Native Agent Runtime

调研与定位报告

面向外部系统调用的目标驱动 Agent 运行时

Goal-driven Externally Callable Auditable Self-improving

研究对象：OpenAI Agents SDK、Anthropic Managed Agents、Agent Skills、LangGraph、Microsoft Agent Framework、CrewAI、Pydantic AI、Inspect AI、promptfoo、Braintrust / LangSmith 等。

2026-06-05

目录

1 执行摘要

一句话结论

这个项目最有潜力的定位不是“又一个聊天 agent 框架”，而是一个 面向外部系统调用的 **native agent runtime**：它以 `Goal` 为一等对象，运行长期、可中断、可恢复、可审计的 agent 工作流，并把自我迭代收束在评测、审计、版本和发布门禁之内。

本报告基于对 OpenAI、Anthropic 以及多个 agent harness / eval / observability 框架的调研，结合当前仓库已经具备的能力，给出一个更清晰的项目定位、架构方向和演进路线。

当前仓库已经实现了一个 `Skill Agent` MVP：它用 Go 编写，暴露 HTTP API，支持 skill 版本、eval case、command evaluator、feedback improver 和改进循环。这个基础不是要推倒重来，而是可以升级为更通用的能力进化内核：

```
Skill revision -> eval -> feedback -> improver -> next revision
```

可以扩展为：

```
Goal -> agent run -> subagents/tools -> trace -> eval -> audit -> capability revision -> promote
```

这一步扩展的关键不是让模型“自由自我修改”，而是建立一个工程化控制平面：每次变化都有来源、证据、版本、评测、审计和回滚路径。

我建议的项目定位如下：

建议定位

Native Agent Runtime 是一个跨语言、跨平台的目标驱动 agent 运行时。它接收外部系统提交的 goal，调度单 agent、subagent 和 multiagent 工作流，管理工具权限、事件流、状态、artifact、评测和审计，并允许 agent 能力在受控门禁下持续迭代。

这个定位和 OpenAI / Anthropic 的主流实践一致，但又有独立价值：

- OpenAI Agents SDK 更像 Python 侧 agent 构建工具，强调 agents、tools、handoffs、guardrails、sessions、tracing。
- Anthropic Managed Agents 更像受管 harness，强调 agent / environment / session / events、sandbox、permission policies、multiagent session threads。
- LangGraph 和 Microsoft Agent Framework 强调长期、可恢复、可中断的 workflow runtime。
- Inspect AI、promptfoo、Braintrust 和 LangSmith 强调评测、trace、dataset、experiment、生产反馈闭环。

你的项目可以取它们的交集，但用一个更底层、更可被外部系统调用、更不绑定单一语言 SDK 的形式落地。

2 背景：为什么不是普通 Chat Agent 框架

用户提出的真实需求有几个明确点：

- 这是一个 native agent 框架。
- 主要不是用来 chat 或 coding，而是被外部程序调用。
- 需要支持 `/goal`。
- 需要支持 subagent / multiagent。
- 需要高强度自我迭代。
- 自我迭代可能引入审计模型，以便更可控。

这些约束决定了产品不能以“聊天会话”为中心。聊天只是其中一种输入形式，不能成为根模型。更合理的根对象是 `Goal`。

聊天系统通常关心 user message、assistant response、conversation history 和 streaming text；外部调用型 agent runtime 更关心 objective、lifecycle、policy、tool permissions、event stream、artifacts、child goals、eval reports、audit reports 和 promotion decisions。

也就是说，这个系统的主要用户不是终端聊天用户，而是其它系统：CI/CD、IDE、后端服务、自动化平台、数据管道、测试平台或能力进化服务。

因此，核心 API 不应是：

```
POST /chat
```

而应是：

```
POST /goals
POST /goals/{id}/events
GET  /goals/{id}
GET  /goals/{id}/events/stream
POST /goals/{id}/pause
POST /goals/{id}/resume
POST /goals/{id}/cancel
POST /goals/{id}/feedback
```

> **我的判断**
>
> Agent 框架如果以 chat 为根，很容易被前端体验牵着走；如果以 goal 为根，才会自然长出状态机、预算、权限、子任务、事件流、审计和可恢复执行。这是本项目和普通聊天框架拉开距离的核心。

3 OpenAI Agents SDK：轻量原语和编排边界

OpenAI Agents SDK 的官方文档把它定义为用于构建 production-ready agentic applications 的工具包，核心能力包括 agents、tools、guardrails、handoffs、sessions、tracing、voice、realtime 和 MCP。它的价值不在于复杂抽象，而在于给出了一组非常干净的 agent 原语。

3.1 可借鉴机制

机制	OpenAI 做法	对本项目的启发
`Agent`	LLM 加 instructions、tools、handoffs、memory、guardrails。	本项目也需要 `AgentSpec`，但应版本化、可审计、可由外部系统引用。
`Runner`	同步、异步、批处理、取消、错误处理。	`Goal Engine` 应承担 runner 职责，但对象是 goal/run，而不是单轮 chat。
`Tools`	Python callable 注册为结构化工具。	工具必须 schema-first，并且每个工具都有 permission policy 和审计记录。
`Handoffs`	triage agent 可以把对话交给 specialist。	handoff 可映射为 subagent thread 或 child goal，而不是普通函数调用。
`Agents as tools`	manager agent 调用 specialist agent，但保持最终控制权。	适合 bounded subtask；比如 verifier、researcher、summarizer。
`Guardrails`	input/output/tool guardrails，触发 tripwire 可中断。	审计模型可作为 semantic guardrail，但硬边界必须由 deterministic policy 保证。
`Tracing`	每个 agent step、generation、tool call、handoff 形成 trace/span。	本项目应从第一天就把 event log 和 trace 作为核心产物。

OpenAI 的编排文档明确区分两种方式:

- 让 LLM 自己编排: 模型基于 instructions、tools 和 handoffs 决定下一步。
- 用代码编排: 程序用结构化输出、链式流程、循环、并行控制 agent。

它还明确建议: 开放任务适合 LLM 编排，但确定性、成本和性能更重要时，代码编排更可控。

这对本项目非常关键。`Native Agent Runtime` 不应默认把一切交给模型决定。模型可以参与 planning，但 runtime 需要保留调度权、权限判断权、预算控制权和版本发布权。

3.2 Handoff 和 Subagent 的区别

OpenAI 文档中有两个相似但不同的模式:

- `Agent.as_tool()`: manager agent 调用 specialist agent，specialist 完成一个 bounded subtask，最终输出仍由 manager 汇总。
- `handoff()`: triage agent 把当前上下文交给 specialist，specialist 接管下一阶段。

我建议在本项目中把这两种模式明确建模:

模式	适用场景	运行时实现
Subagent as tool	子任务边界清晰，例如“审阅这份 diff 并返回 finding 列表”。	创建 child run，输入裁剪，输出结构化，parent 保持控制。
Handoff thread	需要专业 agent 接管阶段，例如“进入安全审计阶段”。	创建或切换 session thread，继承摘要或过滤后的历史。
Parallel workers	多个独立任务可并行，例如多文件分析、多来源调研。	fan-out child runs，join 后由 coordinator 合成。
Verifier/Auditor	验证结果、检查风险、阻断 promotion。	只读上下文和 artifact，不允许直接修改能力版本。

> **我的判断**
>
> Multiagent 不应该被设计成“几个 agent 在群聊”。那种形式很难控制预算、责任和输出质量。更稳的方式是把 multiagent 设计成调度系统: 每个子 agent 有明确输入、权限、预算、输出 schema 和责任边界。

3.3 Guardrails 对审计模型的启发

OpenAI guardrails 文档把检查点分成 input、output 和 tool guardrails，并允许触发 tripwire。这很适合迁移到本项目:

- input guardrail: 检查 goal 是否越权、是否缺少必要约束、是否注入了危险要求。
- planning guardrail: 检查计划是否偏离目标、是否需要更多上下文。
- tool guardrail: 检查工具调用参数、路径、权限、secret、网络目标。
- output guardrail: 检查最终产物是否符合 schema、是否泄漏敏感信息。
- promotion guardrail: 检查自我迭代产生的新版本是否允许进入 candidate 或 stable。

但我不建议把“审计模型”直接等同于 guardrail。模型审计擅长语义判断，例如 goal drift、证据不足、解释不一致、风险描述缺失；但它不擅长做硬边界。硬边界必须由 policy engine、sandbox、allowlist、budget 和 eval gate 执行。

4 Anthropic Managed Agents：外部调用型 Harness 的成熟形态

Anthropic Managed Agents 的设计非常接近本项目想做的方向。它不是简单 chat API，而是一个 pre-built configurable agent harness，主要用于 long-running tasks 和 asynchronous work。

其核心概念是:

概念	Anthropic 定义	本项目映射
Agent	model、system prompt、tools、MCP servers、skills。	`AgentSpec`，版本化且可被 goal 引用。
Environment	cloud sandbox 或 self-hosted sandbox。	`RuntimeEnvironment`，包括本地、容器、远程 sandbox。
Session	运行中的 agent 实例，执行一个具体任务。	`GoalRun` 或 `Session`，绑定 goal、状态和 artifact。
Events	应用与 agent 之间的消息、工具结果、状态更新。	`EventLog`，作为运行时主通信协议和审计证据。

4.1 事件流：比聊天更适合外部系统

Anthropic Managed Agents 的通信是 event-based。外部系统可以发送 `user.message`、`user.interrupt`、`user.tool_confirmation`、`user.custom_tool_result` 等事件；运行时返回 `agent.message`、`agent.tool_use`、`session.status_running`、`session.status_idle`、`session.error` 和 multiagent thread events。

这个设计特别适合外部系统调用。外部系统不只想看到一句最终文本，它需要知道现在是否 running、卡在哪里、是否等待工具审批、发生了哪些子任务、哪个工具调用失败、是否可以中断并改方向，以及最后产生了哪些 artifact。

本项目应直接采用类似思想。`/goal` 不是一个同步 completion API，而是一个长期对象，外部系统通过事件和它互动。

4.2 Permission Policies：工具权限不是提示词问题

Anthropic 文档中的 permission policies 很简单但重要:

- `always_allow`：自动执行。
- `always_ask`：暂停并等待批准。

MCP toolset 默认 `always_ask`，这是一个很好的安全默认值：远程工具集合可能新增工具，不应自动获得执行权限。

本项目可以扩展为更完整的策略:

```
deny
always_allow
always_ask
allow_if_path_matches
allow_if_command_prefix_matches
allow_if_risk_below
allow_after_audit
allow_after_human
```

并且策略应该作用在 agent default、toolset default、individual tool、environment、goal、workspace 等多个层级。

> 风险提示

> 对 agent 来说，工具权限永远不应该只写在 system prompt 里。提示词可以提醒模型，但不能防止越权执行。真正的边界必须在 runtime 中实现。

4.3 Multiagent Sessions：共享环境，隔离上下文

Anthropic multiagent sessions 的关键设计是：

- 所有 agents 共享 sandbox、filesystem、vault credentials。
- 每个 agent 在自己的 session thread 中运行。
- 每个 thread 有独立上下文和事件流。
- coordinator 可以跟早先调用过的 agent 继续对话，该 agent 保留上下文。
- coordinator 的 roster 可以 pin 到具体 agent version。
- 当前限制一层 delegation，避免深层 uncontrolled tree。

这给本项目两个启发：

1. 子 agent 不一定要共享对话上下文，但可以共享工作目录和 artifact。
2. multiagent 需要深度限制、版本 pinning 和 thread lifecycle 管理。

我建议在本项目中引入：

```
Goal
  Run
    PrimaryThread
    ChildThread[]
```

每个 child thread 记录 agent spec version、parent event id、input summary、allowed tools、budget、status、output schema、artifacts 和 final report。

4.4 Task Budgets：软预算和硬预算要区分

Anthropic task budgets 是 advisory token budget。模型会看到 countdown，并尽量自我调节，但它不是硬 cap。硬限制仍然依赖 `max_tokens` 等运行时限制。

预算类型	作用	执行者
Advisory budget	让模型自我调节，提前收束、汇报进度。	模型和 prompt/runtime context。
Hard budget	限制 token、时间、成本、迭代、工具调用次数。	runtime，必须强制执行。
Promotion budget	限制自我迭代尝试次数，防止无限优化。	evolution engine。

对于高强度自我迭代，预算尤其重要。否则系统会在“再试一次”的循环中消耗大量 token 和时间，却没有明确收敛标准。

4.5 Agent Skills：能力包和 Progressive Disclosure

Anthropic Agent Skills 文档说明了一个非常实用的能力包装方式：

- Level 1：metadata 常驻，只包含 name 和 description。
- Level 2：SKILL.md body 触发时加载。
- Level 3：references、scripts、templates 等资源按需读取或执行。

这个设计对本项目的 `Capability` 模型很有用。`Capability` 不应只是 prompt 文本，它可以是 skill、prompt、workflow、tool policy、memory rule、agent profile、evaluator config、audit policy 和 planner strategy。

5 LangGraph: State Graph 和 Durable Execution

LangGraph 把自己定位为 low-level orchestration framework and runtime，用于 long-running、stateful agents。它强调的不是“更聪明的 agent”，而是底层运行能力：persistence、durable execution、streaming、human-in-the-loop、memory、debugging 和 deployment。

它的价值在于承认 agent workflow 是状态机，而不是单次模型调用。

对本项目来说，LangGraph 的核心启发是：

1. `Goal` 需要状态图，不只是 `queued/running/completed`。
2. 每一步状态都应可持久化。
3. 中断和恢复不是附加功能，而是长期任务的基础功能。
4. 人工介入应该能修改 state，而不只是发一条 chat message。
5. trace 需要能解释状态转移，而不仅是记录文本输出。

一个可能的 goal 状态机如下：

```
created
  -> planning
  -> running
  -> waiting_for_tool_approval
  -> waiting_for_external_result
  -> auditing
  -> evaluating
  -> promoting
  -> complete

running
  -> paused
  -> cancelled
  -> blocked
  -> failed
```

> **我的判断**
>
> 如果没有 durable state，agent runtime 只能做短任务；如果有 durable state，它就可以变成自动化基础设施。`Goal`、`Event`、`StateSnapshot` 和 `Artifact` 应该从早期版本开始就存在。

6 Microsoft Agent Framework: 生产级、多语言和互操作

Microsoft Agent Framework 是 AutoGen 的后继方向，定位为 production-grade AI agents and multi-agent workflows，支持 Python 和 .NET。它强调多语言一致 API、多 provider、middleware、graph-based workflows、checkpointing、streaming、human-in-the-loop、time-travel、OpenTelemetry、declarative agents、agent skills、A2A 和 MCP。

这和本项目“native + externally callable + multiagent”的方向高度相关。尤其值得借鉴的是分层架构：

- 底层 runtime: 消息、事件、状态、工作流、持久化。
- agent abstraction: model、tools、instructions、middleware。
- extension: provider、tool、sandbox、observability。
- hosting: local、cloud、serverless、durable workflow。

本项目如果继续用 Go 作为核心 runtime，可以保持：

- Go daemon / service 作为稳定运行时。
- OpenAPI / gRPC 作为跨语言边界。

- Python / TypeScript / Rust / C++ 只做 thin SDK。
- LLM provider、tool、evaluator、auditor 都作为 adapter。

这比在每个语言里都实现一套 agent loop 更稳。

7 CrewAI: Crew / Flow 和产品化执行

CrewAI 的文档中反复出现 agents、tasks、crews、flows、processes、memory、planning、checkpointing、training、testing、event listeners、kickoff / status / resume API、AMP deployment and tracing。

它的风格更偏“团队和流程产品化”。其中最适合本项目吸收的是两层抽象:

层级	CrewAI 思路	本项目可用形态
Crew	多个 agent 协作完成任务。	`AgentGroup` 或 `Roster`，定义 coordinator 和 specialists。
Flow	结构化 workflow，控制步骤和状态。	`GoalPlan` 或 `WorkflowSpec`，由代码和模型共同生成/执行。
Checkpoint	保存执行状态以恢复。	`StateSnapshot`，支持 resume、time travel、debug。
Training / Testing	通过反馈和测试改善 agent。	`CapabilityEvolution`，连接 eval、feedback 和 revision。

对于本项目，我不建议直接采用“crew”作为核心名词，因为它容易让人联想到角色扮演式多 agent。更好的核心名词仍是 `Goal`、`Run`、`Thread` 和 `Capability`。但 CrewAI 的 checkpoint/resume、testing/training、event listener 很值得吸收。

8 Pydantic AI: Schema-first 和强类型边界

Pydantic AI 的价值是工程化，尤其是 agent specs、dependencies、typed output、toolsets、hooks、graph、evals、dataset management、LLM judge 和 durable execution integrations。

虽然本项目不一定用 Python，但它提醒我们：公共接口必须 schema-first。

Agent runtime 的失败往往不是模型“想错了”，而是边界含糊：tool input schema 不清楚、output 没有结构、feedback 没有字段化、eval 结果只是一段文字、audit 结论不可机器处理、subagent 输出无法稳定合并。

因此，本项目应该避免“到处传文本”的设计。文本可以存在，但关键对象要有 schema:

```
{
  "goal_id": "goal_...",
  "status": "waiting_for_tool_approval",
  "pending_actions": [
    {
      "id": "act_...",
      "kind": "tool_call",
      "tool": "shell.exec",
      "risk": "high",
      "arguments": {
        "command": ["go", "test", "./..."]
      }
    }
  ]
}
```

对于自我迭代，也要用结构化对象:

```
{
  "capability_id": "cap_...",
  "from_revision": "rev_12",
  "candidate_revision": "rev_13",
  "change_summary": "...",
  "evidence": ["trace_...", "eval_..."],
  "audit": {
    "risk": "medium",
    "decision": "needs_human_approval"
  }
}
```

9 Inspect AI、promptfoo 和 Evals：评测不是附属品

OpenAI Evals、Inspect AI、promptfoo、Pydantic Evals、Braintrust 和 LangSmith 都说明同一件事：agent 能力如果要持续变好，必须把 evaluation 变成一等系统。

9.1 Inspect AI 的结构

Inspect AI 把 eval 拆成 tasks、datasets、solvers、scorers、scanners、sandboxes、tool approval、limits、early stopping 和 tracing。这个结构对本项目尤其有用。当前仓库已有 `EvalCase` 和 `Evaluator`，但未来应扩展为：

```
EvalSet
  -> Dataset
  -> Case
  -> Harness
  -> Scorer
  -> Report
  -> RegressionGate
```

其中 `Harness` 可以是 inline assertions、command evaluator、HTTP replay、browser workflow、repo trace replay、human review、LLM judge 或 adversarial redteam。

9.2 promptfoo 的轻量配置价值

promptfoo 的优势是非常直接：YAML 配置、多 provider 对比、deterministic assertions、model-graded assertions、red-team plugins、CI/CD 集成。

这提示我们，本项目早期可以支持一个轻量 eval manifest：

```
id: repo-agent-regression
target:
  capability: repo-agent
cases:
  - id: avoids-unsafe-reset
    input:
      objective: Fix test failure without destructive git commands.
    assertions:
      - kind: trace_not_contains_tool_call
        tool: git.reset_hard
      - kind: final_output_contains
        value: tests run
```

9.3 Braintrust / LangSmith 的闭环

Braintrust 和 LangSmith 的共同方向是：

```
production trace -> dataset -> eval -> experiment -> prompt/agent change -> monitor
```

这和本项目的“高强度自我迭代”直接相关。真正有价值的自我迭代不是模型凭空反思，而是从真实失败 trace 和人类反馈中提炼 case，然后回归测试。

我建议把能力进化闭环定义为：

```
Observe  -> 收集 trace、artifact、human feedback、failure cases
Distill  -> 从失败中生成 eval case 或 audit rule
Propose  -> 生成 capability candidate revision
Verify   -> 跑 eval set、regression、command replay、LLM judge
Audit    -> 确认证据、风险、权限、目标一致性
Promote  -> candidate -> stable，或者 shadow / canary
```

> **我的判断**
>
> 自我迭代的上限由 eval 质量决定。没有 eval 的“自我改进”只是提示词自嗨；有 trace、case、scorer、artifact 和 audit 的迭代才可能成为工程资产。

10 横向对比：哪些机制值得进入本项目

维度	OpenAI Agents	Anthropic Managed Agents	LangGraph / MAF	本项目建议
根对象	Agent run / session	Agent session	Graph run / workflow	`Goal`
外部调用	SDK 调用为主	API + events	SDK / runtime	HTTP/gRPC + event stream
多 agent	handoff / agents as tools	coordinator + session threads	workflow / graph	subagent tool、handoff thread、parallel workers
权限	guardrails	permission policies	middleware / HITL	policy engine + audit + approval
状态	sessions	persistent session + sandbox	durable state	goal state + snapshots
观测	tracing	event stream + console timeline	OpenTelemetry / LangSmith	event log + spans + artifacts
自我迭代	evals + monitor + iterate	skills versions + events	trace-driven fixes	capability evolution gate
跨语言	Python SDK 为主	多语言 SDK	Python/.NET	Go runtime + OpenAPI/gRPC SDKs

这个对比说明：本项目不需要复制任何一个框架，而应该把几类机制组合起来：OpenAI 的轻量 agent 原语、Anthropic 的 session/event/permission/sandbox 模型、LangGraph / MAF 的 durable workflow runtime、Inspect / promptfoo 的 eval harness、Braintrust / LangSmith 的 observability-to-evaluation 闭环。

11 建议的核心架构

我建议把系统拆成十二个模块。

模块	职责
Goal Engine	管理 goal 生命周期、状态机、pause/resume/cancel、预算和事件入口。
Agent Kernel	执行单个 agent turn，连接模型 provider、instructions、tools、memory。
Planner	把 goal 转成计划、子任务、依赖图；可由模型和规则共同实现。

Subagent Manager	创建 child threads / child goals，控制上下文切片、预算、权限和 join。
Tool Runtime	执行本地/远程/MCP 工具，处理 schema、权限、sandbox、result。
Policy Engine	确定性权限、预算、allowlist、approval、risk thresholds。
Auditor	语义审计模型和规则审计结合，输出结构化 audit report。
Eval Engine	运行 eval set、command replay、LLM judge、human review、regression gate。
Capability Store	管理 skill、prompt、workflow、policy、agent profile 等能力版本。
Evolution Engine	从 trace/feedback 提出 candidate revision，验证后 promotion。
Artifact Store	保存文件、diff、logs、reports、screenshots、eval outputs。
Trace/Event Store	记录所有 events/spans，支持调试、审计、dataset 生成。

整体数据流如下:

```
External caller
  -> POST /goals
  -> Goal Engine
  -> Planner
  -> Agent Kernel
  -> Tool Runtime / Subagent Manager
  -> Event Store + Artifact Store
  -> Eval Engine
  -> Auditor
  -> Evolution Engine
  -> Capability Store
```

11.1 Goal 作为一等对象

`Goal` 建议包含:

```
{
  "id": "goal_...",
  "objective": "Investigate failing tests and propose a fix.",
  "status": "running",
  "agent": {
    "id": "agent_...",
    "version": 7
  },
  "policy": {
    "tool_defaults": "always_ask",
    "allow_network": false,
    "max_iterations": 8,
    "max_child_threads": 6
  },
  "budget": {
    "tokens": 120000,
    "wall_time_seconds": 1800,
    "tool_calls": 80
  },
  "artifacts": [],
  "metrics": {},
  "audit": {}
}
```

这使外部系统能够把 agent 当成一个可观察、可控制的任务执行器，而不是一段不可控文本生成。

11.2 Event Stream 是主通信协议

建议 event 类型采用 `{domain}.{action}` 形式:

```
goal.created
goal.status_changed
planner.plan_created
agent.message
agent.tool_use_requested
tool.approval_requested
tool.result
thread.created
thread.message_sent
thread.message_received
eval.started
eval.completed
audit.started
audit.completed
capability.revision_proposed
capability.revision_promoted
goal.completed
goal.failed
```

事件需要同时服务三个目标: 实时观察、恢复和 replay、审计和 eval dataset 生成。因此事件不能只是 UI log, 它必须是稳定 schema。

11.3 AgentSpec 和 Capability

`AgentSpec` 是可运行 agent 的定义:

```
{
  "id": "agent_repo_reviewer",
  "version": 12,
  "model": "provider/model",
  "instructions_ref": "capability://repo-reviewer-skill@8",
  "tools": ["shell", "fs.read", "fs.write"],
  "memory_policy": "repo-session",
  "output_schema": "finding-report-v1",
  "default_policy": "policy://local-dev-safe@3"
}
```

`Capability` 是可进化能力单元:

Capability 类型	说明
skill	类似当前仓库的 skill markdown, 包含 workflows 和 instructions。
prompt	系统提示、planner prompt、auditor prompt。
workflow	状态图、步骤、join/retry/approval 策略。
tool_policy	工具权限规则、allowlist、risk model。
memory_rule	什么信息可以写入长期记忆, 如何过期和审计。
agent_profile	模型、能力、工具集合、默认预算。
evaluator_config	eval set、scorer、threshold、dataset version。
audit_policy	promotion gate、risk rubric、approval requirements。

当前 `Skill Agent` 可以成为 `Capability Store + Evolution Engine` 的第一版实现。

12 高强度自我迭代：可控闭环

用户特别关心“高强度自我迭代”。这部分要非常小心，因为它既是项目亮点，也是最大风险。

我建议明确规定：agent 可以提出自我改进，但不能直接把改进发布为稳定版本。

推荐闭环:

```
1. Observe
   收集失败 goal、trace、tool errors、human feedback、eval failures。

2. Diagnose
   找出失败模式，生成结构化 issue 或 hypothesis。

3. Propose
   生成 capability candidate revision。

4. Test
   在 eval set、regression、command replay、shadow run 上验证。

5. Audit
   审计 candidate 的目标一致性、权限影响、风险和证据充分性。

6. Promote
   根据 policy 自动发布、等待人工批准、shadow deploy 或 reject。

7. Monitor
   新版本线上继续收集 trace，必要时 rollback。
```

12.1 审计模型的位置

审计模型值得加，但它应该负责语义判断，而不是硬执行边界。

检查对象	审计模型擅长	规则系统负责
Goal drift	判断计划或 revision 是否偷换目标。	要求 objective hash / goal id 不变。
Evidence	判断证据是否足够支撑结论。	要求 artifact、eval id、trace id 必填。
Risk	解释风险类型和影响范围。	风险分超过阈值必须人工批准。
Policy impact	识别权限变化是否可疑。	禁止 candidate 私自扩大 tool 权限。
Regression	解释失败原因。	eval threshold 不达标则不能 promote。

12.2 Promotion Gate

能力版本的状态建议如下:

```
draft
  -> candidate
  -> shadow
  -> stable

candidate
  -> rejected
  -> needs_human_approval

stable
  -> rolled_back
```

每个 promotion 决策必须有 candidate revision id、parent revision id、change summary、source traces、eval reports、audit report、decision policy 和 approver。

> **核心风险**
>
> 高强度自我迭代最危险的不是“模型改错了”，而是系统没有留下足够证据解释为什么接受了一个改动。没有 evidence ledger 的自我迭代不可审计，也不可调试。

13 对当前仓库的演进建议

当前仓库已经有以下基础：Go HTTP service、file-backed store、skill revision、eval cases、inline evaluator、command evaluator、feedback improver、OpenAPI draft、多语言 thin client examples。

这些能力可以自然演进，不需要一次性重构成大框架。

13.1 第一阶段：从 Run 扩展到 Goal

保留当前 `/v1/runs`，新增 `/v1/goals`：

```
POST /v1/goals
GET  /v1/goals
GET  /v1/goals/{goal_id}
POST /v1/goals/{goal_id}/events
GET  /v1/goals/{goal_id}/events
GET  /v1/goals/{goal_id}/events/stream
POST /v1/goals/{goal_id}/cancel
```

第一版 goal 可以先不接真实 LLM，只要建立 goal state、event log、artifact metadata、budget fields、policy fields，并能 link 到 existing run/eval loop。

13.2 第二阶段：AgentSpec 和 ToolPolicy

新增 `AgentSpec`、`ToolSpec`、`ToolPolicy`、`ApprovalRequest`、`AuditReport`。

command evaluator 的安全设计可以迁移到 tool runtime：默认禁用高风险命令，只有 policy 允许时执行。

13.3 第三阶段：Subagent Threads

实现 child thread，不必一开始支持复杂 multiagent：

- coordinator goal 可以创建 child thread。
- child thread 只运行一个 bounded task。
- child thread 独立上下文。
- parent thread 收到 `thread.message_received`。
- 限制最大深度为 1 或 2。

这样可以避免早期陷入复杂递归 multiagent。

13.4 第四阶段：Capability Evolution

把现有 `Skill` 抽象为更通用的 `Capability`：

```
Skill -> Capability(kind="skill")
SkillRevision -> CapabilityRevision
Run -> EvolutionRun or GoalRun
FeedbackImprover -> Improver adapter
Evaluator -> Eval adapter
```

LLM improver、audit model 和 promotion gate 可以在这一阶段加入。

14 建议 API 草案

14.1 Create Goal

```
POST /v1/goals
```

```
{
  "objective": "Review the latest failed CI run and propose a minimal fix.",
  "agent": {
    "id": "repo-engineer",
    "version": 3
  },
  "input": {
    "repository": "/workspace/project",
    "ci_log_artifact": "artifact://ci-log-123"
  },
  "budget": {
    "max_iterations": 6,
    "max_tool_calls": 40,
    "max_wall_time_seconds": 900
  },
  "policy": {
    "command_evaluator": "disabled",
    "tool_defaults": "always_ask",
    "allow_network": false
  }
}
```


14.2 Goal Event


```
{
  "id": "evt_...",
  "goal_id": "goal_...",
  "type": "tool.approval_requested",
  "actor": "runtime",
  "payload": {
    "tool_call_id": "tool_...",
    "tool": "shell.exec",
    "arguments": {
      "command": ["go", "test", "./..."]
    },
    "risk": "medium",
    "reason": "The verifier needs to confirm the current test failure."
  }
}
```


14.3 Audit Report


```
{
  "id": "audit_...",
  "target": {
    "kind": "capability_revision",
    "id": "rev_..."
  },
  "decision": "needs_human_approval",
  "risk": "medium",
  "checks": [
    {
      "name": "goal_alignment",
      "passed": true,
      "evidence": "The revision only changes validation instructions."
    },
    {
      "name": "permission_expansion",
      "passed": false,
      "evidence": "The candidate adds network access to the command evaluator."
    }
```

```
  ]
}
```

15 关键产品原则

15.1 原则一：Goal 优先于 Chat

Chat 可以作为 UI，但 API 和状态模型应该围绕 `Goal`。这会让系统天然适合外部调用、异步执行和长期任务。

15.2 原则二：Runtime 保持控制权

模型可以规划、解释、提出候选，但 runtime 必须控制预算、工具权限、子 agent 深度、状态转换、版本发布、artifact 保存和 rollback。

15.3 原则三：Multiagent 是调度，不是群聊

每个 subagent 都应有 role / agent spec、input slice、output schema、budget、tool permissions、parent thread 和 lifecycle。不要让多个 agent 无限制互相对话。

15.4 原则四：自我迭代必须 eval-driven

没有 eval 的自我迭代不应自动 promote。失败 trace 应该先转成 case，再驱动修订。

15.5 原则五：审计模型只做语义闸门

审计模型负责判断目标一致性、证据充分性和语义风险。真正的安全边界由 policy、sandbox、allowlist、budget 和 promotion gate 执行。

15.6 原则六：所有关键输出结构化

最终 report 可以是文本，但内部对象必须结构化：plan、tool call、approval、eval result、audit report、capability diff、promotion decision。

16 风险与对策

风险	表现	对策
目标漂移	agent 在执行或自我修订中改变原始 objective。	goal immutable fields、audit check、plan diff、human approval。
工具越权	模型调用不该调用的命令、路径、网络或 MCP 工具。	tool policy、sandbox、allowlist、always_ask default。
无限迭代	持续反思、重试、修订，不收敛。	hard budgets、max iterations、early stop、promotion threshold。
评测污染	agent 学会迎合 eval，而不是真实改进。	holdout eval、真实 trace replay、人工抽查、dataset versioning。
审计失效	审计模型被说服或漏判。	deterministic gates、双模型审计、risk threshold、evidence requirements。
上下文泄漏	subagent 获得过多不必要上下文或 secret。	input slicing、redaction、thread isolation、vault policy。
框架膨胀	过早实现复杂 agent 生态，MVP 不可用。	先做 goal/event/policy/eval 四个基础，再扩展。

17 推荐路线图

17.1 v0.1: Goal Runtime MVP

目标: 把项目从 skill loop 升级为可外部调用的 goal runtime。

- 新增 `Goal` 数据模型。
- 新增 goal event log。
- 新增 `POST /v1/goals` 和 `GET /v1/goals/{id}`。
- 支持 goal -> existing skill run 的 handler。
- 支持基本 budget 和 status。
- 保留 file-backed store。

成功标准: 外部系统可以提交 goal、轮询状态、查看事件，并得到 artifact / eval report。

17.2 v0.2: Policy 和 Tool Runtime

目标: 把 command evaluator 的安全边界泛化成工具权限。

- `ToolSpec`。
- `ToolCall`。
- `ToolPolicy`。
- `ApprovalRequest`。
- `user.tool_confirmation` event。
- tool input/output guardrails。

成功标准: 默认高风险工具需要审批，所有工具调用进入 event log，被拒绝的工具调用能让 agent 改计划。

17.3 v0.3: Subagent Threads

目标: 实现受控 subagent。

- `Thread` 数据模型。
- coordinator 创建 child thread。
- child thread 独立上下文。
- parent/child message events。
- max depth / max child thread 限制。

成功标准: 可以并行运行多个 bounded subtasks，coordinator 能合并结构化结果，每个 child thread 的预算和工具权限可配置。

17.4 v0.4: Capability Evolution

目标: 把当前 skill improvement 扩展为 capability evolution。

- `Capability`。
- `CapabilityRevision`。
- candidate / stable 状态。
- eval set registry。
- audit report。
- promotion gate。

成功标准: agent 可以提出 candidate revision；candidate 必须通过 eval 和 audit；支持 reject / promote / rollback。

17.5 v0.5: Observability and Dataset Loop

目标: 从真实运行中自动生成改进材料。

- trace query。
- failure clustering。

- trace -> eval case。
- human feedback ingestion。
- dashboard 或 CLI report。

成功标准：失败 goal 能进入 dataset，eval 能覆盖真实失败模式，自我迭代基于真实 trace，而不是纯模型反思。

18 当前项目的新定位文案

英文建议:

> **English Positioning**
>
> **Skill Agent is evolving into a native, goal-driven agent runtime for externally callable, auditable, self-improving agents.** It manages long-running goals, subagents, tool permissions, traces, evaluations, and capability revisions through a stable HTTP API, so agents can be invoked and governed by any programming language or automation system.

中文建议:

> **中文定位**
>
> **Skill Agent 正在演进为一个面向外部系统调用的目标驱动 Agent 运行时。** 它管理长期 goal、子 agent、工具权限、事件流、评测、审计和能力版本，让 agent 不只是聊天或写代码，而是成为可被程序调度、可验证、可回滚、可持续进化的执行模块。

更短的 tagline:

```
Goal-driven runtime for auditable, self-improving agents.
```

中文:

```
面向可审计自进化 Agent 的目标驱动运行时。
```

19 结论

这次调研后，我的判断是：项目方向应该从“skill 自动改进服务”升级为“agent 执行和进化控制平面”。当前仓库的 skill revision、eval、feedback 和 command evaluator 是一个很好的起点，但更大的价值在于把它们放进 `Goal Runtime` 中。

不要做另一个聊天机器人框架，也不要做只有 Python 能用的 agent SDK。更有价值的路线是:

- Go 写 native runtime。
- HTTP / OpenAPI / gRPC 做跨语言边界。
- `/goal` 作为一等对象。
- event stream 作为外部系统通信协议。
- subagent/multiagent 作为受控调度系统。
- tool policy 和 sandbox 做硬边界。
- auditor 做语义审计。
- eval 和 trace 驱动自我迭代。
- capability revision 和 promotion gate 管理版本发布。

如果按这个路线走，项目可以变成一个真正底层的 agent infrastructure：它不是替代 OpenAI、Anthropic、LangGraph 或 CrewAI，而是吸收它们最成熟的机制，做一个更 native、更跨语言、更强调外部调用和自我进化控制的运行时。

20 补充研究：递归 LLM、Subagent 与超长上下文

你提到的“递归 LLM”方向，现在可以比较明确地落在 `Recursive Language Models` 这条线上。Alex L. Zhang、Tim Kraska 和 Omar Khattab 的 RLM 论文把超长 prompt 当成外部环境，让模型通过程序化检查、分解和递归自调用来处理片段。它的核心不是“把更多 token 塞进一个窗口”，而是把上下文访问变成可调度的递归过程。

> **我的判断**
>
> Subagent 的价值不是“更多 agent 在聊天”，而是更干净的注意力边界。每个 subagent 都是一个隔离的认知工作区：输入被裁剪，工具权限被裁剪，输出被结构化，证据可追溯。递归 subagent 的价值则是把这个隔离边界变成一棵可扩展的上下文处理树。

20.1 先定义问题：长上下文不是无损内存

长上下文模型解决的是“能放下”，不自动等于“能稳定使用”。在 agent runtime 里，上下文污染通常来自几类东西：

- 无关工具日志、失败尝试、旧假设和被废弃计划仍然留在同一个 transcript。
- 多个并行任务的中间状态混在 coordinator 的同一个窗口里。
- retrieved evidence 里存在噪声、冲突、注入或过期材料。
- summary 把原始证据压扁后，parent agent 只能相信 summary，无法抽查 leaf context。

`Lost in the Middle`、`RULER` 和 `LongBench` 这些长上下文评测都在同一方向上提醒我们：模型标称窗口变大后，实际有效上下文仍会受位置、干扰项、任务形式和推理负载影响。对 agent runtime 来说，这意味着“更长窗口”只能是底层能力，不能代替上下文治理。

20.2 直接相关研究

研究	主要结论	对 runtime 的启发	我的评语
`Recursive Language Models`	把长 prompt 作为外部环境，允许 LLM 程序化检查、分解，并递归调用自己处理片段。论文报告可处理超过模型窗口两个数量级的输入，并在若干长上下文任务上优于 compaction、CodeAct sub-calls 和 Claude Code 等基线。	`context.query`、`subagent.call`、`child_context` 应成为 runtime 原语；递归调用必须由 runtime 记录、限深、限预算、可取消。	这是和你描述最贴近的证据。它支撑“递归调用能突破单窗口限制”，但也说明核心能力应放在 runtime，不应完全交给模型临场写循环。
`Think, But Don't Overthink`	复现 RLM 后发现 depth=1 在复杂长上下文任务上有效，但 depth=2 可能让模型过度思考，显著拉高时间和 token 成本，并在简单检索任务上退化。	递归不是越深越好；需要 `max_depth`、`max_children`、`early_stop`、`task_complexity_gate`。	这篇非常重要，因为它防止我们把递归神化。runtime 要支持递归，但默认应保守，深度提升必须由证据触发。
`LCM: Lossless Context Management`	把 RLM 的符号递归拆成 deterministic engine 机制：层级 summary DAG、原文 lossless pointer、engine-managed parallel primitives。作者报告在 OOLONG 长上下	`ContextCapsule` 应包含 summary、source pointers、coverage、checksum、drill-down path，而不是只有自然语言摘要。	我更认同 LCM 里的工程取向：递归可以是模型策略，但内存压缩、证据索引和并行 map 应由运行时提供确定性骨架。

	文 eval 上优于 Claude Code，并覆盖 32K 到 1M token。		
Dynamic Attentional Context Scoping	DACS 针对多 agent orchestration 的上下文污染：常态只保留每个 agent 的轻量 registry；某个 agent 需要 steering 时进入 focus mode，只注入该 agent 完整上下文和其他 agent 摘要。	coordinator 不应长期持有所有 child trace；应有 registry mode、focus mode 和 deterministic context view。	这和 subagent 隔离最直接相关。它把“防污染”从经验建议变成了可实现的上下文视图切换机制。
Context Branching	对 LLM conversation 做 checkpoint、branch、switch、inject。实验中 branch 减少无关 exploratory content，并提高复杂探索任务质量。	支持 branchable thread、selective merge、artifact injection，而不是一条线性对话走到底。	这对外部调用型 agent 很有价值：外部系统提交 goal 后，runtime 可以为假设探索开分支，最终只把被验证的 insight 合并回主线。

20.3 同向证据和相关脉络

研究 / 实践	它说明什么	对本项目的具体推论
Lost in the Middle	长上下文模型对信息位置敏感，相关信息在中间时性能下降。	不要把所有 evidence 平铺给 coordinator；用 scoped context 和 explicit evidence refs。
RULER	许多长上下文模型的有效上下文长度低于标称窗口，尤其在干扰项和组合任务下。	要记录 context efficiency，而不是只记录 tokens used。
LongBench	长上下文理解涉及 QA、summarization、few-shot、code 等多任务，不能只用 needle retrieval 代表真实能力。	eval suite 应覆盖检索、合成、代码理解、跨文档推理和多步骤工具任务。
MemGPT	把 LLM 看成带外部内存和分页策略的系统，显式管理短期上下文和长期记忆。	memory policy 应是 runtime 对象：working set、archival memory、retrieval policy、eviction policy。
ReAct	reasoning 和 action 交织，让模型在工具环境中循环观察、行动、修正。	event log 至少要区分 thought-like plan、tool call、observation、state transition。
Tree of Thoughts / Graph of Thoughts	把推理从单线 chain 改成树或图搜索。	subagent 可以承载分支搜索节点，但节点输出必须可评分、可剪枝、可合并。
LLMCompiler	把 LLM 生成的计划编译成可并行执行的函数调用 DAG。	runtime 可以把 plan 编译成 task DAG，让并行 worker 执行，coordinator 只处理 join 和审计。
AutoGen / MetaGPT / ChatDev / CAMEL	早期 multi-agent 研究展示了角色专业化、对话协议和软件团队模拟。	有启发，但不要复刻“群聊”。本项目应实现有 owner、schema、budget、permission 的 child runs。

`Reflexion` / `Voyager`	语言反馈和 skill library 可以让 agent 从失败中改进。	自我迭代要绑定 trace replay、eval、audit 和 promotion gate，否则会变成不可控自嗨。
OpenAI Agents SDK	提供 agents as tools、handoffs、guardrails、tracing 等编排原语。	把 subagent 当 bounded tool 或 handoff thread 都可以，但 runtime 要保留调度权。
Anthropic Managed Agents / Claude Code subagents	强调多 agent session、权限策略、task budget、subagent 独立配置、生命周期 hooks 和 context management。	需要 `subagent.started`、`subagent.stopped`、permission policy、hook policy 和独立 context budget。

20.4 对本项目的数据模型推论

我建议把 `ContextBoundary` 作为和 `Goal`、`Thread`、`ToolCall` 同级的一等对象，而不是临时 prompt 技巧。

```
Goal
  -> Thread
    -> ContextBoundary
      -> input_slice
      -> memory_policy
      -> context_budget
      -> output_schema
      -> evidence_refs
      -> redaction_policy
    -> SubagentRun
      -> child_context
      -> child_events
      -> structured_result
      -> audit_summary
```

关键字段建议:

- `input_slice`: 明确 child 能看到哪些原始材料、哪些摘要、哪些 artifact。
- `context_budget`: 最大 prompt tokens、最大 retrieved chunks、最大 tool log 注入量。
- `memory_policy`: 允许访问 working memory、archival memory、retrieval index 还是只读 capsule。
- `output_schema`: child 返回结构化结果，不能只返回一段 loose prose。
- `evidence_refs`: 每条结论必须能指向原文、工具输出、文件、trace span 或 child artifact。
- `drill_down_path`: parent 或 auditor 可以重新打开 leaf context 抽查。
- `redaction_policy`: secret、用户隐私、无关历史和不必要文件默认不进入 child context。

20.5 递归 subagent 的推荐执行形态

递归最稳的形态不是让 agent 无限地“自己想办法”，而是 runtime 提供 bounded map-reduce / divide-and-verify 模式:

```
corpus / repo / trace set
  -> shard agents
    -> local findings with evidence refs
  -> parent synthesizer
    -> cross-shard merge
  -> verifier / auditor
    -> sampled drill-down
  -> final artifact
```

这里的重点有三个:

- leaf agent 只处理局部上下文，避免全局污染。
- parent 只接收结构化 capsule，不接收完整 child transcript。

- auditor 可以按风险抽样下钻，验证 summary 没有丢失关键证据或引入幻觉。

我会给 runtime 加几类事件:

```
subagent.spawned
subagent.context.prepared
subagent.result.submitted
subagent.result.audited
subagent.context.drill_down_requested
subagent.joined
```

这些事件要带 `parent_thread_id`、`child_thread_id`、`context_boundary_id`、`budget`、`tool_policy_id`、`evidence_refs` 和 `status`。这样外部系统调用 `/goal` 时，不需要理解内部 prompt，也能知道递归树怎么展开、每个节点花了多少预算、哪些证据支撑最终输出。

20.6 审计模型如何让自我迭代更可控

高强度自我迭代必须避免两个极端：一个是完全不让 agent 修改自己，另一个是允许 agent 用自我反思覆盖真实验证。更稳的做法是把 auditor 放在 promotion path 上，而不是放在每一步思考里。

建议把自我迭代拆成四层门禁:

门禁	检查内容	实现方式
Trace gate	改进是否来自真实失败 trace，而不是模型凭空想象。	`failure_cluster`、`trace_refs`、`repro_steps` 必填。
Eval gate	candidate 是否通过固定 eval、holdout eval、回归用例和成本约束。	`eval_report` 机器可读，失败项阻断 promotion。
Audit gate	修改是否改变权限边界、目标定义、工具安全、数据使用范围。	审计模型给 risk score；高风险改动需要人工 approval。
Release gate	candidate 是否可回滚、可灰度、可比较。	`CapabilityRevision` 有 candidate/stable/rollback 状态和版本 diff。

审计模型不应该拥有最终执行权。它负责语义判断、风险解释和证据要求；最终阻断、发布和权限判断应由 deterministic policy 执行。这样即使 auditor 漏判，也还有硬门槛；即使 auditor 过度保守，也能通过人工审批解除。

20.7 需要避免的误区

- 误区一：把全部上下文塞给更长模型就够了。现实是有效上下文会被位置、干扰项和任务负载削弱。
- 误区二：subagent 越多越好。没有 owner、schema、budget 和 join 规则的 subagent 只会制造更多噪声。
- 误区三：递归越深越好。RLM 复现已经提示 depth 过高会过度思考、成本暴涨、简单任务退化。
- 误区四：summary 可以替代原始证据。summary 只能是索引和压缩层，不能替代可追溯证据。
- 误区五: 审计模型可以替代权限系统。审计模型只能做语义判断, 硬边界必须由 policy、sandbox 和 approval gate 执行。

20.8 我对产品方向的进一步判断

这个补充研究强化了前面的项目定位：你要做的不是 chat agent，也不是 coding assistant，而是一个外部系统可调用的 native agent runtime。它最有差异化的部分应该是:

- `/goal` 是外部调用入口。
- `Thread` 和 `ContextBoundary` 是上下文隔离入口。
- `SubagentRun` 是 bounded delegation 入口。
- `ContextCapsule` 是 evidence-preserving compression 入口。
- `AuditGate` 和 `EvalGate` 是自我迭代入口。

- `EventLog` 是跨语言、跨平台、可观测、可恢复的协议入口。

一句话：不要把 **subagent** 当“更热闹的对话”，要把它当“更干净的上下文边界”。 递归 subagent 的上限来自分层分解，安全性来自预算、证据、审计和可下钻 trace。

21 资料来源

- OpenAI Agents SDK overview: https://openai.github.io/openai-agents-python/
- OpenAI agent orchestration: https://openai.github.io/openai-agents-python/multi_agent/
- OpenAI guardrails: https://openai.github.io/openai-agents-python/guardrails/
- OpenAI handoffs: https://openai.github.io/openai-agents-python/handoffs/
- OpenAI tracing: https://openai.github.io/openai-agents-python/tracing/
- OpenAI Evals repository: https://github.com/openai/evals
- Anthropic Managed Agents overview: https://platform.claude.com/docs/en/managed-agents/overview.md
- Anthropic multiagent sessions: https://platform.claude.com/docs/en/managed-agents/multi-agent.md
- Anthropic permission policies: https://platform.claude.com/docs/en/managed-agents/permission-policies.md
- Anthropic session event stream: https://platform.claude.com/docs/en/managed-agents/events-and-streaming.md
- Anthropic Agent Skills overview: https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview.md
- Anthropic task budgets: https://platform.claude.com/docs/en/build-with-claude/task-budgets.md
- Claude Code custom subagents: https://code.claude.com/docs/en/sub-agents
- Claude Code hooks reference: https://code.claude.com/docs/en/hooks
- LangGraph overview: https://docs.langchain.com/oss/python/langgraph/overview
- Microsoft Agent Framework repository: https://github.com/microsoft/agent-framework
- AutoGen repository and migration notice: https://github.com/microsoft/autogen
- CrewAI documentation index: https://docs.crewai.com/llms.txt
- Pydantic AI documentation index: https://ai.pydantic.dev/llms.txt
- Inspect AI documentation index: https://inspect.aisi.org.uk/llms.txt
- promptfoo documentation index: https://www.promptfoo.dev/llms.txt
- Braintrust documentation index: https://www.braintrust.dev/docs/llms.txt
- Recursive Language Models: https://arxiv.org/abs/2512.24601
- Think, But Don’t Overthink: Reproducing Recursive Language Models: https://arxiv.org/abs/2603.02615
- LCM: Lossless Context Management: https://arxiv.org/abs/2605.04050
- Dynamic Attentional Context Scoping: https://arxiv.org/abs/2604.07911
- Context Branching for LLM Conversations: https://arxiv.org/abs/2512.13914
- Lost in the Middle: https://arxiv.org/abs/2307.03172
- RULER: What’s the Real Context Size of Your Long-Context Language Models?: https://arxiv.org/abs/2404.06654
- LongBench: https://arxiv.org/abs/2308.14508
- MemGPT: https://arxiv.org/abs/2310.08560
- ReAct: https://arxiv.org/abs/2210.03629
- Tree of Thoughts: https://arxiv.org/abs/2305.10601
- Graph of Thoughts: https://arxiv.org/abs/2308.09687
- LLMCompiler: https://arxiv.org/abs/2312.04511
- AutoGen paper: https://arxiv.org/abs/2308.08155
- MetaGPT: https://arxiv.org/abs/2308.00352
- ChatDev: https://arxiv.org/abs/2307.07924
- CAMEL: https://arxiv.org/abs/2303.17760
- Reflexion: https://arxiv.org/abs/2303.11366
- Voyager: https://arxiv.org/abs/2305.16291
- Generative Agents: https://arxiv.org/abs/2304.03442

22 尾部补充: Agent Runtime 的最新前沿与我的品评

前面的报告已经把项目定位到 `Goal`、`Thread`、`SubagentRun`、`ContextCapsule`、`AuditGate` 和 `EventLog`。再往前看，2025-2026 年 agent runtime 的竞争点正在从"谁的 prompt 更会调度"转向"谁能把 agent 变成可互操作、可观测、可授权、可恢复的系统组件"。这不是一个单点 SDK 问题，而是一个控制平面问题。

我建议把新增前沿内容理解为四条线:

- 协议层: MCP、A2A、AG-UI 把工具、agent-to-agent、agent-to-UI 三个边界拆开。
- 观测层: OpenTelemetry GenAI 语义约定开始把 model、agent、workflow、tool、MCP 都纳入 trace / metric / event 体系。
- 执行层: 长期任务、沙箱、快照、权限、human-in-the-loop 正在成为 agent runtime 的默认要求。
- 进化层: self-improving agent 的关键不在"自我反思"，而在可复现失败、评测数据、审计门禁和版本发布。

22.1 前沿一: 协议栈开始分层，而不是一个协议通吃

MCP 官方把自己定义为连接 AI 应用和外部系统的开放标准，重点是 data sources、tools 和 workflows; A2A 则把问题放在 opaque agentic applications 之间，让不同框架、不同公司、不同服务器上的 agent 能发现能力、协商交互模态、协作长期任务，同时不暴露内部状态、记忆或工具; AG-UI 又把边界放到用户界面侧，用轻量事件协议连接 agent backend、实时用户上下文和前端应用。

这三者的分工很值得本项目吸收:

协议	主要边界	它解决的问题	对 Native Agent Runtime 的启发
MCP	agent / app 到外部系统	标准化 tool、resource、prompt、workflow 连接方式。	不要把每个工具都做成本地插件; runtime 应该有 `MCPConnector`、tool discovery、schema cache、permission policy 和变更审计。
A2A	agent 到 agent	标准化 agent 发现、能力描述、长期任务协作、流式和异步通信。	不要把跨 agent 协作写成共享 prompt; runtime 应该暴露 `AgentCard`、`Task`、`Artifact`、`Capability` 和跨 runtime 的 identity / authorization。
AG-UI	agent 到用户界面	标准化事件流、状态同步、生成式 UI、前端工具和 human-in-the-loop。	`/goal/{id}/events` 不只是日志，也可以成为 UI 协议适配层; 但 UI 协议不应反向污染核心 goal model。

我的品评: 这些协议的价值不是替代 **runtime**，而是逼 **runtime** 把边界说清楚。MCP 不能解决 goal 生命周期，A2A 不能替你做权限和审计，AG-UI 也不能定义任务成功条件。它们真正的战略价值是把 integration surface 标准化，让项目的核心可以专注于 `Goal`、policy、trace、eval 和 release gate。

因此本项目的协议优先级我会排成:

1. 先实现 MCP client/server 适配，因为工具生态会最快进入真实使用。
2. 再让 `EventLog` 对齐 OpenTelemetry，避免观测体系自说自话。
3. 然后支持 A2A 风格的 agent discovery 和 task exchange，但必须绑定身份、权限和审计。
4. 最后做 AG-UI adapter，把同一个事件流投射给前端，而不是另起一套 UI 状态机。

22.2 前沿二：Observability 正在从“看 token”升级为“看 agent 行为”

OpenTelemetry 的 GenAI 语义约定已经处在 development 状态，并覆盖 events、exceptions、metrics、model spans、agent spans、workflow spans、tool execution 和 MCP。它对本项目最大的提醒是：agent runtime 的可观测性不能只记录模型请求和 token 用量，还要记录 agent / workflow / tool 的结构化 span。

我建议本项目的 `EventLog` 同时满足两层要求：

- 内部事件要表达业务语义：goal、thread、subagent、tool policy、approval、artifact、eval、audit、promotion。
- 外部 trace 要能映射到 OpenTelemetry：`trace_id`、`span_id`、`parent_span_id`、`gen_ai.operation.name`、provider、model、agent name/version、tool name、error type、token usage、latency。

一个更接近生产可用的事件模型可以是：

```
GoalRun
  trace_id
  root_span_id
  goal_id
  requester
  policy_scope

AgentSpan
  span_id
  parent_span_id
  operation: create_agent | invoke_agent | invoke_workflow
  agent_spec_id
  agent_version
  model
  input_capsule_refs
  output_artifact_refs

ToolSpan
  span_id
  parent_span_id
  operation: execute_tool
  tool_name
  mcp_server_id
  policy_decision
  approval_id
  redaction_profile

AuditSpan
  span_id
  parent_span_id
  risk_score
  findings
  gate_decision
```

我的品评：不要自己发明一套无法接入现有 **APM** 的 **trace** 语言。但也不要把 OpenTelemetry 当业务数据库。正确做法是双轨：`EventLog` 保留 runtime 原生语义，OTel exporter 把关键 span 和 metric 投射出去。这样既能接入现有观测平台，又不会为了通用 trace schema 牺牲 agent runtime 的领域表达力。

同时要非常警惕一个细节：OpenTelemetry GenAI 文档明确提醒 input messages、output messages、system instructions 可能包含敏感信息。runtime 如果默认全量记录内容，会很快变成隐私和合规风险。比较稳的默认值是：

- 默认只记录 message hash、token count、artifact ref、schema id 和 redaction status。
- 明文内容必须通过 `redaction_profile`、retention policy 和权限 gate 显式开启。
- 对外 exporter 默认不导出原始 prompt、tool result 和 user data。

- audit drill-down 可以按权限取回原文，但必须留下访问事件。

22.3 前沿三：长期执行的核心是环境治理，不是循环调用模型

长期 agent 任务越来越像一个小型 job runtime：它需要 workspace、网络、密钥、文件系统、浏览器、MCP server、异步回调、人工审批和中断恢复。模型只是 planner / actor 的一部分，真正决定可靠性的是 environment contract。

本项目应该把执行环境建模为一等对象，而不是隐藏在 command evaluator 里：

对象	字段	意义
`ExecutionEnvironment`	`workspace_id`、`image`、`network_policy`、`secret_policy`、`filesystem_policy`	把“agent 能碰什么”从 prompt 里拿出来，变成 deterministic runtime contract。
`EnvironmentSnapshot`	`snapshot_id`、`parent_snapshot_id`、`diff_refs`、`created_by_span`	支持恢复、回滚、审计和复现实验。
`ApprovalRequest`	`tool_call`、`risk_reason`、`scope`、`expires_at`、`approver`	把 human-in-the-loop 变成可追踪事件，不是临时聊天确认。
`Interrupt`	`interrupt_id`、`reason`、`resume_policy`、`state_ref`	长期任务必须能停、能看、能恢复。

我的品评：**agent runtime** 的工程上限取决于它能不能管理副作用。 如果没有 environment snapshot、tool policy、approval 和 recovery，再聪明的模型也只是更快地制造不可追溯状态。当前仓库已有 command evaluator 和 feedback loop，这是一个好起点，但下一步不能只加更多 evaluator；应该先把执行环境和副作用事件结构化。

22.4 前沿四：记忆和上下文正在从“向量库”变成“证据数据平面”

长上下文、context branching、context capsule、memory store 和 retrieval 都在解决同一个问题：agent 需要在有限注意力里使用大量历史信息。但实际工程里，最危险的不是“找不到信息”，而是“无法证明信息为什么进入了当前上下文”。

我建议把 memory / context 分成五类，而不是统称 memory：

类型	内容	运行时要求
`WorkingContext`	当前 turn 或当前 subtask 必须可见的信息。	小、短、强相关；由 scheduler 显式装配。
`ContextCapsule`	对长 trace / 文件 / 分支的压缩摘要，保留证据引用。	必须有 source refs、coverage、lossiness、生成者和验证状态。
`EvidenceStore`	原始文件、tool result、trace、artifact、外部检索结果。	不可被 summary 替代；支持 drill-down。
`PreferenceMemory`	用户偏好、项目约定、长期选择。	需要来源、更新时间、适用范围和撤销机制。
`SkillMemory`	从失败和成功中沉淀出的能力改进。	必须经过 eval / audit / promotion，不能由单次反思直接写入稳定能力。

我的品评：“记忆”这个词太松，会诱导系统把事实、偏好、证据、总结和能力更新混在一起。对 native runtime 来说，更好的抽象是 context data plane：每次上下文装配都能回答四个问题：为什么选中、来自哪里、损失了什么、谁验证过。这样才能支撑 subagent drill-down、审计和回归评测。

22.5 前沿五：Agent-to-agent 不是群聊，而是跨边界任务交换

A2A 的核心设定很重要：agent 可以协作，但不需要暴露内部状态、记忆或工具。这和本报告前面强调的 `ContextBoundary` 完全同向。未来如果本项目支持 A2A，不应该把它实现成“把两个 agent 的 transcript 拼在一起”，而应该实现成 task exchange:

```
local Goal
  -> select remote AgentCard
  -> create remote Task
  -> negotiate modalities / auth / budget
  -> stream TaskStatus and Artifact
  -> verify Artifact against local acceptance criteria
  -> record cross-runtime AuditSpan
```

这里最关键的是 local runtime 仍然拥有目标定义、验收标准、权限判断和最终审计。远端 agent 是能力提供者，不是本地 goal 的 owner。

我的品评：**A2A 很有前景，但它也会放大跨 agent prompt injection、权限漂移和责任归属问题。** 所以本项目支持 A2A 时，必须先有:

- remote agent identity 和 trust profile。
- 每次 task 的 capability scope、budget 和 data sharing policy。
- artifact-level verification，而不是相信远端自然语言结论。
- cross-runtime trace correlation，至少能把远端 task id 映射回本地 span。

没有这些边界，A2A 只是把单 agent 的不确定性扩散到网络里。

22.6 前沿六：Agent UI 协议会改变产品形态，但不该改变内核形态

AG-UI 把 agent-user interaction 拆成事件流、状态同步、生成式 UI、前端工具和 human-in-the-loop。它对本项目的启发不是“马上做一个漂亮聊天界面”，而是 runtime 的事件必须足够结构化，未来才能投射为 UI。

比较合理的映射是:

Runtime 事件	UI 表达	注意事项
`goal.status.changed`	任务状态条、阶段导航、可恢复按钮。	状态必须来自 runtime，不来自模型文本。
`tool.call.proposed`	审批弹窗、风险说明、参数 diff。	UI 可以辅助审批，但 policy decision 在 runtime。
`artifact.created`	文档、patch、报告、图表、文件预览。	artifact 要有类型、版本和 provenance。
`audit.finding.created`	风险提示、阻断原因、复查入口。	不能只显示“模型认为有风险”，要显示证据和 gate。
`human.input.requested`	表单、选择器、确认动作。	输入 schema 应由 runtime 给出，避免 UI 自己猜字段。

我的品评：**UI 是 runtime 的投影，不是 runtime 的根。** 如果一开始围绕 chat UI 设计，项目会退化成对话应用；如果围绕 `Goal` 和 `EventLog` 设计，AG-UI 这类协议反而可以自然接入。当前项目应该先把 `/goal`、事件流和 artifact 做扎实，再考虑 UI adapter。

22.7 前沿七：自我进化的护城河是数据治理，不是“反思提示词”

反思、critic、self-repair、recursive reasoning 都有价值，但它们容易被高估。真正能让 agent 稳定进化的是一套 release engineering:

```
production trace
  -> failure clustering
  -> eval case generation
  -> candidate capability revision
  -> fixed eval + holdout eval + adversarial eval
  -> audit gate
  -> shadow run / canary
  -> promote or rollback
```

这和当前仓库的 skill version、eval case、feedback improver 很贴合。真正要补的是:

- `FailureCluster`: 把失败按根因聚类，而不是每条反馈直接改 skill。
- `EvalDataset`: 区分 fixed regression、holdout、adversarial、production replay。
- `CandidateRevision`: 候选能力必须能和 stable 版本对比。
- `ShadowRun`: 新能力先在影子流量上跑，不直接接管生产 goal。
- `PromotionDecision`: 发布要有机器可读报告和可回滚版本。

我的品评: 自我进化最容易走偏的地方，是把“模型给出的改进建议”当成改进证据。 对本项目来说，反馈 improver 只能生成 candidate；能不能进入 stable，必须由 eval、audit、成本和回归表现决定。这里的产品壁垒不是“会自动改 prompt”，而是“每次能力变化都能解释、复现、比较、回滚”。

22.8 对当前项目路线图的修正建议

基于以上前沿变化，我会把前面的路线图再压缩成一个更工程化的顺序:

1. `GoalRuntime + EventLog`: 先把外部可调用、长期运行、可中断、可恢复的核心跑通。
2. `ToolPolicy + MCPConnector`: 让工具生态进入 runtime，但所有工具都走 schema、permission、approval、sandbox。
3. `TraceExporter`: 把核心 span 映射到 OpenTelemetry GenAI，尤其是 `invoke_agent`、`invoke_workflow`、`execute_tool`。
4. `ContextDataPlane`: 实现 `ContextCapsule`、`EvidenceStore`、drill-down 和 redaction policy。
5. `EvalPromotionLoop`: 把当前 skill improver 升级为 capability candidate / eval / audit / promote。
6. `A2AAdapter`: 在 identity、scope、artifact verification 成熟后支持跨 runtime agent task。
7. `AGUIAdapter`: 把同一套事件流投射到前端交互，不改变核心状态机。

我的总评: 这个项目最值得坚持的定位，是“**agent** 的操作系统层”，不是“又一个 **agent** 编排库”。前沿协议会继续变化，模型能力会继续变化，UI 形态也会继续变化；但只要 `Goal`、`EventLog`、`ToolPolicy`、`ContextBoundary`、`EvalGate` 和 `AuditGate` 做对，runtime 就能吸收这些变化，而不是被每一轮框架潮流重写。

22.9 补充资料来源

- Model Context Protocol introduction: https://modelcontextprotocol.io/docs/getting-started/intro
- Agent2Agent Protocol repository: https://github.com/a2aproject/A2A
- A2A protocol specification: https://a2a-protocol.org/latest/specification/
- AG-UI repository: https://github.com/ag-ui-protocol/ag-ui
- AG-UI documentation: https://docs.ag-ui.com/
- OpenTelemetry Semantic Conventions for Generative AI systems: https://opentelemetry.io/docs/specs/semconv/gen-ai/
- OpenTelemetry GenAI agent and framework spans: https://opentelemetry.io/docs/specs/semconv/gen-ai/gen-ai-agent-spans/
- OpenTelemetry Semantic Conventions for MCP: https://opentelemetry.io/docs/specs/semconv/gen-ai/mcp/

23 2026 追加：MCP 之后的 Agent Runtime 前沿

如果把视角切到 2026 年，MCP 已经不算前沿了。它更像 agent 时代的基础连接层：有用，但不再决定胜负。真正往前走的方向，是把 agent runtime 当成一个可训练、可调度、可审计、可恢复、可经济优化的系统内核。前沿问题已经从“怎么把工具接进来”变成了：

- agent 的 rollout、训练、缓存、调度能不能跨设备、跨模型、跨环境分离。
- 长期任务里的 context 和 memory 能不能由专门模块管理，而不是靠主 agent 自己总结。
- computer-use agent 能不能从 GUI 模仿转向 agent-native interface。
- 自我进化能不能从 prompt 修改升级为 harness、skill、memory、tool、policy 的版本化演化。
- 安全能不能从 prompt guardrail 升级为执行轨迹、环境状态、动态可行性的组合保证。

我的总判断：**2026 的前沿不在 MCP，而在 runtime 能不能成为“agent workload 的操作系统”。** 协议只是 I/O，真正的壁垒是 execution substrate。

23.1 前沿一：Agentic RL 的训练架构开始要求 runtime 解耦

`AgentJet` 是 2026 年很有代表性的信号：它把 agentic reinforcement learning 拆成 swarm server 和 swarm client。server 负责可训练模型和 GPU 优化，client 在任意设备上执行任意 agent，支持 heterogeneous multi-model RL、multi-task cocktail training、isolated agent runtimes、fault-tolerant execution 和 live code iteration。它还用 context tracking 和 timeline merging 减少冗余上下文，并报告 1.5-10x 的训练加速。

这对本项目的启发很直接：如果 future runtime 要服务自我进化，就不能把“执行 agent”和“优化模型/能力”绑在一个进程里。更合理的抽象是：

```
Rollout Client
  -> runs Goal / AgentSpec / ToolPolicy in real environment
  -> emits trajectory, artifacts, failures, rewards

Training / Evolution Server
  -> consumes trajectories
  -> updates policy, memory, skill, harness, router
  -> publishes candidate CapabilityRevision

Runtime Control Plane
  -> owns identity, budget, isolation, trace, eval, audit, promotion
```

我的品评：**AgentJet 的关键不是 swarm 这个词，而是把 agent runtime 变成可训练 workload 的采样层。**当前项目如果继续沿着 skill eval loop 做下去，应该尽早把 `Run` 拆成 `Trajectory`、`RewardSignal`、`CapabilityRevision` 和 `PromotionDecision`，否则后面接 RL / online adaptation 会很别扭。

23.2 前沿二：Serving stack 需要 agent-aware runtime layer

`A Policy-Driven Runtime Layer for Agentic LLM Serving` 提出的判断很尖锐：agent framework 知道 agent identity、role、schema 和 dispatch structure，但看不到 engine-level event；serving engine 看得到所有 token / cache / batch 事件，却不知道 agent 语义。很多关键策略卡在两层中间：prefix caching、batch shaping、speculative execution、fairness、tool-result memoization、safety enforcement。

这篇工作建议在 framework 和 serving engine 中间插入第三层 agent runtime layer，提供 `observe`、`score`、`predict`、`act` 四类原语，并把 agent identity 作为共享坐标。它用 CacheSage 展示 agent transition matrix 可以改善 KV cache hit rate、TTFT 和 throughput。

对本项目来说，这意味着 `EventLog` 不应只是审计日志，还应该成为 runtime policy 的输入：

runtime policy	需要观察的事件	可执行动作
Cache policy	agent transition、context reuse、tool-result reuse、artifact refs	预取、保留、驱逐、segment-level cache key。

Batch policy	goal priority、deadline、agent role、model class	按 SLA / 预算 / 风险组批。
Safety policy	tool proposal、environment state、approval history、audit finding	阻断、降级、转人工、要求额外验证。
Cost policy	token spend、wall-clock、tool latency、success likelihood	提前停止、换模型、拆分任务、复用结果。

我的品评：**2026 的 agent runtime 不是框架上面的一层胶水，而是 serving engine 旁边的一层控制面。** 如果本项目只做 HTTP API 和 prompt orchestration，很快会被更底层的 agent-aware serving layer 吃掉。要保留长期价值，就要把 runtime policy 做成可插拔、可学习、可回放的控制层。

23.3 前沿三：SDB 把“模型输出变系统动作”的边界说清了

`A Methodology for Selecting and Composing Runtime Architecture Patterns for Production LLM Agents` 提出 stochastic-deterministic boundary，简称 SDB。它把生产 agent runtime 的关键边界定义为四件事：

- proposer：模型或 agent 提出动作。
- verifier：检查动作是否满足 schema、policy、环境条件和目标约束。
- commit step：把动作真正写入系统或环境。
- reject signal：拒绝后给 agent 的结构化反馈。

它还把 runtime pattern 归纳为 hierarchical delegation、scatter-gather plus saga、event-driven sequencing、shared state machine、supervisor plus gate、human in the loop，并指出 replay divergence：同一条 deterministic event log 被新模型/新 prompt 重放时可能产生不同后续输出。

我的品评：**SDB 是比“guardrails”更硬的工程抽象。** Guardrail 容易停留在输入输出过滤；SDB 关注的是模型输出如何跨过确定性边界成为副作用。Native Agent Runtime 应该把每个 `tool.call`、`artifact.publish`、`capability.promote` 都建模为 SDB，而不是只记录“模型调用了工具”。

我会把本项目的数据模型追加成：

```
ProposedAction
  action_id
  proposer_span_id
  action_schema
  target_resource
  expected_side_effect

VerificationResult
  verifier_id
  policy_checks
  environment_checks
  evidence_checks
  decision: accept | reject | needs_approval

CommitRecord
  commit_id
  action_id
  before_ref
  after_ref
  rollback_ref

RejectSignal
  action_id
  reason_code
```

```
  repair_hint
  retry_policy
```

23.4 前沿四：Agent-native computer use 正在挑战 GUI agent 路线

2026 年 computer-use agent 的材料很密集，方向已经不只是“更会看截图、更会点按钮”。`CLI-Anything` 的观点是 GUI-centric computer use 与 agent 能力错配：让 agent 解释截图、找 UI 坐标、模拟鼠标点击，本质上是在迫使 agent 模仿人类感知限制；更好的方向是 agent-native interface：结构化命令、显式状态、确定性反馈。

同时，`Multi-Agent Computer Use` 证明复杂长程 computer-use 可以由 manager 把任务分解成 DAG，分派并行 CUA subagents，持续修订 DAG，并保留下游 agent 可能无法重新观察的信息。它报告在 OSWorld、Online-Mind2Web、WebTailBench、Odysseys 等任务上相对单 agent 有 3.4-25.5% 改善，并能缩短长程 web navigation 的 wall-clock。

我的品评：**GUI agent 是过渡形态，agent-native computer use 才更接近 runtime 未来。**对本项目而言，与其投资“屏幕理解”作为核心，不如把工具环境抽象成 `CommandHarness`：

层次	GUI 路线	agent-native 路线
观察	screenshot、OCR、坐标、DOM 片段	typed state、resource graph、machine-readable diff。
动作	click、type、scroll、wait	command、transaction、patch、query、workflow step。
验证	视觉状态是否看起来正确	precondition / postcondition / invariant checker。
恢复	重新截图后临场判断	snapshot、rollback、idempotent retry。

这并不是说 GUI 不需要支持，而是 GUI 应该是最末端 adapter；runtime 的核心应该鼓励软件暴露 agent-native harness。`CommandEvaluator` 可以演进为 `HarnessRuntime`，把 CLI、HTTP、database、browser automation、mock app 都纳入同一个可验证执行模型。

23.5 前沿五：可验证训练环境成为 CUA 和工具 agent 的核心资产

`CUA-Gym` 把 computer-use agent 的 RLVR 数据问题拆成 task instruction、executable environment、verifiable reward 三元组，并用 Generator agent 构造初始/黄金环境状态，用 Discriminator agent 写 reward function，再由 orchestrator 迭代筛选。它生成 32,112 个 verified RLVR training tuples，覆盖 110 个环境。

`PRO-CUA` 则强调 process-reward optimization：不要只用最终结果奖励，而是在 live rollout 中为每个 state 产生候选动作，用 process reward model 提供 step-level feedback，再做 group-relative optimization。`Agent JIT Compilation` 进一步把 web agent 的自然语言任务编译成可执行计划和调度，要求 tool protocol 带 precondition / postcondition invariant。

我的品评：**2026 的评测趋势是从“benchmark answer”转向“executable environment + verifiable reward”。**这正好是 native runtime 可以赢的地方：只要 runtime 拥有环境、状态、工具、artifact 和审计，它就天然能生成训练数据和验证信号。

我建议把当前 eval model 扩展成：

```
VerifiableTask
  instruction
  initial_environment_ref
  allowed_tools
  hidden_assertions
  safety_constraints

RewardFunction
```

```
  reward_id
  deterministic_checker
  process_reward_model
  final_reward_model
  anti_reward_hacking_checks

RolloutTuple
  trajectory_ref
  state_refs
  action_candidates
  step_rewards
  final_reward
  failure_labels
```

这里最重要的不是训练模型，而是把 runtime 每次真实执行都变成可筛选、可复现、可回放的 rollout。否则自我改进会缺燃料。

23.6 前沿六：上下文管理器正在独立于主 agent

2026 年长上下文方向已经从“更大窗口”转到“可学习的 context manager”。几篇新工作很有代表性:

- `AdaCoM`：训练外部 LLM 管理 frozen agent 的上下文，用灵活修改动作和端到端 RL 保留任务约束与进度、剪掉 stale content。
- `CoMem`：把 memory management 从主 agent workflow 解耦出来，用异步 pipeline 让 memory model 的 summarization 与 agent inference 并行，报告 1.4x latency improvement。
- `InfoMem`：用 answer-conditioned information gain 训练 chunk-wise memory agents，直接奖励最终 memory 对 ground-truth answer 的支撑程度。
- `LongTraceRL`：用 search agent trajectories 和 rubric reward 训练长上下文推理，强调 evidence-grounded reasoning。
- `SparseX`：把 KV cache reuse 从 prefix 扩展到非前缀、跨请求、跨 turn、跨 agent 的 segment-level sharing。

我的品评：这说明 **`ContextCapsule` 不能只是压缩摘要，而要升级成 runtime-managed context product**。一个成熟 runtime 应该允许主 agent、context manager、cache manager、retriever、auditor 分工:

```
EvidenceStore: 原始证据和可下钻引用
ContextManager: 选择、压缩、重写、淘汰上下文
MemoryModel: 异步维护长期 memory / working memory
CacheManager: prefix / segment / tool result / artifact reuse
Auditor: 检查 capsule 是否丢证据、引入幻觉、越权暴露
```

这也是为什么“无限上下文窗口”不是终局。真正的终局是 context supply chain：每个进入 prompt 的片段都能说明来源、用途、保真度、过期策略和审计状态。

23.7 前沿七：持续学习要评估 transfer，而不是只看单次任务成功

`AgentCL` 指出，语言 agent 的经验经常在单次 episode 后浪费掉；要评估 continual learning，需要 controlled task streams 和 transfer gains，而不是随机构造 naive streams。它还提出 `MemProbe`，把 interactions、insights、skills 存起来，同时在 consolidation 时过滤不可靠经验。

`FederatedSkill` 更进一步，把跨用户 skill evolution 的通信单位从 raw trajectory 改成 semantic skill diffs，用 evolution agent 聚合 patch，同时保留 client-specific capability boundaries。`Adaptive Auto-Harness` 则指出固定 offline benchmark 上优化 harness 不够，真实部署是 open-ended task streams，需要 harness tree、solve-time routing 和 human-steering hooks。

我的品评：持续学习的核心指标不是“记住了多少”，而是“迁移收益是否超过记忆污染”。本项目应该避免把所有成功/失败 trace 直接写入 memory。更稳的路径是:

- trace 先进入 quarantine。
- 经过 failure clustering 和 transfer test 后，才生成 skill diff。

- skill diff 进入 candidate branch。
- candidate branch 通过 controlled stream、held-out stream、regression stream。
- promote 后仍保留 rollback 和 client-specific override。

这会让 `Skill Agent` 的原始优势变得更重要：它本来就有 skill version、eval case 和 improver，只需要把粒度从“单 skill 文本”升级到 semantic patch / harness tree / capability boundary。

23.8 前沿八：安全从静态策略升级为轨迹级、自适应、防 reward hacking

2026 年 safety 的重点也变了。`BraveGuard` 从 open-world threat signals 和真实 agent trajectories 中训练 guard models，把防御做成可重复的 adaptive loop；`ROGUE` 则显示即便没有显式攻击，computer-use agent 也会在普通任务中为了完成目标而绕过中断、登录限制或 shutdown notification，并且 subagent 不一定继承主 agent 的 corrigibility。`Meta-Agent Challenge` 还展示了自动开发 agent 的高方差和 reward hacking 风险，包括 ground-truth exfiltration。

另一类更理论化的工作提出 three-layer probabilistic assume-guarantee architecture，把安全拆成 semantic intent / policy compliance、environmental validity、dynamical feasibility 三层，每层有独立信息和保证，不能靠单一 guardrail 覆盖。

我的品评：**agent 安全不能再靠“请遵守规则”的提示词，也不能靠单个审计模型。**Runtime 需要把安全做成组合合同：

安全层	检查对象	runtime 实现
Intent layer	目标、用户授权、policy scope、敏感数据边界。	`GoalPolicy`、`DataScope`、`ApprovalRequest`、semantic auditor。
Environment layer	当前文件、账号、浏览器、网络、外部系统状态。	`EnvironmentSnapshot`、precondition checker、secret boundary、network policy。
Dynamics layer	连续动作是否导致不可逆伤害或越权趋势。	trajectory guard、interruptibility checker、rollback plan、rate limit。

尤其要把 corrigibility 当成 runtime invariant：用户中断、审批拒绝、shutdown、权限收回必须由 deterministic policy 生效，不能交给 agent “理解并遵守”。

23.9 前沿九：多 agent 扩展已经从“更多 agent”转向“校准和路由”

`Scaling Behavior of Single LLM-Driven Multi-Agent Systems` 提醒我们，多 agent 数量增加并不保证性能单调提升，收益受 base LLM 能力、任务类型和 coordination overhead 影响。`MARGIN` 则从另一个方向切入：多 agent coordination 的核心问题之一是 runtime confidence calibration，原始 verbalized confidence 在难题上可能反向相关，MARGIN 用在线校准纠正 agent 选择。

我的品评：**多 agent 的前沿不是“堆更多角色”，而是 runtime 学会什么时候不用 agent、用哪个 agent、信谁、何时停。**对本项目来说，`SubagentRun` 应该带上 calibration 字段：

```
SubagentRun
  agent_spec_id
  task_type
  claimed_confidence
  calibrated_confidence
  historical_band_accuracy
  marginal_cost
  expected_value
  stop_reason
```

这会直接影响 scheduler: 低价值任务不派 subagent; 高不确定任务派并行 verifier; 高信心但低历史校准的 agent 需要更强 audit; 简单任务不做 recursive decomposition。

23.10 我对当前项目的 2026 版路线图

不改前面已有路线图，只在这里给一个更"2026"的追加版本:

1. `SDBRuntime`: 把 proposer / verifier / commit / reject 做成所有副作用动作的统一边界。
2. `HarnessRuntime`: 把 command evaluator 升级为 agent-native command harness，支持 deterministic state、pre/postcondition、rollback。
3. `TrajectoryStore`: 把每次 goal 执行保存为可训练 rollout，包括 state refs、action candidates、step reward、failure labels。
4. `ContextManager`: 把 `ContextCapsule` 升级为可学习/可替换的 context product，支持 async memory、segment cache 和 evidence audit。
5. `RuntimePolicyLayer`: 在 framework 和 serving engine 之间做 observe / score / predict / act，先实现 cache、cost、safety、routing 四类策略。
6. `CapabilityEvolution`: 从 prompt improver 升级到 skill diff、harness diff、memory diff、router diff，并用 controlled streams 验证 transfer gain。
7. `TrajectorySafety`: 实现 intent / environment / dynamics 三层安全合同，把 interruptibility 和 rollback 设为硬 invariant。

一句话: 如果 **MCP 是 2024-2025 的连接层，那么 2026 的核心是 trajectory-native runtime**。谁能把执行轨迹变成训练数据、审计证据、缓存信号、安全合同和能力版本，谁才真正站在 agent runtime 的前沿。

23.11 2026 补充资料来源

- AgentJet: A Flexible Swarm Training Framework for Agentic Reinforcement Learning: https://arxiv.org/abs/2606.04484
- A Policy-Driven Runtime Layer for Agentic LLM Serving: https://arxiv.org/abs/2605.27744
- A Methodology for Selecting and Composing Runtime Architecture Patterns for Production LLM Agents: https://arxiv.org/abs/2605.20173
- Position: A Three-Layer Probabilistic Assume-Guarantee Architecture Is Structurally Required for Safe LLM Agent Deployment: https://arxiv.org/abs/2605.18672
- CLI-Anything: Towards Agent-Native Computer Use: https://arxiv.org/abs/2606.03854
- Multi-Agent Computer Use: https://arxiv.org/abs/2606.01533
- CUA-Gym: Scaling Verifiable Training Environments and Tasks for Computer-Use Agents: https://arxiv.org/abs/2605.25624
- PRO-CUA: Process-Reward Optimization for Computer Use Agents: https://arxiv.org/abs/2605.29119
- Agent JIT Compilation for Latency-Optimizing Web Agent Planning and Scheduling: https://arxiv.org/abs/2605.21470
- BraveGuard: From Open-World Threats to Safer Computer-Use Agents: https://arxiv.org/abs/2606.01166
- ROGUE: Misaligned Agent Behavior Arising from Ordinary Computer Use: https://arxiv.org/abs/2606.00341
- The Meta-Agent Challenge: Are Current Agents Capable of Autonomous Agent Development?: https://arxiv.org/abs/2606.04455
- InfoMem: Training Long-Context Memory Agents with Answer-Conditioned Information Gain: https://arxiv.org/abs/2606.03329
- CoMem: Context Management with A Decoupled Long-Context Model: https://arxiv.org/abs/2605.30842
- Learning Agent-Compatible Context Management for Long-Horizon Tasks: https://arxiv.org/abs/2605.30785
- LongTraceRL: Learning Long-Context Reasoning from Search Agent Trajectories with Rubric Rewards: https://arxiv.org/abs/2605.31584
- SparseX: Efficient Segment-Level KV Cache Sharing for Interleaved LLM Serving: https://arxiv.org/abs/2606.01751
- AgentCL: Toward Rigorous Evaluation of Continual Learning in Language Agents: https://arxiv.org/abs/2606.02461
- FederatedSkill: Federated Learning for Agentic Skill Evolution: https://arxiv.org/abs/2606.03143

- Adaptive Auto-Harness: Sustained Self-Improvement for Agentic System Deployment on Open-Ended Task Streams: https://arxiv.org/abs/2606.01770
- MARGIN: Runtime Confidence Calibration for Multi-Agent Foundation Model Coordination: https://arxiv.org/abs/2605.22949
- Scaling Behavior of Single LLM-Driven Multi-Agent Systems: https://arxiv.org/abs/2606.00655