容器服务

TKE AI 应用实践

腾讯云

文档目录

TKE AI 应用实践
MCP Server 应用
MCP Server 托管

最近更新时间：2025-10-13 17:17:22

本文档介绍如何在 TKE 上部署基于 Streamable HTTP 传输的 MCP Server。

简介

MCP（Model Context Protocol，模型上下文协议）是一种开放协议，用于规范 AI Agent 与外部环境和工具的交互方式。在该体系中，MCP Client 由 AI Agent 托管，负责发起交互；MCP Server 则承载各种工具和资源，供 MCP Client 调用。
通过 MCP，AI Agent 可以以统一的方式访问外部工具、数据库和服务，从而降低适配成本并提升扩展能力。
MCP 支持 两种主要的通信方式：

- stdio 模式：通常用于本地运行，Client 与 Server 在同一台机器上通过进程间通信交互，适合开发、调试或本地集成。
- Streamable HTTP 模式：基于 HTTP 协议实现跨网络通信，天然支持云端部署，使远程 MCP Client 也能安全访问。

如果您已有自研或开源的 MCP Server，希望在云端托管运行，TKE 支持直接托管 Streamable HTTP 类型的 MCP Server。

环境准备

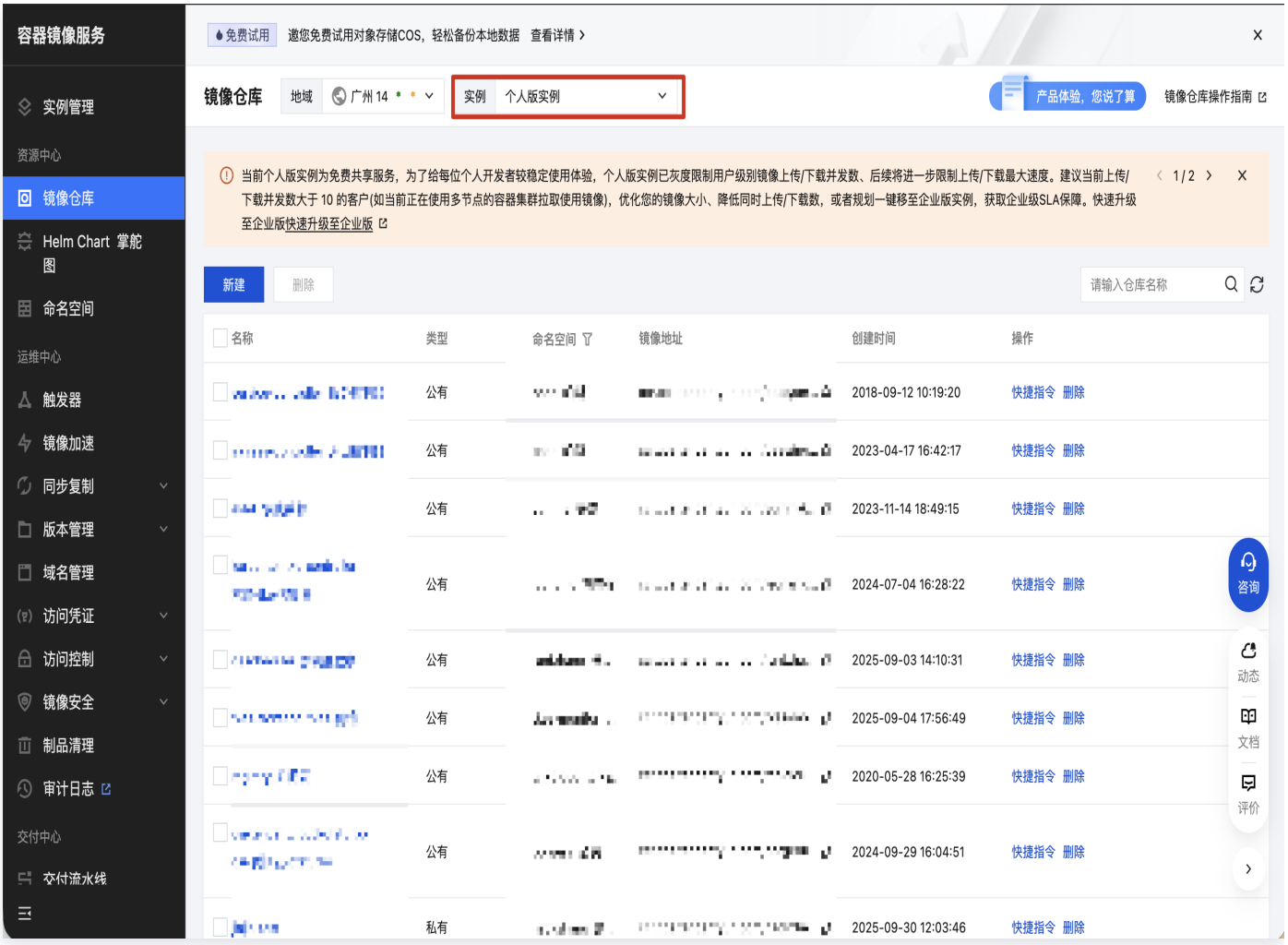
第一步：准备集群

1. 进入腾讯云 容器服务控制台 。
2. 选择您所需要部署 MCP Server 的 TKE 集群，如果您还没有集群，请参见 创建集群 。

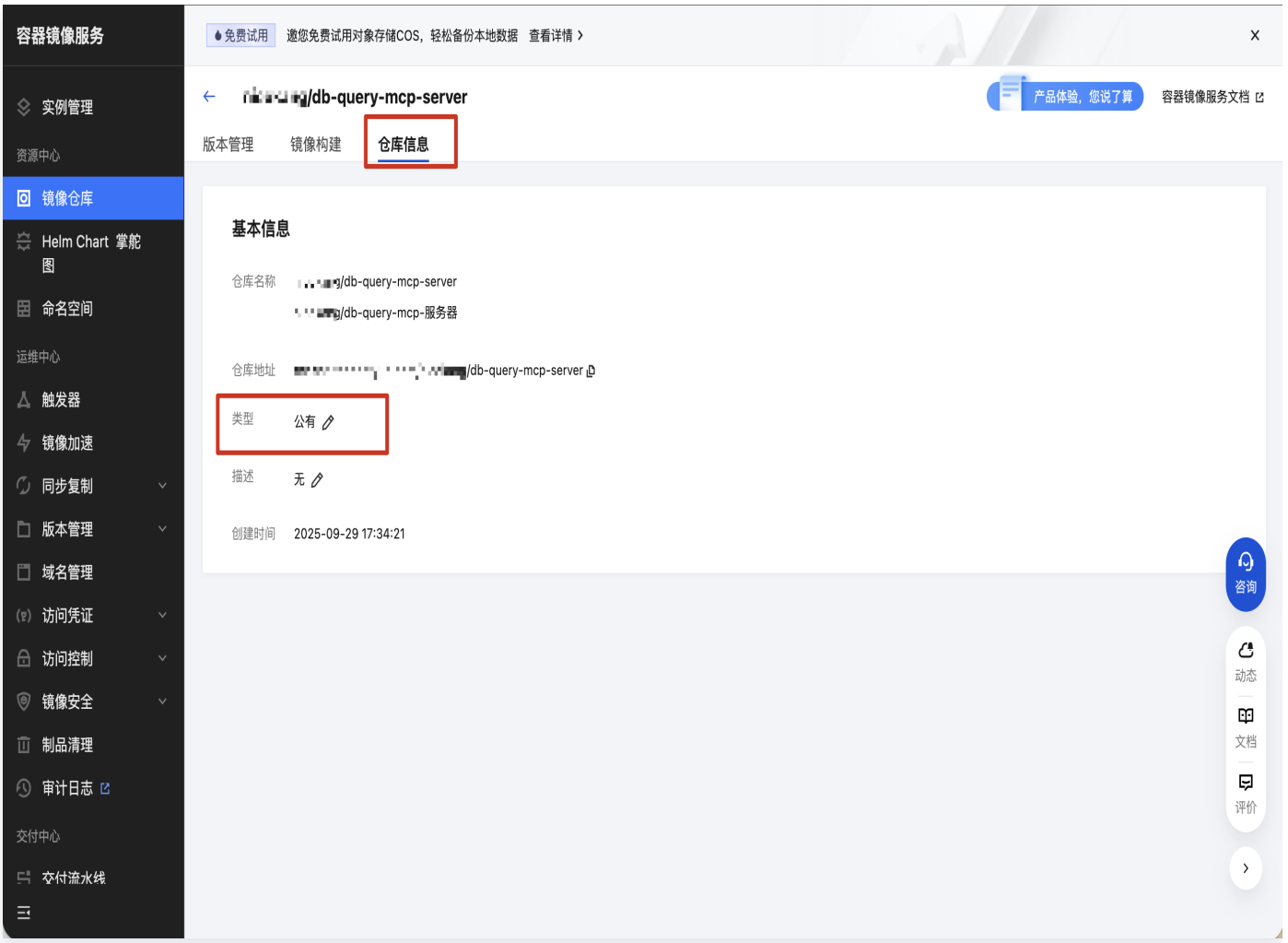
第二步：镜像打包与上传

将您的 MCP Server 代码与依赖打包成标准镜像，并上传至可被集群拉取的仓库，确保后续部署顺利。

1. 进入腾讯云 容器镜像服务控制台 。
2. 在镜像仓库页面，上传镜像到 CCR（腾讯云容器镜像服务个人版，免费但无 SLA 保证）。操作详情请参见 容器镜像服务个人版快速开始 。



3. 在仓库信息中设置您的镜像为公有镜像，方便部署工作负载时拉取镜像。



托管 Streamable HTTP MCP Server

第一步：创建工作负载并配置访问设置

目的：将 MCP Server 部署到 TKE 集群中，并通过 Service 对外暴露，方便外部 MCP Client 访问。

1. 登录腾讯云 容器服务控制台，进入需要部署 MCP Server 的 TKE 集群。
2. 选择左侧导航中的**工作负载**，在 **Deployment** 页签，单击**新建**。
3. 配置 MCP Server 工作负载的基本信息，并选择对应的 MCP Server 镜像。

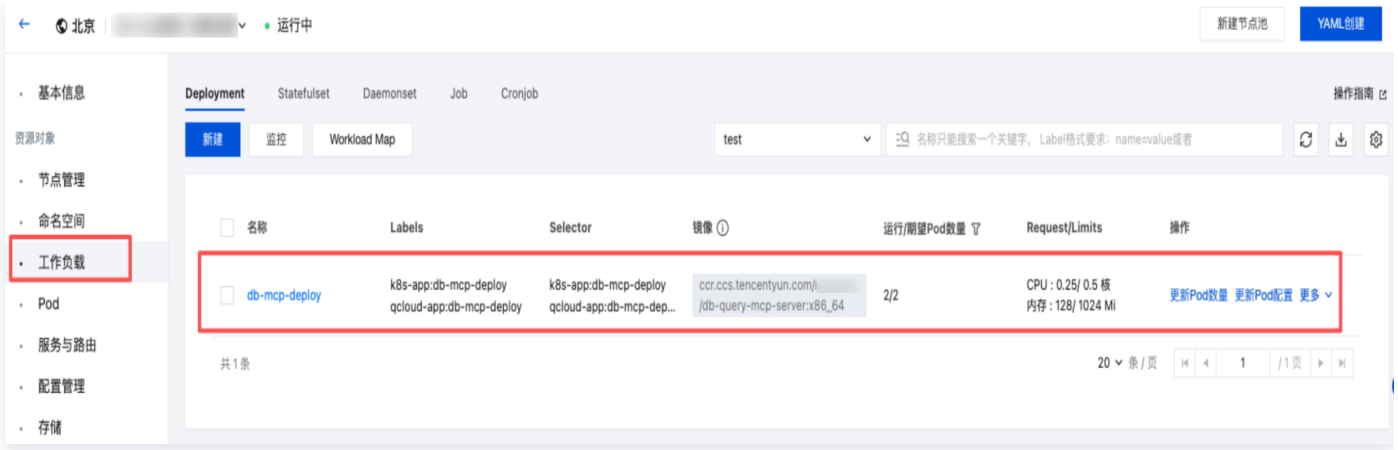
4. 配置访问设置（Service）：

通过 Service 将 MCP Server 暴露到外部网络。本文以将 MCP Server 暴露到公网为例，方便外部 Client 直接访问。

第二步：查看工作负载情况

目的：确认 MCP Server 是否已经成功启动。

1. 进入工作负载页面，查看新建的 Deployment。



2. 确认 Pod 状态为 Running，表示 MCP Server 已正常运行。

第三步：验证 Service 情况

目的：确认 MCP Server 已正确对外暴露，并可被公网访问。

1. 在服务与路由模块中，查找新建的 Service，获取公网访问入口（公网 LB 地址）。
2. 在本地机器上，使用以下命令测试连通性：

```
# 使用 ping 检查公网 LB 地址是否可达
ping 43.xxx.xxx.xx

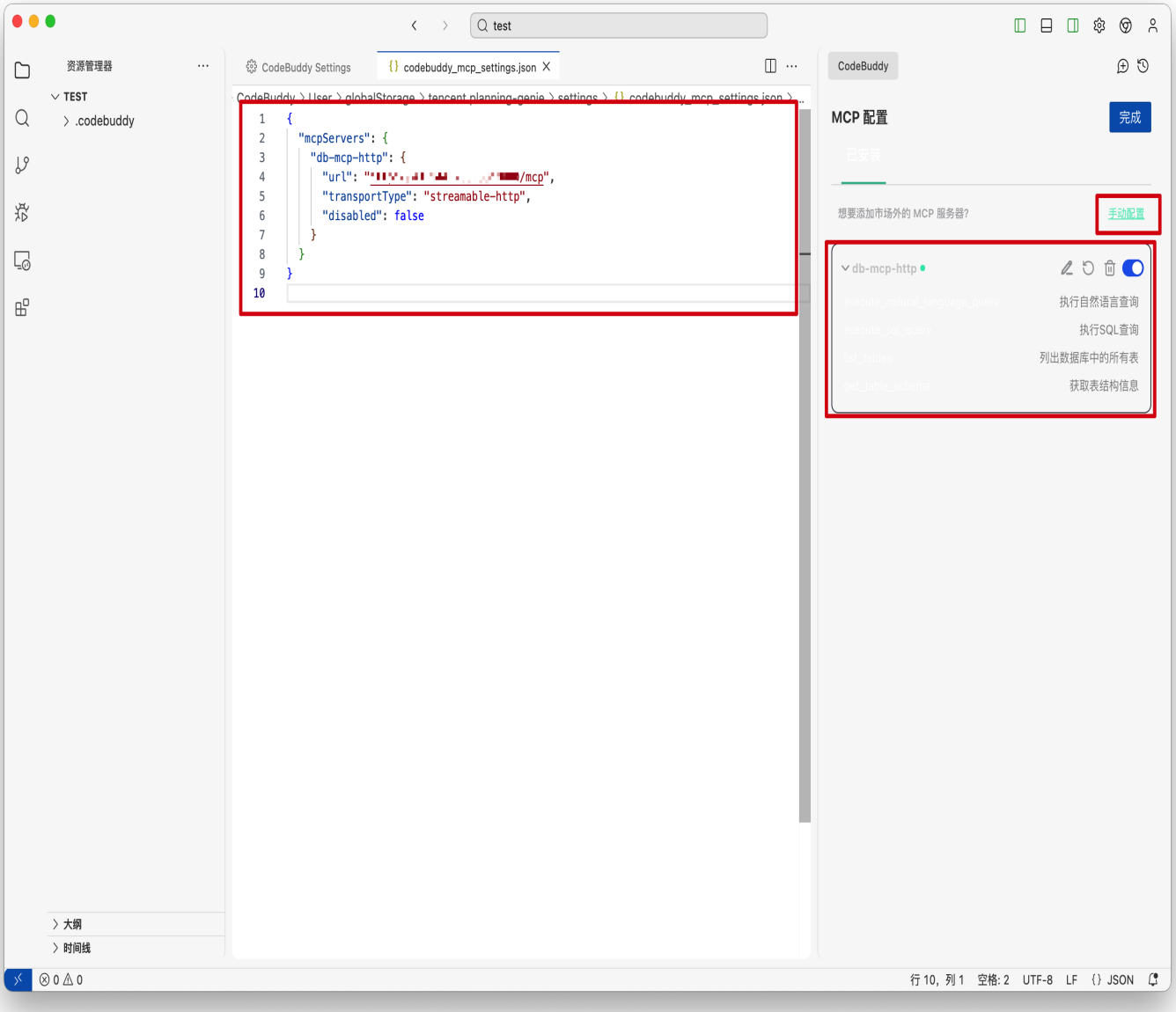
# 使用 telnet 测试 MCP Server 端口连通性（假设端口为 8000）
telnet 43.xxx.xxx.xx 8000
```

3. 若命令执行成功，即表明 MCP Server 已成功对外提供访问能力。

验证 MCP Server

第一步：在 Codebuddy 配置 MCP Server

目的： 将在 TKE 中部署的 MCP Server 与 Codebuddy 对接，使其可以作为外部工具被调用。



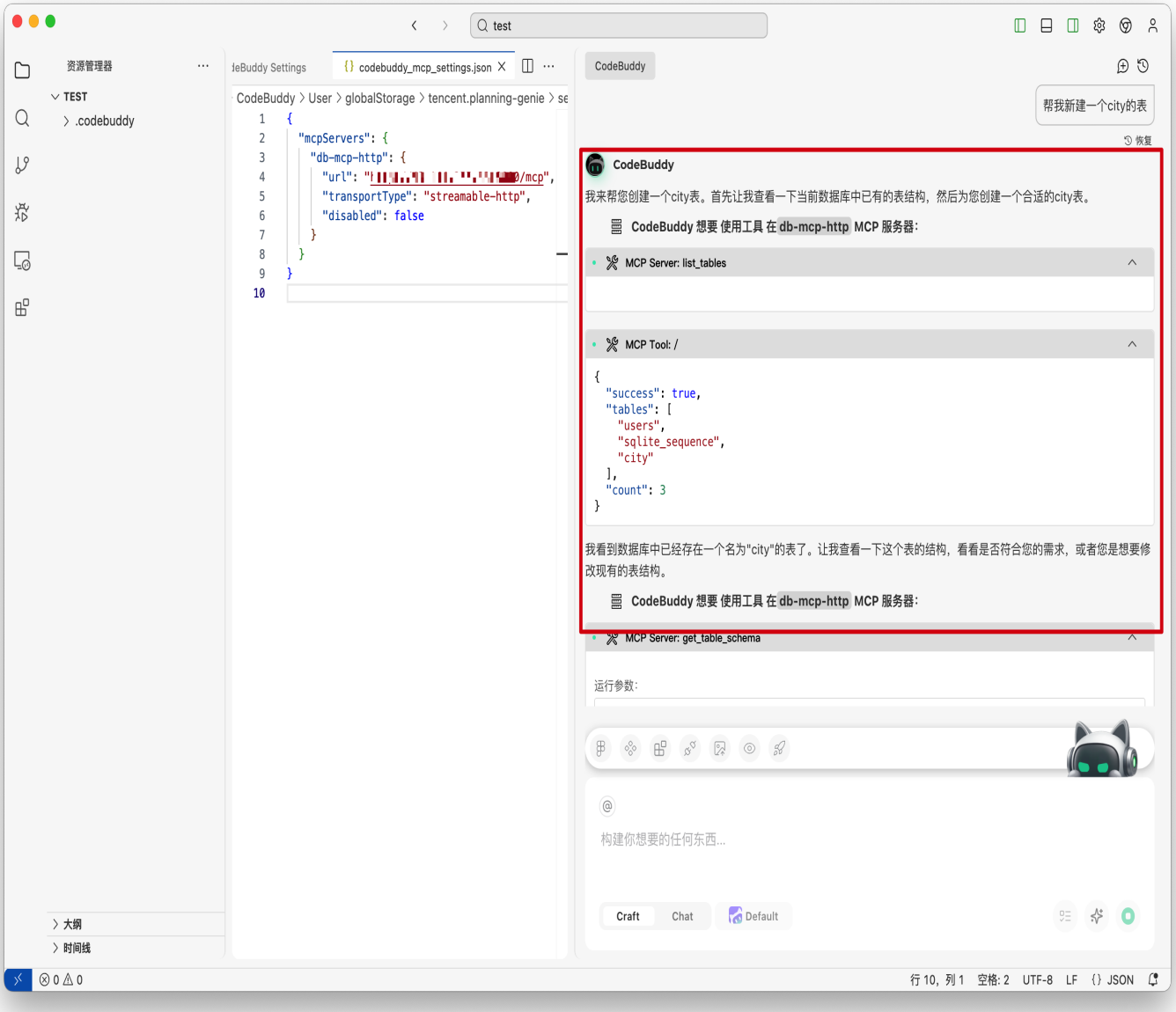
配置示例：

```
{
  "mcpServers": {
    "db-mcp-http": {
      // 将此处的地址和端口替换为 Service 中的公网访问入口和端口
      "url": "http://xx.xx.xx.xx:xxxx/mcp",
      "transportType": "streamable-http",
      "disabled": false
    }
  }
}
```

第二步：验证配置是否生效

目的： 确认 MCP Server 已成功接入并可被访问。

1. 在 Codebuddy 中保存配置并重新加载。
2. 发起一次请求，调用 MCP Server 提供的能力。
3. 如果返回正常结果，即表示 MCP Server 已验证成功。



常见问题

CCR 创建后无法访问

问题现象：在创建 CCR（容器仓库）后，可能出现私有镜像无法在公有环境中直接拉取的情况。

处理建议：在仓库信息中设置您的镜像为公有镜像，方便部署工作负载时拉取镜像。

Pod 处于 Pending 状态

问题现象：部署工作负载后，Pod 长时间停留在 Pending 状态。
处理建议：检查 Deployment 中的 resource request/limit 配置，适当调整 CPU 核数和内存大小，避免因资源请求过高或集群资源不足而无法调度。

Agent 智能体应用
Agent 代码沙箱工具部署

最近更新时间：2025-10-10 10:36:21

本文档介绍在 TKE 上使用 Langchain 构建调用代码沙箱的 Agent。

简介

Agent 代码沙箱是专为 AI Agent 场景设计的新一代运行时基础设施，提供安全隔离的云端执行环境，可防止 Agent 访问或篡改系统外资源。同时，支持多种编程语言，具备快速启动、高并发等特性，能满足多样化的 AI 任务需求。
llm-sandbox 是一个轻量级且便携的沙盒环境，专门设计用于在安全隔离的环境中运行大语言模型（LLM）生成的代码，该开源项目通过 Docker 容器提供了一个易于使用的接口，用于设置、管理和执行代码，简化了运行 LLM 生成代码的过程。
Langchain 是一个用于开发由大型语言模型（LLM）驱动的应用程序的框架，能简化 LLM 应用生命周期各阶段，可借助其开源组件、第三方集成构建有状态智能体。

环境准备

- 已创建并部署好 TKE 集群，如果您还没有集群，请参见 创建集群 。
- 已创建节点池，并且节点池内有至少2个节点，机型推荐 SA5.LARGE8。

环境验证

验证 Python 版本，推荐版本3.11.6。

```
python --version
```

预期结果：

```
Python 3.11.6
```

验证 pip 版本，推荐版本23.3.1，预期结果：

```
pip 23.3.1 from /usr/lib/python3.11/site-packages/pip (python 3.11)
```

如果上述依赖不存在，可以运行以下命令安装依赖。

```
yum install python3 python3-pip -y #安装python3和pip
```

安装 llm-sandbox、langchain 和 langchain_openai。

```
pip install 'llm-sandbox[k8s]' langchain langchain_openai #安装依赖
```

构建一个工具调用的 Agent

我们将在集群中使用 Langchain 框架构建一个可以调用代码沙箱工具的 Agent，并且可以在沙箱中编写代码并返回结果。

步骤1：定义沙箱工具

代码沙箱提供执行代码的环境，并使用 K8s 作为后端，pod 作为执行代码的隔离环境。

```
@tool
def run_code(lang: str, code: str, libraries: list | None = None) ->
str:
    """Run code in a sandboxed environment.

    :param lang: The language of the code, must be one of ['python',
'java', 'javascript', 'cpp', 'go', 'ruby'].
    :param code: The code to run.
    :param libraries: The libraries to use, it is optional.
    :return: The output of the code.
    """
    with
SandboxSession(lang=lang,backend=SandboxBackend.KUBERNETES,kube_namespac
e="default", verbose=False) as session:
        return session.run(code, libraries).stdout
```

步骤2：接入模型

场景接入使用 OpenAI 兼容接口的大模型，可以从 Deepseek 等模型服务提供商处获取 API_KEY，或在 TKE 上部署自建大模型。

```
llm = ChatOpenAI(
    model="xxx", # 填入模型名称，例如：deepseek-chat
    temperature=0,
    max_retries=2,
    api_key="xxxx", # 添入账户密钥，例如：sk-4**********************7
```

```
    base_url="xxxxx",# 添入API端点地址，例如：https://api.deepseek.com
)
```

步骤3：构建工具调用 Agent

使用 Langchain 构建工具调用 Agent，选择代码沙箱工具为可用的工具。

```
prompt = hub.pull("hwchase17/openai-functions-agent")
tools = [run_code]
agent = create_tool_calling_agent(llm, tools, prompt)  # type:
ignore[arg-type]
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
# type: ignore[arg-type]
```

验证代码沙箱

设计如下四个执行代码的测试用例：

```
output = agent_executor.invoke({
        "input": "Write python code to calculate Pi number by Monte
Carlo method then run it."
    })
    logger.info("Agent: %s", output)

    output = agent_executor.invoke({"input": "Write python code to
calculate the factorial of a number then run it."})
    logger.info("Agent: %s", output)

    output = agent_executor.invoke({"input": "Write python code to
calculate the Fibonacci sequence then run it."})
    logger.info("Agent: %s", output)

    output = agent_executor.invoke({"input": "Calculate the sum of the
first 10000 numbers."})
    logger.info("Agent: %s", output)
```

预期结果：

完成上述步骤执行，可以在集群中运行代码，查看运行效果，示例的效果如下：

NAME	READY	STATUS	RESTARTS	AGE
[illegible]	1/1	Running	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
sandbox-python-5f8938de	1/1	Running	0	14h
sandbox-python-bac93d13	1/1	Running	0	14h
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]

```
> Entering new AgentExecutor chain...
HTTP Request: POST http://xxxx/v1/chat/completions "HTTP/1.1 200 OK"

Invoking: `run_code` with `{'lang': 'python', 'code': 'import
random\n\ndef estimate_pi(n):\n    num_point_circle = 0\n
num_point_total = 0\n    \n    for _ in range(n):\n        x =
random.uniform(0,1)\n        y = random.uniform(0,1)\n        distance =
x**2 + y**2\n        if distance <= 1:\n            num_point_circle +=
1\n        num_point_total += 1\n        \n    return 4 *
num_point_circle/num_point_total\n\nprint(estimate_pi(1000000))'}`


3.142396
HTTP Request: POST http://xxxxx/v1/chat/completions "HTTP/1.1 200 OK"
The calculated value of Pi using the Monte Carlo method with 1,000,000
sample points is approximately **3.142396**.

This is a stochastic method, so the result will vary slightly each time
you run it. The accuracy generally improves with a larger number of
sample points.

> Finished chain.
Agent: {'input': 'Write python code to calculate Pi number by Monte
Carlo method then run it.', 'output': 'The calculated value of Pi using
the Monte Carlo method with 1,000,000 sample points is approximately
**3.142396**.\n\nThis is a stochastic method, so the result will vary
slightly each time you run it. The accuracy generally improves with a
larger number of sample points.'}


> Entering new AgentExecutor chain...
HTTP Request: POST http://xxxxx/v1/chat/completions "HTTP/1.1 200 OK"
```

```
Invoking: `run_code` with `{'lang': 'python', 'code': 'def
factorial(n):\n    if n == 0:\n        return 1\n    else:\n
return n * factorial(n-1)\n\nnumber = 5\nprint(f"The factorial of
{number} is {factorial(number)}")'}`


The factorial of 5 is 120
HTTP Request: POST http://xxxxx/v1/chat/completions "HTTP/1.1 200 OK"
Here is the Python code to calculate the factorial of a number:

```python
def factorial(n):
    if n < 0:
        return "Factorial is not defined for negative numbers"
    elif n == 0 or n == 1:
        return 1
    else:
        result = 1
        for i in range(2, n + 1):
            result *= i
        return result

number = 5
print(f"The factorial of {number} is {factorial(number)}")
```

And as shown in the output, the factorial of 5 is 120.

> Finished chain.
Agent: {'input': 'Write python code to calculate the factorial of a
number then run it.', 'output': 'Here is the Python code to calculate
the factorial of a number:\n\n```python\ndef factorial(n):\n   if n <
0:\n   return "Factorial is not defined for negative numbers"\n
elif n == 0 or n == 1:\n   return 1\n   else:\n   result =
1\n   for i in range(2, n + 1):\n   result *= i\n
return result\n\nnumber = 5\nprint(f"The factorial of {number} is
{factorial(number)}")\n```\n\nAnd as shown in the output, the factorial
of 5 is 120.'}

> Entering new AgentExecutor chain...
HTTP Request: POST http://xxxxx/v1/chat/completions "HTTP/1.1 200 OK"
```

```
Invoking: `run_code` with `{'lang': 'python', 'code': '\ndef
fibonacci(n):\n    if n <= 0:\n        return []\n    elif n == 1:\n
return [0]\n    elif n == 2:\n        return [0, 1]\n    else:\n
fib_seq = [0, 1]\n        for i in range(2, n):\n
fib_seq.append(fib_seq[i-1] + fib_seq[i-2])\n        return fib_seq\n\n#
Calculate the first 10 numbers in the Fibonacci sequence\nresult =
fibonacci(10)\nprint(result)\n'}`


[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
HTTP Request: POST http://xxxx/v1/chat/completions "HTTP/1.1 200 OK"

Invoking: `run_code` with `{'lang': 'python', 'code': 'def
fibonacci(n):\n    fib_sequence = [0, 1]\n    while len(fib_sequence) <
n:\n        fib_sequence.append(fib_sequence[-1] + fib_sequence[-2])\n
return fib_sequence[:n]\n\nprint(fibonacci(10))'}`
responded: The previous attempt to run the code did not produce any
output. I will now write the Python code to calculate the Fibonacci
sequence and execute it properly.



[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
HTTP Request: POST http://xxxx/v1/chat/completions "HTTP/1.1 200 OK"
Here is the Python code to calculate the Fibonacci sequence up to the
10th term:

```python
def fibonacci(n):
    fib_sequence = [0, 1]
    while len(fib_sequence) < n:
        next_value = fib_sequence[-1] + fib_sequence[-2]
        fib_sequence.append(next_value)
    return fib_sequence[:n]

# Calculate the first 10 Fibonacci numbers
result = fibonacci(10)
print(result)
```

When executed, this code produces the following output:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
> Finished chain.
Agent: {'input': 'Write python code to calculate the Fibonacci sequence
then run it.', 'output': 'Here is the Python code to calculate the
Fibonacci sequence up to the 10th term:\n\n```python\ndef
fibonacci(n):\n    fib_sequence = [0, 1]\n    while len(fib_sequence) <
n:\n        next_value = fib_sequence[-1] + fib_sequence[-2]\n
fib_sequence.append(next_value)\n    return fib_sequence[:n]\n\n#
Calculate the first 10 Fibonacci numbers\nresult =
fibonacci(10)\nprint(result)\n```\n\nWhen executed, this code produces
the following output:\n[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]'}


> Entering new AgentExecutor chain...
HTTP Request: POST http://xxxx/v1/chat/completions "HTTP/1.1 200 OK"

Invoking: `run_code` with `{'lang': 'python', 'code':
'print(sum(range(1, 10001)))'}`


50005000
HTTP Request: POST http://xxxxx/v1/chat/completions "HTTP/1.1 200 OK"
The sum of the first 10,000 numbers is 50,005,000.

> Finished chain.
Agent: {'input': 'Calculate the sum of the first 10000 numbers.',
'output': 'The sum of the first 10,000 numbers is 50,005,000.'}
```

可参考的完整代码如下：

```
import logging

from langchain import hub
from langchain.agents import AgentExecutor, create_tool_calling_agent
from langchain_core.tools import tool
from langchain_openai import ChatOpenAI

from llm_sandbox import SandboxSession, SandboxBackend

logging.basicConfig(level=logging.INFO, format="%(message)s")
logger = logging.getLogger(__name__)


@tool
```

```
def run_code(lang: str, code: str, libraries: list | None = None) ->
str:
    """Run code in a sandboxed environment.

    :param lang: The language of the code, must be one of ['python',
'java', 'javascript', 'cpp', 'go', 'ruby'].
    :param code: The code to run.
    :param libraries: The libraries to use, it is optional.
    :return: The output of the code.
    """
    with
SandboxSession(lang=lang,backend=SandboxBackend.KUBERNETES,kube_namespac
e="default", verbose=False) as session:
        return session.run(code, libraries).stdout


if __name__ == "__main__":
    # llm = ChatOpenAI(model="gpt-4.1-nano", temperature=0)
    llm = ChatOpenAI(
    model="qwen3-coder",
    temperature=0,
    max_retries=2,
    api_key="1d4a46b238eabef25101b5ff1dc36a150",
    base_url="http://49.233.239.193:60000/v1",
)
    prompt = hub.pull("hwchase17/openai-functions-agent")
    tools = [run_code]

    agent = create_tool_calling_agent(llm, tools, prompt)  # type:
ignore[arg-type]
    agent_executor = AgentExecutor(agent=agent, tools=tools,
verbose=True)  # type: ignore[arg-type]
    output = agent_executor.invoke({
        "input": "Write python code to calculate Pi number by Monte
Carlo method then run it."
    })
    logger.info("Agent: %s", output)

    output = agent_executor.invoke({"input": "Write python code to
calculate the factorial of a number then run it."})
    logger.info("Agent: %s", output)
```

```
    output = agent_executor.invoke({"input": "Write python code to
calculate the Fibonacci sequence then run it."})
    logger.info("Agent: %s", output)

    output = agent_executor.invoke({"input": "Calculate the sum of the
first 10000 numbers."})
    logger.info("Agent: %s", output)
```

常见问题

pip 安装时无法找到 llm-sandbox 包?

Python 和 pip 版本过低，更新 python 和 pip 的版本，推荐 python3.11 以上。

运行测试代码时，遇到权限报错返回?

开启集群内网访问，并更新 config 文件。

AI 模型应用
Embedding 模型部署

最近更新时间：2025-10-13 14:56:12

本文档介绍如何在 TKE 上部署 Embedding 模型。

简介

Embedding 模型是将文本、图像等非结构化数据转化为低维稠密向量的 AI 模型，这些向量能精准捕捉数据的语义或特征关联，让机器可通过向量计算理解数据间的相似性，是语义检索、推荐系统、聚类分析等场景的核心基础，在文档检索中，可通过对比查询与文档的 Embedding 向量快速匹配相关内容。
tke-ai-playbook 是 TKE 团队开源的 AI 大模型相关脚本，包含模型下载、部署推理服务、性能测试等模块，可在 TKE 一站式体验 AI 相关功能，点此获取 开源链接 。

环境准备

步骤1：创建 TKE 标准集群

1. 登录 腾讯云容器服务控制台 ，选择左侧导航栏中的**集群**。
2. 单击集群列表上方的**新建**。
3. 在集群类型中，选择标准集群。
4. 在**创建集群 > 网络配置**中，选择 **VPC-CNI**，其余参数保持默认即可。

5. 在**创建集群 > 组件配置**中，存储组件勾选 **CFS**（用于持久化存储大模型权重文件；若创建时未勾选，后续可在“集群 – 组件管理” 中安装，操作详情请参见 通过组件管理页安装 ）。

6. 创建下载节点池。

   选择 3 个节点（用于后续 3 个并发下载任务），推荐机型 SA5.LARGE8（性价比高，满足下载需求），系统盘≥300GB，并分配 “免费公网 IP” 及带宽≥100Mbps（保障单节点下载速度，缩短整体耗时）。

7. 创建推理节点：需 1 个 GPU 节点（大模型推理依赖高显存），推荐选择 PNV5b.8XLARGE96机型（单卡 48GB 显存，满足模型需求）。

步骤2：创建存储组件

需创建 StorageClass（CFS 存储）和 PVC，确保模型下载后能被推理服务直接调用：

1. 进入集群的**存储 > StorageClass** 页面，单击**新建**，存储类型选 “文件存储 CFS”，可用区与推理节点一致（减少跨区访问延迟），其余参数默认；

2. 进入**存储 > PVC** 页面，单击**新建**，名称为 ai-model（与后续下载脚本参数一致，避免手动修改配置），存储类选择上一步创建的 StorageClass 对象。

部署模型

本实践使用 vllm 推理框架，该框架支持的 embedding 模型主要有：**BGE** 系列模型和 **E5-Mistral** 系列模型，这两个系列的模型均可使用 vllm 框架进行模型推理，**本文采用 E5-mistral-7b-instruct 模型完成验证。**

E5-mistral-7b-instruct：基于 Mistral-7B 架构微调的指令遵循模型，融合了 E5 系列在检索增强领域的优势与 Mistral 的高效推理能力；在中文与多语言理解、长文本处理和检索增强生成（RAG）场景中表现突出，并且拥有较长的上下文长度，支持 vllm 框架下的推理服务。

步骤1：下载依赖

登录集群中的节点，节点镜像为 TencentOS 可以使用以下命令进行安装：

```
yum install -y jq
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
| bash
```

步骤2：下载模型

进入集群中，拉取 tke-ai-playbook，机器若提示安装 git，请先安装 git 后再拉取代码：

```
git clone https://github.com/tkestack/tke-ai-playbook.git
```

执行 playbook 中的下载脚本 tke-llm-downloader.sh，该脚本可以使用多个节点并发拉取模型，加快模型下载速度，PVC 名称需与前述一致，模型名称参数则选择目标模型名称，可前往 ModelScope 获取模型名称。
tke-llm-downloader.sh 是一个在 Kubernetes 集群中自动化下载大语言模型的工具脚本，主要功能包括：

- **模型下载管理：**支持从 ModelScope 或 Hugging Face 下载大语言模型到 Kubernetes PVC 存储；
- **并发下载优化：**通过 Kubernetes Job 的 completions 和 parallelism 参数实现多 Pod 并发下载，提升下载速度；
- **节点调度控制：**避免多个下载 Pod 调度到同一节点。

```
bash scripts/tke-llm-downloader.sh --pvc ai-model --completions 3 --
parallelism 3 --model intfloat/e5-mistral-7b-instruct
```

预期结果：三个 Pod 状态为 Completed 后，模型下载工作则执行完成。

```
tke-llm-downloader-6bcrn-0-dq2mv    0/1     Completed   0
tke-llm-downloader-6bcrn-1-7nlfs    0/1     Completed   0
tke-llm-downloader-6bcrn-2-5kmq7    0/1     Completed   0
```

步骤3：模型部署

vllm-inference-tke 是一个部署 vllm 推理框架并暴露服务的 Helm Chart 包，可支持单机单卡、单机多卡、多机多卡等多种部署模式，修改自定义参数即可部署自建大模型。
进入 vllm-inference-tke 目录，运行 vllm-inference-tke 的 Chart 包，该 Chart 可以在 TKE 上部署基于 vLLM 的 OpenAI 兼容 API 服务，APIKey 需要自行设置，默认无需 APIKey，修改 model name 为自己想要部署的模型，并且 PVC 修改为自己集群的 PVC 名称，具体参数修改参考下述，目前默认参数支持单机单卡部署。

```
cd tke-ai-playbook/helm-charts/vllm-inference-tke
vim values.yaml
helm install vllm-service .
```

本文提供部署 E5-mistral-7b-instruct 模型的具体参数，具体的 values.yaml 文件内容如下：

```
model:
  name: "intfloat/e5-mistral-7b-instruct"
  pvc:
    enabled: true
    name: "ai-model"
    path: "intfloat/e5-mistral-7b-instruct"
  local:
    enabled: false
    path: "/data0/intfloat/e5-mistral-7b-instruct"

server:
  replicas: 1
  lwsGroupSize: 1
  image: "ccr.ccs.tencentyun.com/tke-ai-playbook/vllm-openai:v0.10.1-
20250801"
  imagePullPolicy: IfNotPresent
  apiKey: ""
  resources:
    requests:
      nvidia.com/gpu: 1
    limits:
      nvidia.com/gpu: 1
  args:
    tpSize: 1
    ppSize: 1
    epEnabled: false
    maxModelLen: 2048
    maxBatchSize: 32
  extraArgs:
  - --disable-log-requests
  - --cuda-graph-sizes 1 2 4 8 16 24 32
  env:
  - name: VLLM_WORKER_MULTIPROC_METHOD
    value: "spawn"
  service:
    enabled: true
    type: LoadBalancer
    port: 60000

labels: {}
podAnnotations: {}
```

部署的模型会通过 service 暴露端点和端口，查看 Pod 日志看模型状态是否健康。

```
kubectl get svc | grep vllm-service
kubectl logs -f pod-name
```

预期结果：

```
(APIServer pid=1) INFO 09-25 19:54:39 [api_server.py:1847] Starting vLLM API server 0 on http://0.0.0.0:60000
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:29] Available routes are:
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /openapi.json, Methods: HEAD, GET
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /docs, Methods: HEAD, GET
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /docs/oauth2-redirect, Methods: HEAD, GET
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /redoc, Methods: HEAD, GET
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /health, Methods: GET
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /load, Methods: GET
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /ping, Methods: POST
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /ping, Methods: GET
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /tokenize, Methods: POST
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /detokenize, Methods: POST
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /v1/models, Methods: GET
(APIServer pid=1) INFO 09-25 19:54:39 [launcher.py:37] Route: /version, Methods: GET
```

步骤4：测试向量嵌入

```
curl -X POST http://xx.xx.xx.xx:60000/v1/embeddings \
  -H 'Content-Type: application/json' \
  -H 'Authorization: Bearer 3bxxxxxxxxxxxxxx1a1d' \
  -d '{
    "model": "intfloat/e5-mistral-7b-instruct",
    "input": "用Python实现快速排序算法"
  }'
```

预期结果：


```
{"id":"embd-f3eda39c42dc4ab5a264964760ba370c","object":"list","created":1758856987,"model":"intfloat/e5-mistral-7b-instruct","data":[{"index":0,"object":"embedding","embedding":[0.022728852927684784,-0.01275563798844814 3,-0.005278590135276318,-0.007603001315146685,0.014450282789766788,-0.014072422869503498,0.0020195473916828632,0.015618214383721352,0.03764859959483147,0.00032651255605742335,0.018744146451354027,-0.00916596781462431,0.012629684992134571,0.011759461835026741,0.007545750122517347,-0.006068660877645016,0.010992291383445263,-0.003555319271981716,-0.0031488335225731134,-0.008186967112123966,-0.014770891517400742,-0.018125830218195915,-0.0167403444647789,-0.011456028558313847,0.011249923147261143,0.011965567246079445,0.02628989703953266,-0.013866317458450794,0.0009203181252814829,-0.0017590528586879373,0.015721267089247704,0.005450344644486904,0.03471732139587402,0.008845359086990356,-0.016705993562936783,-0.0021712637972086 67,0.01946551539003849,-0.0026249820366501 81,-0.021423518657684326,0.025213567540049553,-0.0038015006575733423,-0.008106814697384834,-0.006143087986856699,-0.0032547488808631897,0.0013389699161052704,0.010391150601208 21,-0.0013003250351175666,0.015217453241348267,-0.0012144477805122733,0.004903592634946108,-0.01469073910266161,-0.009057190269231796,-0.016156379133462906,0.15709815919399261,0.004811990074813366,-0.005178399849683046,0.0220990851521492,-0.006246140692383051,-0.0095438277348876,-0.020221235230565 07,-8.945549780037254e-05,0.00525282695889473,-0.01725560799241066,-0.028625758364796 64,0.008295744657516 48,0.0007295990362763405,-0.008061014115810394,-0.012309076264500618,0.02285480499267578,-0.02393113449215889,-0.01504569873213768,-0.01777087152004242,-0.008026662282645702,0.005833929404616356,0.005178399849683046,0.001452041557058692,-0.007832007482647896,-0.007110638543963432,-0.004686036612838507,-0.02759523130953312,0.005095385015010834,-0.01163923367857933,0.015125851146876812,-0.0038931032177060843,-0.019488416612148285,0.008473224937915802,0.0014198375865817 07,-0.010562905110418797,0.014793791808187962,-0.010757559910416603,-0.007528574671596289,-0.01053427904844284,-0.002437483286485076,0.052075982093811035,9.446500189369544e-05,-0.009183143265545368,-0.022225039079785347,-0.0021941643208265305,0.00021022041619289666,-0.012893041595816612,0.019946428015828133,-0.008788107894361 02,-0.010225120931863785,0.017404461279511 45,0.001958001870661974,-0.013248001225292 6
```


常见问题

部署 vllm-service 后日志显示未找到模型?

检查模型是否已下载完成，Pod 状态应为 Completed。如果模型下载的 Pod 还是 Running 状态，请等待下载结束后重启服务。

部署 vllm-service 后日志显示 CudaOutofMemory?

说明 GPU 显存不足，请选用更大显存的 GPU，部署本文模型需单卡 L20 及以上机型。

AI 网关部署

最近更新时间：2025-10-10 10:36:21

本文档介绍在 TKE 上如何部署 Envoy AI Gateway 并接入大模型。

简介

AI 网关是面向大模型服务的专用流量治理组件，基于传统 API 网关扩展而来，核心能实现多模型按需切换、租户鉴权与配额管控、Token 级限流、内容安全合规校验，还可在模型异常时自动切换保障稳定性。
Envoy AI Gateway 是一个开源的、基于 Kubernetes 原生架构的 AI 网关，专门用于管理和路由大模型服务流量。该项目构建在成熟的 Envoy Proxy 之上，为应用客户端与各种 AI 服务提供商之间提供了一个统一、安全且可扩展的接入层。

环境准备

1. 已创建并部署好 TKE 集群，建议选择**香港地域**集群，如果您还没有集群，请参见 创建集群 。
2. 已创建节点池，并且节点池内有至少3个节点，推荐机型为 SA5.LARGE8。

环境验证

验证 kubectl：

```
kubectl version --client
```

预期结果：

```
Client Version: v1.32.2-tke.6
Kustomize Version: v5.5.0
```

验证 helm 安装：

```
helm version
```

预期结果：

```
version.BuildInfo{Version:"v3.19.0",
GitCommit:"3d8990f0836691f0229297773f3524598f46bda6",
GitTreeState:"clean", GoVersion:"go1.24.7"}
```

如果显示未找到 helm 命令，TencentOS 系统可以用下列命令一键安装：

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
| bash
```

验证 curl 安装：

```
curl --version
```

```
curl 8.4.0 (x86_64-koji-linux-gnu) libcurl/8.4.0 OpenSSL/3.0.12
zlib/1.2.13 brotli/1.1.0 libidn2/2.3.4 libpsl/0.21.2 (+libidn2/2.3.4)
libssh/0.10.5/openssl/zlib nghttp2/1.58.0 OpenLDAP/2.6.5
Release-Date: 2023-10-11
Protocols: dict file ftp ftps gopher gophers http https imap imaps ldap
ldaps mqtt pop3 pop3s rtsp scp sftp smb smbs smtp smtps telnet tftp
Features: alt-svc AsynchDNS brotli GSS-API HSTS HTTP2 HTTPS-proxy IDN
IPv6 Kerberos Largefile libz NTLM PSL SPNEGO SSL threadsafe TLS-SRP
UnixSocket
```

部署 AI 网关

第一步：部署 Envoy Gateway

Envoy AI Gateway 建立在 Envoy Gateway 之上，使用 Helm 安装并等待部署准备就绪：

```
helm upgrade -i eg oci://docker.io/envoyproxy/gateway-helm \
    --version v0.0.0-latest \
    --namespace envoy-gateway-system \
    --create-namespace

kubectl wait --timeout=2m -n envoy-gateway-system deployment/envoy-
gateway --for=condition=Available
```

第二步：部署 Envoy AI Gateway

安装 AI Gateway 的 Helm chart，完成后等待部署准备就绪：

```
helm upgrade -i aieg oci://docker.io/envoyproxy/ai-gateway-helm \
    --version v0.0.0-latest \
    --namespace envoy-ai-gateway-system \
    --create-namespace
```

```
kubectl wait --timeout=2m -n envoy-ai-gateway-system deployment/ai-
gateway-controller --for=condition=Available
```

第三步：配置 Envoy AI Gateway

安装 Envoy AI Gateway 后，将特定于 AI Gateway 的配置应用于 Envoy Gateway，重新启动部署，然后等待其准备就绪：

```
kubectl apply -f https://raw.githubusercontent.com/envoyproxy/ai-
gateway/main/manifests/envoy-gateway-config/redis.yaml
kubectl apply -f https://raw.githubusercontent.com/envoyproxy/ai-
gateway/main/manifests/envoy-gateway-config/config.yaml
kubectl apply -f https://raw.githubusercontent.com/envoyproxy/ai-
gateway/main/manifests/envoy-gateway-config/rbac.yaml

kubectl rollout restart -n envoy-gateway-system deployment/envoy-gateway

kubectl wait --timeout=2m -n envoy-gateway-system deployment/envoy-
gateway --for=condition=Available
```

第四步：检查健康状态

检查 AI 网关 Pod：

```
kubectl get pods -n envoy-ai-gateway-system
```

检查 Envoy Gateway 容器：

```
kubectl get pods -n envoy-gateway-system
```

第五步：部署测试后端

首先部署包含测试后端的基本 AI 网关设置：

```
kubectl apply -f https://raw.githubusercontent.com/envoyproxy/ai-
gateway/main/examples/basic/basic.yaml
```

等待网关容器准备就绪：

```
kubectl wait pods --timeout=2m \     -l gateway.envoyproxy.io/owning-
gateway-name=envoy-ai-gateway-basic \     -n envoy-gateway-system \     --
```

```
for=condition=Ready
```

设置网关 URL：

```
export GATEWAY_URL=$(kubectl get gateway/envoy-ai-gateway-basic -o
jsonpath='{.status.addresses[0].value}')
```

验证 URL 是否可用：

```
echo $GATEWAY_URL
```

第六步：测试 AI 网关

打开一个新终端，并使用以下命令向 AI 网关发送测试请求：

```
curl -H "Content-Type: application/json" \
-d '{ "model": "some-cool-self-hosted-model","messages":
[{"role":"system","content": "Hi."}]}' \
$GATEWAY_URL/v1/chat/completions
```

预期结果：

```
{"choices":[{"message":{"role":"assistant", "content":"I am
inevitable."}}]}
```

使用 AI 网关接入大模型

第一步：下载配置模板

本文实践以 Kimi 大模型为例，先获取 AI 网关的默认模型配置模板，后续第二步修改模板中的 hostname 和 API_KEY，需要具备相关运营商的 API_KEY。

```
curl -O https://raw.githubusercontent.com/envoyproxy/ai-
gateway/main/examples/basic/openai.yaml
```

第二步：修改配置模板

编辑文件 openai.yaml 替换其中的 hostname 和 OPENAI_API_KEY，下面以 Kimi 为例，yaml 文件配置参考如下：

```
apiVersion: aigateway.envoyproxy.io/v1alpha1
kind: AIGatewayRoute
metadata:
  name: envoy-ai-gateway-basic-openai
  namespace: default
spec:
  parentRefs:
    - name: envoy-ai-gateway-basic
      kind: Gateway
      group: gateway.networking.k8s.io
  rules:
    - matches:
        - headers:
            - type: Exact
              name: x-ai-eg-model
              value: kimi-k2-0905-preview
      backendRefs:
        - name: envoy-ai-gateway-basic-openai
---
apiVersion: aigateway.envoyproxy.io/v1alpha1
kind: AIServiceBackend
metadata:
  name: envoy-ai-gateway-basic-openai
  namespace: default
spec:
  schema:
    name: OpenAI
  backendRef:
    name: envoy-ai-gateway-basic-openai
    kind: Backend
    group: gateway.envoyproxy.io
---
apiVersion: aigateway.envoyproxy.io/v1alpha1
kind: BackendSecurityPolicy
metadata:
  name: envoy-ai-gateway-basic-openai-apikey
  namespace: default
spec:
  targetRefs:
    - group: aigateway.envoyproxy.io
      kind: AIServiceBackend
      name: envoy-ai-gateway-basic-openai
  type: APIKey
```

```
  apiKey:
    secretRef:
      name: envoy-ai-gateway-basic-openai-apikey
      namespace: default
---
apiVersion: gateway.envoyproxy.io/v1alpha1
kind: Backend
metadata:
  name: envoy-ai-gateway-basic-openai
  namespace: default
spec:
  endpoints:
    - fqdn:
        hostname: api.moonshot.cn
        port: 443
---
apiVersion: gateway.networking.k8s.io/v1alpha3
kind: BackendTLSPolicy
metadata:
  name: envoy-ai-gateway-basic-openai-tls
  namespace: default
spec:
  targetRefs:
    - group: 'gateway.envoyproxy.io'
      kind: Backend
      name: envoy-ai-gateway-basic-openai
  validation:
    wellKnownCACertificates: "System"
    hostname: api.moonshot.cn
---
apiVersion: v1
kind: Secret
metadata:
  name: envoy-ai-gateway-basic-openai-apikey
  namespace: default
type: Opaque
stringData:
  apiKey: sk-xxxxxxxxxxxxxx  # Replace with your kimi API key.
```

第三步：应用配置

应用更新的配置并等待网关容器准备就绪。

```
kubectl apply -f openai.yaml

kubectl wait pods --timeout=2m \
  -l gateway.envoyproxy.io/owning-gateway-name=envoy-ai-gateway-basic \
  -n envoy-gateway-system \
  --for=condition=Ready
```

第四步：测试配置

```
curl -H "Content-Type: application/json"   -d '{
    "model": "kimi-k2-0905-preview",
    "messages": [
      {
        "role": "user",
        "content": "Hi."
      }
    ]
  }'   $GATEWAY_URL/v1/chat/completions
```

预期结果：

```
{"id":"chatcmpl-
68d64bc7a5422d1970e278da","object":"chat.completion","created":175887456
7,"model":"kimi-k2-0905-preview","choices":[{"index":0,"message":
{"role":"assistant","content":"Hi there! How can I help you
today?"},"finish_reason":"stop"}],"usage":
{"prompt_tokens":9,"completion_tokens":11,"total_tokens":20}}
```

常见问题

1. 在执行 helm 之后，安装 chart 命令超时，无法安装 chart?

推荐集群选择香港地域，防止由于网络延迟等问题导致安装超时。

2. curl 命令测试之后，返回报错：No matching route found. It is likely that the model specified your request is not configured in the Gateway

检查 yaml 文件中的模型文件是否与 curl 命令中的一致。