

Agent Runtime

Agent 集群



腾讯云

【 版权声明 】

©2013–2026 腾讯云版权所有

本文档（含所有文字、数据、图片等内容）完整的著作权归腾讯云计算（北京）有限责任公司单独所有，未经腾讯云事先明确书面许可，任何主体不得以任何形式复制、修改、使用、抄袭、传播本文档全部或部分内容。前述行为构成对腾讯云著作权的侵犯，腾讯云将依法采取措施追究法律责任。

【 商标声明 】



及其它腾讯云服务相关的商标均为腾讯云计算（北京）有限责任公司及其关联公司所有。本文档涉及的第三方主体的商标，依法由权利人所有。未经腾讯云及有关权利人书面许可，任何主体不得以任何方式对前述商标进行使用、复制、修改、传播、抄录等行为，否则将构成对腾讯云及有关权利人商标权的侵犯，腾讯云将依法采取措施追究法律责任。

【 服务声明 】

本文档意在向您介绍腾讯云全部或部分产品、服务的当时的相关概况，部分产品、服务的内容可能不时有所调整。您所购买的腾讯云产品、服务的种类、服务标准等应由您与腾讯云之间的商业合同约定，除非双方另有约定，否则，腾讯云对本文档内容不做任何明示或默示的承诺或保证。

【 联系我们 】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100或 95716。

文档目录

Agent 集群

产品介绍

什么是 Agent 集群

适用场景

核心概念

AGC、AGS 与 TKE 的关系

资源与费用说明

快速入门

通过 Kubernetes API 运行第一个工作负载

在 Agent 集群上使用 AgentWay 创建办公助手

批量启动沙箱运行 RL Rollout

常见问题与故障排查

常见问题

Agent 集群

产品介绍

什么是 Agent 集群

最近更新时间：2026-10-08 17:40:00

Agent 集群（Agent Cluster，简称 AGC）是腾讯云 Agent Runtime 面向企业提供的 Agent 运行基础设施。AGC 基于腾讯云 TKE，在企业账号内提供运行 Agent 任务所需的计算、隔离环境、网络和存储能力，适合建设 Coding Agent、代码执行平台和企业 Agent 平台。

解决什么问题

Agent 执行任务时，往往需要运行代码、安装依赖、读写文件和访问企业系统。随着用户和任务数量增加，企业需要同时解决多个任务的环境隔离、计算资源分配、工作文件保留和故障排查问题。

AGC 将这些运行需求集中到集群中管理，让企业能够为不同任务配置运行环境和资源，将 Agent 接入自己的业务网络，并保留任务产生的工作数据。

主要功能

功能	说明
集群与算力管理	创建 Agent 集群，通过 TKE 节点池配置计算资源，管理任务可使用的运行容量。
隔离运行环境	为代码执行、构建和工具调用提供隔离环境，配置任务所需的镜像、CPU 和内存。
Kubernetes 工作负载接入	使用 Kubernetes 方式提交和管理工作负载，与企业已有的开发和运维工具结合。
工作文件持久化	通过持久化存储保留代码、依赖和任务输出，供后续任务或重建后的运行环境继续使用。
企业网络与访问控制	结合 VPC、云产品权限和 Kubernetes 权限，管理任务对企业系统及集群资源的访问。
状态查看与故障排查	查看集群、节点和任务状态，通过日志与事件定位调度、镜像拉取和执行异常。

① 说明：

Agent 集群目前处于内测阶段，欢迎 [申请内测](#)。

内测期间仅支持新建 TKE 集群，存量 TKE 集群接入能力正在开发中。

适用场景

最近更新时间：2026-10-08 17:40:00

代码助手与代码执行平台

代码助手需要反复安装依赖、修改文件、执行测试。多个用户共用一个执行环境时，依赖版本和文件容易相互影响；每次重新准备环境，又会增加等待时间。

通过 Agent 集群，为任务分配独立运行环境，将代码、依赖和结果保存在工作区中。任务重建后继续使用原有文件，平台集中管理运行资源和任务结果。

从 [快速入门](#) 开始，可以运行一次文本处理任务，测量启动耗时，再将工作区迁移到另一个节点继续执行。

访问企业内网的助手

企业助手需要调用内网工具、读取业务资料，并将处理结果写回企业系统。运行环境需要接入企业网络，同时限制可访问的服务和使用的身份。

Agent 集群在企业账号内使用 TKE 和 VPC 资源。企业可以为任务配置网络访问和凭证，让助手在所需的权限范围内执行操作。

批量工具调用与长时任务

文档处理、代码检查和数据加工可能同时产生大量任务，且不同任务的执行时长不同。长期保留全部运行环境会占用资源；直接删除环境又可能丢失中间结果。

通过 Agent 集群，将运行环境与需要保留的文件分别管理。执行结束后释放计算资源，将输出归档；需要重试的任务从已保存的业务进度继续，减少重复处理。

核心概念

最近更新时间：2026-10-08 17:40:00

概念	含义	示例
Agent 集群	承载 Agent 工作负载的运行基础设施，管理运行所需的计算、网络和存储资源。	为企业代码助手准备一个集群
节点池	使用相同配置的一组计算节点，用于提供运行容量。	为代码执行任务分配一组节点
运行环境	执行任务代码的独立环境。	一个运行代码检查脚本的 Pod 或沙箱
工作负载	提交到集群执行的程序及其资源配置。	一个有 CPU、内存限制的代码分析任务
工作区 (workspace)	保存代码、输入文件和运行结果的目录。	保存仓库代码和测试输出的 /workspace
镜像与模板	创建运行环境时使用的程序、依赖和初始配置。	包含 Python 和常用库的执行环境

一个 Agent 集群包含提供计算容量的节点池。任务在运行环境中执行，读取和修改工作区中的文件。运行环境释放后，需要保留的文件放在持久存储中，供后续任务继续使用。

使用 Kubernetes 时，通过 Pod 声明工作负载，通过 RuntimeClass 选择运行时，通过 PVC 引用持久存储。

使用 E2B 时，通过 SDK 创建沙箱、执行命令和读写文件。

AGC、AGS 与 TKE 的关系

最近更新时间：2026-10-08 17:40:01

AGC 与 AGS 面向不同的基础设施管理需求。AGS 对应全托管形态；AGC 面向希望在自身云资源和已有基础设施体系中运行 Agent 的团队。选择依据是资源管理方式、网络与存储需求，以及团队愿意承担的运维工作。

维度	选择时要考虑什么
希望减少基础设施管理	评估全托管形态是否覆盖业务需要
已有 Kubernetes 或 Infra 团队	评估 AGC 与现有发布、网络、存储的结合方式
需要掌握算力与容量	明确节点配置、配额、资源预算和运维职责
希望复用现有 TKE	内测期间暂不支持，存量 TKE 集群接入能力正在加速开发中

TKE 是腾讯云的 Kubernetes 容器服务，适用于通用容器应用。AGC 基于 TKE，面向 Agent 的代码执行、工具调用和工作文件管理等运行需求，提供相应的运行环境与管理能力。

希望快速使用托管运行环境的团队，可以选择 AGS；需要结合自身云资源、网络和存储管理 Agent 运行环境的团队，可以选择 AGC。

资源与费用说明

最近更新时间：2026-10-08 17:40:01

AGC 服务费

内测期间，Agent 集群不收取服务费。

正式商用（GA）后，Agent 集群将按纳管的 CPU 和内存规模收取服务费。具体计费标准将在正式商用时公布。

云资源费用

AGC 服务费与计算、存储、网络等云资源费用分别计算。内测期间免收 AGC 服务费，不包含云资源费用；相关资源按对应云产品的计费规则收费。

测试结束后，请及时清理不再使用的节点、存储和网络资源，避免持续产生费用。

快速入门

通过 Kubernetes API 运行第一个工作负载

最近更新时间：2026-10-08 17:40:01

如果您正在开发代码助手或数据分析 Agent，通常需要为每个任务准备独立的执行环境，使其能够接收用户文件、运行脚本、保存结果，并在后续任务中继续使用这些文件。如果您的应用已使用 Kubernetes SDK、kubectl 或工作流编排器，可以通过 Kubernetes API 完成这些操作。

本文以“数据分析 Agent 汇总订单文件”为例，完成一次可复现的任务：

1. 创建 Pod 作为脚本执行环境，挂载工作区保存订单和结果。
2. 统计已支付订单数量与金额，记录环境启动耗时。
3. 结束原 Pod，在另一节点创建新 Pod，接着处理原来的文件。

本例提供固定数据和脚本，无需配置模型或调用外部业务系统。实际接入 Agent 时，可将这些操作分别放到应用的“创建环境、上传文件、执行代码、读取结果、释放环境”流程中。kubectl 在这里作为 Kubernetes API 客户端使用。

开始前准备

您需要一个已创建的 Agent 集群，以及可访问集群的 kubeconfig。本机安装 kubectl 和 Python 3；操作账号需要查询节点、RuntimeClass 和 StorageClass，创建及删除测试命名空间，管理其中的 Pod/PVC，以及执行 `pods/exec` 的权限。

先配置访问，并查看运行环境：

```
export KUBECONFIG='/absolute/path/to/your/agc-kubeconfig'
kubectl config current-context
kubectl get nodes
kubectl get runtimeclass
kubectl get storageclass
```

本例使用以下配置。将值替换为自己集群中的名称，并在同一个终端依次执行后续命令：

```
export AGC_DEMO_NS="agc-orders-$(date +%Y%m%d%H%M%S)"
export AGC_RUNTIME_CLASS='cube'
export AGC_STORAGE_CLASS='awv-btrfs'
export AGC_DEMO_IMAGE='busybox:1.36.1'
kubectl create namespace "$AGC_DEMO_NS"
```

配置	如何选择
----	------

RuntimeClass	选择 Agent 集群中用于运行任务的运行时；本例填写 <code>cube</code> 。
StorageClass	选择支持跨节点重新挂载、 <code>ReadWriteOncePod</code> 和 <code>fsGroup: 1000</code> 的工作区存储类；本例填写 <code>awv-btrfs</code> 。
执行镜像	本例使用 BusyBox 的 Shell、awk 和校验工具；节点需要能够拉取此镜像，也可替换为包含相同工具的企业镜像。
节点	先有一个可调度且支持所选运行时和存储的 Ready 节点；跨节点继续任务还需要第二个这样的节点。

每次实验使用独立命名空间，避免与已有任务重名。下面的 Pod 只运行本例脚本，工作文件保存在 PVC 挂载的 `workspace` 中。

第一步：准备工作区和执行环境

将以下命令复制到终端，生成工作区配置：

```
cat > orders-pvc.yaml <<EOF
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: orders-workspace
  namespace: ${AGC_DEMO_NS}
spec:
  storageClassName: ${AGC_STORAGE_CLASS}
  volumeMode: Filesystem
  accessModes: [ReadWriteOncePod]
  resources:
    requests:
      storage: 1Gi
EOF
```

生成 Pod 配置。Pod 启动后保持运行，便于应用通过 `exec` 提交任务：

```
cat > orders-pod.yaml <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: orders-runner
```

```
namespace: ${AGC_DEMO_NS}
spec:
  runtimeClassName: ${AGC_RUNTIME_CLASS}
  restartPolicy: Never
  automountServiceAccountToken: false
  securityContext:
    fsGroup: 1000
  containers:
    - name: runner
      image: ${AGC_DEMO_IMAGE}
      imagePullPolicy: IfNotPresent
      command: ["/bin/sh", "-c", "sleep 86400"]
      securityContext:
        runAsNonRoot: true
        runAsUser: 1000
        runAsGroup: 1000
        allowPrivilegeEscalation: false
        capabilities:
          drop: ["ALL"]
        seccompProfile:
          type: RuntimeDefault
      resources:
        requests:
          cpu: 100m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 128Mi
      volumeMounts:
        - name: workspace
          mountPath: /workspace
  volumes:
    - name: workspace
      persistentVolumeClaim:
        claimName: orders-workspace
EOF
kubectl create --dry-run=server -f orders-pvc.yaml
kubectl create --dry-run=server -f orders-pod.yaml
kubectl create -f orders-pvc.yaml
```

使用 `WaitForFirstConsumer` 的存储类时，PVC 此时显示 Pending 是正常的；创建 Pod 后才会开始绑定。请勿在创建 Pod 前一直等待 PVC Bound。

第二步：启动环境并记录等待时间

对业务应用而言，需要等待的是“可以执行代码且工作区可以写入”。将下面脚本保存为 `measure-start.py`，测量从本机发起创建到确认工作区可写的耗时：

```
import json
import os
import subprocess
import sys
import time

namespace = os.environ['AGC_DEMO_NS']

def kubectl(*args):
    return subprocess.check_output(
        ['kubectl', '-n', namespace, *args], text=True, timeout=210
    )

started = time.perf_counter()
created = json.loads(kubectl('create', '-f', sys.argv[1], '-o', 'json'))
name = created['metadata']['name']
print('pod:', name, file=sys.stderr, flush=True)
kubectl('wait', '--for=condition=Ready', 'pod/' + name, '--
timeout=180s')
kubectl('exec', name, '-c', 'runner', '--', '/bin/sh', '-ec',
        'echo ready > /workspace/.startup-check; cat
/workspace/.startup-check; '
        'rm /workspace/.startup-check')
elapsed = time.perf_counter() - started
pod = json.loads(kubectl('get', 'pod', name, '-o', 'json'))
container = next(c for c in pod['status']['containerStatuses'] if
c['name'] == 'runner')
print(json.dumps({
    'namespace': namespace,
    'pod': name,
    'node': pod['spec']['nodeName'],
    'workspace_ready_seconds': round(elapsed, 3),
```

```
'image_id': container.get('imageID'),
'restart_count': container['restartCount'],
}, ensure_ascii=False, indent=2))
```

执行脚本：

```
python3 measure-start.py orders-pod.yaml > startup-a.json
cat startup-a.json
```

成功时会输出 Pod 名称、所在节点、`workspace_ready_seconds` 和镜像信息。该耗时包含客户端请求、调度、镜像准备、工作区挂载以及 `exec` 检查；它不是单独的容器启动耗时，也不包含后面的订单统计时间。若 180 秒内未就绪，脚本会失败并保留 Pod，请参见文末排查方法查看原因，请勿重复创建同名资源。

第三步：上传订单并生成统计结果

下面模拟 Agent 接收一份订单文件，并将准备好的分析脚本放到工作区。只有 `paid` 订单计入统计，金额单位为元：

```
kubectl -n "$AGC_DEMO_NS" exec -i orders-runner -c runner -- /bin/sh -s
<<'SCRIPT'
set -eu
cat > /workspace/orders.csv <<'CSV'
order_id,status,amount
A001,paid,100
A002,paid,50
A003,cancelled,80
CSV
cat > /workspace/summarize.sh <<'SH'
#!/bin/sh
set -eu
awk -F, 'NR > 1 && $2 == "paid" { count++; amount += $3 }
END { printf "paid_orders=%d\npaid_amount=%d\n", count, amount }' \
/workspace/orders.csv > /workspace/summary.txt
SH
/bin/sh /workspace/summarize.sh
sync
cat /workspace/summary.txt
SCRIPT
```

预期输出：

```
paid_orders=2
paid_amount=150
```

将本次结果和文件校验值保存到本机：

```
kubectl -n "$AGC_DEMO_NS" exec orders-runner -c runner -- cat
/workspace/summary.txt > summary-a.txt
kubectl -n "$AGC_DEMO_NS" exec orders-runner -c runner -- \
    sha256sum /workspace/orders.csv /workspace/summarize.sh
/workspace/summary.txt > files-a.sha256 || { echo "读取校验值失败，停止实验"
>&2; exit 1; }
test -s files-a.sha256 || { echo "未取得文件校验值，停止实验" >&2; exit 1; }
cat summary-a.txt
```

至此，您已经完成一个可用的最小任务：创建环境、提交文件和脚本、执行分析、取回结果。如果只需体验第一次执行，可以直接跳到清理步骤。

第四步：换一个节点继续任务

假设上一轮分析结束后，您释放执行环境；后来用户追加订单，需要再次创建环境继续处理原文件。下面主动选择另一个节点，验证工作区能否随任务重新挂载。

先查看当前 Pod 的节点，以及可用节点的 hostname 标签：

```
kubectl -n "$AGC_DEMO_NS" get pod orders-runner -o wide
kubectl get nodes -L kubernetes.io/hostname
kubectl get runtimeclass "$AGC_RUNTIME_CLASS" -o yaml
```

选择一个不同于当前节点、状态 Ready 且未标记 SchedulingDisabled 的节点。目标节点必须满足所选 RuntimeClass 的调度条件，并在工作区存储的挂载范围内。将它的 `kubernetes.io/hostname` 标签值填入：

```
export AGC_TARGET_HOSTNAME='REPLACE_WITH_NODE_B_HOSTNAME'
python3 - <<'PY'
import json
import os
import subprocess
pod = json.loads(subprocess.check_output(
    ['kubectl', 'create', '--dry-run=client', '-f', 'orders-pod.yaml',
    '-o', 'json'], text=True))
```

```
pod['metadata']['name'] = 'orders-runner-b'
pod['spec']['nodeSelector'] = {'kubernetes.io/hostname':
os.environ['AGC_TARGET_HOSTNAME']}
with open('orders-pod-b.json', 'w') as f:
    json.dump(pod, f, indent=2)
PY
kubectl create --dry-run=server -f orders-pod-b.json
```

仅释放原 Pod，保留 `orders-workspace` PVC，然后启动新 Pod：

```
kubectl -n "$AGC_DEMO_NS" delete pod orders-runner --wait=true --
timeout=180s
kubectl -n "$AGC_DEMO_NS" get pvc orders-workspace
python3 measure-start.py orders-pod-b.json > startup-b.json
cat startup-b.json
```

这里是“停止原环境，再挂载工作区启动新环境”，不是正在执行的进程无中断迁移。`ReadWriteOncePod` 限定同一时刻由一个 Pod 使用此工作区，因此需要先结束原 Pod。

第五步：校验原文件并处理追加订单

新 Pod 不重新上传文件，先检查原来的数据、脚本和结果是否一致：

```
kubectl -n "$AGC_DEMO_NS" exec orders-runner-b -c runner -- \
    sha256sum /workspace/orders.csv /workspace/summarize.sh
/workspace/summary.txt > files-b.sha256 || { echo "读取校验值失败，停止实验"
>&2; exit 1; }
test -s files-a.sha256 && test -s files-b.sha256 && diff files-a.sha256
files-b.sha256 || { echo "文件校验失败，停止实验" >&2; exit 1; }
```

`diff` 无输出且退出码为 0，说明三个文件的校验值一致。如果不一致，停止后续写入并检查工作区。校验通过后，追加一条 80 元的已支付订单，再执行原脚本：

```
kubectl -n "$AGC_DEMO_NS" exec orders-runner-b -c runner -- /bin/sh -ec
'
printf "%s\n" "A004,paid,80" >> /workspace/orders.csv
/bin/sh /workspace/summarize.sh
sync
cat /workspace/summary.txt
```

```
' > summary-b.txt
cat summary-b.txt
```

预期输出为 `paid_orders=3`、`paid_amount=230`。这说明新环境能够继续读取、修改原文件并执行脚本，不需要重新上传上一轮的工作内容。追加订单这一步只执行一次，重复执行会重复计入该订单。

核对本次体验结果

核对项	预期结果
首轮分析	2 笔已支付订单，合计 150 元，已取消订单不计入。
节点变化	<code>startup-a.json</code> 与 <code>startup-b.json</code> 中的 <code>node</code> 不同。
文件保留	三个文件迁移前后的 SHA-256 一致。
继续分析	追加订单后为 3 笔，合计 230 元。
环境等待时间	两份记录均有 <code>workspace_ready_seconds</code> ，容器 <code>restart_count</code> 为 0。

两次耗时只反映这次体验。首次拉取镜像、节点缓存、卷创建和重新挂载都会影响结果，不能直接将两次耗时差解释为加速比例。需要评估并发或性能时，固定镜像、资源规格及缓存条件，单独设计重复采样实验。

清理测试资源

先保留本机的统计结果、启动记录和校验文件，再删除本次测试 Pod：

```
kubectl -n "$AGC_DEMO_NS" delete pod orders-runner orders-runner-b --
ignore-not-found --wait=true --timeout=180s
```

如果后续还要使用订单文件，保留 PVC。确认测试数据不再需要时，再删除 PVC 和本次独立命名空间：

```
kubectl -n "$AGC_DEMO_NS" delete pvc orders-workspace --wait=true --
timeout=180s
kubectl delete namespace "$AGC_DEMO_NS" --wait=true --timeout=180s
```

删除 PVC 后是否删除后端数据，取决于 StorageClass 的回收策略。使用 Retain 策略时，还需按存储管理流程处理保留卷。

遇到问题时

```
kubectl -n "$AGC_DEMO_NS" get pods,pvc -o wide
```

```
kubectl -n "$AGC_DEMO_NS" get events --sort-  
by=.metadata.creationTimestamp
```

现象	排查方法
Forbidden	核对 kubeconfig 身份及命名空间、Pod、PVC、pods/exec 权限。
Pod Pending	查看调度事件，检查 RuntimeClass 条件、目标节点、资源余量和 PVC 绑定状态。
ImagePullBackOff	检查节点到镜像仓库的网络及镜像权限；私有镜像需配置拉取凭证。
挂载失败或提示卷仍被占用	检查存储组件和原 Pod 的卸载情况，不强制解绑仍在使用的卷。
工作区 Permission denied	检查存储类的 fsGroup 支持及卷目录权限；本例使用 UID/GID 1000。
第二个节点始终无法启动	检查节点是否满足运行时条件、是否安装存储组件，以及卷是否支持在该节点挂载。
校验不一致或统计结果不符	先停止追加数据，核对 PVC 名称、挂载路径及订单追加步骤是否重复执行。

准备接入您自己的 Agent 应用时，可以将本例脚本替换为业务任务，沿用“每次任务创建执行环境、同一工作过程复用工作区、完成后回收计算资源”的流程。

在 Agent 集群上使用 AgentWay 创建办公助手

最近更新时间：2026-10-08 17:40:01

员工经常需要将零散的会议记录整理成纪要、提取待办，并根据后续反馈继续修改。使用 AgentWay，管理员可以准备统一的 Agent 模板，员工从模板创建自己的助手，通过浏览器完成上述工作。Agent 实例使用模板配置的运行环境，在 Agent 集群上运行。

本文以 OpenClaw 办公助手为例，从已有模板创建一个实例，完成“整理会议记录 → 核对待办 → 追加修改 → 保存结果”的完整过程。首次体验使用文中提供的虚构材料，不需要连接企业知识库、安装额外 Skill 或绑定聊天机器人。

完成后，你应能打开自己的助手、获得可核对的会议纪要，并在同一会话中继续处理任务。

开始前准备

在 AGC 控制台一键部署 AgentWay

AgentWay 支持通过 AGC 控制台一键部署，无需手动安装。由具备集群管理权限的管理员完成以下操作：

1. 登录 AGC 控制台，选择要运行办公助手的 Agent 集群，进入集群详情。
2. 进入集群详情页，查看访问方式，单击 **AgentWay 部署**，发起一键部署，按页面提示完成配置并提交。
3. 等待部署完成，打开该集群的 AgentWay 控制台入口。
4. 登录 AgentWay，确认可以进入 **Agent 实例** 页面，再准备下面的模板、模型和访问配置。

集群已部署 AgentWay 时，直接进入已有控制台，无需重复部署。首次使用 AgentWay 的员工可向管理员获取控制台地址和登录方式。

确认可以创建和使用助手

部署完成后，核对以下准备项：

准备项	完成条件
AgentWay 控制台地址与账号	能登录控制台并进入“Agent 实例”页面。
已发布的 OpenClaw 模板	创建实例时可以选到，已配置运行环境和可用模型。
Agent 实例访问入口	实例域名、证书及访问路由已配置，使用者所在网络可以访问。
创建权限与资源额度	账号允许创建实例，集群有模板所需的计算和存储资源，模型调用额度可用。

已有可用模板时，可以直接执行下面的步骤。看不到模板或尚未配置模型时，由管理员完成文末的“为团队准备模板”。

控制台地址用于管理实例；实例创建后生成的访问地址用于打开助手。两者用途不同，应分别确认可访问。

第一步：创建自己的办公助手

1. 登录 AgentWay 控制台，进入左侧的 **Agent 实例**。
2. 单击 **创建实例**，填写以下配置。

配置项	本例填写方式
实例名称	<code>meeting-assistant-demo</code> ，使用英文、数字和中划线；已有同名实例时换一个名称。
Agent 模板	选择管理员发布的 <code>OpenClaw</code> 模板，核对模板版本。
所有者	若页面显示此字段，选择实际使用者；管理员代建时不要把所有员工实例都归到管理员名下。
环境变量	本例无需额外填写，使用模板默认配置。

3. 单击 **创建**，回到实例列表，找到本次创建的实例。
 4. 记录实例 ID，等待状态变为 **运行中**，并检查是否出现 **访问地址**。
- 模板定义了镜像、启动行为、预设文件等配置，创建时平台会按模板准备实例。实例级环境变量会覆盖同名模板变量；首次体验不需要修改这些配置，也不要将模型密钥填写到任务输入中。
- 如果创建后暂时没有看到实例，可以刷新或按名称搜索，先确认已有创建结果，再决定是否重试，避免重复创建。

第二步：打开助手

在目标实例所在行单击 **访问地址**，打开 OpenClaw 的 Web 界面。

首次打开时，如果页面要求访问令牌，回到 AgentWay 的实例列表，在本次实例的 **Gateway Token** 旁单击复制。返回该实例的 OpenClaw 页面，将令牌粘贴到 **网关令牌**，单击 **连接**。连接成功后，页面显示网关在线，并出现聊天输入区。令牌属于实例访问凭证，只在确认无误的实例页面使用，不放入会议内容、共享截图或群消息中。

进入后，找到对话输入区，创建或进入一个会话。本例所有后续操作均在同一会话中进行。

此时检查两个结果：

- 实例访问地址能打开助手页面，而不是控制台首页、404 或网络错误页。
- 可以输入任务并接收回复，而不是仅显示“已连接”或空白聊天界面。

若浏览器提示域名无法解析，先处理实例域名与网络配置；重新创建实例通常不能解决域名问题。实例显示“运行中”也不能替代这一步检查。

第三步：整理一份会议纪要

将下面的虚构会议记录完整复制到助手的对话输入区并发送：

请作为我的办公助手，把下面的会议记录整理成一份中文会议纪要。

要求：

- 1. 分为“会议结论”“行动项”“待确认事项”三个部分。
- 2. 行动项使用表格，包含事项、负责人、截止时间。
- 3. 没有明确负责人或截止时间的内容，写“待确认”，不要自行补充。
- 4. 只依据下面的材料，不联网，也不要向任何人发送消息。

会议记录：

我们讨论了客服知识库试用安排。
决定先让客服一组试用，不扩大到其他团队。
林晨负责整理 20 条高频问题，周三 18:00 前完成。
周宁负责邀请 5 位客服参加试用，周四 12:00 前完成。
试用后要汇总反馈，但这次没有确定负责人和截止时间。
下一次评审时间还没有确定。

收到回复后，核对以下内容：

核对项	应有结果
试用范围	仅客服一组，不扩大到其他团队
高频问题整理	林晨，20 条，周三 18:00 前
邀请试用人员	周宁，5 位，周四 12:00 前
反馈汇总	负责人和截止时间均为“待确认”
下一次评审	时间待确认

文字表达可以不同，但负责人、数量、时间和未确定事项应与原文一致。模型可能遗漏或误解信息，会议纪要仍需人工核对后再用于工作。

第四步：在同一会话中继续修改

继续发送：

补充一项刚确认的信息：反馈汇总由林晨负责，截止时间为周五 18:00。
请只更新这一项，保留其余结论和行动项，输出完整的新版会议纪要。

检查新版纪要是否只更新反馈汇总事项，其他负责人、时间和试用范围保持不变。这个步骤模拟员工在收到补充信息后继续完善材料，验证助手能否利用当前会话上下文完成连续工作。

随后可以让助手生成另一种便于使用的结果：

基于刚才确认的最新版纪要，生成一段适合我复制到工作群的通知。
不超过 150 字，包含试用范围和三项行动安排。
只生成文本，不发送消息。

核对内容后，复制到你自己的工作文档。本例到生成和保存文本为止，不绑定 IM 通道，也不自动向工作群发送消息。

第五步：保存结果并决定是否保留实例

保存最终纪要和通知文本。后续需要继续使用时，可以从 **Agent 实例** 找到自己的实例，再通过访问地址进入助手。会话中的连续修改与文件持久化是不同能力。对重要成果，应主动保存到业务文档；是否能跨重启保留会话或文件，取决于应用和模板的存储配置，不要仅根据一次对话成功就判断数据已持久化。

仅用于体验、后续不再使用的实例，可以在确认所需内容已保存后清理：

1. 在 **Agent 实例** 中核对本次创建的实例名称、实例 ID 和所有者。
2. 单击该行 **更多 > 删除**，阅读确认提示后提交。
3. 刷新列表，确认实例删除完成；仍显示删除中时等待处理，不重复创建替代实例。
4. 若模板配置了持久化数据，由管理员核对关联存储及其保留策略。

不要删除团队共用模板、其他人的实例或整个 AgentWay 服务。实例任务结束不等于计算资源已释放；保留运行实例前，应了解模板资源规格和平台的回收策略。

常见问题

现象	先检查什么
找不到“创建实例”或看不到模板	当前账号的角色、模板可见范围，以及模板是否已有发布版本。
创建失败或长时间未进入运行中	记录实例 ID 和错误信息，由管理员检查资源配额、镜像拉取、存储及启动配置。
已运行但访问地址为空	模板运行环境是否配置了访问端口与 Path，平台是否完成实例入口配置。
访问地址提示域名无法解析	实例域名的 DNS 配置及当前网络；控制台能打开不代表实例域名也能解析。
访问地址打开后出现 404 或页面请求失败	域名、路由转发模式及应用访问路径是否匹配。
页面提示未授权或连接失败	是否访问正确实例、使用该实例对应令牌，以及网关连接配置是否正确。

能打开助手，但模型没有回复	先查看聊天区错误信息。若返回 400 和 Invalid model name，由管理员核对模板请求的完整模型名称与模型网关对外提供的名称是否一致；名称前缀也需要核对。修正后重新发送任务。其他错误再检查模型凭证、Token 配额和网络连通性。
纪要遗漏信息或出现虚构内容	提供明确的输入和输出要求，要求只依据材料，并逐项核对结果。

获取支持时，提供实例 ID、模板名称与版本、发生时间、脱敏错误信息和出错步骤。不要提供 Gateway Token 或模型密钥。

为团队准备模板

本节供管理员首次准备环境时使用。员工日常创建助手，可以复用已准备好的模板。

准备模型和额度

在 **AGENT INFRA > 模型网关** 配置可用模型，确认模板请求的完整模型名称（包括前缀）与网关对外提供的名称一致。使用该模板创建一个测试实例，在助手聊天区发送一条简单任务并确认收到模型回复；只看到实例运行中或网关在线，不能说明模型调用成功。再核对 **成本控制 > Token 配额** 和 **计算资源配额**，确保试用账号有可用额度。模型密钥由管理员通过平台配置管理，不要求员工粘贴到聊天内容或在个人实例中重复配置。

准备运行环境和模板

在 **AGENT 应用 > 运行环境** 核对 OpenClaw 的镜像、启动命令、CPU、内存、访问端口、访问 Path 和健康检查路径。镜像和资源规格应使用已验证的组合，不直接照搬其他环境的配置。

在 **AGENT 应用 > 模板管理** 中，选择已有 OpenClaw 模板，核对运行环境、文件预设、启动脚本和可见范围。需要修改时通过新草稿维护，确认后发布版本，再供员工选择。

模板中的模型连接和应用配置应预先准备好。本文任务仅需文本处理，不要求为助手配置企业资料访问凭证或安装额外 Skill。

配好员工实际使用的访问入口

在 **平台设置** 中核对系统域名和入口配置，使实例生成的域名能从员工网络解析并访问。转发模式应与 OpenClaw 的页面路径和连接方式匹配，不能仅以控制台首页可打开作为验收结果。

正式交给员工前，用普通使用者账号创建一个实例，完整执行本文的会议整理、追加修改和清理步骤。至少确认：模板可见、实例可创建、助手可访问、模型可回复、输出可核对、资源可清理。

下一步

这次体验跑通后，可以将团队常用的写作要求、输出格式和工具配置整理进模板，让新建助手获得一致的初始配置。需要进一步读取企业资料或操作业务系统时，再按业务范围配置凭证、网络权限和 Skill。

批量启动沙箱运行 RL Rollout

最近更新时间：2026-10-08 17:40:01

训练 Coding Agent 时，一轮采样往往要执行多次代码尝试：每次尝试准备独立环境、修改代码、运行测试，再把奖励交给训练系统。如果每次都等待上一个环境启动和执行结束，整轮采样会花更多时间。

本指南演示如何在 Agent 集群中一次提交 10 个独立沙箱，并行完成“修复代码 → 运行测试 → 返回奖励”，同时测量整批环境准备和任务完成的时间。适合正在搭建 RL Rollout、代码评测或批量工具执行平台，希望先验证环境创建和结果收集流程的开发者。

你将完成什么

每个沙箱独立处理同一个代码修复任务：订单金额达到 99 元时应免运费，原函数却使用 `total > 99`。任务先确认边界测试失败，再将条件修复为 `total >= 99`，重新测试，成功时返回 `reward=1`。

一次运行将得到：

- 10 个隔离的任务环境，每个环境拥有独立的代码和测试目录。
- 10 条奖励记录，可按 Episode ID 关联到对应沙箱。
- 整批沙箱全部准备完成的时间、全部奖励返回的时间，以及每个沙箱的启动分布。

为了先测清环境准备耗时，Demo 使用固定修复动作，无需模型密钥或 GPU。接入真实 RL 系统时，再将固定动作替换为模型生成的动作；奖励计算和结果汇总链路可以继续复用。

开始前准备

1. 已有可访问的 Agent 集群，配置好 kubeconfig；本机安装 Python 3.9 或以上版本和 kubectl。
2. 集群中有 `cube` RuntimeClass，节点可运行该运行时，且能拉取 `python:3.12-alpine` 镜像。
3. 账号允许查询 RuntimeClass、创建和删除测试 Namespace、ConfigMap、Job，以及读取 Pod、事件和日志。
4. 集群有足够资源并行运行 10 个沙箱。每个任务容器请求 100m CPU、64Mi 内存，限制为 500m CPU、128Mi 内存；还需计入 RuntimeClass 的 Pod overhead。

```
export KUBECONFIG='/absolute/path/to/your/agc-kubeconfig'
kubectl config current-context
kubectl get nodes
kubectl get runtimeclass cube -o yaml
```

例如 RuntimeClass 每个 Pod 额外需要 768Mi 内存，10 个任务就需额外预留 7.5GiB，不能只按业务容器内存估算容量。

将文末完整脚本保存为 `rl-batch-demo.py`。脚本只依赖 Python 标准库和 kubectl，不需要下载内部仓库。

第一步：一次运行 10 个 Episode

在脚本所在目录执行：

```
python3 rl-batch-demo.py run --count 10 --output rl-parallel-10
```

脚本会生成独立的 `agc-rl-...` 命名空间，创建 `completions=10`、`parallelism=10` 的 Job。每个 Pod 对应一个 Episode，使用自己的 `emptyDir` 工作目录；不共享代码修改结果。

运行后会打印命名空间。任务完成后，终端显示统计摘要，完整结果保存到 `rl-parallel-10/`。已有同名本地目录时脚本会停止，防止覆盖实验记录；再次运行请换一个输出目录。

镜像或 RuntimeClass 名称不同时，通过参数指定：

```
python3 rl-batch-demo.py run --count 10 \
  --runtime YOUR_RUNTIME_CLASS \
  --image YOUR_PYTHON_IMAGE \
  --output rl-parallel-custom
```

自有镜像需包含 `python` 命令，支持以 UID/GID 1000 执行 Python 3.9 以上程序。节点需具备相应拉取权限。

第二步：核对奖励与启动时间

```
cat rl-parallel-10/results.json
```

先检查 `episodes=10`、`successful_episodes=10`、`reward_sum=10`。脚本还会检查每个任务的原始边界测试失败、修复后测试通过、容器退出码为 0 且无重启；任一条件不满足都会报错，不把失败任务排除后继续给出成功报告。

再看以下指标，单位均为秒：

指标	回答的问题
<code>batch_to_all_ready_seconds</code>	从 Job 创建到本批最后一个 Episode 的 Python 进程写好代码和测试文件，需要等多久？
<code>pod_to_ready_p50_seconds</code>	从各自 Pod 创建到任务环境准备完成，一半沙箱在多久内可执行？
<code>pod_to_ready_p95_seconds</code>	本次样本中，较慢沙箱的准备时间是多少？
<code>batch_to_all_rewards_seconds</code>	从 Job 创建到最后一条奖励产生，整批计算花了多久？

`client_job_complete_seconds`

本机提交创建到观察到 Job Complete 的总等待时间是多少？

`ready` 事件由每个沙箱在代码和测试文件准备完毕后写入日志，比只看 Pod Running 更贴近“可以开始执行任务”。所有环境先后达到 ready 不代表它们始终同时存活；先完成的 Episode 会正常退出。

启动统计包含 Job 控制器创建 Pod、调度、运行时、镜像准备和 Python 初始化，不能解读为纯虚拟机启动耗时。`client_job_complete_seconds` 还包含客户端请求、任务退出、控制器更新状态和轮询等待。

Kubernetes 创建时间与容器事件来自不同时间源，测试前需保证集群时钟同步。Kubernetes 创建时间为秒级，结果应按约秒级理解；脚本遇到负时延或时间顺序异常会报错。

第三步：用相同任务数观察并行效果

先保存并清理上一轮，再让同样 10 个任务以单并发执行：

```
python3 rl-batch-demo.py cleanup --output rl-parallel-10
python3 rl-batch-demo.py run --count 10 --parallelism 1 --output rl-sequential-10
cat rl-sequential-10/results.json
```

比较两组的 `batch_to_all_rewards_seconds` 和 `client_job_complete_seconds`，可以看到并行采样对整批任务等待时间的影响。两组任务数保持相同，避免用 5 个串行任务与 10 个并行任务直接比较。

单并发组中，后面的 Episode 需要等待前面的任务结束才会创建。因此该组的 `batch_to_all_ready_seconds` 含排队时间，不等于单个沙箱启动变慢。并行缩短整批耗时，也不代表单个沙箱相对其他运行时快了同样倍数。

一次实测的参考结果

2026 年 9 月 21 日，使用同一任务脚本和计时口径、固定的 Python 镜像 digest、相同的 Cube 运行时和容器资源规格，重新执行了下表三组测试。每个沙箱为单容器、独立 `emptyDir`，容器 CPU 限制为 500m、内存限制为 128Mi；可调度范围固定为两个 64 vCPU 节点。

检查项	10 路并行	单并发	逐节点预热后 10 路并行
测试规模	1 轮 × 10 个任务	1 轮 × 10 个任务	3 轮 × 10 个任务
完成任务 / 返回奖励	10 / 10，奖励均为 1	10 / 10，奖励均为 1	30 / 30，奖励均为 1
沙箱创建 RunPodSandbox： P50 / P95	752 / 765 毫秒	676 / 709 毫秒	761 / 835 毫秒

容器创建 CreateContainer: P50 / P95	15 / 23 毫秒	14 / 15 毫秒	14 / 21 毫秒
容器启动 StartContainer: P50 / P95	176 / 211 毫秒	175 / 209 毫秒	180 / 265 毫秒
Python 模块导入: P50 / P95	1.32 / 1.33 秒	1.32 / 1.35 秒	1.33 / 1.37 秒
导入完成至应用 ready: P50 / P95	2.3 / 28.8 毫秒	2.3 / 2.7 毫秒	2.3 / 2.7 毫秒
整批应用环境准备完成（含 Python 初始化）	约 3.7 秒	约 93.1 秒	三轮: 3.2 / 3.5 / 3.6 秒
单 Pod 创建至应用 ready: P50 / P95	3.7 / 3.7 秒	3.3 / 3.4 秒	3.4 / 3.5 秒
整批奖励全部产生	约 8.4 秒	约 97.6 秒	三轮: 7.9 / 8.2 / 8.2 秒
客户端观察到 Job 完成	约 12.7 秒	约 101.6 秒	三轮: 12.9 / 13.0 / 12.2 秒

执行顺序为 10 路并行、单并发，再在两个节点分别执行一次相同任务预热，最后运行三轮 10 路并行。全部 50 个正式任务均命中镜像缓存；前两组也不是冷启动，第三组用于观察显式预热后的重复运行结果，不能据此计算冷启动与热启动的加速比例。

P50 / P95 以各组单个任务为样本计算：前两列各 10 个样本，第三列合并 30 个样本；整批耗时按各轮分别展示。单并发的整批耗时包含等待前一个任务结束的排队时间，各阶段分位数不能直接相加得到整体分位数。

RunPodSandbox、CreateContainer 和 StartContainer 的耗时来自同一宿主机的运行时调用与返回日志。RunPodSandbox 包含沙箱及网络准备，不包含后续容器启动、Python 初始化和任务文件准备。

Python 导入记录代码段墙钟耗时，可能包含运行时、I/O 和调度等待；“导入完成至应用 ready”还包含文件准备、函数定义和少量事件输出。Job/Pod 创建时间为秒级，整批和应用 ready 时间按约秒级理解。

本次三组都使用调度器正常分配节点，没有改变共享运行时配置。具体节点分布、镜像 imageID、逐任务状态和原始日志已随测试结果保留。镜像、节点负载、任务初始化和并发数都会影响结果，小样本 P95 不作为大规模尾延迟或服务承诺。

保存结果与清理

输出目录包含以下文件：

文件	用途
<code>results.json</code>	汇总和逐 Episode 指标、节点分布、实际镜像 imageID。
<code>rollout-*.log</code>	每个 Episode 的 ready、reward 事件及测试结果。

<code>job.json</code> 、 <code>Pods.json</code> 、 <code>events.txt</code>	任务状态和故障排查依据。
<code>receipt.json</code>	本次测试的 <code>context</code> 、命名空间和所有权信息，用于安全清理。

结果已保存后，清理单并发测试：

```
python3 rl-batch-demo.py cleanup --output rl-sequential-10
```

如果没有执行单并发组，就清理 `rl-parallel-10`。脚本核对测试资源的 UID 和标签后，先删除 Job 及其 Pod，再删除独立命名空间。本机输出目录会保留。Pod 删除后，`emptyDir` 中的代码文件也随之删除；真实任务需在退出前上传需要保留的补丁、轨迹和奖励。

遇到问题时

脚本异常时，先记录终端显示的命名空间，再检查：

```
kubectl -n YOUR_TEST_NAMESPACE get job,pods
kubectl -n YOUR_TEST_NAMESPACE get events --sort-by=.metadata.creationTimestamp
kubectl -n YOUR_TEST_NAMESPACE logs YOUR_POD_NAME -c episode
```

现象	检查方向
沙箱长期 Pending	节点可调度状态、RuntimeClass 调度条件、容器资源与 Pod overhead。
镜像拉取失败	镜像名称、仓库权限和节点网络。
Job Failed 或等待超时	输出目录中的 Pod 状态、事件与各 Episode 日志；脚本不自动重试失败任务。
没有 <code>results.json</code>	任务或结果校验未通过，先处理报错，不将部分完成当作整批成功。
时间顺序异常	集群时钟同步及日志事件是否完整。
清理失败	确认 kubeconfig 仍可访问原 context，检查资源终止状态；不要强制去除共享组件的 finalizer。

接入自己的训练流程

把脚本中的固定代码修复动作替换为真实模型和工具调用，为每个 Episode 准备独立任务输入、工作目录与奖励规则。训练调度器可以按一轮采样的规模设置 `completions`，按集群可用容量设置 `parallelism`，收齐奖励和轨迹后交给训练进程。

本 Demo 不更新模型权重。它验证的是 RL 系统中的环境准备、并行执行和奖励回收；先把这条链路跑通，再接入真实模型、重试策略和结果存储。

完整运行脚本

复制下面代码，保存为 `rl-batch-demo.py`。脚本将体验规模限制在最多 20 个 Episode，便于首次验证；这个限制不代表产品容量上限。

```
#!/usr/bin/env python3
"""Reproducible, bounded RL rollout environment demo; requires
kubectrl."""
import argparse
import concurrent.futures
import datetime
import json
import math
import os
from pathlib import Path
import subprocess
import sys
import time
import uuid

WORKER = r'''
import json, os, pathlib, subprocess, time
p = pathlib.Path('/workspace')
os.chdir(p)
p.joinpath('shipping.py').write_text('def shipping(total):\n    return 0
if total > 99 else 10\n')
p.joinpath('test_shipping.py').write_text("""import unittest
from shipping import shipping
class ShippingTest(unittest.TestCase):
    def test_below(self): self.assertEqual(shipping(98), 10)
    def test_boundary(self): self.assertEqual(shipping(99), 0)
    def test_above(self): self.assertEqual(shipping(100), 0)
if __name__ == '__main__': unittest.main()
""")
```

```
def emit(event, **data):
    print(json.dumps(dict(event=event, at=time.time(),
                          episode_id=os.environ['EPISODE_ID'], **data)), flush=True)

def test():
    result = subprocess.run(['python', '-B', 'test_shipping.py'],
                            text=True, capture_output=True, timeout=20)
    return result.returncode, result.stderr

emit('ready')
before, before_output = test()
if before == 0:
    raise RuntimeError('baseline must fail the boundary test')
# Deterministic stand-in for a coding Agent's action; no external model
call.
p.joinpath('shipping.py').write_text(p.joinpath('shipping.py').read_text
().replace('total > 99', 'total >= 99'))
after, after_output = test()
emit('reward', reward=int(after == 0), baseline_exit=before,
     tests_exit=after, tests_output=after_output)
if after != 0:
    raise RuntimeError('patched code failed its tests')
'''

def percentile(values, q):
    values = sorted(values)
    i = (len(values) - 1) * q
    lo, hi = math.floor(i), math.ceil(i)
    return values[lo] + (values[hi] - values[lo]) * (i - lo)

def collect(pods, logs, job, count):
    origin = datetime.datetime.fromisoformat(job['metadata']
['creationTimestamp']).replace('Z', '+00:00').timestamp()
    rows = []
    if len(pods) != count:
        raise ValueError('Pod count differs from requested episodes;
inspect failures/replacements')
    for pod, log in zip(pods, logs):
        if pod['status']['phase'] != 'Succeeded':
```

```
        raise ValueError('Episode did not succeed: ' +
pod['metadata']['name'])
    events = [json.loads(line) for line in log.splitlines() if
line.strip()]
    ready = [x for x in events if x['event'] == 'ready']
    reward = [x for x in events if x['event'] == 'reward']
    if len(ready) != 1 or len(reward) != 1:
        raise ValueError('Expected one ready and one reward event
per episode')
    r, w = ready[0], reward[0]
    if r['episode_id'] != pod['metadata']['name'] or w['episode_id']
!= r['episode_id']:
        raise ValueError('Episode identity mismatch')
    if w['reward'] != 1 or w['tests_exit'] != 0 or
w['baseline_exit'] == 0:
        raise ValueError('Reward or baseline verification failed')
    created = datetime.datetime.fromisoformat(pod['metadata']
['creationTimestamp'].replace('Z', '+00:00')).timestamp()
    if not origin <= created <= r['at'] <= w['at']:
        raise ValueError('Invalid timestamp order; check clock
synchronization')
    status = pod['status']['containerStatuses'][0]
    if status['restartCount'] != 0 or status['state']['terminated']
['exitCode'] != 0:
        raise ValueError('Unexpected restart or exit code')
    rows.append(dict(pod=r['episode_id'], node=pod['spec']
['nodeName'],
                    image_id=status.get('imageID'), reward=w['reward'],
                    pod_to_ready_seconds=r['at']-created,
                    batch_to_ready_seconds=r['at']-origin,
                    batch_to_reward_seconds=w['at']-origin))
    startup = [x['pod_to_ready_seconds'] for x in rows]
    return dict(epochs=count, successful_epochs=len(rows),
reward_sum=sum(x['reward'] for x in rows),
                batch_to_all_ready_seconds=max(x['batch_to_ready_seconds'] for x
in rows),
                batch_to_all_rewards_seconds=max(x['batch_to_reward_seconds']
for x in rows),
                pod_to_ready_p50_seconds=percentile(startup, .5),
                pod_to_ready_p95_seconds=percentile(startup, .95),
```

```
pod_to_ready_max_seconds=max(startup), pods=rows)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('action', choices=['run', 'cleanup'])
    parser.add_argument('--output', required=True)
    parser.add_argument('--count', type=int, default=10)
    parser.add_argument('--parallelism', type=int)
    parser.add_argument('--image', default='python:3.12-alpine')
    parser.add_argument('--runtime', default='cube')
    args = parser.parse_args()
    output = Path(args.output)
    parallel = args.count if args.parallelism is None else
args.parallelism
    if not 1 <= parallel <= args.count <= 20:
        parser.error('require 1 <= parallelism <= count <= 20')
    if args.action == 'cleanup':
        receipt = json.loads((output/'receipt.json').read_text())
        context = receipt['context']
    else:
        context = subprocess.check_output(['kubectl', 'config',
'current-context'], text=True).strip()
    def kubectl(*parts, data=None):
        return subprocess.check_output(['kubectl', '--context', context,
'--request-timeout=30s', *parts],
        input=json.dumps(data) if data is not None else None,
text=True, timeout=60)
    def create(obj):
        return json.loads(kubectl('create', '-f', '-', '-o', 'json',
data=obj))
    if args.action == 'cleanup':
        ns = receipt['namespace']
        raw = kubectl('get', 'namespace', ns, '--ignore-not-found', '-o', 'json')
        if not raw.strip():
            print('Namespace already absent'); return
        live = json.loads(raw)
        if live['metadata']['uid'] != receipt['uid'] or
live['metadata'].get('labels', {}).get('agc-demo-run') !=
```

```
receipt['run_id']:
    raise RuntimeError('Ownership check failed; refusing
cleanup')
    raw_job = kubectl('-n', ns, 'get', 'job', 'rollout', '--ignore-
not-found', '-o', 'json')
    if raw_job.strip():
        expected_uid = receipt.get('job_uid')
        if expected_uid is None and (output/'job.json').exists():
            expected_uid =
json.loads((output/'job.json').read_text())['metadata']['uid']
        if json.loads(raw_job)['metadata']['uid'] != expected_uid:
            raise RuntimeError('Job ownership check failed; refusing
cleanup')
        kubectl('-n', ns, 'delete', 'job', 'rollout', '--
cascade=foreground', '--wait=false')
        deadline = time.monotonic() + 120
        while kubectl('-n', ns, 'get', 'job', 'rollout', '--ignore-
not-found', '-o', 'name').strip():
            if time.monotonic() > deadline: raise TimeoutError('Job
or Pods still terminating')
            time.sleep(2)
        kubectl('delete', 'namespace', ns, '--wait=false')
        deadline = time.monotonic() + 120
        while kubectl('get', 'namespace', ns, '--ignore-not-found', '-
o', 'name').strip():
            if time.monotonic() > deadline: raise
TimeoutError('Namespace still terminating; inspect finalizers')
            time.sleep(2)
        print('Deleted test namespace:', ns); return
    output.mkdir(parents=True, exist_ok=False)
    run_id = uuid.uuid4().hex[:12]
    ns = 'agc-rl-' + run_id
    receipt = dict(context=context, namespace=ns, run_id=run_id)
    # Save intended namespace before any cluster mutation so failures
    remain traceable.
    (output/'receipt.json').write_text(json.dumps(receipt, indent=2))
    kubectl('get', 'runtimeclass', args.runtime)
    created = create(dict(apiVersion='v1', kind='Namespace',
metadata=dict(name=ns, labels={'agc-demo-run':run_id})))
    receipt['uid'] = created['metadata']['uid']
```

```
(output/'receipt.json').write_text(json.dumps(receipt, indent=2))
print('Test namespace:', ns, flush=True)
create(dict(apiVersion='v1', kind='ConfigMap',
metadata=dict(name='episode', namespace=ns), data=
{'episode.py':WORKER}))
job = dict(apiVersion='batch/v1', kind='Job',
metadata=dict(name='rollout', namespace=ns), spec=dict(
    completions=args.count, parallelism=parallel, backoffLimit=0,
activeDeadlineSeconds=240,
    template=dict(metadata=dict(labels={'app':'rl-batch-demo'}),
spec=dict(
    runtimeClassName=args.runtime, restartPolicy='Never',
automountServiceAccountToken=False,
    securityContext=dict(runAsUser=1000, runAsGroup=1000,
fsGroup=1000, runAsNonRoot=True),
    containers=[dict(name='episode', image=args.image,
imagePullPolicy='IfNotPresent',
    command=['python', '-B', '/demo/episode.py'],
    env=
[dict(name='EPISODE_ID', valueFrom=dict(fieldRef=dict(fieldPath='metadata
.name')))],
    securityContext=dict(allowPrivilegeEscalation=False,
capabilities=dict(drop=['ALL']),
seccompProfile=dict(type='RuntimeDefault')),

resources=dict(requests=dict(cpu='100m',memory='64Mi'),limits=dict(cpu='
500m',memory='128Mi')),
    volumeMounts=
[dict(name='script', mountPath='/demo', readOnly=True), dict(name='workspac
e', mountPath='/workspace')]),
    volumes=
[dict(name='script', configMap=dict(name='episode')), dict(name='workspace
', emptyDir={})]))))
(output/'job-submitted.json').write_text(json.dumps(job, indent=2))
start = time.monotonic()
created_job = create(job)
receipt['job_uid'] = created_job['metadata']['uid']
(output/'receipt.json').write_text(json.dumps(receipt, indent=2))
deadline = start + 270
while True:
```

```

        live = json.loads(kubect1('-n',ns,'get','job','rollout','-o','json'))
        conditions = {c['type']:c['status'] for c in live.get('status',{}).get('conditions',[])}
        if conditions.get('Complete')==True or conditions.get('Failed')==True: break
        if time.monotonic()>deadline: break
        time.sleep(1)
        observed = time.monotonic()-start
        pods = json.loads(kubect1('-n',ns,'get','pods','-l','job-name=rollout','-o','json'))['items']
        (output/'job.json').write_text(json.dumps(live, indent=2))
        (output/'pods.json').write_text(json.dumps(pods, indent=2))
        (output/'events.txt').write_text(kubect1('-n',ns,'get','events','--sort-by=.metadata.creationTimestamp'))
        def get_log(pod):
            name = pod['metadata']['name']
            try: log = kubect1('-n',ns,'logs',name,'-c','episode')
            except subprocess.SubprocessError: log = ''
            (output/(name+'.log')).write_text(log)
            return log
        with concurrent.futures.ThreadPoolExecutor(max_workers=4) as pool:
            logs = list(pool.map(get_log,pods))
        if conditions.get('Complete')!='True':
            raise RuntimeError('Job failed or timed out; inspect saved status/events/logs, then cleanup')
        report = collect(pods, logs, live, args.count)
        report.update(context=context, namespace=ns, image=args.image, runtime=args.runtime,
                      parallelism=parallel, cache_state='uncontrolled', client_job_complete_seconds=observed)
        (output/'results.json').write_text(json.dumps(report, indent=2))
        print(json.dumps({k:v for k,v in report.items() if k!='pods'},indent=2))

if __name__=='__main__':
    main()

```

常见问题与故障排查

常见问题

最近更新时间：2026-10-08 17:40:01

已有 TKE 集群能直接接入吗？

暂不支持。内测期间仅支持新建 TKE 集群，存量 TKE 集群接入能力正在加速开发中。

内测期间如何收费？

内测期间不收取 AGC 服务费。GA 后将按照管理的 CPU 和内存量级收取服务费。计算、存储和网络等底层云资源另行计费。

集群创建完成后为什么任务仍处于 Pending？

集群创建与节点资源准备是两个步骤。先查看是否有 Ready 节点，再通过 Pod 事件检查容量、RuntimeClass、镜像和存储挂载。

删除 Pod 后工作文件还在吗？

保存在容器临时目录中的文件会随环境释放而丢失。需要保留的文件应写入挂载的持久卷，并保留对应 PVC。删除 PVC 可能触发后端卷回收。

工作区迁移后，任务如何继续执行？

新运行环境挂载原工作区后，可以读取原有代码、输入和输出文件，再启动程序继续处理。应用根据已保存的业务进度确定下一步；已完成的外部写操作应避免重复执行。

如何测量启动速度？

[Kubernetes 快速入门](#) 记录提交创建到工作区可写的耗时；比较数据时保持镜像、资源、存储和缓存条件一致，并重复采样。

测试结束后需要清理哪些资源？

释放测试 Pod 或沙箱；按数据保留需求处理 PVC；不再使用的节点和云资源也需要单独清理。