

google ai agent 白皮书笔记

第一份白皮书：Google 对于下一代 Agentic Systems (代理系统) 的工程规范和架构分层

1. 核心痛点与贡献 (Core Contribution)

痛点 (Pain Points):

- **预测式 AI 的局限性:** 传统的 LLM 是被动的、无状态的 (Stateless)，仅在单一回合内进行预测或生成，无法自主完成跨越时间步 (Time Steps) 的复杂目标。
- **工程化缺失:** 目前 Agent 开发多处于 PoC (概念验证) 阶段，缺乏关于安全性、可观测性 (Observability) 和生产级架构 (Production-grade architecture) 的统一标准。

核心贡献 (Key Contributions):

- **架构解耦:** 明确将 Agent 解构为 Model (大脑)、Tools (手) 和 Orchestration (神经系统) 三大组件。
- **五层分级体系:** 提出了从 Level 0 (单纯推理) 到 Level 4 (自我进化) 的 Agent 能力分级标准，类似于自动驾驶的分级。
- **Agent Ops 定义:** 将 DevOps 和 MLOps 的概念延伸至 GenAIOps 和 Agent Ops，引入了“LLM as a Judge”等评估机制。

2. 系统架构 (System Architecture)

我们将 Agent 视为一个在离散时间步 t 上运行的决策系统。

2.1 输入 (Input)

Agent 的输入空间 I_t 不再仅仅是用户的 Prompt，而是一个复合的上下文 (Context)：

- **Mission (目标):** 用户的高层意图 (e.g., "帮我组织这次旅行")。
- **System Instructions (系统指令):** 包含 Persona (角色设定)、Domain Knowledge (领域知识) 和 Constraints (约束条件)。
- **Observation/Scene (环境观测):**
 - **Short-term Memory:** 当前 Session 的对话历史、State (状态机状态)。
 - **Long-term Memory:** 基于 RAG (Retrieval-Augmented Generation) 从 Vector DB 检索到的历史经验或企业知识。
 - **Tool Outputs:** 上一轮 Action A_{t-1} 执行后的返回结果。

2.2 大脑/处理核心 (Reasoning & Orchestration)

这是 Agent 的核心控制流，被称为 Orchestration Layer (编排层)。

- **处理逻辑:** 本质是一个循环 (Loop)，文档定义为 "Think, Act, Observe"。
- 状态转移: LLM 作为一个函数 f ，在每一轮计算下一个动作 A_t 或最终输出 O :
$$Action_t = f_{LLM}(Context_t)$$

其中 $Context_t$ 是经过 Context Engineering 处理后的 Prompt 窗口，包含了 Mission、Memory 和

Available Tools。

- **规划 (Planning):** 使用 CoT (Chain-of-Thought) 或 ReAct 范式，将复杂 Mission 分解为子步骤。文档提到了 Meta-Reasoning (元推理)，即 Agent 思考“我缺乏某种能力，需要创建一个新工具”的能力 (Level 4 特性) 。

2.3 工具/行动 (Tools/Action)

- **Schema 定义:** 工具通过 OpenAPI 规范或 MCP (Model Context Protocol) 进行描述，使 LLM 能够理解输入参数和输出结构。
- **动作空间 (Action Space):**
 - **Information Retrieval:** RAG, Google Search.
 - **Software Execution:** API 调用, Python 代码执行 (Sandboxed).
 - **Human Interaction:** HITL (Human-in-the-Loop) 请求确认或输入.
 - **Agent Creation:** 在 Level 4 中，Action 甚至包括“生成一个新的 Agent”。

2.4 输出 (Output)

最终的文本响应。

- **Side Effects (副作用):** 现实世界的改变 (如：发送了邮件、修改了数据库、支付了款项) 。
- **Self-Correction:** 在多智能体系统中，输出可能仅仅是另一个 Agent (如 Critic) 的输入。

图示说明：Agent 的核心 5 步循环：Get Mission -> Scan Scene -> Think -> Act -> Observe -> Iterate。

3. 关键算法与流程 (Algorithm & Workflow)

文档将 Agent 的核心 Loop 形式化为 5 个步骤。这可以看作是一个强化学习中的 Policy Execution 过程，但没有显式的 Reward Function 训练 (除非涉及 RLHF) 。

伪代码描述:

Python

```
1 def agent_loop(mission):
2     # 1. Get Mission
3     current_goal = mission
4     memory = initialize_memory()
5
6     while not task_completed(current_goal):
7         # 2. Scan the Scene (Perception)
8         context = assemble_context(
9             goal=current_goal,
10            history=memory.short_term,
11            knowledge=memory.retrieve(current_goal) # RAG
12        )
13
14         # 3. Think It Through (Reasoning)
15         # LLM decides the plan or next step based on context
16         plan_or_action = llm.generate(context, stop=["\nobservation:])
17
18         if is_final_answer(plan_or_action):
19             return plan_or_action
```

```

20
21     # 4. Take Action (Execution)
22     tool_name, params = parse_tool_call(plan_or_action)
23     observation = execute_tool(tool_name, params)
24
25     # 5. Observe and Iterate (Learning/Updating Context)
26     memory.update(action=plan_or_action, result=observation)
27
28     # Refine Context for next iteration
29     # In Level 2+, context engineering happens dynamically here

```

分级体系 (Taxonomy) 的技术含义:

- **Level 0:** $Output = LLM(Prompt)$ (无状态, 无外部 IO)。
- **Level 1:** $Output = LLM(Prompt + ToolResult)$ (单步工具调用)。
- **Level 2:** 引入 Context Engineering 和 Planning。Agent 能够根据 $Step_{t-1}$ 的结果动态调整 $Step_t$ 的搜索策略 (如: 先找中间点, 再找咖啡店)。
- **Level 3:** Multi-Agent Systems (MAS)。Task 被分解, 分配给专门的 $Agent_A, Agent_B$ 。通信协议升级为 A2A (Agent-to-Agent)。
- Level 4: Self-Evolving。能够生成代码来扩展自身的 Tools 列表。

$$Tools_{t+1} = Tools_t \cup \{GeneratedTool\}$$

4. 实验与评估 (Experiments & Metrics)

文档并未聚焦于具体的 Benchmark 分数 (如 GAIA)，而是强调了 Agent Ops 的评估方法论。这是针对“不可预测性”的系统工程。

评估指标 (Metrics):

- **业务指标:** 目标完成率 (Goal completion rates)、用户满意度 (CSAT)。
- **系统指标:** 延迟 (Latency)、Token 成本 (Cost)、步骤数 (Step count)。

评估方法: LLM as a Judge:

- 不再使用传统的 `assert output == expected`。
- 构建 "Golden Dataset" (包含 Prompt 和理想 Response/Action)。
- 使用高智商模型 (如 Gemini Ultra) 作为 Judge, 对 Agent 的执行轨迹 (Trace) 进行评分。
- 评分维度: 准确性、安全性、是否遵循指令 (Instruction Following)。

调试 (Debugging):

依赖 OpenTelemetry Traces 来记录完整的执行轨迹 (Trajectory)，包括 Prompt、Internal Monologue、Tool Parameters 和 Observation。

5. 复现与工程视角 (Engineering View)

如果你要复现或构建文档中描述的系统，作为研究生，你需要关注以下难点：

5.1 核心难点

- **Context Window 的管理 (Context Engineering):** 文档多次强调 "Agent is a system dedicated to the art of context window curation"。
 - **挑战:** 如何在有限的 Context Window 内，高效地塞入 System Prompt、Tool Definitions、History 和 RAG Result？若 History 过长，如何进行截断或摘要 (Summary) 而不丢失关键状态信息？
- **Orchestration 的稳定性:**
 - **挑战:** 避免无限循环 (Infinite Loops)。Agent 可能陷入 "Think -> Act -> Error -> Think -> Act -> Error" 的死循环。需要设计强制的 Stop Criteria 和 Error Handling 策略（如重试次数限制）。
- **多智能体通信 (A2A Protocol):** 文档提到了 A2A 协议和 Agent Card (类似于 Service Discovery)。
 - **实现:** 需要设计一种异步的消息传递机制，不仅仅是 HTTP Request/Response，可能需要 WebSocket 或 gRPC 来支持 Streaming Updates。

5.2 安全性 (Security)

- **Prompt Injection:** 必须假设所有 User Input 都是恶意的。
- **Guardrails:** 必须实施 "Defense-in-depth" (纵深防御)。
 - **Layer 1:** 确定性规则 (e.g., 禁止金额 > \$100)。
 - **Layer 2:** 模型级防御 (e.g., 使用专门的 "Guard Model" 在执行 Action 前审查 Plan)。
- **Identity:** Agent 需要有独立的身份 (SPIFFE)，与 User 身份区分，实现最小权限原则 (Least Privilege)。

5.3 高级实例：Google Co-Scientist

- **架构:** 这是一个典型的 Level 3/4 系统。
 - **Supervisor Agent:** 资源分配和任务拆解。
 - **Generation Agent:** 提出假设。
 - **Critique/Reflection Agent:** 审查假设，类似于科学辩论。
 - **Evolution Agent:** 基于反馈改进策略。
- 这展示了 "Iterative Refinement" (迭代优化) 设计模式的强大之处。

第二份白皮书：Agent 的感知与执行边界——即“工具 (Tools)”层

1. 核心痛点与贡献 (Core Contribution)

痛点 (Pain Points):

- **"N × M" 集成灾难:** 在 MCP 出现之前，将 N 个模型连接到 M 个数据源/工具，需要开发 $N \times M$ 个定制连接器 (Connectors)。这种 $O(N \times M)$ 的复杂度导致了生态系统的碎片化和极高的维护成本。
- **Context Window Bloat (上下文膨胀):** 传统的 Tool Use 需要将所有可用工具的 Schema 塞入 Prompt。随着工具数量增加，Context Window 迅速被元数据填满，挤压了推理空间，甚至导致模型注意力分散。

- **安全边界模糊**: Agent 经常面临 "Confused Deputy" (糊涂代理人) 攻击, 即攻击者利用高权限的 Tool 绕过安全检查。

核心贡献 (Key Contributions):

- **MCP 标准化**: 提出了基于 JSON-RPC 2.0 的客户端-服务器 (Client-Server) 架构, 类似于 IDE 领域的 LSP (Language Server Protocol), 实现了 Agent 与工具实现的解耦。
- **反向控制流 (Inversion of Control)**: 引入了 Sampling 和 Elicitation 机制, 允许工具端 (Server) 反向请求 Agent (Client) 进行推理或请求用户输入, 打破了单向调用的限制。
- **企业级安全规范**: 详细定义了动态能力注入 (Dynamic Capability Injection) 和工具遮蔽 (Tool Shadowing) 等新型威胁模型及防御策略。

2. 系统架构 (System Architecture)

MCP 将 Tool 的交互抽象为一个标准化的通信协议。

2.1 拓扑结构 (Topology)

系统被划分为三个实体:

- **MCP Host (主机)**: 发起方, 通常是 Agent 应用 (如 Claude Desktop, Vertex AI Agent) 。负责管理生命周期和用户交互。
- **MCP Client (客户端)**: Host 内部的连接器, 负责与 Server 维持 Session (1:1 连接)。
- **MCP Server (服务端)**: 提供能力的实体 (如 Google Drive Connector, Postgres Interface) 。它暴露 Tools, Resources, Prompts 给 Client。

2.2 通信层 (Communication Layer)

- **协议**: 基于 JSON-RPC 2.0。
- **Message Types**: Request, Result, Error, Notification。
- **传输方式 (Transports)**:
 - **stdio**: 用于本地进程 (Subprocess) 。Server 作为 Host 的子进程运行, 通过标准输入输出通信 (低延迟, 高安全性) 。
 - **Streamable HTTP (SSE)**: 用于远程连接。支持 Server-Sent Events (SSE) 以实现流式传输, 取代了早期仅依赖 WebSocket 的方案。

2.3 核心原语 (Key Primitives)

MCP 定义了六种核心能力, 分为两类:

Server Capabilities (服务端提供):

- **Tools**: 可执行函数 (e.g., `get_weather`)。
- **Resources**: 静态数据或文件流 (e.g., Log files, Database records)。
- **Prompts**: 预定义的 Prompt Templates, 允许 Server 指导 Agent 如何使用它。

Client Capabilities (客户端提供):

- **Sampling**: Server 请求 Agent 的 LLM 进行推理 (例如: Server 抓取了长文, 请求 Agent 总结) 。
- **Elicitation**: Server 请求 Host 弹窗询问用户 (Human-in-the-loop) 。

- **Roots:** 定义文件系统的访问边界 (Sandbox) 。

3. 关键算法与流程 (Algorithm & Workflow)

3.1 工具发现与调用循环 (Discovery & Invocation Loop)

这是一个动态绑定的过程，而非静态编译。

1. **Handshake:** Client 连接 Server，协商协议版本和能力 (Capabilities)。
2. **Discovery:** Client 发送 `tools/list` 请求。
3. **Context Injection:** Host 将获取到的 Tool Definitions (JSON Schema) 注入到 LLM 的 System Prompt 中。
4. **Reasoning:** LLM 生成 Tool Call (e.g., `{"name": "fetch_stock", "args": {"symbol": "GOOG"}}`)。
5. **Execution:** Client 通过 MCP 发送 `tools/call` Request。
6. **Result:** Server 执行并返回 Result (Text/Image/Error)。

3.2 采样机制 (Sampling Workflow) - Level 4 Agent 特性

这是一个反直觉但极其强大的流程，Server 反过来利用 Agent 的大脑：



这允许“哑”工具 (Dumb Tools) 利用 Agent 的智能来处理中间数据，而无需将原始大数据回传给 Agent。

4. 安全性深度分析 (Security Analysis)

这是文档最硬核的部分 (Section 5)，揭示了 Agent Security 的新战场。

4.1 威胁模型 (Threat Landscape)

- **Dynamic Capability Injection (动态能力注入):**
 - **攻击:** 一个名为 "Book Search" 的 Server 突然增加了一个 `buy_book` 工具。Agent 可能在用户不知情的情况下被诱导调用支付接口。
 - **防御:** Tool Pinning (锁定工具哈希) 和 Change Notification (变更强制通知)。
- **Tool Shadowing (工具遮蔽):**
 - **攻击:** 恶意 Server 注册一个名为 `save_secure_note` 的工具，描述写得比官方工具更诱人 (Prompt Hacking)。LLM 可能错误地将敏感数据存入恶意 Server 而非官方 Vault。
 - **防御:** Namespace Isolation 和 LLM-based Name Collision Detection。
- **Confused Deputy (糊涂代理人问题):**
 - **场景:** 用户无权访问代码库，但 Agent (Deputy) 有。用户通过 Prompt Injection 让 Agent 提取代码。
 - **本质:** 权限检查发生在了 Agent 层，而不是 User 层。
 - **防御:** Scoped Credentials 和 Audience Binding。Agent 调用 Tool 时必须传递用户的身份 Token，而非 Agent 自身的 Token。

4.2 污点分析 (Taint Analysis)

文档建议引入“污点源/汇” (Taint Sources/Sinks) 概念：

- **Tainted Source**: 用户输入、第三方 Server 返回的数据。
- **Sensitive Sink**: 发送邮件、写入数据库、执行代码。
- **Rule**: 数据流从 Tainted Source 到 Sensitive Sink 必须经过 Sanitization 或 Human Approval。

5. 实验与工程视角 (Engineering View)

5.1 核心瓶颈: Context Window Bloat

文档坦诚地指出了 MCP 当前架构的一个严重缺陷：扩展性。如果连接了 100 个 Server，每个 Server 有 50 个 Tools，总共有 5000 个 Tool Definitions。将这些全部塞入 Prompt 是不可行的 (Token 昂贵且影响精度)。

解决方案 (RAG for Tools):

不再一次性加载所有工具，而是建立一个 Tool Index。

1. **User Query**: "帮我查一下上季度的销售额。"
2. **Agent 检索 (Retrieve)**: 在 Tool Vector DB 中搜索与 "销售、查询" 相关的工具。
3. **加载 (Load)**: 仅将 Top-5 相关的 Tool Schema 动态加载到 Context 中。
4. **执行 (Execute)**。

5.2 状态管理 (State Management)

MCP over HTTP 是无状态的，但复杂的 Agent 任务通常是有状态的。工程上需要实现 Stateful Middleware 或者在 MCP Server 端自行维护 Session ID，这增加了分布式系统的复杂性。

总结与思考

这份文档实际上是在定义 AI 时代的 TCP/IP 协议。

作为研究生的 Critical Thinking：

- **协议的厚度**: MCP 是否过重？JSON-RPC 加上复杂的 Capabilities 协商，对于简单的 Tool Use 是否过度设计？
- **安全责任转移**: MCP 将安全责任下放给了 Server 和 Host 的实现者。在“Confused Deputy”场景中，如何通过密码学手段（如 ZK-Proofs 或 OAuth 2.0 extension）在协议层强制执行身份传递，而不仅仅是依靠“建议”？
- **Sampling 的风险**: 允许 Server 反向调用 Agent LLM 极大地增加了 Prompt Injection 的攻击面。恶意的 Server 可以通过 Sampling Request 注入 Jailbreak Prompt，从而控制整个 Agent。这是 Level 4 自主系统面临的主要安全挑战。

第三份白皮书：智能体的记忆

1. 核心痛点与贡献 (Core Contribution)

痛点 (Pain Points):

- **无状态的本质**: LLM 本质上是函数 $f(x) = y$ ，每次调用都是独立的，无法自动“记住”上一轮的 x 。
- **Context Window 的稀缺性**: 尽管 Gemini 1.5 Pro 拥有 2M Token 的窗口，但在长程任务中，未经筛选的历

史堆叠会导致 "**Lost in the Middle**" 现象，且推理成本 (Cost) 和延迟 (Latency) 会随窗口长度线性（甚至超线性）增长。

- **信息过载**: 直接将所有历史信息塞入窗口不仅昂贵，还会引入噪音，降低模型的推理准确率。

核心贡献 (Key Contributions):

- **Context Engineering 定义**: 将 Prompt Engineering 升级为 **Context Engineering**，即“系统性地设计、策划和优化输入给模型的信息的过程”。
- **上下文分层架构**: 提出了由 **System Instructions, Few-Shot Examples, Grounding Data, Session History** 组成的四层上下文结构。
- **记忆二分法**: 明确区分了 **Session** (会话/短期记忆) 和 **Memory** (持久化/长期记忆) 的工程实现边界。

2. 系统架构 (System Architecture)

文档将 Context 视为 Agent 的核心资源，并提出了一个分层管理的架构。

2.1 上下文构成 (The Anatomy of Context)

一个标准的 Agent Context Window (C_{total}) 由以下部分组成：

$$C_{total} = C_{system} + C_{examples} + C_{memory} + C_{session} + C_{query}$$

1. System Instructions (MIC - Meta-Information Context):

- Agent 的“宪法”或“核心指令”。定义了角色 (Persona)、语气、边界条件和输出格式（如 JSON Schema）。
- 这是静态的，通常在所有交互中保持不变。

2. Few-Shot Examples:

- 通过提供 input-output 对，利用 **In-Context Learning (ICL)** 引导模型的行为。文档强调这是“通过演示教学” (Teaching by demonstration)。

3. Grounding Data (RAG):

- 这是动态检索的外部知识。它是为了减少幻觉 (Hallucination) 并提供模型训练截止日期之后的信息。

4. Session History:

- 当前对话的即时状态，包含用户的上一句话和模型的上一句回复。

2.2 记忆架构 (Memory Architecture)

文档提出了两种核心的时间尺度：

- **Session (短期记忆/Short-Term Memory)**:

- **存储位置**: 内存或轻量级缓存 (Redis)。
- **生命周期**: 随对话结束而销毁。
- **作用**: 维持多轮对话的连贯性（如：用户说“它多少钱？”，Agent 需要从 Session 中知道“它”指代上一轮提到的产品）。
- **管理策略**:

- **Sliding Window (滑动窗口)**: 只保留最近 k 轮。

■ **Summarization (摘要)**: 将旧的 N 轮对话压缩为一个 Summary string, 释放 Token 空间。

- **Persistence Memory (长期记忆/Long-Term Memory)**:

- **存储位置**: Vector Database (向量数据库) 或 SQL Database。
- **生命周期**: 跨越 Session 存在, 永久保存。
- **类型**:
 - **Episodic Memory (情景记忆)**: 记录过去的事件序列 (“用户上次预订了靠窗的位置”)。
 - **Semantic Memory (语义记忆)**: 记录从交互中提取的事实和偏好 (“用户不喜欢吃辣”)。
- **实现**: 通常通过 **Vector Embeddings** 实现语义搜索。

3. 关键算法与流程 (Algorithm & Workflow)

3.1 上下文组装流程 (Context Assembly Pipeline)

这是一个在 Runtime 进行的动态组装过程。

伪代码描述:

Python

```
1 def assemble_context(user_query, session_id, user_id):  
2     # 1. Load Static Context (System Instructions & Examples)  
3     # Optimization: Use Context Caching here to save cost  
4     static_context = load_cached_prompt("agent_persona_v1")  
5  
6     # 2. Retrieve Short-term Session History  
7     raw_history = session_store.get(session_id)  
8     # Apply strategy: Summarize if too long  
9     if token_count(raw_history) > LIMIT:  
10         session_context = llm.summarize(raw_history)  
11     else:  
12         session_context = raw_history  
13  
14     # 3. Retrieve Long-term Memory (Semantic Search)  
15     query_embedding = embed_model.encode(user_query)  
16     # RAG: Find relevant past interactions or knowledge  
17     relevant_memories = vector_db.search(query_embedding, user_id, top_k=5)  
18  
19     # 4. Assemble Final Prompt  
20     final_prompt = f"""\n{static_context}\n\nRelevant Memories:\n{relevant_memories}\n\nCurrent Conversation:\n{session_context}\n\nUser: {user_query}\n.....
```

3.2 优化算法: Context Caching (上下文缓存)

这是一个非常关键的工程优化，文档特别提到。

对于 C_{system} 和 $C_{examples}$ 这种包含大量 Token 且不经常变化的内容，每次 API 调用都重复传输和处理是极大的浪费。

- **原理:** 将预处理后的 KV Cache (Key-Value states of the Transformer) 存储在服务端。
- **收益:** 显著降低 **Time to First Token (TTFT)** 和推理成本。

4. 实验与评估 (Experiments & Metrics)

文档未提供具体的 Benchmark 数据，但定义了评估 Context Engineering 质量的维度：

- **Information Retrieval Metrics (信息检索指标):**
 - **Recall (召回率):** 是否检索到了回答问题所需的所有关键事实？
 - **Precision (精确率):** 检索到的上下文中，有多少是真正相关的？（低 Precision 意味着 Context 噪音大）。
- **Generation Metrics (生成指标):**
 - **Faithfulness (忠实度):** 回答是否严格基于检索到的 Context，而非模型内部的幻觉？
 - **Context Adherence (上下文依从性):** 模型是否遵循了 System Instructions 中的格式约束？

5. 复现与工程视角 (Engineering View)

作为研究生，在构建这套系统时需要注意以下工程难点：

5.1 状态管理的复杂性

- **难点:** 随着对话进行，Session History 会无限增长。
- **策略:** 不要仅仅截断 (Truncate)。简单的截断会导致丢失开头的关键指令。
- **最佳实践:** 采用 "System Prompt + Summary of Old History + Rolling Window of Recent Turns" 的混合模式。

5.2 长期记忆的“读写”矛盾

- **写入 (Writing):** 什么时候将 Session 转化为 Long-term Memory?
 - 方案 A: 每一轮都存（噪音极大）。
 - 方案 B: 会话结束时，用 LLM 提取关键 Fact 存入（推荐）。
- **读取 (Reading):** 纯向量检索 (Vector Search) 可能会丢失时间信息。
 - **问题:** 用户问“我现在的地址是？”，如果向量数据库里有 3 个历史地址，单纯的相似度搜索可能无法区分哪个是“最新的”。
 - **解决:** 需要在 Metadata 中加入 Timestamp，并结合 SQL 过滤 (Hybrid Search)。

5.3 隐私与合规 (Privacy)

- 文档在末尾强调了隐私。Memory 本质上是在存储用户的数字指纹。
- **挑战:** 如何实现“被遗忘权” (Right to be forgotten) ? 如果用户要求删除数据, 你如何在 Vector DB 中精准定位并物理删除特定的 Embedding Chunk? 这比传统数据库删除要复杂得多。

总结与思考

这份文档的核心论点是: **Context Engineering 是 AI 应用开发的新前沿。**

互动思考:

1. **记忆的污染 (Memory Poisoning):** 如果模型产生了一次幻觉, 并且这个幻觉被作为“事实”存入了 Long-term Memory, 那么这个错误会在未来的交互中被无限放大。你认为该引入什么样的 **Memory Verification (记忆校验) 机制?**
2. **Context Caching 的经济学:** Google 的 Context Caching 通常有 TTL (Time To Live) 限制。在设计系统时, 你需要计算 Token 吞吐量的临界点 (Break-even point) , 判断何时开启 Caching 才是划算的。

第四份白皮书：智能体的质量和评估

1. 核心痛点与贡献 (Core Contribution)

痛点 (Pain Points):

- **单元测试失效:** 传统的软件测试依赖 `assert expected == actual`。但对于 LLM, 即使是同一个 Prompt, 两次运行的 Output 字面量也可能不同 (语义相同但措辞不同), 导致传统 Assert 无法使用。
- **"Works on My Machine":** 开发者在少量样本上手动测试通过, 但在生产环境面对多样化的 User Prompts 时, Agent 表现出脆弱性 (如指令遵循失败、幻觉) 。
- **评估的主观性:** 什么是“好”的回答? 仅靠人类评估 (Human Eval) 难以扩展 (Scale) 且昂贵。

核心贡献 (Key Contributions):

- **Agent Ops 定义:** 提出了 **Agent Ops** 这一新学科, 它是 MLOps 和 DevOps 的结合, 专注于 Agent 的生命周期管理。
- **LLM-as-a-Judge:** 确立了以“大模型作为裁判”的自动化评估范式, 解决了语义一致性判断的难题。
- **评估驱动开发 (EDD):** 提出了 **Evaluation Driven Development**, 类比于 TDD (Test Driven Development), 要求在开发 Agent 之前先定义好评估集。

2. 系统架构 (System Architecture)

这里的“系统”特指 **评估流水线 (Evaluation Pipeline)**。

2.1 评估数据集 (The Evaluation Dataset)

文档强调必须构建 "**Golden Dataset**" (黄金数据集)。这通常包含以下字段:

- `Input Prompt`: 用户的输入。
- `Reference output` (Ground Truth): 期望的理想回答 (由专家撰写) 。
- `Context` (Optional): RAG 检索到的文档片段。

2.2 评估器架构 (Evaluator Architecture)

评估过程不再是简单的字符串比对，而是一个多维度的评分系统：

1. **执行 (Execution):** Agent 运行 `Input Prompt`，生成 `Actual Output` 和 `Trace`（包含中间推理步骤和工具调用）。
 2. **计算 (Computation):**
 - **基于规则 (Rule-based):** 检查 JSON 格式是否合法、代码能否编译运行、不仅包含特定关键词。
 - **基于模型 (Model-based):** 将 `Input`, `Actual Output`, `Reference Output` 发送给 **Judge Model**。
 3. **裁判 (Judge):**
 - Judge Model 根据预定义的 **Rubric (评分细则)** 进行打分（如 1-5 分）并给出理由。
 - Prompt 示例: "你是评分专家，请判断 `Actual Output` 是否包含了 `Reference Output` 中的所有关键信息..."
-

3. 关键算法与指标 (Algorithm & Metrics)

文档详细列举了不同维度的评估指标，这些指标定义了 Agent 的“质量”。

3.1 确定性指标 (Deterministic Metrics)

适用于结构化输出或代码生成：

- **Code Execution:** 生成的代码是否通过了单元测试？
- **Tool Usage:** 是否调用了正确的工具？参数是否符合 Schema？
- **JSON Validity:** 输出是否是合法的 JSON 对象？

3.2 概率性/语义指标 (Stochastic Metrics)

适用于文本生成，通常依赖 LLM Judge 或 NLP 算法：

- **传统 NLP:** `BLEU`, `ROUGE`（基于 n-gram 重叠，但在 Agent 场景下往往相关性不强）。
- **语义相似度:** 使用 Embedding 模型计算 `CosineSimilarity(Vector(Actual), Vector(Reference))`。
- **RAG 专用指标 (Ragas 框架思想):**
 - Faithfulness (忠实度): 回答是否完全基于检索到的 Context? (检测幻觉)。

$$Score = \frac{\text{Number of claims supported by context}}{\text{Total claims in response}}$$

- **Answer Relevance (回答相关性):** 回答是否直接解决了用户的问题？
- **Context Precision (上下文精度):** 检索到的文档中，相关文档的排序是否靠前？

3.3 评估工作流 (Evaluation Workflow)

文档描述了一个闭环流程：

1. **Author:** 编写 Prompt 和 Agent 逻辑。
2. **Run:** 在 Golden Dataset 上运行。
3. **Evaluate:** 计算上述指标。

4. **Analyze**: 如果分数低，检查 Trace。是检索失败？还是推理错误？

5. **Refine**: 优化 Prompt 或 RAG 策略，重复步骤 2。

4. 实验与工程视角 (Engineering View)

作为研究生，在落地这套 QA 体系时，你需要关注以下工程挑战：

4.1 "Evaluating the Evaluator" (评估裁判)

- **问题**: LLM Judge 自身也会犯错。如果 Judge 说 Agent 错了，但其实 Agent 是对的怎么办？
- **解决**: 文档提到需要 **Meta-Evaluation**。你需要构建一个小型的、由人类专家标注的“评估集之上的评估集”，用来测试 Judge Model 的准确率（与人类的一致性）。

4.2 成本与延迟

- **工程视角**: 使用 GPT-4 或 Gemini Ultra 作为 Judge 非常昂贵且缓慢。
- **优化**:
 - **Pairwise Evaluation (成对评估)**: 让模型比较两个版本 (A vs B) 哪个更好，通常比直接打分更准确。
 - **Small Model as Judge**: 蒸馏一个小模型（如专门微调过的 Llama 3 8B）专门做评分任务，以降低 CI/CD 成本。

4.3 数据污染 (Data Contamination)

- **风险**: 如果你的 Evaluation Dataset 不小心混入了模型的训练数据 (Pre-training corpus)，或者被用于了 Few-shot examples，那么评估结果就是虚高的 (Overfitting)。必须严格隔离 **Train/Dev/Test** 集合。

总结与思考

这份文档将 AI 开发从“炼金术”推向了“化学工程”。**Agent Quality 不再是凭感觉，而是通过 CI/CD 流水线中的具体 Metric 来量化的。**

互动思考：

1. **评估的脆弱性**: 针对 RAG 的 **Faithfulness** 指标，如果检索到的 Context 本身就是错误的（源数据错误），Agent 忠实地回答了错误信息，这时候 Faithfulness 分数很高，但事实是错的。如何在评估中解耦 **Retrieval Error** 和 **Generation Error**？
2. **对抗性评估 (Red Teaming)**: 文档简要提到了安全性。在实际工程中，除了 Golden Dataset，通常还需要一个 **Red Teaming Dataset**（包含诱导攻击、Prompt Injection），专门用来测试 Agent 的防御能力。这也是 Agent Quality 的重要组成部分。

第五份白皮书：智能体的协议

如果说前四份文档分别构建了 Agent 的器官（大脑、手、记忆、免疫系统），那么这份文档则是在讨论如何将这些器官组装成一个**可生存、可扩展、可协作的社会化实体**。它解决的是从“Jupyter Notebook 里的 Demo”到“7x24小时运行的企业级服务”之间的巨大鸿沟，并提出了极其前沿的 **A2A (Agent-to-Agent)** 互联协议。

以下是对该文档的深度技术拆解。

1. 核心痛点与贡献 (Core Contribution)

痛点 (Pain Points):

- **Demo Trap (演示陷阱)**: 许多 Agent 在本地运行良好，但在生产环境中因为并发、延迟、状态漂移或版本管理混乱而崩溃。
- **孤岛效应 (Silo Effect)**: 目前的 Agent 大多是孤立运行的单一应用，无法跨组织、跨平台协作（例如：你的旅行 Agent 无法直接与航空公司的订票 Agent 对话）。
- **运营盲区**: 缺乏对 Agent 在线行为的监控，无法形成“数据飞轮”来持续优化模型。

核心贡献 (Key Contributions):

- **全生命周期定义**: 提出了包含 Design, Develop, Evaluate, Deploy, Monitor, Refine 的闭环生命周期。
- **A2A 协议 (Agent-to-Agent Protocol)**: 定义了一种标准化的 Agent 互联语言，使 Agent 能够相互发现、握手并协作完成任务，这是迈向 "Internet of Agents" 的关键一步。
- **反馈循环架构**: 构建了从 Implicit/Explicit Feedback 到模型微调 (Fine-tuning) 或 Prompt 更新的自动化链路。

2. 系统架构 (System Architecture)

生产级 Agent 系统不再是一个脚本，而是一个微服务架构。

2.1 部署架构 (Deployment Architecture)

文档建议将 Agent 容器化 (Dockerized) 并运行在 Serverless 或 Kubernetes 环境中。

- **Runtime**: 必须支持长时间运行的连接 (Long-running connections)，因为 Agent 任务可能耗时数分钟。
- **State Management**: 必须将状态 (Session) 外置到 Redis 或 Database，以支持服务的无状态扩展 (Stateless Scaling)。
- **Sidecars**: 可以在 Agent 容器旁部署 Sidecar 负责日志 (Logging)、追踪 (Tracing/OpenTelemetry) 和安全拦截 (Guardrails)。

2.2 反馈闭环 (The Data Flywheel)

系统必须设计为闭环控制系统：

1. **Capture**: 捕获 (Input, Output, Trace, User Feedback) 四元组。
 - **Implicit Feedback**: 用户接受了建议？用户是否重写了 Prompt？
 - **Explicit Feedback**: Thumbs up/down, star rating.
2. **Filter**: 筛选出“负样本”（错误案例）和“高价值正样本”。
3. **Refine**:
 - **Level 1**: 更新 Few-shot Examples (放入 Context)。
 - **Level 2**: 更新 System Instructions。
 - **Level 3**: 触发 Fine-tuning 流程 (针对基座模型)。

3. 关键算法与协议：A2A (Agent-to-Agent)

这是本文档最具学术价值的部分。A2A 协议试图解决多智能体协作中的 **Discovery** (发现) 和 **Communication** (通信) 问题。

3.1 协议栈 (Protocol Stack)

类似于 TCP/IP 模型，A2A 可以分层描述：

1. Discovery Layer (发现层):

- **Agent Card (Manifest)**: 每个 Agent 必须发布一个 JSON 描述文件，声明其 `Capabilities` (能做什么), `Interfaces` (输入输出 Schema), 和 `Pricing` (如果涉及经济模型)。
- **Registry**: 一个去中心化或联邦式的目录服务，用于存储和检索 Agent Cards。

2. Handshake Layer (握手层):

- Agent A (Client) 向 Agent B (Server) 发起连接。
- **Identity Verification**: 使用 SPIFFE 或 mTLS 验证身份。
- **Contract Negotiation**: 确认 B 是否接受 A 的任务请求，以及费用/Quota 限制。

3. Task Layer (任务层):

- 通信模式不再是简单的 Request-Response，而是 **Asynchronous Task Execution** (异步任务执行)。
- 状态机:

$State \in \{\text{Pending}, \text{Accepted}, \text{Running}, \text{Streaming}, \text{Completed}, \text{Failed}\}$

- Agent B 可以通过流式 (Streaming) 返回中间思考过程 (Thoughts) 或部分结果，让 Agent A 保持感知。

3.2 协作流程示例

假设 "Travel Agent" (A_T) 需要调用 "Airline Agent" (A_A):

1. **Search**: A_T 查询 Registry: `Find agent where capability == "book_flight"`。
2. **Connect**: A_T 发送握手请求给 A_A ，附带自身签名。
3. **Task**: A_T 发送 Payload: `{"origin": "SFO", "dest": "JFK", "date": "2025-11-20"}`。
4. **Async Process**: A_A 返回 `TaskID: 12345, Status: Processing`。
5. **Webhook/Poll**: A_A 完成出票，推送结果 `{"ticket_id": "XYZ"}` 给 A_T 。

4. 实验与监控 (Experiments & Monitoring)

在生产环境中，关注点从“准确率”转向了“健康度”和“业务价值”。

• A/B Testing:

- **Shadow Mode (影子模式)**: 新版本的 Agent 在后台与旧版本并行运行，处理相同的 User Input，但不向用户展示结果。通过比较影子版本的 Output 与旧版本的 User Feedback 来评估风险。
- **Canary Deployment (金丝雀发布)**: 将 1% 的流量切给新 Agent。

• Operational Metrics:

- **Latency:** P95 和 P99 延迟 (Agent 通常较慢, 需要关注长尾)。
 - **Cost per Interaction:** 每次对话的 Token 成本。
 - **Throttling Rate:** 触发 API 限流的频率。
-

5. 复现与工程视角 (Engineering View)

作为研究生, 构建 A2A 网络或生产级 Agent 面临的挑战:

5.1 身份与信任 (Identity & Trust)

- **挑战:** 在开放网络中, 如何防止恶意 Agent 伪装成 "Official Bank Agent"?
- **方案:** 必须建立 **PKI (Public Key Infrastructure)** 体系。Agent 的身份必须由可信 CA 签名。Agent Card 必须包含数字签名。
- **Confused Deputy:** 在 A -> B -> C 的调用链中, C 如何确认最初的发起者是 User? 需要实现 **Delegated Authorization** (委托授权), 类似于 OAuth 的 `on-Behalf-of` 流程。

5.2 经济模型 (Agent Economy)

- 文档展望了 **Agent Payments**。
- **挑战:** Agent A 调用 Agent B, B 需要消耗算力。谁来买单?
- **微支付 (Micropayments):** 可能集成 HTTP 402 (Payment Required) 协议或区块链技术, 实现机器对机器 (M2M) 的毫秒级支付。

5.3 调试分布式系统

- 当 Agent A 报错时, 根因可能在 Agent B 调用的 Agent C 的 Tool D 上。
- **分布式追踪 (Distributed Tracing):** 必须在 A2A 协议头中透传 `Trace-ID`, 并在统一的观测平台 (如 Jaeger/Grafana) 中重建完整的调用链。

总结与思考

这份文档描绘了 "**The Internet of Agents**" 的蓝图。它预示着未来的软件交互方式将从 **API 调用** 演变为 **Agent 协商**。

互动思考:

1. **协议的标准化:** 目前 A2A 尚无全球统一标准 (Google 提出了自己的, OpenAI 有 Swarm)。你认为未来的 Agent 互联协议会像 HTTP 一样统一吗? 还是会形成多个生态壁垒?
2. **死锁与环路 (Deadlocks & Loops):** 在 A2A 网络中, 如果 Agent A 等待 B, B 等待 C, C 又请求 A, 如何检测并打破这种分布式死锁?
3. **社会化风险:** 如果 Agent 可以自主支付和雇佣其他 Agent, 如何防止它们在毫秒级时间内耗尽用户的预算? 这需要什么样的 **Financial Guardrails**?