex Bridge** 用户指南： https://gitcode.com/openFuyao/sig-ai-inference/blob/main/docs/zh/ai_inference_infernex/user_guide/ai_inference_infernex_bridge.md ↗ — 涵盖部署模式、前提条件和使用示例的上游用户指南。
- **InferNex** 源码： https://gitcode.com/openFuyao/InferNex ↗ — 源码、charts、示例和发布标签。
- **InferNex Bridge** 技术规范： https://gitcode.com/openFuyao/InferNex/blob/v26.6.0/component/InferNex-Bridge/docs/InferNex-Bridge-Technical-Specification.md ↗ — 本次发布的架构、所有权边界、webhook 行为和路由契约。
- **KServe** 文档： https://kserve.github.io/website/ ↗ — KServe 概念和 `LLMInferenceService` 文档。
- **Gateway API Inference Extension**： https://gateway-api-inference-extension.sigs.k8s.io/ ↗ — 模型路由使用的具备推理感知能力的 Gateway API 资源。

安装 InferNex Bridge

本指南介绍 **Alauda Build of InferNex Bridge v26.6.0**，该版本发布于适用于 arm64 集群的 OLM `alpha` 通道。

目录

- 前提条件
  - 所需依赖
  - 本 operator 安装的 CRD
  - Runtime 模板和镜像
    - 配置 ImageWhiteList
    - 准备模型
  - 可选依赖
  - 可选的 EagleEye 可观测性
- 上架 Operator 软件包
- 安装 operator
  - 验证
- 社区示例
- 升级 Alauda Build of InferNex Bridge
  - 验证
  - 回滚

前提条件

在安装 **Alauda Build of InferNex Bridge** 之前，请确保目标集群具备所需的平台和推理依赖。

所需依赖

Dependency	Type	Description
Kubernetes cluster	Platform	一个正在运行且具备管理员访问权限的集群。
KServe	Operator	在使用 KServe `LLMInferenceService` 入口点时必需。声明的上游兼容范围为 KServe v0.17.0-v0.19.0。Alauda Build of KServe v0.16.1 也已通过验证。
Envoy Gateway and Gateway API	Operator / CRDs	在通过 Gateway API 资源暴露推理服务时必需。
Gateway API Inference Extension	CRDs	基于 `InferencePool` 的智能路由所必需。
Alauda Build of LeaderWorkerSet	Operator (managed by AmlCluster)	InferNex Bridge 所必需。该控制器会监视 `LeaderWorkerSet` 资源，并将其用于多节点推理工作负载。通常通过 `default` `AmlCluster` 中的 `lws` 组件启用；请参见 安装 Alauda AI。
Inference runtime prerequisites	Runtime	准备所选推理引擎所需的 NPU 节点、模型存储、runtime 模板、runtime 镜像和网络访问。

INFO

`InferNexService` 模式不要求用户先安装 InferNex 主 chart。operator 会安装 InferNex Bridge 控制平面；在部署推理服务之前，需要单独准备服务模板、推理 runtime 镜像、模型文件以及特性相关的前置 CRD。

本 operator 安装的 CRD

Alauda Build of InferNex Bridge OLM bundle 安装了其受管弹性编排组件所使用的两个产品 API 和三个 CRD：

CRD	Installed by this operator
`infernexservices.infernex.infernex.io`	是
`infernexserviceconfigs.infernex.infernex.io`	是
`resourcescalinggroups.autoscaling.openfuyao.com`	是
`elasticscalers.elasticscaler.io`	是
`tidals.tidal.io`	是

以下 CRD 不会由此 OLM bundle 安装。请根据下面显示的要求单独安装：

CRD	Source Project	When Required	How t
`leaderworkersets.leaderworkerset.x-k8s.io`	Alauda Build of LeaderWorkerSet	InferNex Bridge controller 和多节点推理工作负载	确保 Leade 已安装 通常通 `defa` `AmlC` 的 `lw` 行管理 安装 A
`rolebasedgroups.workloads.x-k8s.io`	RoleBasedGroup (RBG) ↗	明确使用 `RoleBasedGroup` 的 ResourceScalingGroup 目标	从上游 v0.7.0 ↗ 安装

`rolebasedgroups.workloads.x-k8s.io` CRD 由上游 RoleBasedGroup (RBG) 项目 ↗ 提供。它是一个条件依赖项，Deployment 或 LeaderWorkerSet 目标不需要它。要安装上游 RBG v0.7.0 控制器和 CRD，请使用带版本的发布清单：

```
kubectl apply --server-side -f \
  https://raw.githubusercontent.com/sgl-project/rbg/v0.7.0/deploy/kubectl/manifests.yaml

kubectl wait deployment/rbgs-controller-manager \
  --namespace rbgs-system \
  --for=condition=Available \
  --timeout=5m

kubectl get crd rolebasedgroups.workloads.x-k8s.io \
  -o jsonpath='{range .spec.versions[?(@.name=="v1alpha2")]}{.name}{" served="}{.served}{" storage="}{.storage}{"\n"}{end}'
```

预期结果为 `v1alpha2 served=true storage=true`。该清单使用 `docker.io/rolebasedgroup/rbgs-controller:v0.7.0-888ade7`，它提供了 arm64 镜像。对于断开连接的集群，请将此镜像镜像到集群可访问的 registry，并在应用之前更新清单中的镜像引用。

在选择其他版本时，请参见上游 RBG releases↗。

启用受管的 Tidal、ElasticScaler 或 ResourceScalingGroup 组件会部署其控制器工作负载。诸如 `Tidal` 和 `ElasticScaler` 之类的扩缩容策略资源，仍必须由用户或平台工作流单独创建。

Runtime 模板和镜像

> **WARNING**
>
> operator 软件包会同步其声明的 `relatedImages`，但 OLM runtime 模板仍保留其社区来源引用。在创建推理服务之前，请配置平台镜像白名单和重写规则，使这些来源引用能够解析为目标集群 registry 中的同步镜像。
>
> 模型服务镜像 `hub.oepkgs.net/openfuyao/ascend/vllm-ascend:v0.18.0` 不包含在 operator 的 `relatedImages` 中。在部署 NPU 推理服务之前，请获取 arm64 镜像，将其上传或镜像到目标 registry，并为完整的源引用添加镜像重写规则。请验证解析后的 Pod `imageID`；工作负载规范中仍保留源引用并不表示重写失败。

配置 ImageWhiteList

平台在上传 InferNex Bridge 软件包时会创建一个由 operator 管理的 `ImageWhiteList`。请勿手动创建或删除此对象。由 operator 管理的 ImageWhiteList 对象由 `olm-registry` 进行协调，删除后可能会被重新创建。

在创建推理服务之前，请验证生成的对象包含所需的源镜像引用以及源到目标的重写规则。保留已存在的无关条目。

下表显示了预期的仓库映射。每条规则都会保留原始镜像标签。

Source repository	Target repository under `<target-registry>`
`cr.openfuyao.cn/openfuyao/cache-indexer`	`3rdparty/openfuyao/cache-indexer`
`cr.openfuyao.cn/openfuyao/proxy-server`	`3rdparty/openfuyao/proxy-server`
`cr.openfuyao.cn/openfuyao/hermes-router`	`3rdparty/openfuyao/hermes-router`
`cr.openfuyao.cn/openfuyao/hermes-router-tokenizer`	`3rdparty/openfuyao/hermes-router-tokenizer`
`cr.openfuyao.cn/openfuyao/hermes-router-prediction`	`3rdparty/openfuyao/hermes-router-prediction`
`hub.oepkgs.net/openfuyao/redis`	`3rdparty/openfuyao/redis`
`cr.openfuyao.cn/openfuyao/tidal`	`3rdparty/openfuyao/tidal`
`cr.openfuyao.cn/openfuyao/resource-scaling-group`	`3rdparty/openfuyao/resource-scaling-group`
`cr.openfuyao.cn/openfuyao/elastic-scaler`	`3rdparty/openfuyao/elastic-scaler`
`cr.openfuyao.cn/openfuyao/eagle-eye-hardware-monitor`	`3rdparty/openfuyao/eagle-eye-hardware-monitor`
`cr.openfuyao.cn/openfuyao/eagle-eye-hardware-diagnosis`	`3rdparty/openfuyao/eagle-eye-hardware-diagnosis`

Source repository	Target repository under `<target-registry>`
`cr.openfuyao.cn/openfuyao/eagle-eye-network-performance-exporter`	`3rdparty/openfuyao/eagle-eye-network-performance-exporter`
`hub.oepkgs.net/openfuyao/alpine/kubectl`	`3rdparty/openfuyao/alpine/kubectl`
`hub.oepkgs.net/openfuyao/ascend/vllm-ascend`	`3rdparty/openfuyao/ascend/vllm-ascend`

最终对象中相关部分应如下所示。完整对象还包含其他软件包镜像和平台管理条目。

```yaml
apiVersion: app.alauda.io/v1alpha1
kind: ImageWhiteList
metadata:
  name: infernex-bridge
  namespace: cpaas-system
spec:
  repoList:
    - cr.openfuyao.cn/openfuyao/cache-indexer:26.6.0
    - hub.oepkgs.net/openfuyao/ascend/vllm-ascend:v0.18.0
    - busybox:1.36.1
    - docker.io/library/busybox:1.36.1
  rewriteRules:
    - regexp: ^cr\.openfuyao\.cn/openfuyao/cache\-indexer:(.*)$
      replacement: <target-registry>/3rdparty/openfuyao/cache-indexe
r:${1}
    - regexp: ^hub\.oepkgs\.net/openfuyao/ascend/vllm\-ascend:(.*)$
      replacement: <target-registry>/3rdparty/openfuyao/ascend/vllm-ascen
d:${1}
```

必须显式添加 `vllm-ascend` 条目，因为该镜像不属于 operator 的 `relatedImages`。在创建推理工作负载之前，请将 arm64 镜像导入目标 registry。例如：

```
export TARGET_REGISTRY='<target-registry-host:port>'

skopeo copy --all \
  docker://hub.oepkgs.net/openfuyao/ascend/vllm-ascend:v0.18.0 \
  "docker://${TARGET_REGISTRY}/3rdparty/openfuyao/ascend/vllm-ascend:v0.1
8.0"
```

对于 BusyBox，请在 `repoList` 中同时保留 `busybox:1.36.1` 和 `docker.io/library/busybox:1.36.1`，但不要添加自定义 BusyBox 重写规则。 平台 `_default` 规则会处理短名称和 Docker Hub 规范化。

可以使用以下命令备份并编辑由 operator 管理的对象。如果对象名称不同，请使用 `kubectl get imagewhitelist -A` 查找。

```
kubectl get imagewhitelist -n cpaas-system infernex-bridge -o yaml \
  > imagewhitelist-infernex-bridge-backup.yaml

kubectl edit imagewhitelist -n cpaas-system infernex-bridge
```

工作负载规范中可能仍会显示社区来源镜像。只有当实际 Pod 的 `imageID` 指向目标 registry 时，重写才算成功。软件包内部组件和软件包外部 runtime 应如下解析：

```
cache-indexer=<target-registry>/3rdparty/openfuyao/cache-indexer@sha25
6:...
main=<target-registry>/3rdparty/openfuyao/ascend/vllm-ascend@sha256:...
```

验证导入镜像的架构并检查工作负载镜像 ID：

```
export TARGET_REGISTRY='<target-registry-host:port>'
export INFERENCE_NAMESPACE='<inference-service-namespace>'

crane config --platform=linux/arm64 \
  "${TARGET_REGISTRY}/3rdparty/openfuyao/ascend/vllm-ascend:v0.18.0" |
  python3 -c 'import json, sys; data=json.load(sys.stdin); assert data.ge
t("architecture") == "arm64", data; print("arm64 image verified")'

kubectl get pods -n "${INFERENCE_NAMESPACE}" \
  -o jsonpath='{range .items[*]}{.metadata.name}{"\n"}{range .status.init
ContainerStatuses[*]}  init:{.name}{"="}{.imageID}{"\n"}{end}{range .stat
us.containerStatuses[*]}  {.name}{"="}{.imageID}{"\n"}{end}{end}'
```

准备模型

模型制品不包含在 InferNex Bridge operator 软件包中。当选择 OCI 模型存储时，请使用 Alauda AI 模型目录准备 ModelCar。请参见 模型存储。

INFO

Alauda OLM bundle 会为该版本示例中使用的 KServe `LLMInferenceService` API 版本注册 InferNex Bridge admission webhook，包括 `serving.kserve.io/v1alpha2`。当在 KServe `LLMInferenceService` 上设置 `infernex.io/runtime: "true"` 时，该 webhook 用于 admission 阶段的兼容性补丁；它本身不会创建或协调 `LLMInferenceService` 资源。

可选依赖

Dependency	Required For	Description
NATS	Eagle-Eye	在启用 Eagle-Eye 硬件监控、诊断或网络性能采集时需要。Service 必须可通过 `nats.nats:4222` 访问。
kube-prometheus-stack	Eagle-Eye	在平台监控需要抓取并展示 Eagle-Eye 指标时需要。

可选的 **EagleEye** 可观测性

在 Alauda OLM 模板中，EagleEye 默认处于禁用状态。禁用 EagleEye 不会影响推理引擎、aggregate 或 prefill-decode 部署模式、Mooncake、cache-indexer 或 PD-Orchestrator。它只会禁用以下可选的可观测性工作负载：

Component	Function
Hardware Monitor	运行在加速器节点上并收集硬件状态信息。
Hardware Diagnosis	提供硬件诊断处理和诊断日志。
Network Performance Exporter	收集网络性能数据并在端口 `8222` 上暴露指标。

在启用 EagleEye 之前，请确认 NATS 具有就绪的 Service 和 endpoint：

```
kubectl get service -n nats nats
kubectl get endpointslice -n nats -l kubernetes.io/service-name=nats
```

EagleEye init 容器连接到 `nats.nats:4222`。如果该 endpoint 不可用，已启用的 EagleEye Pod 将停留在 init 阶段。

在 `InferNexService` 或引用的 `InferNexServiceConfig` 中显式启用 EagleEye：

```
spec:
  components:
    eagleEye:
      hardwareMonitor:
        enabled: true
      hardwareDiagnosis:
        enabled: true
      networkPerformanceExporter:
        enabled: true
```

应用资源后，请验证推理服务 namespace 中的可选工作负载：

```
kubectl get deployment,daemonset,service -n <inference-service-namespace>
\
  -l infernex.io/owner=<infernex-service-name>
```

预期的工作负载包括 EagleEye hardware diagnosis Deployment、hardware monitor DaemonSet 和 network performance exporter DaemonSet。如果不需要 EagleEye，请将所有三个子开关都保持为 `false`。

上架 Operator 软件包

从您的 Alauda 产品交付渠道获取 arm64 安装包，然后按照 上架软件包 的说明上传该软件包。

安装 operator

在 **Administrator** 视图中：

1. 单击 **Marketplace / OperatorHub**。
2. 在控制台顶部，从 **Cluster** 下拉列表中选择要安装 InferNex Bridge Operator 的目标集群。
3. 搜索并选择 **Alauda Build of InferNex Bridge**，然后单击 **Install**。
4. 选择 `alpha` **Channel**。
5. 选择版本 `v26.6.0`。
6. 保持 **Installation Location** 不变，默认应为 `infernex-system`。
7. 在 **Upgrade Strategy** 中选择 `Manual`。
8. 单击 **Install**。

验证

确认 **Alauda Build of InferNex Bridge** 磁贴显示以下状态之一：

- `Installing`：安装正在进行中；请等待其变为 `Installed`。
- `Installed`：安装完成。

验证 operator 控制器和 webhooks 正在运行：

```
kubectl get pods -n infernex-system
kubectl get crd \
  infernexservices.infernex.infernex.io \
  infernexserviceconfigs.infernex.infernex.io \
  resourcescalinggroups.autoscaling.openfuyao.com \
  elasticscalers.elasticscaler.io \
  tidals.tidal.io

kubectl get mutatingwebhookconfiguration -o json |
  python3 -c 'import json, sys; data=json.load(sys.stdin); versions={vers
ion for item in data["items"] for webhook in item.get("webhooks", []) for
rule in webhook.get("rules", []) if "llminferenceservices" in rule.get("r
esources", []) for version in rule.get("apiVersions", [])}; assert "v1alp
ha2" in versions, versions; print("LLMInferenceService webhook version
s:", sorted(versions))'
```

控制器 Pod 应为 `Running`，所有五个捆绑的 CRD 都应已建立，且 `LLMInferenceService` mutating webhook 应包含 `v1alpha2`。

社区示例

有关社区维护的示例，请参见 InferNex Bridge v26.6.0 examples↗。

升级 Alauda Build of InferNex Bridge

1. 备份现有的自定义资源：

   ```
   kubectl get infernexservice,infernexserviceconfig -A -o yaml > infernex
   -bridge-backup.yaml
   ```

2. 按照 上架软件包 的说明上传新版 **Alauda Build of InferNex Bridge** operator 软件包。
3. 前往 `Administrator` -> `Marketplace` -> `OperatorHub` 页面，找到 **Alauda Build of InferNex Bridge**，查看目标版本，然后单击 **Confirm** 以应用。

验证

升级后，确认 **Alauda Build of InferNex Bridge** 磁贴显示 `Installed`，并验证控制器和 CRD 状态：

```
kubectl get pods -n infernex-system
kubectl get crd \
  infernexservices.infernex.infernex.io \
  infernexserviceconfigs.infernex.infernex.io \
  resourcescalinggroups.autoscaling.openfuyao.com \
  elasticscalers.elasticscaler.io \
  tidals.tidal.io
```

回滚

仅当某个旧版本的 CRD 与集群中已存储的自定义资源兼容时，才使用该旧软件包。在 OperatorHub 中，选择可用的上一个版本，并确认手动升级操作。如果该旧版本不可用，请先上传其安装包（参见 上架软件包）。

回滚后，请验证控制器、CRD、推理服务、runtime 镜像 ID，以及一个具有代表性的推理请求。在自定义资源仍然存在时，不要删除这五个捆绑的 CRD；删除它们也会删除已存储的自定义资源。

构建 AI 应用

本章介绍如何构建智能体运行时、MCP servers 和 RAG 应用。

本章所依赖的组件列在下方的 Components 中。启用 secure profile 并安装其依赖项属于管理员任务，详见 Administer。

使用 Core Profile 快速开始

使用 Core Profile 快速开始

前提条件

步骤 1：部署 MCP tool server

步骤 2：部署 agent

步骤 3：检查状态

步骤 4：发送查询（端到端测试）

更新和删除 AgentRuntime

清理

下一步

故障排查

使用 Secure Profile 的 Demo

使用 Secure Profile 的 Demo

前提条件

步骤 1：部署 MCP tool server

步骤 2：部署受保护的 agent

步骤 3：验证 secure wiring

步骤 4：发送已认证查询

AuthBridge 部署模式

清理

故障排查

Demo Advanced：带有 Token Exchange 的 Tool 上的 AuthBridge

Demo Advanced：带有 Token Exchange 的 Tool 上的 AuthBridge

涉及的 ConfigMap

Keycloak client-id 约定

前提条件

第 1 步：部署带有 AuthBridge sidecar 的 tool

第 2 步：为 token exchange 配置 Keycloak

第 3 步：准备 agent 的出站规则

第 4 步：部署受保护的 agent

第 5 步：验证 token-exchange 路径

清理

故障排查

Envoy AI Gateway 后端包含两个 MCP server 的 Kagenti

Envoy AI Gateway 后端包含两个 MCP server 的 Kagenti

已验证版本

拓扑和 HTTPRoute 数量

来源和镜像

部署 gateway 和两个 MCP server

测试统一的 MCP endpoint

部署 Kagenti 智能体

清理

组件

Dify **Llama Stack** **Kagenti**

MCP 生命周期 Operator

使用 Core Profile 快速开始

本指南将带你完成 Kagenti Operator 的 **core profile** 上最小可行的端到端部署——这是默认的安装形态，支持 agent enrollment 和动态发现，但不包含 identity sidecar、Keycloak 或 mesh mTLS。它是验证 operator 安装是否成功、并在逐步叠加 secure profile 之前熟悉 `AgentRuntime` 自定义资源的最快方式。

该场景是一个 **weather agent**：它通过调用用于获取实时数据的 **weather MCP tool** 和一个大语言模型来生成回复，回答类似 *"What is the weather in New York?"* 这样的问题。请求流程如下：

```
client ──A2A message/send──▶ weather-agent ──OpenAI chat API──▶ InferenceService (qwen36-27b-gguf)
                                    │
                                    └──────────MCP──────────▶ weather-tool-mcp
```

在 core profile 下，没有认证或传输安全——下面的每一步都只是集群内的普通 HTTP 调用。关于安全版本（Bearer-token 请求、SPIRE mTLS、每个 agent 一个 AuthBridge sidecar），请在完成本指南后继续阅读 使用 Secure Profile 演示。

目录

前提条件

找到你的模型 endpoint 和名称

步骤 1：部署 MCP tool server

步骤 2：部署 agent

步骤 3：检查状态

步骤 4：发送查询（端到端测试）

更新和删除 AgentRuntime

清理

下一步

故障排查

前提条件

- 已安装 Kagenti Operator，且其 `Kagenti` operand 已完成 reconciliation——请参见 安装。
- 具备目标集群的 `kubectl` 访问权限。
- 一个演示命名空间。本指南使用 `team1`：

```
kubectl create namespace team1
```

- 一个提供 **OpenAI** 兼容 **chat API** 的 **InferenceService**，且可在集群内访问。本指南使用名为 `qwen36-27b-gguf` 的 InferenceService。任何 chat 模型都可以；如果模型支持 tool calling，agent 效果会更好。

找到你的模型 **endpoint** 和名称

agent 通过三个环境变量连接模型——`LLM_API_BASE`、`LLM_API_KEY` 和 `LLM_MODEL`。请从你的 InferenceService 中解析它们：

```
# The predictor Service exposes the OpenAI-compatible API in-cluster.
# Base URL pattern: http://<isvc-name>-predictor.<namespace>.svc.cluster.local/v1
kubectl get svc -n <model-namespace> | grep <isvc-name>-predictor

# The model id to send as "model" — list what the endpoint serves:
kubectl run modelq --rm -i --restart=Never -n <model-namespace> \
  --image=docker.io/alaudadockerhub/curl:8.1.2 --command -- \
  curl -sS http://<isvc-name>-predictor.<model-namespace>.svc.cluster.local/v1/models
```

对于部署在 `models` 命名空间中的 `qwen36-27b-gguf` InferenceService，解析结果如下：

变量	值
`LLM_API_BASE`	`http://qwen36-27b-gguf-predictor.models.svc.cluster.local/v1`
`LLM_MODEL`	`qwen36-27b-gguf`
`LLM_API_KEY`	`dummy`（任意非空值即可；集群内 endpoint 不需要 key）

INFO

在本指南的其余部分，请将示例中的 InferenceService 名称、命名空间和 model id 替换为你自己的值。如果模型是 **reasoning** 模型，响应的前几个 token 会是 reasoning 内容——请像下面这样保持 agent 的 token budget 不受限制，以免最终答案被截断。

步骤 1：部署 MCP tool server

MCP（Model Context Protocol）server 提供 agent 要调用的 `get_weather` tool。请部署它并通过 Service 暴露出来。在 core profile 下，tool 以普通 `Deployment` 方式运行（单容器、无 sidecar）；在 secure profile 下也会继续沿用相同布局，因为 operand 默认值为 `spec.featureGates.injectTools: false`。

```
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weather-tool
  namespace: team1
  labels:
    app.kubernetes.io/name: weather-tool
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: weather-tool
  template:
    metadata:
      labels:
        app.kubernetes.io/name: weather-tool
    spec:
      containers:
      - name: mcp
        image: docker.io/alaudadockerhub/weather_tool:v0.1.0-rc.1 1
        imagePullPolicy: IfNotPresent
        env:
        - name: PORT
          value: "8000"
        - name: HOST
          value: 0.0.0.0
        - name: UV_CACHE_DIR
          value: /app/.cache/uv
        ports:
        - containerPort: 8000
        volumeMounts:
        - mountPath: /app/.cache
          name: cache
      volumes:
      - name: cache
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: weather-tool-mcp 2
  namespace: team1
```

```
  spec:
    selector:
      app.kubernetes.io/name: weather-tool
    ports:
    - name: http
      port: 8000
      targetPort: 8000
  EOF
```

1 MCP server 镜像已镜像到 `docker.io/alaudadockerhub`。在离线集群中，请使用重新分发到你平台 registry 中的副本。

2 该 Service 供 agent 访问，地址为 `http://weather-tool-mcp.team1.svc.cluster.local:8000/mcp`。operator 不会 自动为你创建这个 Service，因此需要你显式定义它，并选择 tool 的 pod。

等待 tool 就绪：

```
kubectl rollout status deploy/weather-tool -n team1
kubectl get pod -n team1 -l app.kubernetes.io/name=weather-tool
# NAME                              READY   STATUS    RESTARTS   AGE
# weather-tool-XXXXXXXXXX-YYYYY   1/1     Running   0          1m
```

`READY 1/1` 表示该 tool 正在以单容器方式运行。

步骤 2：部署 agent

agent 只需要在其 Deployment 上添加一个 `protocol.kagenti.io/a2a` 标签——controller 会应用 `kagenti.io/type`，计算 config hash，并在启用 identity stack 时触发 sidecar 注入。protocol 标签还会告诉 AgentCard sync controller 该 agent 使用哪种 protocol，从而启用自动发现。

```
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weather-agent
  namespace: team1
  labels:
    app.kubernetes.io/name: weather-agent
    protocol.kagenti.io/a2a: "" (1)
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: weather-agent
  template:
    metadata:
      labels:
        app.kubernetes.io/name: weather-agent
    spec:
      containers:
      - name: agent
        image: docker.io/alaudadockerhub/weather_service:v0.1.0-rc.1 (2)
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8000
        env:
        - name: PORT
          value: "8000"
        - name: UV_CACHE_DIR
          value: /app/.cache/uv
        - name: MCP_URL
          value: http://weather-tool-mcp.team1.svc.cluster.local:8000/mcp
(3)
        - name: LLM_API_BASE
          value: http://qwen36-27b-gguf-predictor.models.svc.cluster.loca
l/v1 (4)
        - name: LLM_API_KEY
          value: dummy
        - name: LLM_MODEL
          value: qwen36-27b-gguf (5)
---
apiVersion: v1
kind: Service
```

```
  metadata:
    name: weather-agent
    namespace: team1
  spec:
    selector:
      app.kubernetes.io/name: weather-agent
    ports:
    - name: http
      port: 8000
      targetPort: 8000
  ---
  apiVersion: agent.kagenti.dev/v1alpha1
  kind: AgentRuntime
  metadata:
    name: weather-agent-runtime
    namespace: team1
  spec:
    type: agent
    targetRef:
      apiVersion: apps/v1
      kind: Deployment
      name: weather-agent
  EOF
```

1 `protocol.kagenti.io/a2a: ""` 将该 Deployment 标记为 A2A agent。它会启用 `AgentCard` 的自动创建；`ValidatingAdmissionPolicy` 会禁止直接设置 `kagenti.io/type`，因此 enrollment 必须通过 `AgentRuntime` 进行。

2 agent 镜像已镜像到 `docker.io/alaudadockerhub`。请使用 `v0.1.0` 或更高版本——这些版本会从环境变量中读取 `HOST` / `PORT`，这在 secure profile 下是必需的（AuthBridge sidecar 会重映射应用端口）。

3 `MCP_URL` —— 步骤 1 中的 MCP tool endpoint（Service 名称加上 `/mcp` 路径）。

4 `LLM_API_BASE` —— 你的 InferenceService predictor 的 OpenAI 兼容 base URL（`.../v1`）。

5 `LLM_MODEL` —— 由 InferenceService 提供的 model id。

agent 的 Service 以 Deployment 名称命名（`weather-agent`）；`AgentRuntime` controller 会解析它，以便获取用于发现的 Agent Card。

创建 `AgentRuntime` 后，controller 将执行以下操作：

1. 解析 `targetRef` 并验证 Deployment 是否存在。
2. 应用 `kagenti.io/type: agent` 和 `app.kubernetes.io/managed-by: kagenti-operator` 标签。
3. 计算 config hash，并将其作为 `kagenti.io/config-hash` 注解设置到 pod template 上，从而触发 rolling update。

步骤 3：检查状态

```
# AgentRuntime status — only the agent has one in this pattern
kubectl get agentruntime -n team1
# NAME                    TYPE    TARGET          READY   AGE
# weather-agent-runtime   agent   weather-agent   True    1m

# Conditions (core profile: MTLSReady is False with reason SPIREUnavailable — expected)
kubectl get agentruntime weather-agent-runtime -n team1 \
  -o jsonpath='{range .status.conditions[*]}{.type}={.status}{"\n"}{end}'
# TargetResolved=True
# IstioMeshEnrolled=True
# MTLSReady=False
# ConfigResolved=True
# Ready=True

# Labels applied by the operator
kubectl get deployment weather-agent -n team1 --show-labels

# Pods: agent 1/1, tool 1/1 — no sidecars in the core profile
kubectl get pods -n team1
# NAME                              READY   STATUS    RESTARTS   AGE
# weather-agent-XXXXXXXXX-YYYYY     1/1     Running   0          1m
# weather-tool-XXXXXXXXX-YYYYY      1/1     Running   0          6m

# The AgentCard is created and synced automatically
kubectl get agentcards -n team1
# NAME                            PROTOCOL   KIND         TARGET
AGENT               SYNCED   AGE
# weather-agent-deployment-card   a2a        Deployment   weather-agent
Weather Assistant   True     1m
```

`SYNCED=True` 且 `AGENT` 列已填充（此处为 `Weather Assistant`）表示 sync controller 已获取到该 agent 的 A2A card——动态发现已正常工作。

步骤 4：发送查询（端到端测试）

从一个临时的集群内 pod 向 agent 发送 A2A `message/send` 请求，使用 agent 的内部 Service DNS 名称：

```
kubectl run curl-wq --rm -i --restart=Never -n team1 \
  --image=docker.io/alaudadockerhub/curl:8.1.2 --command -- \
  curl -sS -X POST http://weather-agent.team1.svc.cluster.local:8000/ \
    -H "Content-Type: application/json" \
    -d '{"jsonrpc":"2.0","id":"76CD6BA3-16AA-4CED-8E0E-19156B8C5886","met
hod":"message/send","params":{"message":{"role":"user","parts":[{"kin
d":"text","text":"What is the weather in NY?"}],"messageId":"DF05857B-98B
7-4414-BD63-19E16E684E39"}}}'
```

agent 会调用模型，模型再调用 weather MCP tool，最终 agent 返回一个已完成的 A2A task：

```
{
  "id": "76CD6BA3-16AA-4CED-8E0E-19156B8C5886",
  "jsonrpc": "2.0",
  "result": {
    "artifacts": [
      {
        "parts": [
          {
            "kind": "text",
            "text": "The current weather in New York is 66.4°F with clear
skies. The wind speed is 4.4."
          }
        ]
      }
    ],
    "kind": "task",
    "status": { "state": "completed" }
  }
}
```

完整响应中的 `history` 数组展示了 tool-calling 流程（`assistant` → `tools` → `assistant`），其中包含由 weather MCP server 返回的 `ToolMessage`。

> **INFO**
>
> reasoning 模型会先用前几个 token “思考”，再给出最终答案，因此第一次请求可能会明显慢于非 reasoning 模型。后续请求会更快。

你可以在日志中观察交互过程：

```
# Agent logs
kubectl logs -f -l app.kubernetes.io/name=weather-agent -n team1

# MCP tool logs (in another terminal)
kubectl logs -f -l app.kubernetes.io/name=weather-tool -n team1
```

更新和删除 AgentRuntime

平台配置变更（集群级或命名空间级 ConfigMap）会触发 rolling update，以便 pod 获取新设置。直接编辑 `AgentRuntime` 的 spec 本身不会强制重启——新值只会在创建 pod 时生效。

删除 `AgentRuntime` 会执行优雅清理：controller 会移除 `kagenti.io/type` 标签和 `kagenti.io/config-hash` 注解（从而触发 rolling update，使任何已注入的 pod 被替换），并移除 `app.kubernetes.io/managed-by` 标签。

```
kubectl delete agentruntime weather-agent-runtime -n team1
```

清理

```
kubectl delete namespace team1
```

下一步

你刚刚体验的 core profile 不包含认证或传输安全。要逐步加入这些能力：

1. 安装 Secure-Profile Dependencies — SPIRE、Keycloak、Istio ambient。
2. 启用 Secure Profile — 切换 operand feature gate。
3. 使用 Secure Profile 演示 — 用同样的 weather 场景重新演示，在 agent 上加上 AuthBridge sidecar、每个 workload 一个 SPIRE SVID，以及 Bearer-token 认证请求。
4. 高级演示：在 tool 上使用 AuthBridge 和 Token Exchange — 也为 tool 添加 AuthBridge，在 agent 和 tool 之间使用 RFC 8693 token exchange。

故障排查

症状	检查项
`AgentRuntime` 不是 `Ready`	`kubectl describe agentruntime <name> -n team1` —— 确认 `TargetResolved`。常见原因是目标 Deployment 或其 Service هنوز 还不存在。
agent 无响应 / 超时	检查 agent 是否能访问模型：`LLM_API_BASE` 是否可解析，且 `/v1/models` 是否列出了 `LLM_MODEL`。查看 `kubectl logs -l app.kubernetes.io/name=weather-agent -n team1` 中的 LLM 连接错误。
agent 回答中没有实时数据	确认 `MCP_URL` 指向 `weather-tool-mcp` Service，且 tool pod 处于 `Running`：`kubectl logs -l app.kubernetes.io/name=weather-tool -n team1`。
`ImagePullBackOff`	在没有 `docker.io` 出站访问权限的集群上，请将镜像重新分发到你的平台 registry（例如通过 `violet`），并引用该副本。
`MTLSReady=False`	在 core profile 中这是预期行为（原因是 `SPIREUnavailable`）；它不会影响 agent 或 tool 的功能。

使用 Secure Profile 的 Demo

本指南在 **secure profile** 下重新运行 使用 Core Profile 的快速开始 中的 weather-agent 场景——AuthBridge sidecar 会注入到 agent 中，operator 会为每个 workload 注册一个 Keycloak client，SPIRE 会签发 workload SVID，而每个请求都必须携带有效的 Bearer 令牌。tool workload 仍然以普通 `Deployment` 形式运行，不带 sidecar；这与上游 weather-agent demo 模式一致，并让 tool 到公共地理编码 API 的出站调用继续走 pod 的默认 egress 路径。

如果你需要一个更严格的变体——tool 上也启用 AuthBridge，并在 agent 与 tool 之间使用 RFC 8693 token exchange——请在完成本指南后继续查看 高级 Demo。

目录

- 前提条件
- 步骤 1：部署 MCP tool server
- 步骤 2：部署受保护的 agent
- 步骤 3：验证 secure wiring
- 步骤 4：发送已认证查询
- AuthBridge 部署模式
  - 选择模式
  - 模式变更后重新加载
  - 何时使用每种模式
  - 在 `proxy-sidecar` 模式下允许出站流量
- 清理
- 故障排查

前提条件

开始之前，请确保：

1. **secure-profile** 依赖已安装——cert-manager、SPIRE、Keycloak、Istio ambient mesh。请参见 安装 Secure-Profile 依赖。
2. 在 `Kagenti` operand 上已启用 **secure profile**（`spec.featureGates.globalEnabled: true`、`spec.authbridgeConfig.enabled: true`、`spec.keycloak.publicUrl` 已设置）。请参见 启用 Secure Profile。
3. 集群内可访问一个 提供 **OpenAI-compatible chat API** 的 **InferenceService**。本指南在 `models` 中使用 `qwen36-27b-gguf`；请在你看到这些名称的所有位置替换为你自己的模型——有关如何解析 `LLM_API_BASE`、`LLM_MODEL` 和 `LLM_API_KEY`，请参见 快速开始中的 model-endpoint recipe。
4. 拥有目标集群的 `kubectl` 访问权限。

创建 demo 命名空间并让其启用 secure profile：

```
kubectl create namespace team1
kubectl label namespace team1 kagenti-enabled=true
```

> **WARNING**
>
> `kagenti-enabled=true` 标签是必需的。如果没有它，operator 不会 为该命名空间创建 `authbridge-config` ConfigMap，workload 的 Keycloak client 注册不会启动，并且每个 pod 都会因为缺少凭据 Secret 而一直处于 Pending。

步骤 1：部署 MCP tool server

tool workload 与 core-profile 版本 完全相同。它以普通 `Deployment` 运行，不使用 `AgentRuntime`，也不带 AuthBridge sidecar——默认情况下，tool workload 的 sidecar 注入是关闭的（`spec.featureGates.injectTools: false`），因此即使你尝试，仅仅为该 `Deployment` 添加 `kagenti.io/type: tool` 标签也不会注入 sidecar。其安全模型是 agent

侧的入站保护：agent 的 AuthBridge 会验证客户端的 JWT，然后 agent → tool 调用只是一次普通的集群内 HTTP 跳转。

```
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weather-tool
  namespace: team1
  labels:
    app.kubernetes.io/name: weather-tool
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: weather-tool
  template:
    metadata:
      labels:
        app.kubernetes.io/name: weather-tool
    spec:
      containers:
      - name: mcp
        image: docker.io/alaudadockerhub/weather_tool:v0.1.0-rc.1
        imagePullPolicy: IfNotPresent
        env:
        - name: PORT
          value: "8000"
        - name: HOST
          value: 0.0.0.0
        - name: UV_CACHE_DIR
          value: /app/.cache/uv
        ports:
        - containerPort: 8000
        volumeMounts:
        - mountPath: /app/.cache
          name: cache
      volumes:
      - name: cache
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: weather-tool-mcp
  namespace: team1
```

```
spec:
  selector:
    app.kubernetes.io/name: weather-tool
  ports:
  - name: http
    port: 8000
    targetPort: 8000
EOF

kubectl rollout status deploy/weather-tool -n team1
kubectl get pod -n team1 -l app.kubernetes.io/name=weather-tool
# NAME                            READY   STATUS    RESTARTS   AGE
# weather-tool-XXXXXXXXXX-YYYYY   1/1     Running   0          1m
```

`READY 1/1` 表明即使在 secure profile 下，tool 也仍然以单容器运行——没有注入 sidecar。

> **agent 和 tool 镜像必须遵守**
>
> `$PORT` 在 secure profile 下，AuthBridge proxy-sidecar 会绑定 workload 的 **service port**，而 operator 会将应用重映射到不同的 `PORT` 值。若镜像硬编码了端口，就会与反向代理冲突（`address already in use`）。上游 `weather_service` / `weather_tool` 镜像从 `v0.1.0` 起已遵守 `PORT`，因此可以直接使用。

步骤 2：部署受保护的 agent

agent 的 Deployment、Service 和 `AgentRuntime` 与 core-profile 快速开始 中相同——不需要任何 secure-profile 特有字段，因为 AuthBridge webhook 会根据 `Kagenti` operand 和命名空间标签在准入时自动完成所有挂接工作。`AgentRuntime` 会触发 mutating webhook 注入 sidecar，并挂载 operator 创建的 Keycloak client Secret。

```
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weather-agent
  namespace: team1
  labels:
    app.kubernetes.io/name: weather-agent
    protocol.kagenti.io/a2a: ""
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: weather-agent
  template:
    metadata:
      labels:
        app.kubernetes.io/name: weather-agent
    spec:
      containers:
      - name: agent
        image: docker.io/alaudadockerhub/weather_service:v0.1.0-rc.1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8000
        env:
        - name: PORT
          value: "8000"
        - name: UV_CACHE_DIR
          value: /app/.cache/uv
        - name: MCP_URL
          value: http://weather-tool-mcp.team1.svc.cluster.local:8000/mcp
        - name: LLM_API_BASE
          value: http://qwen36-27b-gguf-predictor.models.svc.cluster.loca
l/v1
        - name: LLM_API_KEY
          value: dummy
        - name: LLM_MODEL
          value: qwen36-27b-gguf
---
apiVersion: v1
kind: Service
metadata:
```

```
  name: weather-agent
  namespace: team1
spec:
  selector:
    app.kubernetes.io/name: weather-agent
  ports:
  - name: http
    port: 8000
    targetPort: 8000
---
apiVersion: agent.kagenti.dev/v1alpha1
kind: AgentRuntime
metadata:
  name: weather-agent-runtime
  namespace: team1
spec:
  type: agent
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: weather-agent
EOF
```

步骤 3：验证 secure wiring

确认 AuthBridge sidecar 已注入、SPIRE 已签发 SVID，并且 operator 已为该 workload 注册 Keycloak client：

```
# AuthBridge sidecar container present alongside the app container
kubectl get pod -n team1 -l kagenti.io/type=agent \
  -o jsonpath='{.items[0].spec.containers[*].name}{"\n"}'
# agent authbridge-proxy

# Agent pod is 2/2 (app + sidecar); tool pod is still 1/1
kubectl get pods -n team1
# NAME                                READY   STATUS    RESTARTS   AGE
# weather-agent-XXXXXXXXX-YYYYY       2/2     Running   0          1m
# weather-tool-XXXXXXXXX-YYYYY        1/1     Running   0          6m

# SPIRE X.509 SVID mirrored into the sidecar
kubectl exec -n team1 -c authbridge-proxy \
  $(kubectl get pod -n team1 -l app.kubernetes.io/name=weather-agent -o n
ame | head -1) \
  -- ls -l /opt/svid.pem

# Keycloak client credentials Secret created by the operator and mounted
at /shared/
kubectl get secret -n team1 | grep kagenti-keycloak-client
# kagenti-keycloak-client-credentials-<hash>   Opaque   2   30s
```

该 Secret 为 workload 动态注册的 Keycloak client（名为 `team1/weather-agent`）提供 `client-id.txt` 和 `client-secret.txt`。webhook 会将其挂载到 agent pod 的 `/shared/`，AuthBridge 从那里读取。

> 现在请求需要 **token**
>
> 启用 secure profile 后，向 agent 发起的 未认证 请求会被 AuthBridge `jwt-validation` plugin 拒绝：
>
> ```
> {"error":"auth.unauthorized","message":"missing Authorization heade
> r","plugin":"jwt-validation"}
> ```

步骤 4：发送已认证查询

由于 workload 自身的 Keycloak 凭据 Secret 已经挂载在集群内，验证受保护路径的最简单方式是启动一个短时 pod，该 pod 会：

1. 将相同的凭据 Secret 挂载到 `/creds/`。
2. 通过 Keycloak `client_credentials` grant 获取 access token。
3. 带上 `Authorization: Bearer <token>` 发送 A2A `message/send` 请求。

无需 port-forwarding，且 client secret 不会离开集群。

查找 agent 的凭据 Secret 名称：

```
kubectl get secret -n team1 -o name | grep kagenti-keycloak-client
# secret/kagenti-keycloak-client-credentials-<hash>
```

从 pod 中运行已认证查询。将 `<credentials-secret>` 替换为上一条命令输出的名称：

```
kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: weather-query
  namespace: team1
spec:
  restartPolicy: Never
  containers:
  - name: query
    image: docker.io/alaudadockerhub/curl:8.1.2
    command: ["/bin/sh", "-c"]
    args:
    - |
      CID=$(cat /creds/client-id.txt)
      CSEC=$(cat /creds/client-secret.txt)

      # 1. Get an access token from Keycloak (client_credentials grant).
      # Use the same Keycloak hostname form set as `keycloak.publicUrl` o
n the
      # Kagenti operand (the JWT `iss` claim must match what AuthBridge v
alidates).
      # The recommended in-cluster form is the short svc URL:
      TOKEN=$(curl -s \
        http://keycloak-service.keycloak.svc:8080/realms/kagenti/protoco
l/openid-connect/token \
        -d grant_type=client_credentials \
        --data-urlencode "client_id=$CID" \
        --data-urlencode "client_secret=$CSEC" \
        | sed 's/.*"access_token":"\([^"]*\)".*/\1/')

      # 2. Call the agent with the token in the Authorization header
      curl -sS -X POST http://weather-agent.team1.svc.cluster.local:8000/
\
        -H "Authorization: Bearer $TOKEN" \
        -H "Content-Type: application/json" \
        -d '{"jsonrpc":"2.0","id":"1","method":"message/send","params":
{"message":{"role":"user","parts":[{"kind":"text","text":"What is the wea
ther in NY?"}],"messageId":"m1"}}}'
    volumeMounts:
    - name: creds
      mountPath: /creds
      readOnly: true
    -
```

```
  volumes:
  - name: creds
    secret:
      secretName: <credentials-secret>
EOF
```

读取结果并清理：

```
kubectl logs weather-query -n team1
kubectl delete pod weather-query -n team1
```

你将获得与 core-profile 第 4 步 相同的已完成 A2A task——现在是通过已认证、JWT 已验证的路径完成的。

> **从集群外部调用**
>
> 如果要从集群外部调用 agent，请先从通过 Ingress/Gateway 暴露的 Keycloak endpoint 请求 token，然后使用相同的 `Authorization: Bearer <token>` header 将请求发送到 agent 的暴露 endpoint。token 请求是标准的 OIDC 调用：`POST <keycloak>/realms/kagenti/protocol/openid-connect/token`，并携带 `grant_type=client_credentials`、`client_id` 和 `client_secret`。

AuthBridge 部署模式

AuthBridge 可以以两种形态注入。模式决定会添加哪个 sidecar 镜像、如何验证入站流量，以及如何拦截出站流量。它是一个 按命名空间 生效的默认值（管理员配置项），而不是按 workload 生效——该命名空间内的所有 agent 和 MCP tool 都共享同一种模式。

模式	**Sidecar** 容器 **/** 镜像	入站（**client** → **workload**）	出站（**workload** → 其他服务）
`proxy-sidecar`（默认）	`authbridge-proxy` 容器，镜像 `authbridge`。	在 workload 的 service port 上进行反向代理 → JWT 验证 → 转发到偏移后的应用端口。	协作式：webhook 注入指向 sidecar 上 `:8081` 处 HTTP forward proxy 的 `HTTP_PROXY` / `HTTPS_PROXY` 环境变

模式	**Sidecar** 容器 **/** 镜像	入站（**client** → **workload**）	出站（**workload** → 其他服务）
			量。应用通过遵守这些环境变量来接入。
`envoy-sidecar`	`envoy-proxy` 容器，镜像 `authbridge-envoy`（Envoy + authbridge 作为 ext_proc）。另外会添加一个 `proxy-init` initContainer，用于安装 iptables 规则。	透明重定向：iptables 将入站流量发送到 Envoy，Envoy 再通过 gRPC ext_proc 调用 authbridge 做 JWT 验证。	透明式：iptables 将出站 TCP 重定向到 Envoy 的 outbound listener。应用上不会设置 `HTTP_PROXY` 环境变量。

选择模式

模式按以下顺序解析（第一个非空项生效）：

1. 命名空间 ConfigMap **`authbridge-runtime-config`** —— `config.yaml` 中的 `mode:` 字段。
2. 集群级默认值：`proxy-sidecar`。

要将某个命名空间切换为 `envoy-sidecar`，请应用一个 ConfigMap 并重启 agent 与 tool 部署（见 模式变更后重新加载）：

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: authbridge-runtime-config
  namespace: team1
data:
  config.yaml: |
    mode: envoy-sidecar
    mtls:
      mode: disabled
```

WARNING

`envoy-sidecar` 在当前版本中会在 `mtls.mode: permissive` 下 正常启动——`authbridge-envoy` 二进制通过 go-spiffe 在进程内将 X.509 SVID 写入 `/opt/svid*.pem`，这正是 operator 的 Envoy filesystem-SDS 配置所期望的行为。不过，目前仍有两个 operator 模板缺口，正在跟进下一个版本：

- 入站认证强制：Envoy 配置模板会在 route 级别注入 `x-authbridge-direction: inbound`，而 router filter 会在 `ext_proc` 之后 应用该值。因此 AuthBridge 会将入站请求归类为出站请求，并应用默认透传策略——JWT-validation plugin 不会运行。后续补丁会将该 header 注入移动到位于 `ext_proc` 之前的 Lua HTTP filter 中。
- 通过 **sidecar** 的外部 **HTTPS**：Envoy 的 outbound listener 默认假定为 HTTP，因此对任意公共主机的客户端 TLS 握手会被当作 HTTP 解析并破坏。该模式下没有 `NO_PROXY` 逃逸口；后续会在 `Kagenti` operand 上添加 `spec.egressBypass` 字段，并下传为 proxy-init 的 `OUTBOUND_PORTS_EXCLUDE` / hosts allowlist。

当前版本建议： 在生产环境中使用 `proxy-sidecar`。它的入站 JWT 验证以及 `NO_PROXY` 出站 egress 选项（见下文）都已完成端到端验证。`envoy-sidecar` 适合用于启动和检查数据路径，但在上述两个项目落地之前，不应将其作为生产认证强制的依据。

模式变更后重新加载

由于 manager 和按 workload 注入的配置都会在 pod 创建时缓存，因此模式变更需要执行三个操作：

```
# 1. Roll the operator so it picks up the new namespace ConfigMap.
kubectl -n kagenti-system rollout restart deploy/kagenti-controller-manag
er

# 2. Delete stale per-workload ConfigMaps so the webhook re-renders them
#    with the new mode. (Kept for both modes; the operator preserves
#    existing values, see issue upstream #433.)
kubectl -n team1 delete cm envoy-config-weather-agent envoy-config-weathe
r-tool 2>/dev/null

# 3. Recycle the workloads so the webhook re-injects the sidecars.
kubectl -n team1 rollout restart deploy/weather-agent deploy/weather-tool
```

确认注入已按预期生效：

```
# proxy-sidecar: container "authbridge-proxy" runs image "authbridge:..."
# envoy-sidecar: container "envoy-proxy" runs image "authbridge-envo
y:...",
#                and an "proxy-init" initContainer appears.
kubectl -n team1 get pod -l app=weather-agent -o jsonpath='{range .spec.c
ontainers[*]}{.name}={.image}{"\n"}{end}'
```

何时使用每种模式

当满足以下任一条件时，使用 `proxy-sidecar`（默认值）——这是当前版本经过测试并受支持的路径：

- 你需要 sidecar-to-sidecar mTLS（`permissive` 或 `strict`）。
- 你只会调用其他集群内 HTTP 服务（agent → MCP tool、LLM InferenceService 等），不需要 `CONNECT` tunneling。
- workload 不需要 SPIRE。

当你明确需要以下功能时，使用 `envoy-sidecar`：

- 数据路径上的 Envoy 路由 / 可观测性能力。
- 透明的 iptables 拦截（应用上没有 `HTTP_PROXY` 环境变量）。

在 `proxy-sidecar` 模式下允许出站流量

本指南遵循的上游 **weather-agent demo** 模式——**tool workload** 上没有 `AgentRuntime`，也没有 **sidecar**（`spec.featureGates.injectTools: false` 是 **operand** 默认值）——会完全绕开这个问题：tool 以单容器运行，没有 `HTTP_PROXY` / `HTTPS_PROXY` 环境变量，因此它对公共 API 的出站调用会直接通过 pod 的默认 egress。对于 tool 不需要自身入站 JWT 验证或出站 token exchange 的场景，这是推荐的布局。

本节其余内容仅适用于你有意将 AuthBridge sidecar 注入到也需要外部 HTTPS 的 workload 中的情况——例如你在 `Kagenti` operand 上将 `injectTools: true` 打开（参见 高级 Demo），或者在单个 Deployment 上添加 `kagenti.io/authbridge-inject: "true"`。AuthBridge 的 HTTP forward proxy 仅支持 **HTTP**——它会对 `CONNECT` 返回 `405 Method Not Allowed`，因此注入后的应用如果通过 sidecar 发起出站 HTTPS 请求，就会遇到

`Tunnel connection failed: 405`。这符合设计预期：出站 `token-exchange` plugin 需要读取和修改请求 header，而这在不透明的 TLS tunnel 中无法实现。

当 outbound-heavy workload 上带有 sidecar 时，有两个后果需要了解：

1. 集群内 **HTTP** 可以直接工作（agent → tool、LLM）。无需配置。
2. 外部 **HTTPS**（例如 weather MCP tool 使用的公共 API `geocoding-api.open-meteo.com`）需要绕过 sidecar。将目标 hostname 添加到 **app** 容器上的 `NO_PROXY`。kagenti webhook 只有在容器中尚未存在时才设置 `HTTP_PROXY` / `HTTPS_PROXY` / `NO_PROXY`，因此在 workload 上预先声明 `NO_PROXY` 可以干净地覆盖默认值：

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weather-tool
  namespace: team1
spec:
  template:
    spec:
      containers:
      - name: mcp                  # the app container, not the sidecar
        env:
        - name: NO_PROXY
          value: "127.0.0.1,localhost,.svc,.svc.cluster.local,geocoding
-api.open-meteo.com,api.open-meteo.com"
```

其效果是：未被 `NO_PROXY` 匹配的任何主机上的出站调用仍会通过 AuthBridge sidecar 路由（并继续具备出站 token exchange、tracing 等能力）；而列出的公共 API 调用会绕过 sidecar，直接通过 pod 的默认 egress。流量是否真正能到达公共目标，由集群的 egress network policy 决定，而不是由 AuthBridge 决定。

> **INFO**
>
> 只添加你确实需要对外访问的主机。`NO_PROXY` 中的每一项都是出站检查路径中的一个漏洞——sidecar 将无法再在该 hostname 上注入 Bearer 令牌。

清理

```
kubectl delete namespace team1
```

这会删除 demo 命名空间及其中的所有资源。集群级别的 secure-profile 安装（SPIRE、Keycloak、Istio ambient、`Kagenti` operand）不会受到影响。

故障排查

现象	检查项
未认证请求被拒绝并返回 `auth.unauthorized`	这是预期行为——secure profile 要求有效的 Bearer 令牌。有关 client-credentials 的示例，请参见 步骤 4。
带 token 的请求返回 `401`	JWT `iss` claim 不匹配：token 是从与 `Kagenti` operand 上设置的 `keycloak.publicUrl` 不同的 Keycloak hostname 获取的。请从 AuthBridge 实际验证的相同 hostname 获取 token（通常是 `http://keycloak-service.keycloak.svc:8080`）。
Agent pod 卡在 `1/1` 而不是 `2/2`	sidecar 未注入。检查命名空间标签（`kagenti-enabled=true`）、`Kagenti` operand（`featureGates.globalEnabled: true`、`authbridgeConfig.enabled: true`），以及 `AgentRuntime` 是否已完成调谐（`kubectl describe agentruntime weather-agent-runtime -n team1`）。
命名空间中没有 `kagenti-keycloak-client-credentials-*` Secret	Keycloak client 注册尚未完成。检查 `kagenti-controller-manager` 日志中的 `ClientRegistrationReconciler` 错误，并确认 `keycloak-admin-secret` 存在于 `kagenti-system` 中。
tool 显示“weather service temporarily unavailable”	tool 的 `get_weather` 会调用 `geocoding-api.open-meteo.com`——这是一个公共 HTTPS endpoint。使用推荐模式（tool 上不带 sidecar）时，它会直接工作。如果你给 tool 注入了 sidecar，`HTTP_PROXY` 环境变量会把 HTTPS 路由到 AuthBridge，而它会用 `405` 拒绝 `CONNECT`；请改回无 sidecar 布局，或者将目标加入 `NO_PROXY`——参见 在 `proxy-sidecar` 模式下允许出站流量。

现象	检查项
AuthBridge 镜像出现 `ImagePullBackOff`	`Kagenti` operand 上的 `defaults.images` 指向了集群无法拉取的 registry。请将 AuthBridge sidecar 镜像迁移到集群可访问的 registry（见 启用 Secure Profile）。
`AgentRuntime` 上的 `MTLSReady=False`	确认 SPIRE 已安装，并且其 trust domain 与 operand 上的 `defaults.spiffe.trustDomain` 匹配。在 `proxy-sidecar` 模式下，即使没有 mTLS，sidecar 仍然可以验证 JWT——对 smoke test 来说，真正需要关注的是 runtime 上的 `Ready=True`。

Demo Advanced：带有 Token Exchange 的 Tool 上的 AuthBridge

带有 Secure Profile 的 Demo 只在 agent 上放置 AuthBridge —— tool 只是一个普通 workload，agent → tool 的跳转只是一次集群内原始 HTTP 调用。本指南是同一天气 demo 的高级 版本：AuthBridge 在 两侧 都会注入，每次 agent-to-tool 调用都会经过 **OAuth 2.0 RFC 8693 token exchange**，因此 tool 看到的 token 具有以 tool 为受众范围的 audience。

具体来说：

- **agent** 的 **AuthBridge** 会将出站主机与 `authproxy-routes` 规则匹配，把其入站 JWT 交换为一个新的 access token，该 token 的 `aud` claim 是 tool 的 Keycloak client id，并在出站请求上注入 `Authorization: Bearer <exchanged-token>`。
- **tool** 的 **AuthBridge** 会在 MCP server 看到请求之前验证这个新 token。任何 token 中不包含 tool audience 的请求都会在 tool 的 ingress 处被拒绝——即使它来自 mesh 内的其他 workload。
- **Keycloak** 通过在两个 client 上都启用 `standard.token.exchange.enabled: true`，以及通过一个 **client scope** 和其中的 token mapper 将 tool 的 audience 写入交换后 token 的 `aud` claim 来完成该 exchange。

我们在每个资源上都使用 `-advanced` 命名后缀，这样该 demo 就可以与同一 `team1` namespace 中的基础 secure-profile demo 共存。

目录

涉及的 ConfigMap

Keycloak client-id 约定

前提条件

第 1 步：部署带有 AuthBridge sidecar 的 tool

第 2 步：为 token exchange 配置 Keycloak

第 3 步：准备 agent 的出站规则

第 4 步：部署受保护的 agent

第 5 步：验证 token-exchange 路径

清理

故障排查

涉及的 **ConfigMap**

高级 demo 会触及 demo namespace 中两个与 AuthBridge 相关的 ConfigMap：

ConfigMap	管理者	目的
`authbridge-config`（以及每个 workload 对应的 `authbridge-config-<workload>`）	operator——你不需要创建它们。	基线 sidecar 配置：Keycloak URL / realm、issuer、mTLS 模式、plugin pipeline。
`authproxy-routes` — 仅包含一个 `routes.yaml` 键	你在下面的步骤 **3** 中创建它。	按主机定义的出站规则，告诉 agent 的 `token-exchange` plugin：*“当出站 `Host` header 匹配此模式时，将当前 JWT 交换为一个 `aud` claim 为该 audience 的 token，并请求这些 scope。”*

`authproxy-routes` 是唯一需要你显式执行 `kubectl apply` 的部分。

Keycloak client-id 约定

operator 的 `ClientRegistrationReconciler` 会使用短格式 `<namespace>/<workload>` 为每个 workload 在 Keycloak 中注册 client（本 demo 中分别是 `team1/weather-tool-`

`advanced` 和 `team1/weather-service-advanced`）。本指南中的每个 `target_audience` 以及 Keycloak client scope 上的每个 audience mapper 都采用这个短格式——这正是 tool 自己的入站 JWT-validation plugin 期望在交换后 token 的 `aud` claim 中看到的值。

前提条件

开始之前：

1. 前两个指南都已完成（或者其步骤已在原地重做）：
   - 安装 Secure-Profile 依赖项——SPIRE、Keycloak、Istio ambient。
   - 启用 Secure Profile—— `featureGates.globalEnabled: true`、`authbridgeConfig.enabled: true`、`keycloak.publicUrl` 已设置。
2. `Kagenti` **operand** 上的 `featureGates.injectTools: true`。 高级 demo 会在 tool 上注入 AuthBridge，而默认情况下 tool workload 的注入是关闭的。请先 patch 一次 operand，然后重启 controller 以让新 gate 生效：

   ```
   kubectl -n kagenti-system patch kagenti kagenti --type merge \
     -p '{"spec":{"featureGates":{"injectTools":true}}}'
   kubectl -n kagenti-system rollout restart deploy/kagenti-controller-man
   ager
   ```

3. demo namespace `team1` 已标记为 `kagenti-enabled=true`：

   ```
   kubectl create namespace team1 --dry-run=client -o yaml | kubectl apply
   -f -
   kubectl label namespace team1 kagenti-enabled=true --overwrite
   ```

4. 一个 提供 **OpenAI-compatible chat API** 的 **InferenceService**（本指南在 `models` 中使用 `qwen36-27b-gguf`）。
5. 具备访问集群的 `kubectl` 权限，并且可以读取 `keycloak` namespace 中的 `keycloak-initial-admin`。

本 demo 依赖的 client 在 Keycloak 中最终会使用如下名称——在整个指南中你会多次引用它们：

Workload	Keycloak client id	SPIFFE ID（供参考）
Agent	`team1/weather-service-advanced`	`spiffe://kagenti.local/ns/team1/sa/weather-service-advanced`
Tool	`team1/weather-tool-advanced`	`spiffe://kagenti.local/ns/team1/sa/weather-tool-advanced`

SPIFFE ID 这一列是 sidecar 通过 SPIRE 做 mTLS 时使用的——Alauda operator 不会把它当作 Keycloak 标识符。

第 1 步：部署带有 AuthBridge sidecar 的 tool

与基础 demo 不同，这个 tool workload 会拥有自己的 `AgentRuntime`，因此 AuthBridge webhook 会注入 sidecar，operator 也会为 tool 的 ServiceAccount 注册一个 Keycloak client。请不要在 Deployment 上手动设置 `kagenti.io/type: tool`——`ValidatingAdmissionPolicy` 会把这个 label 保留给 operator，operator 会根据你创建的 `AgentRuntime` 来应用它。

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: weather-tool-advanced
  namespace: team1
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weather-tool-advanced
  namespace: team1
  labels:
    app.kubernetes.io/name: weather-tool-advanced
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: weather-tool-advanced
  template:
    metadata:
      labels:
        app.kubernetes.io/name: weather-tool-advanced
    spec:
      serviceAccountName: weather-tool-advanced (1)
      containers:
      - name: mcp
        image: docker.io/alaudadockerhub/weather_tool:v0.1.0-rc.1
        imagePullPolicy: IfNotPresent
        env:
        - name: PORT
          value: "8000"
        - name: HOST
          value: 0.0.0.0
        - name: UV_CACHE_DIR
          value: /app/.cache/uv
        - name: NO_PROXY (2)
          value: "127.0.0.1,localhost,.svc,.svc.cluster.local,geocoding-a
pi.open-meteo.com,api.open-meteo.com"
        ports:
        - containerPort: 8000
        volumeMounts:
        - mountPath: /app/.cache
```

```
          name: cache
      volumes:
      - name: cache
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: weather-tool-advanced-mcp
  namespace: team1
spec:
  selector:
    app.kubernetes.io/name: weather-tool-advanced
  ports:
  - name: http
    port: 8000
    targetPort: 8000
---
apiVersion: agent.kagenti.dev/v1alpha1
kind: AgentRuntime
metadata:
  name: weather-tool-advanced-runtime
  namespace: team1
spec:
  type: tool (3)
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: weather-tool-advanced
EOF
```

(1) `serviceAccountName: weather-tool-advanced` —— SPIRE 会签发一个以 `spiffe://<trustDomain>/ns/team1/sa/weather-tool-advanced` 为键的 X.509 SVID，operator 会将 Keycloak client id 注册为 `team1/weather-tool-advanced`。

(2) `NO_PROXY` 覆盖了 tool 对公共地理编码 API 的出站调用。webhook 会尊重预先声明的 `NO_PROXY`，因此这些请求会绕过 tool 自己的 forward-proxy sidecar（参见 允许在 `proxy-sidecar` 模式下的出站流量）。

(3) `AgentRuntime` 上的 `spec.type: tool` —— operator 会应用 `kagenti.io/type: tool` label，注入 AuthBridge sidecar（由 `featureGates.injectTools: true` 控制），并为 tool 注册 Keycloak client。

等待 sidecar 注入完成并且 credentials Secret 出现：

```
kubectl rollout status deploy/weather-tool-advanced -n team1 --timeout=18
0s
kubectl get pod -n team1 -l app.kubernetes.io/name=weather-tool-advanced
# NAME                                        READY   STATUS    RESTARTS
AGE
# weather-tool-advanced-XXXXXXXXXX-YYYYY      2/2     Running   0
1m

# The operator records the Secret name on the Deployment annotation
kubectl -n team1 get deploy weather-tool-advanced \
  -o jsonpath='{.spec.template.metadata.annotations.kagenti\.io/keycloak-
client-credentials-secret-name}'; echo
# kagenti-keycloak-client-credentials-<hash-tool>
```

`READY 2/2` 表示 tool 现在已经拥有一个 AuthBridge sidecar 和 `mcp` 应用容器。

第 2 步：为 token exchange 配置 Keycloak

operator 已经为 tool 创建了一个 Keycloak client（clientId 为 `team1/weather-tool-advanced`）。我们仍然需要进行三项 admin 更改：

1. 创建一个 **client scope** `weather-tool-advanced-aud`，并配置一个 **audience mapper**，把 `team1/weather-tool-advanced` 追加到交换后 token 的 `aud` claim 中。
2. 在 tool client 上启用 `standard.token.exchange.enabled: true`。
3. 将该 scope 作为 tool client 的一个 **optional** client scope，这样请求它的 token-exchange 才能成功。

我们通过一个临时 pod 来完成这三项操作；该 pod 会挂载平台的 `keycloak-initial-admin` Secret。请先把这个 Secret 复制到 demo namespace（因为 pod 只能挂载同 namespace 中的 Secret）：

```
kubectl -n keycloak get secret keycloak-initial-admin -o yaml \
  | sed 's/namespace: keycloak/namespace: team1/' \
  | kubectl apply -f -
```

然后运行 setup Job：

```
kubectl apply -f - <<'EOF'
apiVersion: batch/v1
kind: Job
metadata:
  name: keycloak-setup-advanced-tool
  namespace: team1
spec:
  ttlSecondsAfterFinished: 300
  template:
    spec:
      restartPolicy: OnFailure
      containers:
      - name: setup
        image: docker.io/alaudadockerhub/curl:8.1.2
        command: ["/bin/sh", "-c"]
        args:
        - |
          set -eu
          KC=http://keycloak-service.keycloak.svc:8080
          USER=$(cat /kc/username)
          PASS=$(cat /kc/password)
          T=$(curl -s -X POST $KC/realms/master/protocol/openid-connect/t
oken \
                -d grant_type=password -d client_id=admin-cli \
                --data-urlencode "username=$USER" --data-urlencode "passw
ord=$PASS" \
                | sed 's/.*"access_token":"\([^"]*\)".*/\1/')
          AUTH="Authorization: Bearer $T"
          echo "==> create client scope weather-tool-advanced-aud"
          curl -sS -o /dev/null -w 'scope-status=%{http_code}\n' -X POST
\
            "$KC/admin/realms/kagenti/client-scopes" -H "$AUTH" -H 'Conte
nt-Type: application/json' -d '{
              "name":"weather-tool-advanced-aud",
              "protocol":"openid-connect",
              "attributes":{"include.in.token.scope":"true","display.on.c
onsent.screen":"true"},
              "protocolMappers":[{
                "name":"weather-tool-advanced-aud-mapper",
                "protocol":"openid-connect",
                "protocolMapper":"oidc-audience-mapper",
                "consentRequired":false,
                "config":{
```

```
                  "included.custom.audience":"team1/weather-tool-advance
d",
                  "id.token.claim":"false",
                  "access.token.claim":"true",
                  "userinfo.token.claim":"false"
                }
              }]
            }' || true
          SCOPE_ID=$(curl -s "$KC/admin/realms/kagenti/client-scopes" -H
"$AUTH" \
                       | tr '}' '\n' | grep 'weather-tool-advanced-aud' \
                       | grep -o '"id":"[^"]*"' | head -1 | sed 's/"i
d":"\([^"]*\)"/\1/')
          echo "scope_id=$SCOPE_ID"
          TOOL_ID=$(curl -s "$KC/admin/realms/kagenti/clients?clientId=te
am1%2Fweather-tool-advanced" -H "$AUTH" \
                      | grep -o '"id":"[^"]*"' | head -1 | sed 's/"id":"\
([^"]*\)"/\1/')
          echo "tool_id=$TOOL_ID"
          echo "==> enable token exchange on tool client"
          curl -sS -o /dev/null -w 'te-status=%{http_code}\n' -X PUT \
            "$KC/admin/realms/kagenti/clients/$TOOL_ID" -H "$AUTH" -H 'Co
ntent-Type: application/json' \
            -d '{"attributes":{"standard.token.exchange.enabled":"tru
e"}}'
          echo "==> add scope as optional on tool client"
          curl -sS -o /dev/null -w 'opt-status=%{http_code}\n' -X PUT \
            "$KC/admin/realms/kagenti/clients/$TOOL_ID/optional-client-sc
opes/$SCOPE_ID" -H "$AUTH"
        volumeMounts:
        - name: kc-admin
          mountPath: /kc
          readOnly: true
      volumes:
      - name: kc-admin
        secret:
          secretName: keycloak-initial-admin
EOF

kubectl -n team1 wait --for=condition=Complete job/keycloak-setup-advance
d-tool --timeout=120s
kubectl -n team1 logs job/keycloak-setup-advanced-tool
```

日志末尾的预期输出：`scope-status=201`、`te-status=204`、`opt-status=204`。

第 3 步：准备 agent 的出站规则

agent 的 AuthBridge 会从同一 namespace 中的 `authproxy-routes` ConfigMap 读取出站规则。每一项都在说明：*“当出站请求的 `Host` header 与此模式匹配时，将入站 JWT 交换为一个 `aud` claim 为该 audience 的 token，并请求这些 scope。”*

请在部署 agent 之前先应用该规则，这样 sidecar 在首次启动时就能读取到它：

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: authproxy-routes
  namespace: team1
data:
  routes.yaml: |
    - host: "weather-tool-advanced-mcp.team1.svc.cluster.local"   # [!cod
e callout]
      target_audience: "team1/weather-tool-advanced"              # [!cod
e callout]
      token_scopes: "openid weather-tool-advanced-aud"
EOF
```

1 `host` 会与 HTTP `Host` header 精确 匹配——不是短 service 名称。router 使用 `gobwas/glob`，并以 `.` 作为段分隔符，因此 `*` 不会跨越点。请使用完整的集群内 FQDN（不包含端口；router 在匹配前会去掉端口），或者显式列出你需要的模式。单独的 **`weather-tool-advanced-mcp`** 在这里并不匹配 **`weather-tool-advanced-mcp.team1.svc.cluster.local`**。

2 `target_audience` 是 tool 的 Keycloak clientId（Alauda 的 `<ns>/<workload>` 形式），也就是 tool 自身的入站 JWT-validation plugin 期望在交换后 token 的 `aud` 中看到的内容。

如果没有匹配的 route，agent 的 AuthBridge 会退回到 `default_policy: passthrough`——token 会被原样转发，其 `aud` claim 仍然是 `team1/weather-service-advanced`，而 tool 的 ingress 在严格配置下会拒绝它。

> **routes.yaml 在 Configure 时仅加载一次**

`token-exchange` plugin 会在 sidecar 启动时读取 `/etc/authproxy/routes.yaml`。修改 ConfigMap 不会 立即影响已经运行的 pod，除非它们重启并且 kubelet 已将新内容同步到挂载中（约 60 秒）。当你更改 routes 时，请应用 ConfigMap，等待约 60 秒，然后对受影响的 workload 执行 `kubectl delete pod`。仅执行 `rollout restart` 可能会与 kubelet 的同步发生竞态，使新 pod 仍然读取旧文件。

第 4 步：部署受保护的 agent

与基础 secure demo 的结构相同，但使用 `-advanced` 命名，并指向高级 tool 的 Service。`MCP_URL` 和 route 的 `host` 字段必须一致——两者都必须是完整的集群内 FQDN。

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: weather-service-advanced
  namespace: team1
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: weather-service-advanced
  namespace: team1
  labels:
    app.kubernetes.io/name: weather-service-advanced
    protocol.kagenti.io/a2a: ""
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: weather-service-advanced
  template:
    metadata:
      labels:
        app.kubernetes.io/name: weather-service-advanced
    spec:
      serviceAccountName: weather-service-advanced
      containers:
      - name: agent
        image: docker.io/alaudadockerhub/weather_service:v0.1.0-rc.1
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8000
        env:
        - name: PORT
          value: "8000"
        - name: UV_CACHE_DIR
          value: /app/.cache/uv
        - name: MCP_URL
          value: http://weather-tool-advanced-mcp.team1.svc.cluster.loca
l:8000/mcp
        - name: LLM_API_BASE
          value: http://qwen36-27b-gguf-predictor.models.svc.cluster.loca
l/v1
```

```
        - name: LLM_API_KEY
          value: dummy
        - name: LLM_MODEL
          value: qwen36-27b-gguf
---
apiVersion: v1
kind: Service
metadata:
  name: weather-service-advanced
  namespace: team1
spec:
  selector:
    app.kubernetes.io/name: weather-service-advanced
  ports:
  - name: http
    port: 8000
    targetPort: 8000
---
apiVersion: agent.kagenti.dev/v1alpha1
kind: AgentRuntime
metadata:
  name: weather-service-advanced-runtime
  namespace: team1
spec:
  type: agent
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: weather-service-advanced
EOF

kubectl rollout status deploy/weather-service-advanced -n team1 --timeout
=180s
kubectl get pods -n team1 -l app.kubernetes.io/name=weather-service-advan
ced
# NAME                                      READY   STATUS    RESTART
S   AGE
# weather-service-advanced-XXXXXXXXX-YYYYY      2/2     Running   0
1m
```

一旦 agent pod 就绪并且其 Keycloak client 已注册，请再运行一个第二个 setup Job，在 agent client 上启用 token exchange，并把同一个 scope 也作为 optional 分配给它。这个 Job 是第 2 步的两行变体：

```
kubectl apply -f - <<'EOF'
apiVersion: batch/v1
kind: Job
metadata:
  name: keycloak-setup-advanced-agent
  namespace: team1
spec:
  ttlSecondsAfterFinished: 300
  template:
    spec:
      restartPolicy: OnFailure
      containers:
      - name: setup
        image: docker.io/alaudadockerhub/curl:8.1.2
        command: ["/bin/sh", "-c"]
        args:
        - |
          set -eu
          KC=http://keycloak-service.keycloak.svc:8080
          USER=$(cat /kc/username)
          PASS=$(cat /kc/password)
          T=$(curl -s -X POST $KC/realms/master/protocol/openid-connect/t
oken \
                -d grant_type=password -d client_id=admin-cli \
                --data-urlencode "username=$USER" --data-urlencode "passw
ord=$PASS" \
                | sed 's/.*"access_token":"\([^"]*\)".*/\1/')
          AUTH="Authorization: Bearer $T"
          SCOPE_ID=$(curl -s "$KC/admin/realms/kagenti/client-scopes" -H
"$AUTH" \
                       | tr '}' '\n' | grep 'weather-tool-advanced-aud' \
                       | grep -o '"id":"[^"]*"' | head -1 | sed 's/"i
d":"\([^"]*\)"/\1/')
          AGENT_ID=$(curl -s "$KC/admin/realms/kagenti/clients?clientId=t
eam1%2Fweather-service-advanced" -H "$AUTH" \
                       | grep -o '"id":"[^"]*"' | head -1 | sed 's/"i
d":"\([^"]*\)"/\1/')
          echo "agent_id=$AGENT_ID scope_id=$SCOPE_ID"
          curl -sS -o /dev/null -w 'te-status=%{http_code}\n' -X PUT \
            "$KC/admin/realms/kagenti/clients/$AGENT_ID" -H "$AUTH" -H 'C
ontent-Type: application/json' \
            -d '{"attributes":{"standard.token.exchange.enabled":"tru
e"}}'
```

```
          curl -sS -o /dev/null -w 'opt-status=%{http_code}\n' -X PUT \
            "$KC/admin/realms/kagenti/clients/$AGENT_ID/optional-client-s
copes/$SCOPE_ID" -H "$AUTH"
        volumeMounts:
        - name: kc-admin
          mountPath: /kc
          readOnly: true
      volumes:
      - name: kc-admin
        secret:
          secretName: keycloak-initial-admin
EOF

kubectl -n team1 wait --for=condition=Complete job/keycloak-setup-advance
d-agent --timeout=120s
kubectl -n team1 logs job/keycloak-setup-advanced-agent
```

预期结果：`te-status=204` 和 `opt-status=204` 都会出现。

第 5 步：验证 token-exchange 路径

向 agent 发送一个已认证的 A2A `message/send` 请求，方式与 基础 secure demo 的第 4 步 完全相同。你挂载的 credentials Secret 是 **agent** 的——交换会在 agent 的 AuthBridge 内部透明完成。

从 Deployment 上的 operator annotation 中找到 agent 的 credentials Secret：

```
AGENT_SECRET=$(kubectl -n team1 get deploy weather-service-advanced \
  -o jsonpath='{.spec.template.metadata.annotations.kagenti\.io/keycloak-
client-credentials-secret-name}')
echo "$AGENT_SECRET"
# kagenti-keycloak-client-credentials-<agent-hash>
```

从 pod 中运行已认证查询（这里的 shell 使用上面得到的 `$AGENT_SECRET` 值）：

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: adv-smoke
  namespace: team1
spec:
  restartPolicy: Never
  containers:
  - name: query
    image: docker.io/alaudadockerhub/curl:8.1.2
    command: ["/bin/sh", "-c"]
    args:
    - |
      set -e
      CID=\$(cat /creds/client-id.txt)
      CSEC=\$(cat /creds/client-secret.txt)

      TOKEN=\$(curl -s \\
        http://keycloak-service.keycloak.svc:8080/realms/kagenti/protocol/openid-connect/token \\
        -d grant_type=client_credentials \\
        --data-urlencode "client_id=\$CID" \\
        --data-urlencode "client_secret=\$CSEC" \\
        | sed 's/.*"access_token":"\\([^"]*\\)".*/\\1/')

      curl -sS -X POST http://weather-service-advanced.team1.svc.cluster.local:8000/ \\
        -H "Authorization: Bearer \$TOKEN" \\
        -H "Content-Type: application/json" \\
        -d '{"jsonrpc":"2.0","id":"1","method":"message/send","params":{"message":{"role":"user","parts":[{"kind":"text","text":"What is the weather in Berlin?"}],"messageId":"m1"}}}'
    volumeMounts:
    - name: creds
      mountPath: /creds
      readOnly: true
  volumes:
  - name: creds
    secret:
      secretName: $AGENT_SECRET
EOF
```

```
kubectl -n team1 wait --for=condition=Ready pod/adv-smoke --timeout=60s
kubectl -n team1 logs adv-smoke
kubectl -n team1 delete pod adv-smoke
```

你应该会得到一个带有真实天气数据的已完成 A2A task。请求路径如下：

```
client —(token A, aud=team1/weather-service-advanced)—▶ agent AuthBridge
                                                    | validates token A (inbound)
                                                    ▼
                                                weather-service-advanced
                                                    | MCP call to weather-tool-ad
vanced-mcp
                                                    ▼
                                                agent AuthBridge outbound
                                                    | matches route → RFC 8693 ex
change
                                                    | obtains token B (aud=team1/
weather-tool-advanced)
                                                    ▼
                                                tool AuthBridge
                                                    | validates token B against i
ts own audience
                                                    ▼
                                                weather-tool-advanced (mcp)
                                                    | HTTPS to geocoding-api.open
-meteo.com (NO_PROXY)
                                                    ▼
                                                response
```

在 agent-sidecar 日志中确认 exchange 已发生：

```
kubectl -n team1 logs -c authbridge-proxy \
  -l app.kubernetes.io/name=weather-service-advanced --tail=100 \
  | grep -E 'outbound token exchanged|outbound passthrough'
# time=... level=INFO msg="outbound token exchanged" host=weather-tool-ad
vanced-mcp.team1.svc.cluster.local:8000 audience=team1/weather-tool-advan
ced
```

再确认 tool 侧的入站验证：

```
kubectl -n team1 logs -c authbridge-proxy \
  -l app.kubernetes.io/name=weather-tool-advanced --tail=100 \
  | grep -E 'inbound authorized|inbound rejected'
# time=... level=INFO msg="inbound authorized" subject=<uuid> clientID=te
am1/weather-service-advanced
```

agent 侧的 `"outbound token exchanged"` 加上 tool 侧的 `"inbound authorized"`，就是 RFC 8693 流程端到端接通的两行证明。

清理

```
kubectl delete namespace team1
```

如果你之后不再打算保护 tool workload，也可以把 `Kagenti` operand 上的 `featureGates.injectTools` 恢复为 `false`：

```
kubectl -n kagenti-system patch kagenti kagenti --type merge \
  -p '{"spec":{"featureGates":{"injectTools":false}}}'
```

故障排查

症状	可能原因	解决方法
`Deployment ... forbidden: ValidatingAdmissionPolicy 'agent-label-protection' denied`	你手动在 Deployment 上设置了 `kagenti.io/type: tool`。	从 Deployment 中移除该 operator 会根据你创建的 `AgentRuntime` 来应用它
Tool pod 一直停留在 `1/1`	`featureGates.injectTools` 仍然是 `false`，或者在 patch 之后没有重启 controller-manager。	重新执行前提条件中的两 后删除 tool pod。
Agent-sidecar 日志显示 `outbound passthrough …`	route 的 `host` 模式与出站 `Host` header 不匹配。常见陷阱：短 service	使用完全一致的 FQDN（ `weather-tool-advance`

症状	可能原因	解决方法
`reason="no matching route"`	名称去匹配 FQDN Host，或者对带点的主机名使用 `*` 通配符（`.` 是 glob 分隔符）。	`mcp.team1.svc.cluster` 然后删除 agent pod 以便 `/etc/authproxy/route`
已编辑 Route ConfigMap，但没有任何变化	`routes.yaml` 在 Configure 时读取；ConfigMap 挂载的 kubelet 同步也需要约 60 秒。	应用 ConfigMap，等待约 后对 agent 执行 `kubect` `pod`，让新 pod 的 Confi 取已同步的文件。
Agent 请求返回 `503`，但 agent 侧缺少 exchange 日志行	Token exchange 失败——通常是因为请求的 scope 没有分配给 agent client，或者没有设置 `standard.token.exchange.enabled: true`。	在 agent pod 就绪后重新 的 Job。
Agent 请求从 tool 返回 `401`，且 agent 侧有匹配的 `outbound token exchanged`	tool 的入站 audience 验证不匹配。确认 audience mapper 写入的是 `team1/weather-tool-advanced`（而不是 SPIFFE URI），并且 tool client 已将该 scope 配置为 optional。	重新运行第 2 步的 Job。
Tool 显示 “weather service temporarily unavailable”	tool 到 `geocoding-api.open-meteo.com` 的出站 HTTPS 正在经过 tool 自己的 AuthBridge forward proxy（仅支持 HTTP），因此收到 `CONNECT 405`。	上面的 manifest 已经为 置了 `NO_PROXY`。如果你 了，请重新应用——参见 `proxy-sidecar` 模式下 量。
任意位置设置了 `kagenti.io/client-registration-inject: "true"`	这是早期 kagenti 版本遗留的 label；在当前版本中，它会悄悄禁用 operator 管理的 registration。	从所有 manifest 中移除它

Envoy AI Gateway 后端包含两个 MCP server 的 Kagenti

本示例部署 Rosso 天气和餐厅预订 MCP server，将它们聚合到一个 Envoy Gateway 数据平面之后，并将一个 Kagenti 通用智能体连接到单一的 `/mcp` endpoint。

这些 manifest 位于 `docs/en/ai_applications/components/kagenti/assets/envoy_ai_gateway_mcp`。它们面向 `kagenti-mcp-demo` namespace。如果你的环境使用不同的值，请在应用之前修改 namespace 和 `GatewayClass` 名称。

目录

已验证版本

拓扑和 HTTPRoute 数量

来源和镜像

部署 gateway 和两个 MCP server

测试统一的 MCP endpoint

部署 Kagenti 智能体

清理

已验证版本

本示例于 2026-07-28 在以下环境中完成验证：

组件	版本
Alauda Build of Envoy Gateway	`1.7.0-build.20260625161000`
Alauda Build of Envoy AI Gateway	`0.6.0-1`
`MCPRoute` API	`aigateway.envoyproxy.io/v1beta1`

Envoy AI Gateway 0.6 通过 `MCPRoute` 支持 MCP 聚合、认证、工具授权以及上游凭据。

Kuadrant `mcp-gateway` 与此 API 不可互换。它的 controller 会创建一个 Istio `EnvoyFilter`，并且在 `MCPGatewayExtension` 变为 ready 之前需要该资源。Envoy Gateway 的原生 `EnvoyExtensionPolicy` 不暴露 Kuadrant MCP Gateway 目前用于重写 `:authority` 所需的 `mutation_rules.allow_all_routing` 设置。请在 Envoy Gateway 中使用此处展示的原生 Envoy AI Gateway `MCPRoute`，或者使用 Kuadrant MCP Gateway 及其受支持的 Istio provider。

拓扑和 **HTTPRoute** 数量

```
Kagenti generic agent
        |
        | one authenticated MCP URL
        v
Envoy Gateway :80 /mcp
        |
        | Envoy AI Gateway MCP proxy
        +-----------------------+
        |                       |
        v                       v
weather:8000/mcp       reservation:8000/mcp
```

只需使用两个 `backendRefs` 编写一个 `MCPRoute`；不要手动编写 backend `HTTPRoute`。controller 会准确创建两个 backend 路由和一个必需的面向客户端的路由：

- `ai-eg-mcp-br-kagenti-tools-weather`

- `ai-eg-mcp-br-kagenti-tools-reservation`
- `ai-eg-mcp-main-kagenti-tools`

前两个是请求的按 server 划分的 `HTTPRoute`。第三个暴露统一的 `/mcp` endpoint。三者都归 `MCPRoute` 所有，且不得直接编辑。

来源和镜像

这些 workload 来自 `rossoctl/examples` ↗ 的提交 `ec17c73235c8ef291a6d4aaa346b50a4fba35e72`：

- `mcp/weather_tool`
- `mcp/reservation_tool`
- `a2a/generic_agent`

YAML 固定了发布在 `ghcr.io/rossoctl/examples` 的不可变多架构 image digest。

部署 gateway 和两个 MCP server

如果示例 namespace 不存在，请创建它：

```
kubectl create namespace kagenti-mcp-demo --dry-run=client -o yaml | kube
ctl apply -f -
```

在将此示例用于一次性开发环境之外的场景前，请先替换 `03-mcp-route-auth-rate-limit.yaml` 中的 `replace-this-demo-key`。然后应用 server、gateway 和 policy manifest：

```
EXAMPLE_DIR=docs/en/ai_applications/components/kagenti/assets/envoy_ai_ga
teway_mcp

kubectl apply -f "$EXAMPLE_DIR/01-mcp-servers.yaml"
kubectl apply -f "$EXAMPLE_DIR/02-gateway.yaml"
kubectl apply -f "$EXAMPLE_DIR/03-mcp-route-auth-rate-limit.yaml"
```

该示例使用 API key 认证，并为每个已认证 client ID 设置每分钟 30 次请求的本地限流。`sanitize: true` 会在请求发送到上游之前移除 API key。生成的 client ID 会作为 `x-mcp-client-id` 转发，这也是限流 key。

验证结果：

```
kubectl -n kagenti-mcp-demo rollout status deploy/weather-mcp
kubectl -n kagenti-mcp-demo rollout status deploy/reservation-mcp
kubectl -n kagenti-mcp-demo get gateway kagenti-mcp
kubectl -n kagenti-mcp-demo get mcproute kagenti-tools
kubectl -n kagenti-mcp-demo get httproute
```

预期条件为 `Gateway Programmed=True`、`MCPRoute Accepted`，以及所有三个生成的 `HTTPRoute` 上的 `Accepted=True`。

测试统一的 **MCP endpoint**

转发由 `02-gateway.yaml` 创建的稳定 Envoy Service：

```
kubectl -n envoy-gateway-system \
  port-forward service/kagenti-mcp-envoy 8080:80
```

在另一个终端中初始化会话。省略 API key 必须返回 `401`；提供 API key 必须返回 MCP 响应和 `mcp-session-id` header。

```
MCP_URL='http://127.0.0.1:8080/mcp?api-key=replace-this-demo-key'

curl -i "$MCP_URL" \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json, text/event-stream' \
  --data '{"jsonrpc":"2.0","id":1,"method":"initialize","params":{"protoc
olVersion":"2025-06-18","capabilities":{},"clientInfo":{"name":"curl","ve
rsion":"1.0"}}}'
```

使用返回的 session ID 列出工具：

```
SESSION_ID='<mcp-session-id>'

curl -sS "$MCP_URL" \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json, text/event-stream' \
  -H "mcp-session-id: $SESSION_ID" \
  --data '{"jsonrpc":"2.0","method":"notifications/initialized"}'

curl -sS "$MCP_URL" \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json, text/event-stream' \
  -H "mcp-session-id: $SESSION_ID" \
  --data '{"jsonrpc":"2.0","id":2,"method":"tools/list"}' \
  | sed -n 's/^data: //p' | jq
```

工具名称会以前端 backend 作为前缀，例如 `weather__get_weather` 和 `reservation__search_restaurants`。

部署 **Kagenti** 智能体

创建带有 OpenAI-compatible endpoint 的 LLM 配置 Secret：

```
kubectl -n kagenti-mcp-demo create secret generic generic-agent-llm \
  --from-literal=api-base='https://your-openai-compatible-endpoint/v1' \
  --from-literal=api-key='replace-me' \
  --from-literal=model='replace-me'

kubectl apply -f docs/en/ai_applications/components/kagenti/assets/envoy_
ai_gateway_mcp/04-agent.yaml

kubectl -n kagenti-mcp-demo get agentruntime generic-agent
kubectl -n kagenti-mcp-demo rollout status deploy/generic-agent
```

`MCP_URLS` 只包含统一的 Envoy endpoint，因此智能体通过一次 MCP 连接即可发现两组工具。当前通用智能体仅暴露 URL 配置，而不支持按 server 设置请求 header；因此本示例使用 API key query parameter。智能体会在启动期间记录 `MCP_URLS`，所以仅应在开发环境中使用此方式。对于生产环境，请为智能体添加基于 header 的 MCP client 配置，或使用带有 OAuth/JWT 的 Kagenti AuthBridge，并将 `MCPRoute` 切换为 `securityPolicy.oauth`。

可以尝试如下提示：

- `What is the weather in Taipei?`
- `Find Italian restaurants in Boston.`

清理

生成的 `HTTPRoute` 会随着 `MCPRoute` 一并删除：

```
EXAMPLE_DIR=docs/en/ai_applications/components/kagenti/assets/envoy_ai_gateway_mcp

kubectl delete -f "$EXAMPLE_DIR/04-agent.yaml" --ignore-not-found
kubectl delete -f "$EXAMPLE_DIR/03-mcp-route-auth-rate-limit.yaml"
kubectl delete -f "$EXAMPLE_DIR/02-gateway.yaml"
kubectl delete -f "$EXAMPLE_DIR/01-mcp-servers.yaml"
kubectl -n kagenti-mcp-demo delete secret generic-agent-llm --ignore-not-found
```

组件

本章所依赖组件的介绍和安装指南。

Dify

Dify

Dify

核心概念

文档

安装 Dify

概览

步骤 1 — 发布 Operator 软件包

步骤 2 — 安装 Dify Operator

步骤 3 — 准备外部依赖

步骤 4 — 确定并设置访问方式

步骤 5 — 提供插件安装代理 *(仅限受限网络)*

步骤 6 — 创建 Dify 实例

步骤 7 — 登录

参考：Dify 实例字段

主要特性

LLM 原生应用开

RAG（检索增强

Workflow 与 Ag

API 与集成

Llama Stack

Alauda 的 Llama Stack 构建

Llama Stack

安装 Llama Stack

上传 Operator 包

快速入门

前提条件

文档

启用 Llama Stack

验证安装

部署 Llama Stack Server

快速入门示例

向量存储使用

常见问题

主要特性

基于 Server 的架构

使用工具的 AI 智能体

配置与可扩展性

集成

Kagenti

Kagenti Operator

概述

关键特性

核心配置

使用场景

安装 Kagenti

前提条件

上架 operator 软件包

安装 operator

Profile：Core 与 Secure

安全架构

概览

工作负载身份（

传输安全（Istio

授权（OPA + A

AuthBridge side

AgentRuntime

启用安全 profile

使用场景

相关内容

MCP 生命周期 Operator

Alauda Build 的 MCP Lifecycle

主要特性

工作原理

使用场景

为什么使用 MCPServer

文档

安装 MCP Lifecycle Operator

上传 operator 软件包

安装 operator

验证安装

升级 operator

快速入门

先决条件

创建 MCPServe

验证 MCPServe

测试端点

更新端口或路径

常见配置

清理

Dify

Dify

Dify

核心概念

文档

安装 Dify

概览

步骤 1 — 发布 Operator 软件包

步骤 2 — 安装 Dify Operator

步骤 3 — 准备外部依赖

步骤 4 — 确定并设置访问方式

步骤 5 — 提供插件安装代理 *(仅限受限网络)*

步骤 6 — 创建 Dify 实例

步骤 7 — 登录

参考：Dify 实例字段

主要特性

LLM 原生应用开

RAG（检索增强

Workflow 与 Ag

API 与集成

Dify

目录

Dify
核心概念
文档
官方文档

Dify

Dify 是一个开源的 LLM 原生应用开发平台，为 AI 工作流、RAG 流水线、智能体能力和模型管理提供直观的界面。它使开发者和非技术用户能够通过 Web UI 构建、部署和管理 AI 应用。

Dify 帮助团队快速构建和迭代由 LLM 驱动的应用：从简单的聊天机器人（Assistant）和文本生成器，到复杂的 RAG 流水线和多步骤智能体工作流。有关部署组件和前提条件（API、Worker、Web、PostgreSQL、Redis、向量存储等），请参阅 安装 Dify。

核心概念

- 原生应用 – 用于组织 LLM 能力的基本单元。类型包括 Assistant（聊天）、Text Generator、Agent（推理和工具）以及 Workflow/Chatflow（可视化流水线）。每个应用都可以作为 Web App 发布，或通过 API 使用。

- **Workflow** – 基于节点的可视化流水线（LLM、检索、代码、逻辑节点），用于通过拖放方式构建多步骤 AI 逻辑。
- 知识库 **(RAG)** – 由文档或 URL 构建的数据集，包含分块、嵌入和检索，用于让 LLM 响应基于你的数据。（在 Helm chart 中，**pgvector** 是支持的向量存储。）
- **Agent** – 使用工具和多步骤推理的应用；可部署为 Web App 或 API。

有关主要功能的简要列表，请参阅 主要功能。有关平台上的安装和配置，请参阅 安装 Dify。

文档

Dify 提供了全面的官方文档，帮助用户理解和使用该平台：

官方文档

- 主要文档: https://docs.dify.ai/↗
  - 核心概念、快速开始和部署指南
  - Workflow 设计、RAG 和智能体开发
  - API 参考、SDK 使用和最佳实践
  - 插件开发和 Marketplace: 安装插件↗

安装 Dify

本指南使用 **Dify Operator**（基于 helm 的 OLM operator）在 **Alauda Container Platform** 上安装 Dify。请按顺序执行各步骤——每一步都依赖于上一步。

关于 Dify 的功能以及它会部署哪些组件，请参见 简介。

目录

概览

步骤 1 — 发布 Operator 软件包

步骤 2 — 安装 Dify Operator

步骤 3 — 准备外部依赖

PostgreSQL（必需）

Redis（必需 — 单机或 Sentinel）

pgvector — 向量存储 *(可选，用于 RAG)*

存储 — RWX PVC 或 S3

步骤 4 — 确定并设置访问方式

设置 Gateway（仅限选项 B 或 C）

1. 在 ACP 中注册域名

2. 安装 Envoy Gateway

3. 创建 `Gateway` 资源

步骤 5 — 提供插件安装代理 *(仅限受限网络)*

步骤 6 — 创建 Dify 实例

创建连接 Secret

填写表单

步骤 7 — 登录

参考：Dify 实例字段

概览

你需要	原因
Dify operator 软件包	先下载并按照通用软件包操作步骤上架，以便该 operator 显示在 OperatorHub 中（参见 上架软件包）
外部 **PostgreSQL 12+**	Dify 的主数据库（仅支持 PostgreSQL）
外部 **Redis 6+**（单机或 Sentinel）	缓存 + Celery broker；不支持 Cluster 模式
外部 **pgvector** *(可选)*	用于 RAG 的向量存储；如果不使用 RAG，请禁用
RWX PVC *或* **S3-compatible** 存储	为 API + Plugin Daemon 提供共享、可写存储
浏览器访问所需的 访问方式	以下之一：NodePort/Service、单域名或双域名。域名和 TLS 证书通过 ACP 网络管理 注册。参见 步骤 4
集群内代理 *(仅限受限网络)*	Marketplace 反向代理 + PyPI 镜像，用于离线安装插件

步骤 1 — 发布 **Operator** 软件包

从 Customer Support / Portal 下载 Dify operator 软件包，然后按照 上架软件包 中的说明上传该软件包，使其显示在 **Marketplace / OperatorHub** 中。

步骤 2 — 安装 **Dify Operator**

在 Alauda Container Platform 的 管理员 视图中：

1. 打开 **Marketplace / OperatorHub** 并选择目标集群。
2. 搜索 **Dify**，单击 安装，除非需要非默认命名空间，否则保持默认值不变。
3. 等待卡片显示 已安装。

步骤 3 — 准备外部依赖

PostgreSQL（必需）

使用 **PostgreSQL 12+**。最简单的方式是在 **Data Services** 中使用 PostgreSQL operator：先创建一个集群，然后为 Dify 创建一个空数据库（例如 `dify`）。记录主机、端口、用户名和密码。

Redis（必需 — 单机或 Sentinel）

使用 **Redis 6.0+**。通过 **Data Services** 创建。

- 单机： 使用 **Redis Sentinel** 架构创建，切换到 **YAML**，设置 `spec.arch: standalone`，然后创建。将 `rfr-<instance>-read-write` Service 作为 Redis 主机。
- **Sentinel（HA）**： 保持 Sentinel 架构；在 步骤 6 中使用 Sentinel Service 端点 + 主节点名称。

pgvector — 向量存储 *(可选，用于 RAG)*

仅支持 **pgvector**。请使用安装了 `pgvector` 扩展的 PostgreSQL 实例。它可以与主数据库共享同一个主机（使用不同的数据库名），也可以使用独立主机。如果不使用 RAG，请跳过。

存储 — RWX PVC 或 S3

API、Worker、Worker-Beat 和 Plugin Daemon 需要共享的可写存储：

- 一个在各工作负载之间共享的 **ReadWriteMany (RWX)** StorageClass，或者

- 一个 **S3-compatible** 对象存储（MinIO、AWS S3、OBS 等）——在没有 RWX class 时推荐使用。

记录 StorageClass 名称，或者记录 S3 端点 + 区域 + bucket + access key。

步骤 4 — 确定并设置访问方式

Dify 面向浏览器，并且有两个访问面，因此 Dify 实例有两个 URL 字段：

- **Console URL** — 管理员 / 应用创建者使用的构建界面（创建应用、配置模型、管理知识库）。
- **App URL** — 终端用户打开的已发布应用（类似 `/chat/<code>` 的聊天 / agent 页面）。

这两个 URL 都必须与用户实际访问 Dify 的方式一致，否则前端无法加载。请从以下三个选项中选择一个——每个选项都说明了需要准备什么，以及两个 URL 如何填写。

选项	使用场景	需要准备的内容	**Console URL / App URL**
A. NodePort / Service IP	开发 / 内部使用；没有域名	无	两者都设置为 `http://<node-ip>:<nodeport>`（同一个地址同时用于两者）
B. 单域名	两个访问面使用同一个主机名	1 个 ACP Certificate + 1 个 ACP Domain；Envoy Gateway + 一个 `Gateway` 资源	两者都设置为 `https://<host>`
C. 双域名	生产环境——管理员与终端用户分离访问（DNS / 证书 / 网络暴露不同）	1 个 ACP Certificate（可同时覆盖两个域名，例如 wildcard）+ 2 个 ACP Domains；Envoy Gateway + 一个 `Gateway` 资源	Console URL = `https://<console-host>`，App URL = `https://<app-host>`

对于 选项 **A**，这里无需额外配置——请直接进入步骤 5。对于 选项 **B** 和 **C**，请按如下方式设置 Gateway。

设置 **Gateway**（仅限选项 **B** 或 **C**）

三个子步骤：在 ACP 中注册域名（以及证书）→ 安装 Envoy Gateway → 创建引用证书 Secret 的 `Gateway` 资源。

1. 在 **ACP** 中注册域名

在 管理员 视图中，进入 网络管理 → 域名 → 创建域名，然后填写表单：

字段	值
类型	域名（特定主机名）或泛域名（`*.example.com`）
域名	主机名，例如 `dify-console.example.com`
分配集群	目标集群 + 项目（或所有项目）
证书	将 **Public Key**（`tls.crt`）和 **Private Key**（`tls.key`）直接粘贴到表单中——ACP 会自动创建绑定的 TLS Secret

ACP 会创建：

- 全局集群上的一个 `Domain` 自定义资源（`apiVersion: crd.alauda.io/v2`，集群级别，并带有 `cluster.cpaas.io/name` / `project.cpaas.io/name` 标签）；
- 在 `cpaas-system` 命名空间中一个名为 `<domain>-<random>` 的 `kubernetes.io/tls` Secret（例如 `dify-console.example.com-xfd8x`）；Domain 的 `cpaas.io/secret-ref` 注解会记录该名称。

你可以从 Domain 详情页找到 Secret 名称。完整的 UI 参考：配置域名↗。

然后将该主机名的 DNS 解析 到集群的负载均衡器地址。

对于选项 C 下的第二个主机名，重复上述步骤（双域名）。覆盖两个主机的同一张证书（例如 wildcard）可以复用——在两个 Domain 表单中填写相同的证书，或者通过 网络管理 → 证书 先导入一次证书，再在创建域名时选择它（参见 创建证书↗）。

2. 安装 **Envoy Gateway**

ACP 包含 **Alauda Build of Envoy Gateway** operator。如果尚未安装，请从 **Marketplace / OperatorHub** 安装——参见 安装 Envoy Gateway operator↗。

3. 创建 `Gateway` 资源

引用子步骤 1 中的证书 Secret。由于这些 Secret 位于 `cpaas-system` 中，而 Gateway listener 的 `certificateRefs` 默认只能解析 Gateway 自身命名空间中的 Secret，因此你有两种等价的方式：

- 更简单 — 将 **Gateway** 放在 `cpaas-system` 中，这样就可以直接引用这些 Secret。
- 跨命名空间 — 将 **Gateway** 保留在自己的命名空间中，并在 `cpaas-system` 中添加一个 `ReferenceGrant`，允许你的 Gateway 命名空间引用其中的 Secret。

选项 **a** — 将 **Gateway** 放在 `cpaas-system` 中：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: dify
  namespace: cpaas-system                # co-located with the cert Secrets
spec:
  gatewayClassName: <envoy-gateway-class> # the GatewayClass exposed by Envoy Gateway
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      hostname: "*.example.com"           # cover all hostnames registered in sub-step 1
      tls:
        mode: Terminate
        certificateRefs:
          - {name: <secret-name-from-ACP-domain>}   # from the Domain detail page
      allowedRoutes:
        namespaces:
          from: All
```

选项 **b** — 将 **Gateway** 放在其他命名空间中，并使用 `ReferenceGrant`：

```yaml
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: dify
  namespace: <gateway-namespace>
spec:
  gatewayClassName: <envoy-gateway-class>
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      hostname: "*.example.com"
      tls:
        mode: Terminate
        certificateRefs:
          - {namespace: cpaas-system, name: <secret-name-from-ACP-domain
>}
      allowedRoutes:
        namespaces:
          from: All
---
apiVersion: gateway.networking.k8s.io/v1beta1
kind: ReferenceGrant
metadata:
  name: dify-tls
  namespace: cpaas-system                # grant lives in the Secret's n
amespace
spec:
  from:
    - {group: gateway.networking.k8s.io, kind: Gateway, namespace: <gatew
ay-namespace>}
  to:
    - {group: "", kind: Secret}            # could also restrict by name
```

对于在同一个 Gateway 上使用两个不同主机名的选项 C，请添加第二个 listener（每个 listener 都可以指向自己的 `hostname` 和 `certificateRefs`）。

步骤 5 — 提供插件安装代理 *(仅限受限网络)*

如果集群可以访问公网，请跳过此步骤。否则，请准备两个可通过 HTTP 访问的代理（可按任意方式部署——Nginx、现有 API gateway、devpi 等）：

- **Marketplace** 反向代理 — 代理到 `https://marketplace.dify.ai`，并保留 `Host: marketplace.dify.ai` 请求头。记录其 Service URL。
- **PyPI** 镜像 — 提供 PyPI simple index（例如代理 `https://mirrors.aliyun.com/pypi/simple/`，或任意 PyPI cache）。记录其 Service URL。

> **WARNING**
>
> Marketplace 代理 URL 必须是 **FQDN**（例如 `http://dify-marketplace-proxy.<ns>.svc.cluster.local`）——Dify 的内部 SSRF 代理（squid）无法解析集群内的短名称。

步骤 6 — 创建 Dify 实例

创建连接 Secret

```
# DB / Redis / pgvector passwords are read from key `password`
kubectl create secret generic dify-db       --from-literal=password='<db-password>'
kubectl create secret generic dify-redis    --from-literal=password='<redis-password>'    # omit if Redis has no auth
kubectl create secret generic dify-pgvector --from-literal=password='<pgvector-password>'
# S3 only — keys must be `accessKey` / `secretKey`
kubectl create secret generic dify-s3 --from-literal=accessKey='<ak>' --from-literal=secretKey='<sk>'
```

填写表单

在 **OperatorHub** → 已安装的 **operator** 中，选择 **Dify**，单击 创建 **Dify**，并填写表单。字段分组如下：

- 访问 — 根据步骤 4 中选择的方案设置 **Console URL** 和 **App URL**（在选项 A 和 B 中，两者相同；在选项 C 中，两个主机名不同）。包含协议（NodePort 使用 `http://`，Gateway 使用 `https://`）。
- 数据库 — 主机、端口、用户名（默认 `postgres`）、数据库名，以及 **DB Secret**（`dify-db`）。如果 PostgreSQL 需要 SSL，请打开 **SSL**。
- **Redis** — 选择 **Standalone** 或 **Sentinel**。
  - *Standalone:* 主机、端口、**Redis Secret**（如果没有认证可留空）。
  - *Sentinel:* 逗号分隔的 `host:port` 列表、**Master** 名称、**Redis Secret**，以及如果 Sentinel 本身需要认证，则还需要 **Sentinel Secret**。
- 向量存储（**pgvector**） — 开启 **Enable**。启用后需要填写：主机、端口、用户名、数据库名、**Vector Secret**。
- 存储 — 选择 **PVC** 或 **S3**。
  - *PVC:* RWX StorageClass 和大小。
  - *S3:* 端点、区域、**API bucket**、**Plugin bucket**、**S3 Secret**（`dify-s3`），以及 **Path-style addressing**（MinIO 和大多数 S3-compatible 端点请打开；AWS 请关闭）。
- **HTTP Route** — *仅在使用 Gateway 时（选项 B 或 C）显示。* 设置 **Gateway name**、**Gateway namespace**、可选的 **Section name**（listener）以及 **Hostnames**（必须被 Gateway listener 覆盖）。
- 高级 → **Plugin proxy** — *仅限受限网络集群。* 打开 **Marketplace**，并将 **Marketplace URL** 填写为步骤 5 中的代理地址（必须是 FQDN）。填写 **PIP Mirror URL**。打开 **Ignore uv lock**（否则 `uv sync --frozen` 会绕过镜像）。对于较慢的镜像，请增大 **Plugin init timeout**（例如 `1200` s）。如果你提供的是重新打包后的插件，请关闭 **Verify plugin signature**。

以下是相同配置的 YAML 参考（对应表单的 YAML 视图）：

```yaml
consoleUrl: https://dify-console.example.com
appUrl:     https://dify-app.example.com

database:
  host: <postgres-host>
  port: 5432
  username: postgres
  name: dify
  secret: dify-db
  sslEnabled: false

redis:
  mode: standalone              # standalone | sentinel
  host: <redis-host>
  port: 6379
  secret: dify-redis            # omit if no auth
  cacheDB: 0
  brokerDB: 1

vectorStore:
  enabled: true                 # set false to disable RAG
  host: <pgvector-host>
  port: 5432
  username: postgres
  name: dify_vector
  secret: dify-pgvector

storage:
  type: PVC                     # PVC | S3
  storageClass: <rwx-storage-class>
  size: 20Gi

httpRoute:                      # Option B / C only
  gatewayName: dify
  gatewayNamespace: <gateway-namespace>
  hostnames:
    - dify-console.example.com
    - dify-app.example.com

proxy:                          # restricted network only
  marketplace: true
  marketplaceURL: http://dify-marketplace-proxy.<ns>.svc.cluster.local
  pipMirrorUrl:   http://dify-pip-proxy.<ns>.svc.cluster.local/pypi/simpl
```

```
e/
  ignoreUvLock: true
  pluginInitTimeout: 1200
  verifyPluginSignature: false
```

步骤 7 — 登录

1. 等待 Dify 实例报告其工作负载为 `Ready`。
2. 在浏览器中打开 **Console URL**；Dify 会在首次登录时提示进行 初始管理员设置（邮箱 + 密码）。
3. 登录后，可从控制台创建更多用户。
4. 访问 **App URL** 以确认已发布应用的访问是否正常（可以先从控制台发布一个快速测试应用进行验证）。

参考：Dify 实例字段

字段	表单分组	作用
`consoleUrl`	访问	构建界面的浏览器 URL
`appUrl`	访问	已发布应用的浏览器 URL
`database.{host, port, username, name, secret, sslEnabled}`	数据库	外部 PostgreSQL 连接。`secret` 保存键 `password`。
`redis.{mode, host, port, sentinels, masterName, secret, sentinelSecret, cacheDB, brokerDB, sslEnabled}`	Redis	`mode = standalone` 或 `sentinel`
`vectorStore.{enabled, host, port, username, name, secret}`	向量存储	pgvector
`storage.{type, storageClass, size, endpoint, apiBucket, pluginBucket, s3Secret, region, s3PathStyle}`	存储	`type = PVC` 或 `S3`

字段	表单分组	作用
`httpRoute.{gatewayName, gatewayNamespace, sectionName, hostnames}`	HTTP Route	Gateway API 暴露；如果使用 NodePort，则省略
`proxy.{marketplace, marketplaceURL, pipMirrorUrl, ignoreUvLock, pluginInitTimeout, verifyPluginSignature}`	高级 — Plugin proxy	受限网络下的插件安装

Secret 键约定：

Secret 字段	**Secret** 必须包含的键
`database.secret` / `redis.secret` / `redis.sentinelSecret` / `vectorStore.secret`	`password`
`storage.s3Secret`	`accessKey`, `secretKey`

主要特性

Dify 功能的简明列表。有关概念和部署组件，请参见 Introduction。

目录

LLM 原生应用开发

RAG（检索增强生成）

Workflow 与 Agent

API 与集成

LLM 原生应用开发

- 低代码构建器：通过 Web UI 构建 Assistant、Text Generator、Agent 和 Workflow/Chatflow 原生应用
- 应用类型：用于聊天的 Assistant、用于文本生成的 Text Generator、用于推理和工具使用的 Agent、用于智能体自动化的 Workflow
- 模型支持：集成多个 LLM 提供商（例如 OpenAI、Azure、Anthropic）；在控制台中设置模型提供商
- **Prompt** 管理：可视化 Prompt 设计、变量和版本管理

RAG（检索增强生成）

- 知识库：导入文档和 URL，进行分段和嵌入以支持检索
- 向量存储：Helm chart 当前仅支持 **pgvector**（如果不使用 RAG，则可禁用）；在 values 中配置 pgvector 连接
- 检索选项：混合检索、重排序以及可配置的检索策略
- 上下文增强：父子上下文和扩展上下文，以获得更准确的答案
- 数据集管理：知识数据集的版本管理、更新和质量控制

Workflow 与 Agent

- 可视化 **Workflow**：基于节点的 workflow 编辑器，包含 LLM、检索、代码和逻辑节点；支持拖放式智能体流程
- **Agent** 能力：工具使用、多步推理和会话记忆
- 错误处理：可配置的错误处理和重试机制，保障生产环境的可靠性
- 可观测性：Workflow 和 Agent 的执行日志与调试

API 与集成

- 后端服务 **API**：用于聊天、completion 和 workflow 调用的 REST 端点；可从外部系统（后端、脚本、其他服务）调用你的应用。
- **API Keys**：在 **API Access**（应用侧边栏中）中为每个应用创建和管理 API 凭据；你可以为不同环境或用户创建多个密钥。参见 Developing with APIs↗。
- **Web App**：将你的应用发布为面向最终用户的基于浏览器的 UI：获取公共 URL，或将聊天/completion 组件嵌入到你的网站中（iframe 或 script）。
- **SDK** 与文档：用于集成的客户端 SDK 和 API 文档。

Alauda 版 Llama Stack

Alauda 的 Llama Stack 构建

Llama Stack

文档

安装 Llama Stack

上传 Operator 包

启用 Llama Stack

验证安装

部署 Llama Stack Server

快速入门

前提条件

快速入门示例

向量存储使用

常见问题

主要特性

基于 Server 的架构

使用工具的 AI 智能体

配置与可扩展性

集成

Alauda 的 Llama Stack 构建

目录

Llama Stack

文档

官方文档

Llama Stack

Llama Stack 是一个用于构建和运行带工具的 AI agent 的框架。它提供了一种基于 server 的架构，使开发者能够创建可与用户交互、访问外部工具并执行复杂推理任务的 agent。

主要组件和概念包括：

- **Llama Stack Server**：承载模型、agent 和工具运行时的中心服务。它可以通过 Llama Stack Operator 部署到 Kubernetes 上（参见 Install Llama Stack）。
- **Client SDK**（`llama-stack-client`）：用于连接 server、创建 agent、使用 `@client_tool` 装饰器定义工具以及管理 session 的 Python client。
- **Agents**：可配置的 AI agent，使用 LLM 模型并可调用工具（例如天气 API、自定义 API）来回答用户查询。
- **Tools**：向 agent 暴露的函数（例如天气查询）。通过 `@client_tool` 定义，并在创建时传递给 agent。

- **Configuration**：YAML stack configuration 定义 providers（inference、agents、safety、vector_io、files）、持久化后端以及模型注册（例如通过 OpenAI-compatible API 使用 DeepSeek）。

Llama Stack 支持多种 API provider、存储和持久化后端以及分发选项（例如 starter、postgres-demo、meta-reference-gpu），因此既适合快速实验，也适合生产部署。

文档

Llama Stack 提供官方文档和资源，便于深入了解其用法：

官方文档

- 主文档：https://llamastack.github.io/docs ↗
  - 用法、API provider 和核心概念
- 核心概念：https://llamastack.github.io/docs/concepts ↗
  - 架构、API 稳定性和资源管理

安装 Llama Stack

在 Alauda AI 2.8.x 中，**Alauda Build of Llama Stack** 由 Alauda AI 安装和管理。集群级 `default` `AmlCluster` 中的 `llamaStack` 组件默认值为 `Unmanaged`；将其设置为 `Managed` 后，您无需手动从 OperatorHub 安装 Operator。组件完成 reconcile 后，按照下文所述，通过 `LlamaStackDistribution` 自定义资源部署 Llama Stack Server。

目录

上传 Operator 包

启用 Llama Stack

验证安装

部署 Llama Stack Server

配置 PostgreSQL 存储

在 KServe 上使用 vLLM 进行 tool calling

启用 PGVector Vector Store

启用 Milvus Vector Store

Hugging Face 访问用于 embedding models

上传 Operator 包

下载 Llama Stack Operator 包，然后按照 上架软件包 上架该包。

启用 Llama Stack

在 **Administrator** 视图中：

1. 打开 **Marketplace** > **OperatorHub**。
2. 选择目标集群并打开 **Alauda AI**。
3. 打开 **All Instances** 选项卡，编辑 `default` `AmlCluster` 实例。
4. 将 `spec.components.llamaStack.managementState` 设置为 `Managed`。
5. 保存 `AmlCluster` 实例，并等待 Llama Stack Operator 变为 ready。

您可以在 `AmlCluster` YAML 中应用该组件设置：

```yaml
spec:
  components:
    llamaStack:
      managementState: Managed
```

验证安装

检查 `AmlCluster` 状态和 Llama Stack Operator：

```bash
kubectl get amlcluster default

kubectl get pods -n llama-stack-k8s-operator-system
```

`AmlCluster` 应返回 `Phase=Ready`，并且 Llama Stack Operator controller pod 应处于 `Running` 状态。Operator ready 后，系统会注册 `LlamaStackDistribution` CRD。

部署 Llama Stack Server

安装 Operator 后，通过创建 `LlamaStackDistribution` 自定义资源来部署 Llama Stack Server：

> 注意：请提前准备以下内容；否则 distribution 可能无法变为 ready：
>
> - **Inference URL**：`VLLM_URL` 必须指向一个 **vLLM OpenAI-compatible** 的 HTTP 基础 URL（例如集群内的 vLLM 或 KServe InferenceService），并且该 URL 需要提供目标模型服务。
> - **Secret**（可选）：仅当 vLLM endpoint 需要认证时才需要 `VLLM_API_TOKEN`。如果 vLLM 没有认证，请勿设置。若需要，请在同一 namespace 中创建一个 Secret，并在 `containerSpec.env` 中引用它（请参见下方 manifest 中的注释示例）。
> - **Storage Class**：请确保集群中存在 `default` Storage Class；否则 PVC 无法绑定，资源也不会变为 ready。
> - **PostgreSQL** 存储：本版本中的 `starter` distribution 使用 PostgreSQL 来持久化 Llama Stack。部署前，请为 server pod 配置 `POSTGRES_*` 环境变量。
> - **PGVector**（可选）：若要使用 `provider_id="pgvector"` 的 `vector_stores`，请为 server pod 提供 `PGVECTOR_*` 环境变量。由于 ACP 提供的 PostgreSQL 已包含 `pgvector` 扩展，因此可直接使用。
> - **Milvus**（可选）：若要使用 `provider_id="milvus-remote"` 的 `vector_stores`，请提供 `MILVUS_ENDPOINT`；如果启用了认证，还需提供 `MILVUS_TOKEN`。将 `MILVUS_CONSISTENCY_LEVEL` 设置为有效的 Milvus 一致性级别，例如 `Strong`。
> - **Embedding model** 下载：Llama Stack 为 vector-store 用法提供了默认的 embedding model 配置，但模型制品会在首次使用时从 Hugging Face 下载。如果需要镜像或代理，请配置 `HF_ENDPOINT`。对于完全离线的环境，请在第一次 vector-store 请求之前，预先将模型文件下载到 server PVC 中。

```yaml
apiVersion: llamastack.io/v1alpha1
kind: LlamaStackDistribution
metadata:
  annotations:
    cpaas.io/display-name: ""
  name: demo
  namespace: default
spec:
  network:
    exposeRoute: false                              # Whether to expose th
e route externally
  replicas: 1                                       # Number of server rep
licas
  server:
    containerSpec:
      name: llama-stack
      port: 8321
      env:
        - name: VLLM_URL
          value: "http://vllm-predictor.default.svc.cluster.local/v1"   #
vLLM OpenAI-compatible base URL
        - name: VLLM_MAX_TOKENS
          value: "8192"                             # Maximum output token
s

        # Optional: VLLM_API_TOKEN — add only when the vLLM endpoint requ
ires authentication.
        # If vLLM is deployed without auth, omit the entire block below
(do not set VLLM_API_TOKEN).
        # Example after creating: kubectl create secret generic vllm-api-
token -n default --from-literal=token=<TOKEN>
        # - name: VLLM_API_TOKEN
        #   valueFrom:
        #     secretKeyRef:
        #       key: token
        #       name: vllm-api-token

        # Required: PostgreSQL-backed Llama Stack persistence for this st
arter
        # distribution image.
        - name: POSTGRES_HOST
          value: "<postgresql-service>"
        - name: POSTGRES_PORT
```

```
          value: "5432"
        - name: POSTGRES_DB
          value: "<database-name>"
        - name: POSTGRES_USER
          value: "<database-username>"
        - name: POSTGRES_PASSWORD
          valueFrom:
            secretKeyRef:
              name: <postgresql-credentials-secret>
              key: password

        # Optional: enable PGVector-backed vector stores.
        # Omit the entire block below if you do not need PGVector vector
stores.
        # These settings configure the vector DB provider and are separat
e from
        # the POSTGRES_* persistence settings above, although they may po
int to
        # the same PostgreSQL instance when it has the pgvector extensio
n.
        # ACP-provided PostgreSQL already includes the pgvector extensio
n.
        # - name: ENABLE_PGVECTOR
        #   value: "true"
        # - name: PGVECTOR_HOST
        #   value: "<acp-postgresql-service>"
        # - name: PGVECTOR_PORT
        #   value: "5432"
        # - name: PGVECTOR_DB
        #   value: "<database-name>"
        # - name: PGVECTOR_USER
        #   value: "<database-username>"
        # - name: PGVECTOR_PASSWORD
        #   valueFrom:
        #     secretKeyRef:
        #       name: <pgvector-credentials-secret>
        #       key: password

        # Optional: enable remote Milvus-backed vector stores.
        # Use provider_id="milvus-remote" from the client API.
        # - name: MILVUS_ENDPOINT
        #   value: "http://<milvus-endpoint-host-and-port>"
        # - name: MILVUS_TOKEN
        #   valueFrom:
```

```
        #     secretKeyRef:
        #       name: <milvus-credentials-secret>
        #       key: token
        # - name: MILVUS_CONSISTENCY_LEVEL
        #   value: "Strong"

        # Required for PGVector or Milvus vector stores that use local
        # sentence-transformers embeddings.
        # - name: ENABLE_SENTENCE_TRANSFORMERS
        #   value: "true"
        #
        # Optional: configure a Hugging Face mirror or proxy for the defa
ult
        # embedding model download path.
        # - name: HF_ENDPOINT
        #   value: "<huggingface-mirror-or-proxy>"
        #
        # Optional: configure fully offline model loading. Pre-populate t
he
        # Hugging Face cache under /home/lls/.lls/huggingface/hub, then s
et:
        # - name: HF_HUB_CACHE
        #   value: "/home/lls/.lls/huggingface/hub"
        # - name: HF_HUB_OFFLINE
        #   value: "1"
        # - name: TRANSFORMERS_OFFLINE
        #   value: "1"
        # - name: HF_HUB_DISABLE_XET
        #   value: "1"

    distribution:
      name: starter                                  # Distribution name (o
ptions: starter, postgres-demo, meta-reference-gpu)
    storage:
      mountPath: /home/lls/.lls
      size: 1Gi                                      # Requires the "defaul
t" Storage Class to be configured beforehand
```

部署完成后，Llama Stack Server 将在集群内可用。访问 URL 会显示在 `status.serviceURL` 中，例如：

```
status:
  phase: Ready
  serviceURL: http://demo-service.default.svc.cluster.local:8321
```

配置 PostgreSQL 存储

本版本使用的 `starter` distribution 镜像需要 PostgreSQL 来持久化 Llama Stack。请在 `LlamaStackDistribution` 中配置以下 server 环境变量：

- `POSTGRES_HOST`
- `POSTGRES_PORT`
- `POSTGRES_DB`
- `POSTGRES_USER`
- `POSTGRES_PASSWORD`

这些设置用于 Llama Stack server 状态，不同于仅用于配置可选 PGVector vector-store provider 的 `PGVECTOR_*`。如果同一个 PostgreSQL 实例具备所需的数据库、凭据和 `pgvector` 扩展，您可以将其同时用于这两种角色。

在 KServe 上使用 vLLM 进行 tool calling

以下内容适用于 KServe 上的 **vLLM predictor**，不适用于 `LlamaStackDistribution` manifest。对于使用 **tools**（client-side tools 或 MCP）的 agent 流程，vLLM 进程必须暴露 tool-call 支持。请按上游 vLLM 的要求添加 predictor container `args`，例如：

```
args:
  - --enable-auto-tool-choice
  - --tool-call-parser
  - hermes
```

请根据 **served model** 以及该模型家族对应的 vLLM 文档，选择 `--tool-call-parser`（以及任何相关参数）。

启用 PGVector Vector Store

当在 server 上设置 `ENABLE_PGVECTOR=true` 时，Llama Stack 可以通过 client API 使用 `provider_id="pgvector"` 创建 vector store。

建议准备步骤：

1. 准备一个 ACP PostgreSQL 实例，并记录其 service name、database name、username 和 password。
2. 使用 `PGVECTOR_HOST`、`PGVECTOR_PORT`、`PGVECTOR_DB`、`PGVECTOR_USER` 和 `PGVECTOR_PASSWORD` 将数据库连接信息暴露给 `LlamaStackDistribution`。
3. 设置 `ENABLE_SENTENCE_TRANSFORMERS=true`，并确保默认 embedding model 文件能够在首次使用时被获取。
4. 如果集群使用 Hugging Face 镜像或代理，请相应设置 `HF_ENDPOINT`。
5. 如果集群完全离线，请预先将 embedding model 文件下载到 server PVC 中，并启用与离线缓存相关的环境变量。

distribution 变为 ready 后，您可以使用 Quickstart notebook 中的 PGVector 部分验证该配置。

启用 Milvus Vector Store

当在 server 上设置 `MILVUS_ENDPOINT` 时，Llama Stack 可以通过 client API 使用 `provider_id="milvus-remote"` 创建 vector store。

建议准备步骤：

1. 准备一个 Llama Stack Server pod 可访问的 Milvus endpoint。`MILVUS_ENDPOINT` 必须包含 scheme，即 `http://` 或 `https://`，以及您的 Milvus 服务所需的端口。
2. 使用 `MILVUS_ENDPOINT` 将 Milvus 连接信息暴露给 `LlamaStackDistribution`。
3. 如果启用了 Milvus 认证，请从 Secret 中设置 `MILVUS_TOKEN`。
4. 将 `MILVUS_CONSISTENCY_LEVEL` 设置为字符串值，例如 `Strong`；Milvus provider 需要该字段。
5. 设置 `ENABLE_SENTENCE_TRANSFORMERS=true`，并确保 embedding model 文件可以被获取或已经存在于 server PVC 中。

distribution 变为 ready 后，使用 Quickstart notebook 中的 Milvus 部分验证该配置。client 会使用 `provider_id="milvus-remote"` 创建 vector store，并在 `extra_body` 中传递所选的 embedding model id 以及 embedding dimension。

Hugging Face 访问用于 embedding models

Llama Stack 在 vector-store 操作中使用默认的 embedding model。首次使用时，server 会将模型文件从 Hugging Face 下载到本地缓存中。

推荐的缓存路径：

- `/home/lls/.lls/huggingface/hub`

常见部署模式：

1. 镜像或代理访问：

```
- name: HF_ENDPOINT
  value: "<huggingface-mirror-or-proxy>"
- name: HF_HUB_CACHE
  value: "/home/lls/.lls/huggingface/hub"
```

2. 完全离线访问：
   先将所需模型文件预下载到 PVC 支持的缓存目录 `/home/lls/.lls/huggingface/hub`，然后设置：

```
- name: HF_HUB_CACHE
  value: "/home/lls/.lls/huggingface/hub"
- name: HF_HUB_OFFLINE
  value: "1"
- name: TRANSFORMERS_OFFLINE
  value: "1"
- name: HF_HUB_DISABLE_XET
  value: "1"
```

如果缓存路径预先正确填充，server 就可以在运行时无需下载模型制品，即可创建由 PGVector 或 Milvus 支持的 vector store。

快速入门

本节提供一个使用 Llama Stack 创建 AI Agent 的快速入门示例。

目录

前提条件
快速入门示例
向量存储使用
常见问题
如何在 Notebook 中准备 Python 3.12
其他资源

前提条件

- Python 3.12 或更高版本（如果不满足，请参考 FAQ：如何在 Notebook 中准备 Python 3.12）
- 已通过 Operator 安装并运行的 Llama Stack Server（参见 Install Llama Stack），其中 `VLLM_URL` 指向由 **vLLM** 提供服务的 **model endpoint**，并且已为服务器持久化配置 `POSTGRES_*`（参见安装说明）
- 可访问 Notebook 环境（例如，Jupyter Notebook、JupyterLab）
- 已安装 `llama-stack-client==0.7.1`、`fastmcp`（用于 MCP 部分）以及其他 notebook 依赖的 Python 环境

快速入门示例

以下资源中提供了一个使用 Llama Stack 创建 AI Agent 的简单示例：

- **Notebook**：Llama Stack 快速入门演示

下载该 notebook 并将其上传到 Notebook 环境中运行。

该 notebook 演示了：

- **两种工具选项：**客户端工具（`@client_tool`）和 MCP 工具（FastMCP + `toolgroups.register`）
- **共享的智能体流程：**连接到 Llama Stack Server，选择模型，使用 `tools=AGENT_TOOLS` 创建一个 `Agent`，然后运行会话和流式轮次
- **可选的向量存储流程：**上传文件，创建一个由 `pgvector` 或 `milvus-remote` 支持的向量存储，并运行搜索查询
- 流式响应和事件日志记录
- `agent` 的可选 FastAPI 部署

向量存储使用

可下载的 notebook 包含可选的 PGVector 和 Milvus 部分。

对于 PGVector，请使用 `ENABLE_PGVECTOR=true` 和有效的 `PGVECTOR_*` 连接设置启动服务器，然后执行 notebook 中的 PGVector 单元。ACP 提供的 PostgreSQL 可以直接使用，因为它已经包含 `pgvector` 扩展。

对于 Milvus，请使用 `MILVUS_ENDPOINT`、可选的 `MILVUS_TOKEN` 和 `MILVUS_CONSISTENCY_LEVEL` 启动服务器，然后执行 notebook 中的 Milvus 单元。在客户端请求中使用 `provider_id="milvus-remote"`。

对于这两个向量存储示例，`client.models.list()` 必须包含一个 embedding model，例如 `sentence-transformers/nomic-ai/nomic-embed-text-v1.5`。如果它只返回 LLM models，请使用 `ENABLE_SENTENCE_TRANSFORMERS=true` 重新启动 `LlamaStackDistribution`，并按照 Install Llama Stack 中的说明配置 Hugging Face 缓存/下载访问。

该 notebook 示例涵盖：

- 通过 `client.files.create(...)` 上传文件
- 使用 `provider_id="pgvector"` 或 `provider_id="milvus-remote"` 创建向量存储
- 通过 `client.vector_stores.create(..., extra_body=...)` 传入 `embedding_model` 和 `embedding_dimension`
- 使用 `client.vector_stores.search(...)` 执行搜索；PGVector 在 `extra_body` 中使用 `search_mode="hybrid"`

常见问题

如何在 **Notebook** 中准备 **Python 3.12**

1. 下载预编译的 Python 安装包：

```
wget -O /tmp/python312.tar.gz https://github.com/astral-sh/python-build-standalone/releases/download/20260114/cpython-3.12.12+20260114-x86_64-unknown-linux-gnu-install_only.tar.gz
```

2. 使用以下命令解压：

```
mkdir -p ~/python312
tar -xzf /tmp/python312.tar.gz -C ~/python312 --strip-components=1
```

3. 安装并注册 Kernel：

```
export PATH="${HOME}/python312/bin:${PATH}"

python3 -m pip install ipykernel
python3 -m ipykernel install --user --name python312 --display-name "Python 3.12"
```

4. 在 notebook 页面中切换 kernel：

- 在浏览器中打开你的 Notebook 环境（例如 Jupyter Notebook 或 JupyterLab），然后打开一个已有的 notebook 或创建一个新的 notebook。
- 在 notebook 界面中，找到当前 kernel 名称（通常显示在页面的右上角，例如 “Python 3” 或 “python3”）。
- 单击该 kernel 名称，或使用菜单 **Kernel** → **Change Kernel**。
- 在 kernel 列表中，选择 **“Python 3.12”**（即第 3 步中注册的显示名称）。
- 切换后，新单元将使用 Python 3.12 运行。

注意：当在 notebook 页面中直接执行 python 和 pip 命令时，默认仍会使用 python。你需要指定完整路径，才能使用 python312 版本的命令。

其他资源

有关使用 Llama Stack 开发 AI Agents 的更多资源，请参见：

- Llama Stack 文档↗ - 官方 Llama Stack 文档，涵盖所有与使用相关的主题、API provider 和核心概念。
- Llama Stack 核心概念↗ - 深入了解 Llama Stack 架构、API 稳定性和资源管理。
- Llama Stack GitHub 仓库↗ - 源代码、示例应用、distribution 配置，以及如何添加新的 API provider。
- Llama Stack 示例应用↗ - 官方示例，演示如何在各种场景中使用 Llama Stack。

主要特性

目录

基于 Server 的架构

使用工具的 AI 智能体

配置与可扩展性

集成

基于 **Server** 的架构

- 集中式 **Server**：Llama Stack Server 托管推理、智能体、安全、tool runtime、vector I/O 和 files
- 远程或内联 **Provider**：支持远程 API（例如 OpenAI-compatible）和内联 provider（例如 meta-reference、sqlite-vec、localfs）
- **Kubernetes** 部署：通过 Llama Stack Operator 使用 `LlamaStackDistribution` 自定义资源进行部署

使用工具的 **AI** 智能体

- 智能体创建：使用模型、指令和工具列表创建智能体

- 客户端工具：使用 `@client_tool` 装饰器定义工具；客户端执行工具调用并将结果返回给服务器
- 会话管理：创建会话并运行支持流式响应的多轮对话
- 流式传输：支持智能体响应的流式传输，以实现实时显示

配置与可扩展性

- **Stack** 配置：基于 YAML 的配置，用于 API、provider、持久化（例如 kv_default、sql_default）和模型
- 环境回退：在配置中使用 `${env.VAR:~default}`，以实现灵活部署
- 多种 **Distribution**：Starter、postgres-demo、meta-reference-gpu 以及其他 distribution 选项

集成

- **Python Client**：适用于 Python 3.12+ 的 `llama-stack-client`，提供完整的智能体和模型 API
- 向量存储 **API**：从客户端创建和查询向量存储，包括基于 PGVector 的存储（`provider_id="pgvector"`）以及基于 Milvus 的存储（`provider_id="milvus-remote"`）
- 面向 **REST**：Server 暴露用于推理、智能体和 tool runtime 的 API；可在生产环境中封装到 FastAPI 或其他 Web 框架中使用

Kagenti operator

Kagenti Operator

概述

关键特性

核心配置

使用场景

安装 Kagenti

前提条件

上架 operator 软件包

安装 operator

Profile：Core 与 Secure

安全架构

概览

工作负载身份（

传输安全（Istio

授权（OPA + A

AuthBridge side

AgentRuntime 安

启用安全 profile

使用场景

相关内容

Kagenti Operator

Kagenti Operator 是一个 Kubernetes 原生的 operator，能够注册、发现并保护作为普通 **Kubernetes** 工作负载运行的 **AI agents** 和 **MCP** 工具。它基于开源项目 `kagenti/kagenti-operator` ↗ 构建，可让你通过 Kubernetes Custom Resource Definitions (CRDs)，以声明式方式将普通的 `Deployment` 转换为受管的 agent——并提供一致的标签、agent-to-agent (A2A) 发现以及可选的身份/安全 mesh。

目录

概述

关键特性

核心配置

使用场景

概述

该 operator 提供以下 CRDs（API group `agent.kagenti.dev`）：

- **AgentRuntime**：将目标工作负载（`Deployment` 或 `StatefulSet`）注册为 `agent` 或 `tool`。controller 会应用 `kagenti.io/type` 和 `app.kubernetes.io/managed-by` 标签，计算配置哈希，并且——在启用身份栈时——通过 AuthBridge webhook 触发 sidecar 注入。

- **AgentCard**：表示已发现的 agent。sync controller 会从工作负载的 Service 中获取该 agent 的 A2A↗ Agent Card 并进行索引，同时支持可选的 JWS 签名验证和 SPIFFE 身份绑定。
- **AuthorizationPolicy**：应用级授权意图，operator 会将其转换为底层 mesh（Istio / Kuadrant）。

operator 本身作为一个 **OLM operator** 交付，其 operand 是一个单独的 `Kagenti` custom resource（API group `kagenti.alauda.io`）。安装 operator 并创建 `Kagenti` 资源后，会将 agent controllers、CRDs 和 admission webhooks 部署到集群中。

名为 `agent-label-protection` 的 `ValidatingAdmissionPolicy` 会阻止直接在 `Deployment` 或 `StatefulSet` 上设置 `kagenti.io/type` 标签——它只能由 operator 通过 `AgentRuntime` 应用。

关键特性

- 声明式 **agent** 注册：为 `Deployment` 添加 `protocol.kagenti.io/a2a` 标签，并将其指向一个 `AgentRuntime`；其余部分由 operator 管理。
- 动态 **agent** 发现：`AgentCard` 资源会自动创建并同步，因此 agents 可以相互发现。
- **LLM** 后端无关：agents 可与任何兼容 OpenAI 的 chat endpoint 通信，因此集群内的 灵雀云 AI **InferenceService** 可以作为模型后端。
- **MCP** 工具集成：将 Model Context Protocol↗ servers 部署为 `tool` 类型工作负载，由 agent 调用以扩展其能力。
- 可插拔身份与安全：可启用可选栈（SPIRE/SPIFFE mTLS、Keycloak、Istio），用于工作负载身份和零信任传输。

核心配置

在 灵雀云 AI 上，operator 提供一个 核心配置：已启用 agent 注册、动态发现（`AgentCard`）以及 MCP 工具集成，而身份/安全栈（SPIRE、Keycloak、Istio mTLS）默认禁用，因为它需要基础安装中不包含的组件。在此配置下：

- agent Pod 运行时不会注入 **sidecar**（只有你自己的容器）。

- `AgentRuntime` 会报告 `MTLSReady=False`，原因是 `SPIREUnavailable`——这是预期行为，不会阻止 agent 或 tool 功能。

当集群中已具备 SPIRE、Keycloak 和 Istio 后，即可启用身份栈。

使用场景

- **Kubernetes** 上的 **agentic** 应用：将 A2A agents 和 MCP tool servers 作为一等、可发现的工作负载运行。
- 自托管模型后端：使用由 灵雀云 AI InferenceServices 提供的模型来驱动 agent，而不是外部 LLM 提供商。
- 增强工具的 **agents**：通过 MCP servers 为 agent 扩展实时数据能力（天气、搜索、内部 API 等）。

安装 Kagenti

Kagenti operator 作为 **OLM Operator** 交付，并从平台 **OperatorHub** 安装。

目录

前提条件

上架 operator 软件包

安装 operator

创建 Kagenti operand

验证

Profile：Core 与 Secure

前提条件

- **ACP** 版本：**v4.0** 或更高版本（已在 v4.3 上验证）。
- 目标集群架构为 `linux/amd64` 或 `linux/arm64`（该 operator 提供多架构镜像）。
- 目标集群上可用 Operator Lifecycle Manager (OLM)（由 ACP 提供）。
- 目标集群上已安装 **cert-manager**。该 operator 的 admission webhook 会从 cert-manager 获取其 serving certificate。

上架 operator 软件包

从 灵雀云 Customer Portal / Marketplace 下载 `Kagenti Operator` bundle，然后按照 上架软件包 上传软件包。

> **INFO**
>
> 该 operator bundle 会将其运行时镜像记录在 CSV 的 `relatedImages` 中，因此 `violet` release 会将这些镜像重定位到平台 registry。这使得该 operator 能够在 **air-gapped** 集群上安装，而无需访问 `docker.io` 或 `ghcr.io`。

安装 operator

在 管理员 视图中：

1. 单击 **Marketplace / OperatorHub**。
2. 在控制台顶部，从 **Cluster** 下拉菜单中选择目标集群。
3. 搜索并选择 **Kagenti Operator**，然后单击 **Install**。
4. 保持 **Channel** 不变（`stable`）。
5. 确认 **Version** 与要安装的 release 匹配。
6. 保持 **Installation Location** 不变——其默认值为 `kagenti-system` 命名空间。
7. 选择一个 **Upgrade Strategy**（生产环境建议使用 `Manual`）。
8. 单击 **Install**。

创建 Kagenti operand

从 OperatorHub 安装该 operator 会运行 **OLM operator**。要部署智能体控制器、CRD 和 webhook，请创建一个 `Kagenti` 自定义资源。空的 spec 会选择 **core profile**：

```
kubectl apply -f - <<EOF
apiVersion: kagenti.alauda.io/v1alpha1
kind: Kagenti
metadata:
  name: kagenti
  namespace: kagenti-system
spec: {}
EOF
```

验证

确认 **Kagenti Operator** 磁贴显示为 `Installed`，然后在集群上验证：

```
# the CSV reports Succeeded
kubectl get csv -n kagenti-system | grep kagenti

# the operator and agent controllers are Running
kubectl get pods -n kagenti-system

# the agent CRDs are registered
kubectl get crd | grep agent.kagenti.dev
```

您应当看到 `agentruntimes`、`agentcards` 和 `authorizationpolicies` CRD（group 为 `agent.kagenti.dev`）、处于 `Running` 状态的 `kagenti-controller-manager` 和 `kagenti-helm-operator-controller-manager` pod，以及 `agent-label-protection` `ValidatingAdmissionPolicy`。

该 operator 会监视所有命名空间，因此您可以在任意命名空间中注册智能体和工具。

Profile：Core 与 Secure

上面的空 spec `Kagenti` 资源会选择 **core profile**——智能体注册、动态发现以及 MCP 工具集成，但不包含身份 sidecar。这是默认配置，只需要 cert-manager。

secure profile 在此基础上增加零信任工作负载身份（SPIRE mTLS）、Keycloak OAuth2/JWT 认证，以及 Istio ambient mesh——由 AuthBridge sidecar 强制执行。它需要额外

的集群组件（SPIRE、Keycloak、Istio ambient）以及若干 `Kagenti` spec 设置，并且每个智能体命名空间都必须标记 `kagenti-enabled=true`。要启用它，请参见 启用 Secure Profile；关于架构，请参见 Security Architecture。

INFO

`Succeeded` 的 CSV 加上已协调完成的 `Kagenti` 资源，表示控制器正在运行。只有在您创建工作负载和 `AgentRuntime` 资源之后，智能体和工具本身才会出现——请参见 使用 AgentRuntime 部署智能体。

安全架构

除了注册和发现智能体之外，Kagenti Operator 还可以为你的智能体接入一套 **zero-trust** 安全栈：工作负载身份、传输加密和请求授权。本页将说明这些组件如何协同工作，以便你决定要启用哪些能力。

这套安全栈就是 operator 的 **secure profile**。它在 Introduction 中描述的核心 profile 里是 默认禁用 的——智能体以单个容器运行，`MTLSReady` 报告为 `False` 且原因为 `SPIREUnavailable`，并且不会强制执行任何授权。下面介绍的所有内容都属于可选启用，且依赖额外的集群组件。

目录

概览

工作负载身份（SPIFFE / SPIRE）

传输安全（Istio ambient mesh）

授权（OPA + AuthBridge + Keycloak）

示例：全局 Rego policy

AuthBridge sidecar

AgentRuntime 安全字段

启用安全 profile

使用场景

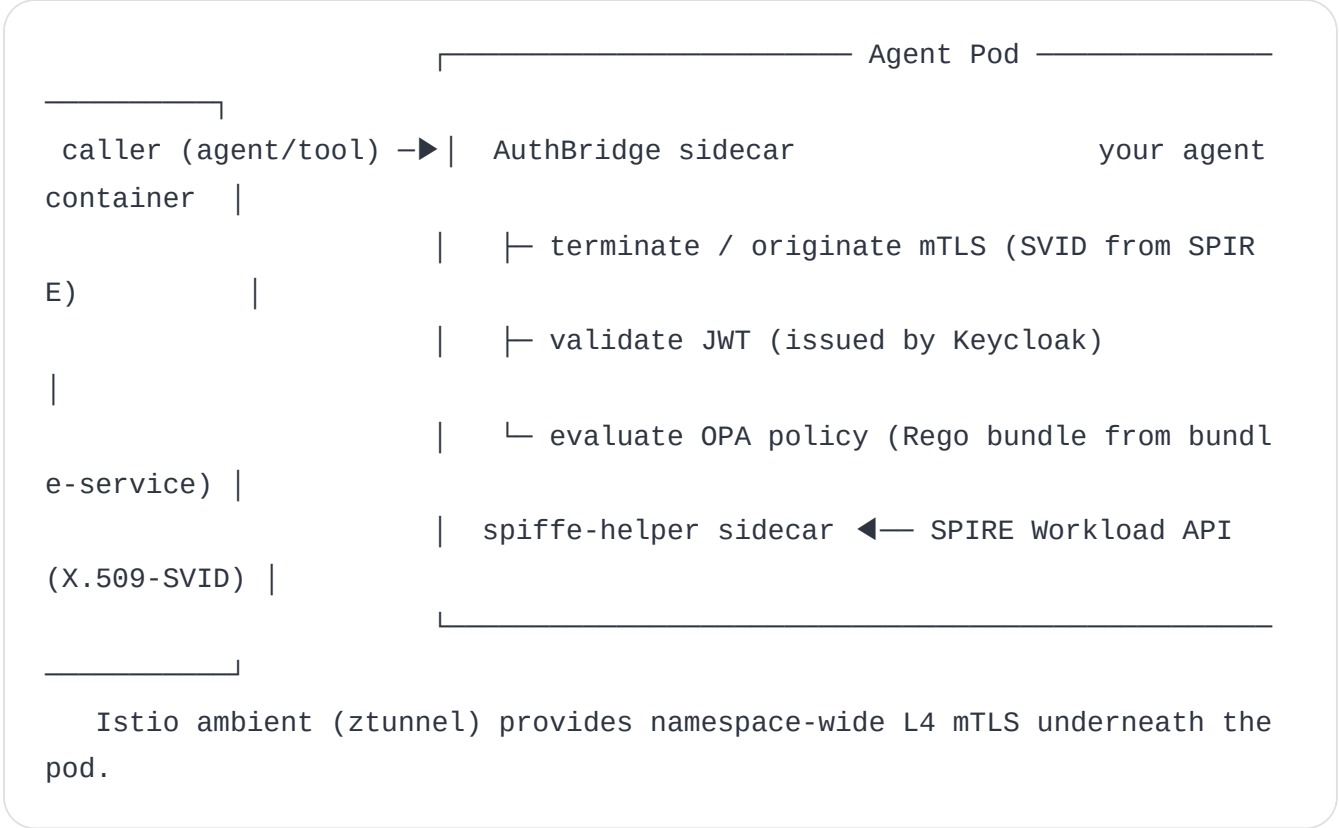
相关内容

概览

三个关注点对应三种技术，并由 operator 注入到每个智能体 Pod 中的 **AuthBridge** sidecar 串联起来：

关注点	问题	技术
工作负载身份	*这是什么工作负载？*	SPIFFE / SPIRE
传输安全	*连接是否已加密并经过身份验证？*	Istio ambient mesh (+ AuthBridge mTLS)
授权	*调用方是否被允许执行此操作？*	Open Policy Agent (OPA) + AuthBridge + Keycloak

经过安全保护的智能体，其请求路径如下：



> **INFO**
>
> 此 operator 不使用 Authorino 进行授权。授权通过 **OPA Rego policies** 表达，并由 AuthBridge sidecar 强制执行。可选的 `Kuadrant` 集成仅确保为 Istio ambient gateway 存在一个最基本的

`Kuadrant` 自定义资源——operator 不会将 policy 转换为 Kuadrant `AuthPolicy` 或 Authorino `AuthConfig` 对象。

工作负载身份（SPIFFE / SPIRE）

SPIFFE↗ 为每个工作负载提供一个加密身份；SPIRE↗ 则是在运行时签发该身份的组件。

- 当集群中存在 Zero-Trust Workload Identity Manager 时，operator 会协调 SPIRE 组件；你也可以自行部署 SPIRE。
- AuthBridge webhook 会注入一个 `spiffe-helper` sidecar 和 SPIFFE CSI socket。`spiffe-helper` 会从 SPIRE Workload API 获取一个 **X.509-SVID**，并自动轮换。
- 该身份的形式如下：

  ```
  spiffe://<trust-domain>/ns/<namespace>/sa/<service-account>
  ```

  由于该身份是从 **ServiceAccount** 派生而来，请为每个智能体分配各自独立的 ServiceAccount，以便获得不同的身份。

这个 SVID 是其他所有能力的信任根：它用于保护传输（mTLS）、命名工作负载的 Keycloak client，并对智能体的 `AgentCard` 进行签名，以支持可验证的发现。

传输安全（Istio ambient mesh）

operator 通过为智能体的 **namespace** 打标签，将其纳入 Istio ambient mode↗：

```
istio.io/dataplane-mode: ambient
istio-discovery: enabled
```

- 这会在 `AgentRuntime` 上作为 `IstioMeshEnrolled` 状态条件上报。（即使在核心 profile 中也会应用该 label，这就是你在那里看到 `IstioMeshEnrolled=True` 的原因。）
- Pod 之间的 L4 mTLS 由 Istio 的 **ztunnel** 透明处理——operator 不会编写 `PeerAuthentication` 或 `DestinationRule`。

- operator 会保持各 namespace 之间的证书颁发机构链一致（通过 cert-manager 的 mesh root 协调 Istio `cacerts`），以确保 SPIRE 和 Istio 签发的证书都链到同一个根。
- 可以通过注解 `kagenti.io/istio-mesh=disabled` 将某个 namespace 排除在外。

在 mesh 级 mTLS 之上，AuthBridge sidecar 还可以使用工作负载自身的 SVID 在智能体之间执行 应用级 **mTLS**——参见 `mtlsMode`。

授权（OPA + AuthBridge + Keycloak）

授权是一种在 AuthBridge sidecar 处进行评估的 **policy-as-code**：

1. 身份 **/ token** —— operator 会在 **Keycloak** 中将智能体注册为一个 OAuth2 client。其凭据会写入 Secret 并挂载到 Pod 中；sidecar 会根据 Keycloak 的 issuer 和 audience 获取并验证 **JWTs**。token exchange 支持 tool-to-agent delegation。
2. **Policy** —— 一个 `AuthorizationPolicy` 自定义资源保存一个或多个 **Rego** policies。`bundle-service` 会将它们编译为 OPA bundle，并按身份提供服务；AuthBridge sidecar 会拉取与其 SPIFFE ID 匹配的 bundle。
3. **Decision** —— AuthBridge 会在四个点进行 policy 评估：inbound/outbound × request/response，并且采用 **fail-closed**（`default allow := false`）。

policy 的作用域采用三层层级：

`spec.scope`	适用范围
`global`	集群中的所有 client（`kagenti-system` 中的默认 policy）
`namespace`	单个 namespace；继承/覆盖 global
`client`	特定的 OAuth2 client（通过 `clientID` 指定的智能体或工具）

示例：全局 **Rego policy**

```yaml
apiVersion: agent.kagenti.dev/v1alpha1
kind: AuthorizationPolicy
metadata:
  name: default
  namespace: kagenti-system
spec:
  scope: global (1)
  policies:
    - path: "inbound/request.rego" (2)
      content: | (3)
        package authbridge.inbound.request
        import rego.v1
        default allow := false
        allow if data.authbridge.ns.inbound.request.override
        allow if {
            ns_ok
            client_ok
        }
        ns_ok     if data.authbridge.ns.inbound.request.allow
        ns_ok     if not data.authbridge.ns.inbound.request
        client_ok if data.authbridge.client.inbound.request.allow
        client_ok if not data.authbridge.client.inbound.request
    # ... plus inbound/response, outbound/request, outbound/response
```

(1) `spec.scope` — `global`、`namespace` 或 `client`。默认 policy 以集群作用域提供，并委托给 namespace 和 client 作用域的规则。

(2) `policies[].path` — 该 Rego 覆盖的决策点（`{inbound,outbound}/{request,response}.rego`）。

(3) `policies[].content` — Rego 源码。默认规则是 fail-closed；团队可以在其上叠加 `namespace` 或 `client` 作用域的 policy，以放行特定调用方。

AuthBridge sidecar

AuthBridge 是负责强制执行身份、mTLS 和授权的数据平面组件。operator 的 mutating webhook 会将其注入到任何带有 `kagenti.io/type=agent` 标签的 Pod 中（以及在启用 `injectTools` feature gate 时注入 `tool`），前提是全局 feature gate 已开启。

sidecar 模式通过 `spec.authBridgeMode` 按智能体进行选择：

模式	注入内容	说明
`proxy-sidecar`	轻量级 AuthBridge proxy + `spiffe-helper`	默认；支持 TLS bridge
`envoy-sidecar`	Envoy proxy + `proxy-init` + `spiffe-helper`	完整的 Envoy 数据平面
`lite`	精简版 proxy	占用最小
`waypoint`	独立 waypoint	适用于 Istio ambient waypoint

Envoy/proxy 和 `spiffe-helper` 镜像随配套的 AuthBridge sidecar 镜像集合一起提供；在启用安全 profile 时，这些镜像必须可供集群使用。

AgentRuntime 安全字段

安全 profile 通过 `AgentRuntime` 按智能体进行配置。所有字段默认均为关闭/安全；一个完全加固的智能体看起来如下：

```yaml
apiVersion: agent.kagenti.dev/v1alpha1
kind: AgentRuntime
metadata:
  name: weather-agent-runtime
  namespace: agents
spec:
  type: agent
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: weather-agent
  authBridgeMode: proxy-sidecar     # proxy-sidecar (default) | envoy-sid
ecar | lite | waypoint (1)
  mtlsMode: permissive              # disabled | permissive (default) | s
trict (2)
  tlsBridgeMode: enabled            # per-agent egress CA, requires cert-
manager (3)
  egressEnforcement: enforce-redirect
```

(1) `authBridgeMode` — 要注入的 sidecar 数据平面类型（见上表）。

(2) `mtlsMode` — 智能体之间的应用级 mTLS。`permissive` 同时接受 TLS 和明文（适合在发布期间使用）；`strict` 表示 TLS-or-fail（生产环境）。只有在 SPIRE 可用时，`MTLSReady` 条件才会变为 `True`，并且它只是信息性状态——不会阻塞 `Ready`。

(3) `tlsBridgeMode` — 通过 cert-manager 为每个智能体签发 CA，使 sidecar 能够检查出站 HTTPS。仅支持 `proxy-sidecar` / `lite`。

启用安全 **profile**

安全 profile 依赖于不属于核心安装的集群组件。请先按大致如下顺序部署它们：

1. **cert-manager** — 为 operator 的 webhook 证书以及 TLS-bridge / Istio CA 链颁发证书。
2. **SPIRE**（或 Zero-Trust Workload Identity Manager）——签发工作负载 SVID。
3. **Keycloak** — 签发并验证 OAuth2/OIDC token；operator 会将智能体注册为 client。
4. **Istio**（**ambient**） — 提供 mesh 和 ztunnel mTLS；ambient gateway 需要一个 `Kuadrant` 自定义资源。

然后启用 operator feature gate（`featureGates.globalEnabled: true`、`authbridgeConfig.enabled: true`），使 AuthBridge webhook 开始注入，准备好 AuthBridge sidecar 镜像，并设置上面显示的 `AgentRuntime` 安全字段。

> **INFO**
>
> 与核心智能体启用相比，启用安全 profile 是一项重大的平台工程。请先在非生产集群中验证，并在切换到 `strict` 之前，先将 `mtlsMode` 逐步以 `permissive` 方式发布。

使用场景

- 智能体到智能体的授权 —— 通过 `client` 作用域的 Rego policy 结合 Keycloak audience / token exchange，允许智能体 A 调用智能体 B，而不允许反向调用。
- 多租户隔离 —— `namespace` 作用域的 policy 可防止一个团队的智能体访问另一个团队的智能体。
- 工具访问控制 —— 通过 `injectTools` feature gate 和按 client 的 policy，控制哪些工作负载可以作为工具使用。
- 可验证的智能体身份 —— `AgentCard`s 使用智能体的 SVID 进行 JWS 签名，因此消费者在信任或调用该智能体之前，可以先验证证书链是否可追溯到 trust bundle。
- 透明传输加密 —— 将 namespace 加入 Istio ambient 后，可获得无需修改应用的 ztunnel mTLS，并可在其上叠加可选的应用级 mTLS。

相关内容

- Introduction — 核心 profile 和 CRD 概览。
- Installation — 安装 operator 并创建 `Kagenti` operand。
- Deploy an Agent with AgentRuntime — 核心智能体 + MCP 工作流。

Alauda 构建版 MCP 生命周期 Operator

Alauda Build 的 MCP Lifecycle

主要特性
工作原理
使用场景
为什么使用 MCPServer
文档

安装 MCP Lifecycle Operator

上传 operator 软件包
安装 operator
验证安装
升级 operator

快速入门

先决条件
创建 MCPServe
验证 MCPServe
测试端点
更新端口或路径
常见配置
清理

Alauda Build 的 MCP Lifecycle Operator

Alauda Build 的 **MCP Lifecycle Operator** 基于 Kubernetes SIG Apps MCP Lifecycle Operator↗ 项目。它提供一个 Kubernetes 原生 API，用于将 Model Context Protocol↗ (MCP) server 作为普通集群工作负载进行部署和管理。

该 operator 引入了 `MCPServer` 自定义资源（`mcp.x-k8s.io/v1alpha1`）。`MCPServer` 描述 MCP server 的容器镜像、网络端口、MCP endpoint 路径、环境变量、挂载配置、运行时资源以及安全设置。controller 会将该声明调和为一个与 `MCPServer` 同名的 Kubernetes `Deployment` 和 `Service`。当前 API 仅支持将 `ContainerImage` 作为 `spec.source.type`。

> **Alpha API**
>
> 上游 `MCPServer` API 当前为 `v1alpha1`。字段名称和行为在未来版本中可能发生变化。

目录

主要特性

工作原理

使用场景

为什么使用 MCPServer

文档

主要特性

- 声明式 **MCP server** 生命周期：通过应用 `MCPServer` 资源来创建、更新、扩缩和删除 MCP server。
- 受管 **Kubernetes** 资源：该 operator 会创建并拥有底层的 `Deployment` 和 `Service`，并在 `MCPServer` 状态中报告它们的名称。
- 内置服务发现：`MCPServer` 状态会暴露一个内部 URL，例如 `http://<name>.<namespace>.svc.cluster.local:<port>/mcp`，供其他工作负载使用。
- 运行时自定义：可配置命令行参数、环境变量、ConfigMap 和 Secret 挂载、`EmptyDir` 存储、资源请求与限制、副本数、探针、service account 以及 security context。
- 具备会话感知的服务：使用 `spec.mcp.stateless` 选择生成的 Service 是采用自由负载均衡，还是客户端 IP 会话亲和性。

工作原理

1. 用户在应用命名空间中应用一个 `MCPServer` 资源。
2. controller 验证目标 MCP server 配置。
3. controller 创建或更新一个同名 `Deployment`，运行 MCP server 容器镜像。
4. controller 创建或更新一个同名 `Service`，在集群内暴露 MCP endpoint。
5. controller 将就绪条件和服务地址写入 `MCPServer.status`。

使用场景

- 面向 **AI** 智能体的 **MCP tools**：运行 tool server，向智能体框架暴露实时数据、内部 API 或运维操作。
- 可复用的平台服务：将 MCP server 打包为 Kubernetes 工作负载，并保持一致的部署、扩缩和状态行为。
- 智能体化 **MLOps** 集成：提供可供 Kagenti agents、Llama Stack agents、notebooks 或其他集群内 AI 应用消费的 MCP endpoint。

为什么使用 **MCPServer**

对于由现有应用部署流水线完全管理的单个 MCP server，普通的 `Deployment` 和 `Service` 可能已经足够。当 MCP tools 需要在多个命名空间和团队之间共享、发现或重复部署时，`MCPServer` 就会体现出价值。

其主要优势包括：

- 稳定的 **MCP** 专用 **API**：用户可以在一个对象中声明 MCP server 镜像、端口、路径、运行时设置和协议行为。
- 一致的状态和发现能力：operator 会在 `MCPServer` 状态中报告 `Accepted`、`Ready`、受管资源名称以及集群内部 MCP 地址。其他工具和平台工作流无需解析任意 Kubernetes 清单即可发现 MCP endpoint。
- 面向协议的默认值：`spec.config.path` 和 `spec.mcp.stateless` 等 MCP 专用字段使 endpoint 路径和会话亲和性显式化。
- 便于平台集成：专用的 MCP API 为平台提供了单一集成点，可用于未来的策略、UI、目录、可观测性以及 agent-tool 工作流。

文档

上游文档和相关资源：

- **MCP Lifecycle Operator** 文档：https://mcp-lifecycle-operator.sigs.k8s.io/↗
- **MCP Lifecycle Operator GitHub**：https://github.com/kubernetes-sigs/mcp-lifecycle-operator↗
- **Model Context Protocol**：https://modelcontextprotocol.io/↗
- **Kagenti Operator**：../kagenti/intro - 部署并发现 AI agents 和 MCP tools。

安装 MCP Lifecycle Operator

Alauda Build of MCP Lifecycle Operator 作为 **OLM Operator** 提供，并从平台 **OperatorHub** 安装。

目录

上传 operator 软件包

安装 operator

验证安装

升级 operator

上传 operator 软件包

从 `<Term name="company" /> Customer Portal / Marketplace` 下载 Alauda Build of MCP Lifecycle Operator 软件包，然后按照 Upload Packages 上架软件包。

> **INFO**
>
> 该 operator 软件包包含 OLM bundle 以及 operator 所需的相关运行时镜像。对于 air-gapped 集群，请确保由你的 `MCPServer` 资源使用的 MCP server 镜像也已镜像到集群可拉取的 registry。

安装 operator

在 **Administrator** 视图中：

1. 单击 **Marketplace** / **OperatorHub**。
2. 在 **Cluster** 下拉列表中，选择目标集群。
3. 搜索并选择 **Alauda Build of MCP Lifecycle Operator**。
4. 单击 **Install**。
5. 保持 **Installation Location** 不变，除非你的环境需要不同的 namespace。默认 namespace 为 `mcp-lifecycle-operator-system`。
6. 选择一个 **Upgrade Strategy**。生产环境推荐使用 `Manual`。
7. 单击 **Install**。

该 operator 会监视所有 namespace。安装成功后，你可以在任意应用 namespace 中创建 `MCPServer` 资源。

验证安装

确认 OperatorHub 磁贴显示为 `Installed`，然后在集群上验证：

```
# the CSV reports Succeeded
kubectl get csv -n mcp-lifecycle-operator-system | grep mcp-lifecycle-ope
rator

# the controller pod is Running
kubectl get pods -n mcp-lifecycle-operator-system

# the MCPServer CRD is registered
kubectl get crd mcpservers.mcp.x-k8s.io
```

你应该能看到 `mcp-lifecycle-operator-controller-manager` pod 处于 `Running` 状态，并且 `mcpservers.mcp.x-k8s.io` CRD 已注册。

升级 **operator**

1. 按照 Upload Packages 上架软件包，上传新的 Alauda Build of MCP Lifecycle Operator 软件包。
2. 转到 **Administrator** -> **Marketplace** -> **OperatorHub**。
3. 找到 **Alauda Build of MCP Lifecycle Operator**，并根据已配置的升级策略应用新版本。
4. 升级后验证 CSV、controller pod 以及现有的 `MCPServer` 资源。

快速入门

本快速入门会创建一个 `MCPServer` 资源，并验证 operator 会创建对应的 Kubernetes `Deployment` 和 `Service`。

目录

先决条件

创建 MCPServer

验证 MCPServer

测试端点

更新端口或路径

常见配置

清理

先决条件

- 已安装 Alauda Build 的 MCP Lifecycle Operator。请参见安装。
- 已为目标集群配置 `kubectl`。
- 已将一个 MCP server 镜像推送到目标集群可以拉取的 registry。

创建 MCPServer

应用一个 `MCPServer` 资源。当前 API 仅支持 `spec.source.type` 使用 `ContainerImage`。请将 `<registry>/<project>/<mcp-server-image>:<tag>` 替换为一个在配置端口上暴露 MCP HTTP endpoint 的 MCP server 镜像。例如，上游 MCP Lifecycle Operator 示例使用的是 `quay.io/containers/kubernetes_mcp_server:latest`；仅当目标集群可以从 `quay.io` 拉取镜像时才使用该镜像。

```
kubectl apply -f - <<EOF
apiVersion: mcp.x-k8s.io/v1alpha1
kind: MCPServer
metadata:
  name: kubernetes-mcp-server
  namespace: default
spec:
  source:
    type: ContainerImage
    containerImage:
      ref: <registry>/<project>/<mcp-server-image>:<tag>
  config:
    port: 8080
EOF
```

operator 会在同一命名空间中创建一个名为 `kubernetes-mcp-server` 的 `Deployment` 和 `Service`。

验证 MCPServer

检查 `MCPServer` 状态：

```
kubectl get mcpservers -n default
kubectl get mcpserver kubernetes-mcp-server -n default -o yaml
```

`kubectl get mcpservers` 的预期输出：

```
NAME                    READY   ACCEPTED   IMAGE
PORT   ADDRESS                                                    AGE
kubernetes-mcp-server   True    True       <registry>/<project>/<mcp-serv
er-image>:<tag>    8080   http://kubernetes-mcp-server.default.svc.cluste
r.local:8080/mcp   1m
```

`ADDRESS` 列是其他工作负载可以调用的集群内部 MCP 端点。

验证受管资源：

```
kubectl get deployment kubernetes-mcp-server -n default
kubectl get service kubernetes-mcp-server -n default
kubectl get pods -n default -l mcp-server=kubernetes-mcp-server
```

`Deployment` 应处于可用状态，`Service` 应已存在，且 pods 应为 `Running`。

测试端点

转发生成的 `Service`：

```
kubectl port-forward service/kubernetes-mcp-server 8080:8080 -n default
```

在另一个终端中测试 server：

```
curl http://localhost:8080/healthz
curl http://localhost:8080/mcp
```

确切的响应取决于 MCP server 镜像。端点可访问即可确认 operator 创建的 `Service` 正在将流量路由到 MCP server pod。

更新端口或路径

默认的 MCP 路径为 `/mcp`。如果要显式设置该路径或更改端口，请更新 `spec.config`：

```
kubectl patch mcpserver kubernetes-mcp-server -n default --type merge -p
'{
  "spec": {
    "config": {
      "port": 9090,
      "path": "/mcp"
    }
  }
}'
```

调和完成后，验证更新后的地址：

```
kubectl get mcpserver kubernetes-mcp-server -n default
kubectl get service kubernetes-mcp-server -n default
```

常见配置

使用 `spec.config` 配置容器级设置，例如参数、环境变量和挂载的配置：

```yaml
spec:
  source:
    type: ContainerImage
    containerImage:
      ref: <registry>/<project>/<mcp-server-image>:<tag>
  config:
    port: 8080
    path: /mcp
    arguments:
      - --config
      - /etc/mcp/config.toml
    env:
      - name: LOG_LEVEL
        value: info
      - name: API_KEY
        valueFrom:
          secretKeyRef:
            name: mcp-server-secret
            key: api-key
    storage:
      - path: /etc/mcp
        permissions: ReadOnly
        source:
          type: ConfigMap
          configMap:
            name: mcp-server-config
```

引用的 ConfigMaps 和 Secrets 必须与 `MCPServer` 位于同一命名空间中。

使用 `spec.runtime` 配置生成的 `Deployment`：

```yaml
spec:
  runtime:
    replicas: 2
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
      limits:
        cpu: 500m
        memory: 256Mi
    health:
      livenessProbe:
        httpGet:
          path: /healthz
          port: 8080
      readinessProbe:
        httpGet:
          path: /healthz
          port: 8080
```

当 MCP server 需要特定的 Kubernetes service account 或更严格的 pod 设置时，使用 `spec.runtime.security`：

```yaml
spec:
  runtime:
    security:
      serviceAccountName: mcp-server
      podSecurityContext:
        runAsNonRoot: true
        seccompProfile:
          type: RuntimeDefault
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
        runAsNonRoot: true
        capabilities:
          drop:
            - ALL
```

如果 MCP server 不保留按客户端划分的会话状态，请将 `spec.mcp.stateless` 设置为 `true`，这样生成的 `Service` 就可以在多个副本之间自由进行负载均衡：

```
spec:
  runtime:
    replicas: 3
  mcp:
    stateless: true
```

清理

删除 `MCPServer` 资源：

```
kubectl delete mcpserver kubernetes-mcp-server -n default
```

operator 会移除受管的 `Deployment` 和 `Service`。

评测与安全

本章介绍模型和 RAG 系统的评测，以及 LLM 输入和输出的护栏。

本章所依赖的组件列在下方的 Components 中。使用 TrustyAI 进行偏差和漂移监控的内容在监控 中介绍。

评估 LLM

评估 LLM

前提条件

运行评估作业

资源状态

获取结果

可选：离线存储和 PVC

使用 Ragas 评估 RAG

使用 Ragas 评估 RAG

每个评估样本需要记录的内容

Ragas 指标概览

调用 Ragas SDK

前提条件
可运行的 notebook
故障排查
结果解读
延伸阅读

用于 LLM 安全的 AI Guardrails

用于 LLM 安全的 AI Guardrails

前提条件
使用 AutoConfig 部署
内置 regex detector
服务端口和 API 参考
API 用法和示例

NeMo Guardrails

NeMo Guardrails

前提条件
架构
NeMo 配置 ConfigMap
部署 NemoGuardrails 自定义资源
认证（已启用 auth）
访问 NeMo Guardrails API
延伸阅读

组件

TrustyAI

评估 LLM

LM-Eval 提供了一个统一框架，用于在广泛的评估任务上测试 LLM。该服务基于 EleutherAI's lm-evaluation-harness ↗ 和 Unitxt ↗。TrustyAI Operator 通过 **LMEvalJob** CRD 实现它，因此可以在集群上创建和管理评估作业。

本文档介绍如何针对作为 Kubernetes InferenceService 提供服务的 LLM 运行评估作业（与 OpenAI API 兼容）。

目录

前提条件
运行评估作业
资源状态
获取结果
可选：离线存储和 PVC
离线模式的 spec 设置
用于离线缓存的环境变量
为离线运行准备 PVC 数据集

前提条件

- 已安装 TrustyAI Operator（参见 安装 TrustyAI）。

- 在目标命名空间中部署了一个作为 InferenceService 的 LLM（例如 vLLM 或 Hugging Face runtime）。
- 对于需要从互联网下载的任务或 tokenizer（例如 Hugging Face）：必须在 LMEvalJob 上启用 **allowOnline**，并且集群必须允许该操作（例如在 DataScienceCluster TrustyAI eval 配置中设置 `permitOnline: allow`）。启用在线访问会带来安全影响；请参阅 Red Hat 文档。

运行评估作业

创建一个指向 InferenceService 并指定评估任务的 **LMEvalJob** 自定义资源。operator 会在 pod 中运行该作业；作业完成后，结果会写入 `status.results`。

示例：使用 **arc_easy** 任务（lm-evaluation-harness 任务名称）评估集群内 LLM。模型通过 predictor service URL 访问；tokenizer 从 Hugging Face 加载（需要 **allowOnline: true** 以及集群授权）。

```yaml
apiVersion: trustyai.opendatahub.io/v1alpha1
kind: LMEvalJob
metadata:
  name: evaljob-sample
  namespace: <your-namespace>
spec:
  model: local-completions
  modelArgs:
    - name: model
      value: <inference-service-name>
    - name: base_url
      value: http://<inference-service-name>-predictor.<your-namespace>.svc/v1/completions
    - name: num_concurrent
      value: "1"
    - name: max_retries
      value: "3"
    - name: tokenized_requests
      value: "True"
    - name: tokenizer
      value: <huggingface-model-repo>            # e.g. Qwen/Qwen3.5-4B
  taskList:
    taskNames:
      - arc_easy
  allowOnline: true
  allowCodeExecution: false
  batchSize: "1"
  limit: "2"
  logSamples: true
  chatTemplate:
    enabled: false
  outputs:
    pvcManaged:
      size: 10Mi
  pod:
    container:
      env:
        # Optional: only set HF_ENDPOINT if your organization provides a trusted mirror.
        # - name: HF_ENDPOINT
        #  value: https://<your-approved-hf-mirror>
```

- 模型类型 (`model`)
  - 对于与 OpenAI API 兼容的 server（例如 InferenceService predictor），使用 `local-completions` 或 `local-chat-completions`。
  - 它们映射到 OpenAI endpoints：
    - `local-completions` 对应 `/v1/completions`
    - `local-chat-completions` 对应 `/v1/chat/completions`
  - **modelArgs.base_url** 必须使用相同的 path（例如 `xxx/v1/completions` 或 `xxx/v1/chat/completions`）。
- 模型参数 (`modelArgs`)
  - `base_url`：包含 path 的 predictor URL
    - `local-completions` 对应 `/v1/completions`
    - `local-chat-completions` 对应 `/v1/chat/completions`
  - `model`：通常与 InferenceService 名称一致。
  - `tokenizer`：当 `tokenized_requests` 为 true 时，用于分词的 Hugging Face model ID。
  - 其他参数（例如 `num_concurrent`、`max_retries`、`batch_size`）遵循 lm-evaluation-harness 文档↗。
- 任务 (`taskList.taskNames`)
  - lm-evaluation-harness 任务名称列表（例如 `arc_easy`、`mmlu`）。
  - 支持的完整任务集合和通配符由 lm-evaluation-harness↗（Task Guide / available tasks）定义。
  - 另外，也可以使用带有 Unitxt card/template 的 `taskRecipes` 来定义自定义任务。
- 在线模式和代码执行
  - `allowOnline`：当为 `true` 时，作业可以从互联网下载数据集和 tokenizer（例如 Hugging Face）；需要集群级别的授权。
  - `allowCodeExecution`：当为 `true` 时，作业可能会运行来自已下载资源的代码；默认值为 `false`，仅在需要且被允许时启用。

- 输出和限制
  - `outputs.pvcManaged`：创建一个由 operator 管理的 PVC 来存储作业结果（大小，例如 `100Mi`）。如果只设置了 **size**，PVC 将使用集群默认的 `StorageClass`；如果不存在默认 `StorageClass`，PVC 将保持 Pending，且不会配置存储。也可以使用 `outputs.pvcName` 绑定到现有 PVC。
  - `limit`：样本数量的可选上限（例如 `"2"`，用于快速运行）。
  - `logSamples`：当为 `true` 时，会保存每个 prompt 的模型输入和输出以供检查。

资源状态

LMEvalJob 的 `status` 子资源会报告作业状态，并在完成后报告评估结果。

- `status.state`：作业的当前状态：`New`、`Scheduled`、`Running`、`Complete`、`Cancelled` 或 `Suspended`。在读取结果之前请等待 `Complete`。
- `status.reason`：在作业结束时设置（例如 `Succeeded`、`Failed`）。
- `status.results`：当 **state** 为 `Complete` 时，此字段包含 JSON 字符串形式的评估结果（按任务/recipe 划分的指标）。
- `status.message`：人类可读的消息；**status.podName** 是作业 pod 的名称。

在读取流量或结果时，应基于 **status.state == Complete**（并且，如适用，基于 **status.reason == Succeeded**）。

获取结果

当 **status.state** 为 `Complete` 时，结果可在 **status.results** 中获取（JSON 字符串）。示例：

```
kubectl get lmevaljob evaljob-sample -n <your-namespace> -o jsonpath='{.status.results}' | jq '.'
```

`arc_easy` 任务的示例结果结构（关键字段；完整输出还包括 `configs`、`config`、`n-shot`、`n-samples` 和环境信息）：

▶ 示例结果（arc_easy）

可选：离线存储和 PVC

在 **offline** 模式下，评估作业不会访问互联网；模型和数据集必须从 PVC（或镜像）中读取。当集群不允许在线访问或用于 air-gapped 环境时，请使用此模式。

离线模式的 spec 设置

- 作业字段
  - `allowOnline: false`：作业不会从互联网下载内容。
  - `offline.storage.pvcName`：现有 PVC 的名称。operator 会将此 PVC 挂载到作业 pod 中；作业会从该挂载点下的路径加载模型和数据集。
- **spec** 中的路径
  - 模型 / 数据集加载器必须指向已挂载的 PVC。
  - 对于 Hugging Face 模型，请配置 **modelArgs**，使模型路径位于 PVC 挂载目录下（例如 `/opt/app-root/src/hf_home/<model-dir>`）。
  - 对于从磁盘加载的 `taskRecipes` 或自定义 Unitxt card，请将加载器路径设置在同一挂载目录下。

用于离线缓存的环境变量

在 `spec.pod.container.env` 中设置环境变量，以便加载器使用 PVC 作为缓存/存储。为提高可靠性，请将以下所有变量设置为 PVC 挂载目录下的同一路径（例如 `/opt/app-root/src/hf_home`）：

- `HF_DATASETS_CACHE`：Hugging Face `datasets` 的缓存目录。
- `HF_HOME`：Hugging Face 主页目录，供 tokenizer 和其他资源使用。
- `TRANSFORMERS_CACHE`：`transformers` models 和 tokenizer 的缓存目录。

离线模式示例片段：

```yaml
spec:
  allowOnline: false
  offline:
    storage:
      pvcName: my-offline-pvc
  pod:
    container:
      env:
        - name: HF_DATASETS_CACHE
          value: /opt/app-root/src/hf_home
        - name: HF_HOME
          value: /opt/app-root/src/hf_home
        - name: TRANSFORMERS_CACHE
          value: /opt/app-root/src/hf_home
```

仅在存储 评估结果 时使用 `outputs.pvcName` 或 `outputs.pvcManaged`；`offline.storage.pvcName` 用于 输入（models 和 datasets）。

为离线运行准备 PVC 数据集

在离线模式下，**dataset**（以及如果使用 HF，则包括 tokenizer/model 文件）必须已经存在于 PVC 中。作业不会从网络获取它们。

准备 PVC 的一种实用方法是：

1. 在线预热作业
   - 创建一个 `allowOnline: true` 的 LMEvalJob。
   - 挂载目标 PVC（即稍后在离线模式中使用的 PVC），例如通过 `offline.storage.pvcName` 或额外卷。
   - 让该作业下载所需的数据集/tokenizer/models，使其存储在 `HF_DATASETS_CACHE`、`HF_HOME` 和 `TRANSFORMERS_CACHE` 所使用的 PVC 路径下，以及配置的 `modelArgs` / 任务加载器路径下。
2. 离线评估作业
   - 创建真正的评估作业，将 `allowOnline: false` 并将 `offline.storage.pvcName` 指向同一个 PVC。

- 现在作业会在没有任何外部网络访问的情况下，从 PVC 中读取所有 models 和 datasets。

使用 Ragas 评估 RAG

检索增强生成（Retrieval-Augmented Generation，RAG）系统将检索器和生成器结合在一起。仅仅衡量最终答案的质量通常是不够的：失败可能来自无关或不完整的检索，来自偏离来源的生成，也可能两者皆有。

Ragas↗（Retrieval-Augmented Generation Assessment）是一个 Python 框架，可使用一组指标来评估 RAG 行为，例如上下文忠实度、答案相关性、检索质量等——具体取决于启用了哪些指标以及评估集中的可用列。

本页面重点介绍在 **notebook** 或批处理作业中直接使用 **Ragas SDK**。与其他评估平台的集成是可选的，本文不作介绍。

目录

- 每个评估样本需要记录的内容
- Ragas 指标概览
  - 核心 RAG 指标
  - 可选 RAG 指标
  - 选择最小 RAG 指标集
- 调用 Ragas SDK
- 前提条件
- 可运行的 notebook
- 故障排查
- 结果解读
- 延伸阅读

每个评估样本需要记录的内容

典型的单轮数据行包括：

字段	角色
`user_input`	用户查询（或等价内容）。
`retrieved_contexts`	该行检索到的片段，作为 字符串列表。
`response`	需要评分的模型输出。
`reference`	参考答案或关键信息；仅某些指标需要（例如 context recall）。

Ragas 指标概览

Ragas 提供了大量指标。对典型的 RAG 评估流程来说，只需要其中一部分即可。每个指标都期望特定的数据集列（例如 `question`、`contexts`、`answer`、`ground_truth`），并且可能需要 **LLM**、**embeddings**、两者都需要，或者两者都不需要。名称和导入路径会随版本演进而变化；请参考 Ragas 指标文档↗中的已安装版本说明。

下面的列表总结了这些指标的意图和常见用途；它们不能替代上游 API 细节。

核心 RAG 指标

这些指标与检索增强生成质量最直接相关，通常是最先跟踪的一组指标：

除非另有说明，本节中的类来自 `ragas.metrics.collections`。

指标	Python 类	必需参数	评估目标
忠实度	`Faithfulness`	`user_input`, `retrieved_contexts`, `response`	答案中的主张是否由检索到的上下文支持（事实依据 / 幻觉控制）。
答案相关性 / 响应相关性	`AnswerRelevancy`	`user_input`, `response`	生成的答案是否回应了用户查询。
上下文精度	`ContextPrecision`	`user_input`, `retrieved_contexts`, `reference`	检索到的片段是否与回答查询相关且有用。
上下文召回率	`ContextRecall`	`user_input`, `retrieved_contexts`, `reference`	检索到的上下文是否覆盖了正确回答所需的信息。
上下文实体召回率	`ContextEntityRecall`	`reference`, `retrieved_contexts`	参考答案中的关键信息实体是否出现在检索到的上下文中。
上下文利用率	`ContextUtilization`	`user_input`, `response`, `retrieved_contexts`	答案实际使用了多少检索到的上下文。

字段和导入要求可能因 Ragas 版本和指标变体而异。请结合 Ragas 指标文档↗确认已安装版本的要求。

可选 RAG 指标

这些指标在特定评估设置下很有用，尤其是在有参考答案，或者需要进行鲁棒性检查时：

指标	Python 类	必需参数	评估目标
答案正确性	`AnswerCorrectness`	`user_input`, `response`, `reference`	生成答案与参考答案的一致性。

指标	**Python** 类	必需参数	评估目标
答案相似度 / 语义相似度	`SemanticSimilarity`	`response`，`reference`	生成答案与参考答案之间的语义接近程度（基于 embedding）。
事实正确性	`FactualCorrectness`	`response`，`reference`	与参考内容或预期事实在事实层面的匹配程度。
噪声敏感性	`NoiseSensitivity`	`user_input`，`response`，`reference`，`retrieved_contexts`	当在上下文或输入中引入干扰项或噪声时的稳定性。

对于与 RAG 核心评估关联较弱的指标（例如通用文本重叠指标、基于评分准则的自定义指标、智能体/工具指标、SQL 指标或多模态指标），请参考 Ragas 指标文档↗。

选择最小 **RAG** 指标集

许多 RAG 基准测试的实用默认配置是：忠实度、答案相关性、上下文精度 和 上下文召回率（召回率以及某些精度变体需要 `ground_truth` 或等效字段）。当有参考答案时，再添加 答案正确性 或 语义相似度。请将指标与数据集中存在的列以及成本约束相匹配（依赖 LLM 的指标更慢且更昂贵）。

调用 **Ragas SDK**

在现代 Ragas 用法中，应从 `ragas.metrics.collections` 实例化指标，并使用 `ascore()`（或在同步脚本中使用 `score()`）对每一行进行评分。

1. 为 LLM 和 embeddings 准备 OpenAI 兼容客户端（`AsyncOpenAI`），然后配置 `llm_factory` 和 `OpenAIEmbeddings`（有关环境变量配置，请参见示例 notebook）。
2. 使用显式依赖项（需要时提供 `llm`、`embeddings`）实例化指标。
3. 遍历各行，并使用指标特定参数调用 `metric.ascore(...)`。

```python
from openai import AsyncOpenAI
from ragas.embeddings import OpenAIEmbeddings
from ragas.llms import llm_factory
from ragas.metrics.collections import AnswerRelevancy, Faithfulness

llm_client = AsyncOpenAI(
    api_key="...",
    base_url="https://your-openai-compatible-endpoint/v1",  # or None for provider default
)
embed_client = AsyncOpenAI(
    api_key="...",  # often same key as LLM when using one gateway
    base_url="https://your-embedding-endpoint/v1",  # optional; can match llm_client
)

llm = llm_factory("your-llm-model", client=llm_client)
embeddings = OpenAIEmbeddings(model="your-embedding-model", client=embed_client)

faithfulness = Faithfulness(llm=llm)
answer_relevancy = AnswerRelevancy(llm=llm, embeddings=embeddings)
```

在选择指标时，下面这些差异会影响评分调用的准备方式：

方面	差异说明
必需列	每个指标都期望特定参数（例如许多基于参考答案的检索指标需要 `reference`）。在评分开始前，缺失字段会导致验证错误。
LLM vs embeddings vs 都不需要	基于 LLM 的指标需要语言模型；基于 embedding 的指标需要 embedding 模型；词法指标可能不需要任何模型。在现代 API 中，创建指标实例时会显式传入依赖项。
指标变体	不同类以不同评分器实现相同意图（例如基于 LLM 与非 LLM 的上下文精度）。相应地，指标的导入和选择方式也会不同。
构造函数配置	评分准则、aspect critics、离散/数值型自定义指标，或专门的忠实度变体，都需要实例化参数或额外设置。

方面	差异说明
基于 **ID** 或多轮数据	基于 ID 的精度/召回率需要数据集中的 ID 列；多轮或智能体/工具指标需要不同的样本布局。这些不属于单轮 notebook 流程的范围。

在实践中，这意味着主要工作是对齐数据集字段和指标选择，然后使用选定的指标实例对各行进行评分。

前提条件

- 建议使用 Python 3.10+。
- 需要可访问 LLM API 的网络连接（对于需要 embeddings 的指标，还需要可访问 embeddings API）。示例 notebook 假定使用 OpenAI 兼容环境，并支持为兼容网关配置凭据和可选的 base URL。
- 需要注意，评估会发起大量模型调用；成本和延迟会随着行数和指标数量而增长。
- 版本固定：Ragas API 和指标类会在不同版本之间变化。为了获得可复现的基准测试，请在环境或 notebook 中固定 `ragas`（以及相关包）的版本；请参见示例 notebook 中注释掉的安装行。

可运行的 **notebook**

在 JupyterLab 或其他 Jupyter 环境中下载并打开该 notebook：

- ragas-rag-eval.ipynb

该 notebook 以简短的 SDK 回顾开始，重点介绍现代指标（`ragas.metrics.collections`）以及显式的 LLM/embedding 设置。权威说明请参考本页的 调用 **Ragas SDK** 部分。

该 notebook 会：

1. 安装依赖项（包含一个可选的注释版本固定，以保证可复现性）。
2. 创建一个包含 `user_input`、`retrieved_contexts`、`response` 和 `reference` 的小型 `datasets.Dataset`。
3. 使用现代指标类，针对 忠实度 和 答案相关性 运行基线评估。

4. 使用现代指标类添加可选的检索类指标（上下文精度 和 上下文召回率）。
5. 展示汇总结果和逐行结果，然后提供一个简短的故障排查部分。

故障排查

- 凭据或端点配置：配置 LLM API 凭据（以及兼容网关所需的可选 base URL）。如果 embeddings 使用独立端点，也要配置 embeddings 凭据，然后将不同的 `AsyncOpenAI` 客户端分别传入 `llm_factory` 和 `OpenAIEmbeddings`。
- 数据集验证错误：检查所选指标所需的参数，并确保数据集键与现代示例一致（`user_input`、`retrieved_contexts`、`response`、`reference`）。
- **Notebook** 异步执行：示例 notebook 使用 `await metric.ascore(...)`。对于同步脚本，请使用 `metric.score(...)`，或用 `asyncio.run(...)` 包装异步代码。
- 版本相关警告：指标类和签名可能会随 Ragas 版本变化。请固定包版本以确保运行可复现，并结合已安装版本文档确认行为。

结果解读

- 仅在相同的数据集和评估配置下比较分数（评审 LLM、embeddings 和提示词）；否则，变化可能反映的是配置差异，而不是 RAG 质量变化。
- 对于以检索为中心的评估，尽可能使用与生产 RAG 检索器相同的 embedding 模型，以减少由 embedding 空间不匹配引起的指标漂移。
- 将汇总分数用于趋势跟踪或质量门禁，将逐行分数用于诊断（例如缺失上下文、幻觉或无关检索）。应将指标值视为方向性信号，而非绝对真理。

延伸阅读

- Ragas 文档：https://docs.ragas.io/↗
- Ragas GitHub 仓库：https://github.com/vibrantlabsai/ragas↗

用于 LLM 安全的 AI Guardrails

TrustyAI Guardrails Orchestrator 会对 LLM 的输入和输出运行检测器，以过滤或标记内容。它基于开源的 FMS-Guardrails 项目。TrustyAI Operator 提供 **GuardrailsOrchestrator** CRD，用于部署和管理它。

本文仅介绍 **AutoConfig** 部署和内置的 **built-in regex detector**。

目录

前提条件

使用 AutoConfig 部署

资源状态

内置 regex detector

Gateway ConfigMap 结构

服务端口和 API 参考

端口和角色

身份验证（已启用 auth）

请求路径参考

API 用法和示例

独立检测（`/api/v1/text/contents`）

Orchestrator API：按请求选择检测器（`/api/v2/chat/completions-detection`）

Gateway API：预设流水线（`/all/v1/chat/completions`）

前提条件

- 已安装 TrustyAI Operator（请参见 安装 TrustyAI）。
- 已在目标命名空间中将 LLM 作为 InferenceService 部署。

使用 **AutoConfig** 部署

使用 AutoConfig 时，operator 会从命名空间中的资源生成 orchestrator 和 gateway 配置；对于基本设置，不需要手动创建 ConfigMap。

创建一个带有 `autoConfig`、并启用内置检测器和 gateway 的 **GuardrailsOrchestrator** 自定义资源：

```yaml
apiVersion: trustyai.opendatahub.io/v1alpha1
kind: GuardrailsOrchestrator
metadata:
  name: guardrails-orchestrator
  namespace: <your-namespace>
  # Optional: enable auth for routes (Bearer token required)
  annotations:
    security.opendatahub.io/enable-auth: "false"
spec:
  autoConfig:
    inferenceServiceToGuardrail: <inference-service-name>
  enableBuiltInDetectors: true
  enableGuardrailsGateway: true
  replicas: 1
```

- **`inferenceServiceToGuardrail`**：要进行 guardrail 保护的 InferenceService（LLM）名称；必须与同一命名空间中已部署的模型名称匹配。
- **`enableBuiltInDetectors`**：当为 `true` 时，会添加一个内置 regex detector sidecar。
- **`enableGuardrailsGateway`**：当为 `true` 时，gateway 会暴露预设路由（例如 `/all/v1/chat/completions`）。

资源状态

该资源包含一个 `status` 子资源。**`status.phase`** 可以是 `Progressing`、`Ready` 或 `Error`。使用 AutoConfig 时，**`status.autoConfigState`** 会保存生成的 ConfigMap 名称（`generatedConfigMap`、`generatedGatewayConfigMap`）、检测到的服务以及 `message`。只有在 `status.phase == Ready` 且对应的 Deployment 已就绪之后，才应发送流量。

operator 会创建一个 orchestrator ConfigMap 和一个名为 `<orchestrator-name>-gateway-auto-config` 的 gateway ConfigMap。内置检测器会注册为 `built-in-detector`。

内置 regex detector

内置检测器提供基于 regex 的算法。支持的算法包括：

Category	Algorithms
regex	`email`、`us-social-security-number`、`credit-card`、`ipv4`、`ipv6`、`us-phone-number`、`uk-post-code`，或自定义 regex

默认的 gateway 配置使用占位 regex（`$^`）。要启用特定算法（例如 `email`），请修补 ConfigMap，并将 `detector_params.regex` 设置为算法名称（例如 `- email`）。

Gateway ConfigMap 结构

ConfigMap 名称：`<orchestrator-name>-gateway-auto-config`。示例：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: guardrails-orchestrator-gateway-auto-config
  namespace: <your-namespace>
data:
  config.yaml: |
    orchestrator:
      host: "localhost"
      port: 8032
    detectors:
      - name: built-in-detector
        input: true
        output: true
        detector_params:
          regex:
            - $^                          # change this placeholder, e.g.
email
    routes:
      - name: all
        detectors:
          - built-in-detector
      - name: passthrough
        detectors:
```

将 `built-in-detector` 下的 `regex` 改为所需的算法（例如 `- email`）。更新后，请等待 Deployment 就绪。

服务端口和 API 参考

Guardrails Orchestrator 通过名为 `<orchestrator-name>-service` 的 Service 暴露。端口号取决于是否启用身份验证（在 GuardrailsOrchestrator 上添加注解 `security.opendatahub.io/enable-auth: "true"`）。

端口和角色

Port name	Auth disabled	Auth enabled	Role
gateway	8090	8490	Guardrails Gateway：预设检测器流水线和 OpenAI 风格的 chat completion 端点。使用它可发送 chat 请求；在调用 LLM 之前和/或之后，内置（或已配置）的检测器会对请求进行检查。
built-in-detector	8080	8480	内置 regex detector API。独立内容检测（不调用 LLM）。请求体：`contents`（字符串列表）和 `detector_params`（例如 `regex: ["email"]`）。
https (orchestrator)	8032	8432	Orchestrator API：直接访问 orchestrator 端点（例如自定义检测流程、health）。
health	8034	8034	健康检查端点。

启用身份验证后，gateway 和 built-in-detector 端口需要 Bearer 令牌。

身份验证（已启用 **auth**）

发送到 gateway 或 built-in-detector 端口的请求必须包含：

```
Authorization: Bearer <token>
```

该 token 必须是有效的 Kubernetes ServiceAccount token（或集群 auth proxy 接受的其他 token），且其主体对该服务具有访问权限（例如 `services/proxy`）。未授权请求会收到 401/403。

如何获取 **token**

在与 Guardrails Orchestrator 相同的命名空间中创建一个 ServiceAccount、一个 Role（对 `services/proxy` 拥有 `get`、`create` 权限）以及一个 RoleBinding，然后为该 ServiceAccount 创建 token：

```
# Replace <your-namespace> and optionally the ServiceAccount name (e.g. guardrails-client)
kubectl create serviceaccount -n <your-namespace> guardrails-client
kubectl create role -n <your-namespace> guardrails-client --verb=get,create --resource=services/proxy
kubectl create rolebinding -n <your-namespace> guardrails-client --role=guardrails-client --serviceaccount=<your-namespace>:guardrails-client
kubectl create token -n <your-namespace> guardrails-client
```

也可以选择设置 token 有效期，例如使用 `--duration=8760h` 表示一年。最后一条命令会输出 token；将其设置为 `Authorization: Bearer <token>` 标头值。

集群内部的客户端可以使用投影的 ServiceAccount token volume 作为 Bearer token。

请求路径参考

Path	Port	Purpose
`POST /all/v1/chat/completions`	gateway (8090 or 8490)	通过 guardrails 的 chat completions：请求会在输入检查后发送到 LLM，响应会再由检测器检查。请求体：`model`、`messages`（OpenAI 风格）。检测器流水线由 gateway 配置固定。
`POST /api/v2/chat/completions-detection`	orchestrator (8032 or 8432)	带有按请求选择检测器的 chat completions。请求体：`model`、`messages`，以及可选的 `detectors`（例如 `{"input": {"built-in-detector": {"regex": ["email"]}}, "output": {...}}`）。当指定了检测器时，返回模型回复以及 `detections` 和 `warnings`。当调用方需要为每个请求选择运行哪些检测器，而不是使用预设 gateway 路由时，请使用此接口。
`POST /api/v1/text/contents`	built-in-detector (8080 or 8480)	独立内容检测。对给定文本运行内置 regex（或已配置）的检测器；不调用 LLM。请求体：`contents`、`detector_params`。

Path	Port	Purpose
`GET /health`	orchestrator (8032 or 8432)	orchestrator 的健康检查。

其他 gateway 路由（例如 `/<preset-name>/v1/chat/completions`）在 gateway ConfigMap 的 `routes` 中定义。

API 用法和示例

独立检测（`/api/v1/text/contents`）

对文本运行内置 regex detector，而不调用 LLM。使用 **built-in-detector** 端口（8080 或 8480）。请求体：`contents`（字符串列表）、`detector_params`（例如 `regex: ["email"]`）。

使用服务地址（集群内部：`<orchestrator-name>-service.<your-namespace>.svc.cluster.local`；集群外部：若已暴露，则使用 Ingress host）以及 **built-in-detector** 端口（请参见 端口和角色）。

```
# If auth is enabled, set TOKEN. Use port 8480 (auth) or 8080 (no auth).
curl -k -s -X POST "https://<service-address>:8480/api/v1/text/contents"
\
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"contents":["hello, my email is test@example.com"],"detector_param
s":{"regex":["email"]}}'
```

响应： 一个数组，数组中的每个条目对应一个 `contents` 项，且每个条目本身是检测对象数组。每个对象包含 `start`、`end`、`text`、`detection`（例如 `EmailAddress`）、`detection_type`（例如 `pii`）以及 `score`。

▶ 响应示例（独立检测，检测到 email）

Orchestrator API：按请求选择检测器（`/api/v2/chat/completions-detection`）

当调用方必须为每个请求选择运行哪些检测器时，请使用 **orchestrator** 端口（8032 或 8432）。请求体：`model`、`messages`，以及可选的 `detectors`（例如，带有检测器参数的 `input` / `output`）。

示例：对输入和输出都运行 regex 为 `email` 的内置检测器：

```
# Use orchestrator port 8432 (auth) or 8032 (no auth).
curl -k -s -X POST "https://<service-address>:8432/api/v2/chat/completion
s-detection" \
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{
    "model": "<inference-service-name>",
    "messages": [{"content": "my email is test@example.com", "role": "use
r"}],
    "detectors": {
      "input":  { "built-in-detector": { "regex": ["email"] } },
      "output": { "built-in-detector": { "regex": ["email"] } }
    }
  }'
```

当检测器在输入中发现匹配项时（例如 email），响应会包含 `detections` 和 `warnings`，并且 `choices` 为空：

▶ 响应示例（输入触发检测）

响应结构与 gateway chat 相同：`choices`、`detections`、`warnings`。若要进行不带检测的普通 chat completion，请省略 `detectors`。

Gateway API：预设流水线（`/all/v1/chat/completions`）

对于使用固定检测器流水线的 chat，请使用 **gateway** 端口（8090 或 8490）（该流水线在 gateway ConfigMap 中定义）。请求体：`model`、`messages`（OpenAI 风格）。如果需要按

请求选择检测器，请改用 orchestrator API。

使用服务地址和 **gateway** 端口（请参见 端口和角色）。

```
# Use port 8490 (auth) or 8090 (no auth).
curl -k -s -X POST "https://<service-address>:8490/all/v1/chat/completion
s" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"model":"<inference-service-name>","messages":[{"content":"my emai
l is test@example.com","role":"user"}]}'
```

当输入/输出通过时： `detections` 和 `warnings` 为 null， `choices` 包含模型回复：

▶ 响应示例（输入/输出通过）

当输入触发检测时（例如 **PII**）： `detections` 和 `warnings` 会被设置， `choices` 为空：

▶ 响应示例（输入触发检测）

NeMo Guardrails

NeMo Guardrails 为 LLM 应用提供可编程的安全控制。它作为位于模型前方的独立服务运行，并且可以强制执行：

- 敏感数据检测（例如，输入和输出中的 PII）。
- 内容策略（例如，禁止的话题、竞品提及）。
- 使用 Colang 和 Python 编写的自定义校验流程。

TrustyAI Operator 通过 **NemoGuardrails** 自定义资源（CR）公开 NeMo Guardrails。本文重点介绍一个基础部署，它：

- 保护在 serving 平台上已部署的现有模型。
- 使用 NeMo Guardrails 进行输入/输出过滤和简单的业务规则检查。

目录

前提条件

架构

NeMo 配置 ConfigMap

`config.yaml` 基础

`rails.co` 基础

`actions.py` 基础

部署 NemoGuardrails 自定义资源

认证（已启用 auth）

如何获取 token

访问 NeMo Guardrails API
基本 chat completion（允许的内容）
消息长度 guardrail 示例
禁止内容示例
敏感数据检测示例
延伸阅读

前提条件

- 已安装 TrustyAI Operator（参见 安装 TrustyAI）。
- 在 serving 平台上已经部署了一个模型（例如 vLLM），并且该模型暴露了 OpenAI-compatible API。

架构

从高层来看，请求路径如下：

```
Client → NeMo Guardrails service → model predictor (OpenAI-compatible API)
```

NeMo Guardrails：

- 接收 OpenAI 风格的 `chat/completions` 请求。
- 执行已配置的 rails（敏感数据检测、长度检查、禁止话题等）。
- 对于允许的请求，将其转发给底层模型。
- 对于被阻止的请求，在不调用模型的情况下返回适当的 assistant 消息。

TrustyAI Operator 通过 **NemoGuardrails** CR 管理 NeMo Guardrails server 的 Pod 和 Service。随后，可以使用集群中选定的 ingress 或 gateway 方案将该 Service 暴露到外部。

NeMo 配置 ConfigMap

NeMo Guardrails 期望一个配置目录，其中通常包含：

- `config.yaml`：NeMo Guardrails 的主配置文件。
- `rails.co`：实现输入/输出 rails 和其他控制逻辑的 Colang flows。
- `actions.py`：Colang flows 可调用的 Python actions，用于执行自定义逻辑。

▶ NeMo 配置 ConfigMap 示例

`config.yaml` 基础

在上面的示例中，`config.yaml`：

- 在 `models` 部分声明一个后端模型，并通过 `openai_api_base` 和 `model_name` 配置 OpenAI-compatible 端点。
- 在 `rails.config.sensitive_data_detection` 下配置内置的 PII 检测：
  - `input.entities` / `output.entities` 列出需要保护的实体类型（例如，`EMAIL_ADDRESS`、`PERSON`）。
  - 当 `detect sensitive data on input` / `detect sensitive data on output` rails 运行时，NeMo 会使用此配置自动调用其内部检测器。
- 通过 `rails.input.flows` 和 `rails.output.flows` 定义哪些 rails 运行，以及运行顺序：
  - `detect sensitive data on input` / `detect sensitive data on output` 是由 `sensitive_data_detection` 支持的内置 rails。
  - `check message length` 和 `check forbidden words` 是在 `rails.co` 中实现、并由 `actions.py` 中的 Python actions 支持的自定义 rails。
- 将 `<model-predictor-host>`、`<port>` 和 `<model-name>` 替换为实际的 predictor service URL 和模型名称。
- 确保后端 predictor 实现了 OpenAI-compatible 的 `/v1/chat/completions` API。
- 如需了解 `config.yaml` 的更高级配置（其他 rail 类型、prompts、tracing、knowledge base，以及与其他安全提供方的集成），请参考官方 NeMo Guardrails YAML 配置说明：Nvidia NeMo Guardrails Configuration↗。

`rails.co` 基础

在此示例中，`rails.co` 定义了 两个自定义输入 **rails**：

- `define flow check message length`：
  - flow 名称 `check message length` 必须与 `config.yaml` 中 `rails.input.flows` 的条目匹配。
  - `$length_result = execute check_message_length` 会运行来自 `actions.py` 的 Python action `check_message_length`，并传入当前对话上下文。
  - `if` 语句根据返回的字符串进行分支，或者：
    - 调用 `bot ...` block 发送回复（例如 `bot inform message too long`），并且
    - `stop` 以中止后续处理并防止调用 LLM，
    - 或者在返回 `"allowed"` 时不执行任何操作，并允许流水线继续到下一个 rail。
- `define flow check forbidden words`：
  - 使用相同模式，但调用 `check_forbidden_words` action，仅在返回值不是 `"allowed"` 时进行阻止。

补充说明：

- `bot ...` blocks（例如 `bot inform message too long`）定义了预设的 assistant 消息；当某个 rail 决定停止流水线时，这些消息会直接发送给客户端，而不会联系后端 LLM。
- 在 `rails.co` 中定义的 rails 会按照 `rails.input.flows` / `rails.output.flows` 中列出的顺序执行。像 `detect sensitive data on input` 这样的内置 rails，会根据它们在列表中的位置在自定义 rails 之前或之后运行。
- 此处展示的 Colang 是一个最小示例。它支持更复杂的 flows（多步骤、变量、附加 actions）；完整语法和能力请参阅 NeMo Guardrails 文档中的 Colang 参考。一个很好的起点是 Colang 2.0 入门指南：Colang Getting Started↗。

`actions.py` 基础

`actions.py` 文件包含带有 `@action` 装饰器的 Python 函数，Colang flows 可以通过 `execute <action_name>` 调用这些函数：

- Actions 接收一个 `context` 对象，它是由 NeMo 填充的类 dict 结构（例如，在 `"user_message"` 下包含最新的用户消息）。
- Actions 返回一个值（通常是简短字符串），由 Colang flows 进行解释并据此分支。

在此示例中：

- `check_message_length`：
  - 检查 `context["user_message"]`，计算词数，并返回：
    - 当消息应被拒绝时返回 `"blocked_too_long"`。
    - 当需要警告但流水线仍可继续时返回 `"warning_long"`。
    - 当消息长度可接受时返回 `"allowed"`。
- `check_forbidden_words`：
  - 将用户消息转换为小写，搜索禁止词，并：
    - 在未发现任何内容时返回 `"allowed"`。
    - 在存在禁止词时返回非 `"allowed"` 值（例如 `"blocked_password"`）。

这些模式可以扩展，以实现更复杂的 guardrails，例如结构化检查、数值阈值或调用外部服务。

部署 **NemoGuardrails** 自定义资源

在 ConfigMap 和 token Secret 就绪后，创建一个 **NemoGuardrails** CR 来部署 NeMo Guardrails service：

```yaml
apiVersion: trustyai.opendatahub.io/v1alpha1
kind: NemoGuardrails
metadata:
  name: nemo-guardrails
  namespace: <your-namespace>
  annotations:
    # When true, the exposed route requires a Bearer token for incoming r
equests to NeMo Guardrails.
    security.opendatahub.io/enable-auth: "true"

    # When the backend LLM is exposed over HTTPS with a custom CA, set th
is annotation
    # to the name of a Secret that contains the CA bundle in a key such a
s `ca.crt`.
    # The operator mounts this Secret and configures NeMo Guardrails TLS
trust accordingly.
    # Example:
    # trustyai.opendatahub.io/ca-secret-name: llm-backend-ca
spec:
  nemoConfigs:
    - name: nemo-config
      configMaps:
        - nemo-config
      default: true
  env:
    - name: OPENAI_API_KEY
      # For authenticated backends, use a Secret-ref token:
      # valueFrom:
      #   secretKeyRef:
      #     name: api-token-secret
      #     key: token
      # For internal, unauthenticated HTTP backends, a placeholder value
is sufficient:
      value: "<placeholder>"

    # Optional: offline environments
    # NeMo Guardrails may fetch the Public Suffix List via tldextract. In
environments
    # without Internet access, set TLDEXTRACT_CACHE to use the cached
    # public_suffix_list data bundled in the NeMo Guardrails Server imag
e. Note that
    # the bundled list may not be the latest.
    # - name: TLDEXTRACT_CACHE
```

```
    #   value: "/app/.cache/"

    # TLS behaviour for backend LLM:
    # - HTTP backends:
    #   * Set SSL_CERT_FILE to an empty string to disable certificate loo
kup.
    #   * Use an http:// URL in config.yaml (openai_api_base).
    # - HTTPS backends with a custom CA:
    #   * Remove SSL_CERT_FILE from env.
    #   * Add the trustyai.opendatahub.io/ca-secret-name annotation abov
e, pointing
    #     to a Secret that contains the CA certificate bundle.
    - name: SSL_CERT_FILE
      value: ""
```

关键字段：

- `nemoConfigs`：引用一个或多个配置包；每个配置包可以映射到一个或多个包含 NeMo Guardrails 配置文件的 ConfigMap。
- `env.OPENAI_API_KEY`：NeMo Guardrails 用于对后端模型端点进行身份验证的 token（例如 vLLM service）。对于内部的、未认证的推理服务，该值可以直接设置为 `value: "<placeholder>"`，后端不会使用它。对于仅 HTTP 的推理服务，后端 URL 不需要 TLS 证书。
- `security.opendatahub.io/enable-auth`：当设置为 `"true"` 时，NeMo Guardrails 的路由会受到集群认证保护，并且需要 Bearer token。

应用：

```
kubectl apply -f nemo-guardrails-cr.yaml -n <your-namespace>
```

CR 创建后，operator 会进行协调并创建：

- 用于 NeMo Guardrails server 的 Deployment。
- 一个在集群内部暴露 NeMo Guardrails HTTP 端点的 Service。

等待 Deployment Pod 变为 `Ready`：

```
kubectl get pods -n <your-namespace> -l app.kubernetes.io/name=nemo-guardrails
```

认证（已启用 **auth**）

当在 NeMo Guardrails 前启用 HTTP 认证时，该 service 会要求传入请求携带 Bearer token。

如何获取 **token**

在与 NemoGuardrails resource 相同的 namespace 中创建一个 ServiceAccount、一个 Role（包含对 `services/proxy` 的 `get`、`create` 权限）以及一个 RoleBinding；然后为该 ServiceAccount 创建 token：

```
# Replace <your-namespace> and optionally the ServiceAccount name (e.g. nemo-guardrails-client)
kubectl create serviceaccount -n <your-namespace> nemo-guardrails-client
kubectl create role -n <your-namespace> nemo-guardrails-client --verb=get,create --resource=services/proxy
kubectl create rolebinding -n <your-namespace> nemo-guardrails-client --role=nemo-guardrails-client --serviceaccount=<your-namespace>:nemo-guardrails-client
kubectl create token -n <your-namespace> nemo-guardrails-client
```

也可以选择设置 token 时长，例如使用 `--duration=8760h` 表示一年。最后一条命令会输出 token；将其设置为 `Authorization: Bearer <token>` header 的值。

访问 **NeMo Guardrails API**

NeMo Guardrails 暴露一个 OpenAI 风格的 chat completions 端点：

- `POST /v1/chat/completions`

使用首选的 ingress 或 gateway 机制（例如 Ingress resource 或 API gateway）暴露 NeMo Guardrails Service，并记录公网 host 和端口：

- 未启用 auth：该 Service 通常以 **HTTP** 方式暴露在端口 **80**。
- 启用 auth：该 Service 通常以 **HTTPS** 方式暴露在端口 **443**。

相应设置 base URL，例如：

```
# No auth (HTTP on 80)
NEMO_GUARDRAILS_URL="http://<nemo-guardrails-host>"

# Auth enabled (HTTPS on 443)
# NEMO_GUARDRAILS_URL="https://<nemo-guardrails-host>"
```

基本 **chat completion**（允许的内容）

请求示例：

```
curl -k -X POST "$NEMO_GUARDRAILS_URL/v1/chat/completions" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{
    "model": "<model-name>",
    "messages": [
      { "role": "user", "content": "hello" }
    ]
  }'
```

典型响应：

```
{
  "messages": [
    {
      "role": "assistant",
      "content": "Hello! How can I assist you today?"
    }
  ]
}
```

消息长度 **guardrail** 示例

`check_message_length` flow 及其对应的 Python action 实现了一个基于长度的简单 guardrail。当用户消息过长时，该 rail 会直接回复，而不会调用后端 LLM：

```
curl -k -X POST "$NEMO_GUARDRAILS_URL/v1/chat/completions" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{
    "model": "<model-name>",
    "messages": [
      {
        "role": "user",
        "content": "This is a very long message that should be considered
far too long for the purposes of this Nemo Guardrails end-to-end test, so
it should clearly exceed the configured word limit and trigger the length
-based blocking behaviour."
      }
    ]
  }'
```

响应由 NeMo Guardrails 生成，而不会调用后端模型：

```
{
  "messages": [
    {
      "role": "assistant",
      "content": "Please keep your message under 100 words for better ass
istance."
    }
  ]
}
```

禁止内容示例

禁止话题由 `check_forbidden_words` action 及其 Colang flow 控制。当用户消息包含如 `"hack"` 或 `"password"` 之类的禁止词时，该 rail 会阻止请求：

```
curl -k -X POST "$NEMO_GUARDRAILS_URL/v1/chat/completions" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{
    "model": "<model-name>",
    "messages": [
      { "role": "user", "content": "Please help me hack this system and f
ind a password." }
    ]
  }'
```

响应由 NeMo Guardrails 生成，而不会调用后端模型：

```
{
  "messages": [
    {
      "role": "assistant",
      "content": "I can't help with that type of request. Please ask some
thing else."
    }
  ]
}
```

敏感数据检测示例

敏感数据检测在 `config.yaml` 的 `rails.config.sensitive_data_detection` 下进行配置。在此示例配置中，输入和输出检测都会标记 `EMAIL_ADDRESS`。

包含电子邮件地址的示例输入：

```
curl -k -X POST "$NEMO_GUARDRAILS_URL/v1/chat/completions" \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $TOKEN" \
  -d '{
    "model": "<model-name>",
    "messages": [
      { "role": "user", "content": "My email is test@example.com" }
    ]
  }'
```

典型响应：

```
{
  "messages": [
    {
      "role": "assistant",
      "content": "I don't know the answer to that."
    }
  ]
}
```

在这种情况下，内置的敏感数据检测 rail 已检测到用户消息中的电子邮件地址，NeMo Guardrails 会返回一个安全的兜底回复，而不是让后端模型返回可能不安全的答案。

延伸阅读

如需更全面了解 NeMo Guardrails 库（使用场景、架构以及生态集成），请参阅官方文档：Overview of NVIDIA NeMo Guardrails Library↗。

组件

本章所依赖的组件的介绍和安装指南。

TrustyAI

TrustyAI 的 Alauda Build

TrustyAI

文档

安装 TrustyAI

上架 Operator 包

启用 TrustyAI

验证安装

后续步骤

部署 Trusty

前提条件

部署 TrustyAISe

验证部署就绪情

下一步

Alauda 版 TrustyAI

TrustyAI 的 Alauda Build

TrustyAI

文档

安装 TrustyAI

上架 Operator 包

启用 TrustyAI

验证安装

后续步骤

部署 Trusty

前提条件

部署 TrustyAISe

验证部署就绪情

下一步

TrustyAI 的 Alauda Build

目录

TrustyAI

文档

TrustyAI

Alauda Build of TrustyAI 基于 TrustyAI Kubernetes operator。 该 operator 可简化在 Kubernetes 上部署和管理 TrustyAI 组件，包括模型可解释性、公平性监控、LLM 评估以及 AI guardrails。

主要组件和能力包括：

- **TrustyAI Service (TAS)**：与 KServe 模型一同部署并收集推理数据的服务。它支持模型可解释性、公平性监控和漂移跟踪。有关部署，请参见 部署 TrustyAI Service；有关数据摄取和指标，请参见 偏差和漂移监控。
- **LM-Eval**：一种基于作业的架构，用于部署和管理 LLM 评估，基于 EleutherAI 的 lm-evaluation-harness。该 operator 提供 **LMEvalJob** CRD，用于运行和管理评估任务。详情请参见 评估 LLM。
- **AI Guardrails (FMS-Guardrails)**：用于为 LLM 提供 guardrail 的模块化框架。该 operator 提供 **GuardrailsOrchestrator** CRD，用于编排 guardrail 策略和组件。详情请参见 面向 LLM 安全的 AI Guardrails。

- **NeMo Guardrails**：与 NVIDIA NeMo Guardrails 集成，使用 YAML 和 Colang 定义用于输入/输出过滤的可编程 rails。该 operator 提供 **NemoGuardrails** CRD，可将 NeMo Guardrails 作为服务部署到集群中。详情请参见 NeMo Guardrails。

有关在平台上安装，请参见 安装 TrustyAI。

安装后如需使用 LMEvalJob 和 GuardrailsOrchestrator，请参见上面链接的各节。

文档

TrustyAI 上游文档和 operator 仓库：

- **TrustyAI Operator (upstream)**：https://github.com/trustyai-explainability/trustyai-service-operator↗ — Kubernetes operator 的概述、安装和使用。
- **TrustyAI Documentation**：https://trustyai.org/↗ — TrustyAI 项目文档。

安装 TrustyAI

在 Alauda AI 2.8.x 中，**Alauda Build of TrustyAI** 由 Alauda AI 安装和管理。集群范围的 `default` `AmlCluster` 中的 `trustyAI` 组件默认值为 `Unmanaged`；将其设置为 `Managed` 后，无需手动从 OperatorHub 安装 Operator。组件完成协调后，即可创建 TrustyAI 自定义资源以使用其功能。

目录

上架 Operator 包

启用 TrustyAI

验证安装

后续步骤

上架 Operator 包

下载 TrustyAI Operator 包，然后按照 上架软件包 的说明上传该包。

启用 TrustyAI

在 **Administrator** 视图中：

1. 打开 **Marketplace** > **OperatorHub**。

2. 选择目标集群并打开 **Alauda AI**。
3. 打开 **All Instances** 选项卡，编辑 `default` `AmlCluster` 实例。
4. 将 `spec.components.trustyAI.managementState` 设置为 `Managed`。
5. 保存 `AmlCluster` 实例并等待 TrustyAI Operator 就绪。

你可以在 `AmlCluster` YAML 中应用该组件设置：

```yaml
spec:
  components:
    trustyAI:
      managementState: Managed
```

验证安装

检查 `AmlCluster` 状态和 TrustyAI Operator：

```bash
kubectl get amlcluster default

kubectl get pods -n trustyai-operator-system
```

`AmlCluster` 应报告 `Phase=Ready`，并且 TrustyAI Operator 控制器 pod 应处于 `Running` 状态。TrustyAI CRD 会在 operator 就绪后注册。

后续步骤

启用该组件后，用户可以使用 **TrustyAIService**、**LMEvalJob**、**GuardrailsOrchestrator** 和 **NemoGuardrails** 等 CRD 来部署推理监控（TrustyAI Service）、运行模型评估任务，以及管理 AI guardrails 编排（包括 FMS Guardrails 和 NeMo Guardrails）。

有关详细信息，请参阅 部署 TrustyAI Service、偏差和漂移监控、评估 LLM、面向 LLM 安全的 AI Guardrails 和 NeMo Guardrails。

部署 TrustyAI Service

TrustyAI Service (TAS) 会与 KServe 模型一起部署，并收集其推理数据，漂移检测和偏差指标就是基于这些数据计算的。本页面会创建 `TrustyAIService` 资源；有关数据摄取以及为其注册指标的内容，请参阅 Monitor 章节中的 Bias and Drift Monitoring。

目录

前提条件

部署 TrustyAIService

DATABASE 模式

PVC 模式

验证部署就绪情况

下一步

前提条件

- 已安装 TrustyAI Operator（请参阅 Install TrustyAI）。
- 如果使用 `storage.format: DATABASE`，则需要一个 MySQL 8.x 数据库。
- 如果使用 `storage.format: PVC`，请确保集群具有可用的默认 `StorageClass`（由 CSI driver 提供支持），用于动态卷供应，这样 operator 创建的 PVC 才能成功绑定。

部署 TrustyAIService

请选择一种存储布局：**DATABASE**（MySQL）或 **PVC**（卷上的本地文件存储）。下面两个小节是替代方案，不是同一流程中的步骤。

DATABASE 模式

MySQL 凭据 Secret

在使用 `storage.format: DATABASE` 时，创建一个包含 TAS 部署所需键值的 secret：

```
apiVersion: v1
kind: Secret
metadata:
  name: <tas-name>-db-credentials
  namespace: <your-namespace>
type: Opaque
stringData:
  databaseKind: mysql
  databaseUsername: <username>
  databasePassword: <password>
  databaseService: <mysql-service-name>
  databasePort: "3306"
  databaseName: <db-name>
  # Database schema generation strategy used by TrustyAI when connecting
to the database.
  # It controls what TAS does to the database schema (tables) during star
tup:
  # - none: do not manage schema
  # - create: create schema from scratch
  # - drop-and-create: drop existing schema, then create
  # - drop: drop existing schema
  # - update: update schema to match the expected model
  # - validate: only validate that the schema matches the expected model
  #
  # Default: update
  databaseGeneration: update
```

说明：

- `databaseName` 引用的 MySQL schema（database）必须预先创建。TAS 不会自行创建数据库。
- 数据库必须能够从 TAS pod 访问到。
- `databaseGeneration` 控制 TAS 启动时如何处理 schema 变更。

TrustyAIService CR

示例：

```yaml
apiVersion: trustyai.opendatahub.io/v1
kind: TrustyAIService
metadata:
  name: <tas-name>
  namespace: <your-namespace>
  annotations:
    trustyai.cpaas.io/monitor-enable: "true"
    trustyai.cpaas.io/monitor-interval: "30s"
    trustyai.cpaas.io/monitor-metric-regex: "^trustyai_.*"
spec:
  storage:
    format: DATABASE
    databaseConfigurations: <tas-name>-db-credentials
  metrics:
    schedule: "5s"
    batchSize: 5000
  replicas: 1
```

在 `DATABASE` 模式下，`storage.databaseConfigurations` 必须设置为上面创建的 MySQL 凭据 Secret 的名称，并且该 Secret 需要与 `TrustyAIService` 位于同一 namespace 中。

`metadata.annotations` 是可选的，用于允许 operator 创建 `ServiceMonitor`，以供 Prometheus 抓取（从而使平台能够自动收集监控数据）。

- 当设置 `trustyai.cpaas.io/monitor-enable: "true"` 时，operator 会生成一个 `ServiceMonitor`。
- `trustyai.cpaas.io/monitor-interval` 和 `trustyai.cpaas.io/monitor-metric-regex` 是可选项；未提供时，operator 会使用默认值。

`trustyai.cpaas.io/monitor-interval` 控制 Prometheus 抓取 TAS 指标的频率（默认值：`30s`）。`trustyai.cpaas.io/monitor-metric-regex` 控制抓取后保留哪些指标名称（默认值：`^trustyai_.*`）。

`spec.metrics` 字段：

- `schedule`（必填）：TAS 执行指标计算的频率（例如每 `5s` 一次）。该值为持续时间字符串。
- `batchSize`（可选）：TAS 在每次指标计算中包含多少条推理记录（值越大，每次运行使用的数据越多）。如果未设置，operator 会使用默认值 `5000`。

PVC 模式

当 `storage.format` 为 **PVC**（无需 MySQL Secret）时，使用此路径。示例：

```yaml
apiVersion: trustyai.opendatahub.io/v1
kind: TrustyAIService
metadata:
  name: <tas-name>
  namespace: <your-namespace>
spec:
  storage:
    format: PVC
    folder: /inputs
    size: 1Gi
  data:
    filename: data.csv
    format: CSV
  metrics:
    schedule: "5s"
    batchSize: 5000
  replicas: 1
```

在 `PVC` 模式下：

- `storage.folder`：TAS 在已挂载 PVC 中存储和读取数据的路径。
- `storage.size`：请求的 PVC 容量（例如 `1Gi`）。

- operator 会自动在与 `TrustyAIService` 相同的 namespace 中创建名为 `<tas-name>-pvc` 的 PVC。由于该 PVC 使用集群默认 `StorageClass`（未显式设置 `storageClassName`），因此集群应提供可用的默认 `StorageClass`（否则 PVC 可能会一直处于 `Pending` 状态）。

当所选模式的 `TrustyAIService` manifest 准备就绪后，应用它（同一命令适用于 `DATABASE` 或 `PVC` YAML）：

```
kubectl apply -f <trustyai-service>.yaml -n <your-namespace>
```

验证部署就绪情况

```
kubectl get trustyaiservices -n <your-namespace> <tas-name>
```

预期的 `status.phase` 应为 `Ready`。

同时检查 pods：

```
kubectl get pods -n <your-namespace> -l app.kubernetes.io/instance=<tas-name>
```

下一步

当服务处于 `Ready` 状态后，继续阅读 Bias and Drift Monitoring，以摄取 reference 和 live 推理数据、注册漂移和偏差指标，并将其暴露给 Prometheus。

监控与运维

本章涵盖推理服务的日志记录和链路追踪、资源监控和仪表板，以及模型偏差和漂移监控。它们共同为推理服务的整个生命周期提供可观测性：实时从节点日志，以及面向基础设施、GPU 资源、令牌吞吐量和 API 流量的多维仪表板。日志记录和资源监控面向推理服务用户，而配置监控仪表板则需要管理员视图。

偏差和漂移监控由 TrustyAI 提供，相关内容位于 Evaluate & Safety 章节；它所需的 `TrustyAIService` 部署在该章节中介绍，详见 Deploy TrustyAI Service。

日志与链路追踪

介绍

日志

功能概览

核心功能

访问日志

使用 Find 功能

导出日志

资源监控

介绍

监控指标和视图

添加监控面板

使用限制 功能概述 步骤

主要特性

监控待处理工作负载

1. API Priority and Fairness
2. 提供用户权限
3. 按需监控待处理工作负载

监控面板卡在

偏差与漂移监控

偏差与漂移监控

概览

前提条件

访问 TAS API

数据摄取

数据漂移指标

偏差指标

日志与链路追踪

介绍

日志

功能概览

核心功能

访问日志

使用 Find 功能

导出日志

介绍

Alauda AI 的 **Logging** 模块是 Monitoring & Ops 套件中的实时日志解决方案，专为 MLOps/LLMOps/GenOps 工作流中的推理服务而设计。它能够即时查看支撑 AI 服务的从节点 pod 的运行状态，从而实现高效调试和可观测性。通过以毫秒级延迟流式传输容器日志，并提供内置分析工具，它帮助用户维护服务健康，同时加速故障响应。

日志

目录

功能概览
核心功能
访问日志
步骤 1：导航到 Inference Service
步骤 2：打开日志界面
步骤 3：选择一个 pod
使用 Find 功能
步骤 1：启用搜索
步骤 2：输入搜索词
步骤 3：浏览结果
导出日志
步骤 1：开始导出
步骤 2：下载文件

功能概览

Alauda AI 中的日志功能可对 Inference Service pod 生成的日志进行实时监控和分析。此功能可帮助用户排查问题、跟踪服务行为，并通过即时访问关键日志数据来确保运行效率。它对于调试模型推理错误尤其有用。

核心功能

- 实时日志流式传输：在所选 pod 生成日志时自动显示。
- **Replica Pod** 选择：在不同 pod 之间切换，以查看来自特定 Replica 的日志。
- 关键字搜索（**Find**）：通过精确或部分关键字匹配定位特定日志条目。
- 日志导出（**Export**）：将日志下载为 `.txt` 格式，以便离线分析或归档。

访问日志

按照以下步骤查看 Inference Service 的日志：

1 步骤 1：导航到 Inference Service

1. 在左侧导航窗格中，进入 **Inference Services**。
2. 单击目标 Inference Service 名称以打开其详情页。

2 步骤 2：打开日志界面

1. 在 Inference Service 名称下方的标签栏中，选择 **Logging** 标签。

3 步骤 3：选择一个 pod

1. 使用 **Replica** 下拉菜单选择一个 pod。
2. 日志查看器将自动显示并流式传输所选 pod 的日志。

使用 Find 功能

通过关键字匹配高效搜索日志：

1 步骤 1：启用搜索

1. 单击日志查看器右上角的 **Find** 按钮。

2 步骤 2：输入搜索词

1. 在搜索字段中输入关键字或短语。

3 步骤 3：浏览结果

1. 匹配项会以黄色高亮显示。
2. 使用 ↑ 和 ↓ 按钮在匹配项之间跳转。

导出日志

导出日志以便离线存储和分析：

1 步骤 1：开始导出

1. 单击日志查看器右上角的 **Export** 按钮。

2 步骤 2：下载文件

1. 浏览器将自动把日志文件下载到本地系统。

资源监控

介绍

使用限制

监控指标和视图

功能概述

主要特性

添加监控面板

步骤

监控待处理工作负载

1. API Priority and Fairness
2. 提供用户权限
3. 按需监控待处理工作负载

监控面板卡在

介绍

Resource Monitoring 是 Alauda AI 的 Monitoring & Ops 模块的核心组件，专门用于跟踪和分析推理服务的资源利用率指标。作为全栈 MLOps 平台的一部分，它可提供基础设施资源消耗的实时可视性，帮助用户优化模型部署、避免资源瓶颈，并确保 AI 工作负载稳定运行。Resource Monitoring 与 Alauda AI 统一的监控生态系统集成，将可操作的洞察直接提供到您的 MLOps 工作流中，从而消除对分散工具的依赖。

目录

使用限制

使用限制

使用 Resource Monitoring 时，请注意以下限制：

- 数据采集间隔
  - 最小指标抓取间隔：60 秒
  - 历史数据保留时间：默认 7 天
- 依赖要求
  - 需要在目标集群中部署 Prometheus/VictoriaMetrics 监控栈
  - Node exporter 必须运行在所有工作节点上

    - DCGM exporter 必须运行在所有 GPU 节点上

- 版本要求

    - Hami 的 GPU 仪表板仅在 Alauda AI 1.4 及更高版本中支持

监控指标和视图

目录

功能概述

主要特性

访问资源监控

步骤 1：进入推理服务详情页

步骤 2：打开监控 dashboard

步骤 3：选择时间范围

监控指标

CPU Usage

CPU Utilization

Memory Usage

Memory Utilization

GPU Usage

GPU Utilization

GPU Memory Usage

GPU Memory Utilization

Token Metrics

Request Metrics

功能概述

Alauda AI 的 Monitoring & Ops 模块中的资源监控可为您的推理服务提供 CPU、内存、GPU、令牌使用情况以及请求指标的实时洞察。此功能可帮助您识别性能瓶颈、优化资源分配，并确保服务稳定运行。它尤其适用于以下场景：

- 性能调优：诊断高资源使用情况并调整资源限制。
- 异常检测：监控资源消耗和请求模式的突发峰值。
- 容量规划：分析历史趋势，以高效扩展资源。
- 成本优化：跟踪令牌使用情况和 GPU 利用率，以便进行预算管理。

主要特性

- 资源监控：
  - CPU Usage：显示 CPU 的绝对使用量（例如，已使用的 cores）。
  - CPU Utilization：以分配资源的百分比显示 CPU 使用率。
  - Memory Usage：跟踪实际内存消耗（例如，GB）。
  - Memory Utilization：以分配资源的百分比显示内存使用率。
- 计算监控：
  - GPU Usage：跟踪 GPU 计算资源消耗。
  - GPU Utilization：以分配资源的百分比显示 GPU 使用率。
  - GPU Memory Usage：监控 GPU 内存消耗。
  - GPU Memory Utilization：以分配资源的百分比显示 GPU 内存使用率。
  - 注意：MPS 部署模式不支持 GPU 计算和内存监控。
- 其他监控：
  - Token Metrics：跟踪 prompt 和 generation tokens（适用于 'vllm' runtime 服务）。
  - Request Metrics：监控响应时间（avg/tp50/p90/p95）、QPS（success/fail）以及流量（in/out）。
- 时间范围选择：可按自定义周期分析指标（从 30 分钟到 7 天）。

访问资源监控

1 步骤 1：进入推理服务详情页

1. 在左侧导航栏中点击 **Inference Services**。
2. 点击目标 Inference Service 名称，打开其详情页。

2 步骤 2：打开监控 dashboard

1. 选择 **Monitor** 选项卡。
2. 确保 **Resource Monitor** 区域已展开（默认视图）。

3 步骤 3：选择时间范围

使用右上角的时间选择器选择预设范围或自定义范围：

预设选项	自定义范围
最近 30 分钟	开始/结束日期时间
最近 1 小时	
最近 6 小时	
最近 24 小时	
最近 2 天	
最近 7 天	

监控指标

CPU Usage

- 描述：显示服务实际消耗的 CPU cores。

- 数据格式：`cores`（float）

CPU Utilization

- 描述：已使用的分配 CPU 资源百分比。
- 计算方式：`(Used Cores / Allocated Cores) × 100%`
- 解读：
  - 持续 >90%：考虑扩展 CPU 分配
  - <20%：可能存在资源过度分配

Memory Usage

- 描述：服务消耗的物理内存。
- 数据格式：`GiB` 或 `MiB`
- 重要提示：当使用量超过分配内存时，Kubernetes 会触发 OOM kill。

Memory Utilization

- 描述：已使用的分配内存资源百分比。
- 计算方式：`(Used Memory / Allocated Memory) × 100%`

GPU Usage

- 描述：服务消耗的 GPU 计算资源。
- 数据格式：计算单元
- 注意：MPS 部署模式下不可用。

GPU Utilization

- 描述：已使用的分配 GPU 计算资源百分比。
- 计算方式：`(Used GPU / Allocated GPU) × 100%`
- 注意：MPS 部署模式下不可用。

GPU Memory Usage

- 描述：服务消耗的 GPU 内存。
- 数据格式：`GiB` 或 `MiB`
- 注意：MPS 部署模式下不可用。

GPU Memory Utilization

- 描述：已使用的分配 GPU 内存百分比。
- 计算方式：`(Used GPU Memory / Allocated GPU Memory) × 100%`
- 注意：MPS 部署模式下不可用。

Token Metrics

- **Token Prompt**：跟踪已处理的 prompt tokens 数量。
- **Token Generation**：监控模型生成的 tokens 数量。
- 可用性：仅适用于使用 'vllm' runtime 的推理服务。

Request Metrics

- **Response Time**：衡量服务响应延迟（avg/tp50/p90/p95）。
- **QPS (Queries Per Second)**：跟踪每秒成功和失败请求数。
- **Traffic**：监控入站和出站数据传输。

添加监控面板

目录

步骤

导航到管理员视图

打开监控面板

配置监控面板信息

编辑监控面板指标

步骤

1 导航到管理员视图

1. 打开页面顶部的产品视图选择器。
2. 选择 管理员视图，切换到管理员界面。

2 打开监控面板

1. 在左侧导航窗格中，依次进入 **Operations Center** → **Monitor** → **Dashboards**。
2. 点击右上角的 **Create** 按钮。

3 配置监控面板信息

1. 提供监控面板名称并选择一个文件夹。
2. 将标签 `cpaas.io/published: "true"` 添加到监控面板中。你可以通过点击监控面板右上角的 **YAML** 按钮，然后编辑资源 YAML 来完成此操作。

> **NOTE**
>
> 文件夹必须设置为 **AML**。

4 编辑监控面板指标

1. 创建监控面板后，点击右上角的 **Edit** 按钮进行修改。
2. 在 **Add Method** 下选择 **Native** 单选按钮。你可以使用下面的示例查询，并将其粘贴到 **PromQL** 字段中。

```
100 *
(
  sum by (deployment)(
    rate(container_cpu_usage_seconds_total{
      namespace="$namespace",
      pod=~"qwen.*"
    }[5m])
  )
  /
  sum by (deployment)(
    kube_pod_container_resource_limits_cpu_cores{
      namespace="$namespace",
      pod=~"qwen.*"
    }
  )
)
```

创建监控面板后，你可以在 **Observe** → **Dashboards** 下的 **Alauda AI View** 中通过点击 **Switch** 按钮切换到该监控面板。

监控待处理工作负载

Alauda Build of Kueue 提供 `VisibilityOnDemand` 功能用于监控待处理工作负载。工作负载是指运行后会完成的应用。它可以由一个或多个 pod 组成，这些 pod 可以松散耦合或紧密耦合，作为一个整体完成一项任务。工作负载是 Alauda Build of Kueue 中的准入单位。

`VisibilityOnDemand` 功能为批处理管理员提供了监控集群队列和本地队列中待处理作业流水线的能力，而批处理用户则只能监控本地队列，并帮助用户估计其作业何时会开始运行。

您可以对入站请求和高请求量进行控制，并为查看待处理工作负载提供用户权限。

目录

1. API Priority and Fairness
2. 提供用户权限
3. 按需监控待处理工作负载
   3.1. 前提条件
   3.2. 操作步骤
      3.2.1. 查看 ClusterQueue 中的待处理工作负载
      3.2.2. 查看 LocalQueue 中的待处理工作负载

---

1. API Priority and Fairness

Alauda Build of Kueue 使用 Kubernetes API Priority and Fairness (APF) 来帮助管理待处理工作负载。APF 是一种流量控制机制，可让您定义 API 级别的策略，以规范发往 API server 的入站请求。它可以防止 API server 因意外的高请求量而过载，同时保护关键流量不受尽力而为型工作负载的限流影响。

示例

```yaml
apiVersion: flowcontrol.apiserver.k8s.io/v1
kind: FlowSchema
metadata:
  name: kueue-visibility
spec:
  distinguisherMethod:
    type: ByUser
  matchingPrecedence: 9000
  priorityLevelConfiguration:
    name: kueue-visibility
  rules:
    - resourceRules:
        - apiGroups:
            - 'visibility.kueue.x-k8s.io'
          clusterScope: true
          namespaces:
            - '*'
          resources:
            - '*'
          verbs:
            - '*'
      subjects:
        - group:
            name: system:unauthenticated
          kind: Group
        - group:
            name: system:authenticated
          kind: Group
---
apiVersion: flowcontrol.apiserver.k8s.io/v1
kind: PriorityLevelConfiguration
metadata:
  name: kueue-visibility
spec:
  limited:
    lendablePercent: 90
    limitResponse:
      queuing:
        handSize: 4
        queueLengthLimit: 50
        queues: 16
      type: Queue
    nominalConcurrencyShares: 10
```

```
  type: Limited
```

其他资源

API Priority and Fairness ↗

2. 提供用户权限

您可以为 Alauda Build of Kueue 部署中的用户配置基于角色的访问控制（RBAC）对象。这些对象决定了哪些类型的用户可以创建哪些类型的 Alauda Build of Kueue 对象。

您需要为需要访问特定 API 的用户授予权限。

- 如果用户需要访问来自 `ClusterQueue` 资源的待处理工作负载，则需要创建一个引用 ClusterRole `kueue-batch-admin-role` 的 ClusterRoleBinding 资源。
- 如果用户需要访问来自 `LocalQueue` 资源的待处理工作负载，则需要创建一个引用 ClusterRole `kueue-batch-user-role` 的 RoleBinding 资源。

3. 按需监控待处理工作负载

要测试待处理工作负载的监控功能，您必须正确配置 `ClusterQueue` 和 `LocalQueue` 资源。完成后，您可以在该 `LocalQueue` 上创建作业。Kueue 会管理从作业创建的 workload 对象，因此，当提交的作业使 `ClusterQueue` 达到饱和时，其对应的 workload 就会显示在待处理工作负载列表中。

3.1. 前提条件

- Alauda Container Platform Web CLI 已与您的集群建立通信。
- 您具有集群管理员权限。

以下操作步骤将指导您如何安装并测试工作负载监控。

3.2. 操作步骤

1. 运行以下命令创建资源：

```
cat <<EOF | kubectl create -f -
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: ResourceFlavor
metadata:
  name: "default-flavor"
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: ClusterQueue
metadata:
  name: "cluster-queue"
spec:
  namespaceSelector: {} # match all.
  resourceGroups:
  - coveredResources: ["cpu", "memory"]
    flavors:
    - name: "default-flavor"
      resources:
      - name: "cpu"
        nominalQuota: 9
      - name: "memory"
        nominalQuota: 36Gi
---
apiVersion: kueue.x-k8s.io/v1beta2
kind: LocalQueue
metadata:
  namespace: "default"
  name: "user-queue"
spec:
  clusterQueue: "cluster-queue"
---
EOF
```

2. 使用以下作业清单创建文件：

```
cat > job.yaml << EOF
apiVersion: batch/v1
kind: Job
metadata:
  generateName: sample-job-
  namespace: default
  labels:
    kueue.x-k8s.io/queue-name: user-queue
spec:
  parallelism: 3
  completions: 3
  suspend: true
  template:
    spec:
      containers:
      - name: dummy
        image: registry.k8s.io/e2e-test-images/agnhost:2.53
        command: [ "/bin/sh" ]
        args: [ "-c", "sleep 60" ]
        resources:
          requests:
            cpu: "1"
            memory: "200Mi"
      restartPolicy: Never
EOF
```

3. 运行以下命令创建六个作业：

```
for i in {1..6}; do kubectl create -f job.yaml;done
```

3.2.1. 查看 ClusterQueue 中的待处理工作负载

要在集群级别查看所有待处理工作负载，管理员可以使用 Alauda Build of Kueue 的 visibility API 中 ClusterQueue 对象可见性端点。该端点会返回当前正在等待该 ClusterQueue 资源准入的所有 workload 列表。

操作步骤

1. 要查看 `ClusterQueue` 中的待处理工作负载，请运行以下命令：

```
kubectl get --raw "/apis/visibility.kueue.x-k8s.io/v1beta2/clusterqueues/cluster-queue/pendingworkloads"
```

您应当得到类似如下的结果：

```
{
 "kind": "PendingWorkloadsSummary",
 "apiVersion": "visibility.kueue.x-k8s.io/v1beta2",
 "metadata": {
   "creationTimestamp": null
 },
 "items": [
   {
     "metadata": {
       "name": "job-sample-job-jrjfr-8d56e",
       "namespace": "default",
       "creationTimestamp": "2023-12-05T15:42:03Z",
       "ownerReferences": [
         {
           "apiVersion": "batch/v1",
           "kind": "Job",
           "name": "sample-job-jrjfr",
           "uid": "5863cf0e-b0e7-43bf-a445-f41fa1abedfa"
         }
       ]
     },
     "priority": 0,
     "localQueueName": "user-queue",
     "positionInClusterQueue": 0,
     "positionInLocalQueue": 0
   },
   {
     "metadata": {
       "name": "job-sample-job-jg9dw-5f1a3",
       "namespace": "default",
       "creationTimestamp": "2023-12-05T15:42:03Z",
       "ownerReferences": [
         {
           "apiVersion": "batch/v1",
           "kind": "Job",
           "name": "sample-job-jg9dw",
           "uid": "fd5d1796-f61d-402f-a4c8-cbda646e2676"
         }
       ]
     },
     "priority": 0,
     "localQueueName": "user-queue",
     "positionInClusterQueue": 1,
```

```
      "positionInLocalQueue": 1
    },
    {
      "metadata": {
        "name": "job-sample-job-t9b8m-4e770",
        "namespace": "default",
        "creationTimestamp": "2023-12-05T15:42:03Z",
        "ownerReferences": [
          {
            "apiVersion": "batch/v1",
            "kind": "Job",
            "name": "sample-job-t9b8m",
            "uid": "64c26c73-6334-4d13-a1a8-38d99196baa5"
          }
        ]
      },
      "priority": 0,
      "localQueueName": "user-queue",
      "positionInClusterQueue": 2,
      "positionInLocalQueue": 2
    }
  ]
}
```


您可以传递以下可选查询参数：

`limit <integer>`：默认值为 1000。指定应获取的待处理工作负载最大数量。

`offset <integer>`：默认值为 0。指定要获取的第一个待处理工作负载的位置，从 0 开始。

2. 要仅查看 1 个待处理工作负载，并从 ClusterQueue 中位置 1 开始，请运行：

```
kubectl get --raw "/apis/visibility.kueue.x-k8s.io/v1beta2/clusterqueues/cluster-queue/pendingworkloads?limit=1&offset=1"
```

3.2.2. 查看 LocalQueue 中的待处理工作负载

要查看某个特定租户在其命名空间内提交的待处理工作负载，用户可以查询 Alauda Build of Kueue 的 visibility API 中 LocalQueue 资源可见性端点。该端点会提供其在该队列中等待的作业有序列表。

操作步骤

1. 要查看 LocalQueue 中的待处理工作负载，请运行以下命令：

```
kubectl get --raw /apis/visibility.kueue.x-k8s.io/v1beta2/namespaces/default/localqueues/user-queue/pendingworkloads
```

您应当得到类似如下的结果：

```
{
 "kind": "PendingWorkloadsSummary",
 "apiVersion": "visibility.kueue.x-k8s.io/v1beta2",
 "metadata": {
   "creationTimestamp": null
 },
 "items": [
   {
     "metadata": {
       "name": "job-sample-job-jrjfr-8d56e",
       "namespace": "default",
       "creationTimestamp": "2023-12-05T15:42:03Z",
       "ownerReferences": [
         {
           "apiVersion": "batch/v1",
           "kind": "Job",
           "name": "sample-job-jrjfr",
           "uid": "5863cf0e-b0e7-43bf-a445-f41fa1abedfa"
         }
       ]
     },
     "priority": 0,
     "localQueueName": "user-queue",
     "positionInClusterQueue": 0,
     "positionInLocalQueue": 0
   },
   {
     "metadata": {
       "name": "job-sample-job-jg9dw-5f1a3",
       "namespace": "default",
       "creationTimestamp": "2023-12-05T15:42:03Z",
       "ownerReferences": [
         {
           "apiVersion": "batch/v1",
           "kind": "Job",
           "name": "sample-job-jg9dw",
           "uid": "fd5d1796-f61d-402f-a4c8-cbda646e2676"
         }
       ]
     },
     "priority": 0,
     "localQueueName": "user-queue",
     "positionInClusterQueue": 1,
```

```
      "positionInLocalQueue": 1
    },
    {
      "metadata": {
        "name": "job-sample-job-t9b8m-4e770",
        "namespace": "default",
        "creationTimestamp": "2023-12-05T15:42:03Z",
        "ownerReferences": [
          {
            "apiVersion": "batch/v1",
            "kind": "Job",
            "name": "sample-job-t9b8m",
            "uid": "64c26c73-6334-4d13-a1a8-38d99196baa5"
          }
        ]
      },
      "priority": 0,
      "localQueueName": "user-queue",
      "positionInClusterQueue": 2,
      "positionInLocalQueue": 2
    }
  ]
}
```

监控面板卡在加载中

当 Alauda AI 部署在 ACP 4.2.0 上时，ACP 中默认安全策略的更改可能会导致 Alauda AI 创建的推理服务的监控详情页上无法显示监控数据。此问题将在未来的 ACP 版本中修复。

作为临时解决方法，请在相应的业务集群中执行以下步骤：

```
kubectl edit clusterroles warlock -n cpass-system
```

添加以下规则：

```
rules:
  - apiGroups:
      - components.aml.dev
    resources:
      - amls/finalizers
    verbs:
      - update
```

应用更改后，重启 warlock deployment：

```
kubectl rollout restart deployment warlock -n cpaas-system
```

偏差与漂移监控

目录

- 概览
- 前提条件
- 访问 TAS API
  - Service 概览与身份验证
  - 获取 token
  - Bearer token 和 base URL
- 数据摄取
  - 训练和参考数据（`POST /data/upload`）
  - 线上推理数据（`POST /consumer/kserve/v2`）
  - 列名映射（`POST /info/names`）
- 数据漂移指标
  - 漂移指标类型
  - 注册计划漂移指标
  - 一次性漂移请求
  - 列出和删除计划漂移任务
  - Prometheus 指标（漂移）
- 偏差指标
  - SPD 和 DIR
  - 注册计划偏差指标
  - 列出和删除计划偏差任务

Prometheus 指标（偏差）

概览

TrustyAI Service (TAS) 会收集模型推理数据（请求输入和响应输出），将其持久化，并组织为用于分析的数据集，包括针对参考流量和线上流量的漂移检测与偏差评估。

主要功能包括：

- 捕获并存储推理记录。
- 按 `model_name` 和 `data_tag` 组织记录。
- 提供元数据 API，用于查看已收集的内容。
- 为已记录字段配置人类可读的列名，以便简化下游分析。
- 针对参考子集和当前生产数据运行漂移检测（例如 KS test、mean shift）。
- 在受保护属性和结果上运行偏差指标（例如 SPD 和 DIR）。

本页介绍如何通过 `POST /data/upload` 摄取参考数据、通过 `POST /consumer/kserve/v2` 摄取线上推理数据，通过 HTTP API 注册漂移和偏差指标，以及如何将时间序列暴露给 Prometheus 以用于监控仪表板。

前提条件

- 在你要监控的模型所在命名空间中，存在一个处于 `Ready` 状态的 `TrustyAIService`。请参阅 部署 TrustyAI Service。

访问 **TAS API**

Service 概览与身份验证

在 TAS Deployment 中，`kube-rbac-proxy` 作为 sidecar 运行以提供身份验证。
operator 会在同一命名空间中创建两个 Service：

- `<tas-name>`：直接将流量路由到 TAS 容器（原始 Service）。
- `<tas-name>-tls`：将流量路由到 `kube-rbac-proxy` sidecar；这是需要 `Authorization: Bearer <token>` 的已认证端点。

获取 token

在与 `TrustyAIService` 相同的命名空间中创建一个 ServiceAccount、一个 Role（对 `services/proxy` 具有 `get`、`create`、`delete` 权限）以及一个 RoleBinding；然后为该 ServiceAccount 创建 token：

```
# Replace <your-namespace> and optionally the ServiceAccount name (for ex
ample, `tas-client`)
kubectl create serviceaccount -n <your-namespace> tas-client
kubectl create role -n <your-namespace> tas-client --verb=get,create,dele
te --resource=services/proxy
kubectl create rolebinding -n <your-namespace> tas-client --role=tas-clie
nt --serviceaccount=<your-namespace>:tas-client
kubectl create token -n <your-namespace> tas-client
```

也可以选择设置 token 的有效期，例如使用 `--duration=8760h` 表示一年。最后一条命令会输出 token；将其设置为 `Authorization: Bearer <token>` 请求头的值。

Bearer token 和 base URL

在访问受保护的 TAS API 时，将上一小节中的 token 作为每个请求的 **`Authorization: Bearer <token>`** 请求头。

为调用选择 **base URL**（**host**）：

- 直接 Service（无 proxy 身份验证）：`https://<tas-name>.<your-namespace>.svc.cluster.local`
- 已认证端点（`kube-rbac-proxy`）：`https://<tas-name>-tls.<your-namespace>.svc.cluster.local` —— 当 API 需要 `Authorization: Bearer <token>` 时使用此 host（通常是 `-tls` Service）。

例如，GET `/info` 请求可以写成：

```
curl -k -H "Authorization: Bearer $TOKEN" \
  "https://<tas-host>/info"
```

当需要该请求头时，请将 `<tas-host>` 替换为已认证的 host（通常是 `-tls` URL）。

数据摄取

训练和参考数据（`POST /data/upload`）

`POST /data/upload` 是用于训练和参考批次的 JSON 上传路径：每次调用都会携带一条记录的完整 request 和 response，其中 `data_tag` 可使用诸如 `TRAINING` 之类的值来表示参考子集。线上生产推理由 `POST /consumer/kserve/v2` 单独摄取（见下文）。

TAS 可以存储由模型运行产生的推理记录。一条记录包含：

- `request`：发送给模型的输入特征
- `response`：模型返回的输出

数据集由 `model_name` 和 `data_tag` 定义。

准备一个数据集子集

1. 选择一个 `model_name` 用于对上传记录进行分组。
2. 为参考子集选择一个 `data_tag`。对于训练/参考数据，使用 `data_tag: TRAINING`。
3. 对于每次推理，上传一条合并后的 `request` + `response` 载荷。

请求体字段

请求体必须包含：

- `model_name`：用于对已收集记录分组的模型标识符
- `data_tag`：数据集子集标签
- `is_ground_truth`：上传的输出是否为 ground truth（当载荷记录的是模型输出而不是已验证标签时，设为 `false`）

- `request`：
  - `id`：该记录的 inference id
  - `inputs`：输入 tensor 列表
- `response`：
  - `model_name`：必须与 `model_name` 一致
  - `id`：必须与 `request.id` 一致
  - `outputs`：输出 tensor 列表

单条记录的语义： TAS 不会训练模型；它仅存储用于监控的行数据。对于 `data_tag: TRAINING`，每次上传都代表一个 参考 样本。`request` 携带会发送给模型的输入特征（相同的 `id` 用于关联这对数据）。`response` 携带该推理观测到的输出——通常是模型预测 tensor，以及用于公平性分析的任何结果字段（例如批准分数和决策）。漂移与公平性指标会将这个参考子集与通过 `/consumer/kserve/v2` 收集到的实时数据进行比较。

下面的示例使用一个小型信用评分风格的特征集：数值输入和一个分组字段（`gender`），以及名为 `predict-0` 和 `approved` 的输出。

```
{
  "model_name": "demo-model",
  "data_tag": "TRAINING",
  "is_ground_truth": false,
  "request": {
    "id": "training-1",
    "inputs": [
      { "name": "credit_inputs-0", "shape": [1, 1], "datatype": "FP32",
"data": [21.0] },
      { "name": "credit_inputs-1", "shape": [1, 1], "datatype": "FP32",
"data": [605.0] },
      { "name": "credit_inputs-2", "shape": [1, 1], "datatype": "FP32",
"data": [12.0] },
      { "name": "credit_inputs-3", "shape": [1, 1], "datatype": "FP32",
"data": [5.0] },
      { "name": "gender", "shape": [1, 1], "datatype": "INT64", "data":
[0] }
    ]
  },
  "response": {
    "model_name": "demo-model",
    "id": "training-1",
    "outputs": [
      { "name": "predict-0", "shape": [1, 1], "datatype": "FP32", "data":
[0.301] },
      { "name": "approved", "shape": [1, 1], "datatype": "INT64", "data":
[0] }
    ]
  }
}
```

使用 **curl** 上传

```
curl -k -X POST "https://<tas-host>/data/upload" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d @<training-data.json>
```

线上推理数据（`POST /consumer/kserve/v2`）

推理时的数据通过 KServe v2 consumer 端点发送到 TAS。每次逻辑推理都使用相同 correlation `id` 的 两次 `POST` 调用：先发送模型输入（`kind: "request"`），再发送模型输出（`kind: "response"`）。请求体为 JSON；tensor 载荷是 **Base64** 编码 的 `ModelInferRequest` / `ModelInferResponse` protobuf 消息，如 KServe prediction API v2 (`grpc_predict_v2.proto`)↗ 中所定义，而不是 `/data/upload` 使用的扁平 tensor JSON。

JSON 字段：

字段	描述
`id`	关联 request/response 对（例如一个 prediction id）。
`kind`	`"request"` 或 `"response"`。
`modelid`	用于对已存储行分组的模型标识符；指标请求使用相同的 `modelId`。protobuf blob 可能省略 `model_name`；TrustyAI 会使用这个 JSON `modelid` 进行持久化。
`data`	该 `kind` 对应的 protobuf 消息的 Base64 编码。

示例序列（Base64 blob 为占位符）：

```
# Request half of one inference
curl -k -X POST "https://<tas-host>/consumer/kserve/v2" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"id":"<prediction-id>","kind":"request","modelid":"<modelId>","dat
a":"<base64-ModelInferRequest>"}'

# Response half (same id)
curl -k -X POST "https://<tas-host>/consumer/kserve/v2" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"id":"<prediction-id>","kind":"response","modelid":"<modelId>","da
ta":"<base64-ModelInferResponse>"}'
```

将 `referenceTag` 设为 `TRAINING` 的漂移指标会将该参考子集（来自 `/data/upload`）与通过此 consumer 路径收集到的自然流量行进行比较。`GET /info/tags` 可以列出诸如 `TRAINING` 之类的标签，以及线上流量对应的未标记/自然流量一侧。

列名映射（`POST /info/names`）

通过 `/data/upload` 摄取训练/参考数据并通过 `/consumer/kserve/v2` 摄取线上推理数据后，TAS 可以报告已记录的内容，并使用 `POST /info/names` 将记录的输入/输出列名映射为人类可读名称。

示例：

```
curl -k -X POST "https://<tas-host>/info/names" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
  "modelId": "demo-model",
  "inputMapping": {
    "credit_inputs-0": "Age",
    "credit_inputs-1": "Credit Score",
    "credit_inputs-2": "Education",
    "credit_inputs-3": "Employment",
    "gender": "Gender"
  },
  "outputMapping": {
    "predict-0-0": "Acceptance Probability",
    "approved-0": "Approved"
  }
}'
```

数据漂移指标

数据漂移指标会将一个 参考 子集（例如 `POST /data/upload` 中标记为 `TRAINING` 的行）与当前 生产数据（通常通过 `POST /consumer/kserve/v2` 摄取）进行比较。注册、列出和删除遵循与其他计划指标相同的请求/响应模式。

漂移指标类型

指标	角色
KSTest	对参考数据与当前数据在所选列上的经验分布进行 Kolmogorov–Smirnov 风格比较。
MeanShift	比较参考数据与当前数据在所选列上的均值（及相关统计量）。
ApproxKSTest	带有可调精度参数（`epsilon`、`thresholdDelta`）的近似 KS 风格漂移检测。
FourierMMD	使用随机 Fourier 特征（`gamma`、`parameters`）通过 maximum mean discrepancy 进行漂移检测。

`GET /metrics/drift/<name>/definition` 会为每个指标返回人类可读的文档说明。

注册计划漂移指标

使用带有 JSON 请求体的 `POST /metrics/drift/<metricName>/request`。常见字段包括：

- `modelId`：数据集 id（必须与上传内容 / consumer 中的 `modelid` 一致）。
- `requestName`：此计划任务的唯一名称。
- `metricName`：必须与路径段一致（`kstest`、`meanshift`、`approxkstest` 或 `fouriermmd`）。
- `batchSize`：每次计算运行中包含的推理行数。
- `referenceTag`：参考子集的标签（通常为 `TRAINING`）。
- `fitColumns`：要评估的输入列名（已记录字段名，例如在 `POST /info/names` 映射之前的 tensor 名称）。

KSTest — 示例：

```
curl -k -X POST "https://<tas-host>/metrics/drift/kstest/request" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
  "modelId": "<modelId>",
  "requestName": "<requestName>",
  "metricName": "kstest",
  "batchSize": 20,
  "referenceTag": "TRAINING",
  "fitColumns": ["credit_inputs-0", "credit_inputs-1"]
}'
```

MeanShift — 示例：

```
curl -k -X POST "https://<tas-host>/metrics/drift/meanshift/request" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
  "modelId": "<modelId>",
  "requestName": "<requestName>",
  "metricName": "meanshift",
  "batchSize": 20,
  "referenceTag": "TRAINING",
  "fitColumns": ["credit_inputs-0", "credit_inputs-1", "credit_inputs-2",
"credit_inputs-3"]
}'
```

ApproxKSTest — 额外包含 `thresholdDelta` 和 `epsilon`（语义请参见 `GET /metrics/drift/approxkstest/definition`）：

```
curl -k -X POST "https://<tas-host>/metrics/drift/approxkstest/request" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
  "modelId": "<modelId>",
  "requestName": "<requestName>",
  "metricName": "approxkstest",
  "batchSize": 20,
  "thresholdDelta": 0.05,
  "referenceTag": "TRAINING",
  "fitColumns": ["credit_inputs-0", "credit_inputs-1"],
  "epsilon": 0.01
}'
```

FourierMMD — 额外包含 `thresholdDelta`、`gamma` 和 `parameters` 对象（请参见 `GET /metrics/drift/fouriermmd/definition`）：

```
curl -k -X POST "https://<tas-host>/metrics/drift/fouriermmd/request" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
  "modelId": "<modelId>",
  "requestName": "<requestName>",
  "metricName": "fouriermmd",
  "batchSize": 20,
  "thresholdDelta": 0.05,
  "referenceTag": "TRAINING",
  "fitColumns": ["credit_inputs-0", "credit_inputs-1"],
  "gamma": 1.0,
  "parameters": { "nWindow": 10, "nTest": 10, "nMode": 50, "randomSeed":
0, "sig": 1.0, "deltaStat": false, "epsilon": 0.01 }
}'
```

一次性漂移请求

对于一次按需运行（非计划任务），`POST /metrics/drift/kstest` 和 `POST /metrics/drift/meanshift` 接受与计划注册相同的 JSON 请求体结构，但不带 `/request` 后缀路径。

列出和删除计划漂移任务

列出计划任务：

```
curl -k -H "Authorization: Bearer $TOKEN" \
  "https://<tas-host>/metrics/drift/kstest/requests"
```

根据列表响应中的 `requestId` 停止任务：

```
curl -k -X DELETE "https://<tas-host>/metrics/drift/kstest/request" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "requestId": "<requestId-uuid>"
}'
```

对于其他漂移类型，将路径中的 `kstest` 替换为 `meanshift`、`approxkstest` 或 `fouriermmd`。`GET /metrics/all/requests` 会列出跨类别的计划指标。

Prometheus 指标（漂移）

计划漂移结果会作为 Micrometer gauges 发布到 `GET /q/metrics`，例如：

- KSTest: `trustyai_kstest`
- MeanShift: `trustyai_meanshift`
- ApproxKSTest: `trustyai_approxkstest`
- FourierMMD: `trustyai_fouriermmd`

每条漂移序列都带有用于标识计划任务及其测量对象的标签。`trustyai_kstest` / `trustyai_meanshift` / `trustyai_approxkstest` / `trustyai_fouriermmd` 上常见的 语义标签包括：

标签	角色
`model`	模型数据集 id（与上传 / 指标请求中的 `modelId` 一致）。

标签	角色
`requestName`	计划漂移请求的名称（来自 `POST .../kstest/request` 等）。
`metricName`	该序列暴露的指标类型（例如 `KSTEST`、`MEANSHIFT`；具体大小写取决于 TAS 构建版本）。
`batch_size`	为计划请求配置的 batch size。
`subcategory`	样本所对应的特征或列（例如某个映射后的名称，如 `Acceptance Probability`，取决于指标和 `fitColumns`）。
`request`	指标请求实例的内部 id（UUID），如果存在。
`endpoint`	来自 Micrometer 的传输或抓取路径提示（例如 `http`）。

抓取 **/ target** 标签（名称取决于集群和 `ServiceMonitor` 配置）通常包括 `namespace`、`pod`、`service`、`job` 和 `instance`。部署的实际值应从 Prometheus 中读取（例如通过指标 UI 中 `trustyai_kstest` 的标签名称或使用 `label_values()`），而不是直接复制文档——环境相关的 id 和地址会因集群而异。

PromQL 示例（缩小到一个模型和一个 requestName）：

```
trustyai_kstest{model="<modelId>", requestName="<requestName>"}
```

如果要过滤单个列或特征，可以在该标签存在时为 `subcategory` 添加匹配器。

偏差指标

本节介绍如何通过 TAS HTTP API（对 `/metrics/group/fairness/...` 执行 `POST` / `GET` / `DELETE`）进行偏差监控（组公平性指标，如 SPD 和 DIR）。

SPD 和 DIR

TAS 在相同的受保护属性和结果上暴露两个相关的 组公平性 指标：

缩写	全称	含义（典型用法）
SPD	Statistical Parity Difference	未特权组的有利结果率与特权组的有利结果率之差。数值越接近 **0**，表示组间越接近平衡。
DIR	Disparate Impact Ratio	未特权组有利结果率与特权组有利结果率的比值。数值越接近 **1**，表示越接近平衡（传统“四分之五”风格的检查通常会将该比值与阈值进行比较）。

两者使用相同的请求字段（`protectedAttribute`、`outcomeName`、`favorableOutcome` 等）。HTTP 路径和 `metricName` 用于区分 **SPD**（`spd`）与 **DIR**（`dir`）。

注册计划偏差指标

通过以下调用，为已部署的模型数据集创建一个周期性的偏差指标请求：

- `POST /metrics/group/fairness/spd/request`（Statistical Parity Difference, SPD）
- `POST /metrics/group/fairness/dir/request`（Disparate Impact Ratio, DIR）

示例（SPD）：

```
curl -k -X POST "https://<tas-host>/metrics/group/fairness/spd/request" \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "modelId": "<modelId>",
    "requestName": "<requestName>",
    "metricName": "spd",
    "batchSize": 20,
    "protectedAttribute": "<protectedAttribute>",
    "privilegedAttribute": <privilegedValue>,
    "unprivilegedAttribute": <unprivilegedValue>,
    "outcomeName": "<outcomeName>",
    "favorableOutcome": <favorableOutcome>
  }'
```

SPD 请求字段：

- `modelId`：要计算该指标的模型数据集 id（必须与上传中使用的数据集/模型一致）。

- `requestName`：该计划指标请求的唯一名称（用于区分周期性任务）。
- `metricName`：指标类型名称；对于此端点使用 `spd`。
- `batchSize`：TAS 在计算计划指标时包含的推理记录数。
- `protectedAttribute`：定义被比较组的特征（如果使用了名称映射，则使用映射后的名称）。
- `privilegedAttribute`：`protectedAttribute` 的值，代表特权组。
- `unprivilegedAttribute`：`protectedAttribute` 的值，代表未特权组。
- `outcomeName`：用于公平性评估的输出字段（例如分类结果）。
- `favorableOutcome`：被视为有利结果的 `outcomeName` 值。

对于 **DIR**，使用相同的 JSON 请求体，将 `"metricName"` 设为 `"dir"`，并调用 `POST /metrics/group/fairness/dir/request`。`GET /metrics/group/fairness/dir/definition` 会以文本形式描述该指标。

列出和删除计划偏差任务

要停止周期性计算，先列出任务，然后将响应中的 `requestId` 发送到对应的删除端点：

指标	列表	删除
SPD	`GET /metrics/group/fairness/spd/requests`	`DELETE /metrics/group/fairness/spd/re`
DIR	`GET /metrics/group/fairness/dir/requests`	`DELETE /metrics/group/fairness/dir/re`

示例（SPD）：

```
curl -k -H "Authorization: Bearer $TOKEN" \
  "https://<tas-host>/metrics/group/fairness/spd/requests"
```

```
curl -k -X DELETE "https://<tas-host>/metrics/group/fairness/spd/request"
\
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{
    "requestId": "<requestId-uuid>"
  }'
```

对于 DIR 任务，使用相同的 `requestId` JSON 请求体和 `.../dir/request` 删除 URL。

Prometheus 指标（偏差）

TAS 会在 `/q/metrics` 上暴露 Prometheus 指标。TrustyAI Operator 会创建一个 `ServiceMonitor`，以持续采集与偏差相关的指标（默认匹配 `trustyai_(spd|dir).*` 的序列）。

基础指标名称：

- SPD（Statistical Parity Difference）：`trustyai_spd`
- DIR（Disparate Impact Ratio）：`trustyai_dir`

每条序列都带有用于标识计划计算及其公平性配置的标签。`trustyai_spd` / `trustyai_dir` 上常见的 语义 标签包括：

标签	角色
`model`	模型数据集 id（与上传 / 指标请求中的 `modelId` 一致）。
`requestName`	计划指标请求的名称（来自 `POST .../spd/request` 或 `POST .../dir/request`）。
`metricName`	该序列暴露的指标类型（例如 `SPD` 或 `DIR`）。
`protected`	受保护属性列（例如 `Gender`）。
`outcome`	结果列（例如 `Approved`）。
`favorable_value`	被视为有利结果的值。

标签	角色
`privileged` / `unprivileged`	`protected` 上的特权组和未特权组取值。
`batch_size`	为计划请求配置的 batch size。
`request`	指标请求实例的内部 id（UUID），如果存在。

抓取 **/ target** 标签（名称取决于集群和 `ServiceMonitor` 配置）通常包括 `namespace`、`pod`、`service`、`job` 和 `instance`。

示例：选择一个模型和一个计划请求名称的 SPD：

```
trustyai_spd{model="demo-credit-model-1774142686-91294", requestName="demo-spd-1774142686-91294"}
```

示例：某个时间窗口内的最新值（将范围与抓取间隔对齐）：

```
max_over_time(trustyai_spd{model="<modelId>", requestName="<requestName>"}[15m])
```

当注册了 DIR 计划任务时，相同的标签过滤条件同样适用于 `trustyai_dir`。