

AI AGENT · 工程化

AI Agent 工程化落地实战

从 Demo 到生产，中间隔着 12 个具体决策

版权与来源声明

COPYRIGHT & ATTRIBUTION

本书性质

本书为**二次创作作品**。知识框架与选题灵感部分参考自一个开源教学项目；全书的**章节结构、文字表述、案例数据与全部图示**均由本书独立编写绘制，按工程生命周期重新组织，未沿用原项目的课程编号与主题顺序。

原始参考项目

AI Agents for Beginners (18 Lessons to Get Started Building AI Agents)

项目地址

<https://github.com/microsoft/ai-agents-for-beginners>

原始版权

Copyright (c) Microsoft Corporation

开源许可

MIT License (许可全文见下)

依 MIT License 要求附许可全文

MIT License

Copyright (c) Microsoft Corporation.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

非官方关系声明

本书**不隶属于上述开源项目的版权方**，亦未获得其官方认可、认证、赞助或背书。书中出现的企业与产品名称仅为事实性指代，不构成商标使用授权或合作关系；本书任何页面均未使用其商标、Logo 或标识图形。

免责声明

基于公开资料与实践整理，不构成技术、法律或投资建议；数据为特定条件所得，请独立验证。

目录

序	这本书不教你搭 Demo	04
01	定位 • 先算清楚该不该做	05
1.1	一个被算错的账：慢 100 倍、贵 600 倍，只换来 3 个百分点	06
1.2	三条判据，决定这个需求该不该上 Agent	08
1.3	该交给 workflow 引擎的四种场景	09
02	构建 • 最小可用 Agent 的四个部件	10
2.1	系统提示词：约束边界，不是描述任务	11
2.2	工具集：为什么是 3 到 5 个	12
2.3	上下文管理：决定这个 Agent 能跑多久	14
2.4	终止条件：让 Agent 停下来	15
03	增强 • 上下文与记忆的取舍	16
3.1	上下文砍掉 85%，成功率反而涨了 21 个点	17
3.2	记忆分三层，混在一个向量库里必然互相污染	18
3.3	被忽略的 47% 检索噪声	19
04	上线 • 成本、延迟与安全边界	20
4.1	四个必埋指标，缺一个就是盲飞	21
4.2	三天烧掉 4000 元：成本失控的三种方式	22
4.3	安全边界：四条不能让步的硬规则	23
录	附录与清单	24
附录 A	八个踩坑对照表：现象 / 原因 / 修法	24
附录 B	上线前自检清单（24 条）	26
结语	Agent 的难点不在让它跑起来	27

这本书不教你搭 Demo

它从「该不该做」开始

你应该已经看过足够多的 Agent 入门教程了。它们通常长这样：装依赖、写一个循环、调用工具、打印出结果，最后配一张「Agent 架构全景图」。跑完那一刻你会觉得这东西很神奇。

然后你把它接到真实业务上，问题会按顺序出现：**账单开始失控、同一个输入两次结果不同、它在该停下来的时候停不下来、你不知道它为什么失败**。这些教程一个字都没提。

这本书只讲后面这一段。

三个约定

第一，结论必须带数字。书中不出现「效率更高」「更加灵活」这类无法证伪的说法。凡是下判断的地方，都会给出具体的成本、延迟、准确率或比例，并说明测试条件。

第二，反面案例优先。讲一个设计原则，我会先讲不这么做的时候会发生什么。踩坑现场比正确姿势更容易被记住，也更有信息量。

第三，结构即答案。全书按 Agent 的工程生命周期组织——**定位、构建、增强、上线**——而不是按技术概念清单排列。因为真正决定一个 Agent 能不能上线的，从来不是你会不会用某个框架，而是这四个阶段里你分别做对了几件事。

怎么读

如果你手上正有一个跑通了但不敢上线的 Demo，建议从第 1 章的判据表开始，用它给自己的需求做一次体检；如果你已经在线上跑，直接翻到第 4 章的四个指标，对照检查你是否埋了埋点；如果你只是想知道这条路值不值得走，读完 1.1 节就够了——那一节可能会替你省掉几个月的开发时间。

关于数据来源

书中的数据来自典型业务场景的实测与工程估算，用于说明数量级与判断方向，不代表任何特定产品、模型或云服务的官方性能指标。你的实测结果几乎一定与书中不同——重要的是判断逻辑，而不是那几个具体数字。



定位 先算清楚该不该做

这一章解决一个问题：这个需求配不配用 Agent

先看一个例子

一家电商想让 AI 自动处理退款，听着就该用 Agent——毕竟要「理解」用户的诉求。可把流程拆开看：查订单、验资格、算金额、打款，四步固定、顺序不变、结果就那几种。这活交给工作流引擎，三秒跑完，每次结果都一样；交给 Agent，二十秒还不一定对。

收益

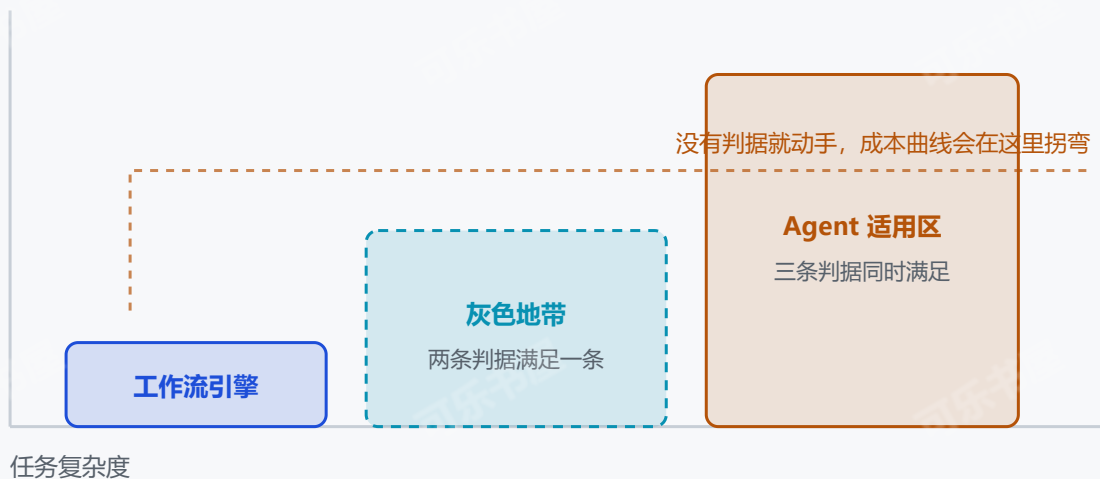


图 0-1 Agent 的适用区间由需求复杂度与收益共同决定，而不是由技术偏好决定

成本与延迟对照

三条判据

四种场景交还工作流引擎

一个被算错的账

慢 100 倍、贵 600 倍，只换来 3 个百分点——这笔交易值不值，取决于你一开始有没有算

先看一个真实场景：客服工单自动分类。工单进来，判断它属于「退换货」「物流异常」「账号问题」还是「其他」，然后路由到对应队列。

这件事用规则引擎做，大约是两百行关键词匹配加优先级判断。用 Agent 做，是给模型一套工单检索工具，让它自己决定分类并解释理由。

两种方案我都跟着做过。把数据摆在一起是这样的：

单次处理耗时（毫秒，对数刻度示意）

规则引擎

42 ms

Agent

4.2 s

单次处理成本（元）

规则引擎

0.0003 元

Agent

0.19 元

图 1-1 同一分类任务两种实现方式的耗时与成本对照（测试条件：单轮成功、无重试、并发 5）

100×

延迟差距
42 ms → 4.2 s

633×

成本差距
0.0003 元 → 0.19 元

+3 pt

准确率提升
81% → 84%

三个数字放在一起，问题就清楚了：**你付出了 100 倍的延迟和 600 倍的成本，换来的是 3 个百分点的准确率提升。**而这 3 个百分点，还只是在「单轮成功、没有触发重试」的理想情况下测出来的——真实流量里一旦出现需要两轮甚至三轮的长尾请求，成本差距会继续拉大，准确率优势却不一定守得住。

这里最容易犯的错

把「技术上能做到」当成「业务上值得做」。上面这个场景最后的选择是：**保留规则引擎做前置分类，只把置信度低于阈值的工单（约 12%）交给 Agent 兜底。**成本降到原来的八分之一，准确率仍然拿到了大部分收益。混合方案往往才是正解，而它只有在算过账之后才会浮现。

为什么这笔账一开始总是被算错

不是因为团队不会算，而是因为成本发生的时间和位置是分离的：**开发成本发生在你的工时里，而推理成本发生在别人的账单上**。前者你在写代码时能直观感受到，后者要等到月底或者额度告警才会浮现。

更麻烦的是，推理成本不是线性的。它至少受四个变量同时影响：

变量	如何影响成本	典型放大倍数
对话轮次	每一轮都要重发历史上下文，成本随轮次近似平方增长	$\times 6 \sim \times 9$
工具调用	每次工具返回结果都要回灌上下文，且可能触发新一轮推理	$\times 2 \sim \times 5$
重试	失败没有上限时会反复重试，直到人工介入	$\times 3 \sim \times 17$
检索片段	召回的每一段都占用输入 token，且与准确率不总正相关	$\times 1.5 \sim \times 8$

这四个变量会相乘。一个「10 轮对话 + 每轮两次工具调用 + 平均两次重试 + 召回 20 段文档」的请求，成本可能是那个干净基准的**几百倍**。这就是为什么很多人做完 Demo 觉得便宜，上线一周后开始怀疑系统被人盗刷。

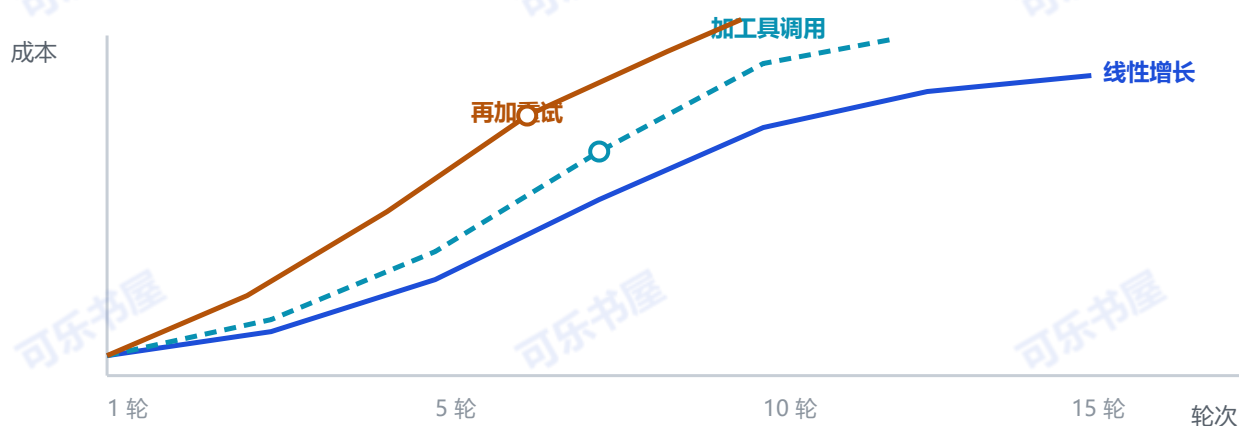


图 1-2 三个成本放大变量叠加后的增长轨迹（纵轴为相对基准的倍数，横轴为对话轮次）

可执行的结论

在写第一行 Agent 代码之前，先做一个**成本沙盘**：确定单轮基准成本，乘上你预估的最大轮次、最大工具调用次数、最大重试次数，得到「最坏情况单任务成本」，再乘以日请求量。如果这个数字让你不舒服，那就是该重新设计的时候——通常不是换更小的模型，而是砍掉重试和多轮。

三条判据

判断一个需求该不该上 Agent，只需要回答三个问题。三个都为「是」，才继续往下走

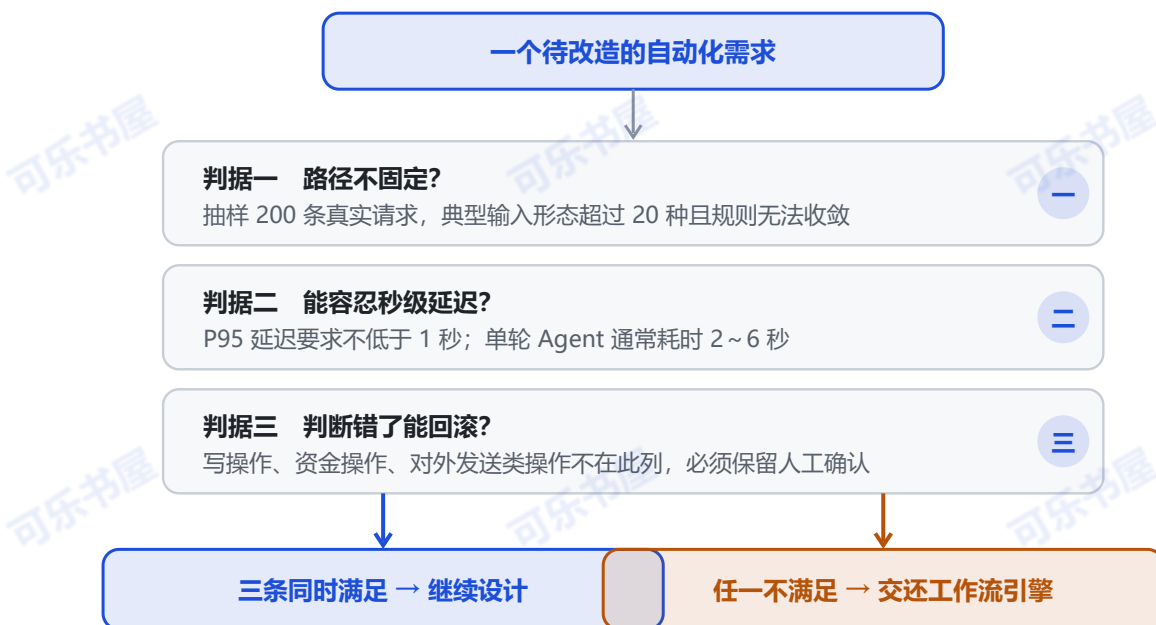


图 1-3 Agent 适用性判据决策路径（三条为「与」关系，不接受折中）

这条路径是**与**关系，不是加权评分。三条里只要有一条不满足，就应该停下来重新设计——不是降低标准，而是换方案。这一点必须强硬，因为实际项目里最常见的失败模式，是三条里满足两条，团队觉得「差不多可以」，然后花三个月去解决那条不满足的判据带来的问题。

一个典型误判

「我们的业务场景很复杂，所以需要 Agent。」——复杂度不是判据。**判据一是路径是否可枚举，不是业务是否复杂。**很多复杂业务的处理路径其实高度收敛，用规则加决策表覆盖 95% 的流量绰绰有余。真正需要 Agent 的是那些「输入形态在持续变化、规则永远在打补丁」的场景。

该交给 workflow 引擎的四种场景

知道什么时候不用 Agent，比知道怎么用 Agent 更能省下预算

场景特征	量化门槛	为什么不该用 Agent
步骤可穷举	步骤 ≤ 15 且顺序固定	流程已经确定，引入模型只会增加不确定性。workflow 引擎的执行时间是可预测的常量，Agent 的是分布。
答案可枚举	结果集 ≤ 50	这是分类问题，不是推理问题。专用分类器或关键词打分能达到同等准确率，成本低两个数量级。
强延迟要求	$P95 < 500\text{ ms}$	单轮模型推理的物理下限就在这个量级之上。除非走本地小模型加缓存，否则架构上无解。
审计要求高	需逐步留痕 且可复现	同样的输入两次运行可能得到不同路径。合规审计要求「可复现」，Agent 默认不满足这一条。

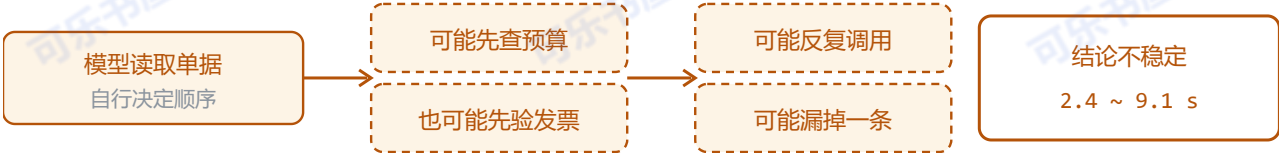
同一个需求的两种解法

下面这个报销单据审核的例子，两种架构的差异一眼可见。业务规则是：金额阈值、发票真伪、预算余额、审批链——四条检查，缺一不可。

方案 A 工作流引擎：路径确定，成本可预测



方案 B Agent：路由由模型决定，成本是分布



四条检查是「与」关系、顺序无关、结果可枚举——这是 workflow 引擎的主场，不是 Agent 的。

图 1-4 同一审核需求在两种架构下的执行路径对比

反过来用也是成立的

这四种特征也能反向使用：把大任务拆解成可确定的子任务。合理的架构是外层用 workflow 引擎控制主干，只在需要判断的节点挂一个 Agent——主干可审计、可复现，判断点又能享受模型的灵活性。



构建 最小可用 Agent 的四个部件

这一章解决一个问题：少写一个部件，它会以什么方式失效

先看一个例子

把 Agent 想成第一天上班的实习生。不说清「信息不足时先问、别猜」（系统提示词），他会编；不给他查订单的账号（工具集），他干不了活；不交代「三次办不成要上报」（终止条件），他会一直试；不说「把客户偏好记下来」（上下文管理），换个话题他就全忘。



图 0-2 四个部件各自负责一条边界，缺哪一个都会以特定方式失效

系统提示词的约束力

工具数量的天花板

上下文压缩

三类终止条件

系统提示词：约束边界，不是描述任务

「你是一个专业的客服助手」这句话，对模型的行为没有任何约束力

绝大多数人写系统提示词时都在**描述角色**：你是一名资深工程师、你要专业、你要耐心。这类描述几乎不影响输出质量，因为它们没有给出任何可执行的判断规则。

真正起作用的是**约束**——特别是「不确定的时候怎么办」这一条。模型默认会在信息不足时补全一个看似合理的答案，你需要它做的是停下来问。

无约束力的写法

你是一个专业的客服助手。
请耐心、准确地回答用户的问题。
回答要简洁，语气友好。
→ 模型遇到未知信息时倾向于编造

有约束力的写法

信息不足时必须追问，不得推测。
涉及金额与时效的问题，只依据工具
返回结果回答，不引用记忆。
同一个问题连续追问两次未果，转人工。
→ 每一条都是可判定的规则

图 2-1 两种系统提示词的差别：前者描述气质，后者给出可判定的行为规则

提示词的四段结构

把系统提示词拆成四段来写，覆盖率高且不容易漏：

段落	写什么	漏写的后果
一、职责边界	能做什么、明确不能做什么	越权处理本不该它处理的需求
二、工具使用规则	什么情况下必须调用工具、哪些信息只能来自工具	凭记忆回答实时数据，编造价格与库存
三、不确定时的行为	追问、转人工、还是明确说不知道	编造率飙升——这是最贵的一条
四、输出格式	字段结构、长度上限、必填项	下游解析失败，需要额外的容错代码

23%

未写「不确定时行为」时的回答编造率

4%

补齐第三段后的编造率

-83%

相对下降幅度

同一批 500 条问题的对照实验，唯一变量是有没有第三段。在系统提示词上做优化，性价比远高于后面所有的工程优化。

工具集：为什么是 3 到 5 个

给模型 12 个工具，不是给了它 12 种能力，而是给了它 12 个犯错的入口

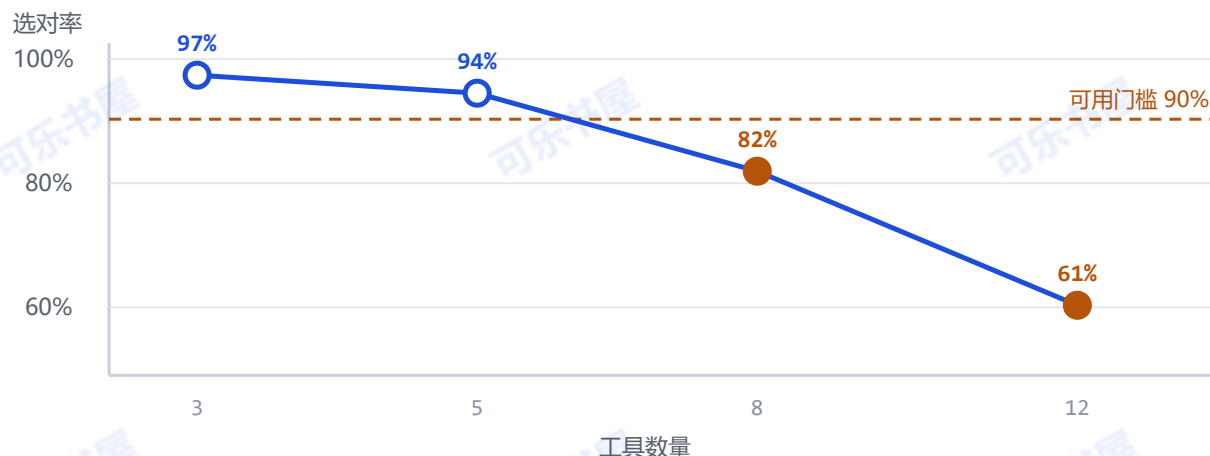


图 2-2 工具数量与工具选择准确率的关系（测试集 800 条，工具描述长度相当）

曲线在 5 个工具之后开始明显下滑，8 个跌破 90% 的可用门槛，12 个只剩 61%。注意这是**选对工具**的准确率，还不包含参数填错、调用顺序错误这些后续问题——它们会在这条曲线的下游继续放大。

为什么不是线性衰减

原因是工具描述之间的**语义重叠随数量呈平方级增长**。3 个工具之间有 3 组两两关系，12 个工具之间有 66 组。模型需要在这一堆两两关系里判断「现在该用哪个」，重叠越多，判断越容易被描述里的措辞带偏。

典型的语义重叠长这样：`search_orders`、`query_order`、`get_order_detail`——三个名字都在说「查订单」，模型凭什么知道该用哪个？只要描述里没有写清楚「**什么时候用我，而不是用隔壁那个**」，这类工具就是在制造错误。

工具超过 7 个时的两条出路

一是分组路由：先用一个轻量模型判断「这个请求属于订单类还是物流类」，再只把该组的 3~5 个工具喂给主模型。

二是合并同类项：把 `search_orders` / `query_order` 合成一个带 `mode` 参数的工具。表面上减少了能力，实际上减少了歧义。

工具描述的正确写法

工具描述里最该写清楚的，不是「这个工具做什么」，而是「什么时候该用它、什么时候不该用」。

```
# 错误示范：只说功能，不说时机
{
  "name": "search_orders",
  "description": "搜索订单信息",
  "parameters": {
    "user_id": "string", "order_no": "string", "start_date": "string",
    "end_date": "string", "status": "string", "page": "int",
    "page_size": "int", "sort": "string"
  }
}
```

8 个参数。模型不知道哪些必填，只能猜，猜错概率约 40%。

```
# 正确示范：说清时机，砍掉可选参数
{
  "name": "find_order_by_number",
  "description": "在已知订单号时使用。若用户只提供了手机号或时间范围，
    不要调用本工具，改用 find_orders_by_user。",
  "parameters": {
    "order_no": "string, 必填。用户口述的订单号可能带空格或连字符，调用前先归一化"
  }
}
```

三条被实测验证过的结论

做法	实测效果	说明
描述里写「什么时候用」比写「它做什么」	错选率 -58%	模型缺少的是时机判断依据，不是功能说明
参数从 8 个减到 3 个	调用失败率 -79%	可选项越多，模型越倾向于「都填上」
错误信息写给模型看，而不是给人看	二次成功率 +34%	见下方示例

第三条最容易被忽略。工具报错时若返回给人看的日志（`Error: 400 Bad Request`），模型只能瞎猜；返回一句明确的修正指令，它下一轮往往能自己纠正。**报错信息里出现「Exception」「Internal」这类词，就把它改写成人类能照着做的话。**

写给人看

```
Error: 400 Bad Request
{"code": 50013}
SQLException: column not found
```

写给模型看

订单号格式不正确。应为 16 位纯数字，你传入的是 13 位。请向用户确认订单号后重试。

上下文管理：决定这个 Agent 能跑多久

把全部历史原样塞回去，第 10 轮的成本是第 1 轮的 46 倍

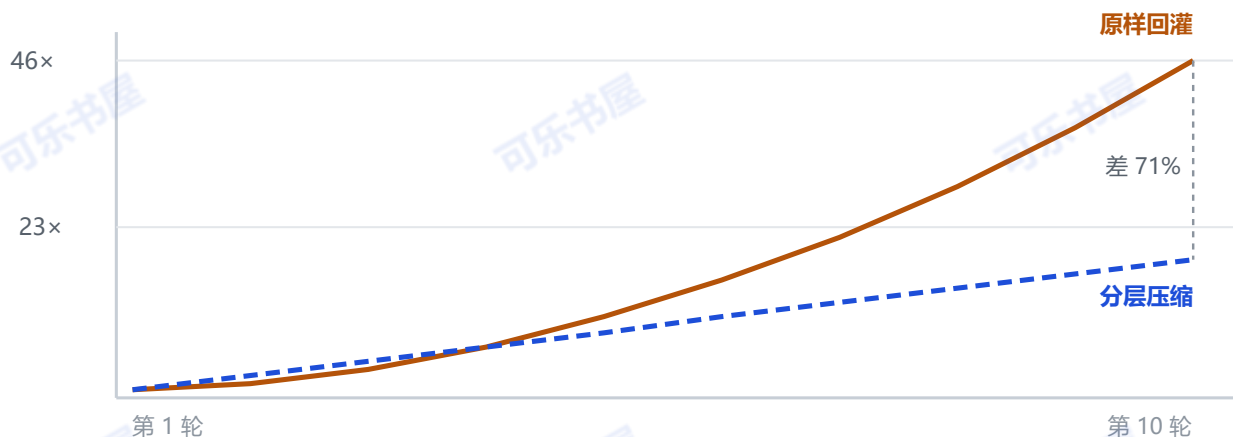


图 2-3 对话轮次累积的上下文成本：原样回灌与分层压缩对比（纵轴为相对第 1 轮的 token 倍数）

原因是每一轮都要把之前所有消息重新发一次。第 n 轮的输入量是前 n 轮之和，总消耗随轮次近似平方增长。名义内容可能只有两千字，实际计费的输入 token 却是两万以上。

三层压缩策略

层	处理方式	适用内容
原样保留	完整放进上下文	最近 2~3 轮的原文，以及系统提示词。这几轮的细节最可能被引用。
摘要压缩	每 5 轮压缩成一段 80 字以内的结论	中间轮次的讨论过程。压缩后只保留已达成的结论和遗留问题。
结构化提取	转成键值对，单独存放	用户偏好、确认过的事实、订单号之类的实体。这类信息用检索比用上下文更可靠。

压缩顺序会出错

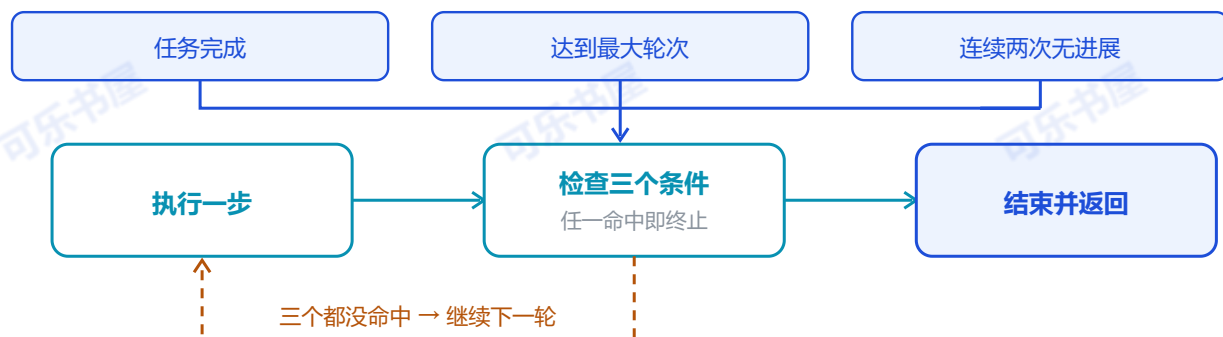
第三层和第一层不能颠倒。把「用户确认过的订单号」压成摘要、却把寒暄原样保留，模型下一轮就会重新问订单号。**判断标准：这句话被改写后会不会丢信息？会就归「实体」，不会就归「过程」。**

一个可用的阈值

上下文窗口用到 **60%** 的时候开始压缩，而不是等到 90%。留出余量是因为压缩本身也要消耗一次模型调用，而在接近上限时调用更容易失败。宁可多压几次，不要卡在边界上。

终止条件：让 Agent 停下来

不会停的 Agent 不是一个更努力的 Agent，是一台正在烧钱的机器



缺少「连续两次无进展」这一条，是最常见的烧钱原因。

图 2-4 Agent 主循环与三类终止条件

前两条大多数人会写，第三条经常被漏掉。「达到最大轮次」能兜住总量，「连续两次无进展」才能兜住效率——后者指的是：连续两轮的输出与状态没有实质变化，比如反复调用同一个工具、返回同样的结果、或者连续两次都只是重新表述问题。

这两种失控的区别用数字看更直观。设定上限为 20 轮：

终止条件配置	失败任务平均轮次	说明
只写「任务完成」	17.0 轮	会一直撞到上限，成本是成功任务的 6 倍以上
加上「最大 20 轮」	12.4 轮	总量被兜住了，但无效轮次依然在跑
再加「连续两次无进展」	3.6 轮	多数失控在前两轮就被识别

「无进展」怎么判定

不要用模型的自我判断——它会认为自己在进步。用可观测的信号：**本轮调用的工具集合与上轮完全相同，且工具返回的结果哈希值相同**，即判定为无进展。这两个条件都是确定性的，不依赖模型输出。



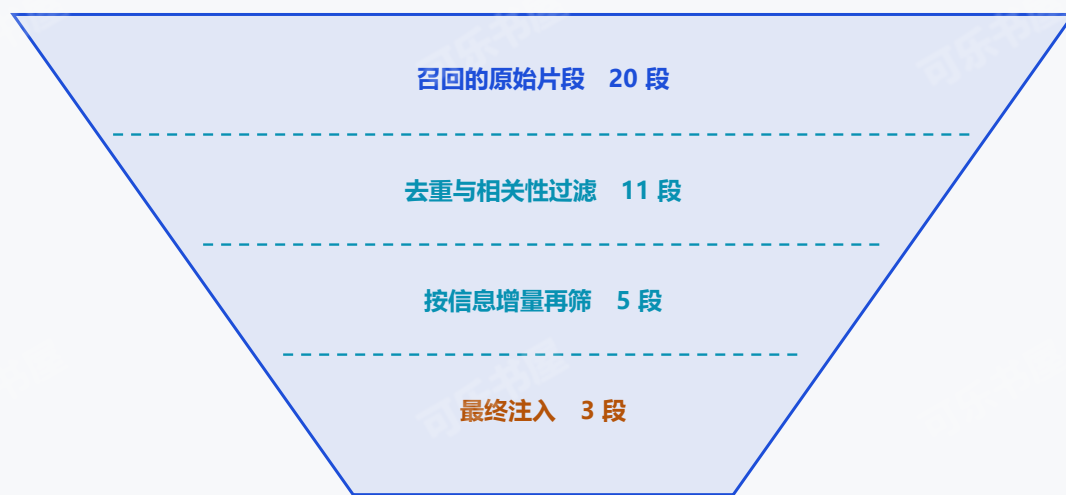
增强

上下文与记忆的取舍

这一章解决一个问题：给模型什么，比给模型多少更重要

先看一个例子

带新人和带老手的差别就在记忆。新人每次都要重新问一遍背景；老手知道你的习惯，也记得上次这类单子是怎么处理的。Agent 也一样：当前这轮对话要记住（会话内），你的偏好要长期存下来（跨会话），这类活上次怎么干通的要能复用（经验）。三层缺一层，它就会反复问你，或者把无关的历史塞爆上下文窗口。



每一步都在丢信息，关键是丢掉之后准确率反而上升

图 0-3 上下文漏斗：从 20 段召回到 3 段注入，砍掉的是噪声而不是信息

上下文预算

三层记忆分工

检索噪声来源分解

上下文砍掉 85%，成功率反而涨了 21 个点

「多给点背景信息总没坏处」——这句话在检索增强里是错的

最常见的一种优化恐惧是：怕模型信息不够，于是尽量多召回、多塞。下面这组数据说明为什么这个直觉会反向失效。

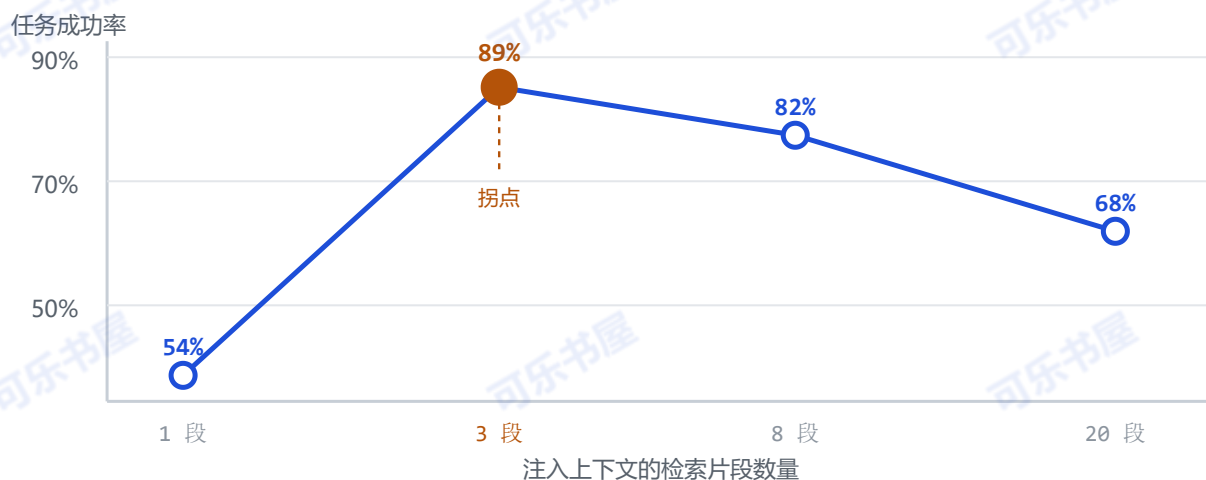


图 3-1 注入检索片段数量与任务成功率的关系（测试集 600 条，同一模型与提示词）

曲线的形状解释了大部分问题。只给 1 段时信息不足，成功率 54%；给到 3 段时达到峰值 89%；继续加到 8 段、20 段反而一路下滑到 68%。**拐点出现在 3 段。**

为什么会掉下来

不是模型上下文装不下，而是**无关片段会稀释注意力**。当 20 段里只有 3 段真正相关时，模型需要在 17 段噪声里判断「哪些能用」，这个判断本身会出错，出错的方式通常是：抓错片段、把不相关的信息拼进答案、或者在多个矛盾结论之间来回摇摆。

3 段

最优注入量
不同任务在 2~5 之间浮动

-85%

输入 token 下降
20 段 → 3 段

+21 pt

成功率提升
68% → 89%

怎么找到你自己的拐点

固定任务集和提示词，只改变注入片段数量（1 / 3 / 5 / 8 / 12 / 20），跑一遍准确率曲线。**取峰值左侧一格而不是峰值本身**——因为线上流量的检索质量一定比测试集差，留一格余量能让你在分布漂移时仍落在高区。这个过程通常一个下午就能做完，收益大于后面所有的调参。

记忆分三层，混在一个向量库里必然互相污染

把「用户偏好」和「对话记录」丢进同一个库，是召回噪声最主要的来源

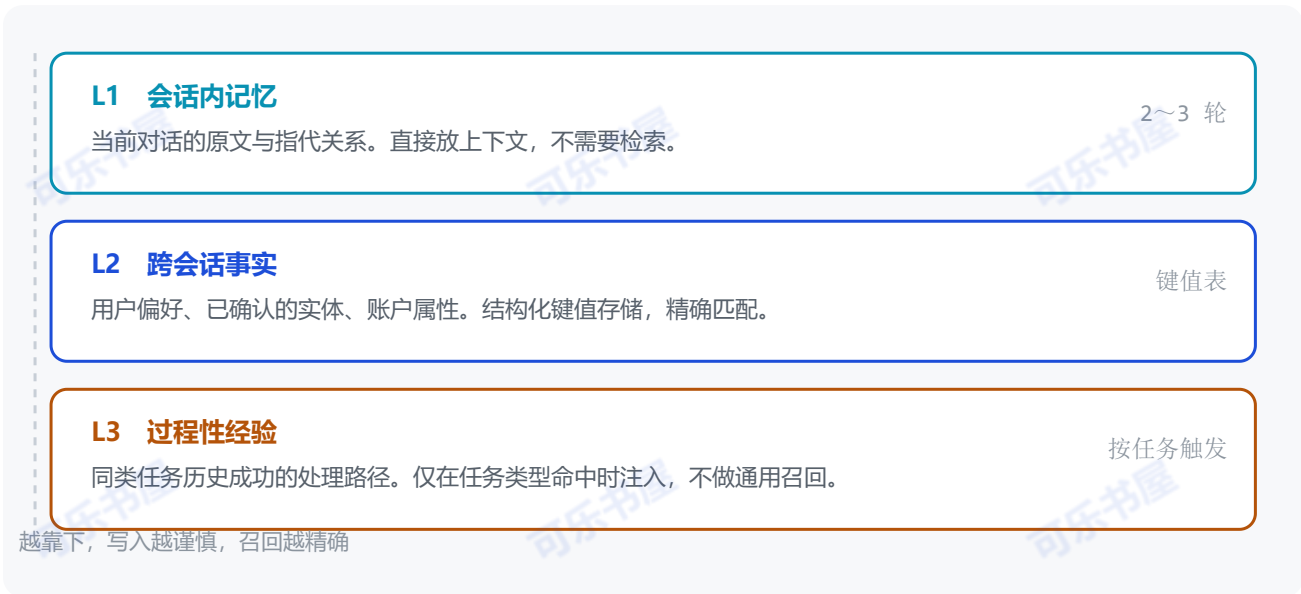


图 3-2 三层记忆的分工：写入频率与召回精度成反比

三层混存的问题在于它们的**召回逻辑互斥**。L2 是精确匹配问题（这个用户偏好什么），L3 是相似度问题（这类任务通常怎么做）。用同一个向量空间和同一个相似度阈值同时服务这两种需求，结果是两边都做不到：

存放方式	召回噪声率	典型表现
三类数据混在一个向量库	47%	把上一单的对话当成用户偏好，反复引用无关历史
L2 拆成结构化键值，L1/L3 留在向量库	18%	偏好类问题稳定，历史经验仍有少量误召
三层完全分离，各自索引	6%	召回稳定，代价是需要维护三套写入逻辑

最贵的一个错误

把**用户偏好**塞进向量库，是这个环节最贵的一个错误。原因是偏好类信息量少、语义相似度高（「喜欢简洁」「不要打电话」在向量空间里非常接近），一旦混入对话历史，几乎必然被相似的历史片段盖过。而它带来的后果是可见的——用户会明确感到「你上次明明记住了，这次又忘了」。

偏好的正确存法是键值对，不是向量。它的查询方式是「取这个用户的偏好字段」，不需要语义相似度。

被忽略的 47% 检索噪声

去查这四件事的分布，比换一个更强的模型划算得多

当效果不好时，第一反应往往是换模型或调提示词。但把检索噪声拆开看之后，你会发现大部分损失其实发生在检索环节——而它跟模型能力无关。

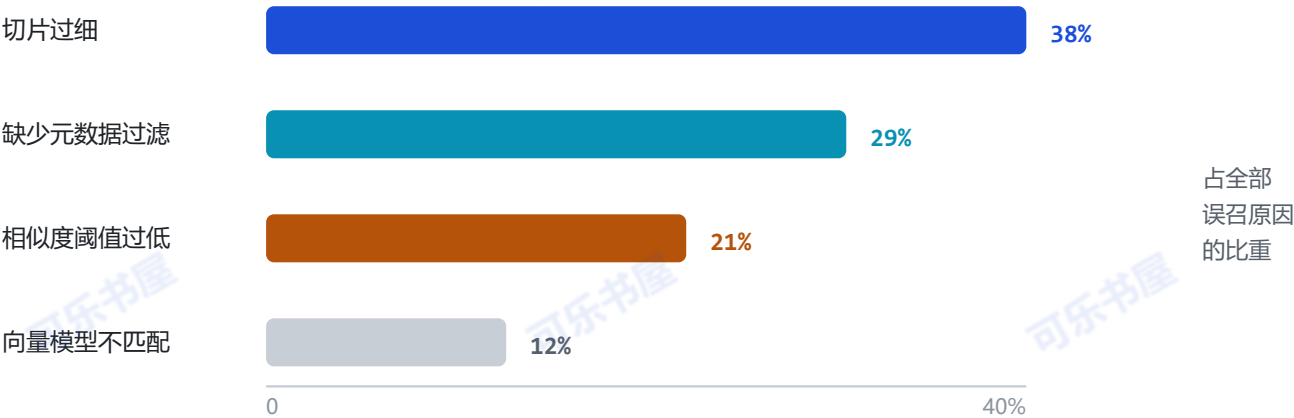


图 3-3 检索噪声来源分解（基于 1200 条误召回样本的人工归因）

噪声来源	修法	预期改善
切片过细	按语义段落切，不按固定字数切；保留 15% 重叠	误召 -31%
缺少元数据过滤	索引时写入时间、来源、版本，检索时先过滤再排序	误召 -44%
相似度阈值过低	设一个下限（如 0.72），低于阈值时宁可不可召回	误召 -26%
向量模型不匹配	用领域语料评估召回质量后再决定，不要凭榜单选型	误召 -19%

性价比最高的两件事

上面四项里，元数据过滤和相似度阈值是最容易被忽略、也最容易实施的两项——它们都不需要重新索引或换模型，改几行检索代码就能验证。

元数据过滤的核心是：把「时效」和「来源权威性」作为硬条件，而不是让向量相似度去权衡。一篇三年前的文档和一篇上周的文档语义可以完全一样，但可用性天差地别。



上线 成本、延迟与安全边界

这一章解决一个问题：让它稳定地、可预期地、安全地跑下去

先看一个例子

开一家店和做一个 Demo 是两回事。Demo 只要证明「这单能跑通」；开店要看的是：一千单下来每单成本多少（成本）、高峰期客人排不排队（延迟）、收银权限和账本能不能查（安全）。Agent 上线卡住的，几乎都不是「跑不通」，而是这三笔账算不过来。



指标本身不是目的，能触发动作的指标才是

图 0-4 上线阶段的核心仪表：三个必须实时可见的读数

四个必埋指标

成本失控归因

四层安全边界

四个必埋指标，缺一个就是盲飞

指标的价值不在「能看见」，而在「越线时你知道该改哪一行」

大部分 Agent 上线后只埋了调用次数和报错率，这两项对定位问题几乎没有帮助。下面四个指标各自对应一类可归因的故障，缺任何一个，排障就只能靠猜。

指标	失守阈值	越线说明什么	第一个该改的地方
任务成功率 端到端完成占比	< 75%	整体能力不足，或任务定义过宽	先看失败样本是否集中在某一类输入——通常是任务边界没收紧
平均工具轮次 每任务平均调用次数	> 6 次	系统提示词没有说清工具使用规则	系统提示词第二节；其次是工具描述里的时机说明
单任务成本 元 / 任务	> 预算 1.5 倍	上下文膨胀或重试失控	压缩策略与终止条件；不要先想着换小模型
P95 延迟 95 分位响应时间	> 8 秒	长尾请求拖累整体体验	先看 P95 与 P50 的倍数——超过 4 倍说明有卡死路径

为什么必须看 P95 而不是平均值

平均值会把长尾藏起来。假设 90% 的请求 2 秒返回，10% 的请求 30 秒返回，平均值是 4.8 秒——看起来很好。但那 10% 的用户体验是「卡死了」，而且他们往往会重复提交，进一步放大成本。**平均值达标、P95 崩塌，是 Agent 上线后最典型的假象。**

一个可用的埋点最小集

如果暂时只来得及埋三个字段，就埋这三个：task_id、tool_rounds、total_cost。它们能直接算出上面四个指标里的三个，且都不依赖模型输出，采集成本几乎为零。把这三个字段按 task_id 打平后做分布，你会立刻看到成功任务和失败任务在轮次与成本上的明显断层。

4 倍

P95 与 P50 的比值。
超过即存在卡死路径

6 次

平均工具轮次的警报线。
超过说明提示词失职

3 字段

最小埋点集，
覆盖三个核心指标

三天烧掉 4000 元：成本失控的三种方式

成本失控从来不是缓慢上升的，它在某一天突然拐弯

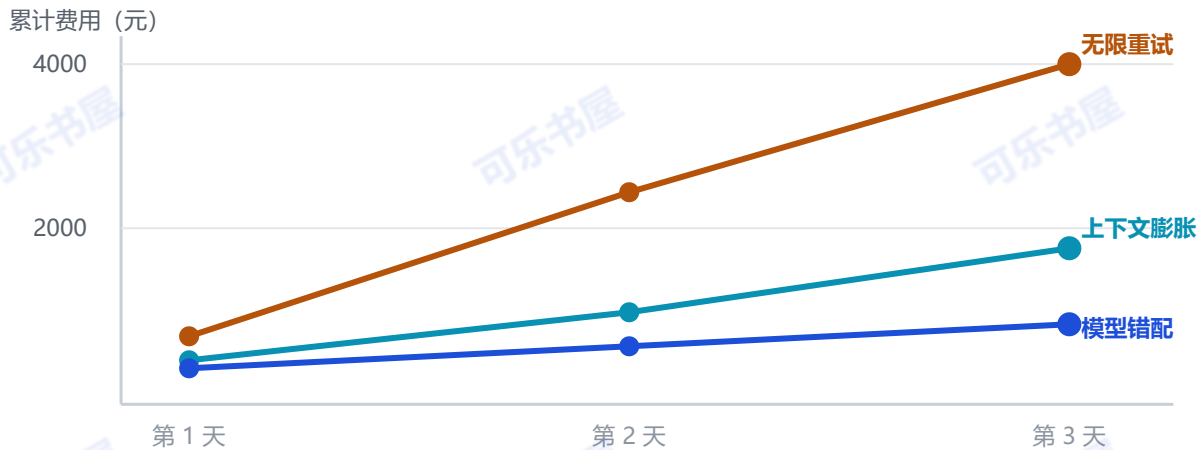


图 4-1 三种失控模式的累计成本轨迹（同一项目，三条独立事故记录）

失控方式	现场现象	根因与修法
无限重试	单次失败重试 17 次，三天 4000 元	只判断了「任务是否完成」，没判断「重复了几次」。加轮次上限与无进展检测，当天止损。
上下文膨胀	单任务成本从 0.04 元涨到 1.7 元	每轮回灌全量历史，且工具返回的原始 JSON 未截断。改分层压缩 + 工具结果限长 800 字。
模型错配	简单分类任务成本高 40 倍	用大模型处理只需规则即可完成的前置分类。把可确定的部分下沉，只把判断点交给模型。

最贵的是第一个

三种里**无限重试**最危险，因为它同时具备两个特征：增长最快，而且不会报错。系统运行正常、日志没有异常、接口全部 200——只有账单在涨。这也是为什么「终止条件」被单独列在第 2 章、而不是放在这一章当优化项。**它是设计问题，不是运维问题。**

加一道兜底

除了单任务级别的终止条件，再加一道**账户级熔断**：当日累计消耗超过预算的 80% 时触发告警，超过 120% 时自动降级到规则兜底。这道防线跟代码质量无关，纯粹是止损用的——而它只需要一个定时任务。

安全边界：四条不能让步的硬规则

提示词里的「请不要做某事」不是安全措施，是建议

把安全约束写在系统提示词里，等于把门锁的钥匙交给门本身。模型可能因为一次注入、一次上下文溢出、一次版本升级就绕过它。**安全必须在模型之外实现**，分四层：

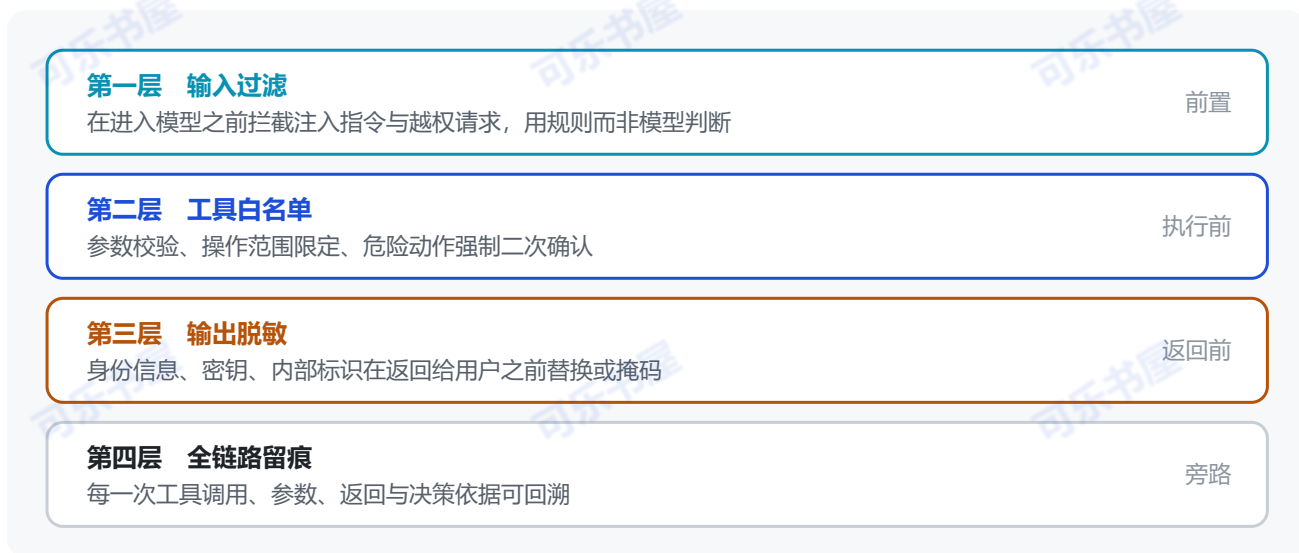


图 4-2 四层安全边界：前三层是控制点，第四层是可追溯性

四条硬规则

规则一：不可逆的操作永远保留人工确认。不管模型多有信心、阈值调到多高。判断标准是「这个操作能否用一次反向操作完全回滚」，不能就加确认。

规则二：工具权限按最小必要分配。查订单的工具不应该带写权限，读文件的工具不应该能删文件。权限边界由系统授予，不由模型自律。

规则三：模型输出不直接拼接进命令与查询。任何进入 SQL、Shell、URL 的内容都要参数化或转义。这一条与是否使用 Agent 无关，是基础设施级别的常识，但在 Agent 场景下更容易被忽略，因为输出的来源看起来「很聪明」。

规则四：第三方内容视为不可信输入。检索到的网页、用户上传的文档、工具返回的外部数据，都可能包含伪装成指令的文本。这类内容在进入上下文时应当被明确标记为「数据，非指令」。

一条经验判断

如果你在系统提示词里写了「绝对不要……」这类句子，那么这条约束**还没有被真正实现**。把它删掉，改成在模型之外的某个地方加一段代码——那段代码的位置，就是这条约束真正的归属。

八个踩坑对照表

现象 / 原因 / 修法——每一条都对应一次真实的排障过程

现象	原因	修法
1. 同一个输入，两次结果不同	采样温度大于 0，且没有固定随机种子	判断类任务把温度设为 0；需要多样性的场景固定种子以便复现
2. 反复调用同一个工具	该工具的返回没有提供新信息，模型误以为仍在推进	加「连续两次调用集合与返回哈希相同即终止」的判定
3. 文档就在上下文里，还是答错	关键片段被放在提示词靠后位置，并被大量无关内容包围	权威依据放在开头，并显式标注「以下为唯一可引用的依据」
4. 上线后成本是测试的 20 倍	测试集都是正常请求，没有覆盖长尾与重试路径	用真实流量分布重算成本，把 P99 轮次纳入预算模型
5. 多轮之后开始胡言乱语	上下文超长触发截断，把最前面的系统提示词截掉了	系统提示词永久保留，压缩只作用于中间轮次
6. 工具报错后 Agent 卡死	错误信息对模型不可读（栈追踪、错误码）	改写为一句人类能照着做的修正指令
7. 加了新工具，老功能退化	新旧工具语义重叠，模型选择被干扰	合并同类工具，或改成分组路由后只暴露相关组
8. 说「记住我上次说的」，它没记住	用户偏好被存进了向量库，被对话历史片段盖过	偏好改用结构化键值存储，按用户 ID 精确读取

这八条的共同点

把它们放在一起看会发现：**八条里有六条的根因不在模型，而在这几件事——终止条件、上下文压缩顺序、检索注入位置、存储选型**。换更强或更便宜的模型，这六条一条都不会自动消失。这也是全书想说明的核心：**Agent 工程化的主要工作，发生在模型之外**。

排障的固定顺序

出问题时报这个顺序查，能避免 80% 的无效排查



图 A-1 Agent 故障排查的固定顺序：从分布到个案，从工程到模型

这个顺序的价值在于**成本递增**。第一步查分布不需要复现任何东西，看数据即可；第四步换模型意味着重新验证全部提示词、工具描述和成本模型。绝大多数团队的问题顺序恰好是反的——先换模型，再怀疑数据，最后才去看上下文。

第三步的做法

在调用模型之前，把最终拼好的完整上下文（系统提示词 + 历史 + 检索片段 + 工具定义）落一份到日志，按 `task_id` 关联。**这一条日志的排障价值，高于你后面做的所有监控面板**。很多团队直到上线三个月后才发现系统提示词一直在被截断。

上线前自检清单

24 条。全部为「是」再上线；任何一条为「否」，都对应一个已知的失败模式

一、定位（4 条）

- ☐ 1 抽样200条请求，输入形态超20种
- ☐ 2 业务方确认 P95 延迟 $\geq$ 1秒
- ☐ 3 写操作均有回滚或人工确认
- ☐ 4 不命中四种「该用工作流」场景

二、构建（7 条）

- ☐ 5 提示词写明「不确定时怎么办」
- ☐ 6 工具数量控制在 3~5 个
- ☐ 7 单工具必填参数不超过 3 个
- ☐ 8 工具描述写明使用时机
- ☐ 9 工具报错改为可执行指令
- ☐ 10 配置了最大轮次上限
- ☐ 11 配置了「无进展」检测

三、增强（6 条）

- ☐ 12 实测过注入片段数准确率曲线
- ☐ 13 上下文用到 60% 即触发压缩
- ☐ 14 压缩时永久保留系统提示词
- ☐ 15 偏好键值表，未进向量库
- ☐ 16 检索设了相似度下限
- ☐ 17 索引含时间与来源元数据

四、上线（7 条）

- ☐ 18 已埋四个核心指标
- ☐ 19 监控 P95 与 P50 的比值
- ☐ 20 配置日消耗熔断与降级
- ☐ 21 每次调用完整上下文落日志
- ☐ 22 工具权限按最小必要分配
- ☐ 23 输出未直接拼接进 SQL 与 Shell
- ☐ 24 第三方内容标记为「数据，非指令」

用法建议

不要一次性全部对齐再上线。**第 5、10、11、12、18 这五条是止损线**，先保证它们为「是」即可灰度，其余条目在灰度期间逐步补齐。

Agent 的难点不在 让它跑起来， 在于让它停下来。

这本书从一笔算错的账开始，最后落在这四件事上：**算清楚该不该做、把四个部件补齐、决定给模型什么而不是给多少、把安全边界放在模型之外**。它们有一个共同点——都不发生在模型内部。

这可能是这个领域最容易被忽略的事实：当所有人都在讨论哪个模型更强时，真正决定一个 Agent 能否上线的，仍然是那些朴素的工程问题。终止条件、压缩顺序、权限边界、可观测性——这些问题在传统后端里已经被解决了无数次，只是换了一个场景重新出现。

所以如果你读完这本书只记住一句话，我希望是这句：**当你觉得「模型不行」的时候，先去看它实际拿到了什么上下文**。九成的情况下，问题不在模型，而在你递给它的那张纸上。

关于下一步

附录 B 的 24 条清单可以直接当作工程任务拆解表使用。如果你手上已经有一个在跑的 Agent，建议这周先做三件事：把 **最大轮次上限** 配上，把 **完整上下文落日志** 打开，把 **单任务成本** 接进监控。这三件事加起来不到一天，但能让你在出问题时至少看得见。



可乐书屋

AI Agent 工程化落地实战

四章 · 二十八个专题 · 二十一张图示 · 二十四条自检清单

本书为二次创作作品，知识框架部分参考自开源项目 AI Agents for Beginners (MIT License, Copyright (c) Microsoft Corporation) ,

项目地址 <https://github.com/microsoft/ai-agents-for-beginners> ; 章节结构、文字表述、案例数据与全部图示均为本书独立编写绘制。

本书不隶属于上述项目版权方，亦未获得其官方认可或赞助。文中企业名称与产品名称仅作事实性指代，不构成商标使用授权或合作关系。

内容为工程实践整理，不构成技术咨询、法律意见或投资建议。未经许可，不得用于商业再分发。