Agenteam——面向团队的集中式智能体系统探索与实践

组会汇报 / Agent 架构、推理 API 演进与系统设计

报告整理版

集中式 Agent

工作宽度

批量多线程

汇报内容基于内部探索报告整理

1. 背景与动机

为什么需要面向团队的集中式 Agent 服务？

教学项目：几百个学生 + Web Coding + Agent 支持

几百个学生

+

Web Coding

+

Agent 支持

01

API Key 泄露风险：
本地运行导致
Token 使用范围不可控

02

执行过程无法监控：
难以分析、监控与
评估学生实验过程

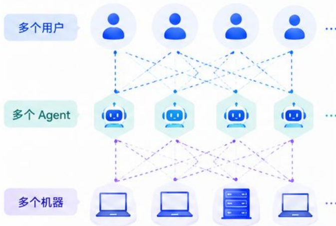
03

知识无法共享，Token 浪费：
相同问题被重复推理

目标：集中式 Agent 服务

—— 让学生电脑主动接入我们统一的 Agent 服务

核心能力：多对多连接与调度



核心诉求：多对多 + 多进程多机器接入

2. 技术原理：大模型推理 API 的演进

从文本续写到结构化消息，再到有状态会话

维度	Chat API（旧）	Message API（当前主流）	Conversation API（新兴）
结构	纯文本 Prompt/Completion	结构化消息（角色 + 内容 + 可选工具调用）	结构化输入，服务端管理会话
状态	无状态（每次请求都需带完整上下文）	无状态（由客户端管理上下文）	有状态（服务端维护会话历史）
上下文控制	完全由客户端拼接，易出错	**精细可控**（可灵活增删、截断、重排消息）	不透明（自动管理，控制粒度较低）
Token效率	低（重复传递历史文本）	中等（可裁剪与压缩，按需传递）	高（增量式，服务端优化）
Agent适用性	不推荐 ⊗	✔ 强烈推荐（最灵活、可控，生态完善）	⚠ 按场景评估（便捷高效，但灵活性有限）

结构化消息是构建稳定、可控 Agent 的最佳实践

2.1 Message API 与 Tool Call 机制

当前主流 Agent 的核心接口形态

请求（Request）结构示例

```
{
  "model": "gpt-4o",
  "messages": [
    { "role": "system",
      "content": "你是一个智能运维助手，擅长命令执行与故障排查。"},
    },
    { "role": "user",
      "content": "请帮我查看 /var/log 下的错误日志，并总结最近的错误。"},
    },
    { "role": "assistant",
      "tontent": null,   "tool_calls": [ /* 上一轮调用产生的 tool_calls (可选) */ },
    },
    { "role": "user",
      "content": "好的，请继续。"},
  ],
  "tools": [  / * 见右侧工具定义示例 * /   ],
  "tool_choice": "auto"
}
```

响应（Response）结构示例

```
{
  "id": "chatcmpl-1234567890",
  "object": "chat.completion",
  "choices": [
    {
      "index": 0,
      "message": { "role": "assistant", "content": null, "tool_calls": [/* 见右侧流程说明 */ },
      "finish_reason": "tool_calls"
    },
  ],
  "usage": { "prompt_tokens": 512, "completion_tokens": 128, "total_tokens": 640}
}
```

执行流程（以一次工具调用为例）

角色（role）说明

角色	说明
system	系统指令，设定 Agent 的身份、能力与约束（可多条，通常放在最前）
user	用户输入，提出问题或提供上下文
assistant	模型生成的回复；当需要调用工具时，content 可能为 null，且包含 tool_calls
tool	工具执行结果，必须通过 tool_call_id 与对应的 tool_calls 关联

工具定义（tools）示例（节选）

```
{
  "tools": [
    {
      "type": "function",
      "function": { "name": "execute_command",
      "description": "在主机上执行命令并返回结果",
      "parameters": { "type": "object",
        "properties": { "command": { "type": "string" } },
        "required": ["command"] } }
    },
     "type": "function", "function": { "name": "talk_to_user",
     "description": "向用户询问澄清信息", "parameters": {
     "type": "object", "properties": { "message": { "type": "string" }},
     "required": ["message"] } } }
  ] ]
}
```

关键要点

- 当 assistant 需要调用工具时，content 通常为 null，而通过 tool_calls 字段返回要调用的工具列表。
- 每个 tool_call 包含唯一的 id（如 call_abc123），用于在后续工具结果中进行关联。
- 工具返回结果必须以 role: tool 的消息携带，并通过 tool_call_id 与对应的 tool_call 绑定。

2.2 实践经验：为什么推荐 Message API

工具调用的关键规则与 Conversation API 的取舍

01 每个 tool_call 只能回复一次

02 assistant 与 tool 之间不要插入 user 消息

03 tool 和 assistant 之间最好也不要插入其它消息

04 多个 tool_call 最好并行执行，等待全部完成后统一回传

assistant (tool_calls A,B) tool (A) tool (B) assistant

Conversation API 为什么不推荐用于 Agent

01 上下文由服务端管理，Agent 失去完全掌控力

02 上下文压缩/截断策略不可见，跨厂商行为不稳定

结论： 对需要精确控制推理上下文的 Agent，**无状态的 Message API 更安全、更可控。**

3.1 整体架构与设计约束

三段式架构：思考、管理、执行

系统组成

Web 服务
FastAPI

Agent 管理模块
Python asyncio

前端
React+Vite

通信模型
Pydantic

设计约束

MVP 最小化
先跑通核心链路，快速验证价值

安全性：全链路可审计
指令、操作、结果与环境事件全量留痕

经济性：面向团队节省 Token
上下文压缩、记忆分层、按需摘要与复用

异步优先：外部操作都可能很慢
全链路异步与超时控制，保障系统可用性

3.2～3.4 核心设计思想

响应式 Agent、工作宽度与语义异步操作

1 事件驱动（Event-Driven）模型

$$S' = E(S, \text{event})$$

S：当前状态（State） S'：新状态 E：转移函数 event：事件

事件（event）包括：

用户消息

来自用户的输入、指令、反馈等

异步操作回调

工具或外部服务完成后的回调事件、通知等

Agent 不主动轮询世界，而是被事件唤醒、驱动决策与行动。

2 工作深度 vs 工作宽度

工作深度（Depth）

能处理多久之前的信息（上下文长度）

VS

工作宽度（Width）

同时能做多少件事，并能统筹彼此影响

多 Agent 并行 ≠ 工作宽度

并行执行 ≠ 全局统筹与影响管理

3 同步 vs 异步的工具设计

同步工具（阻塞）

```
run_command(cmd)
# 启动并等待结果
# 阻塞直到完成
```

异步工具（非阻塞）

```
start_command(cmd)   # 启动
stop_command(id)     # 取消
wait_command(id)     # 等待
```

★ 关键洞察

从语义上，Agent 对世界的所有操作都应是可中断的异步操作；只有这样，Agent 才保留选择取消、打断和切换任务的能力。

汇报内容基于内部探索报告整理

3.5～3.8 三代 Agent 架构演进

从串行锁定到批量多线程

对比维度	SerialLockedAgent	BatchLockedAgent	BatchMultiThreadAgent
事件处理	逐个事件，串行处理	事件入队，窗口聚合后处理	事件批量，线程并发处理
并发能力	✗ 无并发	✗ 无并发	✓ 多线程并发
批处理	✗ 无	✓ 有（窗口聚合）	✓ 有（批量 + 并发分发）
中断机制	✗ 无	○ 有限（仅在等待窗口）	✓ 有（信号中断：外部/内部）
抢占机制	✗ 无	✗ 无	✓ 有（支持抢占与调度）
Token效率	✗ 低（多次推理，重复上下文）	○ 较高（减少推理次数）	✓ 最高（并发复用上下文）
复杂度	✓ 低（实现简单）	○ 中（需队列与窗口管理）	✗ 高（线程/同步/一致性管理复杂）

3.7 BatchMultiThreadAgent 深入展开

线程模型、中断/抢占与工具分层

A 线程模型（Thread Model）

线程对象结构（Thread）

字段	说明
id	线程唯一标识
start	创建时间（timestamp）
active	是否激活（bool）
intent	线程目标/意图（str）
status	运行状态（枚举）
session	会话/上下文引用（id）
operations	待执行/已执行操作队列
interruptions	中断请求队列（FIFO）
preemptions	抢占请求队列（FIFO）
termination	终止标志与原因（可选）

状态枚举（status）

B 线程生命周期与四条件循环

线程生命周期（Life-cycle）

四条件循环（核心调度逻辑）

```
01  while True:
02      # 1. 中断优先
03      if interruptions:
04          handle_interruptions()
06          continue
05      # 2. 未完成操作则等待
06      if has_unfinished_operations():
07          wait_for_operations()
09          continue
09      # 3. 抢占处理
10      if preemptions:
13          execute_preemptions()
13          continue
11      # 4. 终止判断
13      if termination:
13          break
12      # 5. 否则推理下一步动作
12      infer_next_action()
```

四条件优先级（高→低）

1. **中断优先** 有中断请求则立即处理
2. **操作完成约束** 有未完成操作则等待
3. **抢占处理** 有抢占请求则执行抢占
4. **终止判断** 满足终止条件则退出
5. **推理下一步** 无约束则推理下一步动作

C 工具分层与中断/抢占流程

工具分层（Tool Hierarchy）

事件处理上下文（Event Context）

线程上下文（Thread Context）

+ 所有 Operators（业务操作集合）

中断处理上下文（Interruption Context）

中断/抢占流程示例（“等等，改成归并排序”）

→ 事件流 → 中断流 → 取消流 → 抢占流 → 控制流

设计要点： 中断优先 > 操作完成约束 > 抢占处理 > 终止判断 > 推理下一步，保障系统响应性、一致性与可控性。

4. 探索发现 & 5. 总结反思

已有相似工作、我们的定位与后续方向

相关方向概览

企业/开源平台
ADP Agent Portal、Ultron、AutoGPT、·BeeAI

多智能体框架
CrewAI、AutoGen / MAF、Houmao

研究探索
OpenAgents、JiuwenSwarm

我们的差异化定位

1. 教育 / 科研场景
2. Token经济性
3. 响应式/事件驱动
4. 工作宽度
5. 对推理上下文的完全掌控

值得继续探索

Shared Knowledge Base

多 Agent 协作树

动态模型切换

Agent 分支与回溯

Jupyter 操作环境

结论： 两周内完成了一个可运行原型；工程价值显著，但更值得深挖的是工作宽度、记忆组织与跨 Agent 知识共享等科研问题。