

大模型服务 最佳实践

产品版本: v7.0.2

发布日期: 2026-10-08

目录

1 最佳实践	1
1.1 Rerank模型通过 Cohere 规范对外提供服务	1
1.2 使用代理接入外部模型提供商	8
1.3 向量模型部署和访问	23
1.4 LoRA模型部署和访问	28
1.5 vLLM模型启动参数调优方式	32

1 最佳实践

1.1 Rerank模型通过 Cohere 规范对外提供服务

背景描述

模型网关对外提供两种 API 规范：OpenAI 规范提供对话、文生图、向量等接口；Cohere 规范只提供一个接口——重排序（rerank）。两者在网关内部是完全互斥的：网关的 rerank 端点只接受 Cohere 规范的后端，而其余所有端点只接受 OpenAI 规范的后端，互相调用会被网关拒绝。

重排序是检索增强生成（RAG）链路里的第二道筛选。第一道用向量检索从海量文档里粗筛出几十条候选，速度快但精度有限；第二道把「查询 + 每条候选」成对送进重排序模型，逐对打分后重新排序，只把最相关的几条交给大模型。把重排序接入模型网关，意味着它和对话模型一样纳入统一的域名、鉴权、套餐限流和 Token 计量体系，业务侧不需要再为它单独维护一套地址和密钥。

本文以部署一个重排序模型并通过 Cohere 规范对外提供服务为例，介绍从上传推理镜像到验证调用、查看用量的完整流程。

前提条件

- 已具备项目管理员权限，可以初始化命名空间和设置配额。
- 环境中可有可用的 AI 加速卡资源（英伟达 GPU、海光 HCU 或昇腾 NPU）。
- 已准备好用于上传镜像的客户端环境（安装有 `ctr` 或 `docker` 命令）。
- **推理引擎必须使用 vLLM**。具体原因见下方说明。

说明：

SGLang 目前不能用于 Cohere 规范的重排序服务。网关的 Cohere 转换器要求后端提供

`/v2/rerank` 接口，vLLM 会同时注册 `/rerank`、`/v1/rerank`、`/v2/rerank` 三个等价接口，而 SGLang 只注册 `/v1/rerank`。因此用 SGLang 部署的重排序模型，在创建 AI 服务提供者时不会出现 Cohere 选项。此类模型仍可用 SGLang 部署后由业务直连，只是无法通过模型网关以 Cohere 规范发布。

操作步骤

1. 上传推理镜像到容器镜像服务。

1. 在《容器镜像服务》中创建一个共享空间（建议命名为 `engines` ），把所有推理引擎镜像集中存放在该空间下，便于统一管理和授权。
2. 从如下地址列表中拉取与本环境加速卡型号匹配的 vLLM 镜像，重新打标签后推送到环境的镜像仓库。重排序模型的输出是一组分数，属于文本类输出，**使用非 omni 镜像**。

加速卡型号	镜像地址	需打的标签
英伟达 GPU	<code>vllm/vllm-openai:v0.29.0</code> <code>hub.easystack.cn/engines/vllm-openai:v0.29.0</code>	<code>vllm</code> 、 <code>nvidia</code>
海光 HCU	<code>harbor.sourcefind.cn:5443/dcu/admin/base/vllm:0.18.1-ubuntu22.04-dtk26.04-py3.10</code> <code>hub.easystack.cn/engines/vllm:0.18.1-ubuntu22.04-dtk26.04-py3.10</code>	<code>vllm</code> 、 <code>hygon</code>
昇腾 NPU	<code>quay.io/ascend/vllm-ascend:v0.23.0</code> <code>hub.easystack.cn/engines/vllm-ascend:v0.23.0</code>	<code>vllm</code> 、 <code>huawei-ascend</code>

说明：

访问hub.easystack.cn需要联系EasyStack相关人员下载

3. 在《容器镜像服务》的镜像详情页面，为镜像打上表中对应的两个标签。标签决定了后续创建推理引擎时能否自动带出该镜像，**漏打标签会导致推理引擎页面选不到镜像**。

2. 初始化项目命名空间。

项目管理员登录后进入《模型广场》，按页面提示完成本项目所需命名空间的初始化。该操作每个项目只需执行一次。

3. 设置项目配额。

在《配额设置》中为本项目分配所需配额，容器规格选择「安全容器服务」的配置。分配CPU，内存容量和AI加速卡数量，重排序模型通常远小于对话模型（bge-reranker-v2-m3 约 2.3GB），显存配额可以相应调低，但需要为并发预留余量。

4. 新建模型。

1. 进入《模型广场》，单击 **新建模型**。
2. 选择模型来源。可以从 ModelScope 在线导入，也可以先在《模型仓库》创建上传任务手动上传模型文件，再在此处选择「绑定已有模型文件」。
3. 选择适用于重排序的模型。两类模型都能通过校验，但效果差别很大：

模型类型	代表模型	说明
交叉编码器（推荐）	BAAI/bge-reranker-v2-m3	把「查询 + 文档」拼成一对送入模型联合打分，能捕捉两者之间的交互信息，这是重排序真正的价值所在
双塔向量模型	BAAI/bge-m3	查询和文档分别独立编码后算相似度

说明：

双塔向量模型也能通过重排序接口调用并返回分数，但实测其返回的分数与「自行调用向量接口再计算余弦相似度」的结果完全一致（四位小数内无差异）。也就是说，用向量模型做重排序，相当于把粗筛阶段已经算过的相似度重新算了一遍，并不能提升排序质量。**如果目标是提升 RAG 的最终效果，请使用交叉编码器模型。**如果只是想让一个已部署的向量模型顺带提供 rerank 接口，用双塔模型也可以，但要清楚它不会带来额外收益。

5. 创建推理引擎。

1. 进入《推理引擎》，单击 **创建推理引擎**。
2. 选择 AI 加速卡型号，镜像会根据步骤 1 中打的标签自动带出，选择对应的 vLLM 镜像。
3. 其余参数保持默认即可，按需修改。

6. 部署模型。

1. 回到《模型广场》，选择步骤 4 中新建的模型，编辑设置推理引擎，或直接单击 **部署**。
2. 选择步骤 5 中创建的推理引擎，单击 **创建**，等待模型部署状态变为就绪。

3. 部署就绪后，务必先确认该服务已注册重排序接口，再进行下一步。平台正是通过这个接口清单来判断一个部署能否以 Cohere 规范发布的。进入模型部署的 [详情](#) 页面，进入实例的 [终端](#)。验证命令如下：

```
curl -s http://127.0.0.1:8000/openapi.json | grep -o '/v2/rerank'
```

返回 `"/v2/rerank"` 表示正常。如果没有返回，说明该模型不具备重排序能力，或推理引擎未正确识别模型任务，此时后续步骤中不会出现 Cohere 选项。

7. 创建 AI 服务提供者。

1. 进入《AI 服务提供者》，单击 [创建](#)，服务类型选择「内部」。
2. 先选择「API 规范」为 Cohere，再选择模型部署。

说明：

表单是「规范在前、部署在后」的顺序，选定规范后，下方的部署列表会自动过滤，只展示能够提供该规范的部署，并在提示中给出过滤前后的数量。这样可以避免选到一个无法提供重排序的部署。

如果下拉框中没有 Cohere 选项且整个选择框为灰色，说明当前项目下**没有任何一个部署**注册了 `/v2/rerank`。请回到步骤 6 确认接口注册情况。另外，如果部署刚创建尚未完全就绪，接口探测可能超时（探测上限 6 秒），此时关闭对话框稍后重开会重新探测。

3. **API 规范创建后不可修改**。一个 AI 服务提供者只能是一种规范，Cohere 规范的提供者只能用于重排序，无法再用于对话或向量接口。如需同时提供多种接口，请创建多个 AI 服务提供者。
4. 单击 [创建](#)。

8. 创建模型网关路由。

1. 进入《模型网关》，单击 [创建路由](#)。
2. 输入路由域名，后端选择步骤 7 中创建的 Cohere 规范 AI 服务提供者，单击 [创建](#)。
3. 如果为同一个模型配置了多个后端（主用 + 备用），**这些后端必须全部是 Cohere 规范**。同一模型下混用不同规范的后端，在主用后端正常时表现正常，一旦触发故障转移切到另一规范的后端，请求会直接失败。表单和接口都会对此进行校验并拒绝提交。

说明：

同一条路由下的**不同模型**之间可以使用不同规范。例如一条路由里同时挂载一个 Cohere 规范的重排序模型和一个 OpenAI 规范的向量模型，两者互不影响，这是推荐的组织方式——RAG 链路所需的向量和重排序两个能力，可以共用同一个域名和同一个 API Key。

9. 创建套餐。

进入《套餐管理》，单击 **创建套餐**，「可访问路由」选择步骤 8 中创建的路由，按业务需要设置调用频率和 Token 限额后创建。

10. 创建 API Key。

进入《API Key》，单击 **创建**，选择步骤 9 中创建的套餐，单击 **创建并签发密钥**。签发成功后复制密钥妥善保管，**密钥只在此时完整展示一次**。

11. 查看 Token 用量。

进入《Token 用量》，在「请求类型」筛选器中选择「rerank」，即可查看重排序请求的调用次数和 Token 消耗。

结果验证

1. 确认域名可解析到网关地址。路由是按域名匹配的，域名未解析时请求不会到达网关。可以配置 hosts 记录，也可以在 curl 命令中使用 **--resolve** 参数临时指定，后者不会修改本机配置。
2. 执行重排序调用，确认返回结果按相关度排序。具体命令如下：

```
curl -s http://<路由域名>:<网关端口>/cohere/v2/rerank \
-X POST \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer ' \
-d '{
  "model": "<模型名称>",
  "query": "什么是 EasyStack MaaS? ",
  "documents": [
    "EasyStack MaaS 是一个模型即服务平台",
    "今天天气不错",
    "MaaS 平台提供模型部署与网关能力"
  ],
  "top_n": 2
}'
```

正常返回示例如下，**index** 是候选文档在请求 **documents** 数组中的下标，**relevance_score** 是相关度分数，结果按分数从高到低排列：

```
{
  "results": [
    { "index": 0, "relevance_score": 0.9832 },
    { "index": 2, "relevance_score": 0.8107 }
  ],
  "meta": { "tokens": { "input_tokens": 48 } }
}
```

请求体的字段约束如下：

字段	是否必填	说明
model	是	路由中配置的模型名称
query	是	用于排序的查询语句
documents	是	必须是字符串数组 。传入对象数组会被网关在转发前直接拒绝
top_n	否	只返回相关度最高的 N 条
max_tokens_per_doc	否	单条文档的截断长度，默认 4096

3. 确认访问路径正确。网关注册的路径是 `/cohere/v2/rerank`，**必须完全一致**。访问 `/v2/rerank`、`/cohere/v1/rerank` 等其他路径会返回 404，提示路径不受支持。
4. 确认限流与计量生效。
 1. 连续调用超过套餐设置的频率上限，确认返回 429。
 2. 回到《Token 用量》，确认出现请求类型为 rerank 的记录，且 Token 数量与请求内容规模相符。

约束与注意事项

- **规范互斥**：Cohere 规范只提供重排序一个接口。用 Cohere 规范的提供者去调用对话、向量等接口会返回 500；反之用 OpenAI 规范的提供者去调用重排序接口同样返回 500。
- **规范不可变更**：AI 服务提供者的 API 规范在创建后无法修改。若某个提供者已被模型网关路由引用，接口层面也会拒绝变更请求，需先从路由中移除。

-
- **同一模型的后端规范必须一致：**主用和备用后端必须是同一规范，否则故障转移时请求会失败。不同模型之间不受此限制。
 - **请求超时：**网关对单次请求的超时时间为 300 秒，该时长包含重试在内。候选文档数量极大时需要注意单次请求的耗时。
 - **删除模型部署前先检查引用：**若待删除的模型部署正被 AI 服务提供者引用，删除操作会弹出二次确认并列出引用方。删除后提供者会指向一个已不存在的服务，使用该提供者的网关路由将开始报错，且平台不会自动清理，需要手工处理。
 - **推理引擎选型：**如前所述，当前只有 vLLM 可用于 Cohere 规范的重排序服务。
 - **效果预期：**使用双塔向量模型提供重排序接口在功能上可行，但其分数与向量余弦相似度等价，不会带来排序质量的提升。追求 RAG 效果请选用交叉编码器模型。

1.2 使用代理接入外部模型提供商

背景描述

企业 MaaS 平台需要接入 OpenAI、Anthropic 等模型服务，而这些服务在部署环境中无法直连，必须经由出口代理出网。这里采用中转（relay）方案，此方案会将目标地址改变，平台改去连一个集群内的服务，由该服务替你转发到模型提供商。这个方案只需在集群内部署一个中转服务，AI 服务提供者指向它即可。

本文介绍如何部署 OpenAI 和 Anthropic 的中转服务，并将其接入模型网关对外提供服务。

前提条件

- 已具备《安全容器服务》的部署创建权限，以及《大模型服务》中创建 AI 服务提供者、路由、套餐和 API Key 的权限。
- 已获取可用的 OpenAI API Key 和 / 或 Anthropic API Key。两家是完全独立的账号体系，密钥互不通用：

服务商	密钥获取地址	密钥前缀
OpenAI	https://platform.openai.com/login → API keys	sk -
Anthropic	https://console.anthropic.com/settings/keys	sk-ant -

说明：

如果只是功能验证而账户尚未充值，可以在上表对应的控制台自行创建一个密钥。余额为零时模型列表接口（`/v1/models`）仍可正常返回，足以验证链路是否打通；真正发起推理才需要账户有余额。

- 已确认集群内可以访问公司出口代理 `http://cloud-proxy.fake.io`。可在任意一个业务 Pod 内执行以下命令验证：

```
curl -sS -o /dev/null -w '%{http_code}\n' -x http://cloud-proxy.fake.io https://api.openai.com/v1/models
```

返回 401 即表示代理链路正常（401 是因为没带 API Key，说明请求已经到达 OpenAI）。若返回 000 或超时，请先排查代理可达性，后续步骤无法进行。

操作步骤

1. 准备中转服务的 YAML。

中转服务是一个极小的 Go 程序，职责只有三件：接收集群内的请求、经出口代理转发到提供商、把响应原样回传。

两家服务商各用一份独立的 YAML：只接入一家，就用对应的那一份；两家都要，就把两份都部署一次。

(1) OpenAI —— 保存为 `openai-proxy.yaml`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: go-relay
data:
  main.go: |
    package main

    import (
        "log"
        "net/http"
        "net/http/httputil"
        "net/url"
        "os"
        "time"
    )

    func main() {
        targetURL := os.Getenv("TARGET_URL")
        if targetURL == "" {
            targetURL = "https://api.openai.com"
        }

        listenAddr := os.Getenv("LISTEN_ADDR")
        if listenAddr == "" {
            listenAddr = ":8080"
```

```
}

target, err := url.Parse(targetURL)
if err != nil {
    log.Fatalf("invalid TARGET_URL: %v", err)
}

proxy := httputil.NewSingleHostReverseProxy(target)

// 保证发给 OpenAI 的 Host 正确
originalDirector := proxy.Director
proxy.Director = func(req *http.Request) {
    originalDirector(req)
    req.Host = target.Host
}

// 使用 HTTP_PROXY / HTTPS_PROXY / NO_PROXY
// HTTPS 上游会自动通过 HTTP CONNECT 走代理
proxy.Transport = &http.Transport{
    Proxy: http.ProxyFromEnvironment,

    MaxIdleConns:    100,
    MaxIdleConnsPerHost: 100,
    IdleConnTimeout:  90 * time.Second,

    TLSHandshakeTimeout: 10 * time.Second,
    ExpectContinueTimeout: 1 * time.Second,
}

// 对 SSE / OpenAI streaming 尽快 flush
proxy.FlushInterval = -1

mux := http.NewServeMux()

// Kubernetes 探针使用, 不转发给 OpenAI
mux.HandleFunc("/healthz", func(w http.ResponseWriter, r
    *http.Request) {
    w.WriteHeader(http.StatusOK)
    _, _ = w.Write([]byte("ok"))
})
```

```
// 其他请求全部转发
mux.Handle("/", proxy)

log.Printf("relay listen=%s target=%s", listenAddr, targetURL)
log.Printf("HTTP_PROXY=%s", os.Getenv("HTTP_PROXY"))
log.Printf("HTTPS_PROXY=%s", os.Getenv("HTTPS_PROXY"))
log.Printf("NO_PROXY=%s", os.Getenv("NO_PROXY"))

server := &http.Server{
    Addr:          listenAddr,
    Handler:       mux,
    ReadHeaderTimeout: 30 * time.Second,

    // 不设置 WriteTimeout, 避免限制长时间 streaming
    IdleTimeout: 120 * time.Second,
}

log.Fatal(server.ListenAndServe())
}
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: openai-relay
spec:
  replicas: 1

  selector:
    matchLabels:
      app: openai-relay

  template:
    metadata:
      labels:
        app: openai-relay

    spec:
      containers:
        - name: relay
          image: golang:1.23-bullseye
```

```
imagePullPolicy: IfNotPresent

command:
  - go

args:
  - run
  - /app/main.go

env:
  # 真正的上游地址
  - name: TARGET_URL
    value: "https://api.openai.com"

  - name: LISTEN_ADDR
    value: ":8080"

  # =====
  # 修改成你的企业 HTTP 正向代理
  # =====
  - name: HTTPS_PROXY
    value: "https://cloud-proxy.fake.io"

  - name: HTTP_PROXY
    value: "http://cloud-proxy.fake.io"

  - name: NO_PROXY
    value: "localhost,127.0.0.1,.svc,.cluster.local"

ports:
  - name: http
    containerPort: 8080

volumeMounts:
  - name: source
    mountPath: /app
    readOnly: true

readinessProbe:
  httpGet:
    path: /healthz
```

```
        port: 8080
        initialDelaySeconds: 2
        periodSeconds: 5

    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10

    volumes:
      - name: source
        configMap:
          name: go-relay

---
apiVersion: v1
kind: Service
metadata:
  name: openai-relay
spec:
  selector:
    app: openai-relay

  ports:
    - name: http
      port: 8080
      targetPort: 8080

  type: ClusterIP
```

(2) Anthropic —— 保存为 anthropic-proxy.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: go-relay-anthropic
data:
  main.go: |
```

```
package main

import (
    "log"
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
    "strings"
    "time"
)

func main() {
    targetURL := os.Getenv("TARGET_URL")
    if targetURL == "" {
        targetURL = "https://api.anthropic.com"
    }

    listenAddr := os.Getenv("LISTEN_ADDR")
    if listenAddr == "" {
        listenAddr = ":8080"
    }

    // Anthropic authenticates with x-api-key, not "Authorization:
    Bearer",
    // which is the only form the model gateway sends. Set AUTH_MODE to
    // "anthropic" on a relay whose TARGET_URL is api.anthropic.com.
    anthropic := os.Getenv("AUTH_MODE") == "anthropic"
    anthropicVersion := os.Getenv("ANTHROPIC_VERSION")
    if anthropicVersion == "" {
        anthropicVersion = "2023-06-01"
    }

    target, err := url.Parse(targetURL)
    if err != nil {
        log.Fatalf("invalid TARGET_URL: %v", err)
    }

    proxy := httputil.NewSingleHostReverseProxy(target)

    // 保证发给 Anthropic 的 Host 正确
```

```
originalDirector := proxy.Director
proxy.Director = func(req *http.Request) {
    originalDirector(req)
    req.Host = target.Host
    if anthropic {
        // Move the credential onto the header Anthropic reads.
        if v := req.Header.Get("Authorization");
strings.HasPrefix(v, "Bearer ") {
            req.Header.Set("x-api-key",
strings.TrimSpace(strings.TrimPrefix(v, "Bearer ")))
            req.Header.Del("Authorization")
        }
        // Documented as required by Anthropic; nothing upstream
sets it.
        if req.Header.Get("anthropic-version") == "" {
            req.Header.Set("anthropic-version", anthropicVersion)
        }
    }
}

// 使用 HTTP_PROXY / HTTPS_PROXY / NO_PROXY
// HTTPS 上游会自动通过 HTTP CONNECT 走代理
proxy.Transport = &http.Transport{
    Proxy: http.ProxyFromEnvironment,

    MaxIdleConns:    100,
    MaxIdleConnsPerHost: 100,
    IdleConnTimeout:  90 * time.Second,

    TLSHandshakeTimeout: 10 * time.Second,
    ExpectContinueTimeout: 1 * time.Second,
}

// 对 SSE / streaming 尽快 flush
proxy.FlushInterval = -1

mux := http.NewServeMux()

// Kubernetes 探针使用, 不转发给上游
mux.HandleFunc("/healthz", func(w http.ResponseWriter, r
*http.Request) {
```

```
        w.WriteHeader(http.StatusOK)
        _, _ = w.Write([]byte("ok"))
    })

    // 其他请求全部转发
    mux.Handle("/", proxy)

    log.Printf("relay listen=%s target=%s", listenAddr, targetURL)
    log.Printf("HTTP_PROXY=%s", os.Getenv("HTTP_PROXY"))
    log.Printf("HTTPS_PROXY=%s", os.Getenv("HTTPS_PROXY"))
    log.Printf("NO_PROXY=%s", os.Getenv("NO_PROXY"))

    server := &http.Server{
        Addr:          listenAddr,
        Handler:        mux,
        ReadHeaderTimeout: 30 * time.Second,

        // 不设置 WriteTimeout, 避免限制长时间 streaming
        IdleTimeout: 120 * time.Second,
    }

    log.Fatal(server.ListenAndServe())
}

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: anthropic-relay
spec:
  replicas: 1

  selector:
    matchLabels:
      app: anthropic-relay

  template:
    metadata:
      labels:
        app: anthropic-relay
```

```
spec:
  containers:
    - name: relay
      image: golang:1.23-bullseye
      imagePullPolicy: IfNotPresent

      command:
        - go

      args:
        - run
        - /app/main.go

      env:
        # 真正的上游地址
        - name: TARGET_URL
          value: "https://api.anthropic.com"

        # 把网关下发的 Authorization: Bearer 改写成 Anthropic 要的 x-api-
key
        - name: AUTH_MODE
          value: "anthropic"

        - name: LISTEN_ADDR
          value: ":8080"

        # =====
        # 修改成你的企业 HTTP 正向代理
        # =====
        - name: HTTPS_PROXY
          value: "https://cloud-proxy.fake.io"

        - name: HTTP_PROXY
          value: "http://cloud-proxy.fake.io"

        - name: NO_PROXY
          value: "localhost,127.0.0.1,.svc,.cluster.local"

      ports:
        - name: http
          containerPort: 8080
```

```
    volumeMounts:
      - name: source
        mountPath: /app
        readOnly: true

    readinessProbe:
      httpGet:
        path: /healthz
        port: 8080
        initialDelaySeconds: 2
        periodSeconds: 5

    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10

    volumes:
      - name: source
        configMap:
          name: go-relay-anthropic

---
apiVersion: v1
kind: Service
metadata:
  name: anthropic-relay
spec:
  selector:
    app: anthropic-relay

  ports:
    - name: http
      port: 8080
      targetPort: 8080

  type: ClusterIP
```

2. 创建中转服务的部署。

1. 在云平台中依次进入《安全容器服务》-《工作负载》，单击 **创建部署**。
2. 在页面右下角单击 **导入 YAML**，粘贴需要的那一份内容，确认命名空间后提交。
3. 如果两家都要接入，重复上一步，导入另一份。两份之间没有共享资源，先后顺序随意。
4. 等待部署状态变为运行中。创建完成后，服务在集群内的访问地址为：

```
http://openai-relay.<命名空间>.svc.cluster.local:8080      # openai-  
proxy.yaml  
http://anthropic-relay.<命名空间>.svc.cluster.local:8080   # anthropic-  
proxy.yaml
```

后续步骤需要用到这些地址，请记录下实际的命名空间。

3. 验证中转服务本身。

在接入平台之前先单独验证中转链路，可以把问题范围限定在最小。在集群内任意 Pod 中执行：

```
# OpenAI: 带上自己的 API Key, 应返回模型列表  
curl -sS http://openai-relay.<命名空间>.svc.cluster.local:8080/v1/models \  
-H "Authorization: Bearer " | head -c 300  
  
# Anthropic: 注意这里同样用 Bearer, 由中转服务改写为 x-api-key  
curl -sS http://anthropic-relay.<命名空间>.svc.cluster.local:8080/v1/models \  
-H "Authorization: Bearer " | head -c 300
```

两条命令都返回 JSON 模型列表，说明「集群 → 中转 → 公司代理 → 服务提供商」整条链路已经打通。若返回 502，查看对应 Pod 日志中的 **relay error** 一行即可定位是代理不通还是上游拒绝。

4. 接入 AI 服务提供者。

1. 进入《大模型服务》-《AI 服务提供者》，单击 **创建 AI 服务提供者**。
2. 「类型」选择「**外部**」（即「外部 AI 服务（OpenAI / DeepSeek 等，需密钥）」）。
3. 按下表填写：

字段	OpenAI	Anthropic
----	--------	-----------

字段	OpenAI	Anthropic
API 规范	OpenAI	Anthropic
访问地址	<code>http://openai-relay.<命名空间>.svc.cluster.local:8080</code>	<code>http://anthropic-relay.<命名空间>.svc.cluster.local:8080</code>
访问密钥 (API Key)	OpenAI 的 API Key	Anthropic 的 API Key

4. 单击创建。

5. 创建模型网关路由。

1. 进入《模型网关》，单击 **创建路由**，输入路由域名。
2. 添加模型时，后端选择步骤 4 创建的 AI 服务提供者。此时页面会列出该提供者可用的模型名称——**能够成功列出模型，即再次证明网关经中转服务访问服务是通的。**
3. 单击 **创建**。

说明：

同一模型的多个后端必须使用相同的 API 规范，因此不要把 OpenAI 提供者和 Anthropic 提供者配置为同一个模型的主备后端。不同模型之间可以自由混用，一条路由里同时提供 OpenAI 和 Anthropic 的模型是推荐做法，业务侧只需记住一个域名和一个密钥。

6. 创建套餐并签发 API Key。

1. 进入《套餐管理》，单击 **创建套餐**，「可访问路由」选择步骤 5 创建的路由，设置调用频率与 Token 限额后创建。
2. 进入《API Key》，单击 **创建**，选择上述套餐，单击 **创建并签发密钥**，复制并妥善保存密钥。**密钥只在此时完整展示一次。**

结果验证

1. 确认路由域名可以解析到网关地址。可以配置 hosts 记录，也可以在 curl 中用 `--resolve` 临时指定。
2. 调用 OpenAI 模型，确认正常返回：

```
curl -sS http://<路由域名>:<网关端口>/v1/chat/completions \
-X POST \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer <平台签发的 API Key>' \
-d '{
  "model": "<路由中配置的模型名称>",
  "messages": [{ "role": "user", "content": "Hello!" }]
}'
```

3. 调用 Anthropic 模型。**路径和请求体都与 OpenAI 不同**，必须用 Anthropic 的 Messages API：路径是 `/anthropic/v1/messages`，请求体是 Anthropic 原生格式，其中 `max_tokens` 为必填。只有平台签发的密钥仍然放在 `Authorization: Bearer` 里，中转服务会把它改写成 Anthropic 要的 `x-api-key`。

```
curl -sS http://<路由域名>:<网关端口>/anthropic/v1/messages \
-X POST \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer <平台签发的 API Key>' \
-d '{
  "model": "<路由中配置的 Anthropic 模型名称>",
  "max_tokens": 64,
  "messages": [{ "role": "user", "content": "Hello!" }]
}'
```

4. 验证流式响应未被中转服务缓冲。在请求体中加入 `"stream": true`，确认输出是逐段返回而不是最后一次性吐出（下面以 OpenAI 为例；验证 Anthropic 时把路径和请求体换成第 3 条的形式，同样加 `"stream": true`）：

```
curl -N -sS http://<路由域名>:<网关端口>/v1/chat/completions \
-X POST \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer <平台签发的 API Key>' \
-d '{ "model": "<模型名称>", "messages": [{"role": "user", "content": "数到二十"}], "stream": true }'
```

5. 进入《Token 用量》，确认出现对应的调用记录与 Token 消耗。

约束与注意事项

- **两种 API 规范的端点和请求格式不互通。** 网关按路径挑选翻译器，`/v1/chat/completions` 只接受 OpenAI、AWSBedrock、AWSAnthropic、AzureOpenAI、GCPVertexAI、GCPAnthropic 六种规范的后端，**不含 Anthropic**；`Anthropic` 规范的后端只在 `/anthropic/v1/messages` 上提供服务，且请求体必须是 Anthropic 原生格式。
- **一个中转服务只对应一个提供商。** 上游地址在部署时固定。接入新的服务商，复制一份 YAML、改掉资源名和 `TARGET_URL` 即可；注意 ConfigMap 也要一并改名，否则会覆盖已有的那一份。
- **中转服务本身不做鉴权。** 它只负责转发，凭据由调用方携带。
- **请求超时上限为 300 秒**，由模型网关控制，该时长包含重试在内。中转服务未设置写超时，不会先于网关切断长响应。
- **多了一跳网络。** 中转服务是链路上的一个新组件，排查问题时的定位顺序是：先看中转服务 Pod 日志中的 `proxy error`，再看网关侧。
- **本方案是中转，不是正向代理。** 如果后续出现「必须保持目标地址不变」的硬性要求（例如上游按 SNI 做准入），本方案不适用，需要重新评估。

1.3 向量模型部署和访问

背景描述

向量检索是检索增强生成（RAG）链路里的第一道筛选。向量模型把查询和文档分别转换为数值向量，再按相似度从海量文档里粗筛出候选；如果还需要提升排序精度，可以把候选交给重排序模型再次打分。把向量模型接入模型网关，意味着它和对话模型一样纳入统一的域名、鉴权、套餐限流和 Token 计量体系，业务侧不需要再为它单独维护一套地址和密钥。

本文以部署一个向量模型并通过 OpenAI 规范对外提供服务为例，介绍从上传推理镜像到验证调用、查看用量的完整流程。示例模型为 `BAAI/bge-m3`，其他支持向量任务的模型也可按相同步骤部署；验证时应以实际模型的向量维度为准。

前提条件

- 已具备项目管理员权限，可以初始化命名空间和设置配额。
- 环境中可有可用的 AI 加速卡资源（英伟达 GPU、海光 HCU 或昇腾 NPU）。
- 已准备好用于上传镜像的客户端环境（安装有 `ctr` 或 `docker` 命令）。
- 已准备好向量模型文件，或环境能够访问 ModelScope 在线导入模型。

操作步骤

1. 上传推理镜像到容器镜像服务。

1. 在《容器镜像服务》中创建一个共享空间（建议命名为 `engines`），把所有推理引擎镜像集中存放在该空间下，便于统一管理和授权。
2. 从如下地址列表中拉取与本环境加速卡型号匹配的 vLLM 镜像，重新打标签后推送到环境的镜像仓库。
向量模型的输出是一组数值，属于非多模态输出，**使用非 omni 镜像**。

加速卡型号	镜像地址	需打的标签
英伟达 GPU	<code>vllm/vllm-openai:v0.29.0</code> <code>hub.easystack.cn/engines/vllm-openai:v0.29.0</code>	<code>vllm</code> 、 <code>nvidia</code>

加速卡型号	镜像地址	需打的标签
海光 HCU	harbor.sourcefind.cn:5443/dcu/admin/base/vllm: 0.18.1-ubuntu22.04-dtk26.04-py3.10 hub.easystack.cn/engines/vllm:0.18.1-ubuntu22.0 4-dtk26.04-py3.10	vllm 、 hygon
昇腾 NPU	quay.io/ascend/vllm-ascend:v0.23.0 hub.easystack.cn/engines/vllm-ascend:v0.23.0	vllm 、 huawei-ascend

说明：

访问hub.easystack.cn需要联系EasyStack相关人员下载

3. 在《容器镜像服务》的镜像详情页面，为镜像打上表中对应的两个标签。标签决定了后续创建推理引擎时能否自动带出该镜像，**漏打标签会导致推理引擎页面选不到镜像。**

2. 初始化项目命名空间。

项目管理员登录后进入《模型广场》，按页面提示完成本项目所需命名空间的初始化。该操作每个项目只需执行一次。

3. 设置项目配额。

在《配额设置》中为本项目分配所需配额，容器规格选择「安全容器服务」的配置。分配 CPU、内存容量和 AI 加速卡数量；向量模型通常远小于对话模型，可以按模型大小设置资源，但需要为并发预留余量。

4. 新建模型。

1. 进入《模型广场》，单击 **新建模型**。

2. 选择模型来源。可以从 ModelScope 在线导入，也可以先在《模型仓库》创建上传任务手动上传模型文件，再在此处选择「绑定已有模型文件」。

3. 选择适用于向量任务的模型。本文以 [BAAI/bge-m3] 为例。它支持多语言文本向量，稠密向量维度为 1024。

5. 创建推理引擎。

1. 进入《推理引擎》，单击 **创建推理引擎**。

2. 选择 AI 加速卡型号，镜像会根据步骤 1 中打的标签自动带出，选择对应的 vLLM 镜像。

3. 其余参数保持默认即可，按需修改。

6. 部署模型。

1. 回到《模型广场》，选择步骤 4 中新建的模型，编辑设置推理引擎，或直接单击 **部署**。

2. 选择步骤 5 中创建的推理引擎，单击 **创建**，等待模型部署状态变为就绪。

3. **部署就绪后，确认该服务已注册向量接口**，再进行下一步。进入模型部署的 **详情** 页面，进入实例的 **终端**，执行命令比如：

```
curl -s http://127.0.0.1:8000/openapi.json | grep -o '/v1/embeddings'
```

返回 `"/v1/embeddings"` 表示接口已注册。如果没有返回，检查模型是否支持向量任务以及推理引擎的启动日志。

7. 创建 AI 服务提供者。

1. 进入《AI 服务提供者》，单击 **创建**，服务类型选择「内部」。

2. **先选择「API 规范」为 OpenAI**，再选择步骤 6 中的模型部署，向量接口使用 OpenAI 规范。

3. 单击 **创建**。

8. 创建模型网关路由。

1. 进入《模型网关》，单击 **创建路由**。

2. 输入路由域名，后端选择步骤 7 中创建的 OpenAI 规范 AI 服务提供者，单击 **创建**。

3. 如果为同一个模型配置了多个后端（主用 + 备用），**这些后端必须全部是 OpenAI 规范**。同一模型下混用不同规范的后端，在故障转移时请求会失败。不同模型之间可以使用不同规范，例如在同一条路由下同时挂载 OpenAI 规范的向量模型和 Cohere 规范的重排序模型。

9. 创建套餐。

进入《套餐管理》，单击 **创建套餐**，「可访问路由」选择步骤 8 中创建的路由，按业务需要设置调用频率和 Token 限额后创建。

10. 创建 API Key。

进入《API Key》，单击 **创建**，选择步骤 9 中创建的套餐，单击 **创建并签发密钥**。签发成功后复制密钥妥善保存，**密钥只在此时完整展示一次**。

11. 查看 Token 用量。

进入《Token 用量》，在「请求类型」筛选器中选择「embeddings」，即可查看向量请求的调用次数和 Token 消耗。

结果验证

1. 确认域名可解析到网关地址。路由是按域名匹配的，域名未解析时请求不会到达网关。可以配置 hosts 记录，也可以在 curl 命令中使用 `--resolve` 参数临时指定，后者不会修改本机配置。
2. 执行向量调用，确认返回结果包含与输入文本一一对应的向量。具体命令如下：

```
curl -s http://<路由域名>:<网关端口>/v1/embeddings \
-X POST \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer ' \
-d '{
  "model": "<模型名称>",
  "input": [
    "EasyStack MaaS 是一个模型即服务平台",
    "今天天气不错"
  ],
  "encoding_format": "float"
}'
```

正常返回的 `data` 数组包含两个元素；每个元素的 `embedding` 是一个数值数组，`index` 与输入文本的下标对应。响应还应包含 `usage.prompt_tokens`。使用 `BAAI/bge-m3` 时，稠密向量的维度应为 1024。请求体的主要字段如下：

字段	是否必填	说明
<code>model</code>	是	路由中配置的模型名称
<code>input</code>	是	待转换的文本或文本数组
<code>encoding_format</code>	否	设置为 <code>float</code> 时返回数值数组，便于检查向量维度

3. 确认访问路径正确。网关注册的路径是 `/v1/embeddings`，**必须完全一致**。访问其他路径可能返回 404，提示路径不受支持。
4. 确认限流与计量生效。
 1. 连续调用超过套餐设置的频率上限，确认返回 429。
 2. 回到《Token 用量》，确认出现请求类型为 embeddings 的记录，且 Token 数量与请求内容规模相符。

约束与注意事项

- **规范互斥**：向量模型应通过 OpenAI 规范提供 `/v1/embeddings`，用 Cohere 规范的提供者去调用向量接口会失败。
- **规范不可变更**：AI 服务提供者的 API 规范在创建后无法修改。若某个提供者已被模型网关路由引用，需要先处理路由引用。
- **同一模型的后端规范必须一致**：主用和备用后端必须是同一规范，否则故障转移时请求会失败。不同模型之间不受此限制。
- **模型体验**：当前《模型体验》页面没有向量模式，不能通过对话功能验证纯向量模型；请使用本文的接口调用命令。
- **向量一致性**：同一知识库中的查询和文档应使用同一模型及版本生成向量。更换模型或维度后，需要重新生成文档向量。

1.4 LoRA模型部署和访问

背景描述

LoRA 可以在共享基础模型权重的前提下，为同一个模型部署加载一个或多个微调模型。业务请求通过 `model` 字段选择基础模型或指定的 LoRA 微调模型，无需为每个 LoRA 单独创建一套完整模型部署。

本文以基础模型 `Qwen2.5-0.5B-Instruct` 和 LoRA 微调模型 `Qwen-2.5-0.5b-instrunt-lora-J` 为例，介绍如何在创建模型部署时，加载 LoRA 微调模型，并通过 OpenAI 兼容接口验证基础模型和 LoRA 微调模型。

前提条件

- 已在 Beacon 中准备好基础模型、LoRA 微调模型、vLLM 推理引擎、算力和项目配额，并具有创建模型部署的权限。
- LoRA 微调模型必须基于所选基础模型训练。模型架构、权重版本和 Tokenizer 应与基础模型兼容。

操作步骤

1. 准备基础模型。

- 进入《模型广场》，确认基础模型及其模型文件已经准备完成。
- 本文使用以下基础模型：

配置项	示例值
基础模型	Qwen2.5-0.5B-Instruct

2. 准备 LoRA 微调模型。

- 进入《模型仓库》，上传或导入 LoRA 微调模型文件。
- 上传模型文件时勾选 `LoRA 微调模型`；如果模型文件已经上传，可以编辑模型文件并补充该标记。
- 如果模型在 ModelScope 有 lora 标记，导入时自动打上 `LoRA 微调模型` 标签。

4. 等待模型文件状态正常且发布完成。本文使用的 LoRA 模型文件为 `Qwen-2.5-0.5b-instrunt-lora-J`。

3. 创建vLLM 推理引擎。

4. 创建模型部署。

1. 进入《模型广场》，选择 `Qwen2.5-0.5B-Instruct`，单击 `部署`。
2. 选择步骤 3 中准备的 vLLM 推理引擎
3. 选择基础模型文件，`SERVED_MODEL_NAME` 可配置基础模型名称比如 `Qwen2.5-0.5B-Instruct`。
4. 在 `LoRA 微调模型` 中单击 `添加 LoRA 微调模型`。示例：

配置项	示例值	说明
模型名称	<code>qwen-hi</code>	调用接口时填写到请求体 <code>model</code> 字段中的名称
LoRA 微调模型	<code>Qwen-2.5-0.5b-instrunt-lora-J</code>	已发布的 LoRA 模型文件

5. 如果需要加载多个 LoRA 微调模型，继续单击 `添加 LoRA 微调模型`，并为每个 LoRA 设置不同的模型名称。
6. 单击 `创建`，等待模型部署状态变为就绪。

`模型名称` 是该 LoRA 对外提供服务时使用的模型标识。它可以与模型文件名称不同，但应使用稳定、易识别的英文名称，不要使用中文，也不要与基础模型名称或同一部署中的其他 LoRA 模型名称重复。

5. 检查模型部署。

1. 进入《模型部署》详情页，确认基础模型文件和基础模型名称正确。
2. 确认 `LoRA 微调模型` 区域中显示了 LoRA 模型文件、模型地址和模型名称 `qwen-hi`。
3. 进入模型部署的 `详情` 页面，进入实例的 `终端`，执行命令：

```
curl -s http://127.0.0.1:8000/v1/models
```

4. 返回结果中应同时包含基础模型名称和 LoRA 模型名称。例如：

```
Qwen2.5-0.5B-Instruct
qwen-hi
```

6. 创建 AI 服务提供者。

1. 进入《AI 服务提供者》，单击 **创建**，服务类型选择 **内部**。
2. API 规范选择 **OpenAI**，模型部署选择步骤 4 中创建的部署。
3. 确认该提供者可以识别 **Qwen2.5-0.5B-Instruct** 和 **qwen-hi**，然后单击 **创建**。

7. 创建模型网关路由。

1. 进入《模型网关》，单击 **创建路由**。
2. 添加 LoRA 模型 **qwen-hi**，后端选择步骤 6 中创建的 AI 服务提供者，后端模型选择 **qwen-hi**。
3. 如果还需要通过同一域名调用基础模型，可以继续添加模型 **Qwen2.5-0.5B-Instruct**，并将后端模型选择为 **Qwen2.5-0.5B-Instruct**。
4. 输入路由域名并完成创建。

8. 创建套餐和 API Key。

1. 进入《套餐管理》，创建套餐并选择步骤 7 中创建的路由。
2. 进入《API Key》，选择该套餐并签发密钥。
3. 复制并妥善保存 API Key，密钥只在签发成功时完整展示一次。

结果验证

1. 确认模型部署状态为 **就绪**，部署详情中同时显示基础模型和 LoRA 微调模型 **qwen-hi**。
2. 调用 LoRA 微调模型。请求体中的 **model** 必须填写设置的 LoRA 模型名称：

```
curl -s http://<路由域名>:<网关端口>/v1/chat/completions \
-X POST \
-H 'Content-Type: application/json' \
-H 'Authorization: Bearer ' \
-d '{
  "model": "qwen-hi",
  "messages": [
    {
```

```
        "role": "user",
        "content": "你好，请用一句话介绍自己。"
    }
],
"temperature": 0,
"max_tokens": 128
}'
```

3. 确认接口返回成功，响应中的 `model` 为 `qwen-hi`，并且包含正常生成的内容。
4. 使用相同请求调用基础模型，只需把 `model` 改为基础模型名称：

```
{
  "model": "Qwen2.5-0.5B-Instruct",
  "messages": [
    {
      "role": "user",
      "content": "你好，请用一句话介绍自己。"
    }
  ],
  "temperature": 0,
  "max_tokens": 128
}
```

5. 使用 LoRA 验证问答分别调用基础模型和 `qwen-hi`。固定提示词、`temperature`、`max_tokens` 等请求参数，确认 LoRA 模型的回答符合微调目标。普通问候的回答可能相近，不能单独证明微调效果。

约束与注意事项

- **基础模型必须匹配：**LoRA 微调模型只能与其训练时使用的基础模型配合使用。名称相似不代表模型权重和 Tokenizer 一定兼容。
- **基础模型仍可调用：**加载 LoRA 后，同一部署仍可通过基础模型名称调用原始基础模型。调用 LoRA 时必须显式填写对应的 LoRA 模型名称。
- **使用 vLLM 引擎：**LoRA 微调模型配置仅用于 vLLM 推理引擎；当前不支持 SGLang，vLLM-Omni 引擎。
- **多个 LoRA 名称必须唯一：**同一部署中可以添加多个 LoRA 模型，但每个模型名称必须唯一。模型网关中配置的后端模型名称应与这里保持一致。
- **修改会触发更新：**添加、删除或替换 LoRA 微调模型后，需要等待模型部署重新变为就绪，再执行调用验证。

1.5 vLLM模型启动参数调优方式

背景描述

本文以 Beacon 的《基准测试》为例，介绍如何为指定模型建立延迟和吞吐基线，依次调整 vLLM 启动参数，并选择满足业务延迟要求、运行稳定且吞吐较高的参数组合。

延迟测试固定使用 1 并发，用于比较模型在低负载下的响应速度；吞吐测试可以调整并发数，用于比较模型在不同负载下的吞吐和延迟。

前提条件

- 已在 Beacon 中准备好模型、vLLM 推理引擎、算力和项目配额，并具有创建模型部署及基准测试的权限。
- 测试对象须为单副本且状态正常的 vLLM 模型部署。
- 调优期间保持模型、vLLM 版本、镜像、加速卡、CPU 和内存规格一致。
- 同一部署上的基准测试应依次执行，避免测试任务相互影响。

操作步骤

1. 确定模型和调优目标。

先确定要调优的模型及业务目标，例如“TTFT P95 不超过 1 秒、测试期间没有错误，在此前提下尽量提高输出 Token/s”。示例阈值仅作说明，实际值由业务确定。

2. 创建基线模型部署。

1. 在《推理引擎》中创建或选择 vLLM 引擎。
2. 在《模型广场》中部署要调优的模型，设置副本数为 1，等待部署状态正常。
3. 记录模型版本、vLLM 版本、镜像、算力规格和启动参数。

3. 建立延迟基线。

进入《基准测试》，单击 **创建测试**，选择基线部署和 **延迟** 测试，然后开始测试。延迟测试固定使用 1 并发，不需要设置并发数。

测试完成后，记录测试 ID、TTFT P95、TPOT P95、请求延迟 P95，以及成功、错误。后续每组候选参数都应运行相同的延迟测试，并与基线比较。

4. 建立吞吐基线。

保持基线部署不变，创建多个 **吞吐** 测试，每次设置不同的并发数。建议从较低并发开始，逐步增加：

1 → 2 → 4 → 8 → 16 → 32 → 64

如果 64 并发时吞吐仍在提高、延迟仍满足要求，并且没有错误，可继续测试 **128 → 256 → 512**。512 是可配置的最大并发数，不代表模型部署能够稳定承载 512 并发。

每个并发点记录以下结果：

测试并发	实际并发均值 / 最大值	输出 Token/s	请求/s	TTFT P95	
1	待记录	待记录	待记录	待记录	
2	待记录	待记录	待记录	待记录	
4	待记录	待记录	待记录	待记录	
8	待记录	待记录	待记录	待记录	
16	待记录	待记录	待记录	待记录	
32	待记录	待记录	待记录	待记录	
64	待记录	待记录	待记录	待记录	

吞吐不再明显提高，或者延迟快速增加时，说明部署已接近吞吐拐点。如果出现错误或延迟超过业务要求，则该并发点不作为稳定承载结果。

5. 制定候选启动参数，每次只调整一项，比如

参数	调整目的	必须检查
----	------	------

参数	调整目的	必须检查
<code>--max-model-len</code>	设置最大上下文长度	是否覆盖业务输入和输出长度
<code>--gpu-memory-utilization</code>	调整 vLLM 可使用的显存比例	部署能否正常启动，运行时是否显存不足
<code>--max-num-seqs</code>	调整可同时处理的序列数量	吞吐是否提高，延迟是否符合要求
<code>--max-num-batched-tokens</code>	调整批处理的 Token 数量	吞吐和 TTFT 是否符合要求

每个部署只调整一个参数，其余配置与基线保持一致，以便判断指标变化来自哪个参数。

6. 调整模型部署的启动参数，点击 [编辑](#) 原模型部署。

1. 在原启动参数基础上修改一项参数，不要删除模型服务所需的基础参数。
2. 使用容易识别的部署名称，例如 `模型名-seqs32`，设置副本数为 1，并等待部署状态正常。

7. 复测并比较候选参数。

每组候选先运行固定 1 并发的延迟测试，再运行吞吐测试。不同部署必须在相同测试类型和相同吞吐并发下比较。

部署	参数变更	测试并发	输出 Token/s	TTFT P95	
baseline	无	32	待记录	待记录	
seqs32	<code>--max-num-seqs 32</code>	32	待记录	待记录	

建议：

如果要进一步提升吞吐，可以部署多实例；如果要进一步提升延迟性能，可以部署多卡。

结果验证

1. 确认延迟和吞吐测试均为 **已完成**，错误请求数为 0。
2. 延迟测试固定使用 1 并发，将候选参数的 TTFT P95、TPOT P95 和请求延迟 P95 与基线比较。
3. 吞吐测试应核对实际并发均值和最大值，并在相同测试并发下比较基线与候选参数的输出 Token/s、请求/s 和延迟。
4. 选择吞吐拐点附近、延迟满足业务要求且没有错误请求的并发数作为推荐并发。
5. 按以下格式记录最终结果，并保留测试 ID 以便复核：

```
模型：<模型名称和版本>
运行环境：<vLLM 版本和镜像>; <加速卡型号 × 数量>; 1 副本
启动参数：<最终启动参数或相对默认值的变更>
延迟测试：TTFT P95 <值> ms; TPOT P95 <值> ms; 请求延迟 P95 <值> ms
推荐并发：<并发数>
吞吐测试：输出 <值> Token/s; 请求 <值>/s; 错误 <数量>
测试任务：延迟测试 <任务 ID>; 吞吐测试 <任务 ID>
最终结论：<是否达到业务目标以及采用该参数的原因>
```

约束与注意事项

- **单部署测试**：基准测试用于比较单个模型部署，不能直接代表模型网关或多副本场景的整体性能。
- **测试类型**：延迟测试固定使用 1 并发；吞吐测试通过调整并发数比较不同负载下的性能，两类结果不能相互替代。
- **并发设置**：调整吞吐测试的并发数只会改变测试负载，不会修改 vLLM 启动参数。
- **结果可比性**：比较时保持模型、vLLM 版本、镜像和算力规格一致，并使用相同的测试类型和吞吐并发数。
- **重新建立基线**：修改模型、vLLM 版本、镜像或算力规格后，需要重新运行延迟和吞吐基线测试。

咨询热线：400-100-3070

北京易捷思达科技发展有限公司：

北京市海淀区西北旺东路10号院东区23号楼华胜天成科研大楼一层东侧120-123

南京分公司：

江苏省南京市雨花台区软件大道168号润和创智中心B栋一楼西101

上海office：

上海黄浦区西藏中路336号华旭大厦22楼2204

成都分公司：

成都市高新区天府五街168号德必天府五街WE602

邮箱：

contact@easystack.cn (业务咨询)

partners@easystack.cn(合作伙伴咨询)

marketing@easystack.cn (市场合作)

training@easystack.cn (培训咨询)

hr@easystack.cn (招聘咨询)