7. 应用：智能体

Table of contents

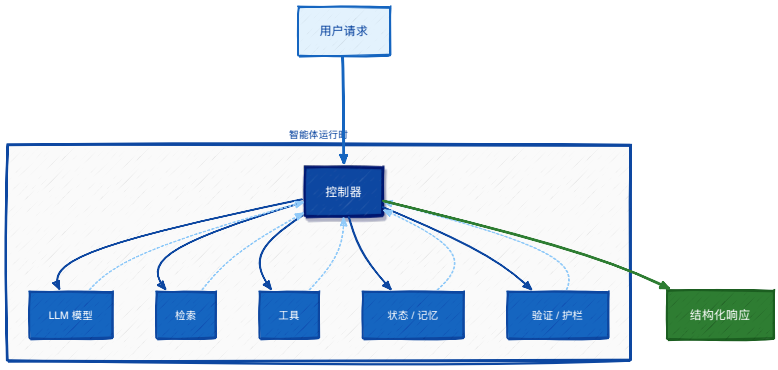

Figure 1: 系统大于模型——一个现代智能体系统总图

1 概览

一位财务工程师问内部助手："帮我总结上周影响计费系统的变更，并告诉我哪些企业客户可能受到了影响。"答案几秒钟就出来了。它写得很像样，简洁、流畅，而且是错的。系统漏掉了一条迁移说明，忽略了一次支持升级，还把受影响客户标错了，因为它从来没有检查过生产状态。另一次事故里，客服让系统发起退款，运行时在超时后重试，结果退款被执行了两次。这里没有任何一件事属于"AI 魔法"。它们都是系统失败。

这就是模型演示与 AI 应用的差别。一个真正的应用不会只生成文字。它会检索证据、调用 API、管理状态、验证结果、执行权限边界，并决定什么时候应该停止。一旦 AI 开始接触真实世界，难点就不再只是 prompting，而是系统工程。

用户并不是直接与模型对话，而是与运行时对话。运行时决定模型如何被使用、检索哪些证据、允许调用哪些工具、保留哪些状态，以及哪些验证器必须先放行结果。模型位于中心，但模型本身不是系统。

一个很有用的工程近似是：

$$P(\text{任务成功}) \neq P(\text{模型答对})$$

更具体一点，可以写成：

$$P(\text{任务成功}) \approx P(\text{找到足够证据}) \cdot P(\text{选对动作}) \cdot P(\text{答案有 grounding}) \cdot P(\text{通过验证}).$$

这不是在声称这些事件统计独立。它表达的是一种系统分解：可靠性不是只在模型内部获得的，而是在多个组件上共同挣出来的。

因此，本章真正要回答的问题是：

> 工程师如何构建能够与真实世界交互、并且仍然可靠的 **AI** 系统？

2 智能体是什么

2.1 智能体是控制器，而不是“会说话的模型”

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 什么是“智能体应用”，它与单次 LLM 调用有何不同？
> 2. 工作流和智能体该怎么选？
> 3. 为什么现代 AI 应用的关键是系统，而不是模型本身？

一个很有用的定义是：智能体是一个控制器，它协调模型、检索、工具、状态和验证，去完成一个可能需要多步推进的任务。关键词是协调。模型并不直接构成应用；应用是围绕模型构建出来的一整套执行系统。

可以把它形式化成：

$$a_t \sim \pi_\theta(a \mid o_t, s_t, b_t),$$

其中 (o_t) 是当前观察，(s_t) 是显式任务状态，(b_t) 是剩余预算，(a_t) 是下一步动作。这个动作可能是“直接回答”“执行检索”“调用工具”“请求澄清”，或者“停止”。动作执行后，系统得到新的结果，并更新状态：

$$s_{t+1} = f(s_t, o_t, a_t, y_t), \qquad b_{t+1} = b_t - c(a_t).$$

这也解释了为什么智能体不是“加了 prompt 的模型”。它是一个在状态、成本和风险约束下运行的控制器。

从工程上看，单次 LLM 调用很适合总结、改写、分类、抽取这类相对封闭的任务。但一旦任务需要外部事实、跨系统动作、多轮纠错或阶段性决策，一次调用就不够了。系统必须进入“观察—决策—行动—检查”的循环，这才是智能体真正开始发挥作用的地方。

Workflow、Agent，还是 Agentic RAG？

最稳妥的工程默认值是：能用 **workflow** 解决，就不要急着上 **agent**。在生产里，低方差通常比额外自治更值钱。

请从上到下阅读图 Figure 2。如果步骤可以穷举并测试，就优先用 workflow；如果主要问题只是需要 grounding，而不需要动态规划，那么 `workflow + RAG` 通常已经足够；只有当系统真的需要迭代搜索、重规划或与实时世界状态交互，并且你已经具备验证器和预算控制时，agent 才是合理抽象。

一个更实用的决策表是：

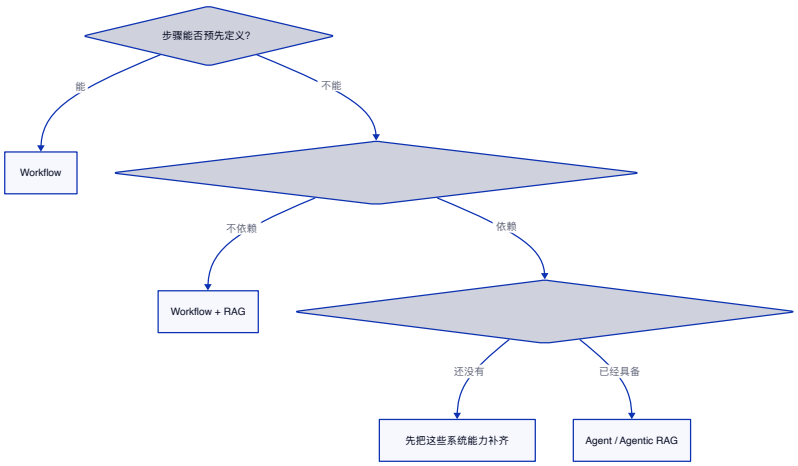

Figure 2: Workflow 与 Agent 的决策框架。

场景	更稳妥的默认方案
步骤稳定、可预定义	Workflow
需要外部知识，但本质仍是单轮问答	Workflow + RAG
需要多步搜索、重规划或跨工具判断	Agentic RAG 或 Agent
操作风险很高，但验证器、预算、停止条件还不成熟	暂时不要上线 Agent 自治

之所以要这么判断，是因为现实里大多数失败都不是“模型太小”，而是“系统没有为不确定性设计边界”。没有显式状态对象，你就无法恢复和回放；没有工具契约，就无法安全执行；没有验证层，就无法把概率系统约束成产品。

2.2 智能体控制循环：从一次回答，到一个迭代系统

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 智能体控制循环的最小形式是什么？
> 2. 为什么智能体不是“一次生成答案”，而是一个迭代系统？
> 3. 反馈回路为什么能提升复杂任务的成功率？

理解智能体最干净的方式，不是背某个框架名，而是掌握一条控制循环：

感知 $\longrightarrow$ 思考 $\longrightarrow$ 行动 $\longrightarrow$ 检查 $\longrightarrow$ 更新状态 $\longrightarrow$ 重复

每一次行动之后，系统都可以重新进入感知阶段。这正是智能体与单次生成的根本差别。

最小执行模型可以写成：

$$a_t = \pi(o_t, s_t, b_t), \quad y_t = \text{Execute}(a_t), \quad v_t = \text{Verify}(y_t), \quad s_{t+1} = \text{Update}(s_t, y_t, v_t).$$

当成功条件满足或预算耗尽时，循环终止：

$$\text{done}(s_t) = 1 \quad \vee \quad b_t \leq 0.$$

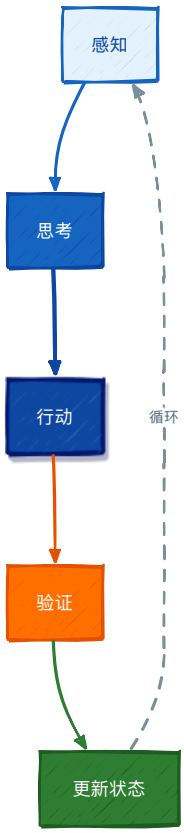

Figure 3: 智能体控制循环——感知、思考、行动、验证、更新

- 更新状态：沉淀会影响后续决策的事实、结论和进度。

很多所谓的“agent demo”其实只有“思考”和“行动”，几乎没有“检查”和“更新状态”，因此一旦进入长任务，很快就会飘掉。

2.3 提示、RAG、工具与技能在循环中的位置

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 提示、检索、工具和技能分别解决什么问题？
> 2. 为什么它们都不是“智能体本身”？
> 3. 它们如何被控制循环编排？

“什么会改变下一步决策？”提示和契约影响思考方式与输出形状；RAG 把证据送入感知与思考；工具把行动连接到外部世界；技能提供可复用的过程知识，例如研究流程、客服 SOP 或内部操作手册。

一个很好的工程类比是新人入职：

- 工具是他能使用的软件和 API；
- 技能是团队已有的 SOP、模板和脚本；
- **RAG** 是他能查阅的文档和知识库；
- 提示与契约是岗位说明和交付格式；
- 智能体则是那个决定“现在该查什么、做什么、何时停止”的控制器。

这个区分之所以重要，是因为很多团队试图靠“写更长的 prompt”来解决智能体问题。现实里，更稳的收益通常来自更好的契约、更好的检索、更清晰的工具接口和更严格的验证。

3 系统架构

3.1 智能体系统的核心构件

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 一个现代智能体系统通常由哪些组件构成？
> 2. 策略、规划器、状态、工具和验证之间是什么关系？
> 3. 为什么架构决策通常比再改一轮 prompt 更重要？

把一个生产级智能体系统拆开，通常会看到这些构件：

- 策略（**Policy**）：模型选择、路由逻辑、高层动作偏好；
- 提示 / 契约层（**Prompt / Contracts**）：输出 schema、指令、重试约束和结果形状；
- 规划器（**Planner**）：可选的任务分解与下一步决策；
- 检索器（**Retriever**）：访问外部知识；

Figure 4: **智能体核心构件——策略、规划器、检索器、工具、验证器、终止条件**

- 工具接口（**Tool Interface**）：以结构化方式访问外部系统；
- 状态 / 记忆（**State / Memory**）：保存显式任务进度；
- 验证器 / 护栏（**Verifier / Guardrails**）：做验收、安全检查和升级；
- 终止逻辑（**Termination**）：预算上限与停止原因；
- 环境（**Environment**）：系统观察和作用的外部世界。

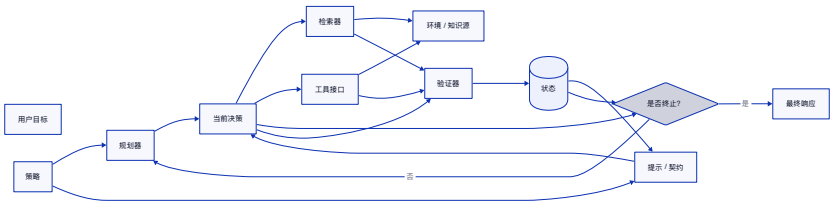

Figure 5: 智能体系统的核心构件。

请把图 Figure 5 当成一张控制架构图来读。策略决定系统整体行为；规划器把目标压缩成当前最值得做的一步；检索器与工具把环境暴露给系统；验证器决定结果是否可接受；状态保留真正影响后续的内容；终止节点则阻止无限探索。

控制器的一个紧凑目标可以写成：

$$a_t^* = \arg\max_{a \in \mathcal{A}(s_t)} \mathbb{E}[U(a \mid o_t, s_t)] - \lambda C(a) - \mu R(a),$$

其中 (U) 是预期效用，$(\mathrm{C(a)})$ 是延迟或算力成本，$(\mathrm{R(a)})$ 是操作风险。工程上未必真的这样显式求解，但这个公式抓住了核心直觉：每一步动作都是质量、成本与风险的权衡。

快模型、强模型与路由

一个很常见的系统模式是：用快模型做路由、抽取或低风险格式化；用强模型做复杂推理、综合与歧义消解。这样能在不显著牺牲质量的前提下，把总体系统维持在可接受的延迟和成本预算内。

规划器可以是可选的，但终止条件不可以

有些任务需要显式规划器，很多任务并不需要。但每个生产级智能体都必须有明确的终止策略。如果系统没有定义“覆盖已经足够”“预算已用尽”或“必须转人工”，那它就不是一个合格的控制器，而是一个无界循环。

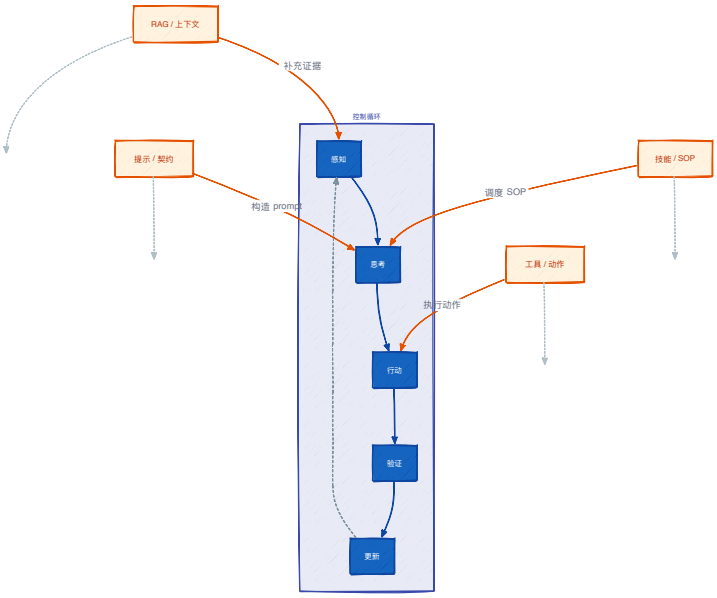

Figure 6: RAG / 工具 / 技能 / 提示在循环中的位置

3.2 RAG：把外部知识接入系统

> **Important**
>
> 本节先回答几个关键问题：
> 1. 标准的 RAG 管线是什么？
> 2. 为什么检索能提升准确性与 grounding？
> 3. 什么时候单次 RAG 就够了，什么时候需要 agentic RAG？
> 4. 应该如何评估检索质量？

RAG (Retrieval-Augmented Generation) 本质上是把外部知识接到 LLM 前面，让模型不是只靠参数记忆回答，而是先检索证据，再基于证据生成。RAG 的工程目标很直接：在生成答案之前，先取回与当前问题最相关的外部证据。它解决的是“系统有没有足够证据来回答”，而不是“模型会不会讲得流畅”。

一条标准 RAG 管线通常包括：

1. 摄入（**Ingestion**）：解析文档、元数据和权限；
2. 分块（**Chunking**）：切成可检索单元；
3. 嵌入 / 稀疏表示（**Embedding / Sparse Representation**）；
4. 索引（**Indexing**）：构建可查询结构；
5. 检索（**Retrieval**）：取回候选；
6. 重排（**Reranking**）：更精确地排序；
7. 生成（**Generation**）：基于证据回答，并附引用。

摄入阶段要先解析文档、抽正文、保留元数据和权限；分块阶段不能只按 token 硬切，最好沿标题、段落、表格、代码块这些自然边界切；表示层通常会同时做 dense embedding 和 sparse retrieval；索引层把向量和元数据放进可查询结构；查询时先做召回，再用 reranker 精排，最后把证据喂给模型生成，并尽量要求引用。

真正难的是每一步怎么设计。比如 chunk 太小，信息会碎；chunk 太大，噪声会变多。检索一般我们会优先考虑 hybrid retrieval，因为 dense 更擅长语义相似，BM25 这类 sparse 检索更擅长错误码、产品名、ID、函数名这种精确术语。召回后如果对精度要求高，会加 **cross-encoder reranker**，因为它通常更准，但更贵，所以只对 top-k 候选做。

> **Note**
>
> 一个具体的检索例子
> 假设用户问：
> “三月份的计费迁移有没有改变企业发票的重试行为？”
> 如果系统只查询 `billing migration retry behavior`，很可能会先取回通用 FAQ、旧的重试 runbook，以及一份不相关的事故复盘。真正关键的迁移说明仍然缺失，因为文档里用的是 **dunning backoff** 这个术语，而不是 **retry behavior**。
> 更成熟的做法会额外做三件事：
> 1. 查询改写：加入 `retry`、`dunning`、`backoff`、`enterprise invoice`、`migration` 等同义词；
> 2. 混合检索：把语义搜索与关键词精确匹配结合起来；

3. 重排：把迁移说明与支持升级排到通用 FAQ 之前。

一个简化候选集可以是：

阶段	Top 结果
查询改写前	计费 FAQ、通用重试 runbook、旧事故复盘
查询改写 + 混合检索后	迁移说明、支持升级、重试 runbook
再经过重排	迁移说明、支持升级、重试 runbook

这样得到的答案就不再是“模型看起来说得通”，而是“系统确实找到了迁移说明，并且用支持升级进行交叉验证，再附带引用给出答案”。

为什么检索能提升准确性

检索的价值在于：它能补足模型参数中没有的知识，让答案 grounded 在可见证据上，并引入新鲜度与可追溯性。但 RAG 不是“把文档塞给模型就会更准”。它通常会在以下地方失败：

- 分块失败：块太小导致语义断裂，块太大导致信号被淹没；
- 召回失败：真正相关的证据没有被取回来；
- **grounding** 失败：证据已经在上下文里，但模型没有真正依据它作答。

稠密检索、稀疏检索与混合检索

- 稠密检索擅长语义相似和释义匹配；
- 稀疏检索如 BM25 擅长精确术语；
- 混合检索通常是生产里的稳妥起点。

如果系统处理的是错误码、法条编号、字段名或 SKU，精确命中非常关键；如果处理的是自然语言意图，语义相似又不可缺。因此成熟系统很少只押宝单一检索模式。

重排、引用与评估

检索给你候选，重排决定真正进入上下文窗口的内容。交叉编码器重排器常常有效，因为它能联合建模 $((\text{query}, \text{document}))$ 对，但代价是延迟与成本。

至少两个检索指标值得明确写出来：

$$\text{Recall@}k = \frac{1}{|Q|} \sum_{q \in Q} \mathbf{1}\{\exists d \in top_k(q) : rel(q, d) = 1\}$$

以及

$$\text{MRR} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{rank_q}.$$

`Recall@k` 问的是：正确证据有没有出现在前 (k) 个结果里？`MRR` 问的是：第一个相关结果出现得有多靠前？但它们仍然不够。端到端评估还要继续看最终答案是否 grounded、是否完整、是否在证据不足时正确弃答。

检索层看 recall@k、MRR 以外，回答层还要看忠实度、完整性、引用覆盖率和冲突率。

线上还要记录命中的文档、检索分数、rerank 顺序和最终引用来源，不然出了问题没法定位。

单次 RAG 与 Agentic RAG

如果问题足够清晰、证据库足够稳定，单次 RAG 已经足够。Agentic RAG 则会重新进入控制循环：改写查询、补齐覆盖面、抓取更多来源，并在证据足够或预算耗尽时停止。研究助手、复杂企业搜索和多文档综合，通常属于后者。

3.3 工具使用：让系统与真实世界交互

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 为什么 AI 系统需要外部工具？
> 2. 一个好的工具接口应该长什么样？
> 3. 什么是结构化输出，为什么它是工具使用的前提？
> 4. 为什么权限、幂等性和错误语义必须在模型之外实现？

如果说检索让系统知道外部事实，那么工具让系统触碰外部世界。没有工具的模型最多只能生成语言；有了工具，系统才可能读取实时状态、执行操作、核验结果，甚至改变世界。

常见工具类型包括：

- API 调用
- 数据库查询
- 代码执行
- 搜索与抓取
- 工单、邮件、退款、部署等业务动作

工具契约才是真正的接口

在生产里，工具不是“给模型开放一个函数”这么简单。它首先是一个接口契约。最少应定义：

- 输入 schema
- 输出 schema
- 参数约束
- 超时与重试语义
- 错误类别
- 副作用边界
- 风险级别
- 是否要求幂等键

一个带类型的工具签名，比一句自然语言说明更能体现设计意图：

```
def refund_payment(
    payment_id: str,
    amount_cents: int,
    reason: Literal["duplicate_charge", "service_failure", "goodwill"],
```

```
    idempotency_key: str,
    dry_run: bool = False,
) -> RefundResult:
    ...
```

这比“助手在需要时可以退款”有用得多。契约告诉系统“能做什么”“必须怎么做”“运行时可以施加哪些约束”。

结构化输出：从文本变成可执行对象

工具调用之所以成立，前提是模型输出不再被当作自由文本，而是被当作带类型的对象。例如：

```
{
  "action": "refund_payment",
  "payment_id": "pay_01483",
  "amount_cents": 4999,
  "reason": "duplicate_charge",
  "idempotency_key": "refund:pay_01483:4999:v1",
  "dry_run": true
}
```

相比“我觉得应该退一下这笔款”，上面的对象可以被验证、回放、限流、审计，也可以在违反策略时被直接拒绝。

对写工具来说，幂等性不是可选项

对于会产生副作用的写工具，一个最小正确性要求是：

$$f(f(s, x, k), x, k) = f(s, x, k)$$

也就是说，在相同状态 (s)、相同参数 (x) 和相同幂等键 (k) 下，重复提交不能重复放大副作用。

> **事故案例：重复退款**
>
> 一个客服助手选对了退款工具。工具在网络层超时后，运行时进行了自动重试。由于工具没有实现幂等性，同一笔退款被执行了两次。根因不在模型，而在接口契约。

权限、审批与回滚必须在模型之外实现

模型可以提出建议，但不能决定自己有没有权限执行。权限、allowlist、租户边界、审批流和人工升级必须在运行时与工具层 enforced。

一个实用的风险分级方法是：

风险级别	示例	典型控制
`read_low`	搜索、查询、读取状态	超时限制、结果数限制、最小权限

风险级别	示例	典型控制
`write_medium`	创建工单、起草邮件	幂等键、dry-run、结果确认
`write_high`	退款、取消订单、生产环境变更	双重确认、审批、人工介入

工具错误也必须结构化

不要把堆栈信息原样灌给模型。更稳的做法是返回结构化错误标签，例如：

- `retryable_timeout`
- `invalid_args`
- `not_found`
- `permission_denied`
- `rate_limited`

这样运行时才能区分“修复后重试”“请求用户补充信息”以及“必须立刻停止”。

3.4 状态与记忆：让任务跨越单轮推理

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 为什么长任务必须做状态管理？
> 2. 状态与上下文有什么区别？
> 3. 短期上下文和长期记忆该如何分工？
> 4. 为什么记忆失控会拖垮系统性能？

一旦任务超过单轮调用，系统就必须表示“已经发生过什么”和“接下来该做什么”。这正是状态与记忆的作用。

状态不是聊天记录的堆积

状态应该是结构化对象，保存当前任务真正依赖的事实、约束、决策和中间结果。上下文则是当前这一步实际送入模型的 token 载荷。两者有关联，但不是一回事。

一个很有用的预算方程是：

$$L_{\text{system}} + L_{\text{working}} + L_{\text{retrieved}} + L_{\text{response}} \leq L_{\max},$$

其中 $L_{\max}$ 是模型上下文窗口。这就是为什么上下文工程很关键：你分给旧日志的每一个 token，都意味着当前证据或最终推理少了一个 token。

分层记忆架构

系统知道的所有内容，并不应该一直待在模型窗口里。固定上下文小而稳定；工作集随任务推进快速变化；检索上下文按需进入；长期记忆常驻在窗口外，只在真正相关时被拉入。

短期上下文 **vs** 长期记忆

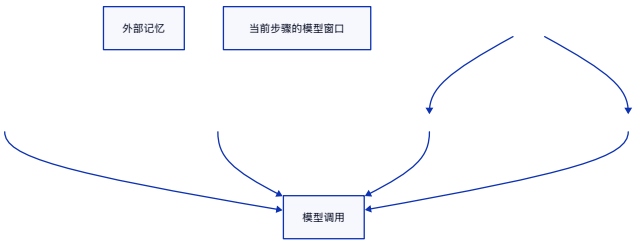

Figure 7: 分层记忆与上下文架构。

- 短期上下文服务当前任务：最近交互、活跃工作集、临时笔记；
- 长期记忆保存跨会话仍然有价值的内容：用户偏好、持久事实、批准过的流程、稳定配置。

没有 provenance、时间戳、权限与 TTL 的长期记忆，很快就会变成幻觉缓存。它看起来让系统“记住更多”，实际上只是把过时或越权信息长期保存下来。

压缩与 Context Rot

随着任务轨迹拉长，模型越来越慢，也越来越难准确定位关键信息。这通常被称为 **context rot**。应对方法是压缩：

1. 保留事实，而不是保留原始载荷；
2. 保留引用，而不是整篇文档常驻；
3. 把长轨迹压缩为结构化摘要；
4. 真正需要原始证据时，再按 ID 回取。

一个有用的压缩度量是：

$$\rho = \frac{L_{\text{raw}}}{L_{\text{summary}}},$$

但高压缩率本身不是目标。真正目标是：在上下文预算内，保持高信息密度。

> **Note**
>
> 具体例子研究任务做了 25 次工具调用之后，系统不应该把 25 篇原始文档都留在活动窗口里。它应该保留任务目标、已尝试查询、已接受断言、未解决冲突、引用和预算消耗，并在确实需要时重新抓取原始材料。

4 工程

4.1 验证与护栏：把概率系统约束成产品

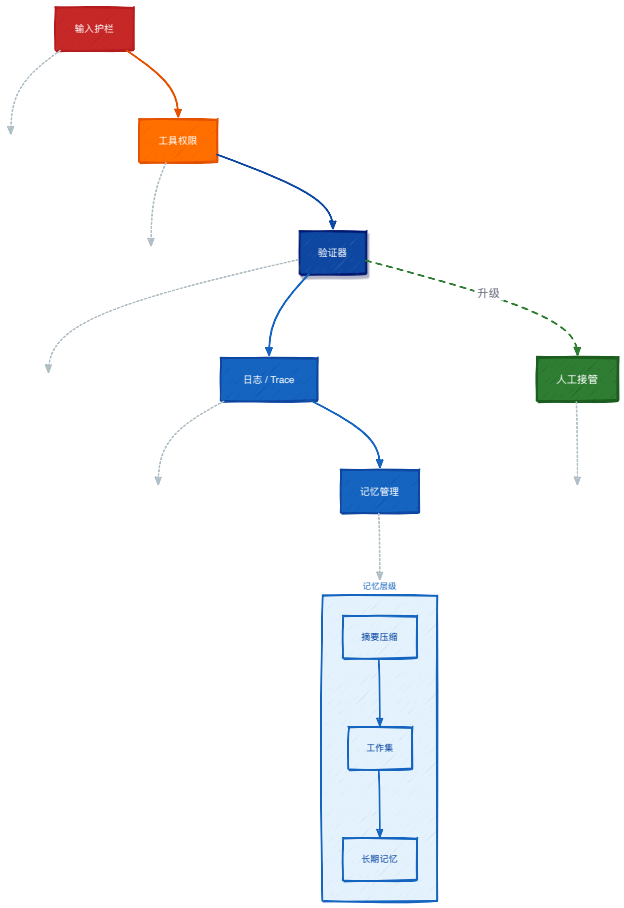

Figure 8: 生产就绪智能体——护栏、可观测性、记忆管理

> **Important**
>
> 本节先回答几个关键问题：
>
> 1. 验证机制如何提升系统可靠性？

2. 护栏在生产系统中到底做什么？
3. Schema 检查、规则检查和 LLM judge 各自适合做什么？
4. 为什么越狱和提示注入必须分开防御？

如果说模型负责“给出候选结果”，那验证负责的就是“判断这个结果能不能进入系统”。这也是 AI 工程最像软件工程的地方：输出必须先变成一个可检查、可拒绝、可修复、可升级的对象。

护栏不只存在于 prompt。它存在于输入边界、运行时、工具权限、验证器、记忆架构、日志系统和人工升级链路中。这就是为什么生产可靠性是一种系统属性，而不是 prompt 属性。

三层验证栈

比较实用的一套验证栈通常包含：

1. **Schema** 验证：结构是否合法？
2. 规则检查：权限、引用、不变量、风险阈值是否满足？
3. **LLM-as-judge**：结果是否完整、连贯、覆盖任务？

一个紧凑的验收函数可以写成：

$$\text{accept}(y) = \mathbf{1}[S(y) = 1 \wedge R(y) = 1 \wedge J(y) \geq \tau],$$

其中 (S) 是 schema 验证，(R) 是确定性规则检查，(J) 是 judge 分数，( ) 是通过阈值。顺序很重要：凡是确定性能检查的东西，都不应该交给 judge。

生成 → 验证 → 修复

生产里更可靠的模式是：

```
for attempt in range(3):
    out = llm(prompt, context)
    ok, err = validate(out)  # schema + 规则
    if ok:
        return out
    prompt = prompt + f"\n请修复以下验证错误：{err}"
raise RuntimeError("validation_failed")
```

这比“再试一次”强得多，因为重试不是盲采样，而是在验证器的约束下进行定向修复。

> **Note**
>
> 一个例子：验证带引用的回答
> 假设模型输出：
>
> ```
> {
>   "answer": " 这次迁移缩短了企业发票的重试窗口。",
>   "citations": []
> }
> ```
>
> 这段话在语言上也许很像样，但如果任务要求必须带引用，那么 schema 或规则层就应

该立即拒绝它。judge 层不应该被用来解决一个本来可确定检查的问题。

越狱 vs 提示注入

- 越狱（**jailbreak**）：用户直接试图覆盖系统策略；
- 提示注入（**prompt injection**）：外部文档在被检索或读取后，试图伪装成更高优先级指令。

防御方式不同。越狱需要强化策略与拒绝逻辑；提示注入则要求你始终把外部内容当作数据而不是指令。检索文本只能作为证据，不能改写系统策略。

什么时候必须人工升级

当系统面对高风险写操作、连续工具失败、证据冲突无法消解，或者低置信度但又牵涉真实业务影响时，人工介入不是“保守”，而是正确的系统设计。

事故案例：企业搜索权限泄漏

某企业搜索系统确实检索到了正确的机密文档，并将其排在很前的位置，但只在渲染时才做访问控制。结果虽然正文被挡住了，结果列表中的 snippet 仍然泄露了敏感内容。真正的修复是架构性的：在检索前、排序中和展示前都执行访问控制。

4.2 可观测性、评估与监控

Important

本节先回答几个关键问题：

1. 工程师如何监控 AI 系统行为？
2. 一个生产智能体最少应该记录什么？
3. 好的评估打分卡和发布门槛长什么样？
4. 模型升级后线上任务退化，应该如何调试？

一个只检查最终答案的系统，几乎不可能被真正调试。智能体系统是沿着执行轨迹失败的，不是只在一段字符串上失败的。因此，可观测性必须覆盖完整 trace。

至少应该记录什么

层	最少记录项
输入层	请求、会话 ID、任务目标、用户或租户范围、版本信息
模型层	模型版本、prompt 版本、token 消耗、延迟、stop reason
工具层	工具名、参数、结果摘要、错误标签、重试次数
检索层	查询、命中文档 ID、得分、重排顺序、引用来源
状态层	状态快照、预算消耗、未解决冲突
安全层	护栏触发、拒绝原因、人工接管原因

用正确顺序读故障

不要一上来就怪模型。先看工具层，再看检索层，再看上下文构造，然后才是生成，最后再看护栏。这种顺序很省时间，因为许多看起来像“推理失败”的问题，根源其实是 schema drift、索引陈旧或上下文装配错误。

发布打分卡与发布门槛

一个严肃的智能体项目需要黄金任务集（gold set）和发布门槛（release gate）。一张紧凑的打分卡至少应追踪：

- 任务成功率
- 工具选择准确率
- grounding / 引用通过率
- 安全违规率
- 单任务成本
- (p95) 延迟
- 升级率

可以把发布门槛写成：

$$\text{release} = \mathbf{1}[SR \geq \tau_s \wedge TA \geq \tau_t \wedge GP \geq \tau_g \wedge SV \leq \tau_v \wedge P95 \leq \tau_l \wedge C \leq \tau_c],$$

其中 (SR) 是成功率，(TA) 是工具准确率，(GP) 是 grounding 通过率，(SV) 是安全违规率，(P95) 是延迟，(C) 是单任务成本。

一个 benchmark 更高、但工具准确率和 grounding 通过率更差的模型，不应该发布。

成本也是质量的一部分

一个有用的单任务成本分解是：

$$C_{\text{task}} = c_{\text{tok}} T + \sum_i c_{\text{tool},i} + c_{\text{rerank}} R + c_{\text{human}} H.$$

一个系统就算准确，如果成本高出一个数量级，也仍然不算生产就绪。

> **Note**
>
> 事故回放：模型升级导致 **schema drift**
> 想象一次模型升级后，通用 benchmark 变得更好，但线上任务成功率下降。为什么？日志显示某个工具的 `invalid_args` 错误急剧上升。升级前，模型常输出 `{"ticket_id": "INC-10428"}`；升级后，它频繁输出 `{"id": "INC-10428"}`。benchmark 变好了，但工具契约坏了。
> 这就是为什么回放评估是必要的。如果你只比较最终的自然语言答案，很可能看不到新模型已经破坏了工具调用习惯。真正的修复可能是契约加固、验证器更新，或者为该模型单独调整工具提示，而不是简单回滚。

5 常见故障模式

> **Important**
>
> 本节先回答几个关键问题：
> 1. 智能体最常见的故障模式是什么？
> 2. 这些故障通常起源于哪一层？
> 3. 团队应该从中学到什么工程教训？

智能体失败通常并不神秘。大多数生产事故最后都会收敛到少数几种重复模式。

Table 5: 智能体常见故障模式、根因与工程教训

故障模式	现象	根因	工程教训
工具误用	选错工具、参数错误，或把只读请求误导向写操作	工具命名含糊、接口重叠、schema 弱，运行时缺少权限与副作用边界控制	工具设计本质上是交互设计——模型看到的是契约，不是你的意图
检索失败	回答错问题、引用无关内容，或忽略知识库里明明存在的证据	分块差、索引陈旧、检索模式不匹配、缺少重排，或生成没有真正 grounded 在来源上	先检查召回和上下文构造，再去改 prompt
状态爆炸	任务一长，系统变慢、变贵，更容易忘记关键事实	原始工具结果和全部历史被不断追加到窗口里，没有压缩和分层	上下文是预算，不是仓库
幻觉式输出	高置信度给出无来源支持的结论，或提出越权动作	验证薄弱、没有引用要求、缺少规则检查、没有审批路径	“看起来像对”不是正确——系统必须通过验证才有资格行动
无限探索与预算失控	不停搜索、抓取、改写查询，但始终不收敛	没有显式预算、没有停止条件、没有定义“覆盖已经足够”	没有终止逻辑的自治，只是失控成本的另一种形式

6 案例研究

6.1 玩具新闻研究智能体

> **Important**
>
> 本节先回答几个关键问题：
> 1. 如何设计一个 AI 研究助手？
> 2. 什么架构适合搜索、抓取、综合与停止？

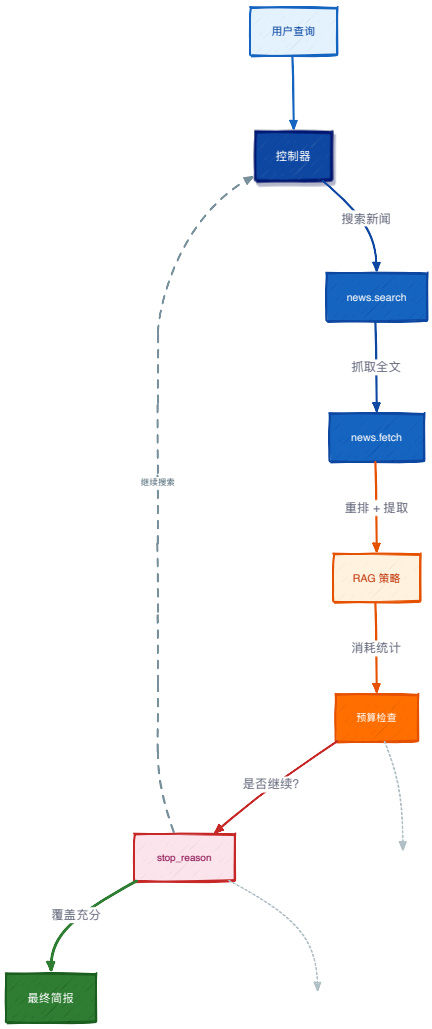

Figure 9: 玩具新闻智能体案例——从查询到简报的完整流程

3. 一个端到端的 agentic RAG 循环长什么样？

研究助手是最适合理解 agent 的案例之一，因为任务天然需要迭代搜索。用户问题往往是欠明确的，证据之间可能冲突，系统必须判断是否继续搜索，还是已经足够停止。

运行时掌握控制循环；news.search 与 news.fetch 是工具；RAG 层负责去重、重排和断言提取；状态对象记录覆盖面和冲突；预算与 stop_reason 防止无限搜索。

一个端到端走查

假设用户提出：

> “请在 30 秒内写一份 6 条 bullet 的简报，总结本周 AI agent 方向的重要发布，并附引用。”

一个合理的系统路径是：

1. 初始化目标与预算：最大来源数、最大工具调用数、最大墙钟时间；
2. 生成简短查询计划；
3. 调用 news.search 做广覆盖搜索；
4. 只抓取最有希望的结果；
5. 抽取候选断言并绑定引用；
6. 判断覆盖面和冲突是否已经足够；
7. 若足够则停止，否则继续搜索；
8. 生成结构化简报，并附带 stop_reason。

一个最小循环可以写成：

```
goal = {"topic": "AI agent announcements", "max_sources": 10, "max_tool_calls": 20}
state = {"queries_tried": [], "claims": [], "sources": [], "stop_reason": None}

while True:
    if len(state["sources"]) >= goal["max_sources"]:
        state["stop_reason"] = "source_budget_reached"
        break
    if tool_calls_used() >= goal["max_tool_calls"]:
        state["stop_reason"] = "tool_budget_reached"
        break

    plan = llm("Choose next action: search, fetch, or stop", context={"goal": goal, "state"

    if plan.action == "search":
        hits = news.search(plan.query)
        state = update_with_hits(state, hits)
    elif plan.action == "fetch":
        doc = news.fetch(plan.url)
        state = update_with_doc(state, doc)
    elif plan.action == "stop":
        state["stop_reason"] = plan.reason
```

```
        break

brief = llm("Write a cited briefing", context={"state": state})
```

真正重要的不是这段代码本身，而是它体现出的架构：系统会决策、行动、更新状态，并在预算内停止。这正是“控制器”与“聊天包装器”的差别。

6.2 客服与运营智能体

> **Important**
>
> 本节先回答几个关键问题：
> 1. 一个客服智能体需要哪些工具和护栏？
> 2. 什么情况下系统应自动处理，什么情况下必须升级？
> 3. 为什么这类系统比研究助手更依赖权限与风险控制？

客服与运营智能体不只是回答问题。它们可能会查询订单、取消订阅、重发发票，甚至准备退款。因此，这类系统的核心难点不只是回答质量，而是受控授权。

一条更安全的路径通常是：

1. 对意图与风险级别做分类；
2. 低风险只读请求走确定性 workflow；
3. 中风险请求可以由系统起草动作，但要求用户确认；
4. 高风险写操作必须经过审批或人工升级。

一个简单的状态机可以写成：

$$\text{requested} \to \text{drafted} \to \text{confirmed} \to \text{executed} \to \text{verified} \to \text{closed}.$$

如果系统会在高风险操作里从 `requested` 直接跳到 `executed`，那它还没有成熟到能进生产。

6.3 编码智能体

> **Important**
>
> 本节先回答几个关键问题：
> 1. 为什么编码智能体离不开“行动后验证”？
> 2. 什么样的循环适合代码搜索、编辑、测试与修复？
> 3. 为什么测试反馈在编码场景里如此关键？

编码智能体之所以特别，是因为软件工程提供了强有力的外部裁判：测试、编译器、linter 和静态分析器。这让编码天然适合进入 生成 $\longrightarrow$ 执行 $\longrightarrow$ 验证 $\longrightarrow$ 修复的循环。

常见工具包括：

- 仓库搜索
- 文件读取与编辑
- 测试执行器
- lint / type-check / compile 工具
- diff 审查与回滚工具

成熟的编码智能体不会在修改完文件后立刻宣布成功。它会继续观察测试结果、编译输出和策略约束，再判断是否值得继续修复。编码场景因此最能说明：可靠的 AI 并不是来自“第一稿写得多好”，而是来自“系统能否借助外部验证器持续收敛”。

6.4 企业搜索

> **!** Important
>
> 本节先回答几个关键问题：
> 1. 为什么企业搜索往往是公司最值得做的 AI 系统之一？
> 2. 它与通用 RAG 有什么不同？
> 3. 为什么权限、元数据与新鲜度在这里比文风更重要？

企业搜索往往比一个“会聊天的通用智能体”更有价值，因为许多真实业务问题本质上都是“在正确权限边界内找到正确的信息”。系统要为正确用户，在正确时间，用正确来源，给出正确答案。

与开放域 RAG 相比，企业搜索更强调：

- 私有数据连接器
- 权限过滤
- 新鲜度与版本管理
- 引用与可追溯性
- 基于元数据的排序

一个优秀的企业搜索系统往往比一个“看起来更聪明”的通用 agent 更能直接解决业务痛点。但它也更快地惩罚架构粗糙：权限检查、元数据过滤和索引新鲜度都必须被提升为一等设计问题。

7 本章小结

一旦 AI 开始接触真实世界，设计单位就不再是 prompt，而是控制循环：系统能观察什么、允许做什么、如何保存状态、怎样验证结果、以及在不确定性仍然存在时如何安全失败。这也是为什么 AI 工程正在越来越像系统工程。模型当然重要，但产品真正活在它周围的契约、预算、trace、验证器和恢复路径里。可靠的智能体不是靠“希望模型像工程师一样表现”构建出来的，而是靠“把好的工程行为设计成系统默认值”构建出来的。

7.1 问题小结

1. 什么是 **AI** 智能体？智能体是一个控制器，它协调模型、检索、工具、状态和验证，通过多轮反馈来完成任务。

2. **AI** 系统与单独的大语言模型有什么区别？单独模型只负责推理和生成；AI 系统把模型放进一个包含检索、工具、状态、护栏与监控的闭环里。

3. 一个现代 **AI** 系统一般由哪些组件构成？模型、检索、工具、状态或记忆、验证或护栏，以及协调这些组件的控制循环。

4. 工作流和智能体有什么区别？工作流的步骤由工程师预定义；智能体在运行时决定下一步。前者更稳，后者更灵活。

5. 为什么 **AI** 系统需要外部工具？因为模型参数不是实时世界状态。要获得最新信息、执行真实动作并验证结果，就必须接入外部工具。

6. 智能体控制循环是如何工作的？系统读取环境和状态，决定下一步，执行动作，验证结果，更新状态，然后在成功或预算耗尽前持续循环。

7. 检索在 **AI** 系统里扮演什么角色？检索把外部证据带进当前推理步骤，提升 grounding、新鲜度和可追溯性。

8. 为什么长任务必须做状态管理？多步任务需要保存中间结论、已尝试路径、未解决冲突和预算消耗。没有状态管理，系统只能不断把原始历史重新塞进上下文。

9. 验证机制如何提升系统可靠性？它把模型输出变成可检查对象。Schema 验证、规则检查和 judge 模型让系统能够运行“生成—验证—修复”的闭环。

10. 智能体最常见的故障模式是什么？工具误用、检索失败、状态爆炸、无 grounding 的幻觉输出，以及因为缺少预算或停止条件而产生的失控探索。

11. 你会如何设计一个 **AI** 研究助手？从 agentic RAG 开始：查询改写、检索、抓取、重排、断言抽取、引用绑定、预算控制，以及显式的停止原因。

12. 什么样的架构适合同时支持工具调用与检索？一个专门的 agent runtime，作为控制器连接模型层、检索层、工具层、状态层、验证层以及日志 / 预算基础设施。

13. 如何让智能体在生产环境里保持安全？使用工具契约、结构化输出、模型外权限控制、风险分级、验证器、人工升级、可观测性和发布门槛。

14. 工程师如何监控 **AI** 系统的行为？记录输入、模型版本、工具调用、检索命中、状态迁移、预算使用、停止原因、安全事件、成本和延迟。

15. 智能体自治程度与系统可靠性之间有哪些权衡？自治越高，灵活性越强，但方差、调试难度和安全风险也越高。默认路径应当是从低自治 workflow 开始。

16. 如何判断一个任务应该做成 **workflow**、**agent**，还是 **agentic RAG**？如果步骤可枚举，就用 workflow；如果核心问题只是 grounding，就用 `workflow + RAG`；如果系统必须迭代搜索、重规划或与实时状态交互，才考虑 agentic RAG 或 agent。

17. 如果智能体可以执行写操作，你会如何设计幂等性、回滚与审批？要求强类型工具契约、幂等键、明确风险分级、dry-run、确认阶段、必要时的补偿操作，以及高风险写入必须经过人工审批。

18. 模型升级后 **benchmark** 更好，但线上任务成功率下降，你会如何调试？先回放黄金任务集，对比工具调用 trace，检查 schema 验证失败、检索与上下文构造，再判断是否真的是模型本身导致的退化。

7.2 给 AI 工程师的实践清单

先定义任务，再讨论架构

先澄清：

- 输入是什么？
- 输出是什么？
- 什么叫成功？
- 延迟和成本预算是什么？
- 出错的代价有多高？

如果这些问题还说不清楚，那还不到讨论 agent 架构的时候。

先做 **workflow**，再把真正的长尾交给模型

先把确定性路径硬编码，再只把真正需要柔性推理的部分交给模型。

把工具当作契约来设计

每个工具都定义：

- 输入 / 输出 schema
- 超时与重试语义
- 错误标签
- 副作用边界
- 风险级别
- 幂等策略
- 审计字段

让状态最小且显式

只保存会真正影响任务推进的事实、决策、引用、冲突和预算值。

把验证前移

尽早使用结构化输出、schema 检查、引用规则、风险规则与升级逻辑。

默认做可观测性

如果上线后你无法回放一次失败任务，那它还不配叫生产系统。

给自治设置预算

至少要显式限制：

- 最大迭代次数
- 最大工具调用数
- 最大 token
- 最大墙钟时间
- 高风险写权限

用评估集驱动系统演进

代表性任务集至少应覆盖：

- 高频路径
- 长尾异常
- 多工具流
- prompt injection 尝试
- 高风险请求
- schema drift 与发布回归

一个紧凑的总结表可以是：

构建阶段	主要目标	关键产物
Workflow	确定性	固定 DAG、测试
Workflow + RAG	Grounding	检索指标、引用
Agentic RAG	迭代搜索	预算 + stop logic
Full Agent	受控动作	工具契约、验证器、人工接管

8 参考资料

1. OpenAI. *A Practical Guide to Building Agents*. 2025.
2. Anthropic. *Building Effective Agents*. 2024.
3. Anthropic. *Effective Context Engineering for AI Agents*. 2025.
4. Anthropic. *Equipping Agents for the Real World with Agent Skills*. 2025.