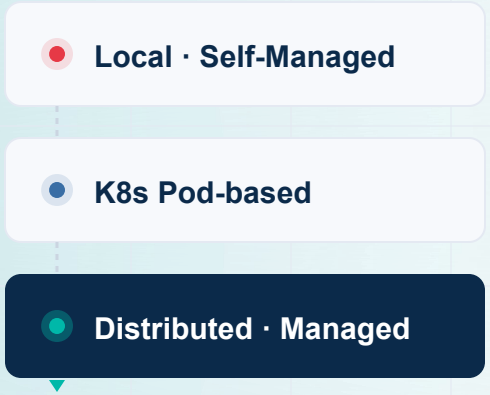
Agent 架构演进

从「单体自管」到「分布式托管」

From Monolithic Self-Managed to Distributed Managed



我们正站在一个决定性拐点

01

Agent 正从**「概念验证」**走向**「大规模生产化部署」**，
这是一次决定性的拐点，不是 Demo 的胜利。

02

但业界的视线还停留在工作流编排与提示词调优——
一个更深、更决定性的工程挑战正在浮现。

03

当成千上万 Agent 长驻企业网络、自主调用工具、处理核心资产，
我们究竟需要一个**怎样的基础设施底座？**

> 工作流是序幕，
> 分布式托管
> 基础设施
> 才是终局。

— 本次演讲的核心讯息

今天我们走这条路径

起点 · 单体 Agent

终点 · 多 Agent 协作

01

CHAPTER 01

单体 Agent

Agent 特殊性 · 三种典型形态
从本地部署到 Runtime Sandbox

02

CHAPTER 02

分布式 Agent

存算分离 · Session · 环境沙箱
从无状态 Loop 到全托管 ReAct

03

CHAPTER 03

多 Agent 协作

Registry · A2A · 协议统一
跨 Agent 协作层的雏形

CHAPTER 01

01

01 / 03 · 单体 Agent

单体 Agent — Agent 与状态放一起的三种形态

先看清单体 Agent 是什么，再谈为什么必须走向分布式

MONOLITHIC AGENT · THREE TYPICAL FORMS

A **本地部署**
个人 · IM 接入

B **K8s Pod 化**
企业 Agent 平台

C **Runtime Sandbox**
安全沙箱

为什么 Agent 这种应用很特殊？

Agent 是一种特殊的工作负载 — 三个维度决定了它的基础设施诉求

01

自主性

AUTONOMY

主动决定「下一步做什么」，
而不是被动响应请求。

vs. 传统请求-响应服务

02

行为不确定性

NON-DETERMINISM

大模型驱动 — 同样输入也可能
走出完全不同的执行路径。

vs. 幂等函数式服务

03

工具环境有依赖

ENV-COUPLED

通过工具调用感知与改造世界，
环境即状态，环境即 Agent。

vs. 无状态微服务

三大特殊性 → 决定了 Agent 需要一整套全新的基础设施，而不是把普通服务框架改一改就够。

形态 A：本地单体部署 — 个人 Agent 的起点

场景：OpenClaw 本地部署 + 企微 / 飞书 IM 接入 — 社区最典型的个人 Agent 用法

四大痛点 · 4 PAIN POINTS

01 · 可用性

机器休眠 / 关机后无法使用

Agent 与个人电脑生命周期强耦合。
合上电脑，服务就消失。

02 · 环境

环境被破坏难以恢复

Agent 装的依赖、临时文件、密钥无处备份，
出错就要花几小时重装环境。

03 · 安全

敏感信息泄密风险

API Key、企业数据散落在个人机器上，
审计和最小权限都无从谈起。

04 · 管理

企业视角难以管理

企业无法看到员工装了几个 Agent、在做什么、
做了些什么 — 治理和合规是黑盒。

形态 B：企业 Agent 平台的 Pod 化托管

典型架构：一个 Agent = 一个 Pod + 挂 PVC 存状态；成千上万个 Agent 长驻集群

现状架构

01 **隔离性不足**

容器级隔离可被突破 · Agent 有可能渗透到 Node，威胁其他租户与集群基础设施

02 **资源成本高**

Agent 是 IO 密集型 — **大部分时间处于空闲**，Pod 常驻占资源，成本极高

03 **可用性差**

单点应用 · 环境异常或 PVC 数据损坏后恢复困难，用户体验断崖

04 **难以运维**

升级期不可用 · 临时数据丢失 · 客户装的环境组件**升级后突然失效**

05 **性能有瓶颈**

当 Agent 从「一人用」变成「作为服务用」，**单 Pod 撑不住**并发压力

Pod 化已经推进一大步 — 但把 Agent 当无差别工作负载塑进 Pod，无法真正解决架构问题。

形态 C · 沙箱化：让 Agent 跑在专用安全沙箱中

论点 — 把 Agent 从 Pod 切换到安全沙箱；先看沙箱本身的性能红利

01 VM级别隔离，毫秒级启动

FAST COLD START

80 ms

COLD START LATENCY

VM级别隔离，按需拉起 · 用完即释放

完美适配任务型 Agent，尤其是 Coding、批处理场景。

Pod 冷启动通常在秒级 — 沙箱级别快了整整两个数量级。

vs. Pod 秒级冷启动

02 秒级暂停 / 恢复

PAUSE / RESUME

暂停期不占资源 · 恢复后状态完好

Agent 是 IO 密集型 — 大部分时间在等用户或等模型。

秒级暂停恢复直接砍掉长期空闲的成本浪费。

成本 · 从常驻到按秒

Agent Wall：Sandbox 之外的安全总关

Sandbox 流量透明转发到 Agent Wall — 三层网络防护 + 凭证注入统一在这里落地

A 三层网络防护

3-LAYER NETWORK GUARD

LAYER 1
网络触达边界 · 只允许白名单出口

LAYER 2
站点防护策略 · 域名 / 路径 / 参数级细化

LAYER 3
敏感操作人工审核 · Human-in-the-Loop

B 凭证注入

ZERO-TRUST CREDENTIAL INJECTION

Agent 身份签发
每个 Agent 实例都有可撤销的短期身份

运行时自动注入
出流量到 Wall 时按目标服务临时换发凭证

Agent 代码零硬编码密钥
最小权限 · 可撤销 · 可审计，符合零信任原则

但单体式，还没到终点

安全防护住了 — 但运维、高可用、横向扩缩容依然是单体的宿命问题

01 运维难

升级即中断

单副本升级过程中服务不可用；
滚动更新 = 每次都要接受一段
"用户会感知到"的窗口期。

SPOF Ops Debt

02 高可用差

一挂就挂到底

Agent 沙箱是有状态单点，
一旦故障即用户不可用；
状态在本地，切主也切不走。

Single Replica

03 无法横向扩缩容

高并发撑不住

当 Agent 从「一人用」变成
「作为服务用」，性能瓶颈
依旧存在 — 单沙箱扩不起来。

No Horizontal

→ **结论：单体架构无解 — 必须走向分布式架构；接下来的每一页，都是围绕"如何做分布式 Agent"展开。**

CHAPTER 02

02

从沙箱到分布式：Runtime + 存算分离才是真正的底座

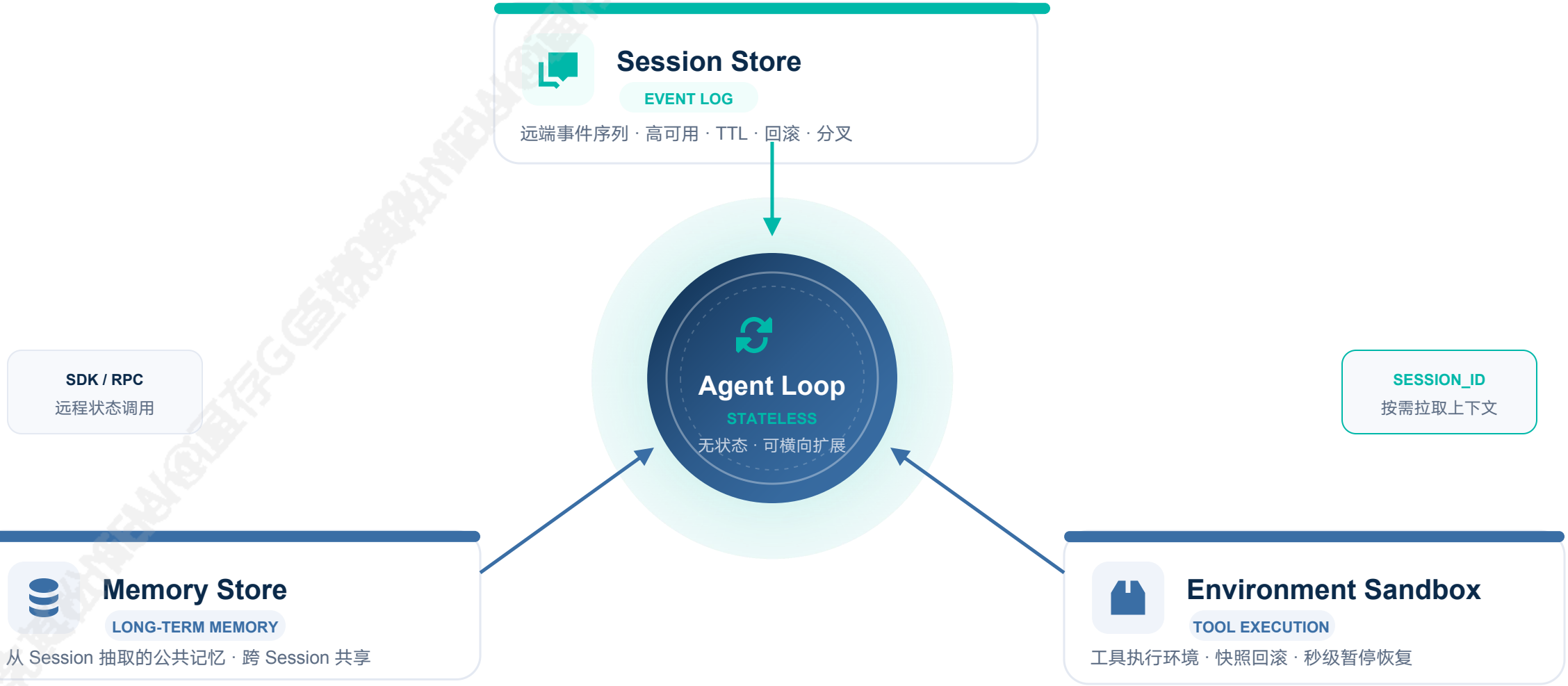
先修安全，再谈分布式；先剥状态，再谈托管

FROM SANDBOX TO DISTRIBUTION

本章路径

Runtime Sandbox → 存算分离 → Session + 环境沙箱 → 无状态 Agent Loop → 函数式托管 → **全托管 ReAct**

存算分离：Agent 分布式架构的核心



存算分离后，Agent Loop 才能被当作普通无状态服务去调度、扩缩容、灰度、迁移。

状态一：Session 剥离 — 事件串搬到远端

DEFINITION

Session 是什么？

一个**事件串**，记录 Agent 与
大模型 / 工具 / 用户的每一次交互。

每次对话，Agent 把过往事件
打包为上下文传给大模型 —
这就是"记忆"的物理形态。

开源 vs. 云上

开源做法	分布式做法
本地文件 .session.json	远端存储 按 SESSION_ID 拉取

CLOUD SESSION · 5 CAPABILITIES

Agent Runtime 的 Session 五项增强能力

01 **事件序列存储**
按顺序追加，只增不改；提供强一致读写与快速查询

02 **高可用 · 自动备份**
多副本存储 + 快照 · 数据丢失概率降到基础设施级别

03 **TTL 自动过期**
按业务生命周期设置过期策略；成本可控，合规也可控

04 **Session 回滚**
回滚到任意历史事件点 · Agent 犯错后可"reset 到刚才那步"

05 **Session 分叉**
从某点复制出并行探索分支 · 支持"如果我当时…"式假设推演

Agent Loop 请求带 SESSION_ID → 从远端拉取过往事件 → 打包 context → 调大模型

SESSION AS DATA PLATFORM

Session 上云之后，还能做什么？

事件流一旦沉淀到远端，就不只是"聊天记录" — 它会变成新的数据源

CLOUD SESSION

事件流

Agent · LLM · Tool · User
的所有交互事件

01 USE CASE

公共记忆抽取

MEMORY EXTRACTION

后台记忆抽取引擎读取 Session 事件，抽取"这个用户是谁、爱好什么、约束是什么"，写入 Memory Store，**跨 Session 共享上下文。**

Async Job

02 USE CASE

审计追踪

AUDIT TRAIL

工具调用事件同步进审计系统 — 谁在什么时候，让 Agent 调用了哪个工具、返回了什么，全部可回溯，**满足合规与安全要求。**

Compliance

03 USE CASE

事件触发器

EVENT TRIGGER

Session 事件流可被订阅，变成其他系统的输入源 — "Agent 一旦调用了 X 工具，就通知运维值班"，**从数据到反应。**

Sub / Push

Session 上云 = 从"Agent 私有的记忆"变成"平台级数据资产" — 是存算分离带来的第一个红利。

状态二：执行环境沙箱 — 让工具在远端执行

> **工具调用在哪个沙箱，Agent 本质上就运行在哪里**
> 大模型是通过工具调用感知环境的 — 环境沙箱是 Agent 状态最厚重的一部分

01 暂停恢复

秒级恢复不占资源

大部分对话不需要工具执行，
沙箱可以暂停不占计算资源，
恢复后**内存 · 临时数据都在**。

Pause / Resume

02 快照回滚

全盘状态可回退

支持任意点全盘快照 —
Agent 装错了包 · 改坏了配置
秒级回滚到干净状态。

Snapshot / Rollback

03 绑定语义

与 Session / User 绑定

同一 User 跨 Session 对话，
感知到**同一环境** —
环境是记忆的物理容器。

Bind: Session / User

Agent Loop 变成无状态服务，就够了吗？

状态都抽出去后，Loop 是无状态的 → 直接当 Deployment 用 K8s 部署 · 前面挂 LB 负载均衡

已解决 · SOLVED

水平扩展

Agent Loop 无状态 → 想加多少副本加多少

支持突发流量，性能瓶颈消失

单点故障隔离

某个 Loop 副本挂了，会话可切到其他副本

用户不再感知"某台机器坏了"

升级不丢数据

状态在远端 → Loop 副本随便滚动升级

再也不用担心"改环境要求用户重启"

待解决 · REMAINING

Session 排他性

同一 Session 必须由单一 Loop 独占，否则内容混乱

→ 需要 Session 抢主 · 分布式锁语义

K8s 集群运维负担

用户仍要管理 Deployment · HPA · Ingress · 监控告警

→ 每个 Agent 团队都要一位 SRE

开源框架需改造

主流开源 Agent 都是本地状态假设 · 不能直接跑

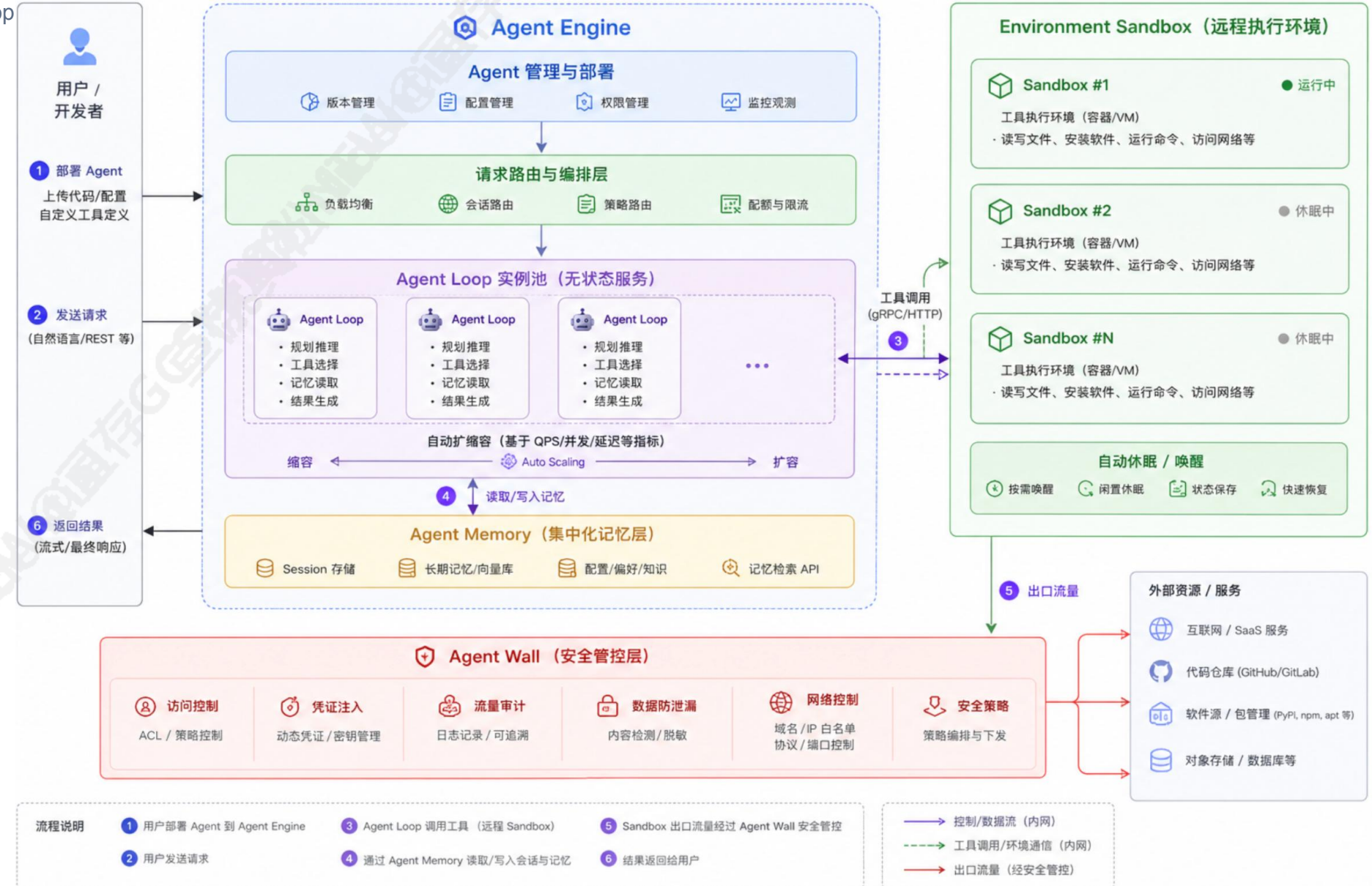
→ Runtime 提供云上 Session · 远程工具 SDK 兼容

下一步

用**函数式 · 全托管**把剩余的运维负担推给平台，进一步降低用户心智

Agent Engine 核心架构

托管你的Agent loop



Agent Engine - 函数式托管：用户只写 loop() 主函数

Session、环境、扩缩容、生命周期全部由平台框架代管 — 用户回到"只写业务逻辑"

agent_loop.py — user code (only this)

```python
from agent_runtime import agent

@agent loop (name="code_reviewer" )
def run (ctx):
    # 只写 loop 主逻辑
    while not ctx.done:
        msg = ctx.llm(ctx.history)
        if msg.tool_call:
            result = ctx.tool.call(msg.tool_call)
            ctx.append(result)
        else :
            return msg.text
```

USER'S FULL CODE

用户交付的就这一段 · 没有 Deployment / HPA / 抢主锁 / 存储挂载

平台框架代管的 5 件事

FRAMEWORK-MANAGED CAPABILITIES

01 **Session 生命周期**
SESSION_ID 路由 · 抢主锁 · 事件持久化

02 **环境沙箱调度**
ctx.tool.call() 自动路由到用户绑定的 Sandbox

03 **扩缩容 & 冷启动**
按请求粒度弹性 · 空闲缩到 0 · 秒级唤醒

04 **可观测 & 灰度**
日志 / 追踪 / 指标 / 版本灰度全部开箱即用

05 **部署编排**
git push 触发 · 无 K8s / 无 Dockerfile / 无 YAML

从写整个应用 → 只写一个函数 · Agent Loop 的开发运维模型，本质上被"Serverless 函数化"了

Agent Engine - 全托管：用户不再写 Loop

大部分场景 ReAct 通用循环够用 — 用户只声明 System Prompt / Skills / MCP，Loop 由平台内置

agent.yaml — declarative only

```yaml
apiVersion: agent/v1
kind: Agent
metadata:
  name: code-reviewer
spec:
  systemPrompt:
    |-
    你是一个代码审查专家，
    找出潜在缺陷并给出建议。
  skills:
    - code_review
    - git_read
  mcp:
    - gitlab-mcp://internal
```

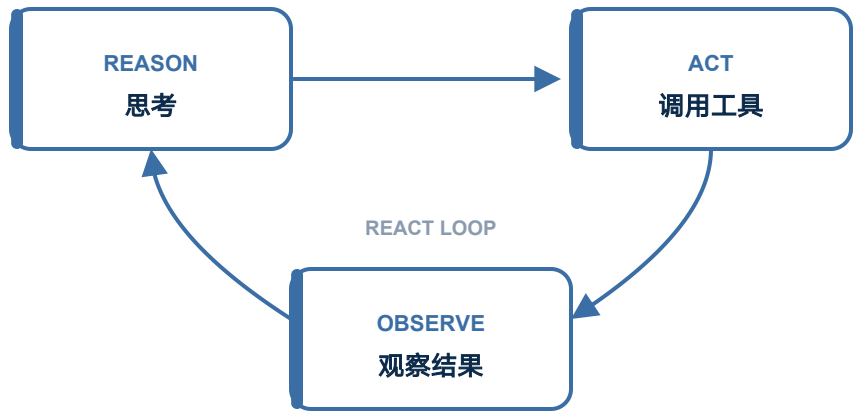
DEPLOY

一键发布

平台内置的 ReAct 循环

LOCKED

BUILT-IN LOOP · YOU DON'T WRITE THIS



- Prompt 直接映射到 Reason 阶段的角色约束
- Skills / MCP 声明 → 自动填充 Act 阶段可用工具集
- 循环控制 / 上下文压缩 / 错误重试 全部平台代管

Agent 从"应用"回到"配置"

托管度演进：从单体到全托管· 四阶阶梯

越往上走，用户负担越轻，标准化程度越高 — 后两阶（③ ④）我们接下来分别展开

Claude Code/Openclaw

元宝/Manus

AWS AgentCore Runtime

Anthropic Claude Managed Agents

CHAPTER 03

03

从分布式 Agent 到多 Agent 协作 Registry & A2A

单个 Agent 分布式化只是起点，Agent 之间还需要协作

FROM DISTRIBUTED AGENT TO MULTI-AGENT

1 Agent 互相发现

2 Agent 互相调用

3 上下文共享

Agent 之间要协作，至少需要三种基础能力

这三种能力对应了三个物理层：**发现层 · 调用层 · 协作层**

01

Agent 互相发现

DISCOVERY

解决"我要用一个能
写代码的 Agent，
他在哪？"

类似传统微服务的
服务发现，
但索引维度更丰富。

02

Agent 互相调用

A2A INVOCATION

解决"Agent A
怎样把任务交给
Agent B？"

Agent 也是 Tool —
调用语义与 MCP 统一，
复用同一套 SDK。

03

跨 Agent 上下文共享

CONTEXT SHARING

解决"多个 Agent
如何在同一件事上
不重复对齐？"

STILL EVOLVING

目前多复用 Git Issue
或已有的 Human 协作平台

→ **为此我们引入 Agent Registry — 把①②统一在一个组件里，让「协作」有一个物理落点。**

Agent Registry：可发现 · 可调用 · 可扩展 的中枢

留白 · 上下文共享

目前尚未提供上下文共享基础设施 — 用户仍复用 Git Issue 或其他 Human 2 Human 协作平台，是下一代基础设施的空白。

腾讯云 Agent Runtime

CLOSING · THANK YOU

从工作流到底座 —
Agent 架构范式变革，正在发生

01 路径规范
单体自管 → 分布式托管，是必经之路

02 物理层规范
存算分离 + 沙箱化，是最小公倍数

03 协作层规范
Registry + A2A，是多 Agent 时代的第一块基石

感谢大家

Q & A