多云资源编排平台

Multi-Cloud Resource Orchestration Platform

架构设计文档

版本 v0.1 · 草案

面向对象：平台工程团队 / 各业务团队基础设施负责人

目录

1. 背景与设计目标

1.1 现状与问题

1.2 设计目标

2. 总体架构

2.1 各层职责一览

2.2 端到端请求时序

3. 声明式 API 层设计

3.1 YAML 声明式提交（推荐路径）

3.2 REST / gRPC API（面向自动化脚本与非 K8s 团队）

4. 控制平面核心设计

4.1 Composition：抽象与具体资源的映射

4.2 生命周期语义

5. 多租户隔离设计

5.1 逻辑隔离：Namespace + RBAC

5.2 物理隔离：独立云账号 / 订阅 / 项目

5.3 准入策略：防止团队申请到不合规配置

6. 审计设计

7. 成本统计与资源泄漏检测

7.1 前提：强制标签

7.2 成本统计

7.3 资源泄漏检测

8. 迁移路径

9. 方案取舍：为什么不是“统一 Terraform 规范”

附录：术语表

1. 背景与设计目标

1.1 现状与问题

Apple 内部各团队目前各自使用 Terraform 脚本在 AWS、Azure、GCP 上独立管理资源，缺乏统一管控，长期积累下来暴露出三类典型问题：

- 资源泄漏：各团队 state 相互独立，没有全局视角能够核对“云上实际存在的资源”与“应当存在的资源”是否一致，孤儿资源难以被发现。
- 配置不一致：不同团队各自编写 Terraform module，标准、命名规范、标签策略不统一，跨团队复用困难，安全基线难以强制落地。
- 权限混乱：云账号凭证的授予和收回缺乏统一入口，难以确保团队之间的最小权限边界，也难以在出现问题时快速定位责任方。

1.2 设计目标

- 统一入口：业务团队通过声明式 YAML 或标准 API 提交资源需求，无需关心底层云厂商差异。
- 最终一致性：平台持续核对期望状态与云端实际状态，自动纠正漂移，而非一次性执行后不再过问。
- 生命周期管理：创建、更新、删除、查询状态四类操作均可通过统一接口完成。
- 多租户隔离：不同团队只能操作、查看自己名下的资源，逻辑隔离与云账号物理隔离双重保障。
- 可审计、可计量：每一次变更都可追溯到发起人，每一份资源都能归集到对应团队的成本账单。

> **设计取舍说明**
> 本方案不是要替换 Terraform，而是在其之上构建一个持续运行的控制平面：核心引擎采用 Crossplane（Kubernetes 原生的基础设施编排框架），对暂不支持的资源类型通过 provider-terraform 桥接。选择这一路线的原因、以及与“统一 Terraform 规范”这一更轻量方案的取舍对比，见第 9 节。

2. 总体架构

平台采用分层架构，自上而下分别是声明式 API 层、控制平面核心、多租户隔离层、Provider 适配层、以及贯穿始终的审计与成本层。核心设计理念是“持续调谐”而非“一次性执行”：控制平面常驻运行，不断比对期望状态与云端实际状态，这是解决资源泄漏、配置漂移问题的根本机制。

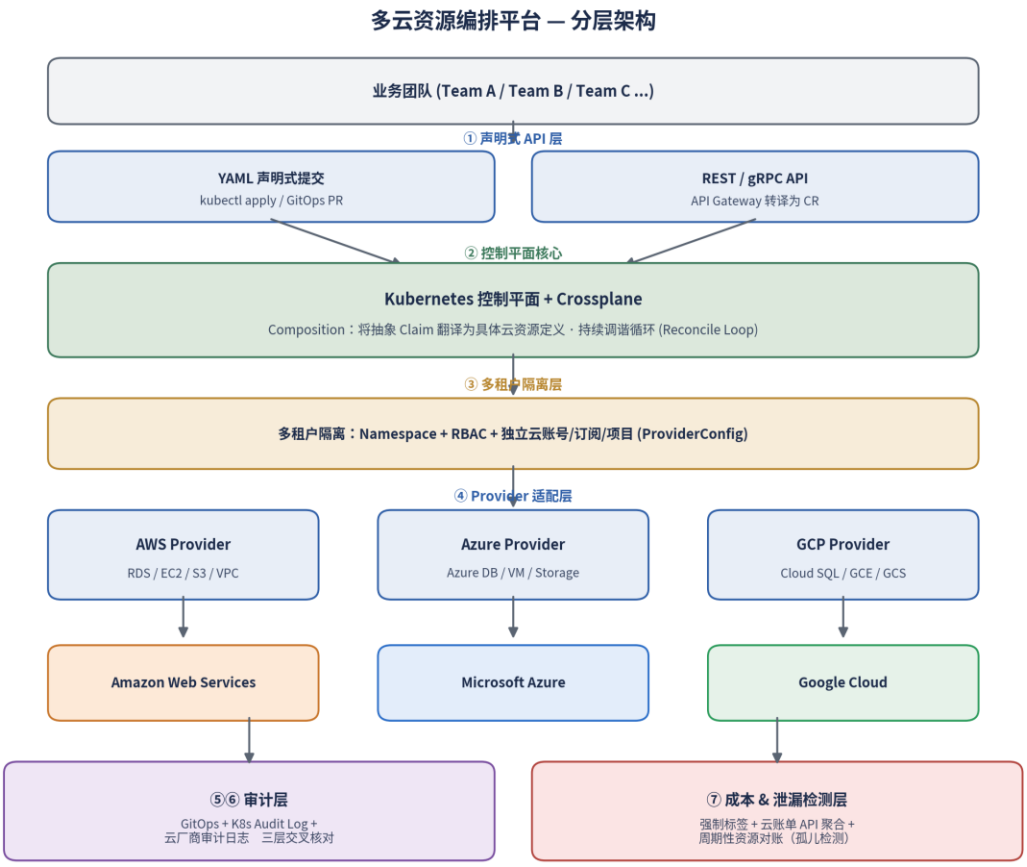

图 2-1　多云资源编排平台分层架构

2.1 各层职责一览

层级	职责	关键技术
① 声明式 API 层	接收团队提交的资源需求，统一转换为内部资源对象	YAML / kubectl / REST / gRPC / API Gateway
② 控制平面核心	将抽象需求翻译为云厂商具体资源，持续调谐保持一致性	Kubernetes + Crossplane Composition
③ 多租户隔离层	逻辑与物理双重隔离，确保团队间资源、权限互不越界	Namespace / RBAC / ProviderConfig
④ Provider 适配层	对接各云厂商 API，屏蔽厂商差异	provider-aws / provider-azure / provider-gcp

层级	职责	关键技术
⑤⑥ 审计层	记录“谁在何时做了什么”，支持事后追溯	GitOps 记录 + K8s Audit Log + 云厂商审计日志
⑦ 成本与泄漏检测层	资源打标、账单聚合、孤儿资源对账	强制标签策略 + 云账单 API + 周期对账 Job

2.2 端到端请求时序

以“团队申请一个数据库实例”为例，说明一次典型请求从提交到资源就绪、再进入持续调谐循环的完整过程：

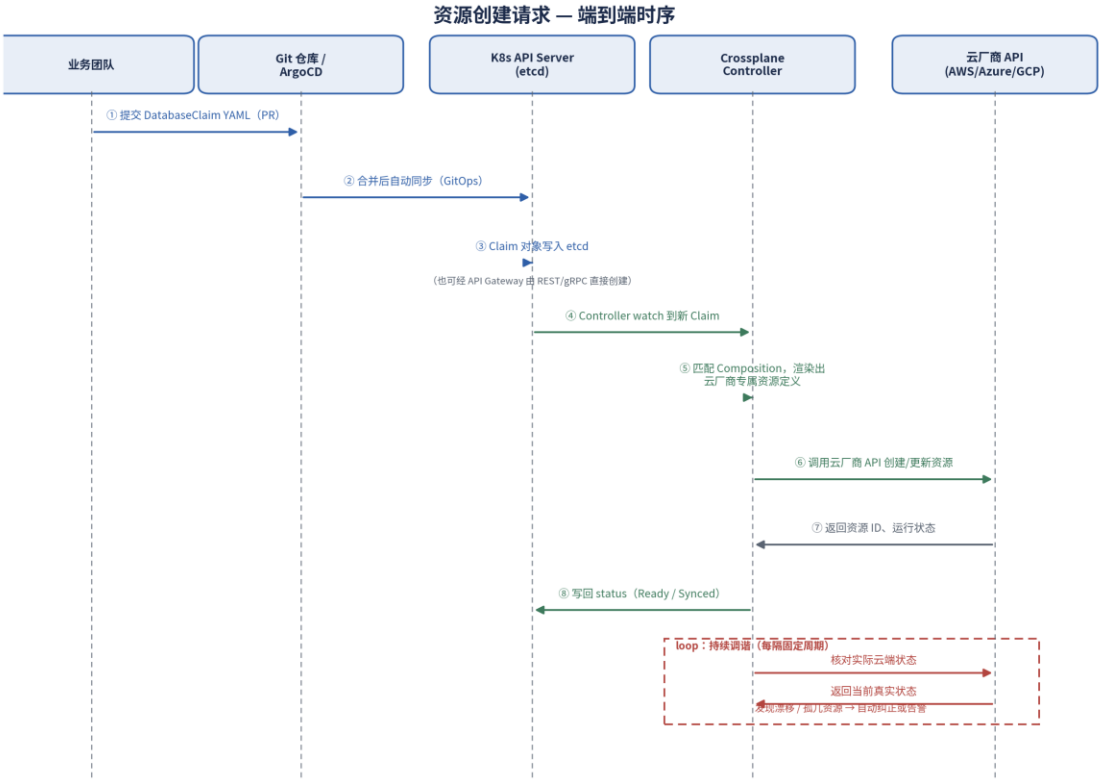

图 2-2　资源创建请求端到端时序（含持续调谐 loop）

需要特别说明第 ⑧ 步之后的 loop：Controller 不会在资源创建成功后就停止工作，而是按固定周期（典型配置为 1–10 分钟）反复核对云端真实状态与期望状态。一旦发现有人绕过平台在云控制台手动改动了资源，或者资源被误删，Controller 会在下一次调谐周期内自动感知并按策略纠正或告警——这正是“持续一致性”需求的落地方式，也是纯命令行式 Terraform 工作流天然缺失的能力。

3. 声明式 API 层设计

平台对外暴露两种等价的提交方式，内部最终都归一为同一种 Kubernetes 自定义资源（Custom Resource），保证无论团队走哪条路径，后续的调谐、审计、成本统计逻辑完全一致。

3.1 YAML 声明式提交（推荐路径）

适合已经熟悉 GitOps 流程的团队：资源需求以 YAML 形式提交 Pull Request，经 Review 后由 ArgoCD/Flux 自动同步到控制平面。这条路径天然带有变更记录和审批留痕。

```
apiVersion: platform.apple.com/v1alpha1
kind: DatabaseClaim
metadata:
  name: payments-db
  namespace: team-payments
spec:
  provider: aws          # aws / azure / gcp
  engine: postgres
  size: db.t3.medium
  region: us-west-2
  environment: prod
  backupRetentionDays: 7
```

3.2 REST / gRPC API（面向自动化脚本与非 K8s 团队）

不熟悉 Kubernetes 的团队可以通过标准 REST 接口提交，由 API Gateway 在后端转译为等价的 Custom Resource。对需要高频调用、强类型契约的内部系统（如 CI/CD 流水线），额外提供 gRPC 接口，复用同一套后端逻辑。

方法	路径	说明
POST	/v1/teams/{team}/databases	创建数据库资源请求
GET	/v1/teams/{team}/databases/{name}	查询资源当前状态
PATCH	/v1/teams/{team}/databases/{name}	更新资源规格（如扩容）
DELETE	/v1/teams/{team}/databases/{name}	发起删除（进入保护期）
GET	/v1/teams/{team}/resources	列出该团队名下全部资源

4. 控制平面核心设计

核心引擎建议直接采用 Crossplane 而非自研调谐引擎：它天然提供持续调谐能力、原生复用 Kubernetes 的存储 / RBAC / 审计体系，并已有成熟的云厂商 Provider 生态。对于个别 Crossplane Provider 暂未覆盖的资源类型，通过 provider-terraform 桥接，在后台调用既有 Terraform 配置补齐，避免生态空窗期阻塞业务。

4.1 Composition：抽象与具体资源的映射

平台团队预先定义一组 Composition，将业务团队看到的抽象概念（如“一个数据库”）映射为具体云厂商资源。业务团队只需在 Claim 中指定 provider 字段，无需了解三家云各自的资源模型差异：

```
# 平台团队维护，业务团队不可见/不可修改
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: database-aws-postgres
spec:
  compositeTypeRef:
    apiVersion: platform.apple.com/v1alpha1
    kind: XDatabase
  resources:
    - name: rds-instance
      base:
        apiVersion: rds.aws.upbound.io/v1beta1
        kind: Instance
        spec:
          forProvider:
            engine: postgres
            instanceClass: db.t3.medium
            tags:
              apple:owner: "{{ claim.namespace }}"
              apple:platform-managed: "true"
```

4.2 生命周期语义

操作	实现方式	备注
创建 / 更新	kubectl apply（幂等）或对应 REST 接口	重复提交同一份声明不会重复创建
删除	kubectl delete，配合 Finalizer 二次确认	有状态资源（数据库等）默认开启删除保护
查询状态	读取 CR 的 status 字段（Ready / Synced / 错误信息）	REST 层对状态做 watch，转为 webhook 通知

5. 多租户隔离设计

隔离设计分两层：Kubernetes 层面的逻辑隔离解决“谁能看到 / 操作哪些资源对象”，云账号层面的物理隔离解决“出了问题影响面有多大”。两层缺一不可——只做逻辑隔离，一旦控制平面本身被攻破，所有团队的云资源都暴露在同一组凭证之下。

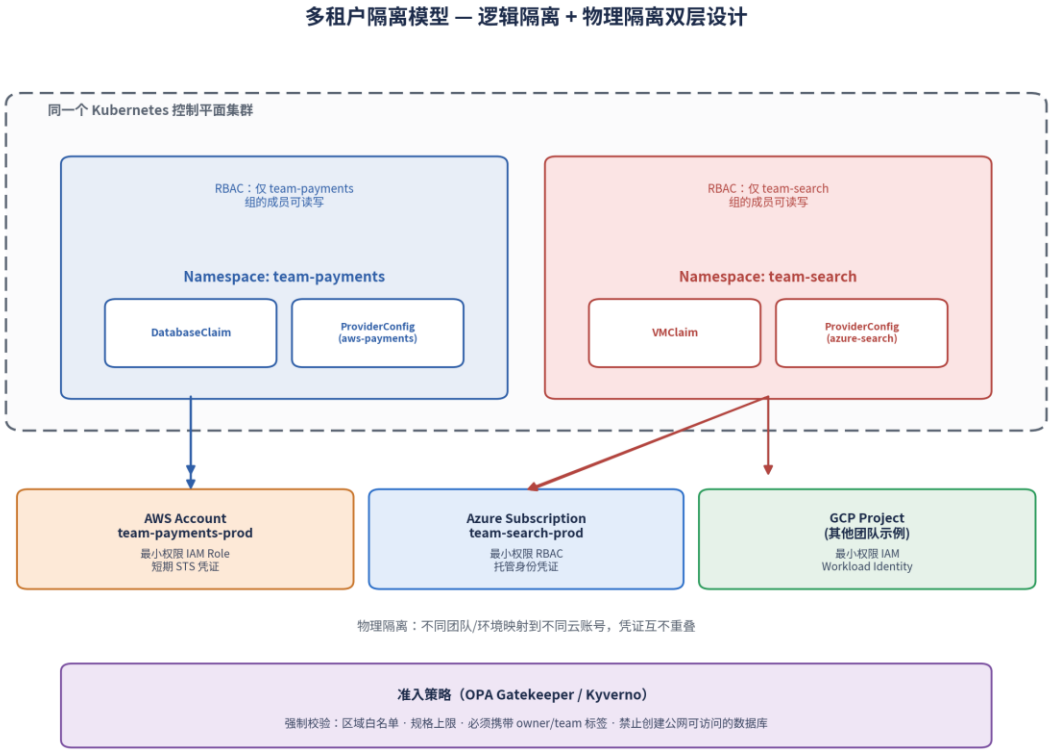

图 5-1　多租户逻辑隔离 + 物理隔离模型

5.1 逻辑隔离：Namespace + RBAC

- 每个团队对应一个独立 Kubernetes Namespace，团队成员通过身份组（如企业 SSO 的 AD 组）映射到该 Namespace 的 RoleBinding。
- 默认策略为最小权限：团队只能读写自己 Namespace 下的资源对象，跨 Namespace 操作需要平台管理员显式授权。

5.2 物理隔离：独立云账号 / 订阅 / 项目

- 每个团队在每个云厂商下拥有独立的 AWS Account / Azure Subscription / GCP Project，通过该 Namespace 专属的 ProviderConfig 绑定对应凭证。
- 凭证优先使用短期、可自动轮换的机制（AWS STS AssumeRole、Azure 托管身份、GCP Workload Identity），避免长期静态密钥落地到集群中。
- 即使控制平面本身出现权限配置错误，影响范围也被限制在单一云账号内，不会波及其他团队。

5.3 准入策略：防止团队申请到不合规配置

在 Namespace / RBAC 边界之外，叠加基于 OPA Gatekeeper 或 Kyverno 的准入策略，在资源被创建前做强制校验，避免出现“权限上允许、但违反公司安全基线”的配置：

- 区域白名单：只允许在审批通过的数据合规区域创建资源。
- 规格上限：限制团队可申请的实例规格，超出部分需要走单独审批。
- 强制标签：拒绝不带 owner / team / environment 标签的提交，为下文的成本统计提供数据基础。
- 安全基线：例如禁止创建对公网开放的数据库实例、强制启用存储加密。

6. 审计设计

审计设计为三层交叉验证，而不是依赖单一数据源，目的是在“记录被篡改”或“操作绕过平台”的情况下依然能够发现异常。

层级	记录内容	回答的问题
GitOps 记录	PR 提交、Review 意见、合并记录	谁请求了变更、谁批准了变更
Kubernetes Audit Log	对 API Server 的每一次读写调用	控制平面实际执行了什么操作、何时执行
云厂商审计日志 (CloudTrail / Activity Log / Cloud Audit Logs)	云端 API 调用的原始记录	云端实际发生了什么，是否与平台记录一致

三层记录理论上应当完全对应；一旦云厂商审计日志中出现某次变更，而 Kubernetes Audit Log 中找不到对应操作，几乎可以确定是有人绕过平台直接操作了云控制台——这类事件应触发告警并计入第 7 节的资源对账流程。

7. 成本统计与资源泄漏检测

7.1 前提：强制标签

Composition 在渲染具体云资源时，强制注入 owner、team、environment、platform-managed 等标签，业务团队无法在提交阶段跳过这一步——这是后续所有成本归集和对账工作的数据基础，如果标签不可靠，后面的统计和检测都无从谈起。

7.2 成本统计

周期性从三家云厂商的账单接口拉取数据（AWS Cost Explorer / CUR、Azure Cost Management API、GCP Billing Export to BigQuery），按标签聚合后汇总到统一看板，团队可以按 Namespace 或标签维度查看自己名下资源的实际花费。由于平台本身运行在 Kubernetes 之上，也可以直接引入 Kubecost / OpenCost 一类现成工具，减少自研成本。

7.3 资源泄漏检测

这是直接针对背景中提出的“资源泄漏”问题设计的机制。分散式 Terraform workflow 下，每个团队的 state 相互独立，没有任何一方拥有全局视角；而在本方案中，所有资源都汇聚到同一个控制平面账本，具备做全局对账的天然条件：

- 周期性任务扫描三朵云上所有带 platform-managed 标签的资源。
- 与 Crossplane 当前管理的资源清单做全量比对。
- 云上存在但账本中没有记录的资源判定为孤儿资源，自动标记并通知对应团队。
- 超过宽限期未被认领或清理的孤儿资源，按预设策略自动下线或升级告警至平台团队。

8. 迁移路径

现状是各团队已有一批基于 Terraform 独立管理的存量资源，不建议大爆炸式切换，采用团队级别的渐进式接管：

- 第一步：对存量资源使用 Crossplane 的 Import 能力（或先经由 provider-terraform 原样接管既有 Terraform 配置），将资源 ID 登记进平台账本，而不是销毁重建。
- 第二步：资源被接管后即刻享受持续调谐、审计、成本统计等平台能力；新建资源一律通过平台声明式流程创建，不再允许团队直接跑 Terraform apply。
- 第三步：新老并存期间，通过第 7.3 节的对账机制持续监控，确保没有游离在平台之外的资源。
- 第四步：按团队分批完成接管后，收回团队对云控制台 / Terraform 直接操作云账号的长期权限，仅保留平台的服务账号具备写权限。

9. 方案取舍：为什么不是“统一 Terraform 规范”

一个更轻量的替代方案是：不建控制平面，只是统一各团队的 Terraform module 规范和 remote backend。这个方案成本更低，但无法覆盖背景中提出的全部需求，取舍对比如下：

需求	统一 Terraform 规范	本方案（Crossplane 控制平面）
配置一致性	可以做到，依赖流程约束	可以做到，Composition 强制约束
资源泄漏检测	困难：每个团队 state 独立，无全局账本	支持：统一账本 + 周期对账
最终一致性 / 自愈	不支持：一次性执行模型	支持：持续调谐
细粒度多租户权限	需要额外自建	原生复用 K8s RBAC
非 Terraform 背景团队接入	有门槛，需学习 HCL	可走 REST/gRPC，无需了解底层
实施与运维成本	较低	较高，需要维护 Kubernetes 控制平面

结论：如果目标只是解决“配置不一致”，统一 Terraform 规范已经足够；但背景中明确提出资源泄漏、权限混乱、需要自助式声明式入口这几项需求，已经超出规范化流程能覆盖的范围，值得投入建设一个真正的控制平面。

附录：术语表

术语	含义
Claim	业务团队提交的资源需求对象，表达意图而非具体云资源
Composition	平台团队定义的映射规则，将 Claim 翻译为具体云厂商资源
Reconcile Loop（调谐循环）	控制器持续比对期望状态与实际状态并纠正差异的机制
ProviderConfig	存放某个云账号访问凭证的配置对象，按 Namespace 隔离
漂移（Drift）	云端实际资源状态与平台记录的期望状态不一致的现象
孤儿资源	存在于云端但未被平台账本记录的资源，即潜在的资源泄漏