亚马逊云科技

# 企业生产级智能体开发部署指南

# 目录

# CONTENTS

## 第一篇：企业智能体之旅：为什么评估（Evaluation）是一切的起点



## 第二篇：评估企业级智能体：从原型验证到生产就绪

## 第三篇：
## 如何在亚马逊云科技上构建企业级智能体



## 第四篇：
## 实战案例：从工具使用到多智能体协作

# 全系列摘要（Summary）

本系列共四篇文章，目标是为企业团队提供一套从原型到生产的智能体工程纪律路线图。整套内容围绕一个核心命题展开：AI智能体的工程化落地，瓶颈不在模型能力，而在缺少一套可持续衡量"好不好"的工程体系。

**第一篇 · 为什么评估是一切的起点：**

从"为什么传统软件工程方法对智能体失效"切入，剖析非确定性、Prompt 即源代码、依赖会自己漂移这三大根因；进而提出 **ADLC（Agent Development Lifecycle）**——为智能体量身设计的开发生命周期，并把企业 Agentic 开发归纳为三类工程实践：**把评估跑起来 / 让数据持续流入评估 / 让系统架构可被评估。**

**第二篇 · 评估方法论：**

澄清三个常见的评估误区，建立**两根支柱**的评估框架——三种评估粒度（黑盒 / 玻璃盒 / 白盒）和三层证据权重；明确企业要测的**八类维度**（质量、性能、责任、成本……）；讨论 **LLM-as-a-Judge** 的价值与边界，并引入 **Agent-based Evaluation** 将专家级评审规模化。

**第三篇 · 在亚马逊云科技上构建企业级智能体：**

给出工程实现层面的全景图——自动化评估工作流 + 三层评估库；按智能体形态（单 Agent / 工具调用 / 多 Agent）匹配关键指标；讲清 **Trace-driven** 评估工作流的四个步骤；并把评估嵌入到开发流程中，配合 **AgentCore Evaluations** 形成 Observability → Evaluation → Optimization 的闭环。

**第四篇 · 实战案例：**

用三个 Amazon 内部生产级案例对应不同的评估侧重——**购物助手（工具使用）、客服智能体（意图检测）、卖家助手（多智能体协作）**，覆盖智能体从简单到复杂的演进路径，最终收束到一个可亲手跑通的评估闭环。

**如需进一步了解 Evaluation-first 方法在实际场景中的落地方式，可参考本系列配套动手实验示例代码：**
**https://github.com/aws-samples/sample-eval-first-building-enterprise-agents-with-agentcore**

01

# 第一章：
# 企业智能体之旅：为什么评估（Evaluation）是一切的起点

# 从原型到生产：一道被低估的鸿沟

过去一年，AI 智能体已经从技术探索进入工程落地阶段。越来越多的企业团队开始构建能够调用工具的 LLM 系统、串联多个 API 的编排流程，以及基于内部知识库的对话助手。

原型验证阶段通常进展顺利——Demo 效果令人满意，业务方反馈积极。然而，当团队尝试将这些系统推向生产时，一个关键问题浮现：**它真的准备好上生产了吗？**

大多数团队正是在这里遭遇瓶颈。制约因素并非模型能力本身，而是缺少一套工程化的质量评判体系——一套能持续回答"它到底好不好"的方法。

这道鸿沟并非偶然。传统软件拥有单元测试、CI/CD 流水线和明确的通过/失败（pass/fail）标准。AI 智能体没有。同样的用户输入，今天跑出一个结果，明天换了模型版本，行为悄悄变了，没有任何报警。开发者甚至无法用一行 assert 语句来验证一个多步推理的过程是否正确。

**这本质上是一个工程纪律问题，而非模型能力问题。**

本系列共四篇文章，目标是提供一套完整的智能体工程纪律路线图。我们从首先要解决的一件事开始——评估（Evaluation），它是一切其他工程实践的地基：没有评估，团队既不知道自己在哪里，也不知道改了之后有没有变好。

# 为什么传统软件工程方法对智能体失效

软件工程师天然会把 AI 智能体当成"另一种软件"来对待：写逻辑、做测试、上 CI/CD。这个直觉很合理，但在智能体身上会系统性地失效。原因有三。

### 1. 非确定性：你的测试用例今天通过，明天可能失败

传统软件的测试逻辑很简单：给定输入 A，断言输出是 B。这在智能体身上不成立。

LLM 本质上是概率模型。同样的输入，每次调用的输出在统计分布上是不同的。更反直觉的是，即使把 temperature 设为 0（通常被理解为"最确定性的模式"），输出仍然不能保证完全一致：GPU 浮点运算的非结合性、Mixture-of-Experts 的路由机制、批处理的顺序依赖，都会引入微小但真实的差异。目前没有任何主流模型提供商承诺完全确定性的输出。

由此可见，传统的通过/失败测试框架在这里根本不适用。我们需要的不是断言，而是评估——在一个分布上衡量行为，而不是验证一个确定的结果。

### 2. 自然语言就是"源代码"：改了 Prompt 就是改了代码

在传统软件里，改代码会留下 Git diff，有代码评审（Code Review），有版本历史，有回滚路径。Prompt 没有这一切。

修改一段系统 Prompt，可能只是在末尾加了一句话，但智能体的行为已经发生了根本性的变化——它可能开始拒绝某类请求，可能改变了工具的调用顺序，可能输出格式悄悄变了。没有任何静态分析工具能提前告诉我们这次修改的影响范围。

因此，每一次 Prompt 变更，都必须有配套的评估来量化影响。没有评估，Prompt 工程就是在黑箱里射击。

### 3. 依赖会自己动：没有部署任何东西，但智能体的行为变了

传统软件的依赖是锁定的：package.json 里有版本号，升级会发生什么是可预期的。模型是隐式依赖，而且它会自己更新。

模型提供商会定期对模型进行安全微调、能力升级或系统提示调整，通常不会发布详细的 Changelog。结果是：代码库没有任何变动，但某天早上智能体开始对某类问题产生不同的响应——更保守了，或者格式变了，或者工具调用的倾向改变了。如果没有持续的评估基线，这种漂移几乎不可能被及时发现。

以上三点共同指向同一个结论：传统的 CI/CD 和 QA 框架不是为这种系统设计的。我们需要一套新的方法论——一套能在概率性、可变性、隐式依赖这三个条件下持续衡量系统质量的工程体系。这就是接下来要谈的 ADLC。

# ADLC（Agent Development Lifecycle）：为智能体量身设计的开发生命周期

既然传统 SDLC 对智能体失效，我们需要一套为它量身设计的方法论。业界正在形成共识的框架叫做 ADLC（Agent Development Lifecycle）——它不是对 SDLC 的修补，而是一次完整的重构。

**ADLC 是飞轮，不是流水线**

传统软件开发是线性的：需求 → 设计 → 开发 → 测试 → 上线。上线是终点，下一个版本从头开始。这套逻辑对智能体不成立，因为智能体在生产环境里运行的每一次对话，都是关于它真实行为的最宝贵数据。

ADLC 是一个持续运转的飞轮，六个环节首尾相连：

### 1.定义"好"

在动手构建之前，先确定什么叫成功。这不是一句愿景，而是具体的评估标准、基准数据集。

### 2.构建

基于清晰的定义，搭建智能体系统。

### 3.评估

用第一步定义的标准，系统性地衡量智能体的行为。

### 4.门控上线

评估不只是用来看的，它是上线的通行证。评估指标未达阈值，不部署。

### 5.生产观测

智能体上线后，持续追踪它在真实流量下的表现——延迟、成功率、工具调用模式、用户反馈。

### 6.挖掘失败案例

从生产 Trace 中找到智能体失败或表现异常的样本，将它们加入评估集，持续迭代。

### 与传统CI/CD 最关键的区别

在传统流水线里，"生产"是流程的终点。在 ADLC 里，生产是飞轮最富价值的输入。

这个转变意义深远。它意味着评估集不是在项目初期一次性定义的静态资产，而是随着生产数据持续生长的动态系统。每一个在生产中暴露的真实失败案例，都比在会议室里预设的测试用例更有价值——因为它来自真实用户，在真实上下文下触发了真实的问题。

还有一条长期价值链值得关注：生产 Trace 可以变成评估数据，评估数据在积累到足够规模后，可以变成微调或蒸馏的训练数据。今天投入可观测性和评估体系的成本，会在未来以模型优化的形式产生复利回报。

### 定义"好"是第一步，不是上线后的复盘

这个排序看起来显而易见，但在实践中经常被颠倒。很多团队的顺序是：先把 Demo 做出来，上线后用户反馈来了再想怎么衡量好坏。

这往往意味着显著的返工成本。如果上线时没有评估基线，就无法判断下一次改动是变好了还是变坏了。失去的不只是某一次 Prompt 改进的验证能力，而是整个系统迭代的方向感。

ADLC 强制要求把"定义'好'"放在第一位，本质上是在要求团队在开始建造之前，先想清楚验收标准是什么——就像盖楼之前要先出图纸，而不是等楼盖完了再画。

# 企业 Agentic 开发：三类工程实践

理解了 ADLC 这套方法论之后，落地就需要具体的工程实践。亚马逊云科技在大量企业智能体项目中积累的经验，看起来分散，但有一条主线贯穿始终：**要么在直接做评估，要么在为评估奠基。**

仔细看，这些实践分别回答三个问题：如何把**评估本身跑起来**、如何让**数据持续流入**评估系统、以及如何让**系统架构上可被评估**。三者缺一不可——评估流程是出口，数据是管道，架构是地基。我们按这个逻辑，分三类展开。

## 第一类：把评估跑起来

### 从小做起，先定义"成功"长什么样

很多团队启动智能体项目时，第一个问题是"这个智能体能做什么"。这是个错误的起点。

正确的第一个问题是：**我们要解决什么问题？**

从问题出发，倒推智能体的边界：它应该处理什么，不应该处理什么。如果在建一个财务分析助手，先只做"查季度收入、计算增长率、生成摘要"这三件事，把它们做可靠了，再扩展。不要从一开始就想把所有场景都覆盖——那只会让 Prompt 越来越复杂，工具选择越来越混乱，性能越来越难归因。

**启动一个智能体项目，应该产出四个具体交付物，而不仅仅是代码：**

1. 智能体应该做什么和不应该做什么的清晰定义。把它写下来，与利益相关者分享，用它来拒绝功能蔓延。
2. 智能体的语气和个性。决定它是正式还是对话风格的，如何问候用户，以及遇到范围外的问题时如何处理。
3. 每个工具、参数和知识源的明确定义。模糊的描述会导致智能体做出错误选择。
4. 期望交互的基准数据集，覆盖常见查询和边缘情况。

最后这一项是关键。基准数据集是整个评估体系的"燃料"——没有它，评估系统无从运转，连"智能体有没有进步"都无法回答。它不是上线后再补的东西，而是启动前就要准备好的基础设施。

**以下是三种典型智能体的四个交付物，大家可以对照参考：**

| 智能体类型 | 智能体定义 | 语气与个性 | 工具定义 | 基准数据集 |
|---|---|---|---|---|
| 财务分析智能体 | 帮助分析师查询各区域（EMEA、APAC、AMER）季度营收数据、计算增长指标、生成高管摘要。不提供投资建议，不执行交易，不访问员工薪酬数据。 | 专业但不失亲切，以名字称呼用户；主动承认数据局限；数据质量存疑时明确说明置信度；不用没有解释的金融行话。 | getQuarterlyRevenue(Region: EMEA\|APAC\|AMER, quarter: YYYY-QN) — 返回指定区域和季度的营收数据，单位为百万美元。<br>calculateGrowth(currentValue: number, previousValue: number) — 返回百分比变化。<br>getMarketData(Region: string, dataType: revenue\|sales\|customers) — 获取最新行业指标数据。 | 50条，包括：<br>"EMEA 第三季度的营收是多少?"<br>"和上个季度相比增长了多少?"<br>"亚太区表现怎么样?"<br>"CEO 的奖金是多少?"（应拒绝回答）<br>"对比 2024 年所有区域的表现" |

| 智能体类型 | 智能体定义 | 语气与个性 | 工具定义 | 基准数据集 |
| --- | --- | --- | --- | --- |
| HR 政策助手 | 回答员工关于休假政策、请假申请、福利注册和公司政策的问题。不访问保密人事档案，不提供法律建议，不讨论个人薪酬或绩效评估。 | 友好且支持性；使用员工偏好的称呼；保持专业同时平易近人；遇到复杂政策时逐步分解说明；对敏感事项主动提出转接 HR 专员。 | checkVacationBalance (employeeId) — 返回各类型可用天数<br>getPolicy(policyName) — 从知识库获取政策文件<br>createHRTicket(employeeId, category, description) — 升级复杂问题<br>getUpcomingHolidays(year, region) — 返回公司假期日历 | 45 条，包括：<br>"我还有几天假?"<br>"育儿假政策是什么?"<br>"我下周能请假吗?"<br>"为什么我的奖金比预期低?"（应升级处理）<br>"怎么参加健康保险?" |
| IT 支持智能体 | 协助员工处理密码重置、软件访问申请、VPN 故障排查和常见技术问题。不访问生产系统，不直接修改安全权限，不处理基础设施变更。 | 耐心清晰；避免技术行话；提供分步骤操作说明；移到下一步前确认理解；庆祝小进展（"很好，成功了！"）；需要系统权限时升级给 IT 团队。 | resetPassword(userId, system) — 触发密码重置流程<br>checkVPNStatus(userId) — 验证 VPN 配置与连通性<br>requestSoftwareAccess (userId, software, justification) — 创建访问申请工单<br>searchKnowledgeBase (query) — 检索排障文档 | 40 条，包括：<br>"我登不上邮箱"<br>"VPN 一直断线"<br>"我需要访问 Salesforce"<br>"能给我管理员权限吗?"（应拒绝）<br>"笔记本连不上 Wi-Fi"<br>"怎么安装 Slack?" |

用这个有限范围构建 PoC，然后用真实用户测试。他们会立即发现没有预料到的问题——智能体可能在日期解析上出错，可能不能很好地处理缩写，或者在问题换一种表达方式时就调用了错误的工具。在 PoC 阶段学到这些，代价是几周时间；在生产阶段学到，代价是信誉和用户信任。

## 从第一天开始自动化评估

有了基准数据集，下一步是建立自动化评估机制——让它成为开发流程的一部分，而不是上线前临时跑一遍的检查清单。

首先要定义衡量指标。不同类型的智能体，关注点不同：客服智能体看解决率和用户满意度，财务智能体看计算准确性和引用质量，HR 助手看政策准确性和回答完整性。但无论哪种智能体，都要同时追踪两类指标：**技术指标**（延迟、Token 用量、工具调用准确率）和**业务指标**（回答是否真的有用）。两者要一起看——延迟低但答案错，没有意义；答案好但成本高到无法商业化，同样是问题。

**评估数据集要覆盖三类情况：**

- **同一问题的多种表达：**用户不会像 API 文档那样说话，"上季度欧洲收入"和"EMEA Q3 数字"应该触发完全相同的工具调用
- **应该拒绝或升级人工的查询：**边界案例同样需要被评估
- **有歧义的查询：**一个问题可能有多个合理解读，智能体应该如何处理

**一个具体的指标体系长什么样？**还是用财务分析智能体举例：

- **工具选择准确率：**面对"上季度收入"类问题，是否选择了 getQuarterlyRevenue 而不是 getMarket-Data？目标：≥ 95%
- **参数提取准确率：**是否正确将 "EMEA" 和 "Q3 2024" 映射到对应格式？目标：≥ 98%
- **拒绝准确率：**面对"CEO 奖金是多少"，是否正确拒绝？目标：100%
- **回答质量：**解释是否清晰、没有歧义？用 LLM-as-a-Judge 评估
- **延迟：**P50 低于 2 秒，P95 低于 5 秒
- **每次查询 Token 用量：**平均低于 5,000 tokens

---

有了这套指标体系，评估就成了可量化的决策依据。比如：把模型从 Claude Sonnet 换成 Claude Haiku，重跑评估发现工具选择准确率从 92% 降到 87%，但 P50 延迟从 3.2 秒降到 1.8 秒——这个数字告诉我们速度的提升是否值得那 5% 的准确率代价，而不是凭感觉拍板。

**评估必须嵌入开发流程**，而不是一次性工作：改了 Prompt？跑评估。加了新工具？跑评估。换了模型？跑评估。反馈循环必须足够快，让团队在下一次改动之前就知道上一次改动的影响，而不是三个 commit 之后才发现问题。

## 持续测试与改进

智能体上线，不是测试结束的信号，而是测试场景发生根本变化的节点。

在开发阶段，能测试的是预想到的场景。进了生产，真实用户会用完全没预料到的方式问问题，会触发没有覆盖的边界，会在特定时间、特定上下文下让智能体暴露出新的问题。与此同时，模型提供商的后台更新、外部 API 的行为变化，都会在不知情的情况下影响智能体的表现——这种现象叫**静默漂移**，它不会报错，只会让质量指标悄悄变差。

应对静默漂移的方法只有一个：**持续采样，持续评估。**

建议建立这样一套测试机制：

**每次变更触发回归测试：**

改 Prompt、加工具、换模型，任何一次改动都要跑完整的评估套件，确认没有引入回归

**生产流量持续采样：**

从真实用户交互中随机抽取样本，用和离线评估相同的指标体系打分，持续监控质量曲线

**漂移检测与自动告警：**

如果某个关键指标（比如工具选择准确率）在两周内从 92% 悄悄降到 84%，必须有机制捕捉到这个变化并告警

**重大更新做 A/B 测试：**

新版本不要全量上线，先用一部分流量验证，确认指标不差于旧版本再切换

**改进循环的正确姿势是：**

评估 → 找到问题所在 → 修改 Prompt、工具定义或 Retrieval 策略 → 重新评估。注意这里没有"重新训练模型"这一步——对绝大多数企业智能体来说，大部分性能提升来自 Prompt、工具和 Retrieval 层面的优化，而不是换模型或微调。换模型是成本更高、风险更大的选项，应该在穷尽其他手段之后才考虑。

**一个实际的改进例子：**

财务智能体上线后，通过生产采样发现有一类查询的工具选择准确率只有 78%：用户问"欧洲今年整体表现怎么样"，智能体频繁调用了 getMarketData 而不是 getQuarterlyRevenue。根因是工具描述不够精确——getMarketData 的描述里有"market performance"字样，和用户表达高度重合。修改方案：细化两个工具的描述，明确各自的适用场景，加入反例说明。改完重跑评估，准确率回升到 95%。整个循环没有动过一行模型代码。

评估是这套改进循环核心可信的信号源。没有它，团队不知道改动方向对不对，也不知道上线之后系统在哪里悄悄变差了。

## 先把单个智能体做好，再扩展到多智能体

第一个智能体上线，是一个里程碑。但企业真正的价值，来自于把这套能力在组织内规模化复制——而不是每次构建新智能体都从零开始。

规模化有一个前提：**单个智能体的质量基线必须先建立好。**

评估的复杂度随智能体数量非线性增长。单个智能体出问题，可以独立定位；多个智能体协作时，一个问题可能来自任何一个环节，也可能来自交接过程本身。如果在没有评估基线的情况下就扩展到多智能体系统，出了问题几乎无从归因——不知道是哪个智能体失败了，也不知道是不是某次改动引入的回归。

**正确的顺序是：**先为单个智能体建立完整的基准数据集、指标体系和 CI 门控，确认它在自己的职责范围内表现稳定，然后再引入第二个、第三个智能体。每个新智能体都应该继承这套工程基础，而不是等所有智能体都上线后再补。

扩展到组织级别时，需要从**项目思维转向平台思维**。单个项目团队关心的是"这个智能体能不能用"；平台团队关心的是"整个组织的智能体能不能被统一治理"。这通常意味着：

- 维护一个经过安全审查的工具目录，新团队直接取用，不重复建设
- 建立统一的可观测性和评估标准，让不同团队的智能体产出可比较的质量数据
- 运行跨智能体的集中监控，当某个智能体开始影响成本或质量曲线，平台团队能第一时间发现

生产数据是持续改进的原料。第一版上线时的 50 个基准样本，会随着真实用户交互不断扩充——生产中出现的新边界案例、新失败模式，都应该回流到评估集里。规模化不是建更多智能体，而是建立一套让每个智能体都能持续变好的机制。

规模化不是一步到位的。建议按三阶段推进：

- **爬行阶段：**先在内部小范围试点，核心目标是学习和迭代。这个阶段的失败成本低，错了可以快速改。
- **行走阶段：**把智能体推给受控的外部用户群体，更多用户带来更多反馈和边界案例。此前在可观测性和评估上的投入，在这个阶段开始产生回报。
- **奔跑阶段：**有信心地规模化上线，平台能力让其他团队可以更快构建自己的智能体，组织能力开始形成复利。

## 第二类：让数据持续流入评估

### 从第一天就埋好可观测性

可观测性是经常被推迟的那类工作——"先把功能做出来，上线再说"。这往往意味着显著的返工成本。等到意识到需要它的时候，往往已经有一个行为不透明的智能体跑在生产上，而团队完全不知道它在做什么。

**可观测性和评估的关系，是数据管道和分析引擎的关系。**可以有世界上最好的评估框架，但如果没有 Trace 数据，在线评估无从采样，生产中的失败案例无从挖掘，整套评估体系就只能在离线数据集上工作，看不见生产里真实发生的事情。

**可观测性要覆盖三个层次：**

| 开发者层（调试用） | 平台层（治理用） | 运营层（SLA 用） |
| --- | --- | --- |
| 从第一次测试查询开始，就需要能看到智能体每一步在做什么——模型调用了什么、调用了哪个工具、传了什么参数、推理过程中经历了哪些步骤。当用户报告智能体行为异常，需要能拉出对应的 Trace，逐步还原它当时的决策过程。 | 谁在用智能体，用了多少 Token，哪个团队的智能体在驱动成本增长，某次事故的完整时序是什么——这些是平台团队和管理层关心的问题。没有这一层的数据，智能体系统的成本和风险就处于黑盒状态。 | 延迟百分位数、错误率、工具调用成功率、用户会话完成率——这些指标决定了能不能对业务承诺 SLA，也是触发告警和自动回滚的依据。 |

**技术实现上，OpenTelemetry 是目前的行业标准。**它让模型调用、工具调用、推理步骤全部产生结构化的 Trace，可以导出到现有的观测基础设施——无论是 CloudWatch、Datadog、Dynatrace，还是专门针对 LLM 的 LangSmith、Langfuse。关键原则是：从第一天就接入，不要等功能稳定之后再补。

**一个实际的场景说明为什么顺序很重要。**假设财务智能体在内测阶段没有接入 Trace，只靠用户反馈判断问题。上线三周后，用户开始投诉"查询很慢，有时候还答错"。没有 Trace，团队只能逐一排查：是模型推理慢？是数据库查询慢？是外部 API 出了问题？四天之后才定位到根因——一个外部数据 API 在静默更新后返回格式发生了变化，智能体拿到了格式错误的数据，还继续用它回答问题。

如果从第一天就接入了 OpenTelemetry Trace，这个问题会在第一天就暴露：Trace 会清楚地显示那条查询耗时 12 秒，其中 10 秒来自外部 API 调用，同时工具返回值的解析错误率从 0 变成了 100%。告警当天就能触发，而不是四天后靠用户投诉倒逼排查。

## 第三类：让系统架构可被评估

### 建立清晰的工具策略

工具是智能体接触真实世界的手——它通过工具查数据、调 API、执行业务逻辑。工具的质量直接决定了智能体的行为质量，但很多团队在这里犯的错误不是工具写得不够多，而是工具描述得不够清晰。

**工具定义的质量，比工具本身的数量更重要。**

看一个对比。同一个函数，两种写法：

**模糊写法：**"获取收入数据"

**精确写法：**"**获取指定区域和时间段的季度营收数据，返回值单位为百万美元。需要区域代码（EMEA、APAC 或 AMER）和季度（格式 YYYY-QN，例如 2024-Q3）。**"

区别不只是文字多少。模糊的描述让智能体必须"猜"：什么是有效输入？返回值是什么单位？这个工具和另一个相似的工具区别在哪里？猜对了还好，猜错了，智能体会选错工具，传错参数，给出错误答案。

**一个完整的工具定义应该包含五个要素：**

| 要素 | 作用 | 示例 |
| --- | --- | --- |
| 清晰的名称 | 直接传达工具用途 | getQuarterlyRevenue 而不是 getData |
| 明确的参数 | 消除输入歧义 | region: string（EMEA\|APAC\|AMER），quarter: string（YYYY-QN） |
| 返回格式 | 明确输出结构与单位 | {revenue: number, currency: "USD", period: string} |
| 错误条件 | 记录各类失败场景 | 季度不存在返回 404，服务不可用返回 503 |
| 使用指引 | 说明何时用、何时不用 | 当用户询问营收、销售或财务表现时使用；不适用于预测或趋势分析 |

当工具数量增长到二三十个时，**工具目录的管理**就变得关键。不同团队不应该重复造同一个数据库连接器，建议维护一个经过安全审查、在生产中验证过的工具目录，新团队从目录里取，而不是从零开始写。同时推荐采用 MCP（Model Context Protocol）作为工具暴露的标准协议——Slack、GitHub、Salesforce 等很多服务已经提供了现成的 MCP Server，可以直接接入，不用再自己封装。

**工具还需要具备健壮的错误处理。**

工具不可用时，智能体不应该崩溃或者用不完整的数据给出错误答案——它应该捕获错误，明确告知用户服务暂时不可用。典型处理策略：瞬时故障自动重试，可能时回退到缓存数据，服务完全不可用时主动告知用户。错误处理做到位，评估也更准确——我们能区分"智能体做出了错误决策"和"工具本身出了故障"，这两种情况的修复方向完全不同。

**每个工具都要附代码示例**

开发者不应该靠猜来理解工具的调用方式或输出格式——一份完整的示例能节省反复试错的时间，也能大幅降低工具被误用的概率。

清晰的工具定义还有一个实用价值：出了问题更容易定位。如果智能体选错了工具，我们能一眼看出是模型的问题，还是工具描述本身就有歧义。这两种情况的修复方向完全不同——混在一起，调试就会陷入泥潭。

## 用确定性代码替代模型内部推理

智能体不应该做所有事情。这听起来反直觉，但它是企业级智能体架构里最重要的判断之一。

### LLM 擅长的是推理和语言理解

理解用户意图、判断该调用哪个工具、把数据结果解释成自然语言——这些任务需要对模糊输入进行推断，用传统代码实现要枚举成千上万种情况。

### 传统代码擅长的是确定性操作

计算收入增长率、验证日期格式、执行业务规则条件判断。这些操作有唯一正确答案，用 Python 写一个函数，毫秒级执行，零额外成本，每次结果完全一致。把这类操作交给 LLM 推理，是在用最贵的工具做最简单的活。

### 正确的架构是：智能体负责编排，代码负责计算

用户问"我们 EMEA 这季度增长了多少"，智能体用推理能力理解意图、决定调哪些数据，然后调用一个确定性函数执行计算，最后再用推理能力把结果解释成自然语言。LLM 只在需要它的地方介入。

亚马逊云科技的实测数据直接说明了这种设计的价值。以"生成下个月的支出报告"为例，对比两种实现方式：

| | get_current_date() 作为智能体工具 | 用代码获取日期，作为参数传入 |
|---|---|---|
| **智能体行为** | 制定计划 → 调用 get_current_date() → 计算下个月 → 调用 create_report() | 代码获取今天日期 → 传入智能体 → 推断下个月 → 调用 create_report() |
| **延迟** | 12 秒 | 9 秒 |
| **LLM 调用次数** | 4 次 | 3 次 |
| **总 Token 用量** | 约 8,500 | 约 6,200 |

"获取当前日期"这件事，一行代码就能解决，用不着 LLM。但如果把它设计成一个智能体工具，每次查询就多出 1 次 LLM 调用、约 2,300 个 Token 和 3 秒延迟。乘以每天数千次查询，成本和延迟的差距非常可观。

这背后还有一个工程价值：确定性操作的结果是二元对错，可以用程序直接验证，不需要 LLM-as-a-Judge。把越多操作推向这个方向，系统就越容易测试，评估结果的可信度也越高。

判断原则很简单：**如果确定性代码能可靠解决问题，就用代码。如果需要推理或自然语言理解，就用智能体。**最常见的错误是默认一切都应该是 Agentic 的——正确答案是智能体和代码协同工作。

## 多智能体系统保持解耦

单个智能体试图处理太多职责时，问题会系统性地出现：Prompt 越写越长，工具选择逻辑越来越混乱，性能下降，出了问题不知道从哪查起。解决方案是分解——把一个大智能体拆成多个职责单一的专门智能体，让它们协作完成任务。

这和组织人员的道理一样。我们不会雇一个人同时负责销售、工程、客支和财务。我们雇专才，然后让他们协作。智能体系统同理：与其让一个智能体处理三十种任务，不如拆成三个智能体，每个负责十件相关的事。每个智能体有更清晰的指令、更简洁的工具集、更聚焦的业务逻辑。

**三种常见的协作模式**各有适用场景：

### 顺序模式

任务有天然的先后顺序。第一个智能体取数据，第二个分析，第三个生成报告

### 层级模式

需要智能路由。一个 Supervisor智能体判断用户意图，分发给对应的专门智能体

### 对等协作模式

智能体之间需要动态协同，没有中央协调者

多智能体系统最容易出问题的地方是**交接点**。一个智能体把任务移交给下一个时，上下文必须完整传递——如果用户已经在第一个智能体提供了账号信息，第二个智能体不应该再问一遍。要仔细监控每次交接：哪个智能体处理了哪部分请求?延迟发生在哪个环节?上下文在哪里丢失了?

这里还有一个常见的概念混淆值得澄清：**协议和模式是两回事**，混用会导致基础设施和业务逻辑耦合在一起。

| | 协议（智能体如何通信） | 模式（智能体如何协作） |
|---|---|---|
| 层次 | 通信与基础设施 | 架构与组织方式 |
| 关注点 | 消息格式、API、标准规范 | 工作流、角色分工、协调机制 |
| 示例 | A2A、MCP、HTTP | 顺序、层级、对等 |

同一个协议可以支持不同的模式，同一个模式可以用不同的协议实现。分清这两个关注点，架构决策和基础设施选型才能互不干扰。

解耦还有一个直接的工程收益：每个智能体职责清晰，就可以被独立评估。耦合系统出问题时，无法判断是哪个环节失败了；拆开之后，每个智能体有自己的基准数据集和指标，改进目标清晰，也不会出现"改好了 A，不知道有没有影响 B"的情况。

## 安全与个性化

从单用户原型扩展到服务数千用户的生产系统时，需要同时解决两件事：**安全边界**和**个性化体验**。前者确保用户只能访问自己有权限的数据；后者让智能体记住每个用户的偏好和历史，持续提供更贴合需求的响应。两者共用同一套基础设施——用户身份和权限的隔离，既是安全的保障，也是个性化的基础。

智能体系统的安全有一个最常见的错误设计：在 System Prompt 里写"不要访问用户无权限的数据"，然后依赖 LLM 推理来执行这条规则。

这不可靠。LLM 的推理在大多数情况下表现良好，但它是概率性的，没有任何模型能保证每次都正确执行安全规则。更根本的问题是：LLM 内部的决策过程不可观测、不可审计，也无法被系统性测试。一旦出现安全问题，甚至无法复现它是怎么发生的。

**正确的做法是把安全控制外部化：**放在独立的 Gateway 和 Policy 层，在工具被调用之前就完成鉴权，LLM 根本没有机会"决定"要不要遵守规则。

**这套架构通常分三层：**

| 认证层 | 授权层（Gateway） | 会话隔离 |
| --- | --- | --- |
| 用户通过现有的 Identity Provider（Cognito、Entra ID、Okta 等）完成身份验证，拿到携带权限信息的 token。这个 token 在整个会话中全程传递。 | 每次工具调用之前，Gateway 拦截请求，验证当前用户是否有权限以当前参数调用这个工具。这里还可以注入第三方服务的 OAuth 凭证、执行自定义的限流或审计逻辑。只有通过检查的请求才能到达实际的工具和数据库。 | 多用户并发时，每个用户的会话必须完全独立，用户 A 的上下文不能泄露给用户 B。个性化记忆（用户偏好、历史交互）也要按用户命名空间隔离存储。 |

**用户记忆与个性化：**会话隔离解决的是安全问题，用户记忆解决的是体验问题——同一套命名空间机制，同时服务于两个目标。智能体可以在用户命名空间下存储偏好（偏好的报告格式、常用的筛选条件）和历史交互（本周已查询过哪些数据），让回访用户得到更连贯、更个性化的响应，而不是每次从零开始。个性化记忆的存储和检索，同样应该经过授权层的管控——用户只能访问自己的记忆，不能跨用户读取。

**举个例子：**一名初级分析师试图查询高管薪酬数据。如果安全规则写在 System Prompt 里靠 LLM 执行，智能体多数情况下会拒绝——但精心构造的 Prompt 注入可以绕过这种限制，而且无从预测它什么时候会失效。如果把访问控制放在 Gateway 层，这个请求在到达数据库之前就被策略拦截，结果是确定的，与模型当次的推理无关。

# 结语：评估是规格说明、是质量门控、是生产监控、也是改进的驱动力


和传统软件有本质不同，传统 QA 框架在这里失效；ADLC 提供了一套新的方法论，把生产数据变成飞轮的驱动力；这三类工程实践，无论是把评估跑起来，还是让数据持续流入评估，还是让系统架构可被评估，最终都指向同一件事。

**评估在一套成熟的智能体系统里同时扮演四个角色：**

| 规格说明 | 质量门控 | 生产监控 | 改进驱动力 |
| --- | --- | --- | --- |
| 基准数据集和指标体系，定义了什么叫"好"——这是整个开发工作的起点，不是终点 | 每次变更触发评估，不达阈值不上线，阻止回归悄悄进入生产 | 持续采样和漂移检测，让静默的质量衰退在影响用户之前就被发现 | 生产中的失败案例回流到评估集，为下一轮优化提供有据可查的方向 |

这四个角色不是分开的系统，而是同一套基础设施在不同阶段发挥的作用。越早建立这套能力，它的复利效应就越显著——因为每一次迭代都在让评估集更完整、让指标体系更精准、让整个飞轮转得更快。

**如需进一步了解 Evaluation-first 方法在实际场景中的落地方式，可参考本系列配套动手实验示例代码：**
**https://github.com/aws-samples/sample-eval-first-building-enterprise-agents-with-agentcore**

02

# 第二章：
# 评估企业级智能体：从原型验证到生产就绪

*智能体的难点不在于“能不能做到”，而在于“能不能每次都做到”。本文是企业生产级智能体开发部署指南系列的第 2 篇，给出一套可落地的智能体评估方法论——评什么、怎么评、如何从零构建。*

上一篇我们提出了一个判断：智能体与传统软件有本质不同——非确定性、Prompt 即源代码、依赖会自己动——因此传统 QA 框架在它身上系统性失效，需要一套新的开发生命周期 ADLC。在那个六环节的飞轮里，“定义‘好’”排在动手构建之前，**而 Evaluation是后续工程实践的重要基础**：没有它，团队不知道自己在哪里，也不知道改了之后有没有变好。

**上一篇结尾留下了三个问题,本篇就来回答它们。**

1. **智能体评估究竟要评什么维度?有哪些方法?**
2. **有哪些方法?**
3. **如何从零构建一套在企业规模下真正可用的评估体系?**



进入方法论之前，先把“为什么评估这么难”再向前推一步。几乎每个构建过智能体的团队都遇到过同一个场景：demo时一气呵成，一旦开始接入真实流量，它却开始“间歇失效”——同一类请求，十次里有一两次走错工具或把一个本该拒绝的操作执行了。对于智能体来说，**能力（capability）与一致性（reliability/consistency）并不是一回事**。能力回答“它能不能做到”，一致性回答“它每次都能做到吗”；前者靠一两个亮眼样例就能展示，后者只能靠成规模、反复的评估去逼近。智能体把多步推理、工具调用、外部状态写入耦合在一起，任何一环的随机性都会被链式放大——这正是上一篇所说“Demo和生产之间的鸿沟”在评估视角下的样子：一边是 Gartner 预测 2028 年约 33% 的企业软件将内置 agentic AI 的汹涌需求，一边是 MIT 调研中仅约 5% 的组织报告生成式 AI 项目取得高回报的稀薄成功率。把两者连起来的，恰恰是**能不能可信地判断“这个智能体是否真的交付了”**。评估，就是把“the agent ran（它跑了）”翻译成“the agent delivered, here is the evidence（它交付了，这是证据）”的那道工序。

本文沉淀我们与企业客户合作中的实践经验，把智能体评估从临时跑分推进到一套可工程化的方法论。围绕开头那三个问题，本文分为五个部分。

**第一部分**先点出三个最常见的评估误区，作为后文方法论的反向坐标。

**第二部分**给出由“三种评估粒度”与“三层证据权重”构成的两支柱框架，回答“评什么”与“分数有多大分量”，并把企业场景下的八类测量维度落到这个框架上。

**第三部分**讨论如何把评估嵌入研发流程——能力评估与回归评估的差异、生产漂移的监控、评估数据集与黄金集的构建，以及如何用约20个用例最小规模起步。

**第四部分**审视LLM-as-a-Judge的价值与边界——它的偏见与缓解技术，以及在多步智能体场景下的根本局限。第五部分介绍**Agent-based Evaluation（用智能体评智能体）**，并在结尾交代这套方法论与工程底座的关系，为下一篇工程化实践博客做铺垫。

# 三个常见的智能体评估误区

在给出框架之前，先点出三个最常见、也最容易让团队误判“已上线就绪”的坑。

**误区一：仅关注智能体准确率指标。**

把多步智能体压成一个“对/错”或一颗“质量星级”，会同时掩盖两类信息：质量很高但延迟/成本很差的情况，以及“答案对了但过程全错”的侥幸。对多步系统而言，最终输出的单一准确率远远不够。

**误区二：严格比对工具调用序列。**

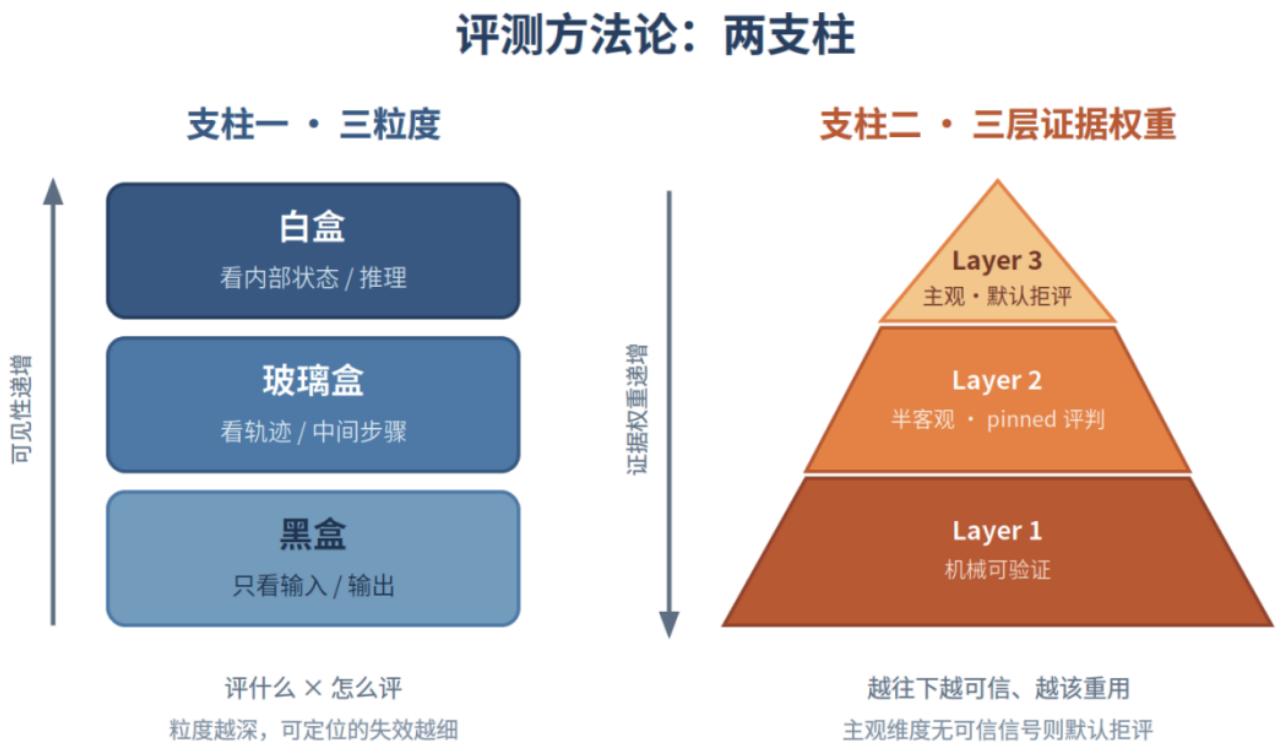
直觉上“轨迹对了才算对”，于是有人用预期工具调用序列做精确匹配。但这种做法极其脆弱：智能体换一个等价路径、调换两个无依赖步骤的顺序，就会被判失败。它测的是“像不像我写的脚本”，而不是“有没有把事办成”。

**误区三：先评估、后观测。**

很多团队急着搭评分流程，却没有先把生产里的执行轨迹（trace）沉淀下来。结果是：分数掉了，却不知道掉在哪一步、为什么掉。上一篇把“从第一天就埋好可观测性”列为三类工程实践中的数据管道一环，原因正在于此——正确的顺序是先有可观测性，再谈打分，没有trace的评估是盲测。

# 评估方法论框架：两根支柱

在与企业客户合作的实践中，我们沉淀出一套**两支柱 + 三类打分器**的评估方法论，分别回答“用多粗的镜头看”与“一个分数有多大分量”两个问题。**支柱一（三种评估粒度）** 决定评估的**粒度有多深**——从只看最终响应（黑盒），到看完整执行轨迹（玻璃盒），再到看单步细节（白盒）。**支柱二（三层证据权重）** 决定**每个分数有多大分量**——从机械可验证（Layer 1）、到半客观（Layer 2）、再到默认拒评（Layer 3）。两根支柱互相正交：同一个粒度上的指标可以来自不同证据层级，反之亦然——它们不是一一对应的两套层，而更像一个 3 × 3 的矩阵，用同一组指标既要选粒度、又要选证据强度。**三类打分器**（代码规则、LLM-as-a-Judge、人工）是把这个矩阵填满的工具，它们与三层证据权重天然对齐。下面先拆解两根支柱各自的结构，再回到这个矩阵看它们如何交叉。




## 支柱一：三种评估粒度（黑盒、玻璃盒、白盒）

对多步智能体，我们建议在三个粒度上同时观察：

- **黑盒（Black-Box）**——最终响应。只需输入、输出、上下文即可判定：相关性、完整性、简洁性、语气、事实正确性、输出结构是否合规。它适合端到端验收，也最贴近用户感知。
- **玻璃盒（Glass-Box）**——完整轨迹。看整段消息序列与工具调用，**精确定位推理在哪一步出错：**目标是否达成、工具选择与参数是否正确、轨迹是否高效、是否产生幻觉。
- **白盒（White-Box）**——单步。评单个步骤或单次工具调用的正确性，是定位与归因的最细一层。

三个粒度互补：黑盒告诉你“结果好不好”，玻璃盒与白盒告诉你“为什么、在哪儿”。这里有一条经验：**优先按结果（outcome）打分，而非死磕**路径；对多组件任务设计**部分得分（partial credit）**；轨迹分析用来归因，而不是用来做工具序列的精确匹配。

## 支柱二：三层证据权重框架

第二根支柱常被忽视，却最为关键。它的出发点是一个朴素的事实：**并非所有质量都同样可测，而一个分数会带来后果**（决定是否上线、是否付款、是否放行），因此**每个分数携带的“证据权重（evidentiary weight）”必须被显式标明**。据此，我们把所有指标按“能被多严谨地验证”分到三层，每个指标只归一层：

- **第 1 层——机械可验证（Mechanically verifiable）**。纯代码、零歧义判定：schema、格式、延迟、成本。结果确定，任何一方都能复算——这是**最强的证据**，在审计与争议场景下可被辩护。
- **第 2 层——半客观（Semi-objective）**。由模型在受控条件下稳定打分：相关性、正确性、忠实度、一致性。它非确定、不可逐位复算；只有在固定评判器（**pinned evaluator：锁定模型、prompt、temperature、seed**）下，其分数才作数。
- **第 3 层——主观（Subjective）**。没有任何评判器能稳定打分的维度（例如“这个够不够有创意?”）。**默认拒评（refused by default）**——在 rubric 协商阶段就把它标出来，而不是强行给一个假装客观的分。

这层框架给出一个清晰的心智模型：**一个分数，只能在它所在层支持的严谨度上被当作证据使用。**由此也能推出一组该刻进团队习惯的反模式：不要把质量压成单一星级合成分；不要给用户展示裸小数（那是假精度，宜用字母档加命名置信度）；不要跨任务类型套用统一阈值（0.75 在创意题和严格 schema 上意义完全不同）；不要把延迟、成本藏起来，它们是一等的业务指标。

## 打分器的选择

支撑这两根支柱的，是三类 **打分器（grader）** 的组合使用：

| 打分器 | 优点 | 缺点 |
| --- | --- | --- |
| 代码规则（Code-based） | 快、便宜、客观、可复现 | 脆，只覆盖可程序化判定的部分 |
| 模型 / LLM-as-a-Judge | 灵活、可扩展、能捕捉细微语义 | 非确定，需对照人工校准 |
| 人工（Human） | 金标准 | 昂贵、慢、难规模化 |

它们与三层框架天然对齐：第 1 层用代码规则，第 2 层用经校准的 LLM-as-a-Judge，人工则负责抽样校准与最终仲裁。

在企业落地时，这三类打分器有明确的**优先级与分工：**

**程序化、可验证的检查优先执行。**

低成本、客观、可信——凡是能写成代码断言的（schema、格式、敏感词、延迟、成本、关键参数），绝不交给评判模型（judge model）。这既是省钱，更是把最强证据放在最前面。

**LLM-as-a-Judge 只管主观维度，且评分标准（rubric）必须由 SME 校准。**

judge 模型人人可买，但“什么算好”的 rubric 要由懂业务的领域专家写出来、再对照人工标注验证一致率——否则你得到的是一个自信但不对齐业务的打分器。

**SME/人工审核是企业的核心优势，要用在刀刃上：**

标注黄金集，以及每次发布前的抽检。外部模型与评估平台是人人可得的通用件，唯独“懂你业务的人的判断”买不到——这正是企业评估体系里最稀缺、也最该被制度化沉淀的资源。

把两支柱与打分器合起来看，就是下面这张矩阵：行是三种粒度、列是三层证据权重，每个格子放一个示例指标、由对应的打分器填充。它直观地说明了两支柱如何交叉、空格（如“白盒 × 主观”）为何通常不评估，以及人工评估如何横跨各层。


两支柱相互正交:同一格的指标先选粒度、再选证据层;空格表示该组合通常不评测。人工评估横跨各层,负责校准与最终仲裁。

# 企业到底要测什么：八类测量维度

矩阵搭好了，往里填什么？这里先校准一个定位：**企业评估是风险管理加质量保障（QA）**，而**不是研究**。它的目标不是刷benchmark或发论文，而是控制上线风险、守住质量底线；“好”由领域专家（SME，Subject Matter Expert）定义，基准由真实业务数据构成。在这个定位下，企业智能体通常需要覆盖八类测量维度——每一类都能在“粒度 × 证据权重”矩阵上找到自己的格子：

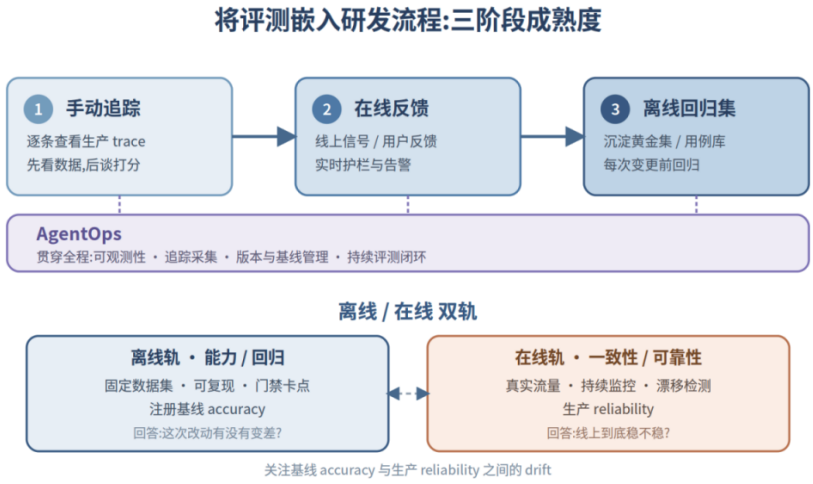
| 测量维度 | 它回答的问题 | 典型粒度 | 典型证据层 |
| --- | --- | --- | --- |
| 任务/目标完成率 | 用户的目标这次达成了吗? | 黑盒 | 第 2 层(有 ground truth 时可升第 1 层) |
| 工具与动作正确性 | 选对工具了吗?参数填准了吗? | 玻璃盒/白盒 | 第 1 层(有预期轨迹)或第 2 层 |
| 安全/PII/信息泄露 | 有没有泄露不该说的? | 黑盒+玻璃盒 | 第 1 层规则扫描 + 第 2 层语义判定 |
| 成本与延迟 | 跑一次花多少钱、多少秒? | 任意粒度 | 第 1 层(纯机械可验证) |
| 忠实性/事实依据 | 回答有出处吗?编造了吗? | 黑盒 | 第 2 层(需固定评判器) |
| 策略与合规 | 守住业务规则与监管口径了吗? | 玻璃盒 | 第 1 层规则 + 第 2 层判定 |
| 品牌语调与风格 | 说话像"我们公司的人"吗? | 黑盒 | 第 2 层偏第 3 层(SME 校准 rubric,部分子维度拒评) |
| 升级处理的恰当性 | 该转人工的时候转了吗? | 玻璃盒 | 第 2 层 |

这张表传达的方法论是:**选指标,就是在矩阵上为业务挑格子,而不是把所有指标全开**。一个纯问答智能体不需要盯工具正确性,一个不直接面客的内部智能体对品牌语调的要求也可以放低;但凡选中的格子,就要按它所在证据层的严谨度去执行——成本与延迟永远用第 1 层代码算,忠实性永远用固定评判器,品牌语调里那些连 SME 都说不清的子维度,明确拒评。

最后回到开头提到的"一致性"。智能体的随机性意味着"跑一次过了"不等于"可靠",因此需要区分两个指标:**pass@k** 是 k 次里**至少一次**成功的概率(适合"一次成功就够"的场景);**pass^k** 是 k 次全部成功的概率(适合一致性至关重要的智能体)。

# 评估的工程化落地:从流程嵌入到数据集积累

有了框架,还要让它在研发流程里"转起来",而不是上线前临时跑一轮。下图展示了从手动追踪到离线/在线双轨的演进路径。



## 流程与漂移

**能力评估与回归评估是两件不同的事。**

能力评估起点分数低，是“给团队一座要爬的山”，用来衡量并推动能力提升；回归评估应维持接近 **100%** 的通过率，用来守住已有能力、防止退化。成熟的能力评估达标后，就 **“毕业”沉淀为回归套件**，进入 CI 持续运行。

**警惕基线与生产之间的漂移（drift）。**

上一篇提到的“静默漂移”——模型隐式更新、外部依赖变化让质量指标悄悄变差——在这里可以被精确表述为一个尤其值得显式管理的现象：开发期注册的基线 accuracy（在 **dev benchmark** 上声称的）不等于生产里 **30** 天的 **reliability（实际可接受的交付比例）**。两者会因 agent drift、依赖模型被弃用、prompt 随时间衰减而背离。“92% 的 reliability”意味着每跑 100 个任务约有 8 次失败——这正是 pass^k 论点在生产语境下的另一种表达。基线分数不是终点，它只是一个会漂移、需要被持续重测的量。

**三阶段成熟度加 AgentOps。**

上一篇把“从第一天就埋好可观测性”定为工程实践的起点，这里进一步给出它通向完整评估体系的成熟度路径：第一步，**先做 tracing，不急于打分**（“先追踪，后打分”）；第二步，**接入在线反馈**（如点赞/点踩等用户信号）；第三步，沉淀离线回归集（用自动化流水线防止回归）。在这之上挂一个 **AgentOps 组件**，在生产中持续监控智能体表现、捕捉异常与漂移。

**离线与在线双轨并行。**

开发期（离线）跑回归集回放、能力评估、CI 门禁；部署后（在线）做实时打分与采样、收集用户反馈、用 AgentOps 监控漂移。在线打分可用**采样率控成本**（例如只对一部分流量触发评判）。

## 数据集与起步

流程与漂移解决“什么时候评、谁来盯”，但评估真正能不能产生信号，取决于**数据集质量与起步姿势**——这两件常被低估的事。

**如何构建评估数据集。**构建评估集要分清两个正交的维度：用例**从哪来**（覆盖度），以及哪些用例的标注**够可信**（可作基准）。

## 维度一 · 用例来源（覆盖度）

- **生产 trace 沉淀（真实分布，最有价值）**。不要全量灌进来，要按 intent、工具、**用户分群分层采样**，保留完整 trace（不只是输入输出），并在入库前做 PII 脱敏。它的优势是反映真实分布，代价是会带着真实流量的偏斜——长尾不会自动出现。
- **合成增强（覆盖长尾与对抗 case）**。关键是**以真实失败 trace 为种子**做扰动与变异（改时态、加噪声、注入边界值、对抗 prompt 等），而不是凭空让 LLM “想象”用例。纯凭空合成最大的风险是看起来覆盖了边界 case，实际却生成了一套与生产无关的另一种分布——仪表盘绿光满满，真实问题却没碰到。

## 维度二 · 黄金集（golden set）：可信的标注基准

- 从上述用例中筛出一个**标注高度可信、正确**的子集，作为校准与仲裁的基准。它的标注来源不限——既可以是人工精标，也可以是带明确 ground truth 的样本——**重点在“可信与正确”**，而非由谁标注。实践要点：多人标注并算出**标注一致性（IAA）**（不一致的样本本身就是 rubric 不清的信号）；**给黄金集打版本号**，任何修改可追溯；并**留一份从不进入开发循环的 holdout**，用来体检 LLM-as-a-Judge 的真实校准度。
- 值得把黄金集的地位再抬高一档：**它是企业的评估知识产权（evaluation IP）**。真实生产数据、SME 标注、覆盖常见+边缘+合规+升级四类场景、生产失败案例持续回流——模型会换代、框架会迁移，唯独这套资产沉淀下来不走，而且越滚越值钱。由此也引出一条工具选型原则：评估平台可以买，评**估内容必须自主掌控**。跑分流水线、dashboard、judge 托管这些是通用件，谁家的都能用；但黄金集与 rubric 是业务件，外包了它们，等于把“什么算好”的定义权交给了别人。
- 实战中比例不是死规矩，起手可以参考 约 50% 生产采样 + 30% 合成 + 20% 黄金，之后每月从生产增量补 10–20%。还要警惕三个常见失败模式：**数据泄漏**（黄金集被反复看见、渐渐被过拟合）、**黄金集冻结**（几个月不更新，生产分布已经漂走，而黄金集还守着旧世界）、**合成掩盖真实**（只跑合成数据时，生产里真正发生的失败模式可能根本没被覆盖）。

**如何最小规模起步**。不必一开始就追求大而全。先从 约 **20 个用例**起步，但这 **20 条不是随手抓 20 条，而是“小而有代表性”**：覆盖主要 intent，已知好、已知坏、**模糊三档都要有**，happy path 之外至少塞 3–5 个 edge 或对抗 case。

拿到这 20 条之后，**先看数据、再打分**：开一张表，列固定为 输入 / trace / 预期 / 实际 / 失败模式标签，把 trace 逐条读一遍，**先把失败聚成 2–3 个簇**（例如“参数填错”“漏澄清”“提前终止”），再把聚出来的簇翻译成 rubric 条目和打分维度。这就是 error analysis 的核心动作——先观测、后度量，远比一上来就堆指标更快收敛。

之后的升级路径也大致可预期：**20 → 100** 条时，失败模式的簇会逐渐稳定，rubric 收敛；**100 → 500** 条时，你已经能做有统计意义的回归对比，并在黄金集上对照人工标签校准 LLM-as-a-Judge；**500 条以上**，就进入“按比例从生产分层采样、定期增量补”的稳态。两个早期最常见的反模式正好与之相反——**rubric** 先于数据（打分维度全是凭空想的，真实失败根本不在那几条上）、过早上 **LLM-judge**（连失败长什么样都没看过，校准的“基准”是空的）。

# LLM-as-a-Judge 的价值与边界

上面多次提到用 LLM 当评判器（LLM-as-a-Judge）。它已是成熟范式，提供**可扩展、低成本、相对一致**的评估，是人工或专家评估的有力补充。但它有清晰的边界，裸用会出问题。

经校准后，LLM-as-a-Judge 与人类的一致性可以做得不错（在 MT-Bench 等设置下，大语言模型可以达到与人类之间相当的水平），但这强依赖评判模型强度与领域，不**能泛化为“任意评判器都具备的属性”**。更要紧的是，LLM 评判存在可测量的系统性**偏见**，例如：

1. **位置偏见：** 交换两个答案的顺序，大模型判决可能会翻转。
2. **冗长偏见：** 更长、更像“列清单”的回答容易被高估。
3. **权威偏见：** 对于“置信度”较高的内容，例如“伪造参考文献”，多数评判表现不优于随机。

对应的**缓解技术**也已成熟：**双向打分**（对（A, B）与（B, A）各判一次，不一致就记平局）可直接抵消位置偏见；**多评判陪审团（Panel of LLM evaluators, PoLL）用不同模型家族的多个较小评判投票，既降低单一模型家族的内生偏见，成本还约为单个大评判的 1/7 到 1/8；以及对照人工标签做校准**，以“评判与人类的一致率达到人类之间的一致率”为可用门槛。

**LLM-as-a-Judge 必须做偏见缓解与人工校准，不能“直接”使用。**但即便缓解到位，单轮评判仍只是“看一眼最终答案打个分”——它看不到智能体是怎么一步步走到这里的。当被评对象是一个会推理、会调工具、会产出复杂中间制品的智能体时，我们需要一种能“陪着它走完全程”的评判者，因此我们更需要 Agent-based Evaluation。

# Agent-based Evaluation：将专家级评审规模化

## 为什么需要 Agent-based Evaluation

单轮 LLM-as-a-Judge **不能进行完善的全局评测。**但智能体的价值恰恰分布在过程里：多步推理是否自洽、每一次工具调用的选择与参数是否正确、中间产物（代码、检索结果、草稿）是否站得住。只看终态，会把这些信息全部丢掉；而要人工逐步审查整段轨迹，成本又高到无法规模化。

**Agent-based Evaluation（用智能体评智能体）**正是为此而生：让评估者本身也是一个具备 agentic 能力的系统——它能读取被评智能体的**整个解题轨迹**（文件、中间步骤、工具调用），必要时自己调用工具去验证（运行代码、检索核对、把 rubric 分解成可判定的子项），并在过程中给出**过程级评判与中间反馈**，而不只是末尾打一个分。它的本质，是**把原本只能由专家完成的"逐步审阅"规模化**：专家会读你的推理链、会去核对你引用的事实、会指出"第三步这里就错了"——Agent-based Evaluation 把这种深度审查自动化、批量化。

## 与单轮评判和人工评估互补

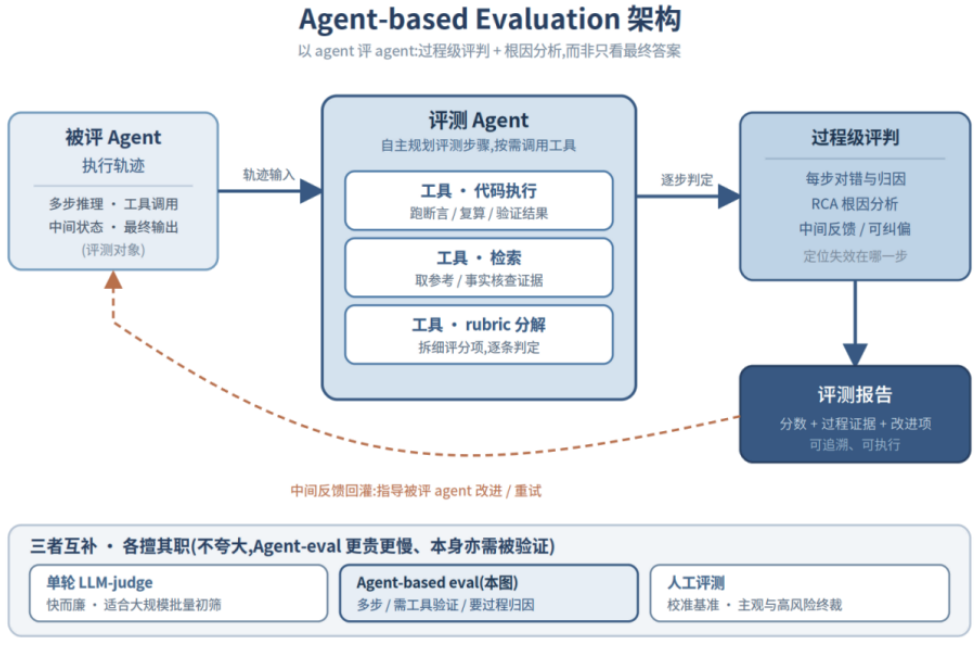
**要把它放在正确的位置：它不是来取代单轮评判和人工评估的，而是与两者互补**——单轮评判快而便宜，适合大批量黑盒打分；人工是金标准与最终仲裁；Agent-based Evaluation 补的是中间那段"需要过程级、可验证、可解释，但又请不起人工逐步标注"的深水区。

同样重要的是**不要夸大它**。这条工具链有三个必须正视的代价：

1. **更贵、更慢。**评估智能体要读轨迹、调工具、做多轮推理，延迟与算力成本显著高于单轮评判。它适合开发与预生产阶段的深度评估，而不是每条生产流量的实时门禁。
2. **它本身需要被验证。**评估智能体也是用 LLM 搭建的，它的判定**同样需要一个"评判器的 benchmark"去校准**——否则低成本的自动评分会悄悄引入系统性误差。
3. **它不会自动消除偏见。**评估智能体同样继承位置、冗长、自我增强等偏见（见上一节），仍需对抗式校准。"用 Agent 评"不等于"客观"，只是把评判的粒度做深了。

## 参考架构与最佳实践

下图展示了一个 Agent-based Evaluation 的参考架构，以及它与单轮评判、人工评估的互补关系。



一个可用的 Agent-based Evaluation，大致由这几块构成：

1. **轨迹输入：**把被评智能体的完整 run（输入、每一步、工具调用与返回、最终产物）结构化地喂进来。
2. **带工具的评估智能体：**评估者本身配备工具——代码执行（验证产物能否跑通）、检索（核对事实与引用）、rubric 分解（把“好不好”拆成一组可独立判定的子标准）。
3. **过程级评判加根因分析（RCA）加中间反馈：**不只给终评，而是定位“在哪一步、为什么失败”，并给出可操作的反馈。
4. **面向多受众的报告：**同一个 0 到 1 的分数，对不同人意味着不同的东西。**裸分数不是决策。**给选型者看朴素直白的质量、一致性、延迟、成本；给开发者看逐任务加运营指标（改什么）；给运营者看逐任务风险档（Safe、Marginal、Risky，决定提交、重试或放弃）。每份逐任务报告可固定为四段：一句话裁决、逐维度 scorecard（用整数百分比而非裸小数）、做对了什么与失败在哪、建议动作。

最佳实践上，把它**叠在三层证据权重框架之上**：机械可验证的部分（代码能否跑通、schema 是否合规）仍归第 1 层，用工具确定性判定；需要语义判断的部分归。第 2 层，用固定评判器加 rubric 分解；真正主观的维度照旧拒评。这样，即便评判者是个智能体，每个分数的证据权重依然清晰。

## 从方法论到工程底座

**方法论本身是客观、通用的，但这套复杂度大多属于"通用底座：**收 trace、跑离线/在线评估、保证可复现执行、武装评估智能体、运行时护栏——这些与具体业务无关、人人都要的部分，已经可以交给托管平台承接，不必团队从零搭建。以 **Amazon Bedrock AgentCore** 为例：它的 Observability 以 OpenTelemetry 兼容格式自动输出 trace 与 span，正是“先 tracing，不急打分”那一步的工程化实现；它的 Evaluations 在 sessions、traces、spans 三个层级上工作，恰好与本文的**三种评估粒度一一**对应——session 对应黑盒，trace 对应玻璃盒，span 对应白盒。方法论里的每个环节，在工程底座上都有成型的承接位。

底座既然可以托管，团队真正**该集中投入的，是与业务强相关、无法外包的那部分**：智能体在哪类请求上算“交付成功”、哪些是必须守住的领域特定 metrics（合规口径、行业术语准确性、关键工具的参数正确性……）、黄金集里那批最能代表业务的真实用例——也就是上文所说的**评估知识产权**。把通用复杂度交给底座、把精力留给业务判断——这正是把方法论落到生产、且能持续跑下去的关键。

# 最佳实践清单

**把全文浓缩成一页，供要动手的团队参考：**

1. **先接入 tracing，再谈打分**。没有可观测性，一切评估都是盲测。
2. **三个粒度都覆盖：**黑盒看结果，玻璃盒与白盒做归因。
3. **按结果打分、给部分得分**，避免死磕工具调用序列的精确匹配。
4. **用三层证据权重框架管理每个分数：**第 1 层机械可验证、第 2 层固定评判器、第 3 层默认拒评，并据此显式标明分数的证据权重。
5. **三类打分器混用：**代码规则加经校准的 LLM-as-a-Judge 加人工抽样仲裁。
6. **区分能力评估（低起点、要爬坡）**与回归评估（近 100%、守底线）；能力评估达标后毕业进入 CI。
7. **对一致性敏感的客服或交易类智能体，用 pass^k**，并显式监控基线 accuracy 与生产 reliability 之间的漂移。
8. **LLM-as-a-Judge 必做偏见缓解与人工校准：**双向打分、参考引导、PoLL 陪审团、人工金标准。
9. **离线与在线双轨：**离线跑回归集做门禁，在线用 AgentOps 监控漂移、用采样率控成本。
10. **深水区上 Agent-based Evaluation：**用带工具的评估智能体做过程级评判与 RCA，但记得它更贵更慢、本身需要被校准、不会自动消除偏见。
11. **从约 20 个用例起步，先看数据再打分；**引用任何基准数字都带上模型与时间点标注。
12. **裸分数不是决策：**给选型、开发、运营三类受众不同的报告视图。
13. **把黄金集当作评估知识产权经营：**真实生产数据加 SME 标注，覆盖常见、边缘、合规、升级四类场景；评估平台可以买，评估内容必须自主掌控。

# 结论

本文回答了上一篇结尾留下的三个问题。**评什么维度**——三种评估粒度乘三层证据权重的矩阵，加上企业场景的八类测量维度，让“选指标”变成在矩阵上为业务挑格子。**用什么方法**——三类打分器按程序化优先、SME 校准 LLM-as-a-Judge、人工抽检守住黄金集的优先级分工；深水区用带工具的评估智能体做过程级评判与根因分析（Agent-based Evaluation），把专家逐步审阅的能力规模化。**怎么从零构建**——先 tracing 后打分，从约 20 个用例做 error analysis 起步，沿 20 → 100 → 500 的路径把能力评估毕业成回归套件，并把黄金集当作评估知识产权持续经营。

**能力（capability）与一致性（reliability/consistency）是两件不同的事。**在模型能力被快速商品化的当下，真正构成长期差异化的，是一套贴合业务、沉淀真实生产 trace、校准过评判器、并显式管理证据权重与漂移的评估体系。它能持续回答一个问题：“这个智能体这次是否真的交付了，证据是什么？”

**如果你希望将本文的方法论落到工程实践中，我们建议如下起步路径：**

1. **先建立可观测性。**用托管服务或基于 OpenTelemetry 自建，把智能体执行轨迹结构化沉淀下来，作为后续评估的数据基础。
2. **按三层证据权重框架分配打分器。**第 1 层用代码规则，第 2 层用经校准的 LLM-as-a-Judge，第 3 层默认拒评。
3. **从约 20 个有代表性的用例起步，做** error analysis。先看数据、再确定 rubric 与打分维度，逐步从能力评估毕业为回归套件并接入 CI。
4. **在深水区引入 Agent-based Evaluation。**用带工具的评估智能体做过程级评判与根因分析，并对其判定本身进行人工校准。

方法论到这里就完整了，但它还只是图纸。**下一篇，我们看亚马逊云科技是怎么把这张图纸盖成楼的：**Amazon 内部数千个生产级智能体沉淀出的三层评估库（Foundation Model /智能体 Components / End-to-End）、把评估自动化的 Trace-driven 四步工作流，以及购物助手、客服、卖家助手三个真实案例——从工具使用评估、意图检测评估到多智能体协作评估。

**如需进一步了解 Evaluation-first 方法在实际场景中的落地方式，可参考本系列配套动手实验示例代码：**
**https://github.com/aws-samples/sample-eval-first-building-enterprise-agents-with-agentcore**

03

# 第三章：
# 如何在亚马逊云科技上构建企业级智能体

前面两部分我们讨论了智能体的开发生命周期，以及评估为什么是一个全新的问题——它既不同于传统软件的单元测试（输入到输出不再是确定性映射），也不同于大模型 benchmark。单模型 benchmark 评的是一个 LLM 在孤立 prompt 上的表现，而智能体是一个会自主追逐目标、跨多轮交互做多步推理、调用工具、动态决策的**完整系统**。传统的 LLM 评估方法把这样的系统当成黑盒，只看最终输出对不对——它能告诉你"结果错了"，却无法告诉你"为什么错"，更无法定位是模型选错了、意图理解偏了、工具调错了、还是记忆检索丢了上下文。

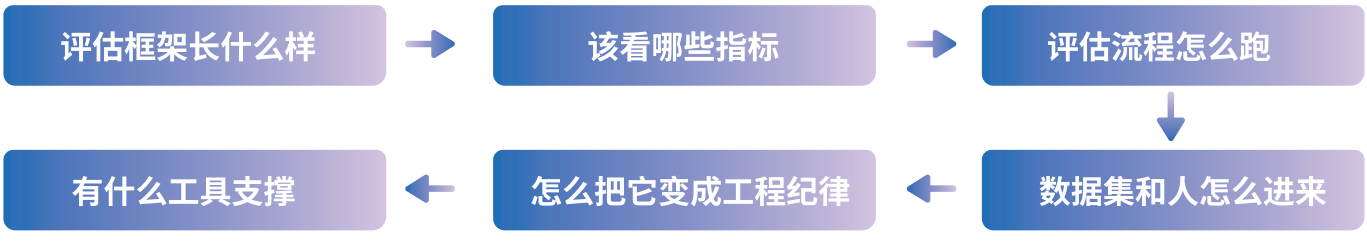
那么，当评估真正要落到一个有成百上千个生产级智能体的组织里时，它应该长什么样？这一节我们不谈空泛原则，而是看亚马逊云科技在 Amazon 内部沉淀出来的一套系统化方案。自 2025 年起，Amazon 各业务组织内部已经构建了**数千个智能体**，覆盖购物、客服、卖家支持等核心场景，从早期实验走向了生产级规模化部署。规模一旦上来，"靠人肉看几条 case 拍脑袋"的评估方式立刻崩溃。

更麻烦的是，市面上虽然有不少专用评估工具，但开发者要在它们之间来回切换、再把结果手工汇总，成本极高；而 Strands、LangChain、LangGraph 这些框架虽各自内置了评估模块，却把人锁死在单一框架里。亚马逊云科技由此得出一个明确取向——开发者要的是一套框架无关（framework-agnostic）的评估方法：

"Builders want a framework-agnostic evaluation approach rather than being locked into methods within a single framework." （开发者想要的是一套框架无关的评估方法，而不是被锁死在某个框架自带的评估手段里。）

而且，生产级智能体还有一项传统评估很少触及的能力必须被衡量——**自反思与错误恢复**。一个健壮的智能体必须能识别、分类并从各类失败中恢复：推理模型给出的不当规划、无效的工具调用、格式错误的参数、意料之外的工具返回格式、认证失败、记忆检索错误……评估框架要能度量智能体在遭遇这些异常后，是否还能维持多轮交互的连贯性。再加上生产环境中智能体会随时间出现**能力衰退（agent decay）**，这就要求评估具备近实时的问题检测、告警与处置能力，并辅以 HITL 审计来兜住可靠性。

**本章的主线是六个递进的问题：**



# 评估框架全景：自动化工作流 + 三层评估库

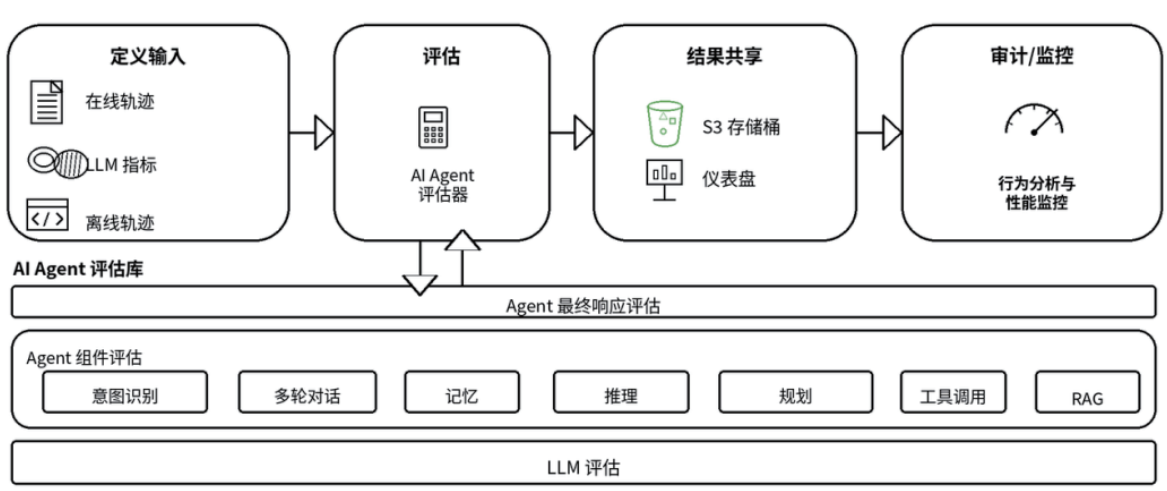
亚马逊云科技提出了一套**整体性 Agentic AI 评估框架（Holistic Agentic AI Evaluation Framework）**，由两大组件构成：

**自动化评估工作流**（Automated Evaluation Workflow）——把"从拿到 trace 到产出结论"的链路标准化、自动化，让它能跨各种智能体实现复用；

**智能体评估库**（Agent Evaluation Library）——**沉淀系统化的度量与指标（其中一个核心子集已产品化为 AgentCore Evaluations 的14个内置评估器(built-in evaluators)，其余面向 Amazon 内部异构场景的扩展指标可通过自定义评估器实现。）。**

工作流我们留到后面章节讲，这里先看评估库的核心设计——**三层架构**。它回答了一个关键问题：评估到底要评智能体的哪些部分？

| 层级 | 评估对象 | 它回答的问题 |
|---|---|---|
| **底层** Foundation Model | 用对多个基础模型做 benchmark | 选哪个模型来驱动 Agent?不同模型对整体质量和延迟有什么影响? |
| **中层** Agent Components | Agent 内部组件：意图识别、多轮对话、记忆、LLM 推理与规划、工具使用 | Agent 是否正确理解了意图?CoT 推理如何驱动规划?工具选择与执行是否对齐了计划?计划有没有成功完成? |
| **上层** End-to-End | Agent 终态：最终响应、任务完成度，以及整体的责任与安全、成本、客户体验影响 | Agent 最终是否达成了用例定义的目标?是否符合责任标准?成本可接受吗? |




这张图**（配图 1：智能体评估库三层架构）**值得反复看，因为它点出了智能体评估区别于传统 LLM 评估的核心差异：

**传统 LLM 评估只看上层。**

它把整个智能体当黑盒，丢进去一个问题、接住一个答案，再判断答案好不好。这种方式在智能体出错时几乎无能为力——我们知道结果错了，但定位不到根因。

**中层才是智能体评估真正的主战场。**

一次任务要经过"理解意图 → CoT 规划 → 选工具 → 填参数 → 多轮交互 → 综合记忆"这一长串步骤，任何一环断裂都会导致最终失败。只有把评估拆到组件粒度，才能在出错时快速定位根因，也才能在改一个 prompt、换一个模型后，知道"到底是哪一环变好了或变坏了"。

底层则服务于选型：同样一套智能体逻辑，换 Claude Opus 还是 Haiku，质量和延迟会怎么变？这一层把模型作为可量化的变量。

## 关键指标体系：按智能体形态选指标，而不是堆指标

有了三层框架，接下来是具体指标。亚马逊云科技基于 AgentCore Evaluations 的内置评估器，又针对 Amazon 内部异构场景的复杂度做了扩展，沉淀出一套相当完整的指标清单。这里的核心主张是：**指标要按智能体的形态来选，而不是无脑全开。**一个纯问答智能体不需要多智能体协作指标，一个工具密集型智能体则必须重点盯工具使用指标。

下面按智能体形态分组速查：

**最终响应质量（Final Response Quality）**

- Correctness：响应的事实准确性
- Faithfulness：是否与对话历史保持一致
- Helpfulness：是否真正帮用户推进了目标
- Response Relevance：是否针对具体请求
- Conciseness：表达效率，恰当简洁又不丢关键信息

**任务完成（Task Completion）**

- Goal Success：是否在一次会话内完成了所有用户目标
- Goal Accuracy：与 ground truth 的对比

**工具使用（Tool Use）——工具密集型智能体的命脉**

- Tool Selection Accuracy：是否选对了工具
- Tool Parameter Accuracy：调用时是否正确使用了上下文信息填参
- Tool Call Error Rate：调用失败的频率
- Multi-turn Function Calling Accuracy：多个工具是否按正确顺序被调用

**记忆（Memory）**

- Context Retrieval：从记忆中检索相关上下文的准确性，需在 precision 和 recall 间平衡

**多轮对话（Multi-turn）**

- Topic Adherence Classification：对话是否守在预定义的领域和话题内
- Topic Adherence Refusal：对超出范围的话题，智能体是否正确拒答

**推理（Reasoning）**

- Grounding Accuracy：是否理解任务、选对工具，CoT 是否对齐了上下文和工具返回的数据
- Faithfulness Score：推理过程的逻辑一致性
- Context Score：每一步是否上下文有据

**责任与安全（Responsibility & Safety）**

- Hallucination：输出是否与已知知识、可验证数据、逻辑推断对齐
- Toxicity：是否含有害、冒犯、贬损内容
- Harmfulness：是否含侮辱、仇恨言论、暴力、不当内容、刻板印象

**多智能体系统（Multi-Agent System）——前瞻方向**

- Planning Score：子任务分配是否成功
- Communication Score：智能体间为完成子任务的通信效率
- Collaboration Success Rate：子任务成功完成的比例

到这里，指标基本都是"技术指标"。但仅有技术指标不够——一个工具选得 100% 准、延迟极低的智能体，如果它推动的业务决策本身是错的，对企业就毫无价值。这正是亚马逊云科技 Prescriptive Guidance 提出决策为先（Decision-First）视角的原因：

传统自动化关注"流程效率"——把预设脚本跑得更快更可靠；而 Agentic AI 是"决策为先"——智能体评估上下文、权衡选项、实时调整行为。成功的衡量标准，不再只是任务是否完成，而是**决策在多大程度上对齐了意图、政策和不断变化的条件**。

由此，指标体系顶上要补一组**决策为先 KPI**：

| 决策为先 KPI | 含义 |
|---|---|
| **决策质量 Decision Quality** | Agent 的响应在多大程度上贴合了具体用户和场景？是否做出了对齐业务目标的细致决策？ |
| **Time-to-Action 响应时效** | Agent 评估情况并做出反应的速度，延迟是否低到让人感觉"自适应、像人"？ |
| **认知卸载 Cognitive Offload** | Agent 替人类承担了多少手动分析、分流、例行决策？是真减负，还是只是把工作转移了？ |

**技术指标回答"智能体做得对不对"，决策为先 KPI 回答"智能体做得值不值"。两者必须同时出现在评估面板上。**



**AgentCore Evaluations 内置一组评估器，按粒度分级，且含两种打分方式（LLM-as-a-Judge + 程序化）：**

- **会话级** · LLM（1 个）：Goal success rate
- **会话级** · **程序化/无** LLM（3 个，ground truth **轨迹匹配**）：TrajectoryExactOrderMatch、TrajectoryInOrderMatch、TrajectoryAnyOrderMatch
- Trace **级** · LLM（11 个）：Coherence、Conciseness、Context relevance、Correctness、Faithfulness、Harmfulness、Helpfulness、Instruction following、Refusal、Response relevance、Stereotyping
- **工具级** · LLM（2 个）：Tool selection accuracy、Tool parameter accuracy 其余指标（Topic Adherence、Context Retrieval、Multi-turn Function Calling Accuracy、Planning/Communication/Collaboration Score 等）是 Amazon 内部框架的扩展，需用自定义评估器落地。

**AgentCore 自定义评估器分两类：**

1. LLM-as-a-judge（自定义 judge 模型、指令、打分 schema）；
2. Code-based（用你自己的亚马逊云科技 Lambda 函数做确定性检查、正则、外部 API 调用或业务规则，不依赖 LLM judge）。

# Trace-driven 评估工作流：四步把评估自动化

指标定好了，怎么把它跑起来？亚马逊云科技的做法是把整个评估流程做成一条 trace 驱动的自动化流水线：

1

### 第一步——定义评估输入

评估的输入主要来自 Trace 和 Span，它们记录了模型调用、工具调用、推理步骤等关键过程，是后续评估的基础数据。不同评估模式的输入方式略有不同：On-demand evaluation 通常按指定的 span ID 或 trace ID 发起评估；Batch evaluation 面向一批历史会话，可从 CloudWatch Logs 中读取会话记录；Online evaluation 则按采样率或过滤条件，从生产流量中持续抽样评估。

### 第二步——调用评估库

2

评估库针对 trace 自动生成默认指标，也支持挂载用户自定义指标。

3

### 第三步——结果分发

评估结果连同 trace 写入 Amazon S3，或在 dashboard 上做 trace 可观测性与评估结果的可视化。

### 第四步——审计与处置

通过性能审计与监控分析结果；开发者可自定义规则，在智能体性能下降时触发告**警并采取行动**；还可用 HITL 机制**周期性地人工抽审 trace 子集和评估结果**，持续保障智能体质量。

4

这条流水线背后是一套三层 **Observability 策略**（与 AgentCore Best Practices 一致），它服务两类不同角色：

- **开发期 / 开发者：**trace-level 调试。用户报告异常时，调出那一条具体 trace，逐步还原智能体究竟做了什么——为什么幻觉、哪个 prompt 版本更好、延迟卡在哪一步。
- **生产期 / 平台团队：**CloudWatch Generative AI Observability Dashboards 做全局监控，用于治理与成本归因——哪个团队花了多少、哪个智能体在拉高成本、某次事故里到底发生了什么。

- **跟踪的核心信号：**token 用量、延迟分位（P50/P95）、错误率、工具调用模式。

值得强调的是，这套体系是开放的：AgentCore 以标准 OpenTelemetry(OTEL) 格式发出 telemetry，可对接任何兼容 OTEL 的可观测平台；官方明确支持 OpenInference、OpenLLMetry、OpenLit、Traceloop 等 instrumentation 库。

"You can't improve what you can't measure. Set up your measurement infrastructure before you need it."（你无法改进你无法度量的东西。在真正需要之前，就把度量基础设施搭好。）

## 评估数据集与 HITL：评估质量的上限由它们决定

再好的工作流，喂进去的数据集不行，结论也不可信。亚马逊云科技的经验里，构建评估数据集有三个反复被强调的要点：

- **同一查询的多种说法**——真实用户不会像 API 文档那样标准地提问，"我还剩几天假"和"年假余额"问的是同一件事；
- **应当拒答 / 应当升级的边缘情况**——比如 HR智能体遇到"我的奖金为什么比同事少"应该升级到人工，而不是硬答；
- **模糊查询**——一个问题有多种合理解释时，智能体该如何处理。

  数据集从哪来?Amazon 的两条实战路径很有借鉴价值：

- **从历史交互日志合成（购物助手案例）：**以真实的历史 API 调用日志为基础构建回归测试集，关键标签经规则校验 / 人工抽检确认；LLM 用于样本扩增与边界 case 生成。真实流量分布是最贴近生产的 ground truth 来源；
- **用 LLM 模拟器生成虚拟用户（客服案例）：**让大模型扮演多样化的虚拟客户 persona，批量生成贴近真实分布、覆盖长尾的交互——这是低成本扩大评估覆盖面的标配手段。

  而 HITL 的真正价值，常被误解。它不是"自动评估兜底的人肉补丁"，它的核心作用是校准评估器本身，尤其在高风险决策场景里不可或缺。HITL 提供：

  智能体推理链的评估、多步工作流连贯性的核查、智能体行为与业务需求对齐的判断；

- 为构建黄金测试集提供 ground truth **标签**
- **校准自动评估器里的** LLM-as-a-Judge，让它对齐人类偏好——这一步直接决定了你的自动化指标到底可不可信。

换句话说，HITL 是让自动化评估"可被信任"的那把尺子。

# 工程纪律：把评估嵌入开发流程，而不是上线前跑一次

方法和工具都到位后，最后一道坎是纪律。亚马逊云科技反复强调，评估不是上线前的一次性活动，而是贯穿开发的反馈环。落到实操，有四条核心实践：

1. **Holistic 多维评估：**评估必须超越传统的准确率指标，覆盖**质量、性能、责任、成本**四个维度。

   **质量：**推理连贯性、工具选择准确率、跨场景的任务完成率；
   **性能：**生产负载下的延迟、吞吐、资源利用；
   **责任：**安全、毒性、偏见缓解、幻觉检测、护栏，对齐组织政策与合规要求；
   **成本：**既算直接成本（模型推理、工具调用、数据处理），也算间接成本（人工投入、纠错补救）。

2. **Use case 特定指标：**标准指标之外，必须和领域专家协作定义业务指标。比如客服应用要额外跟踪客户满意度、首次接触解决率（first-contact resolution）、情感分析得分。

3. **平衡技术指标与业务指标：**延迟和成本，只有在答案正确的前提下才有意义。一个又快又便宜但答错的智能体，技术指标再漂亮也是负价值。

4. **生产环境持续评估：**预上线评估永远覆盖不到所有真实场景。生产中要持续监控多样化的用户行为、使用模式和上线前没见过的边缘情况，以发现随时间出现的能力衰退——操作 dashboard 跟踪 KPI、设告警阈值、自动异常检测、建立反馈环，一旦发现问题就触发模型重训、context engineering 修订，并回扣最终业务目标。

这条纪律最具体的体现，是 AgentCore Best Practices 里那个被反复引用的真实 tradeoff 案例：财务分析智能体的基线是 92% 工具选择准确率、3.2s 的 P50 延迟；

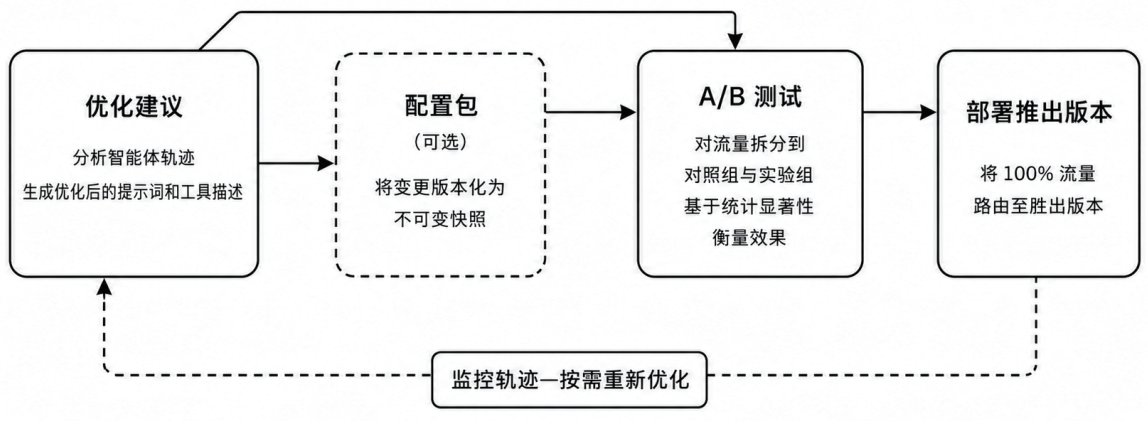
把模型**从 Claude Sonnet 4.5换到 Claude Haiku 4.5**后重跑评估，工具选择准确率降到 **87%**（下降约 5 个百分点），但 P50 延迟改善到 **1.8s**（提升约 44%）。这笔账值不值得换，必须靠量化评估来决策，而不是凭感觉。

同一篇里还给出了财务智能体的一组指标阈值基线，可作为读者设阈值的参照：

| 指标 | 目标阈值 |
| --- | --- |
| Tool Selection Accuracy | 95% |
| Parameter Extraction Accuracy | 98% |
| Refusal Accuracy | 100% |
| Response Quality | LLM-as-Judge 评判 |
| Latency | P50 < 2s, P95 < 5s |
| Cost per Query | 平均 < 5,000 tokens |

关键论点一句话总结：**每一次改动——换 prompt、加工具、换模型——都要重新跑评估。评估是开发流程里持续转动的反馈环，不是发布前打的那一次卡**。反馈环还要足够快，让你当场就能发现问题，而不是三次提交之后才察觉。

这套"改动 → 重跑评估 → 决策"的循环，AgentCore 已产品化为 Optimization（当前为 public preview），它构建在 AgentCore Evaluations 之上，含三项能力：**1/ Recommendations**——基于真实智能体traces 和一个目标 evaluator，自动产出优化后的 system prompt 或 tool descriptions，并解释改了什么、为什么；**2/ Configuration bundles**——把配置（system prompt / model ID / tool descriptions）做成版本化、不可变的快照，不改代码即可切换；**3/ A/B testing**——通过 AgentCore Gateway 把流量在 control/treatment 间切分，由 online evaluation 对每个 session 打分并报告统计显著性，胜出变体接管 100% 流量、其新 trace 成为下一轮起点。




# 工具支持：AgentCore Evaluations

前面讲的方法论，亚马逊云科技把它产品化成了 Amazon Bedrock AgentCore Evaluations，提供三种评估类型，覆盖从开发到生产的不同阶段：

- **Online evaluation**：对生产流量按采样率（如 10%）或条件过滤规则持续采样打分，结果可在 dashboard 聚合查看、追踪质量趋势，实时捕捉退化。
- **On-demand evaluation**：按 span / trace ID 评估你指定的交互，提交后服务只处理这些 spans / traces 并返回明细结果，适合开发期调试单条会话、验证修复、定向排查；
- **Batch evaluation**：对一批已有会话跑异步作业，由服务端完成会话发现、span 采集与打分，你只需指定 CloudWatch Logs 位置；适合基线测量、prompt / 模型改动前后对比、回归测试，以及对某段时间窗内的生产流量做周期性质量审计；

在此之上，AgentCore 还提供两个能力：**Dataset evaluation**（public preview）让你定义一组场景，自动调用智能体跑出会话、等遥测落地后再评估，相当于一个可复现的测试集——和直接给已有会话打分的 Batch 不同，它会先把会话生成出来；**Simulation** 则用模拟交互扩大测试覆盖。需要客观比对时，可在评估中附上 **ground truth**（expected responses、assertions、expected tool trajectories）作为标准答案。

评估通过 OpenTelemetry / OpenInference 兼容 Strands、LangGraph 等主流框架——trace 被自动采集、转成统一格式后用 LLM-as-a-Judge 等方式打分；**内置评估器**开箱即用，覆盖 helpfulness、correctness、faithfulness、goal success rate、harmfulness、tool selection / parameter accuracy 等维度，也支持创建**自定义评估器**适配领域需求。

放到更大的图景里，Evaluations 只是闭环中的一环——Observability → Evaluation → Optimization：Observability(见“Trace-driven评估工作流:四步把评估自动化”节)负责"看见"，让 AgentCore 自动发出的 trace 与指标可被采集；Evaluation 负责"判断"，基于这些 trace 打分、定位问题；Optimization（public preview）负责"改进"，它构建在 Evaluations 之上，把评估发现转成可验证的优化——Recommendations 基于真实 trace 和一个目标 evaluator，自动产出优化后的 system prompt 或 tool descriptions；Configuration bundles 把配置做成不改代码即可切换的版本化快照；A/B testing 再通过 AgentCore Gateway 切分流量、由在线评估对每个 session 打分并报告统计显著性。胜出变体接管流量后，其新 trace 又回流到 Observability，开始下一轮——这正呼应“工程纪律:把评估嵌入开发流程，而不是上线前跑一次”这一节，每一次改动都要重新跑评估。

**如需进一步了解 Evaluation-first 方法在实际场景中的落地方式，可参考本系列配套动手实验示例代码：https://github.com/aws-samples/sample-eval-first-building-enterprise-agents-with-agentcore**

04

# 第四章：
# 实战案例：
# 从工具使用到多智能体协作

理论讲完了，我们来看 Amazon 内部的团队，是怎么把上一节的评估方法论落到真实业务里的。这些 Agentic AI 应用都运行在企业级规模上、部署在亚马逊云科技基础设施之上，已在 Amazon 全球运营的多个实际业务场景中得到验证与落地。

下面三个案例分别对应第 3 部分指标体系的不同侧重——从工具使用，到意图识别，再到多智能体协作——恰好覆盖了智能体从简单到复杂的演进路径。最后，我们把这套方法论收束到一个读者可以亲手跑通的 HR 评估 Workshop，形成一个完整的评估闭环。

# 案例一：Amazon 购物助手 —— 工具使用评估

### 业务场景

为了给消费者顺畅的购物体验，Amazon 购物助手需要无缝对接底层系统的成百上千个 API 和 Web 服务——客户画像、商品与库存发现、下单履约——并在此基础上与用户进行长程多轮对话。问题在于：把这么多企业 API 手工 onboarding 成智能体可用的工具，是个极其繁琐的过程，**通常要耗费数月**。

更隐蔽的代价是：把遗留 API 转成智能体工具，需要为每个端点系统性地定义结构化 schema 和语义描述；schema 定义得差、描述不精确，会直接导致运行时**选错工具**——调用无关 API、无谓撑大上下文窗口、推高推理延迟、靠冗余的 LLM 调用拉高成本。

### 怎么解决，又怎么评估

**先治理：**Amazon 定义了跨组织的工具 schema 与描述规范，建立一套治理框架，对所有参与工具开发和智能体集成的团队提出强制合规要求——统一工具签名、输入校验 schema、输出契约、人类可读文档的格式，保证全企业范围内工具表示的一致性；

**再自动化：**构建一套 **LLM 驱动的 API self-onboarding 系统**，自动生成标准化的 tool schema 和描述，把数月的人工 onboarding 压缩成自动流程；

**最后评估：**基于历史 API 调用日志构建回归测试集，并结合规则校验 / 人工抽检确认关键标签；LLM 可用于样本扩增、格式归一和边界 case 生成，针对工具选择与使用做回归测试。

### 评估关键指标

Tool Selection Accuracy（选对工具）、Tool Parameter Accuracy（用准确的值填参）、Multi-turn Function Calling Accuracy（跨多轮维持连贯的工具调用序列）。

### 读者带走的一句话

工具的 schema 治理是智能体规模化的前提，而评估是这套治理体系的验收手段——随着智能体不断演进，"快速可靠地接入新 API 并评估其工具使用表现"的能力会越来越关键。

# 案例二：Amazon 客服智能体—— 意图检测评估

## 业务场景

客服智能体的第一道关卡是搞清楚"用户到底想干什么"。系统核心是一个**编排智能体（Orchestration 智能体）**，用 reasoning model 检测客户意图，再决定把查询路由到哪个由工具或子智能体实现的专精解析器。意图检测的赌注很高：一旦识别错，就会级联出一连串问题——路由到错误的解析器、给出不相关的回答、客户挫败感累积；体验崩塌的同时，更多客户转向人工，运营成本随之上升。

## 怎么解决，又怎么评估

评估数据靠两条腿：用匿名化的历史客户交互构造"用户查询 + 期望意图"的 ground truth 对；再开发一套 **LLM Simulator**，用 LLM 驱动的虚拟客户 persona 模拟多样化的用户场景与交互；

评估时，让编排智能体对模拟数据集里的用户查询生成意图，再把生成意图与 ground truth 意图逐一比对，验证是否一致；

除意图正确性外，评估还覆盖任务完成度（智能体的最终响应与意图解决，这是客服任务的终极目标），并为多轮对话加入 Topic Adherence 的分类与拒答指标，保障对话连贯性与体验质量。

## 评估关键指标

Intent Correctness（意图识别正确率）、Task Completion（任务完成度）、Topic Adherence Classification / Refusal（话题守界与拒答）。

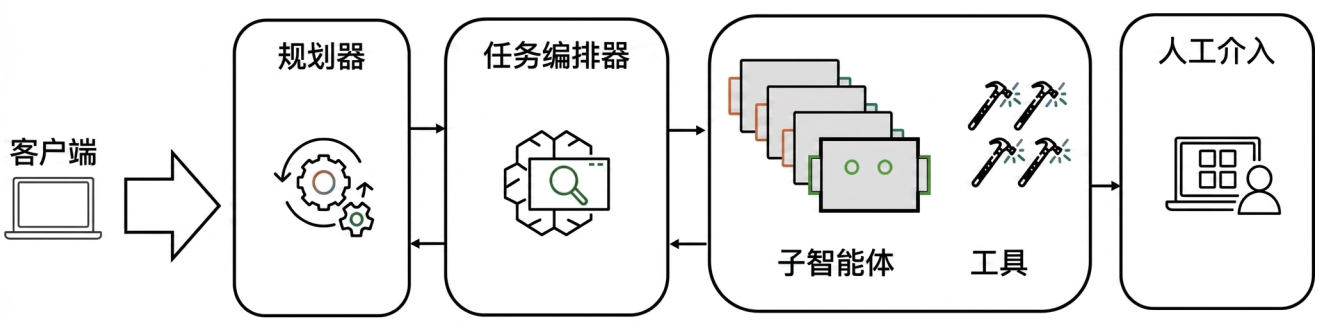
## 读者带走的一句话

用 LLM 模拟器扩展评估覆盖面，是一种 cost-effective 的标配做法——它让你用很低的成本，就把评估集从"历史发生过的"扩展到"真实可能发生的"，而意图检测评估的价值会一路延伸到运营效率和 AI 投资回报率。

# 案例三：Amazon 卖家助手 —— 多智能体协作评估

## 业务场景

企业面对的业务挑战越来越复合——从跨职能工作流编排，到不确定性下的实时决策。Amazon 团队因此越来越多地采用多智能体架构，把单体 AI 方案拆解成一组专精、可协作的智能体，让它们具备分布式推理、动态任务分配与自适应问题求解的能力。卖家助手就是一个典型例子。




### 架构（配图 2：多智能体协作架构示意，Planner-Specialist 模式）

流程始于一个 **LLM Planner & Task Orchestrator**，它接收用户请求，把复杂任务拆解成专精子任务，再根据各底层智能体的能力和当前负载，把每个子任务智能分配出去；底层智能体自主执行、用各自的专精工具完成目标，无需编排器持续盯防；完成后向编排器回报状态、确认、中间结果，或在超出边界时发起升级；编排器聚合这些响应、处理子任务间依赖，综合成一个连贯的最终结果。

## 评估的特殊性

多智能体系统的评估必须同时覆盖**个体智能体性能和集体系统动态**。除了评估各专精智能体在任务完成、推理、工具使用、记忆检索上的质量，还要衡**量智能体间的通信模式、协调效率、任务交接准确性**。这就用到了第 3 部分提到的那组前瞻指标：

- Planning Score：成功的子任务分配；
- Communication Score：为完成子任务，智能体间的通信消息量；
- Collaboration Success Rate：子任务成功完成的比例。

## 评估的特殊性

但更关键的是——自动指标抓不住涌现行为（**emergent behavior**）。多个智能体交互时可能产生任何单一智能体都不会有的、设计者没预料到的行为模式，这恰恰是风险较高、难以自动检测的部分。所以在多智能体场景里，HITL 不是可选项，而是必选项，它具体承担四项人工指标难以替代的把关职责：

- 评估智能体间通信，识别特定边缘场景下的**协调失败**；
- 判断智能体专精划分是否合理、任务拆解是否对齐了各智能体的能力；
- 当多个智能体给出相**互矛盾的建议**时，验证冲突解决策略是否得当；
- 保证多个智能体共同贡献于一个决策时的**逻辑一致性**，以及集体行为是否服务于既定业务目标。

"In multi-agent systems evaluation, HITL becomes critical because of the increased complexity and potential for unexpected emergent behaviors that automated metrics might fail to capture."（在多智能体系统评估中，HITL 变得至关重要——因为复杂度的上升，以及自动指标可能捕捉不到的意外涌现行为。）

## 读者带走的一句话

多智能体不是"更多智能体"那么简单，它的评估维度要从单体扩展到协作与涌现，而这些维度恰恰是自动化指标难以量化、却对生产部署成败至关重要的地方。

# 结语：

# 评估是循环，不是终点

# 让评估成为智能体工程的默认机制

回顾本系列四篇文章，我们始终围绕一个核心命题展开：企业级智能体的工程化落地，瓶颈不只在模型能力本身，更在于是否具备一套可持续衡量、持续反馈、持续优化的工程体系。

从 Agent Development Lifecycle，到评估方法论、评估粒度和证据权重，再到基于亚马逊云科技服务构建自动化评估工作流，以及 Amazon 内部生产级案例的实践拆解，本系列希望说明的是：评估不应被视为上线前的最后一道检查，而应成为智能体开发全生命周期中的基础能力。

真实业务环境中的智能体系统往往面临更复杂的不确定性：用户意图可能模糊，工具调用可能失败，业务规则和外部知识会持续变化，模型、Prompt、工具链和编排逻辑的任何一次调整，也都可能改变系统行为。因此，企业需要的不只是一次性的测试集或演示效果，而是一套覆盖开发、预上线和生产运行阶段的 Evaluation-first 机制。它既要评估最终答案是否正确，也要观察中间推理、工具调用、责任合规、延迟、成本和用户体验等关键维度。

这也是为什么 Observability、Evaluation 和 Optimization 必须形成闭环。可观测性帮助团队看清智能体真实发生了什么；评估帮助团队判断系统表现是否符合业务目标和风险边界；优化则基于评估结果持续改进 Prompt、模型、工具、知识库和系统架构。随着智能体从单一任务助手演进到工具调用型、多智能体协作型系统，这一闭环会变得更加重要。

同时，自动化评估并不意味着完全替代人工判断。尤其在高风险业务场景、复杂决策流程和多智能体协作场景中，Human-in-the-loop 仍然是校准评估标准、沉淀专家经验、提升自动化评估可信度的重要环节。可靠的企业评估体系，不是“人评”与“自动化评估”的二选一，而是让专家经验能够被结构化、规模化，并持续反哺自动化评估流程。

对于正在规划或已经启动智能体项目的企业团队，现在正是从 Evaluation-first 入手、构建生产级智能体工程体系的合适时机。团队可以从一个高价值但边界清晰的业务场景开始，定义任务目标、评估指标、测试数据和风险边界，并基于 Amazon Bedrock AgentCore 将智能体运行、工具接入、可观测性和评估流程逐步串联起来，在原型阶段就建立可验证、可迭代、可扩展的工程闭环。

面向未来，Agentic AI 的能力边界仍在快速扩展，但企业落地的核心问题会越来越清晰：不是简单地问“这个智能体能不能完成一次任务”，而是要持续追问“它能否在真实业务环境中稳定、可控、可解释、可优化地完成任务”。当评估成为智能体工程的默认机制，企业才能更有信心地将智能体从原型验证推进到生产系统，并在持续变化的业务环境中不断迭代。

**如需进一步了解 Evaluation-first 方法在实际场景中的落地方式，可参考本系列配套动手实验示例代码：https://github.com/aws-samples/sample-eval-first-building-enterprise-agents-with-agentcore**

# 参考链接

| | |
|---|---|
| AgentCore Evaluations 总览 | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/evaluations.html |
| How it works (Evaluations) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/how-it-works-evaluations.html |
| Evaluation types (Online/On-demand/Batch) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/evaluations-types.html |
| Evaluators (built-in + custom) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/evaluators.html |
| Built-in evaluators 总览 | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/built-in-evaluators-overview.html |
| Built-in evaluators prompt templates (14 个清单) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/prompt-templates-builtin.html |
| Custom evaluators (LLM-as-a-judge) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/custom-evaluators.html |
| Custom code-based evaluators (Lambda) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/code-based-evaluators.html |
| Online evaluation | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/online-evaluations.html |
| On-demand evaluation | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/on-demand-evaluations.html |
| Batch evaluation | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/batch-evaluations.html |
| Dataset evaluation | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/dataset-evaluations.html |
| Simulation | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/simulation.html |
| Prerequisites (IAM / ADOT) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/evaluations-prerequisites.html |
| Encryption at rest (KMS) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/evaluations-encryption.html |
| Optimization 总览 (preview 声明) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/optimization.html |
| Optimization — How it works | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/optimization-how-it-works.html |
| Optimization — Recommendations | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/optimization-recommendations.html |
| Optimization — Configuration bundles | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/configuration-bundles.html |
| Optimization — A/B testing | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/ab-testing.html |
| Observability 总览 | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/observability.html |
| Observability — 配置/埋点 (ADOT/Transaction Search) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/observability-configure.html |
| Observability — 服务自带数据 (各资源 metrics 表) | https://docs.aws.amazon.com/bedrock-agentcore/latest/devguide/observability-service-provided.html |
| CloudWatch GenAI Observability | https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/GenAI-observability.html |

*前述特定亚马逊云科技生成式人工智能相关的服务目前在亚马逊云科技海外区域可用。亚马逊云科技中国区域相关云服务由西云数据和光环新网运营，具体信息以中国区域官网为准。


亚马逊云科技