Eval Harness 源码内核

从一个样本到一次发布决定，读懂评测系统如何运行

完整中文离线版 · 源码课程、工程机制、案例与实验

原创文档：CC BY 4.0 原创代码：MIT

目录

Eval Harness 源码内核

Eval Harness 源码内核

EVAL HARNESS INTERNALS

从一个样本到一次发布决定，读懂评测系统如何运行

从一个样本到一次发布决定，读懂评测系统如何运行。

这是一套写给开发者的中文 Eval Harness 源码教材，仓库里还配了一个可以离线运行的 Python Reference Harness。你会先分清 Task、Dataset、Trial、Attempt 和 Trace，然后对照六套真实的开源实现，看懂系统怎样执行、评分和做统计比较，最后又如何作出发布 Gate 决定。先跑通，再拆开看。

开始学习 选择学习路线 下载完整中文 PDF

姊妹项目

这套教材从「一次运行留下了什么证据」问起，重点是怎样根据证据作出判定。如果你还想知道这些证据是怎样产生的， 也就是模型给出下一步意图后，到底由谁把它变成可以控制、能够恢复的真实动作，可以去看 Agent Harness 源码内核 里， 那套教材对着六套编程智能体的锁定源码，一路追查配置怎样组出来、工具循环如何往下走、权限在哪几层拦住操作， 以及会话中断后怎样恢复状态。两套教材用的是同一种读法：找到已经锁定的源码，再从精确行号一步步往下追。

三条主线

主线	你会得到什么
共同语言	分清 Agent Harness 与 Eval Harness，建立 Sample、Trial、Attempt、Artifact、Score、Metric 和 Gate 的严格边界
源码知识库	沿锁定 commit 阅读 lm-evaluation-harness、Inspect AI、OpenAI Evals、Promptfoo、DeepEval、Harbor 与 Terminal-Bench 1
可运行参考	在无模型凭据、无容器的条件下执行 Target、保留证据、重评分、比较 Candidate/Baseline 并生成 Gate

第一次运行

```
uv sync --frozen
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/shipping
uv run eval-harness-ref inspect output/shipping
uv run eval-harness-ref compare output/shipping --candidate-target fixed --baseline-target
buggy
```

完整目录已经把案例、实验和验证方法按学习顺序排好，你可以从文档总目录 进入。

从这里开始：第一次把 Eval 跑成完整证据链

返回目录 · 进入第一章

这套教材回答什么

你可能已经会调模型、写测试样本，甚至跑出了一个准确率，但要让 Eval Harness 产生可信的结果，你还得追问：到底测了哪个系统版本，运行前把哪些 Sample 排进了计划，超时重试是否掩盖了失败，Scorer 看到了哪些证据，分母怎么算出来，Candidate 和 Baseline 的差异稳不稳定，以及证据不够时 Gate 为什么不能放行？

为了把这些问题逐一落到可检查的记录上，本仓库把一次评测拆成下面这条对象链，并规定每个对象管哪一段、留什么记录。

```
EvaluationSpec
  → Sample × Target × Repetition
  → Trial
  → Attempt
  → TraceEvent / Artifact
  → ObservationBundle
  → ScoreRecord
  → MetricEstimate / Comparison
  → GateDecision / Report
```

先别急着背类名。读到任何一个对象时，你都可以顺着三个问题往下查：它要处理哪类错误，身份在哪里定下来，后面的环节又怎样倒查回来。

贯穿全书的第一个问题

规则说订单满 100 元就免运费，buggy 程序却写成 amount > 100，fixed 程序则写成 amount >= 100。把 99、100、101 元这三个 Sample 分别交给两个 Target 跑一遍，你在运行前就能确定会产生 6 个 Trial。

金额为 100 元的 buggy Trial 虽然把进程正常跑完了，答案却是错的，所以记录中会同时出现 Trial completed、Attempt succeeded/canonical、Score failed、Metric 2/3 和 Gate failed。这些状态并不冲突。你先把它们分清，后面查证据链时才不会搅在一起。

第一次运行

准备好 Python 3.12 和 uv 以后，直接执行下面这组命令，不用提供模型凭据，也能稳定地复现同一份运行结果。

```
uv sync --frozen
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/shipping
uv run eval-harness-ref inspect output/shipping
```

接着依次查看以下文件。

1. output/shipping/run.json：每个 Trial 有几个 Attempt，哪个是 canonical；
2. output/shipping/traces/：事件怎样按因果顺序写入 JSONL；
3. output/shipping/artifacts/：输出字节怎样按 SHA-256 内容寻址；
4. output/shipping/evidence.json：Observation Bundle 怎样冻结评分输入；
5. output/shipping/report.json：Score、Metric 和 Gate 怎样保留引用。

后面再跑 score 和 gate 时，程序不会重跑 Target，只会读取已经冻结的证据并重新计算，然后分别写出 rescore.json 和 regate.json。如果重算后的结果变了，先去核对证据和 Scorer 的身份，别从几份报告里挑一个更好看的数字。

阅读约定

Python 标识符和上游项目名称保留英文，首次出现给出中文解释；

「源码事实」必须落到锁定 commit 的永久链接；

跨多个调用点重建的责任边界标为「机制解释」；

本仓库缩小的伪代码和 Reference Harness 标为「教学简化」；

公开源码不足时明确写「不可核对」，不猜闭源内部实现；

每篇核心课都给实验、预期输出和答案，你不需要等待逐题互动才能继续。

接下来怎么走

如果你是第一次系统学评测，可以从Agent Harness 与 Eval Harness 起步，按顺序读完七篇基础课。如果你已经做过评测工程，就直接打开学习路线，按手头遇到的问题选源码、Agent 环境或 Eval-to-RL 路径。

Eval Harness 源码内核：完整目录

第一次读这套教材，建议你从学习入口 开始，先跟着运费案例跑完整条证据链，再决定后面要往哪里深挖。目录依次安排了「共同语言 → 最小实现 → 上游源码 → 工程专题 → 横向比较 → 案例与实验」，按这个顺序读，你就不会一开始便钻进各种工具名称，更不会把某个项目自己的叫法当成通用语义。

入口

第一部分：基础篇

第二部分：可运行参考

Reference Harness 用一份最小合同把 Trial/Attempt、Trace、按内容寻址的 Artifact、Observation、Score、Metric、Comparison、Gate 和离线报告串了起来。你可以拿它检验教材里讲的机制，但不能因此宣称它复刻了某套上游工具。这条边界不能越过。

第三部分：上游源码课程

5 DeepEval：Golden、TestCase、Metric 与执行策略

6 Harbor 与 Terminal-Bench 1：环境、Agent、Verifier 与 Trial

每条源码课程都先用导读带你进门，再分三篇往深处讲。正文会对着锁定的提交，依次找到源码从哪里进入、调用怎样往下走、哪些数据结构最关键，以及失败如何一层层传出来，然后再让你用实验、预期输出和参考答案检验自己是否真的看懂了这些设计取舍。别只记名词。

第四部分：工程篇

1 最小 Eval Loop

2 Run Identity 与可复现性

3 Retry 与 Recovery

4 LLM-as-a-Judge

5 统计比较

6 Agent Environment

7 Quality Gate

8 Eval-to-RL

第五部分：横向比较

1 Task、Dataset 与 Target

2 Runner、并发、缓存与 Retry

3 Trace、Artifact 与血缘

4 Scorer、Judge 与 Outcome

5 Metric、统计单位与不确定性

6 Agent Environment 与 Final State

7 Report、CI 与 Release Gate

8 平台适配器边界

第六部分：案例

运费边界

退款 Agent

企业知识助手

合同审查 Agent

SWE-bench 补丁评测机制

前四个案例都准备了结果确定的 buggy/fixed Target，你也可以直接把门禁跑起来。SWE-bench 则把重心放在环境怎样影响机制，你会看到补丁、隔离环境、测试证据和实例级判定如何互相约束。

第七部分：实验

1 运行一份确定性 Eval

2 新增 Target Adapter

3 编写 Scorer

4 重复运行并比较

5 导入 Agent Trace

6 构建 Release Gate

附录

术语表

验证与证据边界

来源与许可证

怎样读一篇源码课

读源码课时，先查清课程锁定了哪个版本，并找到入口，然后顺着调用链逐站追问：谁发起调用，传进来什么，状态在哪里变了，结果怎样返回，失败又怎样传出去，以及哪组测试锁住了这些行为。读完再跑一遍本仓库的确定性实验，拿实际输出检验自己是否看懂了机制，别只记住 API。

学习路线

返回目录 · 回到最后一篇基础课

路线 A：第一次系统学习 Eval Harness

按顺序读七篇基础课，每篇都先把实验跑起来，看完输出再对照答案，检查自己是不是真的理解了。基础打好后，再读 Reference Harness 工程篇，然后进入六条上游源码课程。这条路线会训练你从任意 Harness 里找到谁在规划 Trial、谁在留证据、谁负责评分和聚合，以及 Gate 到底在哪里作出判定，不会只让你背下某个工具的配置。

先把一条链走通。

验收时，给你一份只写了最终准确率的报告，你应该能找出它还缺哪些身份、分母、血缘和不确定性证据，并说清证据缺失会让 Gate 判定失败、直接阻断，还是只能给出「无法判断」。

路线 B：源码阅读与工具选型

先读基础 01、02、04、05，然后按「完整端到端 README → 入口 → 数据结构 → 执行循环 → Scorer → 报告 → 测试」这条线追每一门源码课。等你能说清一套工具怎样把状态传下去，再进入横向比较，这样才不会拿功能勾选表充当语义分析。

验收时，面对两个项目中同名的 metric，你应该能查清它们收到的是一条 Observation 还是一组 Score，它们有没有负责 Judge、聚合或 Gate，并且能从锁定的源码里拿出证据。

路线 C：Agent 与 Coding Agent 评测

重点读基础 01、03、04、06，再跟着 Harbor/Terminal-Bench、SWE-bench 机制案例和 Agent Environment 工程篇往下学。建模时要把工具事件、Diff、测试结果和环境最后的状态分开记，因为 Agent 在最后自己说「完成了」，不能替代独立断言。

验收时，如果一个 Coding Agent 超时后又成功了，你应该能分清 Agent 自己做的恢复和 Harness 记录的 Attempt，证明统计分母没变，还能从 Gate 一路倒查到补丁和测试 Artifact。

路线 D：Eval-to-RL 与质量发布

重点读基础 05、06、07，再学 Judge 怎样校准、结果如何做统计比较、质量 Gate 根据什么放行，以及 Eval-to-RL 工程篇怎样连接评测和训练。在整个过程中，训练 reward、checkpoint choice 和 independent release eval 必须分开记。边界不能混。

验收时，给你一条 Score，你应该能写出 RewardAdapter 的能力合同，判断它能不能支持 DPO 偏好、GRPO/RFT 标量 reward，是只能支持一部分还是完全不可用，并设计一份不参与训练的独立发布集。

推荐实践节奏

每学完一个阶段，都要留下一份可以核对的产物：第一阶段保留完整的运行目录，第二阶段写出带永久链接的调用链笔记，第三阶段加入一个 Target Adapter 或 Scorer，第四阶段则做完一份对比报告，并在里面保留计划分母和独立 Gate。只要这些产物真的能跑、能回放，你就会比单纯统计「读完了多少页」更早发现自己还有哪里没看懂。

01 | Agent Harness 与 Eval Harness：先分清「执行」与「评测」

上一章 · 下一章

本篇要解决什么问题

Agent 和评测系统都会调用模型、执行工具，也都会保存 Trace，所以只看表面动作，很容易把它们当成同一种 Harness。要分清两者，就看各自负责做什么决定：Agent Harness 要把一项用户任务推进到可以结束，Eval Harness 则按同一套规则比较多次被测行为，再根据证据判断质量。

两边一旦混在一起，评测层就可能替被测 Agent 改提示、补工具结果，甚至让已经答错的任务再答一次。这样算出的分数反映了两个系统联手后的结果，已经不是 Agent 原本交出的答案。反过来，如果只留下 Agent 的最终回答，却没冻结 Dataset、Target 身份和评分策略，你仍然回答不了「这个版本在这组任务上是否更好」。

学完你能解释什么

- 为什么 Agent Loop 的停止条件属于被测系统，而 Trial 的停止、取消和预算属于评测协议；
- 为什么 Trace 是两个仓库的接口，却不能因此把两套 Harness 合并；
- 为什么一个 Agent 自己声称「任务完成」只是观察值，不是独立 Score；
- 怎样判断一项能力应该写进 Agent Harness 课程还是 Eval Harness 课程。

贯穿案例

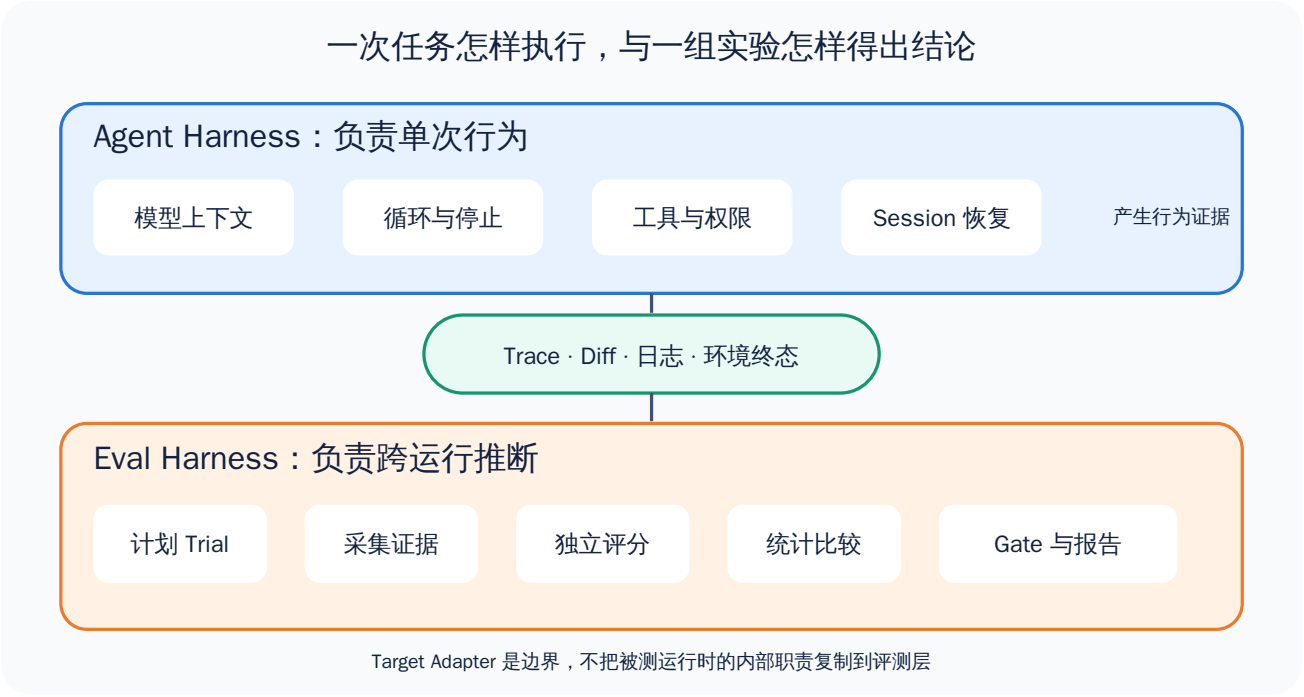
运费函数规定订单金额达到 100 元就免运费，旧实现却写成 amount > 100。Agent Harness 要管模型怎样读文件、改比较符、跑测试，以及什么时候停下来。Eval Harness 管的是另一组问题：它先冻结金额为 99、100、101 的三个样本，让 buggy 和 fixed 两个 Target 各跑一次，再保存输出、按同一规则评分并形成 Gate。前者解释修复过程，后者回答我们凭什么相信边界行为确实改善了。

核心概念与边界

Agent Harness 管理模型上下文、Agent Loop、工具暴露、权限、Session、压缩和恢复，所以它通常要判断「这次任务还能不能继续，什么时候可以结束」。Eval Harness 管理 EvaluationSpec、Dataset、Trial Plan、Target Adapter（被测对象适配器）、Observation、Scorer、Metric、比较和 Gate，但这些环节各有自己的状态：一次运行可以正常完成，评分却可能失败，即使一组评分全部通过，整体 Gate 仍可能因为缺少样本而无法判断。

Target Adapter 把两边明确隔开：它接收 Trial 和运行约束，调用被测系统，再把系统行为和可观察证据交回来。你可以用它接普通函数、RAG 服务、Coding Agent 或多智能体系统，却不能让它替这些系统重新做内部决策。Agent Harness 产出带有明确语义的 Trace，Eval Harness 则检查 Trace 是否完整、是否确实来自指定 Target，并决定哪些字段可以交给评分环节。

机制图



Target Adapter 是边界，不把被测运行时的内部职责复制到评测层

调用链与状态变化

1. Eval Planner 依据冻结配置生成 Trial，把输入、Target 身份和重复序号固定下来。
2. Runner 把 Trial 交给 Target Adapter。此后，模型选择工具、Agent 是否继续、如何恢复 Session，均由被测 Agent Harness 负责。
3. Adapter 返回输出、Trace、Diff、日志或环境终态。Runner 只区分产品结果与基础设施故障，不替 Agent 改答案。
4. Eval Harness 把 canonical Attempt 的事件和 Artifact 组成 Observation Bundle。
5. 独立 Scorer 消费 Bundle，Metric 对预声明 Trial 集合聚合，Gate 再依据冻结阈值作决定。

你可以直接从 Reference Harness 的 run_evaluation 和 SubprocessTarget 看出这条边界。

关键数据结构

对象	属于哪一侧	必须保留的信息
Agent Session	Agent Harness	消息、工具状态、压缩与恢复点
TargetSpec	Eval Harness	被测适配器、版本与有效配置
Trial	Eval Harness	Sample、Target、重复序号和稳定 ID
TraceEvent	接口	因果父子关系、事件类型与可观察 payload
ObservationBundle	Eval Harness	canonical Attempt、Trace 与 Artifact 摘要

设计取舍

这里适合用依赖倒置，让 Eval Harness 只认 Target Adapter 协议，不去绑定 Claude、Codex 或某个 RAG 框架的内部类。这样一来，同一份 Dataset 和同一个 Scorer 就能拿来比较不同系统，不过 Adapter 必须说清楚它能不能导出工具事件、重置环境，以及报告实际模型版本。做不到的能力就直接标成不可用。填一个假的空字段，只会把证据缺口藏起来。

Inspect AI 的锁定源码把通用评测入口放在 eval() 和 eval_async()，再把 Task 的执行逻辑下沉到 _eval/task。这是上游源码事实。为了比较多套实现，本篇把这套分工归纳成两层责任，这属于机制解释，不要求各个项目使用相同的类名或目录结构。

失败语义

Agent 返回错误运费：Target 运行完成、Score 失败；不得换一个 Attempt 再答一次。

宿主进程启动失败：Attempt 为基础设施失败；在预算内可创建恢复 Attempt。

Trace 缺关键终态：运行也许完成，但 Scorer 应返回 unscorable 或 invalid。

Target 身份与计划不一致：Trial 证据无效，不能用看似正确的输出替代身份错误。

质量阈值未达到：Gate 失败；这不等于 Harness 崩溃。

动手实验

在仓库根目录运行：

```
eval-harness-ref run reference/examples/shipping/eval.yaml --output output/shipping
eval-harness-ref inspect output/shipping
```

打开 output/shipping/report.json，分别找出 buggy Target 跑到了什么状态、Score 怎样判、Gate 又怎样判。然后查看 run.json，确认错误答案没有触发第二个 Attempt。

预期输出与答案

你应该看到 6 个计划 Trial，因为 3 个样本分别交给 2 个 Target 后，正好组成六种配对。buggy 在金额 100 上答错了，但对应 Trial 仍是 completed，随后 Score 和 buggy Gate 才各自判为 failed。fixed 的三个 Score 都会通过，Gate 为 passed。这里要读成「运行已经完成，产品结果随后被独立判错」。如果写成「buggy 运行失败」，就把执行状态和质量结论混到了一起。

常见误解

「Agent 已经跑过测试，所以不需要外部 Scorer」这句话漏掉了几种情况：被测系统可能少跑了测试、改过测试，也可能读错输出。「Eval Harness 也有循环，所以它就是 Agent Harness」则把负责调度的循环当成了替 Agent 做决定的循环。日志再多也不等于 Trace 可信，因为一旦缺少身份、因果顺序和摘要约束，你拿到的只是更长的文本。

如何核对

先读 runner.py，看它怎样分开产品失败和基础设施异常，再读 pipeline.py，看 Bundle、Score、Metric 和 Gate 按什么顺序生成。核对上游实现时，可以从锁定的 Inspect AI 入口继续追到 _eval/task/run.py，确认「公共 Eval 入口」和「Task 执行」确实由不同位置负责。

与其他 Harness 的关系

lm-evaluation-harness 主要围绕 Task、Model Adapter 和批量请求组织代码，Promptfoo 更强调 Provider、Test Case 与 Assertion，Harbor 则更重视怎样管理 Agent 和环境的整个生命周期。它们切分代码的方式各不相同，但你仍然可以沿着「被测执行、证据、评分、聚合」四个环节来比较。Agent Harness 仓库研究 Claude、Codex、Gemini、DeepSeek Harness、pi、OpenCode 等运行时内部怎样工作，本仓库关心的是如何把这些运行时当作 Target，公平地执行并评分。

本篇不能证明什么

分清这条边界，不能证明某个 Agent 已经安全，也不能证明某套 Eval Harness 已经可以用于生产。一次确定性示例更不能直接授权真实业务发布。它能做的是把每项职责摆清楚，并给出可核对的最小实现。真正落到实际系统时，你还要准备独立 Dataset、隔离环境，补上统计设计和风险门禁。

上一章 · 下一章

02 | Task、Dataset、Target 与 Environment：把问题定义完整

上一章 · 下一章

本篇要解决什么问题

「评测一下退款 Agent」还不能直接拿来执行，因为你不知道要测的是退款资格判断、对客话术，还是实际产生的副作用，也不知道样本取自哪里、测哪个版本，账户、订单和工具又处在什么状态。很多评测争论表面上围着分数打转，继续追问才会发现，真正混在一起的是四件事：Task 规定要做什么，Dataset 提供具体样本，Target 指定测谁，Environment（环境）则准备运行时会被读取或改变的外部状态。这四件事不能混。

学完你能解释什么

为什么同一 Dataset 可以服务多个 Target，但不能自动代表同一 Task；

为什么 prompt 或模型名字不足以形成 Target 身份；

为什么 Agent 评测必须把 Environment 初态和终态纳入协议；

怎样从一句业务问题得到可物化的 EvaluationSpec。

贯穿案例

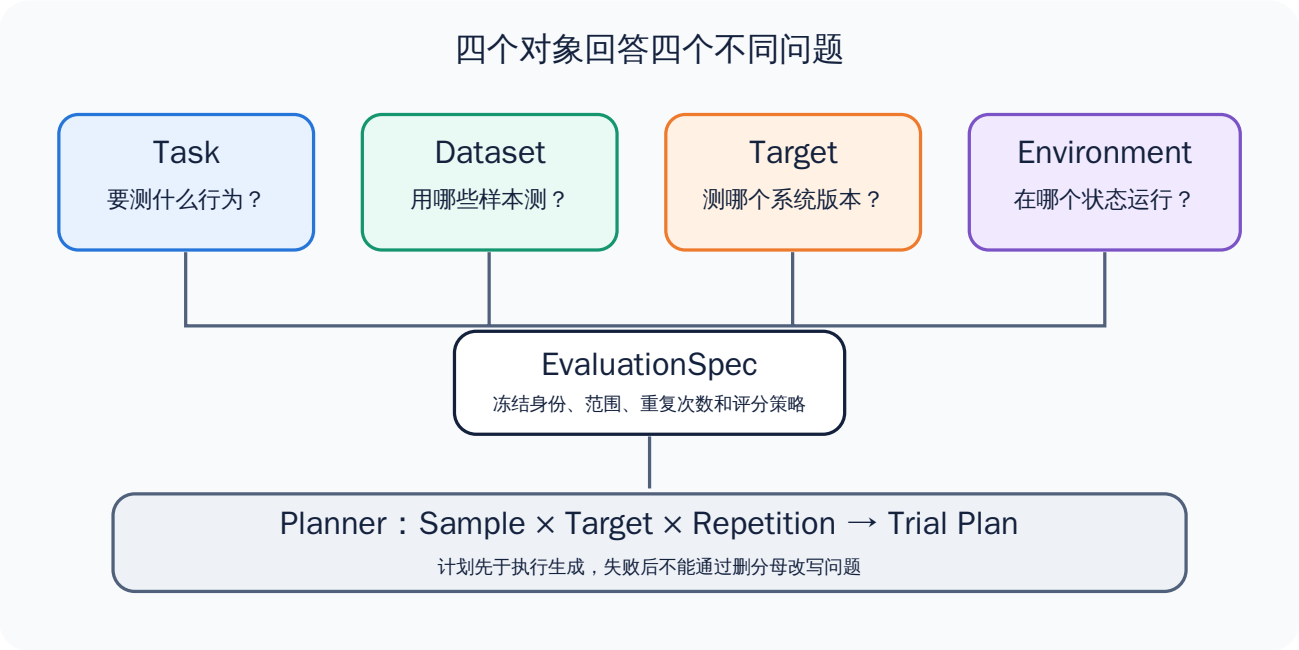
运费案例要求系统「根据订单金额返回运费」，金额达到 100 时费用为 0，Task 要守住的就是这条判定边界。Dataset 收进金额为 99、100、101 的三个 Sample，buggy 和 fixed 两个本地脚本则是 Target，所以在这个最小例子里，Environment 只要准备 Python 3.12 子进程和只读输入即可。换成退款 Agent，外部状态就复杂多了，Environment 必须包含订单余额、退款 API、权限，以及能够复位的数据库。

核心概念与边界

Task 说明要观察什么行为、输入输出要遵守什么契约，以及系统可以做哪些事，通常会覆盖多条样本。Dataset 把选出的 Sample 按版本和分组组织起来，但它只能代表实际业务分布的一个切面。Target 要说清楚究竟测哪个系统，至少包括 Adapter（适配器）、模型或程序版本、实际生效的配置和依赖。Environment 则装下系统运行时能够读取或改变的外部状态，比如文件、容器、服务、账户、时钟和网络策略。

把这四件事合在一起，你才能提出一个能实际评测的问题：「先把环境 E 冻结在指定初态，再让 Target T 处理 Dataset D 中用于检验 Task K 的样本，它会表现得怎样？」其中任何一项变了，结论都可能跟着变。如果你把 Environment 藏进 Target 名称，或者把 Reference 答案放进 Agent 能看到的环境，评测还没开始，身份就已经混乱，答案也可能提前泄露。这会直接污染结果。

机制图



调用链与状态变化

1. 设计者把业务决定改写为 Task：输入是什么、允许什么行为、什么证据支持通过。
2. Dataset Builder 选择 Sample，并固定 Sample ID、Split、Reference 与来源。
3. Target Resolver 把逻辑名称调和为实际 Adapter 和版本，拒绝身份漂移。
4. Environment Factory 创建干净初态，限制网络和凭据，声明重置方式。
5. EvaluationSpec 冻结 Target 列表与重复次数；Planner 执行笛卡尔积，生成不可随结果缩减的 Trial Plan。

Reference Harness 用 EvaluationSpec、plan_trials 和 shipping 配置展示了最小实现。后面的工程篇会给复杂 Agent 环境加上环境工厂，但这四个对象各自表达的意思不会因此改变。

关键数据结构

```
evaluation_id: shipping-boundary
dataset: dataset.jsonl
repetitions: 1
targets:
  - target_id: buggy
    adapter: python_script
    script: target_buggy.py
scorer:
  scorer_id: shipping-fee:v1
  field: fee
```

这里的 target_id 表示计划里认定的逻辑身份，解析器则根据 script 找到真正的执行入口。当 Target 换成真实模型，运行记录还要写下模型版本、系统提示摘要、工具集合，以及服务端实际返回的身份。Environment 也要分别保存镜像摘要、初态 Fixture（测试夹具）和重置结果。如果把这些信息全塞进一个 config 字符串，以后就很难查清每项配置来自哪里。

设计取舍

Dataset 可以直接写进配置，也可以放在外部文件里单独管理版本。使用外部 JSONL 后，你更容易维护稳定 ID、查看 Diff 和切分数据，但也得额外管好文件从哪里来、经历过哪些变化。Target 可以由构造函数直接创建，也可以交给 Registry（注册表）按名称加载。Registry 用起来方便，却可能遮住真正生效的配置，所以运行报告必须保存解析后的身份。容器能把 Environment 隔离得更严，不过纯函数评测通常不需要这么重。只要 Agent 会产生副作用，就得先解决隔离和重置问题。

lm-evaluation-harness 分别用 Task 承载任务，用 LM 适配模型，这是上游源码事实。本篇为了比较不同 Harness，才把它们归到四个对象中，这属于机制解释，并不要求上游也定义一个名为 Environment 的类。

失败语义

Dataset 为空或 Sample ID 重复：规格无效，不能开始运行。

Target 声明模型 A，服务实际返回模型 B：身份调和失败，证据应标为 invalid。

Environment 创建失败：Trial blocked，可在基础设施预算内恢复；不能当作产品零分。

Environment 未成功重置：后续 Trial 可能互相污染，应阻断而非继续累计分数。

Task 只写「效果好」：Scorer 无法知道观察边界，评测设计尚未完成。

动手实验

把 shipping 配置复制到临时目录，将 repetitions 改成 2，同时保留 3 个 Sample 与 2 个 Target。运行 eval-harness-ref run 后，先查看 evidence.json 里的 trials，暂时别看报告分数。

预期输出与答案

你应该得到 12 个 Trial，因为 3 个 Sample 分别交给 2 个 Target，再各跑 2 次，正好组成 12 种计划内的组合。每个 Trial 都在同一 Task 下绑定一个 Sample、一个 Target 和一个重复序号。增加重复次数会改变实验怎样安排，却不会复制 Sample ID。如果 Environment 不能让每次 Trial 都从等价初态开始，这 12 次运行就不能直接放在一起比较。

常见误解

如果认定「Dataset 就是 Task」，问题换了以后，你可能还在沿用旧分数。如果只把模型名当作 Target，提示、工具和服务版本就会从身份记录里消失。把 Docker 当作 Environment 可复现的充分条件也站不住，因为外部 API、时间和凭据仍可能变化。样本变多同样不保证更接近业务，因为增加数量不会自动消除采样偏差和重复实体。

如何核对

运行 python -m pytest tests/test_planner.py -q，确认 Target × Sample × Repetition 展开后的数量和顺序保持稳定。然后对照 sources/sources.lock.yml 与正文里的永久链接，你会看到「上游源码身份」同样要由 commit 固定，不能跟着浮动的 main 变化。上游 Harbor 的 environments/base.py 则能帮你核对，为什么 Agent 环境需要单独抽出来。

与其他 Harness 的关系

OpenAI Evals 主要通过 Registry 与 Eval Spec 组织评测，Promptfoo 用配置把 Provider 和 Test Case 接起来，DeepEval 沿用测试框架里的 Test Case 思路，Harbor 则把任务包、Agent 和 Environment 组合成 Trial。这些名字不能逐字对齐：Provider 可能同时负责解析并调用 Target，Golden 也可能同时装着输入和 Reference。比较不同实现时，你得先看每个对象实际负责什么，再对照字段。

本篇不能证明什么

即使四个对象都定义完整，Dataset 仍可能没有代表性，Environment 仍可能泄漏，Target 也可能无法复现。定义完整还不够。这套定义的作用，是给每类风险找到明确的负责人和能够检查的字段，让你知道出了问题该往哪里追。至于评测是否有效、结果是否可靠，以及能不能据此发布，还要看后续章节提供的证据。

上一章 · 下一章

03 | Sample、Trial 与 Attempt：别让重试改写统计分母

上一章 · 下一章

本篇要解决什么问题

调用超时后再试一次很合理，模型答错后再问一次也可能得到正确答案，但这两种「重试」改动的不是同一层。前一种是在基础设施出故障后，设法把同一个实验对象跑完。后一种却让被测产品多答了一次。如果你只留下最后成功的结果，原本失败的样本就会被一次次试成通过，成功率的分母也会跟着能用的结果来回变化。

学完你能解释什么

- Sample、Trial、Attempt 为什么是三层对象，而不是三个同义词；
- 为什么 Trial 是统计单位，Attempt 是恢复记录；
- canonical Attempt 怎样阻止多个结果同时进入评分；
- blocked、completed 与 Score failed 为什么可以同时出现在不同层。

贯穿案例

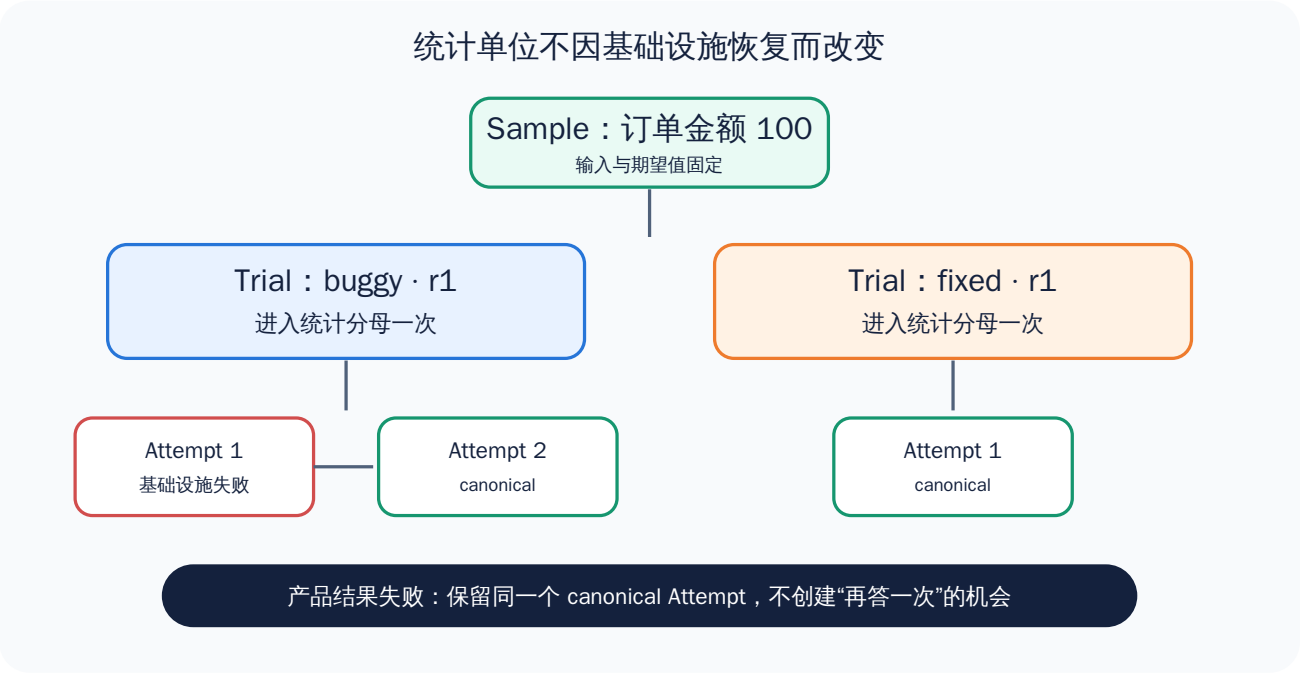
把金额为 100 的 Sample 分别交给 buggy 和 fixed Target，会得到两个 Trial。如果 buggy 脚本正常启动并返回运费 10，这个 Trial 就已经跑完，即使答案错了，也不能为了「希望它答对」再启动一次。如果 Python 子进程因为临时资源错误没能启动，Harness 才可以动用预先声明的基础设施恢复预算，创建 Attempt 2。等恢复成功后，Attempt 2 会成为唯一的 canonical Attempt，而 Attempt 1 仍要保留在证据里。

核心概念与边界

Sample 是 Dataset 中一份带稳定身份的输入和 Reference。Planner 把 Sample × Target × Repetition 展开后，才得到 Trial。每个 Trial 都是统计时要计算的一次观察，而且在执行前就已经进入计划分母。Attempt 则记录 Harness 为了跑完某个 Trial，实际在基础设施上尝试了几次，包括每次的序号、错误码和是否为 canonical。

重复次数不能算作 Attempt。让随机模型重复运行 5 次，会生成 5 个 Trial，因为计划把每一次都当成独立观察。某个 Trial 因为网络握手失败而恢复 2 次，统计时仍然只能算 1 次。如果 retry 原本就是 Target 算法的一部分，比如 Agent 会在 Loop 内再次调用模型，就应该让它留在同一次 Target 行为里，并通过 Trace 暴露出来，Eval Harness 不能暗中多加 Attempt。

机制图



调用链与状态变化

1. Planner 在运行前生成稳定 Trial ID，格式包含 run、Target、Sample 与重复序号。
2. Runner 创建 Attempt 1，调用 Target Adapter。
3. Adapter 返回正常结果时，Attempt 状态为 succeeded 且成为 canonical；Trial 状态为 completed。
4. Adapter 抛出被声明为基础设施错误的异常时，Attempt 为 infra_failed；若还有预算，Runner 创建下一个 Attempt。
5. 所有恢复机会耗尽后，Trial 为 blocked，没有 canonical Attempt，也不能伪造 Observation Bundle。
6. 产品失败由 Adapter 作为正常返回的一种结果类型传回——它仍产生 canonical Attempt，随后由 Scorer 判断失败。

run_trial 直接实现了这套状态机，而 tests/test_runner.py 则分别用测试锁住三条规则：基础设施可以按预算恢复，产品失败不能由 Harness 重试，canonical 必须唯一。

关键数据结构

```
Trial
  trial_id
  target_id
  sample
  repetition

Attempt
  attempt_id
  trial_id
  ordinal
  status
  canonical
  error_code
```

Trial 不保存「最终分数」，因为 Harness 要先完成执行，再把证据交给评分阶段。Attempt 也不保存 Dataset 权重，因为负责基础设施恢复的这一层不该决定怎样统计。canonical 指向协议允许送入后续证据链的唯一一次执行，并不代表「表现最好的一次」，所以你必须在看到结果之前定好选择规则。

设计取舍

最简单的本地 Harness 可以同步执行，再从内存里的结果选出 canonical。分布式系统还得处理 Lease、Fencing Token 和幂等提交，否则旧 Worker 超时后，新 Worker 虽然已经接管，旧 Worker 仍可能迟到并写入结果。首版 Reference Harness 没有声称已经解决分布式一致性，它只规定「一个 Trial 最多有一个 canonical」，这样既把语义说清楚，也不会让复杂基础设施遮住真正的统计问题。

Harbor 锁定版本里的 Trial 抽象基类 展示了 Agent Trial 从创建到结束怎样变化，这是上游源码事实。Reference Harness 单独划出 Attempt 这一层，是本仓库的教学实现，不能把其中字段逐项套到 Harbor 的内部结构上。

失败语义

超时、进程无法启动、临时工作区失败：基础设施错误，可按预算增加 Attempt。

程序返回非零且表示被测测试失败：产品失败，Trial 完成，不由 Harness 重试。

两个 Attempt 都标 canonical：Trial invalid，后续 Score 不可信。

只有失败 Attempt：Trial blocked，不是 Score 0。

运行后才删除困难 Trial：破坏预声明分母，整次 Metric 无效。

动手实验

运行 python -m pytest tests/test_runner.py -q，然后打开测试里的 FlakyTarget 和产品失败 Target，先自己判断每种情况会创建几个 Attempt、哪一个是 canonical，以及 Trial 最后处于什么状态，再对照断言。最后运行 shipping 示例，统计 run.json 里每个 Trial 实际留下了几个 Attempt。

预期输出与答案

基础设施第一次失败、第二次成功时，你会看到 2 个 Attempt，其中第二个是 canonical，Trial 为 completed。产品失败时，只有 1 个 succeeded/canonical Attempt，Trial 仍是 completed，同时记录 product_failed=true。如果基础设施用完所有恢复机会仍然失败，Attempt 数会等于预算，没有任何一个是 canonical，Trial 为 blocked。shipping 正常运行时，每个 Trial 都只有一个 Attempt，即使 buggy 最后有一个 Score 判为失败也不会增加。

常见误解

如果认为「只要最后成功，前面的 Attempt 就可以删掉」，你会一并丢掉可靠性和成本证据。把更多 Attempt 当成更多样本，则会制造伪重复。直接给 blocked 记 0 分看似保守，其实是把平台故障算到了产品头上。如果再规定「模型为了自洽而重试也必须拆成 Attempt」，Target 的内部策略和 Harness 的基础设施恢复就又混在了一起。

如何核对

先读 models.py，看各层状态怎样枚举，再运行 eval-harness-ref inspect，分别查看计划里有多少 Trial、最后生成多少 Observation Bundle。随后检查 Metric，确认它从 Trial 清单取得 denominator，而没有拿 Bundle 数量充当分母。上游 SWE-bench 的 infra_failure.py 还提供了另一处源码证据，说明环境型评测会单独处理基础设施失败。

与其他 Harness 的关系

benchmark 型 Harness 常把请求重试藏进 Model Adapter，声明式应用评测可能直接把每个 Test Case 称作一条结果，Agent 环境 Harness 则通常会完整管理 Trial 从创建到结束的过程。比较这些实现时，你要追踪「统计分母何时确定」「产品错误能不能重试」以及「哪个结果会进入评分」。如果只对齐 retry 这个配置名，找不到真正的答案。

本篇不能证明什么

把这三层分清，并不会自动算出最佳重试预算，更不能保证分布式执行恰好一次。它能证明的范围很具体：在已经声明的本地协议内，Harness 恢复基础设施时没有暗中增加成功样本，也没有删掉失败的 Trial。

上一章 · 下一章

04 | Trace、Artifact 与 Observation：让分数能回到证据

上一章 · 下一章

本篇要解决什么问题

报告里就算写着「通过 92%」，你也没法立刻核对这个结果，因为还得查清每个 Score 来自哪次执行、看过哪些事件和文件，以及评分之后这些文件有没有被改写。Trace（轨迹）负责串起事件，Artifact（产物）负责保存大对象，两者先把执行证据留住。

到了评分前，Observation Bundle（观测包）再冻结 Scorer 能读取的内容，让你从一个分数一路查回当时真正观察到的证据。

学完你能解释什么

- 事件流、文件产物和评分输入为什么要分开；
- Trace 的父子关系和序号怎样表达因果，而不仅是时间戳；
- 内容摘要怎样检测 Artifact 被替换；
- 为什么 Score 必须绑定 Observation Bundle 和 canonical Attempt。

贯穿案例

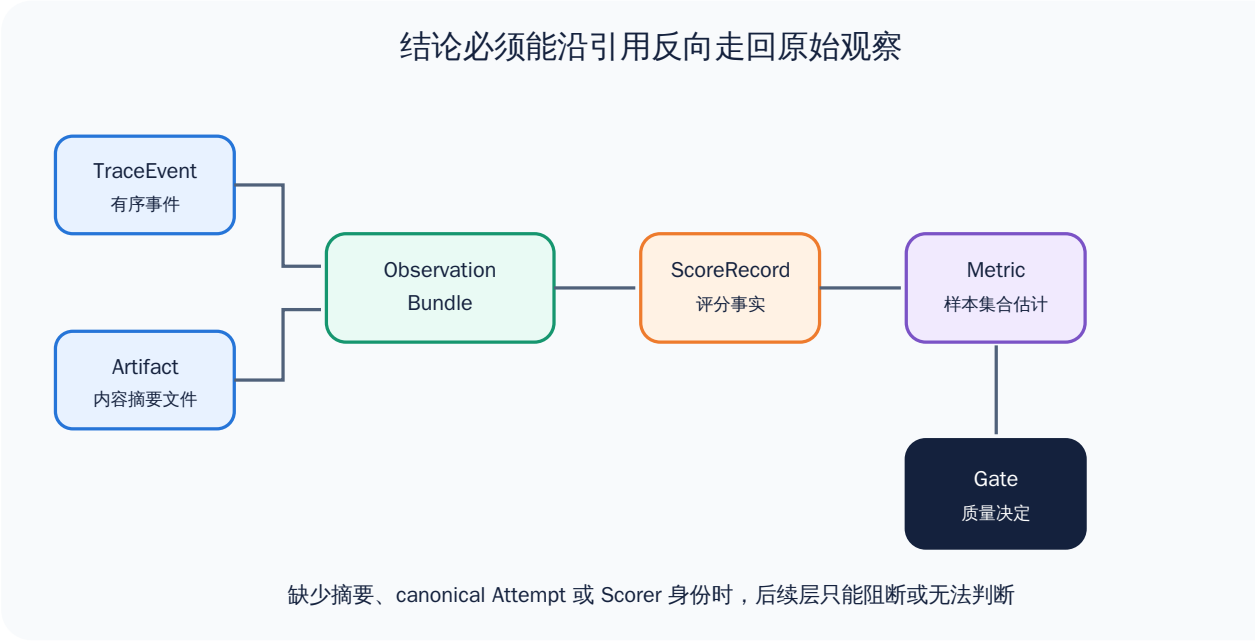
shipping Target 返回 {"fee": 10} 时，Runner 先后写下 Trial 开始和 Target 完成两个事件，在后一个事件的 payload 里留下 output 与 expected，再把原始输出按内容地址存成 Artifact。Bundle 随后把这些事件、Artifact 摘要和 canonical Attempt 绑在一起，Scorer 只读这份 Bundle，不再调用 Target，因此以后执行 score 时不必重跑脚本，也能按同一份证据得到结果。

核心概念与边界

TraceEvent 记下运行中能够排序并相互连接的事实，至少要带稳定的 event_id、sequence、type、parent_event_id 和 payload。事件里不适合直接塞入的字节内容，例如 Diff、日志、截图、测试报告或环境终态，则交给 Artifact 保存，其他对象通过 SHA-256 摘要和相对路径引用它。到了评分前，Observation Bundle 再把允许 Scorer 读取的事件和产物冻结下来，并明确它们属于这个 Trial 的哪一个 canonical Attempt。

Trace 不包括隐藏思维链，因此 Eval Harness 应该记录外部看得见的模型消息、工具调用、状态变化和输出，却不能要求模型交出内部推理，更不能假装自己拿到了。Artifact 也有边界，附件只有带上摘要、类型和来源关系，才能进入评分。

机制图



调用链与状态变化

1. Runner 为 canonical Attempt 产生有序事件；TraceWriter 拒绝重复 event_id 和先于父事件出现的子事件。
2. 大对象进入 ArtifactStore，相同字节得到相同摘要并复用同一个内容地址。
3. Bundle Builder 验证 Trial 只有一个 canonical Attempt，再把事件和 ArtifactRef 规范化哈希。
4. Scorer 读取 Bundle，ScoreRecord 保存 Bundle digest、canonical_attempt_id 和 scorer_id。
5. Metric 保存参与聚合的 score_ids，Gate 保存 metric_ids；报告因此可以从决定逐层反查。

Reference Harness 分别在 tracing.py、artifacts.py 和 reporting.py 实现这三段。

关键数据结构

```
TraceEvent(event_id, sequence, type, parent_event_id, payload)
ArtifactRef(kind, digest, relative_path)
ObservationBundle(bundle_id, digest, trial_id,
                  canonical_attempt_id, events, artifacts)
ScoreRecord(..., observation_bundle_digest, scorer_id)
```

时间戳可以拿来分析性能，却不能代替 sequence 和 parent 关系，因为不同机器的时钟可能并不一致。relative_path 让报告换到另一台机器后仍能找到文件。如果公开文档和证据写的是本机绝对路径，别人既难复现，还可能从中看到用户名。digest 则必须覆盖 Scorer 真正读到的内容，只对文件名做哈希，发现不了文件内容已经被换掉。

设计取舍

JSONL 很适合逐条追加事件，也方便逐行恢复和用命令行检查，不过它不会替你验证跨事件约束。把完整输出同时放进事件和 Artifact 会多存一份数据，但教学实现借此既能展示便于阅读的 payload，也能演示怎样按内容寻址。生产系统可以只在事件里保存 ArtifactRef，不过必须把 Scorer 实际读取的内容算进 Bundle digest。

Inspect AI 锁定提交里的 EvalLog 怎样组织 Eval Log，属于上游源码事实。OpenAI Evals 的 RecorderBase 和 Event 怎样划定事件记录的范围，同样属于上游源码事实。本篇把这两套做法放到一条证据血缘里讲，才是我们的机制解

释。

失败语义

子事件引用尚未出现的父事件：Trace invalid。

Artifact 文件缺失或摘要不匹配：Bundle 不能评分；不能只相信报告里的旧值。

Trial 有零个或多个 canonical Attempt：Bundle 构造失败。

Scorer 需要字段但事件和 Artifact 都未提供：Score unscorable，不是 0。

Trace 包含秘密或不必要个人数据：属于安全失败，应在保存前过滤并阻断发布。

动手实验

先运行 shipping 示例，再执行：

```
eval-harness-ref inspect output/shipping
eval-harness-ref score output/shipping
```

打开 evidence.json 的第一个 Bundle，顺着 Artifact digest 找到对应的相对路径，然后复制文件内容并计算 SHA-256，看看结果是否与文件名一致。接着临时移动一个 Artifact，再运行 inspect，确认检查会因为找不到证据而失败。实验结束后记得把文件放回原位。

预期输出与答案

正常情况下，inspect 会报告 6 个 Trial、6 个 Bundle 和 6 条 Score，每个 Artifact 文件名都应等于 sha256: 后面的 64 位十六进制摘要。只要少了一个已经被引用的 Artifact，inspect 就应以非零状态退出并显示「运行证据无效」，不能再把原来的 Gate 打印出来。score 则从冻结的 Bundle 重算 6 条评分，不会产生新的 Attempt。

常见误解

「有日志就能审计」忽略了身份和因果关系，「数据库记录不会变，所以不用摘要」也没把管理员操作、迁移和导出算进去。如果只拿最终回答评分，你就看不到工具造成的副作用和环境终态。反过来，把全部内部推理存下来也换不来可靠透明，只会带来隐私、安全和来源无法核对等新问题。

如何核对

运行 python -m pytest tests/test_lineage.py tests/test_cli.py -q，看看测试有没有覆盖父子顺序、事件去重、Artifact 去重、canonical Attempt 绑定和文件缺失。然后从 report.json 的 Gate 找到 metric_id，再从 Metric 找 score_ids，最后从 Score 找 observation_bundle_digest，确认你确实能沿这条链查回去。

与其他 Harness 的关系

Promptfoo 常围绕测试结果和 Trace 类型来组织应用评测，Inspect AI 用 Eval Log 更完整地保存样本与事件，Agent Environment Harness 还得留下容器日志、补丁和终态。它们写得详略不同，但都绕不开三个问题：Scorer 当时究竟看见了什么，这些证据属于哪次运行，评分以后内容有没有变化。

本篇不能证明什么

完整血缘能帮你核对报告是否忠实反映已保存的证据，也能在评分规则改动后确认重算读的仍是同一份冻结内容，而且引用没有在重算途中漂移。不过，它证明不了 Dataset 选得正确、Reference 没有偏差，也证明不了环境里不存在尚未记录的外部副作用。证据链回答的是「结论从哪里来」，至于「问题问得对不对」，还得回到评测设计里找答案。

上一章 · 下一章

05 | Scorer、Judge、Score 与 Metric：先判单次，再描述总体

上一章 · 下一章

本篇要解决什么问题

评测项目常用「指标」同时指三件事：判断一条回答、计算一组平均值，以及决定能不能发布。这样一来，Judge 的一次主观判断很容易被当成总体质量，缺失的输出也可能悄悄变成 0。要把这条链路理清，你得分开看谁负责评分、单个 Trial 得到了什么结果，以及多个 Trial 合起来说明什么。

学完你能解释什么

- Scorer、LLM-as-Judge、ScoreRecord 和 MetricEstimate 的责任差异；
- 为什么确定性证据应优先于模型自评；
- failed、uncertain、unscorable 与 invalid 为什么不能都压成 0；
- 为什么 Metric 的分母来自 Trial Plan，而不是成功返回的 Score 数。

贯穿案例

shipping 的 Scorer（评分器）从 Observation 里读出 fee，再与 Sample 预先写好的期望值比较，相等就记 passed/1，不相等就记 failed/0。规则只看留下来的证据，所以 Target 背后跑的是脚本还是模型，都不会改变判法。聚合三个 Sample 时，Metric 的分母始终是计划中的三个 Trial，即使某个 Bundle 没有 fee，对应 Score 也只能记为 unscorable，Gate 会因为缺证而无法判断，却不能顺手把分母减成二。

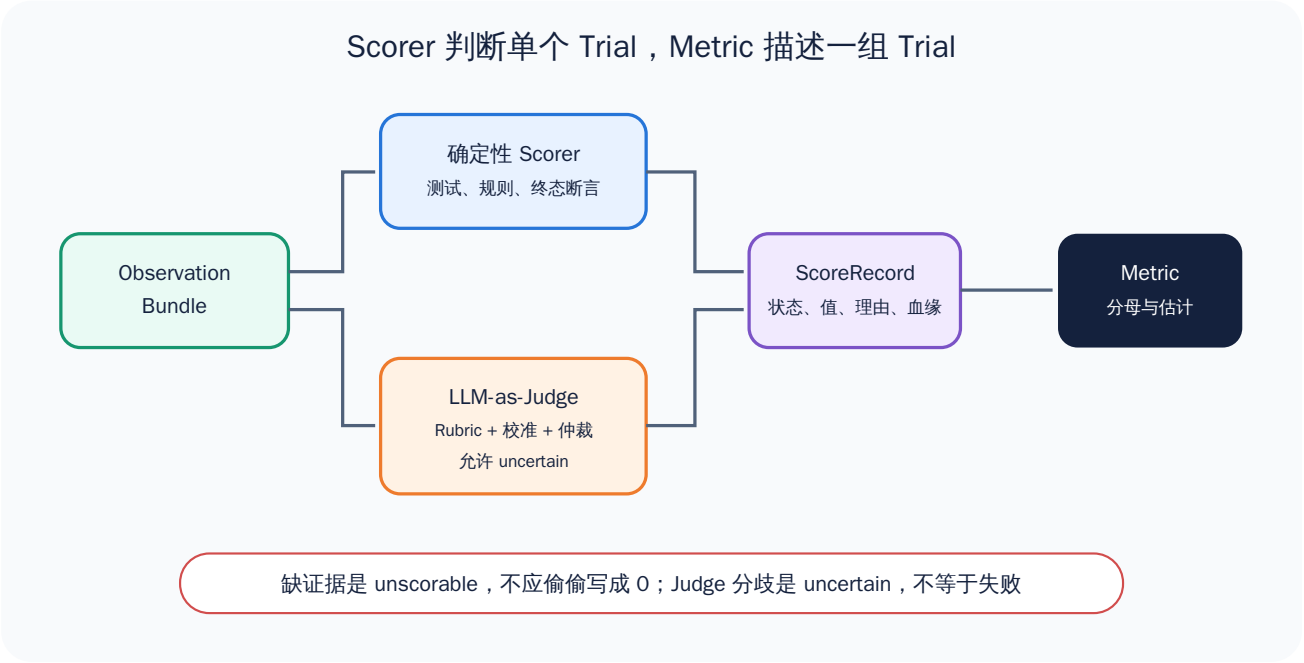
核心概念与边界

Scorer 是可以单独版本化的函数，它读取 Observation Bundle，再产出 ScoreRecord（评分记录）。LLM-as-Judge 只是在 Scorer 里面调用模型，并没有跳出评分流程另占一层。

模型的判断会随提示词和采样变化，因此你还得准备 Rubric（评分标准）、采样设置和校准集，并处理偏差与分歧。ScoreRecord 记下单个 Trial 已经发生的评分，包括状态、值、理由、Scorer 身份和证据血缘。等每次评分都落下来以后，MetricEstimate（指标估计值）才聚合预先声明的 Trial/Score 集合，同时保存分子、分母、估计值和 score_ids。

如果能直接检查环境事实，测试、规则和终态断言通常比模型自评更可靠。Judge 更适合判断没有唯一答案的文本质量、语义是否符合要求，以及必须参照 Rubric 才能下结论的结果，不过它仍要允许输出 uncertain，并持续拿人工标注或确定性证据做校准。模型自己说「我完成了」，只算 Target 的一段输出，不能拿来给自己作证。

机制图



调用链与状态变化

1. Scorer 从 Bundle 中寻找被协议允许的观察字段，不直接重跑 Target。
2. 字段存在且满足规则时生成 passed Score；存在但不满足时生成 failed Score。
3. 关键字段缺失时生成 unscorable；协议或血缘破坏时应生成 invalid；Judge 无法稳定裁决时生成 uncertain。
4. Aggregator 以计划 Trial ID 为分母，验证 Score 没有引用计划外 Trial，也没有重复 score_id。
5. Metric 保存 numerator、denominator、value 和参与的 score_ids，供比较与 Gate 使用。

本仓库的 FieldMatchesExpectedScorer 与 aggregate_pass_rate 是最小可运行实现。

关键数据结构

层	关键字段	不能缺少的身份
Scorer 配置	field、Rubric、阈值、采样参数	scorer_id 与版本
ScoreRecord	status、value、reason	Trial、Attempt、Bundle、Scorer
MetricEstimate	numerator、denominator、value	metric_id 与 score_ids

值和状态必须一起保存，因为它们回答的是两个问题：规则算出了多少，以及这次评分能不能用。value=0 可以表示规则明确判错，却说不出证据缺失或协议已经失效。Gate 得先看状态，确认这个数能不能继续拿来推断。reason 只方便人来核对，替代不了结构化状态。

设计取舍

二元规则简单，也容易复现，但有些质量很难只用通过或失败说清楚。连续分数能拉开更细的差别，可它究竟代表什么，又会受到阈值、尺度和校准方式影响。你可以让多个 Scorer 分别检查正确性、安全和格式，再交给非补偿 Gate 单独拦住关键风险，否则一次安全失败很可能被较高的正确率平均掉。Judge 能省下一部分人工，仍然要冻结模型、prompt、Rubric 和随机设置，并定期拿人工样本对照。

DeepEval 的 BaseMetric 怎样抽象 Metric/Judge，属于上游源码事实。Promptfoo 的 runAssertion 怎样分派 Assertion，也是上游源码事实。这里把两者都放到 Scorer 的职责下理解，属于我们的机制解释，不代表它们使用相同的状态枚举。

失败语义

可观察值与 Reference 不符：failed，可计入明确失败。

需要字段没有采集：unscorable，必须暴露缺证问题。

Judge 两次判断冲突且无仲裁：uncertain，不能挑有利一次。

Scorer 版本或 Bundle digest 不匹配：invalid，整个 Score 不应聚合。

计划 100 个 Trial 只产生 90 个 Score：Metric 分母仍为 100，并单独报告缺失。

动手实验

运行 shipping 示例并打开 evidence.json，复制一份文件，从其中一个 target_completed 事件里删掉 output，然后在测试代码里让 FieldMatchesExpectedScorer 给这个 Bundle 评分。接着保留完整的 3 个计划 Trial，只提供其中 2 条 passed Score，再调用 aggregate_pass_rate。

预期输出与答案

Bundle 少了字段时，Scorer 应产出 unscorable，并把 value 设为 null，不能默认判成 failed。两条 passed Score 对应三个计划 Trial 时，Metric 应当算成 2/3，不能算 2/2。把 unscorable Score 一并交给 Gate 后，结果应是 inconclusive，因为现有证据既撑不起通过，也不足以确认失败。

常见误解

如果你认为「Judge 给了数字就很客观」，就会漏掉校准过程和模型偏差。把 unscorable 记成 0 看起来保守，却会把产品质量问题和 Harness 自己没采到证据混在一起。只盯着平均分还会遮住关键风险和分组差异，直接比较两个同名 metric 也一样危险，因为数据、Scorer 或分母的定义可能早已变了。

如何核对

运行 python -m pytest tests/test_scoring.py tests/test_metrics.py tests/test_gates.py -q，重点检查缺字段时记什么状态、聚合时用哪个分母、重复 score_id 是否被拦下，以及无效证据能不能通过 Gate。随后阅读锁定的 DeepEval 实现，判断上游所说的 Metric 在哪些情况下既负责本篇的 Scorer 工作，又负责汇总 Score。

与其他 Harness 的关系

在 lm-evaluation-harness 里，Task 通常既构造请求，也聚合 metric。Inspect AI 会明确露出 Scorer 和 Metric，Promptfoo 则让 Assertion 面向测试配置，DeepEval 又用 Metric 类判断单个案例，其中也包括 Judge。名字相同，所处的层级未必相同，所以比较这些工具时，你得继续问一句：它接收的是一条 Observation，还是一组 Score？

本篇不能证明什么

评分血缘完整、状态也记对了，仍然证明不了 Rubric 有效、Judge 没有偏差，或者 Dataset 能代表真实线上流量。高分是否覆盖了必须守住的业务风险，也要由领域专家继续复核评分目标。这套链路能做的事情很具体：不让互不兼容的证据悄悄挤成一个看起来很精确的数字。

上一章 · 下一章

06 | 不确定性、比较与 Gate：从分数到可执行决定

上一章 · 下一章

本篇要解决什么问题

Candidate（候选版本）得分 82%，Baseline（基线版本）得分 80%，这就足以支持发布吗？两个点估计回答不了，因为你还不知道双方测的是不是同一批样本，也不知道差异是否只由少数样本拉开。重复运行之间可能彼此相关，证据可能缺失，关键安全检查甚至可能已经失败，而且有人还可能看完结果才临时挑一个判定阈值。必须把统计比较和决策政策分开，否则一个百分比就会把这些问题全遮住。

学完你能解释什么

为什么同一 Sample 上的 Baseline/Candidate 应保持配对；

Bootstrap 区间表达什么、不表达什么；

Metric、ComparisonResult 与 GateDecision 的区别；

为什么 Gate 需要 passed、failed、blocked、inconclusive 四种结果。

贯穿案例

shipping 一共有三个 Sample，buggy 通过 2/3，fixed 通过 3/3，因此点估计相差 1/3。两个 Target 跑的是完全相同的 Sample 和 repetition，所以可以先算出每一对 Score 的差，再对这些差值重采样。这里只有三个样本，无论怎么算区间，都变不出真实业务里的确定性，不过配对至少不会把样本本身的难度差异错算成模型差异。发布 Gate 还会另行检查 fixed pass-rate 是否达到 1.0，并确认整次运行里没有 unscorable 证据。

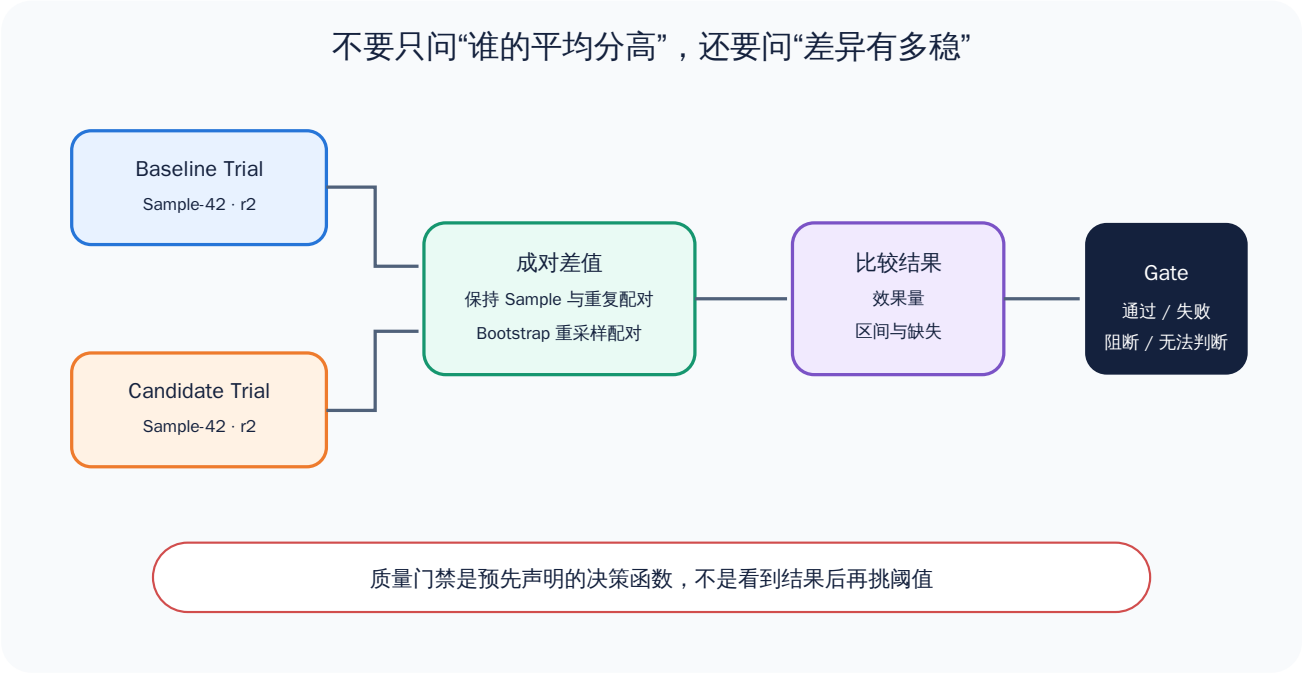
核心概念与边界

MetricEstimate（指标估计值）告诉你某个 Target 在一组计划 Trial 上测出了什么。ComparisonResult 再把 Candidate 和 Baseline 放到一起，看效果相差多少、方向如何，以及这个差异有多不确定。

预先声明的 GatePolicy 读取有效 Metric、比较结果和关键检查，最后由 GateDecision（门禁决策）记下这次运行究竟是通过、失败、被阻断还是无法判断，同时保存判定依据。

blocked 表示计划预先要求的条件还没满足，例如运行环境不可用，或者唯一的 Metric 没有生成。inconclusive 表示已经拿到运行证据，但证据仍不足以判断通过或失败，例如关键 Score 处于 uncertain。failed 说明有效证据明确达不到政策要求，passed 才说明所有声明过的条件都满足了。这四种结果会引向不同的后续动作，压成布尔值以后就分不清了。

机制图



调用链与状态变化

1. 依据 Sample ID 和 repetition key 连接 Baseline 与 Candidate Trial，保留无法配对的缺失记录。
2. 对每对 Score 计算差值；若一个实体产生多个相关样本，还需在实体或 cluster 层重采样。
3. 用固定 seed 的 paired Bootstrap 反复抽取配对，得到效果量分布和区间。
4. Gate 先检查证据可用性与关键非补偿条件，再读取 Metric 或 Comparison。
5. GateDecision 保存 metric_ids、状态与原因；Reporter 不得把它扩大为真实生产发布授权。

在 Reference Harness 里，paired_bootstrap 负责做统计比较，evaluate_gate 则负责执行政策，两层各管一件事。

关键数据结构

```
MetricEstimate(metric_id, numerator, denominator, value, score_ids)
ComparisonResult(baseline_id, candidate_id, mean_difference,
                 interval_low, interval_high, pairs, seed)
GateDecision(gate_id, status, metric_ids, reason)
```

计算区间时还要保存 iterations、按什么单位重采样，以及遇到缺失值怎么办，否则两个都叫「95% 区间」的结果，背后可能用的是完全不同的假设。GatePolicy 也得版本化，因为阈值、关键检查和例外流程如果只藏在 CI 脚本的一行 if 里，事后就很难查清某次决定到底遵循了哪套规则。

设计取舍

非配对比较写起来简单，但双方跑同一个 Dataset 时，这种做法会白白丢掉样本之间原有的对应关系。配对比较通常更容易看出差异，代价是你必须把双方身份逐一对齐，并认真处理每一项缺失。Bootstrap（自助法）能把复杂分布直观地摆出来，却不能让小样本凭空变得可靠，也修不好 Dataset 自带的偏差。质量 Gate 可以只检查绝对阈值，也可以再要求 Candidate「不劣于 Baseline」。关键安全风险通常要用非补偿规则拦住，因为再高的平均正确率也抵消不了这类失败。

lm-evaluation-harness 的 evaluator.py 和 evaluator_utils.py 怎样运行任务、整理结果，属于上游源码事实。本仓库则把发布 Gate 单独建成一层，这是面向质量工程的教学实现，不表示上游也提供了相同的 Gate API。

失败语义

Baseline 与 Candidate Sample 集不一致：比较 blocked，不能比较两个孤立平均值冒充配对。

区间跨越预声明的最小效果边界：结果 inconclusive，不等于 Candidate 一定更差。

关键安全 Score invalid/unscorable：Gate inconclusive 或 blocked，绝不能 passed。

有效 pass-rate 低于阈值：Gate failed。

看到 82% 后把阈值从 85% 改为 80%：政策污染，需新版本与独立运行。

动手实验

运行 python -m pytest tests/test_metrics.py tests/test_gates.py tests/test_scoring.py -q，再在 Python REPL 中对 [0, 0, 1] 这组三个配对差值调用 paired_bootstrap(seed=7) 两次。随后把 seed 改为 8，比较点估计与区间。

预期输出与答案

seed 和 iterations 相同时，两次运行应生成完全相同的结果，这只能证明算法回放稳定。如果输入、配对键或算法版本变了，即使碰巧算出同一个数字，也不能说两次分析是同一次。改变 seed 可能会改变有限 Bootstrap 样本给出的区间，但原始配对差的均值仍是 1/3。在 shipping 中，fixed Gate 应为 passed，buggy Gate 应为 failed。只要任一关键 Score 改成 unscorable，相关 Gate 就必须返回 inconclusive，不能撇开它，再用剩余样本算出 100%。

常见误解

如果看到「区间不重叠」就直接宣布可以发布，你会漏掉政策约束和多重比较。增加样本只能帮你控制一部分随机方差，不会自动洗掉数据里的系统偏差。较大的 p 值也说明不了两者相同，因为没能拒绝差异，并不等于已经证明等价。Gate failed 是有效证据支持的负面结论，程序执行故障则是另一回事。

如何核对

先看 tests/test_metrics.py 有没有锁住计划分母，再看 tests/test_gates.py 能不能拦住无效证据。你还可以从 report.json 手工取出 Score，重算 numerator/denominator，然后核对 Gate 给出的原因，确认里面的阈值与配置一致。

与其他 Harness 的关系

Promptfoo 常把评测接进 CI，再用阈值下判断。DeepEval 更适合用测试断言做回归，benchmark Harness 关心怎样聚合结果和生成排行榜，平台型产品则可能提供实验比较。界面可以各不相同，但你始终要能分开复核统计估计和政策决定，并看清哪些证据缺了、哪些结果彼此依赖。

本篇不能证明什么

在本地跑一次 Bootstrap，再加一道阈值 Gate，还算不上完整的统计审查，更不会自动获得真实生产发布的授权。这里展示的只是一个最小闭环：从预先声明 Trial 开始，把 Baseline 和 Candidate 的 Score 配好对，最后生成 Decision，把结论和依据都写清楚，也让每个状态都能查回证据。分布漂移、多重检验和长期风险预算还没有处理。你仍要拿新的运行证据检验外部有效性，并在这套系统之外完成风险接受和组织审批。

上一章 · 下一章

07 | Eval-to-RL 与独立 Release Eval：改进闭环不能吃掉验收集

上一章 · 下一章

本篇要解决什么问题

评测找出失败样本后，人们往往会拿这些样本生成偏好对、验证器奖励或在线强化学习任务。可同一批样本和 Scorer 如果既用于训练和选择 checkpoint，又用来证明发布质量，模型很可能只是学会了怎样通过一道已知门禁。这就污染了验收证据。Eval-to-RL 的关键不是把分数接上训练 API，而是把三类决定所依据的证据隔开：训练 reward、checkpoint 选择和独立 Release Eval（发布评测）。

学完你能解释什么

- Evaluator、RewardAdapter、DPO/GRPO/RFT 和独立发布评测怎样形成完整链路；
- 为什么并非所有 Score 都能直接变成 reward；
- 开发集、训练反馈集、选择集和独立留出集怎样分工；
- 为什么改进循环可以快，而发布 Gate 必须保持独立。

贯穿案例

shipping 里金额 100 的失败可以收进训练难例。做 DPO（直接偏好优化）时，你可以构造一个偏好对，让「免运费」这条正确输出优于「收 10 元」的错误输出。

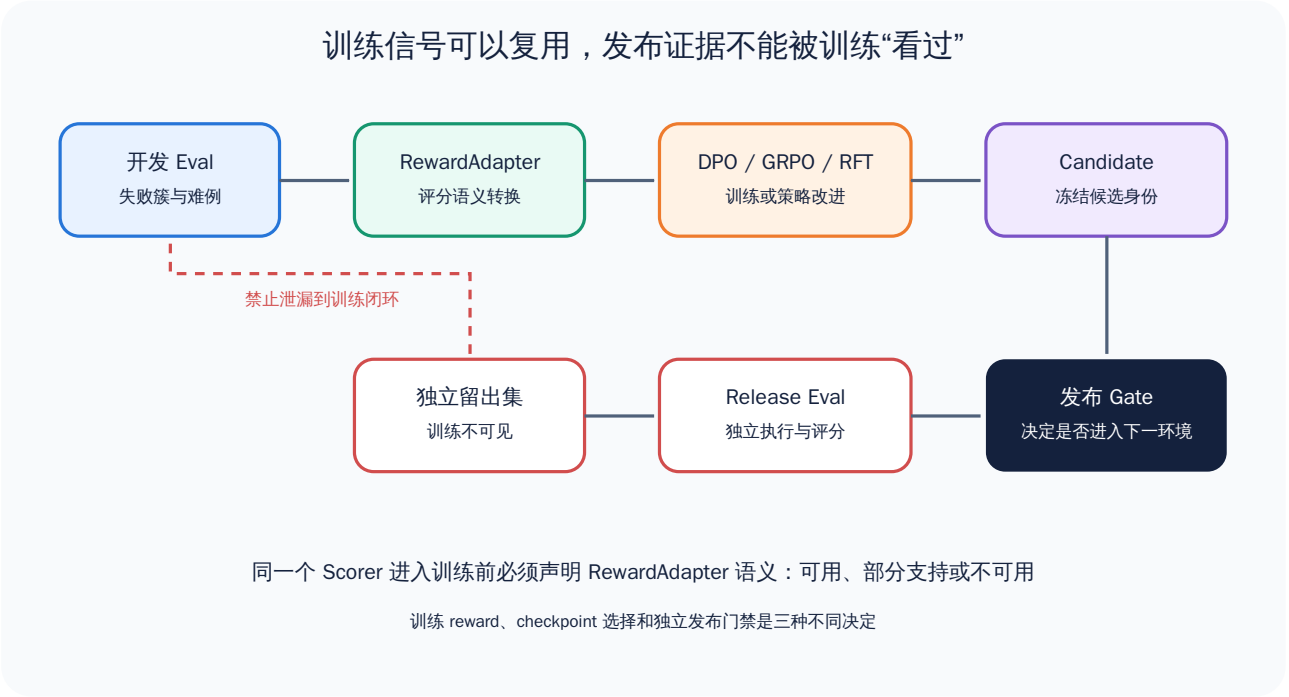
做 GRPO（组相对策略优化）或 RFT（强化微调）时，则可以让确定性规则给出 0/1 reward。发布评测不能再只考 99、100、101 这三个已经被优化器反复看过的样本，还得冻结没有参与训练的边界组合、数据类型和异常路径，让 Candidate 在一次全新的运行里独立接受同一条业务规则检查。

核心概念与边界

Evaluator（评测器）产出带血缘的 Score，并把失败样本聚成簇。随后，RewardAdapter（奖励适配器）按照明确规则，把评测结果翻译成训练接口能接收的标量 reward、偏好对或验证器结果，实在不能转换就标记为不可用。DPO 读取偏好数据，GRPO/RFT 则常用可验证轨迹或模型生成轨迹上的奖励。训练期间可以运行 checkpoint choice eval 来挑选候选，并允许结果影响后续优化。等候选身份冻结以后，independent release eval 才在隔离的留出集上另跑一次，判断它能不能进入下一个环境。

RewardAdapter 必须明说自己能转换什么、不能转换什么。确定性测试通过后，通常可以安全地映射成相应的验证器 reward，但 Judge 给出的 uncertain 不能硬压成 0/1，需要跨样本聚合的公平性 Metric 也不能无缘无故摊到每条轨迹上。这两类结果不能混。有些 Score 还带着人工标注时产生的隐私数据，因此不能进入训练。完全表达不了就写 unavailable，只能保留部分含义就写 partial。

机制图



调用链与状态变化

1. 开发 Eval 保存 Trial、Observation、Score 和失败原因；Failure Miner 按错误机制聚类，而不是只挑最低分。
2. RewardAdapter 检查 Scorer 身份、证据许可、粒度和不确定状态，输出偏好、标量 reward 或拒绝转换。
3. DPO/GRPO/RFT 使用这些数据更新模型或策略——训练日志与 reward 版本被保存，但不成为发布证据。
4. Checkpoint Eval 在开发/选择 Split 上比较多个候选，选出一个冻结 Candidate 身份。
5. 独立 Release Eval 在训练不可见的留出集和干净环境重跑 Candidate，对照 Baseline 与 Gate Policy。
6. 只有独立 Gate 的证据进入发布报告；生产事件随后可以回流为下一轮开发数据，但不能改写历史 Gate。

关键数据结构

对象	需要的字段	主要风险
RewardAdapterContract	scorer_id、输出类型、状态映射、限制	语义错误转换
PreferencePair	prompt、chosen、rejected、来源 Score	偏好泄漏或伪标签
VerifierReward	Trial/Attempt/Bundle、reward、verifier 版本	奖励黑客
SplitManifest	entity group、用途、版本、授权	样本或实体泄漏
ReleaseEvalSpec	Candidate 身份、独立 Dataset、Gate	训练后调阈值

设计取舍

训练和发布复用同一个 Scorer，确实能让目标保持一致，却也会放大奖励黑客的风险，因此 Release Eval 还应加入独立的检查角度、不同实现或人工抽查。把留出集完全藏起来能减少泄漏，但维护更贵、反馈也更慢。代价很清楚。实践中通常让开发集承担快速迭代，再定期更新独立集。自动 reward 方便扩大训练数据，人工偏好能够表达更细腻的语义，不过两者都得留下来源和授权记录。

OpenAI Evals 锁定提交里的 eval.py 怎样抽象样本 Eval，属于上游源码事实。本仓库再让 RewardAdapter 把 Score 接到 DPO/GRPO/RFT，这一步属于架构机制解释，因为 OpenAI Evals 的这个文件并没有实现这里描述的训练适配器。

失败语义

Judge uncertain 被强制转成负 reward：Adapter 语义无效，应拒绝导出。

同一客户的近重复样本跨训练与留出集：实体泄漏，Release Eval invalid。

训练循环根据留出结果多次调参：留出集转化为开发集，必须更换独立集。

checkpoint 在开发集提升、独立 Gate 失败：有效的负面发布结论，不得用训练 reward 覆盖。

Reward 很高但真实终态不满足：奖励黑客，需要环境事实 Scorer 和新回归样本。

动手实验

从 shipping 的 report.json 里挑出 failed Score，再顺着 observation_bundle_digest 找到金额 100 时的输出和 expected。然后画一张表，分别判断这条 Score 能不能转成 DPO 偏好对、GRPO 标量 reward 和独立发布 Score，最后把一条 unscorable Score 也放进来比较。

预期输出与答案

金额 100 的样本错得很明确，而且带有确定性 Reference，因此可以生成 chosen=fee:0、rejected=fee:10，也可以映射成 0 reward。不管最后选择哪种训练数据形式，都必须完整留下它从哪条 Score 转换而来。这个样本既然已经进入训练，就不能再给独立发布评测作证。unscorable 同样不能直接映射成失败 reward。这时不能判错。应该先把它标记 unavailable，修好观测链。独立发布 Score 必须来自一个新的 Trial，由冻结的 Candidate 在从未参与训练的 Dataset 上运行得到。

常见误解

把 Eval 分数直接当成 reward，会把评分粒度和状态都丢掉。只划分 train/test 也挡不住所有泄漏，因为同一实体的样本、近重复内容和反馈循环仍可能越过边界。Release Eval 做得晚，不等于天然独立，只要结果一直用来指导调参，这批数据实际上就成了开发集。它已经不再独立。RL 把 reward 提高，只能说明优化目标上的数字变了，不能直接证明产品在真实场景里更有效。

如何核对

随便挑一条训练样本，反查它原来的 Score、Bundle 和 Scorer 版本，并确认 RewardAdapter 给每一种状态都写了转换测试。还要检查 SplitManifest 是否按实体分组，确认独立集没有混进训练、prompt 选择或阈值调节。Reference Harness 目前只提供 Eval 证据，没有实现训练，这是特意守住的边界，并非遗漏。

与其他 Harness 的关系

主流 Eval Harness 通常负责运行评测、打分或接入 CI，训练框架则盯着怎样优化模型。两类系统接起来时，明确写出的 Adapter 比共享一个数据文件更重要，因为 Adapter 会告诉你每种评分含义究竟怎样变成训练信号。Agent Environment Harness 还能提供验证终态后得到的 reward，LLM-as-Judge Harness 则更依赖校准和人工仲裁。工具可以自由组合，但独立 Release Eval 必须保留只用于发布判断的 Dataset、运行记录和 Gate 血缘。

本篇不能证明什么

本篇没有实现 DPO、GRPO 或 RFT 训练，也没有声称哪种算法必然提升真实质量。这里规定的是怎样隔离证据：训练信号可以来自 Eval，但用于发布的结论必须来自训练之外的一次独立评测。这条线不能越过。

上一章 · 下一章

lm-evaluation-harness 源码课程：Task 怎样变成批量模型请求

上一节 · 下一节

本篇要解决什么问题

lm-evaluation-harness 常被当成「跑 benchmark 的命令行工具」，可源码更有意思的地方在于，它能把不同 Dataset、prompt 写法和 metric 转成少数几类模型请求。只看 CLI 参数，你就看不到 Task 何时把一份文档拆成多条请求，也不会明白 Model Adapter（模型适配器）为什么无需理解具体 benchmark，更难跟清每条样本的值怎样重新组合并汇成任务结果。

课程以锁定提交 ffb2f7b0dfbb05a8095b04947a15cc0a70d54c66 为准，不跟着浮动的 main 变化。阅读时会依次经过装配、构造请求、执行并回填、处理样本和聚合结果五个环节，同时把它与本仓库 Trial/Attempt/Observation 对不上的地方明确标出来。

先建立源码地图

站点	锁定文件	主要责任
高层入口	evaluator.py	simple_evaluate 装配模型、Task、缓存与运行配置
Task 抽象	api/task.py	文档、上下文、Instance、过滤、逐样本 metric 与聚合函数
Model Adapter	api/model.py	loglikelihood、rolling likelihood、generation 与缓存代理
请求对象	api/instance.py	连接文档、请求参数、重复响应和过滤响应
聚合	evaluator_utils.py	Task/Group metric、stderr、配置和样本数整理

表中内容都能从上游源码直接核对，所以属于上游源码事实。课程若进一步把它们映射成「Planner、Target Adapter、Scorer、Metric」，给出的就是机制解释，因为上游并没有采用 Reference Harness 的全套对象和状态枚举。

完整调用链

1. CLI 最终调用 simple_evaluate；若 model 是字符串，Registry 创建 LM 子类，若启用缓存，再用 CachingLM 包装。
2. TaskManager.load(tasks) 返回 tasks、groups 等结构，入口覆盖 generation kwargs、few-shot 数和随机种子，并可运行 Task 完整性测试。
3. simple_evaluate 调用较低层 evaluate(lm, task_dict, ...)。后者为每个 Task 调用 build_all_requests，把评测文档变成 Instance。
4. evaluate 按 Instance.request_type 分桶，复制 repeats，必要时为分布式 rank 补齐，然后通过 getattr(lm, reqtype) 批量调用 Model Adapter。
5. 响应按顺序追加到 Instance.resps；Task 运行 filter 后，源码按 doc_id 重新分组、按 idx 排序，把过滤响应交回 task.process_results。
6. 每个文档得到的 metric 值进入 raw_metrics[(metric, filter)]；_process_results 先做 Task 聚合和 stderr，再自底向上做 Group 聚合。
7. 高层入口补充模型、batch、device、seed、git hash、日期、环境与 tokenizer 信息，返回结果字典。

关键数据结构

要跟懂整条链路，先得认清 Instance：它既不代表 Dataset 里的一条 Sample，也不代表单独运行一次进程，只是交给 Model Adapter 的一条请求。

```
Instance
  request_type     loglikelihood / loglikelihood_rolling /
                   generate_until / multiple_choice
  doc              原 Task 文档
  arguments        交给 LM 方法的参数
  task_name, doc_id, idx
  repeats          同一请求需要几次响应
  resps            原始响应
  filtered_resps   filter 后交给 process_results 的响应
```

一份多项选择文档可以生成多条 loglikelihood Instance，它们共享 doc_id，再用不同 idx 表示各自在文档里的顺序，所以「请求数」完全可能比「样本数」大得多。基数必须分清。上游结果还会保存 doc/prompt/target hash，方

便你核对 sample log，但这些血缘字段不能与本仓库的 Observation Bundle（观测包）逐项对应。

实现取舍与失败语义

所有模型只要实现少数几种请求方法，Task 就能和具体后端分开，系统也更容易批处理或缓存响应，但复杂的 Agent 轨迹、工具副作用和环境终态很难自然塞进这些接口。Task 既要构造请求，又要运行 filter、算出每条样本的 metric，再声明 aggregation，扩展点虽然集中，承担的责任也更重。如果自定义 metric 没写 aggregation，源码会先警告再 fallback 到 mean，这只是为了让运行别立刻停下，不能证明 mean 真符合这个 metric 的统计含义。

Task 如果标着 unsafe code，却没有得到明确确认，evaluate 会拒绝运行。多模态 Task 遇上不支持多模态的 LM，系统同样会提前失败，而各个分布式 rank 的请求数不相等时，代码则会补齐调用。Model 请求出错后怎样恢复，主要由具体 Adapter 决定，因为核心 evaluate 里的 repeats 只用来采样或取得多次响应，不能直接算成基础设施 Attempt。这不是同一种重试。

动手实验

不下载模型也能验证这条链路怎样变换对象：打开锁定源码里的 tests/test_aggregation_pipeline.py，找到一项包含 4 个 raw acc 值的测试，手工算出 Task 和单 Task Group（任务组）的结果，然后在本仓库运行下面两条命令。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

第二条命令不会运行上游模型，它只会确认课程里的源码链接都落在锁定范围内，并检查正文和图示是否满足学习合同。

预期输出与答案

[1, 0, 1, 0] 做 mean 后得到 0.5，单 Task Group 的结果仍是 0.5，此时 sample_len 为 4。一份文档若有四个候选选项，通常会产生四条 loglikelihood Instance，最后却只得到一组文档层 metric，所以请求数绝不能直接充当准确率的分母。

sources.py verify 应该报告 8 个来源锁文件都有效，课程测试也应该通过。只要上游链接用了 main 而没有写 40 位 commit，或链接路径超出锁定 scope，测试就该失败。

如何核对

先在 evaluator.py 搜索 build_all_requests，沿着调用继续追到 getattr(lm, reqtype)、req.resps.append、process_results 和 _process_results，然后再读 api/instance.py，逐项验证 doc_id、idx、repeats 与 resps 各自起什么作用。不要只引用 README 里的概念描述。

本篇不能证明什么

锁定调用链只能解释这个提交怎样组织 benchmark 型评测，不能证明某个 benchmark 能代表真实业务，也不能证明模型排名可靠，或上游同样适合评测 Agent 环境。数据是否有效、Adapter 怎样从网络错误中恢复、发布 Gate 又该采用什么政策，都要另行审查。

上一节 · 下一节

01 | 入口与 Task 加载：simple_evaluate 实际装配了什么

上一节 · 下一节

本篇要解决什么问题

很多调用示例只写一行 simple_evaluate(model=..., tasks=...)，看起来像是它亲自跑完整个执行循环，其实源码先让它解析模型、加上缓存、加载 Task 和 Group（任务组）、改写 Task 配置并记录实验身份，随后才把真正的执行交给 evaluate。你得先分清这两个入口各管哪一段，才能判断某项配置是在运行前就定好了，还是会跟着样本循环继续变化。

先建立源码地图

主要证据在锁定的 evaluator.py 里，simple_evaluate 会接收模型字符串或已经初始化的 LM，还会收下 Task 名称、few-shot、batch、device、cache、seed、chat template 和日志选项。它随后调用外部的 TaskManager.load，不过这篇先不展开 Manager 怎样扫描资源，只看加载出来的对象如何进入核心循环。

api/model.py 定义了 Model Adapter（模型适配器）这一层，LM 规定模型必须提供哪些请求方法以及怎样处理 rank/world_size 和分布式原语，CachingLM 则代理其中几种请求，并按参数哈希缓存响应。Task 从 Task 基类 展开，请求也由 Task 这一侧来建，入口是 build_all_requests()。

完整调用链

1. simple_evaluate 设置 Python、NumPy、Torch 和 few-shot 随机种子，而这些种子用途不同，不能只保存一个 seed 字段。
2. 模型参数若是字符串，Registry 找到 Model 类并调用 create_from_arg_string；传入对象则必须是 LM 子类，否则抛出 TypeError。

3 开启请求缓存时，以 rank 区分 SQLite 文件，再用 CachingLM 包住真实模型。非确定性 generation 参数会影响是否缓存。

4 若调用者未传 TaskManager，入口创建一个；load(tasks) 返回任务、组和层级信息。

5 对每个 Task，入口覆盖 generation kwargs、predict-only metric、num_fewshot 和 few-shot seed。Task 显式配置 0-shot 时不会被命令参数强行改成别的值。

6 check_integrity 为真时执行 Task 测试；EvaluationTracker 存模型来源、参数、system instruction 和 chat template。

7 调用 evaluate 后，只有主 rank 构建最终结果，并补充实际模型信息、batch、device、各类 seed、git hash、日期、环境和 tokenizer 信息。

关键数据结构

TaskManager.load 返回的并非一个简单的 dict[str, Task]，结果里至少有 tasks 和 groups，每项 Task 还带着 output_type、few-shot、generation_kwargs、metric 与 filter 等配置。高层结果会在 config 中同时记下逻辑模型名和运行参数，如果 LM 还提供 get_model_info，代码就继续把更具体的模型信息合进来。

如果你想复现运行，只保存字符串 model="hf" 还不够，它只说明用了哪类 Adapter，真正的 checkpoint 多半藏在 model_args 里，服务端 API 甚至可能把同一个逻辑名字指向不同的实际版本。名字不够用。因此，本仓库只把这类配置算作 Target Identity 的一部分，不会拿它冒充完整的实际身份。

实现取舍与失败语义

高层入口集中处理兼容问题和便捷选项，evaluate 因而可以假定 LM 已经初始化、Task 也已经加载好。可配置是在对象加载以后才被覆盖的，所以你必须继续追到最终 Task config，不能看完 YAML 就停下。源码还把 predict-only 通过把 metric 改成 bypass 来实现的办法称为较 hacky，这其实是在提醒你，「只生成输出」和「产出可以解释的 Score」属于两种运行模式。若模型类型根本不满足 LM 协议，运行会在开始前直接失败。

Task 完整性测试只要失败，运行就会立即停下，最终结果也只由 rank 0 返回并写入文件，避免多个进程重复输出。缓存不代表身份。缓存文件按 rank 拆开后，确实能避免并发写入互相冲突，但这不能证明不同机器、不同 tokenizer 或浮动模型身份之间可以安全共用缓存。

动手实验

读一遍 simple_evaluate 的参数表，然后为 shipping Reference Harness 画一张映射表：把 model/model_args 对到 TargetSpec，把 tasks 对到 Task/Dataset 选择，把 limit 对到计划抽样，把 bootstrap_iters 对到统计配置，再把 use_cache 对到执行优化。每一项都要标明是「完全对应」「部分对应」还是「不等价」。

然后执行：

```
python scripts/sources.py links
```

检查输出里的 evaluator.py 链接，确认它带着锁定 commit，而没有指向会继续变化的 main。

预期输出与答案

model/model_args 只能部分对应 TargetSpec，因为系统可能要等运行结束后才能核对实际服务身份。tasks 也只能部分对应 Task/Dataset，毕竟 Task 对象同时装着 prompt 和 metric。limit 不能等跑完以后随手截结果，它应该在运行前就写进 Trial/Sample 计划，bootstrap_iters 则用来配置聚合时怎样估计不确定性，use_cache 只负责优化执行，不该改变逻辑输出。

链接命令应该逐项列出 evaluator.py、api/task.py、api/model.py、api/instance.py 和 evaluator_utils.py 的永久 URL，这几份文件合起来，正好圈出本篇要讲的源码范围。

如何核对

从 simple_evaluate() 开始，按执行顺序找到 create_from_arg_string、CachingLM、task_manager.load 和覆盖 Task 配置的代码，最后跟到 evaluate()。然后再进 api/model.py，核对缓存究竟代理了哪些请求类型，以及 generation 采用采样时为什么可能禁用缓存。

本篇不能证明什么

保存入口参数，确实比留一张命令行截图更能说明当时怎样运行，但它依然证明不了模型服务端的实际版本、Dataset 文件内容和网络环境完全相同。你还得用内容摘要、服务返回的实际身份和 Artifact（产物）血缘把证据补齐。

上一节 · 下一节

02 | 请求执行：Instance 怎样分桶、批量调用并回填

上一节 · 下一节

本篇要解决什么问题

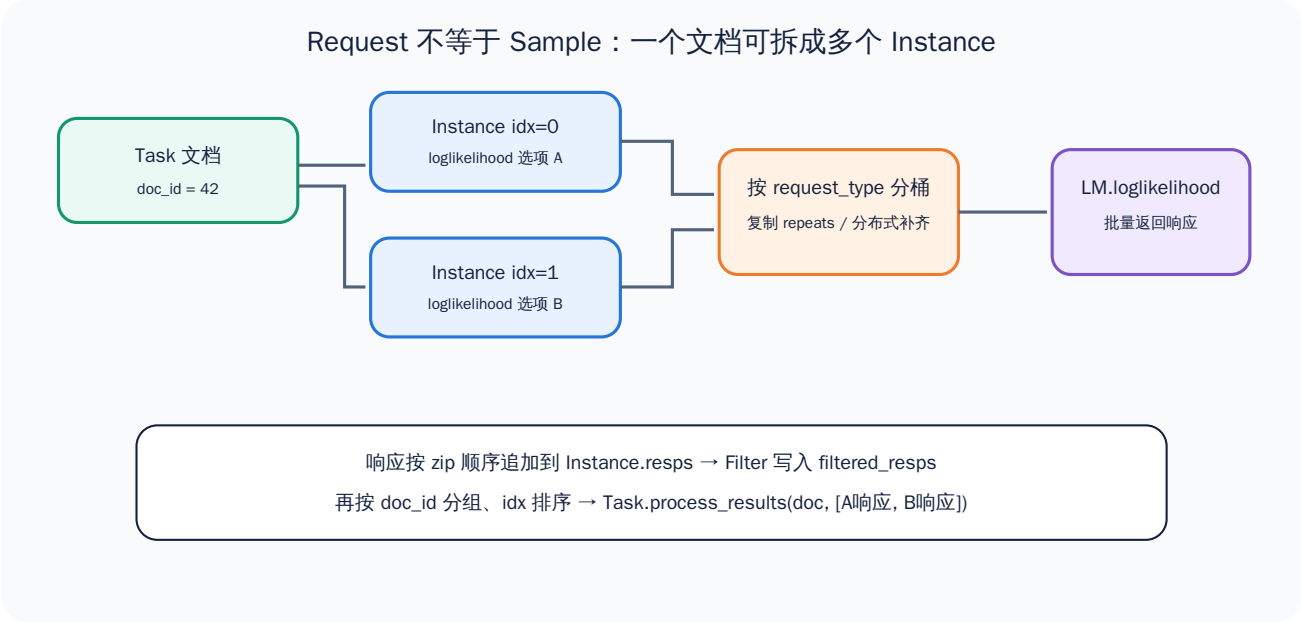
一条 benchmark 文档未必只调用一次模型，例如多项选择题会为每个选项各建一条 loglikelihood 请求，生成题可能只建一条 generate_until，repeats 还会把同一条请求复制出多份。只要把「模型请求数」「文档数」和「统计样本数」混在一起，成本、分母以及重试分别该怎么算都会跟着出错。这一篇就沿着 evaluate 的主循环走一遍，看 Instance 从建出来到收到响应、再回到文档结果的全过程。

先建立源码地图

调度循环从锁定的 evaluate() 开始，请求由 Task 这一侧来建，入口是 build_all_requests()。api/instance.py 规定每条 Instance 要保存哪些字段，api/model.py 则列出 Model Adapter（模型适配器）必须实现的三个请求方法。

OutputType 包含 loglikelihood、loglikelihood_rolling、generate_until 和 multiple_choice，不过源码注释说得很清楚，multiple_choice 真正进入调度时会拆成若干 loglikelihood 请求，所以 output_type 不一定和最后调用的 LM 方法同名，两者不能混称。

完整调用链



1. evaluate 为每个 Task 计算 limit 或指定 sample indices，再调用 task.build_all_requests，传入 rank、world_size、cache、system instruction 和 chat template 等上下文。
2. Task 遍历评测文档、构造上下文并创建 Instance。metadata 在 __post_init__ 中拆为 task_name、doc_id 和 repeats；idx 表示同一文档内请求顺序。
3. 调度器遍历 task.instances，按 request_type 放入全局 requests 分桶。分布式模式先收集各 rank 数量，为较短 rank 计算 padding_requests。
4. 每个分桶内按照 req.repeats 复制对象；分布式补齐再增加伪请求，使各 rank 前向批次数一致。

5 getattr(lm, reqtype)(cloned_reqs) 调用对应 Model Adapter 方法。返回列表与 cloned_reqs 用严格 zip 对齐，每个响应追加到原 Instance 的 resps。

6 Task 执行 filter，生成 filtered_resps。调度器按 doc_id 分组、按 idx 排序，恢复「同一文档的多条请求」顺序。

7 task.process_results(doc, filtered responses) 把多个响应变为文档级 metric。可选 sample log 同时记录 arguments、原始响应、过滤响应和 doc/prompt/target hash。

关键数据结构

下面三种基数必须分别记录：

基数	由什么决定	用途
eval docs	Task Dataset 与 limit/samples	文档级结果的候选分母
Instance	每个文档的请求构造	模型调用与 batching
cloned requests	Instance.repeats 与分布式 padding	实际 Adapter 调用数量

resps 收下每次重复请求返回的结果，filtered_resps 按名字保存 filter pipeline 的输出，等到调用 process_results 时，它拿到的是同一 doc 在某个 filter_key 下的过滤响应列表。所以 filter 并非报告格式化步骤，它会在评分前直接改数据，换一个 filter，metric 也可能跟着变化，最后的分数也会直接改变。

实现取舍与失败语义

调度器按 request_type 在全局分桶后，不同 Task 产生的同类请求便能放在一起批处理，从而提高模型吞吐，不过原来的文档顺序也会暂时散开，返回后必须依靠 doc_id/idx 重新排好。repeats 复制的是同一个 Instance 引用，虽然省下了反复创建对象的成本，却要求响应严格按请求顺序追加。严格 zip 能查出返回数量对不上，却发现不了 Adapter 把响应顺序排错。

分布式运行需要 padding 来对齐计算，但补出来的伪请求不能进入最终文档的 metric。构造缓存键时还必须纳入 chat template 和 tokenizer identity，否则系统可能在模板已经变化后，误用旧请求的缓存结果。unsafe Task 或多模态不兼容会让构造过程提前停下，具体 LM 遇到网络超时、速率限制后怎样 retry，则交给 Adapter 处理，因为核心 Instance 并没有像本仓库 Attempt 那样明确记录失败状态。

动手实验

设想一份四选一文档，Task 为四个选项各建一条 loglikelihood Instance，它们在 metadata 里的 doc_id 都是 7，idx 依次为 0..3，repeats=1。请写出 requests 怎样分桶，以及 LM 返回 [(-1.2, true), (-0.4, true), (-2.1, true), (-3.0, true)] 后各条 Instance 的 resps，然后把 repeats 改成 2，判断文档层的统计单位会不会也变成两个。

你也可以直接运行本仓库的静态课程合同：

```
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

四条 Instance 都会进入 loglikelihood 分桶，返回的响应按顺序写进各自的 resps，等代码按 doc_id=7 把它们归到一起，再按 idx 排好，process_results 就能选出 loglikelihood 最大的选项 B。repeats=2 会得到八次响应，可这些响应最终算不算两个统计观测，仍由 Task 处理重复结果的方式决定，不能看到 cloned request 变多，就直接把文档分母改成 2。

课程测试只检查本节有没有图示、完整章节、锁定链接和上下导航，它并不执行上游模型，因此不会下载 checkpoint，也不会访问模型 API。

如何核对

源码里的调度按顺序往前推进，依次打开下面七处代码，就能跟完一条 Instance 怎样创建、调用模型并收回响应。

1. 建立按 request_type 的全局分桶 — requests = defaultdict(list)
2. 请求 Task 构造全部 Instance — task.build_all_requests(...)
3. 按 repeats 复制、按分布式补齐 — cloned_reqs
4. 调用对应的 Model Adapter 方法 — getattr(lm, reqtype)(cloned_reqs)
5. 响应严格 zip 回填到原 Instance — req.resps.append(x)
6. 按 doc_id 分组、按 idx 排序 — instances_by_doc_id
7. 把多个响应变成文档级 metric — task.process_results(...)

这些行号从 489 一直递增到 639，足以说明上面七步不是为了讲解方便而重新排列的顺序，源码本来就这样执行。随后再读 Instance.args，确认代码怎样把非 tuple 参数包装成 tuple。

本篇不能证明什么

源码顺序和哈希日志只能解释 Harness 核心怎样组织请求，不能证明 Model Adapter 没在服务端重试，也不能证明响应确实来自声明的模型，或每份 benchmark 文档在统计上彼此独立。真实身份、样本相关性和基础设施 Attempt 都还需要外层证据，因为日志记录不了这些边界。

上一节 · 下一节

03 | 评分、聚合与测试：从 process_results 到 Group 结果

上一节 · 下一节

本篇要解决什么问题

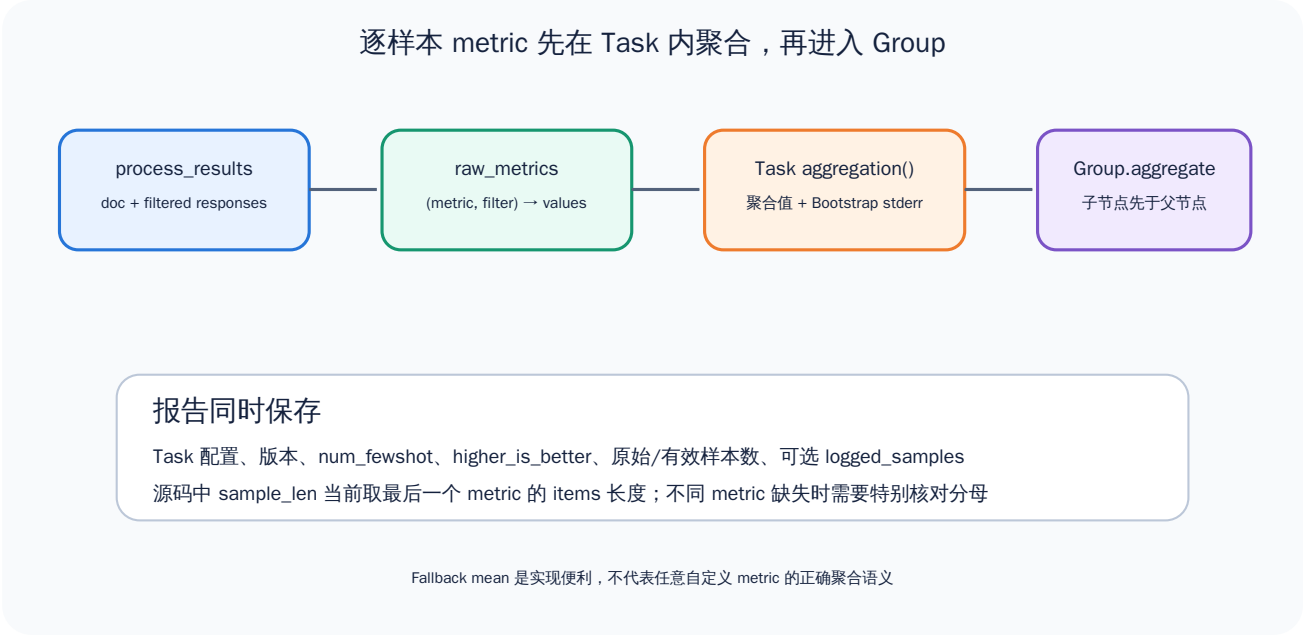
模型响应写回 Instance 后，系统还算不出「准确率」。分母也还没定。Task 得先读懂同一文档的几条响应，把它们变成这条样本的 metric，再为每个 metric 选择聚合函数、计算标准误，有些配置还会把多个 Task 继续汇成 Group（任务组）。这里有几个问题不能漏：自定义 metric 没声明 aggregation 时系统会怎样处理，有效样本数到底怎么算，Group 为什么要从最深一层往上归并，以及测试锁住了哪些行为？

先建立源码地图

每条样本的结果由 Task.process_results() 来解释，aggregation 也配置在同一个类里，随后 evaluate() 的核心循环才把这些结果加入 raw_metrics。Task 和 Group 怎样聚合、stderr 怎样计算、最终结果怎样收集，分散在 evaluator_utils.py 的四个函数里，后文会按实际执行顺序一一走过。

这个名字很容易误导人。process_results 实际只处理一份 doc 的结果，并不负责整次运行报告。它返回的字典会先把 value 一条条积累起来，等到 aggregation 真正执行时，系统才得到 Task 层的 metric。

完整调用链



1. filter 完成后，调度器为每个 doc 调 task.process_results(doc, responses)，得到 {metric_name: sample_value}；
2. 每个值追加到 raw_metrics[(metric, filter_key)]，若启用 log_samples，同一记录保存 doc、target、arguments、resps、filtered_resps 与三个 hash；
3. _process_results 调 _collect_results，后者对每个 Task 调 _compute_task_aggregations；
4. 对每个 metric/filter，代码查询 task.aggregation()[metric]，找不到时记录 warning 并 fallback 到 mean；

5 聚合函数作用于 items，结果键写成 metric,filter，若 bootstrap_iters > 0，再由 stderr_for_metric 选择可用标准误函数；样本不足或不支持时写 N/A；

6 _collect_results 保存 Task config、version、num_fewshot、higher_is_better、logged samples，以及 original/effective n_samples；

7 aggregate_groups 以 post-order 收集 Group，确保子组先于父组，再调用每个 Group 自己的 aggregate。

关键数据结构

```
eval_results_acc[task] = {
  task,
  raw_metrics: {(metric, filter): [sample values]},
  logged_samples: [...]
}

EvalAcc
  metrics / configs / versions / num_fewshot
  higher_is_better / samples / n_samples / groups
```

锁定源码遍历 raw_metrics 时，会不断把 sample_len 改成当前 items 的长度，注释也明确提醒你，这个值最后反映的是最后一个 metric 的 count。这个坑要看清。只要不同 metric 因为缺失而积累出不同数量的 items，一个 sample_len 就会遮住它们的分母差异。这项边界可以直接从代码里核对，所以本仓库才要求每个 Metric 都明确记录自己的 denominator。

实现取舍与失败语义

Task 自己声明 aggregation 后，metric 怎样归并就能贴着任务语义来写，也可以采用 mean 以外的函数。可一旦代码找不到声明就 fallback 到 mean，扩展时虽然不容易立刻报错，却可能把原本需要配对、加权或非线性处理的 metric 算错。Bootstrap（自助法）stderr 能给出一层基础的不确定性估计，但它以 items 为重采样单位，如果多份 doc 来自同一实体或同一种 prompt 模板，你还得另外检查独立同分布假设是否站得住。

Group 会按后序遍历处理嵌套层级，这样父组开始归并时，所有子组都已经算出结果。higher_is_better 还要沿着子任务往上传，只要方向发生冲突，系统就不能悄悄选一个继续算。结果里能同时保存有效样本数和原始样本数，确实方便核对，但它仍不等于事先声明的 Trial Plan，limit 怎样生效、filter 去掉了什么、哪些值缺失，都要逐项查清。

动手实验

请手工计算两项 Task：T1 有 2 条样本，acc 为 [1, 0]，T2 有 4 条样本，acc 为 [1, 1, 1, 0]。先按 Group 声明的 weight_by_size=true 求出结果，再把两个 Task metric 直接做不加权 mean，并解释这两种算法为什么会得到不同答案。

再去看上游 tests/test_aggregation_pipeline.py 中单 Task、双 Task 加权、嵌套 Group 和错误路径各有哪些测试，然后列出你会为自定义 aggregation 补上的最小测试集。

预期输出与答案

T1=0.5，T2=0.75。若按样本数加权，结果是 (2×0.5 + 4×0.75)/6 = 2/3，两个 Task 等权做 mean 则得到 0.625。哪一个答案才符合任务，要看 Group 事先声明了什么 estimand，不能等结果出来后再挑算法。自定义 aggregation 至少要测试正常输入、空样本或单样本、不同 filter、stderr 支持以及与 Group 组合时的行为。

上游测试要求单 Task Group 保留原来的 Task metric，双 Task weighted Group 按 sample_len 加权，嵌套 Group 则必须验证子节点先算出结果。

如何核对

按顺序阅读 _compute_task_aggregations、_collect_results、aggregate_groups 和 _process_results，沿着返回值看结果怎样逐层汇总。这里要重点核对 fallback mean、bootstrap 上限的特殊处理、sample_len TODO 和 post-order traversal，然后回到 api/task.py，看 ConfigurableTask 怎样注册 aggregation 和 higher_is_better。

本篇不能证明什么

聚合测试通过，只能说明锁定实现会怎样把每条样本的值汇成 Task 或 Group 输出，不能证明 benchmark 采样有效，也不能证明 stderr 覆盖了真实的不确定性，更不能据此把排行榜当成发布 Gate。样本是否独立、结论能否推广，以及哪些风险不允许互相补偿，都要在外层另行设计。

上一节 · 下一节

Inspect AI 源码课程：把 Solver、Sandbox、Scorer 和 EvalLog 放进同一条链

上一节 · 下一节

本篇要解决什么问题

Inspect AI 不只是「另一套 benchmark runner」。它用 Dataset、Solver（求解器）、Scorer（评分器）和可选的 Sandbox 组成 Task，所以既能跑普通模型任务，也能评测会用工具、会改变状态的 Agent。读源码时，你要看清公共 eval 怎样解析 Task，eval_run 怎样给每项 Task 准备隔离环境和日志，task_run 又怎样执行每条 Sample，最后 Scorer 如何读取 TaskState 与 Target，并把评分写进结构化 EvalLog。

课程锁定提交 ebf4815ee260afcc8c34ad9d66e6f8d98a89e905。正文会把上游源码里能直接核对的类和调用写成源码事实，若把这些对象对应到 Trial、Attempt、Observation 和 Gate，则会明确标成机制解释，不会因为名称相近就假定两边字段一一对应。

先建立源码地图

站点	锁定源码	责任
公共入口	_eval/eval.py	同步/异步 API、Task 解析、模型与运行上下文
Task 调度	_eval/run.py	TaskRunOptions、SandboxManager、并发与 Task retry
单 Task 执行	_eval/task/run.py	Sample 生命周期、Solver Plan、Scorer 与日志更新
Scorer 协议	scorer/_scorer.py	Scorer(state, target)、注册、重建规格与 Metric 元数据
日志模型	log/_log.py	EvalSpec、EvalSample、EvalPlan、EvalResults、EvalLog

完整调用链

1. eval 作为同步包装进入 eval_async，后者建立运行上下文，解析模型、Task 来源、审批和日志位置。
2. eval_resolve_tasks 把 Registry 名称、文件或 Task 对象解析成 ResolvedTask。同一入口还支持 TaskSource 动态注入。
3. eval_run 创建 SandboxManager，为每个 ResolvedTask 合并调用参数与 Task 默认值，解析 Solver、Scorer、Reducer，并生成 TaskLogger 与 TaskRunOptions。
4. run_task_retry_attempts 调度多项 Task 并按模型平衡并发；Task 出错或显式请求 retry 时可重新排队，已写出的失败日志仍保留，而不是被成功结果悄悄覆盖。
5. _run_task 调用 task_run。后者为 Dataset Sample 建立 TaskState，在 Sandbox 中运行 setup 和 Solver Plan，记录模型、工具、状态与错误事件。
6. Scorer 是异步 callable，接收最终 TaskState 与 Target；多个 Scorer 可分别产生 Score，跨 epoch 结果还可交给 Reducer。
7. EvalLog 汇总 EvalSpec、EvalConfig、EvalPlan、样本事件、Score、Metric、usage、stats 和 error，形成可查看与可重评分的运行证据。

关键数据结构

EvalSample 会保存 id、input、target、metadata、sandbox、files、setup、messages、output、scores、events、model_usage 和 error，EvalPlan 则按顺序记下 Solver steps。EvalScore 收下 Scorer 名、参数、Metric、scored_samples 和 unscored_samples，等这些对象都准备好后，EvalLog 再把 spec、plan、results、samples、stats 和 status 汇成一份日志。

和只给最终准确率的报告相比，这些对象留下的信息多得多，但它们不会自动变成内容寻址证据。日志可能按配置裁掉 explanation 和 metadata，也可能根本不写 Sample，所以你在把日志交给下游 Scorer 或审计器之前，必须先查清这次运行究竟记录到了哪一层。

实现取舍与失败语义

Task 可以选用不同 Sandbox，因此同一个 Solver 能在本地、Docker 或其他环境实现里运行，Agent 最后留下的环境状态也能交给评分器检查。代价也很具体：创建环境、复制文件、执行 setup 和 cleanup、应用网络策略，这些动作都会成为运行协议的一部分。Scorer 必须写成 async callable，方便 Judge（裁判模型）或外部检查并发执行，可日后能否复现，仍取决于模型身份和参数有没有记完整。

Task retry 和 Sample 没有通过评分是两件事。run_task_retry_attempts 只会因为整项 Task 出错，或收到 cancel/retry 请求而重新调度，score_on_error 则决定 Sample 执行异常后还要不要评分，普通 Score 没通过本身就是一条有效结果。若把这三种情况统称为「失败重试」，你最后解释统计结果时一定会算错。

动手实验

打开锁定的 log/_log.py，把 EvalSample 字段按输入身份、执行轨迹、评分结果、资源与错误分成四类，再用同样的方法整理 shipping Reference Harness 的 evidence.json，看看 Inspect AI 额外记录了哪些信息，本仓库又更强调哪些证据。

运行：

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

输入身份至少包括 id/input/target/metadata，执行过程会留下 sandbox/files/setup/messages/output/events，评分写进 scores，资源和错误则由 model_usage、total_time 与 error 记录。Inspect AI 对模型运行和 Sandbox 留下了更细的信息，Reference Harness 则会明确绑定 canonical Attempt、Artifact（产物）digest 和事先计划的 Trial 分母，两边正好可以互补，不能数一数字段多少就判断谁更完整。

来源锁应该验证 8 项。只要这门课程的四篇文章还没全部写好，课程合同就该继续失败，防止有人用一篇概览冒充完整课程。

如何核对

先在 _eval/eval.py 搜索 eval_async、eval_resolve_tasks 和 eval_run，看入口怎样把参数传下去，再到 _eval/run.py 找 TaskRunOptions 与 run_task_retry_attempts，最后去 _eval/task/run.py 跟进 task_run。每往下走一层，都要记下谁调用它、传入了什么、状态怎样变化以及最后返回什么。

本篇不能证明什么

结构化 EvalLog 只能告诉你运行记录成了什么样，不能证明 Sandbox 已经彻底隔离、Judge 确实有效，也不能证明 Dataset 足够代表线上任务。Inspect AI 的 Task retry 也不能直接等同于本仓库的 Attempt，只有先把失败怎样分类、统计怎样计划说清楚，才能可靠地映射这两个对象。

上一节 · 下一节

01 | Eval、Task 与 Solver：从公共 API 到可执行 Plan

上一节 · 下一节

本篇要解决什么问题

Inspect AI 的 eval() 可以在入口改写模型、Task、Sandbox、Solver（求解器）、epochs、并发、预算、日志和 retry 等参数，如果只把它当成一个大函数，你就很难看出哪些值只管整次 Eval 的调度，哪些值会传给每个 Task，又有哪些值最终会改动单个 Sample 要执行的 Solver Plan。这一篇从同步 API 一路跟到异步上下文、Task 解析和 TaskRunOptions，看这些参数怎样逐层落到可执行计划上。

先建立源码地图

公共入口分成同步和异步两个：eval() 与 eval_async()。模型和 Task 交给 eval_resolve_tasks() 解析，随后 eval_run() 为各项 Task 做准备并安排运行，最后 _eval/task/run.py 把 Solver Plan 解析出来，再进入单项 Task 的主循环。

锁定源码里，eval_async 同一时间只允许一个调用处于活动状态，并用 anyio TaskGroup 运行 _eval_async_inner。别把它当成永久限制。这个结论只说明当前锁定版本怎样工作，不能顺手推广到其他版本或所有部署方式。

完整调用链

1. eval 处理同步显示与异常边界，再调用 eval_async，后者规范化 checkpoint 与 control server 参数，创建 TaskGroup。
2. _eval_async_inner 解析 model_args/task_args、成本表与日志配置，生成 run_id，并初始化模型、并发限制和运行 hook。
3. eval_resolve_tasks 在已初始化的模型/角色上下文中解析 Task，输入可以是 Registry 名称、文件、Task 对象或 TaskSource；每个逻辑 Task 与实际 Model 组合成 ResolvedTask。

4 入口调用 eval_run，该函数创建 SandboxManager，并为每个 ResolvedTask 合并 Eval 参数与 Task 默认配置；调用级 Solver 可以覆盖 Task 自带 Solver，但覆盖结果必须进入日志规格。

5 resolve_plan 把单 Solver、Chain 或 Plan 统一为 Plan；Task 的 setup Solver 会被复制后前置，避免重复调用在原对象上不断叠加 setup。

6 Scorer 被转成可重建的 ScorerSpec，Reducer、审批、限制、日志与 SampleSource 一起进入 TaskRunOptions。

7 run_task_retry_attempts 以 Task × Model 为调度单位，在并发上限内尽量平衡不同 Model，最终把每项 Task 交给 task_run。

关键数据结构

这里要分清两个对象。Registry（注册表）名称、File 或对象经过解析后，会连同来源一起装进 ResolvedTask。等到单项 Task 真要运行时，代码再把 Task、Model、Sandbox、EvalConfig、Solver、Scorer、Logger、SampleSource 和各项限制冻结到 TaskRunOptions 里。Plan 则按执行顺序保存 Solver steps，每一步还会记成 EvalPlanStep，方便你从日志里还原「当时究竟跑了哪些 Solver」。

Task 从 Dataset 里拿到 Sample 后，Solver 会不断改写 TaskState 中的 messages、output、tools 和 store，等 Solver 跑完，Scorer（评分器）才读取 state 与 Target。这样拆开后，prompt、模型调用和评分各自由谁处理都看得见，比全塞进一个 callback 更容易审计，不过 Task 仍把多项配置收在同一个对象里。

实现取舍与失败语义

同步入口最终走到同一套异步实现，调用者更容易上手，anyio 也便于安排并发的 Task 和 Sample，不过全局一次只能有一个活动的 eval_async，所以你仍不能在同一进程里随意嵌套运行。入口参数还能覆盖 Task 默认值，这对做实验很方便，但日志必须保存解析后配置，否则只看 Task 源码，根本还原不了当时真正采用了哪些值。

用入口参数替换 Solver，适合拿同一份 Dataset 比较不同策略，可它会同时改动工具、消息和停止方式，不能只记一个 Solver 名就把它当普通超参数。只要 Task 没解析成功、模型角色缺失或 Sandbox 规格无效，系统就该在 Sample 开始前停下。边界要守住。至于 Solver 做出了错误行为，那是 Sample 层的产品结果，不能自动算成 Eval 的基础设施错误。

动手实验

你可以为同一个虚构退款 Dataset 设计两个 Solver：direct_answer 只生成一次，tool_agent 可以查询订单并调用退款工具。随后列出在入口替换 Solver 时必须一起冻结的信息，包括 Solver 的 Registry 名和参数、工具集合、审批策略、Sandbox、Model，以及 message、turn、token、time、cost 等限制。

再到锁定源码里找到 resolve_plan，看看 Task.setup 为什么只能加到复制出来的 Plan 前面，不能直接修改还会继续复用的原 Plan。

预期输出与答案

比较这两种 Solver 时，不能只看最终文本，因为 tool_agent 还会留下工具副作用并改变环境终态，所以至少要冻结 SolverSpec、Model、工具权限、审批规则和各项预算。resolve_plan 会先复制 Plan，是因为同一个 Task 或 Plan 可能被多次 eval，如果直接在原对象前面插入 setup，第二次运行就会再插一次，实际执行的 Plan 也就偏离了原先声明。

课程里另外两篇还没写好或没有满足内容合同时，测试就该继续失败，以免有人只交一张全局图，便跳过 Sample 和 Scorer 的具体机制。

如何核对

按顺序找到 eval_async、_eval_async_inner、eval_resolve_tasks 和 eval_run，先看参数怎样一路传下去。然后在 _eval/run.py 看代码怎样建出 TaskRunOptions，最后到 _eval/task/run.py 核对 resolve_plan 处理 Solver、Chain、Plan 和 setup 的各条分支。

本篇不能证明什么

把 Plan 完整记下来，只能说明锁定实现怎样把声明变成执行计划，却不能证明 Solver 足够安全、工具权限已经收紧，或这项任务真的有效。实际模型身份、Sandbox 隔离效果和 Scorer 是否可靠，都还要拿出别的证据。

上一节 · 下一节

02 | Sandbox 与 Sample：一次 Agent Eval 怎样真正运行

上一节 · 下一节

本篇要解决什么问题

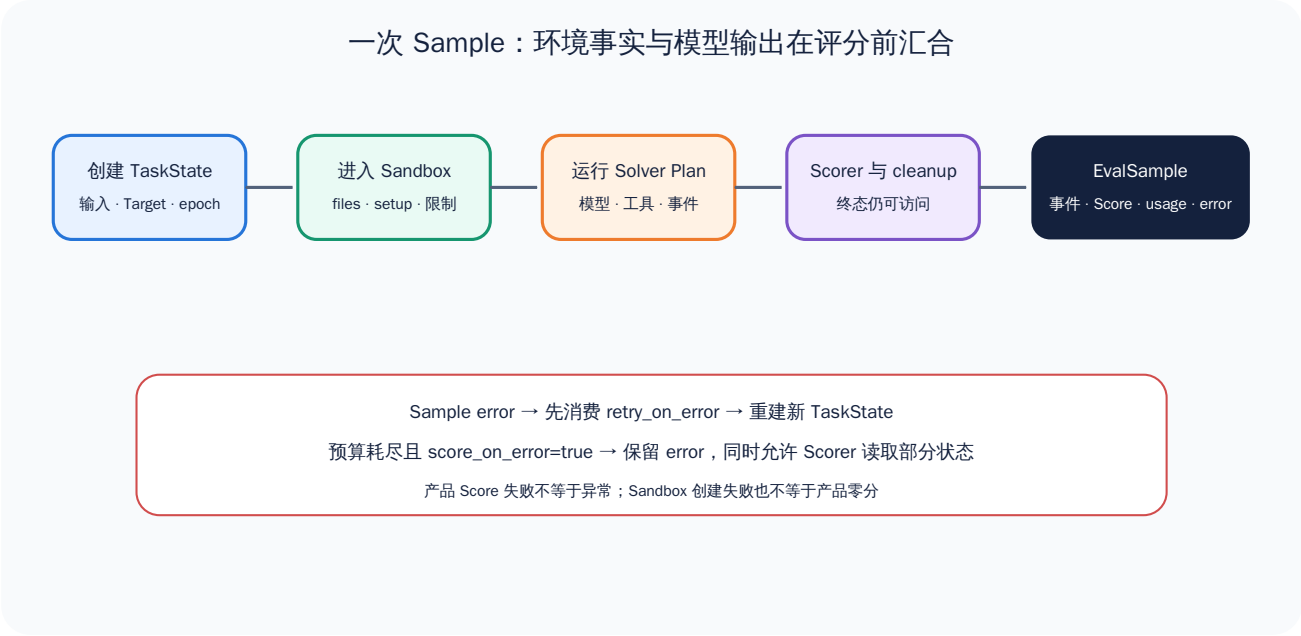
做普通问答时，你可能觉得 Sample 只是从 input 走到 output，可到了 Agent Eval，一条 Sample 还会复制文件、执行 setup、创建 Sandbox、开放工具、产生多轮消息，并可能撞上 token、time 或 cost limit，系统还得在清理环境前检查终态。只盯着最终 output，会漏掉很多真正影响产品结果的行为。这一篇跟着 task_run 往下走，看它怎样把 Dataset 里的 Sample 装进 TaskState，再怎样在隔离环境里运行 Solver（求解器）并保存 EvalSample。

先建立源码地图

一条 Sample 从开始到结束怎样变化，主要写在锁定的 _eval/task/run.py 里。Task 这一层怎样准备 SandboxManager 并安排并发，可以到 _eval/run.py 找，而日志要保存的 EvalSample、SandboxEnvironmentSpec、EvalSampleLimit 和事件容器，则定义在 log/_log.py。

把这三个文件连起来看就会发现，Environment 要真正启动、建立连接、执行限制、负责清理并留下记录，不能把它缩成 Target 上的一个字符串参数。

完整调用链



1. task_run 解析 Plan、Scorer 名称、epoch 与 Sandbox limits，创建进度和结果容器，并从 Dataset/SampleSource 取得待运行 Sample。
2. create_sample_state 把 Sample 输入、目标、Model、metadata 和初始消息装入 TaskState；同时，每个新运行具有独立 UUID，避免 sample retry 复用错误状态。

3 task_run_sample 初始化 scoring context 和 Sandbox context，其中 Sample 自己的 sandbox 可以覆盖 Task 默认；files、setup 与连接信息在环境初始化阶段处理。

4 进入 Sandbox 后，Solver Plan 依次运行，而 Generate 函数更新 TaskState，模型调用、工具事件、store、消息和 output 进入 transcript/Event。

5 消息数、token、turn、wall time、working time 与 cost 等限制在运行期间检查。达到限制可形成 EvalSampleLimit，并根据策略停止、评分或标错。

6 Solver 完成后运行 Scorer；随后 Task cleanup 仍在 Sandbox context 内执行，以便检查或清理环境。

7 make_eval_sample 从 state、events、scores、usage、error、limit 与 sandbox 信息构造 EvalSample。只有在清理和日志阶段完成后，Sample 才进入 Task 级聚合。

关键数据结构

运行过程中，代码会持续改写 TaskState，里面放着 model、sample_id、epoch、input、messages、output、tools、store、metadata 和 scores。等到要写日志时，代码才把当时的状态固化成 EvalSample，除了输入输出，还把 target、sandbox、files、setup、events、scores、model_usage、error、limit 和时间摘要一并记下来。

SandboxEnvironmentSpec 只说明环境采用什么类型、带哪些配置，真正能证明环境里发生过什么的，还包括连接记录、文件、命令结果和最终状态。EvalSampleLimit 也必须说清撞上的是 token、message、turn、time、working time 还是 cost 限制，因为不同停止原因会导向不同的产品解释。停止原因要记清。

实现取舍与失败语义

每条 Sample 各用一个 Sandbox，通常最容易复现，也方便清理，只是每次启动环境都会多花一些成本。复用环境虽然更快，但你得先证明 reset 真能清掉上一条 Sample 留下的状态。Inspect AI 允许你在 Task 或 Sample 层设置 Sandbox 和并发上限，实现者因此可以按场景取舍，不过教材只确认它提供了这种能力，不会看到「用了容器」就认定环境没有污染。

Sample 出错时，retry_on_error 会递归创建新的 TaskState，并把 error_retries 留下来，所以它和 Solver 在内部自我修正、整项 Task 的 retry、普通 Score 没通过都不是一回事。等重试次数用完，score_on_error 才决定要不要把出错的 Sample 交给 Scorer（评分器），让评分器衡量已经完成的部分，但这个错误仍会进入 fail_on_error，不能靠 Score 抹掉。操作员取消运行时选择 score、error、abort 或 retry，也会让任务以不同方式结束。

动手实验

你可以设计一条「删除指定临时文件」的 Agent Sample：Sandbox 开始时有 target.txt 和 keep.txt，Agent 只能调用 shell 工具，Scorer 最后检查环境。请列出必须留下的观察，包括初态摘要、每次工具调用、退出码、最终目录清单、目标文件是否消失、保留文件是否还在，以及 Sandbox reset 的结果。

然后再回答一个具体问题：Agent 的最终文本虽然写着「已删除」，但 target.txt 仍然存在，此时 TaskState.output、EvalSample error 和 Score 应该分别记录什么。

预期输出与答案

最终文本只会进入 output，所以即使整个执行过程没有抛出异常，EvalSample 可以不带 error，负责检查环境事实的 Scorer 仍该给出 failed。Agent 自己声称成功不能当作通过证据。反过来，如果 Sandbox 根本建不起来，这条 Sample 遇到的就是基础设施错误，可以交给 retry_on_error 恢复。若 Agent 执行 rm 时收到权限错误，它又成了被测行为的一部分，究竟算不算产品失败，要由 Task 和 Scorer 的契约判断，平台不能默认重试。错误归属必须说清。

在这些观察中，最终目录和文件摘要应以 Artifact（产物）或结构化 Event 的形式交给评分器。只要 reset 失败，后面的 Sample 就可能读到残留状态，此时应该直接阻断运行，不能再继续累计分数。

如何核对

在 _eval/task/run.py 依次搜索 task_run_sample、sandboxenv_context、TaskState(、span("solvers")、Scorer 循环、make_eval_sample 和递归 retry，跟清一条 Sample 怎样运行。然后到 log/_log.py 逐项核对 EvalSample 字段，分清哪些观察能长期保存，哪些只活在运行内存里。

本篇不能证明什么

即使已经有 Sandbox 和事件日志，你也不能据此证明环境从未逃逸、reset 清得足够干净，或所有副作用都被观察到了。若要把 Agent 评测用到发布流程里，还得补上威胁模型、最小权限、外部服务隔离和独立的终态断言。

上一节 · 下一节

03 | Scorer、EvalLog 与 Retry：怎样保留可重评分的运行事实

上一节 · 下一节

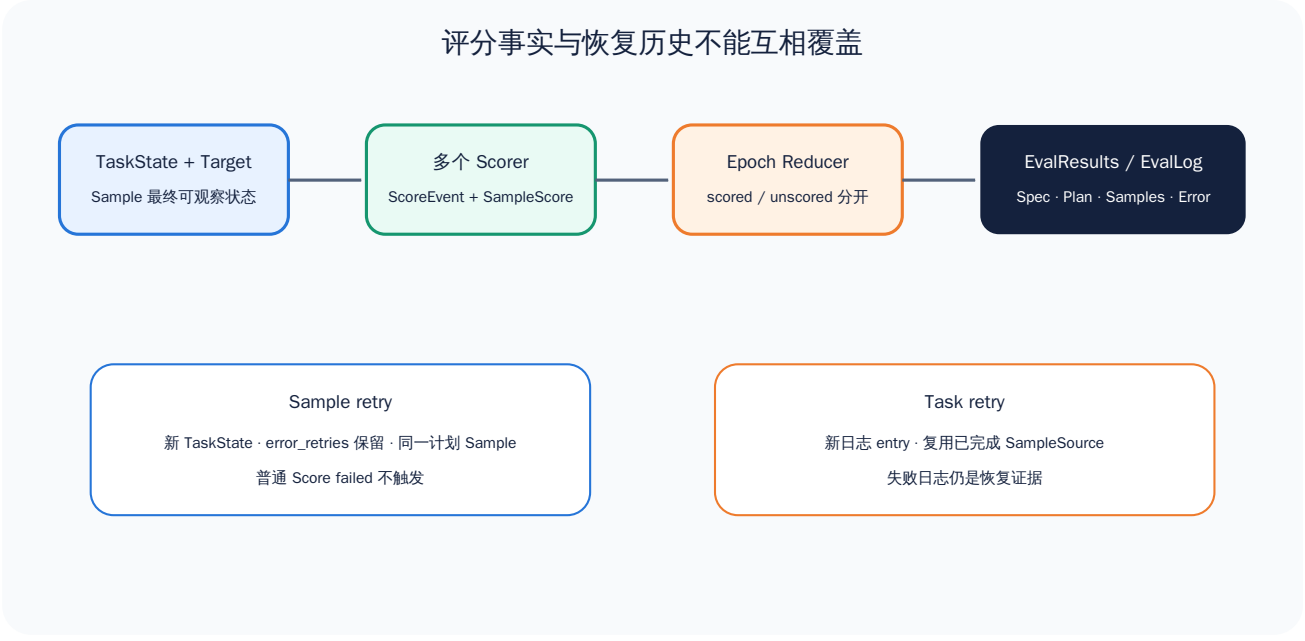
本篇要解决什么问题

Inspect AI 既能保存完整 EvalLog，也允许 Sample retry、Task retry、score_on_error、多个 Scorer（评分器）和跨 epoch 的 Reducer（归并器）同时工作。功能一多，解释结果时更得把边界看清：失败后重试产生的结果怎样写进日志，Scorer 能不能按记录重建，未评分样本会不会从分母里悄悄消失，Task retry 又是否会拿成功尝试盖住失败历史？这一篇把评分和恢复两条路径分别追到最终保存的对象上。

先建立源码地图

评分相关的四个对象都能在锁定的 scorer/_scorer.py 中找到，而且职责分得很清楚：Scorer Protocol 规定怎样调用评分器，ScorerSpec 保存注册时留下的元数据，@scorer 装饰器 把对象包装进 Registry（注册表），scorer_metrics 则取出 Metric 元数据。Sample 怎样调用 Scorer、Reducer 怎样参与处理、Task 怎样汇总结果，都要从三千多行的 task_run() 入口往下追。整项 Task 的重试由 eval_run() 调度，至于日志最终长什么样，则由 EvalSample、EvalScore 和 EvalLog 一起规定。

完整调用链



1. eval_run 把 Task 的 Scorer 转为 ScorerSpec，保存 Registry 名、构造参数、metadata 和 Metric 规格；未通过 @scorer 创建的对象无法可靠记录为可重建规格。
2. task_run 为多个 Scorer 生成唯一名字。每个 Sample 完成 Solver 后，依次执行 await scorer(state, Target(sample.target))。
3. Score 写入 TaskState.scores、ScoreEvent 和 SampleScore；Scorer 若直接篡改同名 state.scores，运行会检测冲突。

4 多 epoch 的同一 Sample 可由 Reducer 合并；与此同时，Task 结果明确保存每个 Scorer 的 scored_samples 与 unscored_samples，而不是只给一个平均值。

5 TaskLogger 把 EvalSpec、Plan、Samples、Results、Stats 和 Error 写成 EvalLog，而 log_header_only 或关闭 log_samples 会改变可审计深度，必须进入运行配置。

6 Sample error 先消费 retry_on_error；Task error 或显式 cancel/retry 由 run_task_retry_attempts 重新排队，Task retry 新建日志 entry，并从失败日志构造 SampleSource 复用已完成 Sample。

7 调度器按原 index 返回最新 Task 结果，但失败日志位置仍存在，因此外层审计必须把多条 entry 视为恢复历史，不能只看最后一条。

关键数据结构

ScorerSpec(scorer, args, metadata, metrics) 记下以后怎样重建评分器，EvalSample.scores 则按 scorer-name 保存各个 Score。再沿着写盘路径往外看，EvalScore 会收下 name/scorer、参数、Metric、scored_samples、unscored_samples 和 metadata，EvalResults.scores 把各个 Scorer 的结果汇在一起，最后 EvalLog 才把 status、eval spec、plan、samples、results、stats、error 和 location 装进一条完整日志。

Reference Harness 会让 ScoreRecord 强制绑定 canonical Attempt 和 Observation digest，Inspect AI 则更看重 TaskState、Sample events 以及能按记录重建的 Scorer。给两者做 Adapter（适配器）时，若缺少严格 digest 或 canonical 语义，就该老实标成 partial，不能看字段名字相近便宣称完全对应。

实现取舍与失败语义

把 Scorer 注册进 Registry 后，其他人更容易按日志重评分或共享它，可动态闭包和本地对象如果记不下构造参数，后面就很难复现。多个 Scorer 配合 Reducer 能表达更丰富的评测方式，但每个 Metric 仍要各自写明分母以及 reducer 怎样归并。缺失不能藏起来。scored_samples 和 unscored_samples 就是在告诉你有多少样本没有得分，所以发布 Gate 不能只读已评分子集的平均值。

Sample retry 会重新创建状态，适合拿来处理瞬时错误，如果模型只是答错、运行并未异常，就不该触发 retry_on_error。Task retry 可以复用已经完成的 Sample，省下一部分成本，但前提是 Task、Model、Sandbox 和 Scorer 都没有换身份，而且新日志必须能追溯到原来的失败日志。源码注释还把 retry、abort、score/error graceful resolution 和外部取消分别处理，只有任务确实出错或收到明确的 retry，并且重试预算还没用完，调度器才会把它重新排队。

动手实验

设想一次运行事先计划了 100 条 Sample，其中 90 条拿到 Score，具体是 80 passed、10 failed，另有 5 条遇到 Solver error 且 score_on_error=false，还有 5 条被 Scorer 返回 None。请分别计算「只在 scored samples 上的通过率」和「以计划 Sample 为分母的通过率」，然后判断 Gate 应该给出 passed、failed 还是 inconclusive。

再比较两条恢复路径：一条是 Sample 23 首次调用 API 时超时，retry 后成功，另一条是 Task 写日志时失败，随后 Task retry 复用已经完成的 Sample。请画清每条路径里第一次错误、后续重试和最终 Score 怎样关联。

预期输出与答案

只看已经评分的子集，通过率是 80/90≈88.9%，若把事先计划的全部 Sample 放进分母，结果就是 80/100=80%。假如政策要求至少达到 85%，同时规定关键缺失不能忽略，Gate 就不能拿 88.9% 判为 passed，而要因为 10 条 Sample 没有评分而给出 inconclusive，或者按事先声明的保守政策判为 failed。这项选择必须提前定好。

Sample retry 应把第一次错误留在 error_retries 里，再让新状态产生的 Score 进入结果。Task retry 则要保留失败的 EvalLog，并让新的 entry 记录复用过的 SampleSource 和这一次 Task 尝试。两条恢复路径都不能伪装成「从未失败过」。

如何核对

先在 scorer/_scorer.py 看 Scorer Protocol、装饰器和 as_scorer_spec 怎样配合，再从 task_run() 进入 Task 主循环，跟着 Reducer 参与的指标刷新 和 重排队时怎样处理未评分样本 两段代码看清状态怎样变化。最后到 _eval/run.py 阅读 run_task_retry_attempts，核对它怎样根据 cancel_type 和 result.status 选择分支。

本篇不能证明什么

日志字段再齐全，也只能让你看见评分和恢复怎样串起来，不能证明 Scorer 已经校准、Judge 没有偏差，更不能保证外部服务变化以后，Task retry 前后的结果还可以直接比较。独立的发布 Gate 仍要固定身份、分母和缺失政策。

上一节 · 下一节

OpenAI Evals 源码课程：Registry 怎样驱动 Eval 与事件记录

上一节 · 下一节

本篇要解决什么问题

OpenAI Evals 的经典实现围绕 Registry（注册表）、EvalSpec、CompletionFn（补全函数）和 Recorder 运转。命令行虽然只收到几个字符串，程序跑起来后却要逐个把它们解成 Eval 类、数据路径、模型调用对象和日志后端。如果只看 YAML，你容易把配置当成已经执行的运行，但只盯着某个 Eval 子类，又会漏掉运行身份和统一记下的事件。因此这组课程会顺着 oaieval.run 往下走，看配置键怎样一步步变成 final report。

本课锁定在 8eac7a7de5215c907fbddc30efdaf316913eccdd。这份源码保留了较早的 CompletionFn/Eval Registry 设计，也加入了 SolverEval 等扩展，所以课程只用它讲清一套由配置驱动的 Eval Harness 怎样工作，不会拿这些公开接口去推测 OpenAI 现在所有内部评测系统。

先建立源码地图

站点	锁定文件	责任
CLI	cli/oaieval.py	参数、RunSpec、Eval/CompletionFn/Recorder 装配
Registry	registry.py	加载 YAML、解引用 spec、实例化类
Eval	eval.py	Sample 遍历、CompletionFn 使用与聚合返回
Completion API	api.py	CompletionFn Protocol 与匹配记录 helper
Recorder	record.py	Sample 事件、Local/HTTP/DB 后端与 final report

完整调用链

1. oaieval.run 创建或接收 Registry，追加自定义 registry paths，再通过 eval key 取得 EvalSpec。
2. CLI 合并 Eval 参数和 completion_args 后，将 CompletionFn 字符串按逗号拆分，再由 registry.make_completion_fn 解析为模型快捷实现、Registry CompletionFn 或 Solver。
3. 入口构造 RunSpec，选择 Dummy、Local、HTTP 或数据库 Recorder，并通过 Registry 取得 Eval class。
4. Eval class 用 completion_fn_instances、EvalSpec args、eval_registry_path 和 Registry 实例化，数据相对路径则会以该 Registry 的 data 目录为基准解析。
5. eval.run(recorder) 遍历样本时，每个 Sample 进入 recorder.as_default_recorder(sample_id) 上下文，Eval/CompletionFn 通过 helper 写 sampling、match、function_call、metrics 或 error 事件。
6. Eval 返回聚合结果 dict；CLI 随后可从 sampling 事件补充 token usage，再调用 record_final_report。

关键数据结构

EvalSpec 记下 key、class、args 和 registry_path，CompletionFnSpec 记下类与参数，RunSpec 再把 completion_fns、EvalSpec 和 run_id 组成运行头。Event 至少要连上 run_id/sample_id/type/data/time，CompletionFn Protocol 则接收 prompt 并返回 CompletionResult。这份结果最后会变成 match 还是其他 metric，得由 Eval 做决定。

这里的 Sample event 与 Reference Harness TraceEvent 有些像，但前者没有强制记录 parent/sequence、canonical Attempt 或 Observation Bundle digest，正好少了关键血缘，因此课程只能把这项能力标成 partial。两边确实都写 JSONL，但存储格式一样不等于证据合同也一样。

实现取舍与失败语义

Registry 让新增 Eval 或模型配置时少改代码，也让同一个类可以带着不同 args 反复实例化。可这样一来，运行身份就散在 YAML、路径优先级、解引用链和 CLI override 里，要复现时一处都不能漏。Recorder 通过 ContextVar 暗中带上 sample_id，Eval 代码的确会更简洁，但实现者得小心处理异步边界、线程边界和上下文缺失。

HTTP Recorder 失败后可以改用 LocalRecorder，避免远端日志服务一出故障，证据就全部丢失。但这个动作只恢复记录层，不能因此让产品 Sample 重答。此外，核心对象没有把 CompletionFn 的网络 retry 和 Eval 的 Sample 语义

明确分成 Trial 与 Attempt 两层。

动手实验

写一份运行身份清单，收入 Eval key、Registry 路径与 commit、解析后的 Eval class/args、CompletionFn key/class/args、数据文件摘要以及 Recorder 类型与路径，然后再指出如果只保存 CLI 字符串，其中哪些信息会消失。

执行：

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

如果只留下 oaieval model eval-name，你会同时丢掉 Registry 路径优先级、spec 解引用、CLI override、模型实际版本、数据内容摘要和 Recorder fallback。所以要想复现这次运行，记录里必须有解析后的 RunSpec 与真正用到的资源版本。

只有四篇正文、图示、锁定链接和上下导航都齐全，课程门禁才会放行，这一检查过程不需要 OpenAI API key，也不会真正调用外部模型。

如何核对

从 cli/oaieval.py 顺序追 get_eval、make_completion_fn、RunSpec、build_recorder、get_class、eval.run 和 record_final_report。再到 registry.py 核对 Registry 怎样加载不同资源目录和拒绝重复 key。

本篇不能证明什么

即使 Registry 和 Recorder 都留下了完整记录，你也不能由此证明 Eval 数据有效、CompletionFn 就是所声明的模型，或者这些事件已经足够审计 Agent 的副作用。final report 也不能直接当作发布 Gate，因为本课只讲配置怎样驱动整套机制，不把它扩大成生产质量证明。

上一节 · 下一节

01 | Registry 与 EvalSpec：字符串怎样变成真实对象

上一节 · 下一节

本篇要解决什么问题

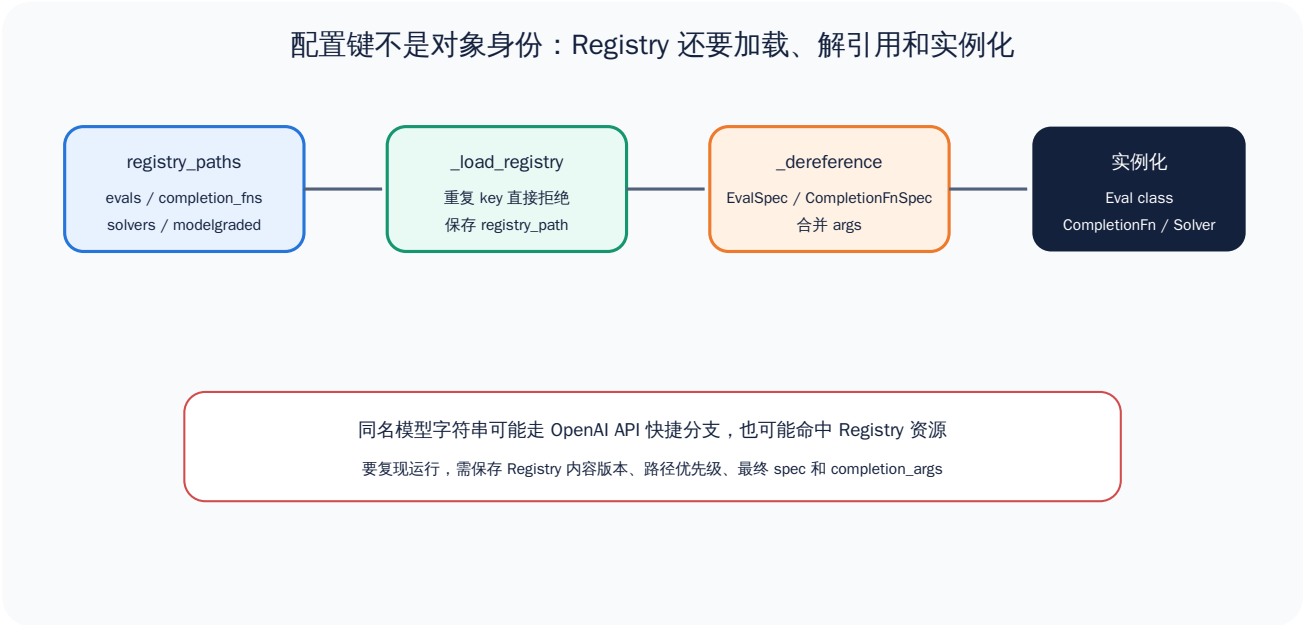
同一个 eval-name 放进不同的 Registry（注册表）路径，可能会找到不同的 class、args 和数据。模型字符串也有两条去向：要么命中 API 模型的快捷分支，要么由 Registry 解成 CompletionFn（补全函数）或 Solver。你若跳过加载顺序和解引用过程，两条看似相同的运行命令就可能造出不同对象。因此这一篇先看 Registry 究竟怎样认出每个对象。

先建立源码地图

核心代码在锁定的 Registry 类 里，读时可以按 get_eval、_dereference 和 _load_registry 三段往下追。要看 CLI 何时加入 registry paths、又怎样覆盖 args，得转到 run()，因为入口会合并参数，Registry 并不接手这个动作。至于 Eval 如何根据 eval_registry_path 找到数据，答案写在 Eval 基类 中。

Registry 会管住 completion_fns、solvers、eval_sets、evals 和 modelgraded 等多类资源。代码访问相应属性时，它才从配置路径加载目录，记下每项资源来自哪里，并在发现重复条目时立即断言失败。

完整调用链



1. Registry 初始化默认路径；CLI --registry_path 可追加更多目录。
2. _load_resources 扫描某一资源类型的 YAML，_load_registry 合并条目；重复 name 立即失败，避免静默覆盖。
3. get_eval(name) 通过 _dereference 把原始字典解析为 EvalSpec，并把 registry_path 留在 spec 中。
4. CLI 把 extra_eval_params 合入 EvalSpec.args，此时 spec 已不同于仓库静态 YAML，必须记录解析后值。
5. make_completion_fn 先识别 dummy 或已知 OpenAI 模型名，再查询 CompletionFn/Solver Registry；最终加载类、合并 kwargs，并验证实例满足 CompletionFn。

6 get_class(eval_spec) 取得 Eval class；实例化时传入 eval_registry_path，Eval 的数据路径通过 _prefix_registry_path 落到对应 Registry 的 data 目录。

关键数据结构

RawRegistry 先用 name 指向原始 spec，BaseEvalSpec/EvalSpec/CompletionFnSpec 再把其中的 key、class 名、args、metrics 和 registry_path 拆成结构化字段。这些字段各管一件事：key 负责引用，class 选中代码，args 改变行为，registry_path 则告诉程序去哪个相对路径找数据。少一项，运行身份都会走样。

RunSpec（运行规格）还会记下 run_id、completion_fns 和 eval_spec，Recorder 写每个 Event 时都可以引用它，因此 RunSpec 比一条 EvalSpec 更能表示某次具体运行。不过，它没有收进数据文件的内容、代码 commit 和服务端真正使用的模型，你还得在外部摘要里补齐这些信息。

实现取舍与失败语义

允许多条 Registry 路径，私有 Eval 就有了扩展位置，但当前代码一旦碰到重复 key 就会失败，不让后加载的条目覆盖前者。冲突因此会当场暴露。另一方面，等到访问属性时才加载资源固然直观，却会让性能和结果是否稳定受文件系统当时的状态影响，所以做离线复现时要把解析结果和来源版本一起冻结。

CLI override 能让你更快调实验参数，但它也会让静态 YAML 与真正跑起来的配置不再一样。CompletionFn 的模型快捷分支便于调用常见 API，同时也把字符串怎样解析变成了运行身份的一部分，因此这套规则也要记录版本。如果 key 不存在、class 没能载入，或者实例不满足 Protocol，系统应该在运行前就停下来，不能留下一份看似已经完成的空结果。

动手实验

假设两个 registry paths 都定义了 shipping.eval，但一个 class 是规则 Eval，另一个却是 ModelGraded Eval，请判断加载器能不能静默挑一个。然后只保留一个条目，让 CLI 把 max_samples 从 100 改成 10，再看报告究竟该记 YAML 中的默认值，还是程序解析后真正使用的值。

列出生成永久运行指纹所需字段，并与本仓库 canonical_digest 的输入思路比较。

预期输出与答案

锁定的实现一看到重复 key 就会拒绝加载，不会悄悄挑前者或后者。报告要记下解析后的 10，也要留住 override 来自哪里，因为静态 YAML 里的 100 只表示默认值。要算出能区分实际运行的指纹，你至少还得把 Registry commit/路径、EvalSpec class/args、CompletionFn class/args、数据摘要和代码版本收进去。

如果只哈希逻辑 key，两个 Registry 里的同名 Eval 就会撞在一起，但只哈希 YAML 又会漏掉 CLI override。两种做法都不足以代表真正跑过的那次运行。

如何核对

依次打开 _load_registry（重复条目在这里被发现）、_dereference 和 make_completion_fn，再在 run() 核对 extra_eval_params 和 completion_args 的合并时点。

本篇不能证明什么

即使你已经确定 Registry 最后解出了哪个对象，也不能由此证明 YAML 写对了、数据得到了充分授权，或者 class 不会带来副作用。Registry 只找配置并实例化对象，它代替不了 Dataset 治理、Sandbox 隔离和 Scorer 有效性验证。

上一节 · 下一节

02 | CompletionFn 与 Sample：谁负责调用，谁负责判断

上一节 · 下一节

本篇要解决什么问题

CompletionFn（补全函数）这个名字容易让人以为它要「完成一次评测」，其实它只接收给定的 prompt，再返回 CompletionResult。这条边界很窄。至于怎样遍历数据、组装 prompt、对照 Reference 以及聚合 metric，都是 Eval 在做。你先把这两层分开，就不会把模型重试、Sample 重复和 Scorer 判断全塞进同一个 callback，也更容易看懂 SolverEval 为什么另立了一条执行边界。

先建立源码地图

锁定的 api.py 只有一百多行，却把 CompletionFn Protocol、CompletionResult 和匹配 helper 都放在了一起。Eval ABC、遍历 Sample 的 helper、CompletionFn 包装以及 SolverEval 则集中在 eval.py，如果还想看 CompletionFn 从哪里开始解析，就顺着调用走到 registry.py。

Eval 接收一组 CompletionFn，completion_fn property 只是让代码取其中的单项时更方便。SolverEval 则要求你恰好传入一个 completion_fn，然后将它包成 Solver，因此两者留给扩展代码的含义不同，不能只凭名字判断。

完整调用链

1. Registry 按模型名或 spec 创建 CompletionFn，对象只需满足协议，不需要知道 Eval key 或最终 Metric。
2. Eval class 从 Registry 相对 data 路径加载 JSONL Sample，可通过 _index_samples 给每条记录稳定运行内 index，并受 max_samples 限制。
3. Eval.run 为每条 Sample 设置 recorder.as_default_recorder(sample_id)，可先记录 raw_sample。
4. Eval 根据 doc 构造 prompt，调用 CompletionFn，取得 CompletionResult，具体实现可能记录 sampling 事件。

5 Eval 解析 sampled response，并调用 record_match、record_metrics 或自定义事件；record_and_check_match 同时记录 expected、picked、sampled 和 options。

6 Eval 累计局部结果，最后返回 dict，而 CLI 不理解每个 metric 的语义，只把 dict 交给 Recorder final report。

7 SolverEval 把单一 CompletionFn 作为 Solver 使用，Eval 代码与 Solver 交互；这允许多步逻辑，却仍需要具体 Eval 定义其状态和评分。

关键数据结构

CompletionFn(prompt, **kwargs) -> CompletionResult 把模型调用的边界圈了出来：CompletionFn 吃进 prompt，CompletionResult 向外给出 completion 或其他响应信息。Sample 通常只是一个 JSONL dict，具体 Eval class 自己解释其中结构，这比强制所有 Eval 共用一种 Sample 模型更灵活，但也削弱了跨 Eval 复用和 schema 检查。

Recorder 写 Event 时会把 sample_id 与 sampling、match 等 data 连在一起，但它没有统一记录 Trial ID、Target resolved identity 或 Attempt ordinal。所以把这些事件接入本仓库时，Adapter（适配器）要补上事先规划的 Trial 和基础设施恢复记录，否则很容易把多次 sampling event 错算成多个 Trial。

实现取舍与失败语义

CompletionFn Protocol 很小，各类模型和 Solver 因此容易接进来，Eval class 也可以自行安排大部分逻辑。代价也很具体：batching、重试、缓存、身份调和和错误分类会散到不同实现中。Eval 子类直接写事件固然开发得快，可你很难只靠静态代码确认每条 Score 都连到了同一组证据字段。

模型 API 超时之后，CompletionFn 内部可能会重试，但核心接口没有把 Attempt 明确记下来。如果 CompletionFn 正常返回了一个错误答案，这仍然是有效的产品结果，应该让 Eval 的 match 或 scorer 来判错。可要是 Sample schema 写错、Reference 缺失，或者 Eval class 抛出异常，系统就应记下 error，并说清它会不会进入最终分母。此外，Solver 自己发起的尝试属于被测算法，外层不能拿它当基础设施恢复。

动手实验

用伪代码写一个确定性 CompletionFn，让它接收金额、返回 shipping fee，并保证相同输入总会得到相同结果。然后再写一个 Eval，依次读入 99、100、101，将它们交给 CompletionFn，记下 expected/picked/match 并算出 accuracy，同时标明哪段代码在做 Target Adapter、Scorer 和 Metric 各自的事。

把金额 100 的 buggy 返回当作正常 CompletionResult，并说明为什么 CompletionFn 外层不应该反复重试，直到得到 fixed 答案。

预期输出与答案

CompletionFn 在做 Target Adapter 的工作，Scorer 负责比较 expected 和 picked，而把三条 match 取 mean 之后才得到 Metric。buggy 结果应该记为 match=false，但这条 Sample 已经完整跑完了。如果外层一直重新调用，直到 match=true 才停，Eval 就把 Reference 泄露给了被测策略，最后算出的分数也就没有意义了。

报告还要保留三条 Sample 事件，并把分母固定为 3，不能因错误类型不同就改变分母。如果某条记录因基础设施错误而没有产生结果，就把它明确标成 unscored/blocked，别从 mean 列表里直接删掉。

如何核对

阅读 CompletionFn、DummyCompletionFn 与 record_and_check_match，再对照 Eval、SolverEval 和 _index_samples，核对职责确实分开。

本篇不能证明什么

协议把扩展代码从哪里接进来说清了，却不能证明具体 Eval 子类已经正确处理缺失、并发和网络重试，也不能保证响应真的来自所声明的模型版本。最后能不能信这份证据，还得回到具体代码和实际运行记录上。

上一节 · 下一节

03 | Recorder 与 Metric 边界：事件很多，推断仍要另建合同

上一节 · 下一节

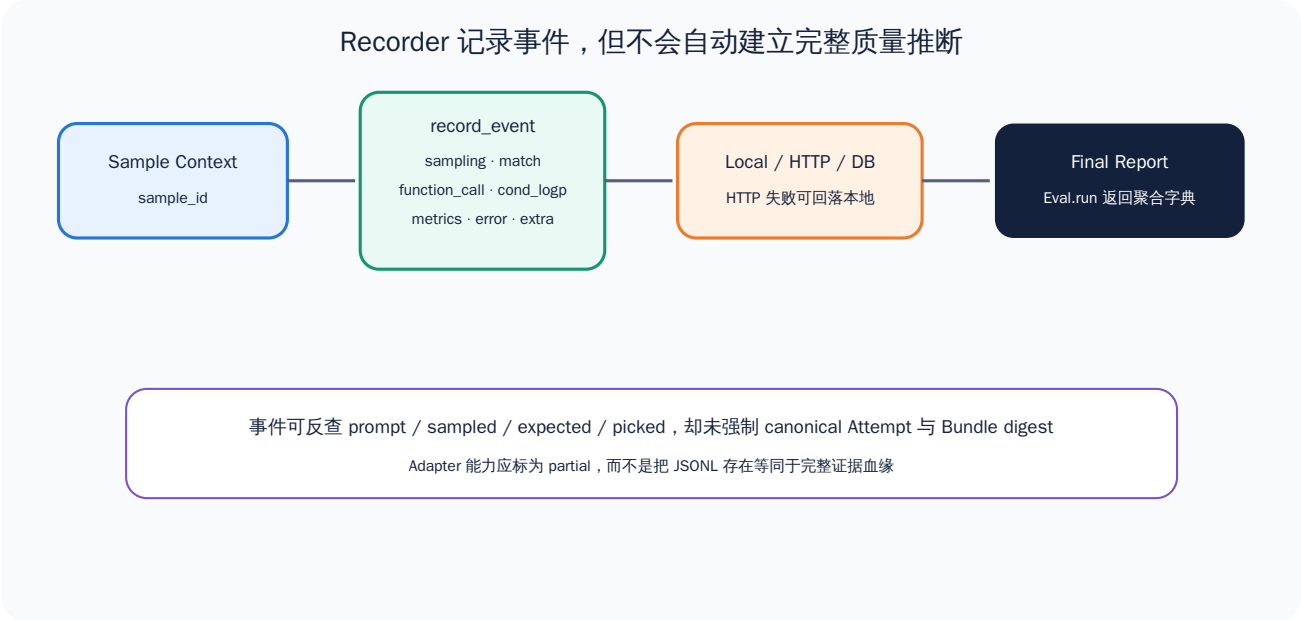
本篇要解决什么问题

OpenAI Evals 里的 Recorder（记录器）能记下 sampling、match、function_call、cond_logp、metrics、error 和任意 extra，LocalRecorder 还可以写入 JSONL，并在 HTTP 失败后改存到本地。但事件已经写下来，不代表你就知道它们之间有没有因果顺序、哪次执行才是 canonical，或者还有多少样本没评分。Recorder 也不会替你给出 accuracy 的分母和发布结论，因此这一篇要把记录层和推断层分开，这两层不能互相顶替。

先建立源码地图

锁定的 record.py 把 RecorderBase、Event、default recorder ContextVar 和 record_event 放在一起。其中 LocalRecorder 与 HttpRecorder 用不同方式存事件，你选哪一个，证据最后就会落到哪里。CLI 用 build_recorder 挑选后端，之后由 run() 补上 token usage 并写入 final report，match helper 则在 api.py 里。

完整调用链



1. CLI 根据 dry-run、record_path、remote URL 等选择 Recorder。每个 Recorder 持有 RunSpec。
2. Eval 进入 as_default_recorder(sample_id)，ContextVar 暂存当前 Recorder 与 sample_id；离开上下文后恢复旧值。
3. CompletionFn/Eval 调用全局 helper，最终进入 record_event(type, data, sample_id)；没有显式或上下文 sample_id 时抛错，防止事件成为孤儿。
4. LocalRecorder 把 Event 写入本地 JSONL；HTTP Recorder 发送远端，异常时写 local fallback 并提示。
5. Eval.run 返回 final report，而 CLI 从 sampling events 可累计 token usage，再单独调用 record_final_report。

6 消费者可按 type 查询 events，但 Metric 的聚合公式主要由 Eval 返回 dict 决定，Recorder 不验证分母或统计假设。

关键数据结构

Event 会把 run、sample、type 和 data 连到同一条记录上。具体写什么由事件类型决定：sampling data 可以带 prompt/sampled/usage，match 可以带 correct/expected/picked/sampled/options，function_call 可以带 name/arguments/return，error 记消息和异常，metrics 则由 Eval 自己填键值。

RunSpec 保存运行头，final report 给出聚合后的输出，但代码没有强制它们通过 ScoreRecord 和 MetricEstimate 相互引用。final report 也未必会列出每个 Metric 用过哪些 Event ID，所以证据 Adapter 要留住原始 Event，同时另行生成 Observation 与 Score 之间的明确血缘。

实现取舍与失败语义

通用 Event API 允许实验快速记下各种事实，但调用者要自己约定 type/data schema，不同 Eval 写出来的事件因此可能同名不同义。ContextVar 比全局变量更适合并发上下文，不过你仍要测它能不能正确穿过线程和异步任务边界。Local fallback 让记录更不容易丢，却会把证据分到不同位置。如果远端和本地都留下了记录，后续汇总时就必须去重，并标明数据真正落到了哪个 sink。

记录服务挂了，说明 Harness 的基础设施出了故障，不能因此改写模型 Score，而如果 sampling event 已经存在，match 却没有出现，这条样本可能处于 unscored 状态，你不能默认它失败，也不能让它悄悄退出分母。final report 若只给平均值，却没有列出 Sample 事件，它能提供的证据就只能标成 partial，而 Recorder 自己也不执行发布 Gate，所以阈值判断应当交给独立的政策层。

动手实验

给你 100 个 raw_sample、90 个 sampling、80 个 match=true、10 个 match=false、5 个 error，再加上 5 个只有 sampling 却没有 match 的事件。请据此分别设计事件完整性报告、Score 状态映射、Metric denominator 和 Gate 输入，并说明 HTTP fallback 重复上传后要用什么依据去重。

再把这些字段映射到 Reference Harness 的 TraceEvent、ObservationBundle、ScoreRecord 与 MetricEstimate，标出 unavailable/partial。

预期输出与答案

计划里共有 100 个 Sample，所以分母是 100，其中 80 个 passed、10 个 failed、5 个 error，error 还要按类别分别映射为 blocked 或 invalid，另有 5 个 unscored。如果只用 90 个 match 算出 88.9%，你不能立刻宣布通过，因为做质量决策时必须把这 10% 的缺口摆在明面上。去重时至少要结合 run_id、sample_id 以及 event identity 或内容摘要，不能只因事件文本相同就删除。

把 Event 映射成 TraceEvent 时只能标为 partial，因为原始事件不一定记了父子关系和 sequence。sampling 与 match 虽然可以合成 Observation，但其中还是没有 canonical Attempt digest，而 match 转成 Score 之后，final report metric 若没有 score_ids，同样只能标成 partial。核心 Recorder 也不执行 Gate，你要在独立政策层补上这一步。

如何核对

依次阅读 as_default_recorder、record_event、LocalRecorder 和 HttpRecorder 的 fallback，再在 cli/oaieval.py 核对 final report 与 token usage 是入口后处理，而不是 Recorder 自动推断。

本篇不能证明什么

你能读取 JSONL，fallback 也确实执行成功，依然不足以证明证据没有缺口、事件未被改写，或者 Metric 统计有效，因为 Recorder 只是把观察存下来。要做出质量结论，你还得补上身份、血缘、分母、不确定性和 Gate Policy。

上一节 · 下一节

Promptfoo
源码课程：配置矩阵怎样变成可判定的评测运行

上一节 · 下一节

本篇要解决什么问题

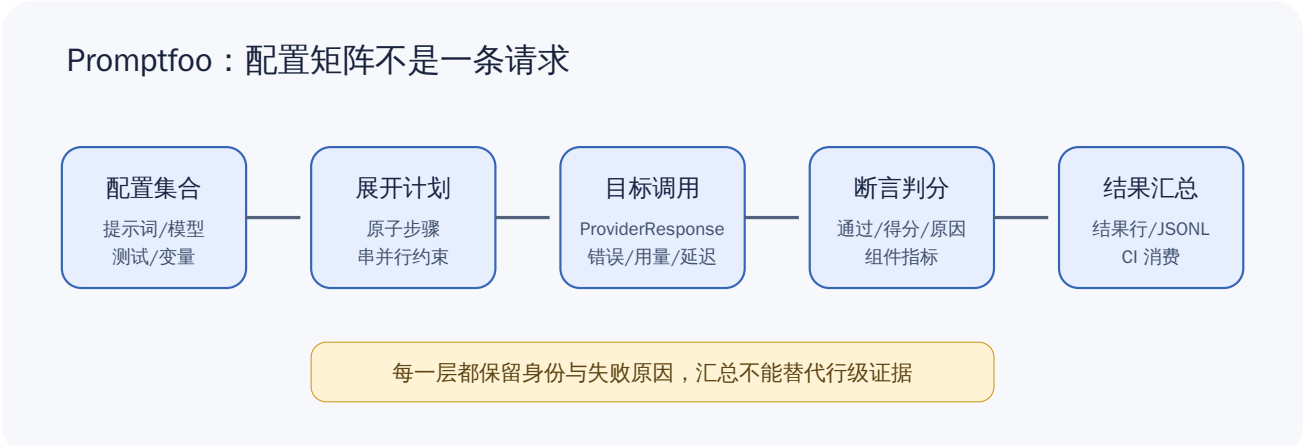
Promptfoo 常被概括成「用 YAML 对比多个提示词和模型」，可这句话省掉了 Eval Harness 真正干的大部分工作。配置里的 prompt、provider、test、变量、重复次数和断言，合在一起也还不是一条请求，它们只是规定了一张运行计划。程序要先展开这张计划，再逐步调度、执行、判分和持久化，每一步都会改变最后能看到的证据。这组课程会说清配置对象如何变成原子测试、Provider（提供方）如何统一承接调用、运行器为什么有时必须放弃并发，以及断言如何把响应判成通过、得分或失败原因。最后再看 CI 手里的汇总数字，与每条运行留下的证据究竟是什么关系。

课程锁定在 ce89186a22c59543f4f71a55d42442ff3f0e3654，只根据这个版本里列出的五个源码文件讲可以核对的事实，同时说清这些事实适用到哪里为止。Promptfoo 还有红队、缓存、Web 界面、远程服务和大量 Provider，但这组入门课不承诺覆盖这些部分。

先建立源码地图

站点	锁定文件	本课关注的问题
总调度器	src/evaluator.ts	测试展开、运行步骤、并发、超时、判分和统计
运行时契约	src/evaluator/runtime.ts	Store、结果写入器和运行时依赖
Provider 装配	src/providers/index.ts	字符串、配置、函数和文件怎样解析为 ApiProvider
断言执行	src/assertions/index.ts	单断言、断言集合、模型判分和 trace-aware 判分
公共类型	src/types/index.ts	AtomicTestCase、EvaluateResult、GradingResult 与选项

完整调用链



1. 配置层提供 prompts、providers、tests、scenarios、defaultTest 与运行选项；加载后的 TestSuite 仍不是执行队列。
2. Evaluator 合并默认测试和场景，把变量数组展开为组合，并为每个 test × vars × prompt × provider 生成 RunEvalOptions。重复运行和比较断言会继续改变步骤数与分组关系。
3. Provider 解析器把字符串 ID、内联对象、函数或外部文件统一为带 id() 与 callApi() 的 ApiProvider；运行步骤因此不必知道某家模型 SDK。
4. 调度器检查 conversation 变量、storeOutputAs 和持久浏览器会话等跨步骤状态；这些特性存在时——并发会被降为 1。其余步骤用受限并发执行，并共享速率限制注册表。
5. 单步渲染 prompt 和变量，调用活跃 Provider，得到 ProviderResponse；空响应、Provider 错误、超时和目标不可用分别进入不同结果分支。
6. runAssertions 对 assertion 或 assertion set 求值，形成 GradingResult；Evaluator 再把 success、score、namedScores、failureReason、tokenUsage 和 latency 写回 EvaluateResult。
7. Store 与 JSONL writer 持久化单行结果，末尾聚合每个 prompt 的通过数、失败数、错误数和得分，供报告或 CI 决策使用。

关键数据结构

AtomicTestCase 已经表达了可以运行的测试，其中保留 vars、assert、provider override、metadata 等字段。RunEvalOptions 在此基础上再选定具体的 prompt/provider/testIdx/promptIdx，并把超时、缓存和信号一起带进单次执行。目标调用会从 ProviderResponse 这条边界返回结果，其中可以有 output，也可以有 error、tokenUsage、latencyMs 和 metadata。EvaluateResult 把原测试、响应、评分、成功状态和失败原因留在同一行运行结果中，GradingResult 则只收 pass、score、reason、namedScores、componentResults 和 assertion 等判分信息。

这些结构会相互引用，但谁都不能代替谁，类型边界必须留着。Provider 返回了结果，不表示断言就能通过，而断言通过了，也不能证明整个 Trial 有统计代表性，更不能拿 prompt 汇总直接当作发布 Gate。Reference Harness 会用 Trial/Attempt 和独立 Gate 继续把这几层意思拆开。

实现取舍与失败语义

配置矩阵让同一组测试可以快速横向比较多个模型和提示词，但程序一旦展开矩阵，配置的一行就不再对应结果的一行，排查数量异常时要先算清展开规则。Provider 统一接口之后，新接一家模型会更省事，可调用中间仍然经过了字符串解析、插件加载和运行时配置。所以复现记录要保存解析后的 Provider 身份，不能只留原始短名。

调度器降低并发，是因为会话历史或前一步输出参与后一步输入时，乱序会直接改变样本本身。Provider 重试、缓存命中和断言发起的模型调用分属不同恢复层，底层多调了几次，不能就多算几个独立样本。如果持久化失败，系统会记下这个故障，并尽量留住 JSONL 权威副本。它能保住已经产生的证据，却不能自动证明整个结果集没有缺口。

动手实验

假设配置里有 2 个 prompt、3 个 provider 和 4 个 test，其中一个 test 的变量 tone 又有 2 个值。先别跑代码，手算最小步骤数，然后加入 repeat: 2 再算一次。随后把字段分到配置身份、目标响应、评分结果和运行汇总四类里，最后解释为什么某个 test 一旦用了会话变量，即使把并发设成 8，也不能真的并发执行。

运行仓库的离线检查：

```
python scripts/sources.py verify
```

```
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

如果四个 test 都只有一种变量组合，基础步骤数就是 $2 \times 3 \times 4 = 24$。如果其中一个 test 因 tone 展开成两个原子测试，总数就会变成 $2 \times 3 \times (3 + 2) = 30$，全部再重复两次，结果就是 60。真实配置还会受 scenarios、过滤、已完成结果恢复、比较断言和 provider override 影响，因此这里只用最小模型说明展开规则。

会话变量要读前序结果，一旦并发，当前输入就可能读到错的历史，或者读到与预期不同的历史，因此调度器应该让相关步骤串行。课程测试不依赖模型密钥，也不发起网络调用，它只检查课程结构和锁定链接，并没有执行 Promptfoo 的整套集成测试。

如何核对

先在 src/evaluator.ts 依次找 buildTestsFromSuite、generateVarCombinations、appendRunEvalOptionsForProvider、adjustConcurrencyForSerialFeatures、runConcurrentEvalSteps 和 runEval。再到 src/providers/index.ts 核对 Provider 的多种输入形式，最后在 src/assertions/index.ts 核对断言聚合。

本篇不能证明什么

这一篇不能证明某份 Promptfoo 配置已经适合生产发布、模型判分足够可靠、缓存从未被污染，或者所有 Provider 都会做出一样的行为。CI 偶尔一次全绿，也不代表结果具有统计显著性。本篇只给出锁定版本的运行骨架，并告诉你该去哪里核对。

上一节 · 下一节

Promptfoo 配置与 Provider：先解析身份，再生成请求

上一节 · 下一节

本篇要解决什么问题

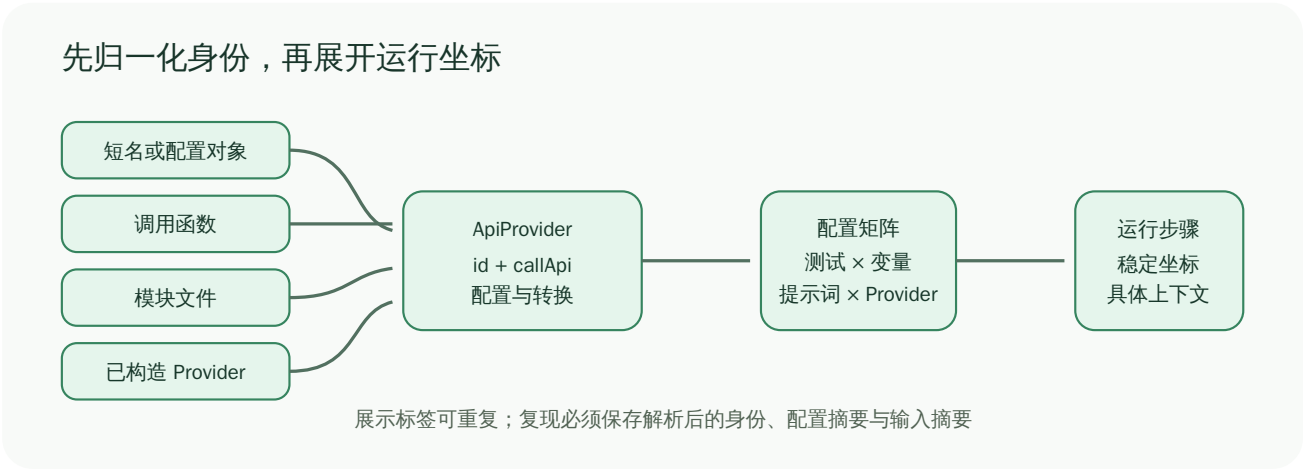
看到 providers: [openai:gpt-4o-mini] 时，你可能以为 Evaluator 拿这个字符串直接调 SDK 就行，但实际代码还得识别短名、带 config 的对象、JavaScript 函数、模块文件和多 Provider（提供方）配置。prompt 自己可以带标签、模板和配置，测试还能换掉默认 Provider。如果不先把这些写法归一，执行层就得到处判断输入类型，也很难准确记下这次究竟调了谁。这一篇就来看 Provider 如何解析，prompt、provider 和 test 又是在哪里组合起来的。

源码版本锁定之后，你可以逐段核对每条解析路径，但这一篇不会穷举 Promptfoo 支持的所有 Provider。同一个短名将来可能指向另一种实现，所以核对时要同时看锁定 commit，以及解析后的 id()、配置和输入，不能只看那个短名。

先建立源码地图

源码位置	作用	阅读问题
src/providers/index.ts	loadApiProvider、resolveProvider、loadApiProviders	输入形式怎样变成 ApiProvider
src/evaluator.ts	合并 prompt 配置、生成 RunEvalOptions	解析对象怎样进入运行矩阵
src/types/index.ts	TestSuite、RunEvalOptions、EvaluateResult 等契约	哪些字段跨越层级

完整调用链



1. loadApiProviders 接受单个或多个 Provider 引用。已经满足 ApiProvider 契约的对象可直接使用；函数被包装为 callApi；字符串和配置对象进入 loadApiProvider。
2. 路径型配置可导出一个或多个 Provider。单 Provider 入口若读到多个 Provider 会明确报错，避免调用者误以为只选择了第一个。
3. 解析阶段合并 basePath、环境、Provider options、label、transform、delay 与 inputs。带模板的动态配置保留到实际 callApi() 上下文再渲染，而不是在全局加载时过早求值。

4 Evaluator 先构建 tests，再对每个 test 计算变量组合；随后按 prompt 和 provider 逐层附加 RunEvalOptions。测试级 Provider override 可以替换套件默认 Provider。

5 createRunEvalOption 固定 testIdx、promptIdx、具体 ApiProvider、具体 Prompt、vars、timeout 和运行控制项。到这里，一个配置矩阵才变成可调度步骤。

6 单步运行时把 Provider 配置与 prompt config 合并为调用上下文，渲染模板后调用 provider.callApi(renderedPrompt, context, options)，并把返回值标准化为结果行。

关键数据结构

ApiProvider 关心的不只是厂商名称，还要给出稳定身份和调用能力。它用 id() 标明结果归谁，用 callApi() 圈出目标调用边界，再用 label/config/transform/delay/inputs 说明额外行为。Prompt 会把 raw、label 和 config 一起留下，否则报告里只有一大段模板，你根本分不清它属于哪个候选。TestSuite 收着尚未展开的配置，AtomicTestCase 则是合并 default/scenario 后的测试，直到 RunEvalOptions 出现，程序才真正得到某个 provider × prompt × test × vars 的执行单位。

想复现这次运行，清单里至少要记下锁定 commit、解析后的 Provider ID 与 label、非敏感配置摘要、prompt 摘要、变量值、测试断言、输入文件摘要和运行选项。环境变量里若有密钥，只记它的名称或来源，别把密钥本身写进证据包。

实现取舍与失败语义

输入既可以是简单 YAML，也可以一路扩展到自定义函数，用起来很灵活，但配置解析也因此进入了可信计算范围。如果一个文件导出多个 Provider，解析器会拒绝暗中挑选，直接报错，以此保证你能预料它会调谁。动态配置则等到真正调用时才渲染，因为只有那时才拿得到 per-test 变量。于是，两个 Provider 加载时看起来一样，各自测试真正发出的请求却可能不同。

如果 Provider 没能解析，说明装配就出了错，系统应该在调用目标前就报出来。callApi 返回 error，说明错误落在 Provider 调用这条路径上，不能单凭这个字段断定目标已经真正执行。transform 或 prompt 渲染失败，问题则在输入或适配层。这三种情况要是都被压成 success: false，用户便无法判断应该修哪一层。还要注意，缓存命中只是在说这次执行的结果从哪里来，它没有新增一条独立观察。

动手实验

设计三个逻辑相同的 Provider，分别用短名、内联函数和模块文件来实现，再写出它们各自的 identity 字段，说明为什么只留展示 label 无法复现。随后加入一个 test 级 Provider override，画出它与 suite 默认 Provider 谁优先，最后给 prompt 配置加入 temperature，并指出程序应该在哪一层合并它、又该怎样把它写进审计信息。

离线核对命令：

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

短名至少要留下解析后的实现 ID 和配置，函数要留下模块或代码摘要以及稳定 label，文件则要留下规范化相对路径、文件摘要和导出对象 ID。展示 label 可能重复，也可能被用户修改，所以它不能单独证明身份。测试级 override 只会改变该测试生成的步骤，其他测试仍然调用 suite providers。创建调用上下文时，程序应该按明确优先级合并 prompt config 与 Provider/prompt 配置，再把非敏感快照随结果或运行清单一起保存。

如果一个文件导出了多个 Provider，你却用单 Provider API 去加载，程序应该给出明确错误，不能暗中挑选其中一个。如果配置里有每个测试自己的模板变量，就等运行到当前测试时，再根据当时的 vars 渲染。

如何核对

先在 src/providers/index.ts 从 loadApiProviders 追到 loadApiProvider，观察函数包装、对象直通、文件配置和多导出拒绝逻辑，然后再到 src/evaluator.ts 追 appendRunEvalOptionsForTestCase、appendRunEvalOptionsForVars、appendRunEvalOptionsForProvider 和 createRunEvalOption。

本篇不能证明什么

即便 Provider 已经加载成功，你仍然不知道凭据是否有效、模型版本是否固定、外部 API 能否访问，也无法确认服务端真的按所声明的参数执行了请求。配置摘要替代不了完整的供应链锁定，自定义函数是否安全、沙箱是否圈住了它，也都要另外审计。

上一节 · 下一节

Promptfoo 测试运行时：展开、串并行与结果落盘

上一节 · 下一节

本篇要解决什么问题

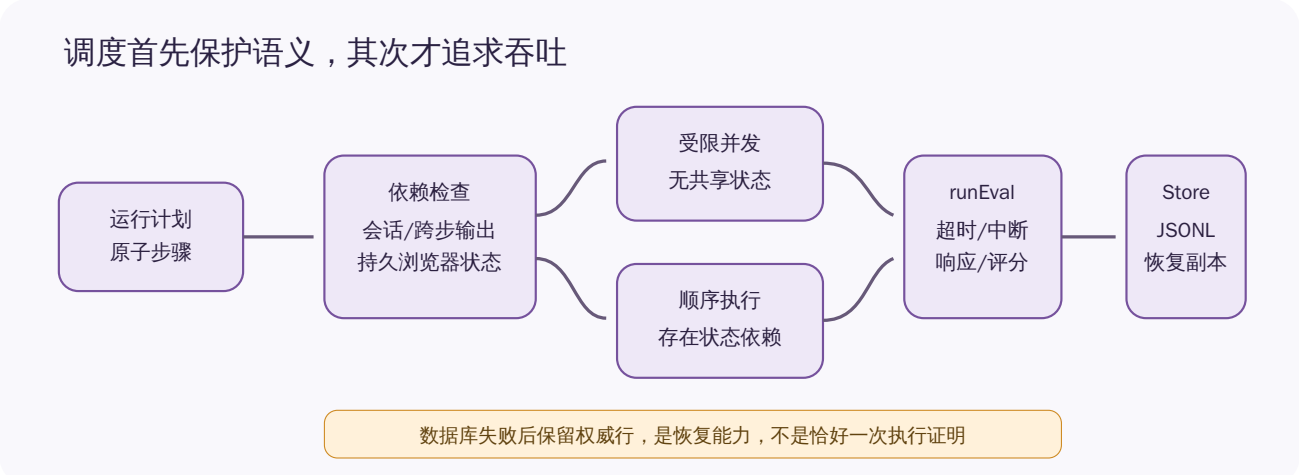
遇到性能问题，人们很容易怪「并发太低」，遇到正确性问题又容易怪「模型随机」，但下判断前得先看懂 Promptfoo 如何生成并调度每个运行步骤。变量数组会把测试展开成更多份，resume 会滤掉已经跑完的步骤，会话历史和跨步骤输出则会逼着相关步骤串行。调度规则从这里开始改变运行语义，而单步超时、全局超时和持久化失败又会留下各自不同的结果。这一篇会跟着一个原子运行往前走，看它怎样从队列进入结果存储，中间状态又怎样随之变化。

你不用背下每个私有函数，应该检查的是计划有没有正确展开、依赖有没有按顺序执行、错误有没有分层，以及最后以哪份结果为准。

先建立源码地图

源码位置	责任	需要核对的行为
src/evaluator.ts	Evaluator、runEval、串并行调度	计划、超时、速率限制、统计
src/evaluator/runtime.ts	Store 与 writer 契约	结果怎样追加、读取和恢复
src/types/index.ts	运行选项与结果类型	timeout/cache/abort 和结果字段

完整调用链



1. Evaluator 合并 default tests 与 scenarios，准备变量并应用 input transform，得到 AtomicTestCase 列表。
2. 变量数组经 generateVarCombinations 展开，随后按 Provider 和 Prompt 生成 RunEvalOptions。resume 模式会读取已有结果并过滤已完成坐标。
3. adjustConcurrencyForSerialFeatures 检查 conversation 变量、storeOutputAs 和浏览器持久会话。任何一步依赖前一步状态时，最大并发被改成 1；其余步骤进入 forEachOfLimit。

4 调度器为每步检查 abort/global duration，并给单步包裹 timeout；共享 RateLimitRegistry 可以根据 Provider 的限流反馈调整并发。

5 runEval 创建状态，渲染 prompt，构造 Provider context，调用 Provider，再处理 trace、delay、transform、空响应与 grading。

6 EvaluateResult 先更新 prompt metrics 和全局 stats，然后追加到 Store 与 JSONL writer。数据库追加失败会登记 persistence failure；之后在内存保留权威副本，供终局合并覆盖陈旧行。

7 中断时保存已完成进度；目标被判定不可用时结束剩余步骤并保存当前结果。最终 Store 汇集结果、prompts、vars 与统计。

关键数据结构

testIdx + promptIdx + provider + vars 合在一起才能定位一次运行，你不能只看它在数组里排第几个。EvalProcessingContext 会收住 concurrency、共享变量集合和目标不可用状态，也会跟着记录它们在本次运行中如何变化。EvaluateResult 把 testCase、prompt、provider、response、gradingResult、success、score、namedScores、failureReason、latency 和 tokenUsage 收进同一条结果，Store 则通过 appendResult、readResults、hasResultPersistenceFailure、recordFinalResult 等接口与 Evaluator 对接。这样 Evaluator 就不用绑死在某一种数据库实现上。

Prompt metrics 可以写进报告做汇总，但它们代替不了每行结果留下的证据。只看总通过数，你既不知道哪条输入失败、有没有命中缓存、究竟是哪个 Provider 报错，也无法追出比较断言怎样改变了分数。

实现取舍与失败语义

受限并发可以提高吞吐，检测到状态依赖时改用串行，又能保住有状态语义，但没有声明的共享状态仍然可能漏过检测。自适应限流会减少连续出现的 429，也会重新排列每步等了多久，所以解释 latency 时要把排队时间与 Provider 真正调用的时间分开。单步 timeout 会留下一条失败记录，全局 max duration 却可能让剩下的步骤根本没有结果。只看失败率，这两种缺口就会混在一起。

持久化会同时借助「数据库 + 流式 JSONL + 失败后内存权威副本」，尽量避免进程跑到中后段时丢掉已有结果。恢复时要把这些来源合到一起，仍然必须靠稳定坐标找回同一条运行，再按明确规则去重。中断保存让运行可以恢复，但恢复前后对同一逻辑测试发起的多次底层调用，不能被算成多个独立 Trial。

动手实验

构造六个 RunEvalOptions：前三个不读写状态，第四个写入 storeOutputAs，第五个读取该值，第六个使用 conversation 变量。请分别排出并发 3 和并发 1 时的执行顺序，再指出哪一种才能保住原有语义。然后模拟第三条结果没能写进数据库，却成功写入 JSONL，据此给出恢复时的合并键和优先级，并分别记录 Provider 500、断言失败、单步超时、全局到时和用户中断。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

前三个无状态步骤可以在限制内并发，但只要出现跨步骤存取或会话历史，相关序列就必须按原顺序跑，当前实现会直接把整次运行的并发降到 1。恢复键不能省，至少要包含稳定的运行身份以及 test/prompt/provider/vars 坐标。如果数据库已经报告持久化失败，最终就应该让内存或流式权威行覆盖数据库中的陈旧行，同时删掉重复记录。

Provider 500 表示目标出错，断言失败说明有效响应没有满足判据，单步超时应该留下结果行，全局到时则说明仍有步骤没来得及运行。如果是用户中断，还要记下为什么终止。界面完全可以把后四种情况都显示成红色，但机器读到的原因必须把它们区分开。

如何核对

先在 src/evaluator.ts 核对 adjustConcurrencyForSerialFeatures、runSerialEvalSteps、runConcurrentEvalSteps、processEvalStepWithTimeout、persistEvalRow 与 saveInterruptedEval，再到 src/evaluator/runtime.ts 核对 Store 的读取、追加和 persistence failure 契约。

本篇不能证明什么

即使代码里已经有 resume、JSONL 和自适应限流，你也不能据此证明每个动作恰好执行了一次、跨进程事务完整、远程 Provider 能幂等处理请求，或者中断后的统计没有偏差。课程并未对真实 API 注入故障，这些问题都要另做专项验证。

上一节 · 下一节

Promptfoo 断言与 CI：从响应判分到门禁还差哪一步

上一节 · 下一节

本篇要解决什么问题

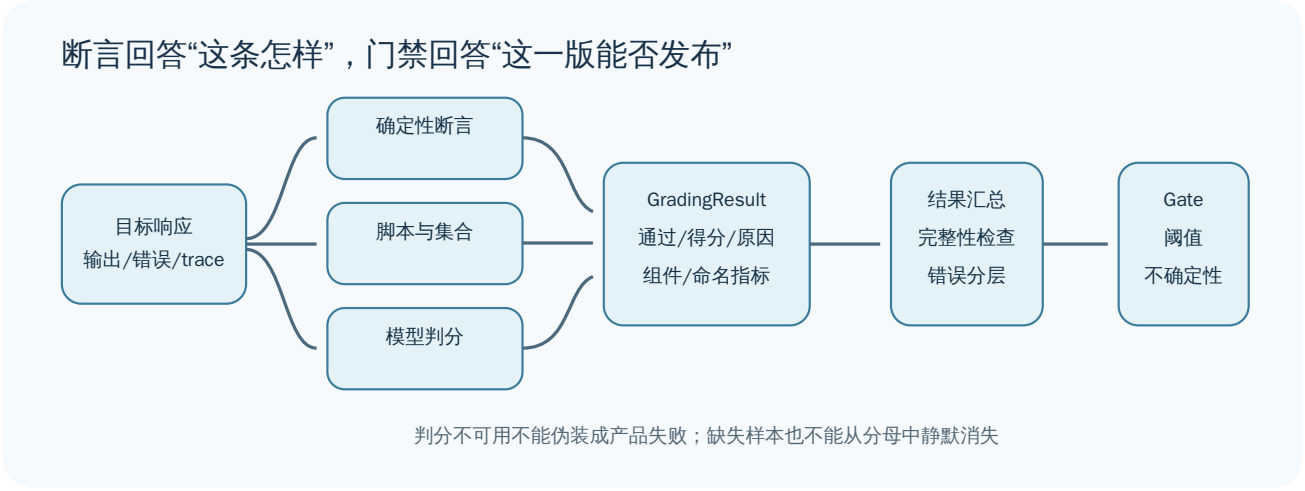
同一个响应可以同时交给 contains、JSON schema、自定义脚本和模型判分，每条断言还可以带权重、阈值、命名指标、否定前缀或嵌套集合。所以「pass」不是 Provider 自带的属性，它是一组判据看过某次响应后聚合出的结论。这一篇会追着 runAssertions 看它如何生成 GradingResult，再看 Evaluator 如何把这份判分写进结果和 metrics，最后说清这些统计为什么还不是一道完整的发布 Gate（门禁）。

读完以后，你应该能分清是某条断言没通过，还是执行本身出了错，并能说明模型判分消耗了多少 token、断言集合留下了哪些组件结果、比较型断言为什么要延后判分，以及 CI 应该从哪个稳定字段读阈值。

先建立源码地图

源码位置	责任	阅读焦点
src/assertions/index.ts	断言分派、求值、聚合与比较	pass/score/reason 怎样生成
src/types/index.ts	Assertion、AssertionSet、GradingResult	判分契约保存什么
src/evaluator.ts	grading、comparison、metrics 和统计	单行结论怎样汇总

完整调用链



1 ProviderResponse 进入 gradeRunEvalResponse；无断言时返回专门的 no-assert 结果，而不是伪造一个命中的断言。

2 runAssertions 将普通 assertion 与 assertion set 展开为待执行项，为每个集合建立子聚合器。若存在 trace-aware assertion，会用 traceId 加载相关 trace 数据。

3 单项进入 runAssertionInternal：模板值先按当前 vars 渲染；脚本型值可能加载外部函数；base type 与 not- 反向语义被标准化；最终 handler 接收 AssertionParams。

4 模型判分型断言可通过 grading Provider 发起额外调用；它返回的 tokenUsage 属于评分成本。不应混入目标模型成本后丢失来源。

5 每个结果写入主 AssertionsResult；属于集合的结果同时写入子聚合器。结束后集合按 threshold/权重形成组件结论，并保留 componentResults。

6 Evaluator 把 GradingResult 应用于 EvaluateResult：设置 success、score、namedScores、reason 和 failureReason，同时累计 assertion token、通过数、失败数。

7 select-best、max-score 等比较断言必须等待同组候选结果齐备后执行；其结论可能回写某行成功状态与 prompt metrics。

8 CI 可消费结果或汇总并按配置退出非零，但真正发布 Gate 还需要预注册数据版本、统计规则、不确定性处理和缺失结果策略。

关键数据结构

Assertion 用 type、value、threshold、weight、metric、provider 等字段写一条判据，AssertionSet 则在 assert 数组里继续嵌套，以此组合多条判据。运行时，AssertionParams 会把 assertion、标准化的 baseType、inverse、output、renderedValue、test、provider、ProviderResponse 和上下文一起交给 handler，GradingResult 再至少记下 pass、score 和 reason，也可以继续带上 namedScores、tokensUsed、componentResults、assertion 与 metadata。

EvaluateResult.success 给出整条结果最后通过与否，failureReason 区分目标出错还是断言失败，namedScores 则允许同一条结果携带多个指标，PromptMetrics 再把 testPassCount、testFailCount、assertPassCount、assertFailCount 和 score 继续汇总起来。如果最后只导出 success，你就看不到结果为什么失败、又是哪条判据变了，这些诊断证据会全部丢失。

实现取舍与失败语义

统一分派之后，各种判据可以共用模板渲染、反向语义和错误处理，但动态脚本和模型判分也会带来外部代码安全、Judge（裁判模型）偏差和网络失败等问题。集合完成聚合后仍会留下 componentResults，因此比只返回一个总分多了一层可核对证据。但阈值和权重依然是产品政策，你必须记下它们用的版本和选用理由。

断言得到 false，说明被测对象没有满足契约，而 handler 抛错则说明这次根本无法得出结论。评分 Provider 不可用时，故障出在 Judge 一侧，trace 缺失则会让 trace-aware 判据无从判定，这四件事不能混。发布规则如果把后三种情况当成普通负样本算进失败率，就会把基础设施故障伪装成产品质量下降。可你也不能丢掉这些记录，然后拿缩小过的分母宣布通过。

动手实验

给同一个响应设计三项断言，格式正确、事实一致和安全约束的权重分别是 1、2、3，然后计算全部通过、只有事实失败以及 Judge 超时时，各自会得到什么组件结果和总体状态。随后为候选 A/B 设计一个 select-best，说明为什么 A 刚返回时还不能判分。最后写出 CI Gate 的输入契约，把结果集完整性、错误率、核心指标、最小样本数和置信区间策略都收进去。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

三项断言都通过时，总体状态是 pass，但每项组件结果依然要保存。如果事实项失败，总体规则就要按配置的阈值计算，同时留下该项的 reason，不能只写一个总分。Judge 超时时，结果应该标成评分不可用或评测错误，不能把它当成事实断言得到 false。A/B 必须等同组输出到齐以后再比较，否则无法得出相对结论。

一道合格的 Gate 要先检查预期坐标是否全部出现，并排除无法解释的重复，然后把产品失败和 harness error 分开。接着按预注册方案计算核心指标及其不确定性，最后再对照明确阈值给出 pass、fail 或 inconclusive。Promptfoo 的 assertions 和 CI 退出码提供了重要构件，但这组课程不会把它们直接当成一套完整的发布制度。

如何核对

先从 src/assertions/index.ts 的 runAssertions 追到 runAssertion、runAssertionInternal 和 handler 映射，观察集合、并发与 trace-aware 分支，再到 src/evaluator.ts 查 gradeRunEvalResponse、applyGradingResult、comparison merging 与 metrics 更新。

本篇不能证明什么

无论断言类型多么丰富，它们都无法证明 Judge 与人类会做出一样的判断、指标真的反映线上业务、阈值设得合理，或者结果具有统计显著性。CI 退出码也不是发布授权，真正的组织流程仍然要独立处理数据治理、人工复核、例外和回滚。

上一节 · 下一节

DeepEval 源码课程：从测试用例到 MetricData 的评测生命周期

上一节 · 下一节

本篇要解决什么问题

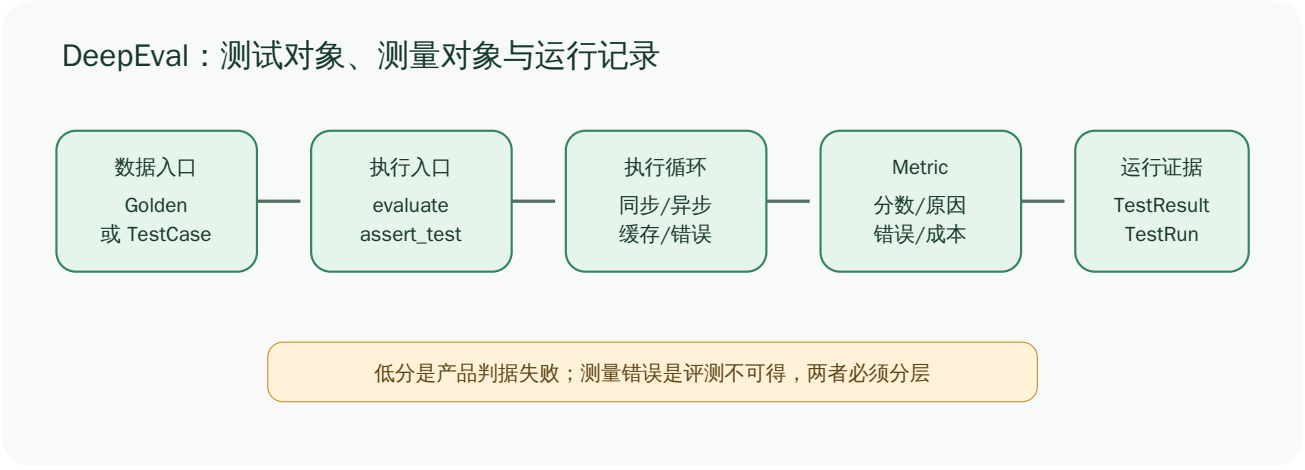
人们常把 DeepEval 说成「像 pytest 一样测试大模型应用」，可源码做的事远不止检查一次 assert score >= threshold。输入可以是已经带有 actual_output 的 LLMTestCase（大语言模型测试用例），也可以是等着用户代码去执行的 Golden（黄金样本），指标既能同步或异步运行，也能测单轮、对话或 trace。再加上缓存、忽略错误、并发和测试运行管理器，程序怎样执行、结果怎样记录都会跟着变化，所以这组课程会沿着调用链，看这些对象最后怎样组成 TestResult、MetricData 和一次 TestRun。

课程锁定版本为 a2e0d4cfd3118352d321c1c84bdeba17d4a201bc，DeepEval 还提供许多具体指标、追踪功能、平台集成和 Agentic 入口，这里只讲锁定源码里能够直接核对的执行骨架，再把各环节对应到统一的 Dataset → Sample → Target → Scorer → Result 模型。遇到 Golden、TestCase 与 Metric 时，正文仍沿用 DeepEval 自己的叫法。

先建立源码地图

站点	锁定文件	责任
公开入口	deepeval/evaluate/evaluate.py	assert_test、evaluate、配置与 TestRun 收尾
执行循环	deepeval/evaluate/execute/loop.py	测试、trace、metric 的同步/异步执行与错误处理
指标基类	deepeval/metrics/base_metric.py	score、threshold、success、reason 与测量接口
数据集	deepeval/dataset/dataset.py	Golden/TestCase 集合、身份和 agentic iterator
单轮用例	deepeval/test_case/llm_test_case.py	input、actual_output、expected_output、context、tools 等观测

完整调用链



1. 调用者选择两种边界：把已执行结果封装成 LLMTestCase 交给 evaluate，或遍历 Dataset 的 Golden，在用户代码产生输出和 trace 后由 agentic loop 构造 TestCase。
2. evaluate 验证输入与 metrics，配置 AsyncConfig、DisplayConfig、CacheConfig、ErrorConfig，并重置或取得全局 TestRunManager。
3. 同步路径调用 execute_test_cases，异步路径通过事件循环调用 a_execute_test_cases；指标对象会按运行模式初始化并在每个测试上测量。
4. 指标的 measure/a_measure 写入 score、reason、error 等状态；基类 is_successful 按 threshold、strict_mode 和 error 得出成功语义。
5. 执行器把指标状态转换为 MetricData，附着到 TestResult/API test case，更新 TestRun。缓存可复用先前评分，ignore_errors 决定错误是终止还是保留为结果。
6. evaluate 输出控制台/文件报告，写入运行时长和超参数，保存临时 TestRun，并根据 CLI 与普通调用模式选择由谁完成 wrap-up。
7. assert_test 复用执行器，但在任何有阈值的指标失败或出错时抛出 AssertionError，适合测试框架门禁，不等同统计发布判定。

关键数据结构

Golden 更像一份等着执行的样本规格，它存着输入、期望和上下文，通常还没有被测应用生成的 actual_output。LLMTestCase 记录的则是一次可以评分的观测，里面会放 input、actual_output、expected_output、context、retrieval_context、tools_called、expected_tools、metadata 等内容，这两种对象不能互换。BaseMetric 是有状态的 Scorer，测量过后，score、threshold、reason、error、success、verbose_logs 和 evaluation_cost 都留在实例上，MetricData 再把这些状态写死到结果里，最后交给 EvaluationResult 汇总 TestResults、confident link 和 run identity。

一旦同时跑多个测试，同一个 Metric 实例怎样重置状态就很关键。把结果留在对象字段里，确实方便实现具体指标，可执行器每次测量前都得清掉 error 等旧值，还得防止不同协程互相覆盖。保存结果时也要留下 metric 名称、实现版本、模型版本、threshold、strict mode 和这次测量的输出，不能只留最后那个布尔值。

实现取舍与失败语义

用户可以直接提交 TestCase，于是 DeepEval 能拿任意外部系统的输出做评测，agentic iterator 还能把被测代码怎样运行一并记进 trace。两条路径留下的证据强弱不同：前一条要相信调用者填入的 actual_output，后一条虽然能关联运行 trace，仍然必须锁定用户代码和执行环境。

Metric 判定失败，说明测量已经有效完成，只是分数没有达到阈值，metric.error 则表示这次根本没拿到可信的测量结果。ignore_errors=True 可以让批量任务继续往下跑，但统计时不能让报错项悄悄消失，分母怎么算必须提前约定。缓存确实能少调几次 Judge，可 key 必须覆盖测试内容、指标配置、模型和依赖版本，否则你无法解释为什么会命中。异步并发能提高吞吐，也会逼着你处理好有状态 Metric、事件循环和全局 TestRun 管理之间的隔离。

动手实验

给问答系统设计一个 Golden 和相应的 LLMTestCase，分别填好 input、actual_output、expected_output、context 与 retrieval_context，并说明每个字段是谁写进去的。接着设计两个 Metric，一个检查确定性格式，另一个调用模型 Judge，再列出 score、threshold、reason、error 和 cost 应该怎样落盘。最后让 Judge 模拟超时，比较 evaluate(..., ignore_errors=True) 与 assert_test 各自在调用方看来会发生什么。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

Golden 里的 input、expected 和 context 由数据集预先定义，actual_output 要等 Target 跑完后写入，retrieval_context 记的则是系统这次真正检索到了什么，并非事先准备好的答案。格式 Metric 可以离线算出确定分数，Judge Metric 还得记下评分模型、prompt/config 和成本。开启忽略错误后，程序应该保留 error，再继续跑其他测试，assert_test 则应把 score、threshold、strict、error 和 reason 写进失败信息，然后抛出异常。

课程门禁不需要 API key，因为它只检查源码有没有锁定、教学结构是否完整，以及本仓库的离线实验能不能通过，并没有声称它真的运行过某个 DeepEval Judge。

如何核对

先到 deepeval/evaluate/evaluate.py 比较 evaluate 与 assert_test 分别做了什么，再进 deepeval/evaluate/execute/loop.py 检查同步、异步和 agentic 三条路径怎样执行。最后对照 deepeval/metrics/base_metric.py，确认 success 到底按什么规则得出。

本篇不能证明什么

API 好用、指标设了阈值，或者 pytest 能在条件不满足时报错，都证明不了指标真的有效、Judge 没有偏差、数据集代表线上分布，或这次运行能够复现，更不能据此批准当前版本发布。它们只提供评测机制，不提供发布授权。

上一节 · 下一节

DeepEval 数据对象：Golden 何时变成 LLMTestCase

上一节 · 下一节

本篇要解决什么问题

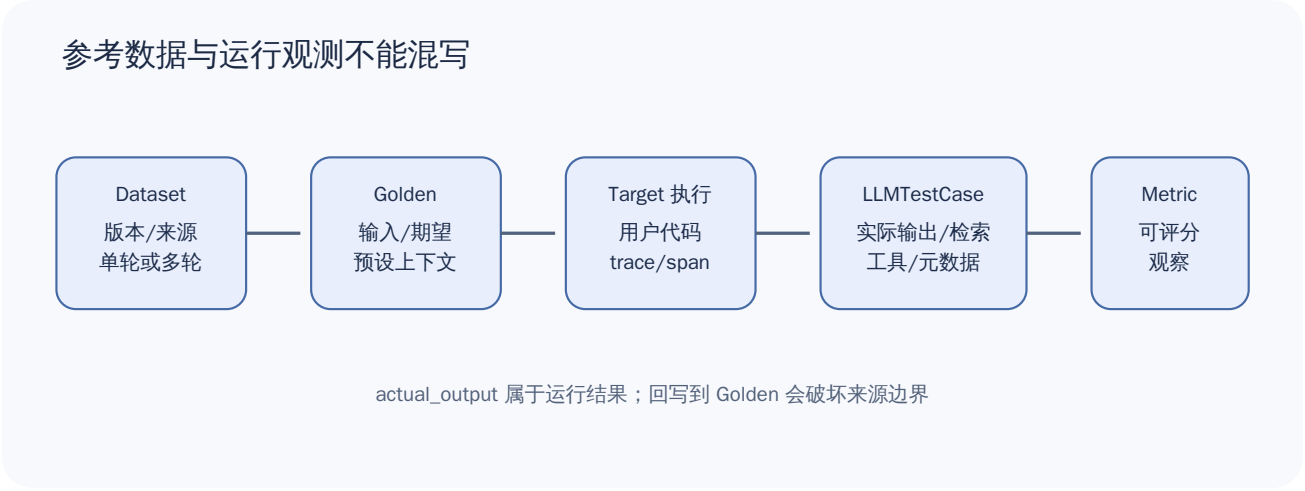
评测数据常把「给模型的输入」「参考答案」「模型实际输出」和「运行时检索证据」全塞进一个含糊的 JSON 对象，DeepEval 却把 Dataset、Golden（黄金样本）和 LLMTestCase（大语言模型测试用例）分开了。你可以顺着数据往下看：Dataset 怎样保证单轮和多轮不混在一起，又怎样标记数据集，Golden 为什么在 Target 运行前就能存在，LLMTestCase 又为何要等 actual_output 写进来，才能交给大多数 Metric 测量。

这不是字段字典。你要跟着对象一路往下追，看每个字段由谁写入、到什么时候不能再改、又靠什么回连数据集，同时分清哪些运行时观测绝不能倒灌进参考数据，最后再看 agentic iterator 怎样等用户代码跑完后补齐 TestCase。

先建立源码地图

源码位置	责任	核对问题
deepeval/dataset/dataset.py	EvaluationDataset、导入、身份与 iterator	单轮/多轮、rank、alias/id 怎样维护
deepeval/test_case/llm_test_case.py	LLMTestCase 与 ToolCall 等观测	可评分用例保存哪些运行事实
deepeval/evaluate/execute/loop.py	Golden 到 TestCase 的运行转换	用户代码和 trace 何时进入证据

完整调用链



1. EvaluationDataset 创建后可接收 goldens 或 test_cases；构造器检查元素类型，并根据首个对象确定单轮或多轮数据集。
2. 加入对象时写入私有 _dataset_alias、_dataset_id 和 _dataset_rank。单轮数据集拒绝 ConversationalGolden/TestCase，多轮数据集拒绝单轮对象，防止执行器面对混合语义。
3. CSV/JSON 导入把列映射为 LLMTestCase 或 Golden 字段；导入只完成结构转换，不能证明来源、授权、去重或参考答案质量。
4. 直接 evaluate(test_cases, metrics) 时，actual_output 已由调用者提供，执行器只负责测量，不执行 Target。

5 agentic Dataset iterator 逐个 yield Golden，让用户代码处理输入并产生 trace；控制权返回后，执行循环取得当前 trace，从 Golden 和 span attributes 构造 LLMTestCase。

6 新 TestCase 继承 dataset alias/id/rank，以便 TestResult 回连原样本；trace/span 上的 input、output、context、retrieval context、tools 等成为运行观测。

7 若 span 声明指标却无法构造 LLMTestCase，执行器记录可解释错误，而不是用空字符串伪造正常评分输入。

关键数据结构

Dataset 是带身份的容器，却不是统计单位。Golden 记下的是待执行规格，所以可以带 input、expected_output、context、additional_metadata 等字段，LLMTestCase 则记下实际发生的一次观测，其中最要紧的是 input 和 actual_output。参考型指标从 expected_output 取答案，context 给出预期可用的背景，retrieval_context 保存系统这次真正检索到的内容，tools_called 和 expected_tools 还让你能判断工具用得对不对。私有 dataset rank 可以帮你还原原先的排列顺序，但整理数据时位置会变，要想稳稳认出同一个样本，最好另外保存 sample id 或内容摘要。

按统一术语来对应，一个 Golden 可以看作 SampleSpec，Target 每运行一次，就会产生 Trial/Attempt 和相应的 LLMTestCase 观测。DeepEval 自己没有强制把 Trial 与 Attempt 分开，因此调用者只要遇到网络重试或重新生成，就得在框架外逐次留档，否则最后写进去的 actual_output 会遮住中间过程，看起来像是从来只有这一个结果。

实现取舍与失败语义

直接提交 TestCase，可以把 Target 执行和 Eval 测量拆开，很适合离线导入日志，可 Harness 也就无法独自查明 output 到底怎么来的。agentic iterator 会把执行过程一并记进会话，前后证据接得更紧，不过它依赖全局 trace/session 管理，也要求用户把埋点写对。Dataset 会拦住单轮和多轮混用，让结构错误早点暴露，但它不会替你查语义重复、数据泄漏或时间切分。

空数据集、对象类型不对以及单轮多轮混用，都说明输入没有遵守合同，程序应该在运行前就报错。用户代码抛异常，则是 Target 或场景执行出了问题。若 trace 缺字段，执行器连 TestCase 都建不出来，问题就在观测或适配这一层。只有 Metric 拿到合法输入并完成测量后给出低分，才算产品没有通过判据。这四类失败不能混成一个失败率。

动手实验

选一个「根据知识库回答退款问题」的样本，先写出 Golden 和运行后 LLMTestCase 最小需要哪些 JSON 字段，再把预先给定的政策放进 context，把系统这次真正检索到的片段放进 retrieval_context，并解释这两份内容为什么不能混。随后模拟两次网络 Attempt，第一次超时、第二次成功，再指出除了 DeepEval TestCase 之外，每次尝试还要留下哪些元数据。最后补齐 Dataset 版本清单，写明来源、许可证、内容摘要和切分信息。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

Golden 至少要有 input 和 expected_output/expected context，系统跑完之后，TestCase 还要添上 actual_output、retrieval_context、tools_called 和运行 metadata。context 是评测规格给出的背景，或是规格要求系统使用的背景，retrieval_context 则是这次运行真正观察到的检索结果。两者一旦混在一起，检索指标就没了可以比较的两边。两次 Attempt 都要保存 attempt_id、起止时间、错误、重试原因和输入摘要，还要写清为什么第二次结果被选为 canonical。

Dataset 的每个版本还要写明数据从哪里生成或采集、拿到了什么授权、怎样去重和检查污染，并留下时间戳、分区策略与摘要。能序列化，只是起点。

如何核对

先看 deepeval/dataset/dataset.py，弄清 EvaluationDataset 怎样构造对象，goldens/test_cases setter 与 add 方法怎样收进数据，再找到 agentic evaluate iterator。然后去 deepeval/evaluate/execute/loop.py，核对执行器究竟怎样把 Golden 变成 LLMTestCase。

本篇不能证明什么

Dataset alias、云端 ID、rank 或一次成功导入，都证明不了样本是否唯一、是否泄漏、能否代表真实用户，也证明不了它可以合法公开。即便 LLMTestCase 的字段一个不少，你仍然无法确认 actual_output 真由声明的模型生成，还得另存运行清单和证据摘要。

上一节 · 下一节

DeepEval Metric：有状态评分器怎样产出可解释结果

上一节 · 下一节

本篇要解决什么问题

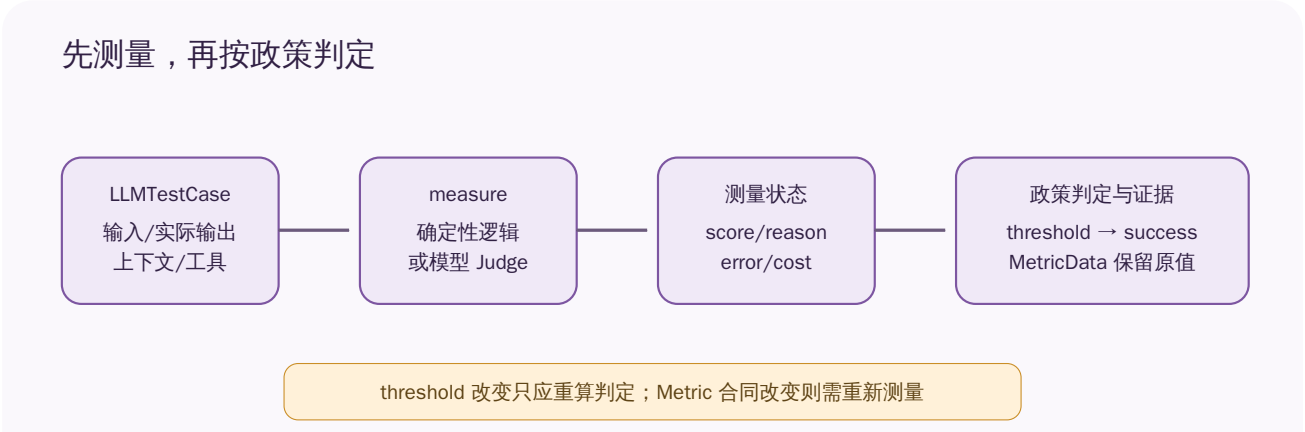
你很容易把 Metric 当成一个简单的 test_case -> float 函数，可 DeepEval 的 BaseMetric 会把 threshold、score、reason、error、success、verbose_logs、evaluation_model 和 cost 全留在自己身上。具体 measure 先写回这些状态，执行循环再把它们读出来，组装成 MetricData（指标数据）。这样写自定义指标确实方便，但同一个对象能不能并发复用、旧状态怎样清掉、阈值该怎么解释、错误又该归到哪一类，都成了必须说清的合同。

读完后，你应该能写出一个最小的自定义 Metric，并分清哪些字段记原始测量，哪些字段表达政策判断，哪些字段只用来排查运行问题。你还要能解释，为什么 Judge（裁判模型）和确定性 Scorer（评分器）即使都给出 score，也不能直接横向比较。

先建立源码地图

源码位置	责任	阅读焦点
deepeval/metrics/base_metric.py	Metric 抽象和 is_successful	测量状态与阈值语义
deepeval/evaluate/execute/loop.py	执行、跳过、错误与结果更新	对象状态怎样冻结为证据
deepeval/evaluate/evaluate.py	指标输入验证和入口行为	阈值与 assert_test 的关系

完整调用链



1. 调用者构造 Metric 实例并设置 threshold、strict_mode、evaluation model 等配置；入口验证至少一项指标具有阈值，确保 assert 语义可判定。
2. 执行器为 TestCase 与 Metric 建立 pair。在每次执行前清理 metric.error 等上次状态，并根据同步/异步模式调用 measure 或 a_measure。
3. 具体 Metric 从 LLMTestCase 读取所需字段。确定性指标直接计算；LLM-as-a-Judge 指标可能生成 prompt、调用评估模型、解析结构化结果并累计 evaluation_cost。

4 Metric 写回 score、reason、verbose_logs 等；若异常被保留，则写 error。is_successful 在 error 时返回 false，在无 threshold 时依据 score 是否存在，否则比较 score 与 threshold；strict_mode 可要求满分。

5 执行公共逻辑把 Metric 状态转换为 MetricData，包括 name、threshold、success、score、reason、strict、evaluationModel、error 与 cost，并挂到 TestResult。

6 assert_test 遍历 metrics_data，把 error 非空或 success 为 false 的指标列为失败，生成可读字符串并抛出 AssertionError。

7 批量 evaluate 不只给布尔断言，还保留每个 TestCase 的指标结果供控制台、文件和 TestRun 汇总。

关键数据结构

BaseMetric 把 measure/a_measure 定义成抽象方法，返回的 float 只是接口露出来的一小部分，真正的结果仍留在实例字段里。score 记测量值，threshold 记当前采用的政策，程序把两者合起来才得出 success。reason 解释这次测量，error 说明没能测出来，evaluation_cost 则记下 Judge 花了多少成本。若还要往下排查，可以查看 score_breakdown 和 verbose_logs，了解各个组成部分出了什么情况。分数只是其中一项。

因此，保存证据时要把原始 score 和测量配置一并留下，不能只存 success，因为阈值变了以后，你可以不重跑 Target，直接按旧 score 重新判断是否通过。阈值不是测量值。但只要 Metric 换了 prompt、model 或解析逻辑，测量所依据的合同也就变了，此时旧 score 还能不能用，必须看这个指标怎样定义自己的版本。

实现取舍与失败语义

有状态基类让自定义 Metric 更好写，控制台也能直接读 reason，可同一个实例一旦同时服务多个 TestCase，字段就可能互相覆盖。执行框架要么为每个任务复制实例，要么串行访问，也可以采用别的隔离办法。无论选哪一种，Metric 开始测新用例前都得清掉上次留下的 error 和 score，否则前一个用例会污染后一个。

分数低于阈值，说明测量已经完成，只是结果没通过。Judge 的输出解析失败，要记成 Metric error。TestCase 少了必需字段，是输入或适配出了问题。Judge API 遭到限流，则是评分基础设施故障。这些情况不能混着算。strict_mode 只会按特定规则收紧通过条件，它属于政策配置，不会让分数变得「更准确」。模型 Judge 写出的 reason 也只是模型生成的解释，不能当作事实证明，因此你仍要做校准、人工复核和对抗样本测试。

动手实验

先在纸面上写一个 ExactJsonMetric，让它检查 actual_output 能否解析成 JSON，并确认其中有 answer 字段，条件满足就给 1，否则给 0，同时写出 reason。接着列出 measure 前后 BaseMetric 的每个字段怎样变化，再设计 FaithfulnessJudgeMetric，说明还要锁定哪些模型、模板、温度、解析 schema 和重试策略。最后把 0.7 与 0.9 两个 threshold 分别套在同一批 score 上，并解释为什么这里只是重新判断有没有通过，并没有重新测量。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

ExactJsonMetric 测量前要清空或重置 score、reason 和 error，测量后把 score 写成 0 或 1，并让 reason 说清究竟满足或违反了什么条件。只有指标自己跑不起来时，才应该写 error。Judge Metric 不能只靠一个类名标记身份，至少还要带上评估模型标识、系统与用户模板摘要、参数、输出 schema、库 commit 和依赖版本。

把不同 threshold 套在同一批结果上，只会改变 success 和汇总数字，不会改写原始 score。strict_mode 只要变了，也必须跟着记进结果。若 Judge 已经换了判断逻辑，原 score 所依据的测量合同就不再相同，此时不能只重算

success，再把它说成同一版指标的结果。

如何核对

先读 deepeval/metrics/base_metric.py 里的字段、抽象方法和 is_successful，弄清基类怎样保存并判断状态。再到 deepeval/evaluate/execute/loop.py 搜索 _execute_metric、MetricData 和 test run update，核对程序遇到异常后还会不会往下走，以及它究竟在什么时候把结果定下来。

本篇不能证明什么

Metric 即使带着 reason、threshold 和 cost，也证明不了它测的东西真的有效，Judge 与专家意见一致，或者阈值经过了业务验证。不同指标各自给出的 0.8 也不能直接比较。你还得用独立的验证数据，分别检查指标是否可靠、有没有效度、偏差多大，以及遇到扰动是否稳定。

上一节 · 下一节

DeepEval
执行策略：异步、缓存与错误不能改变统计单位

上一节 · 下一节

本篇要解决什么问题

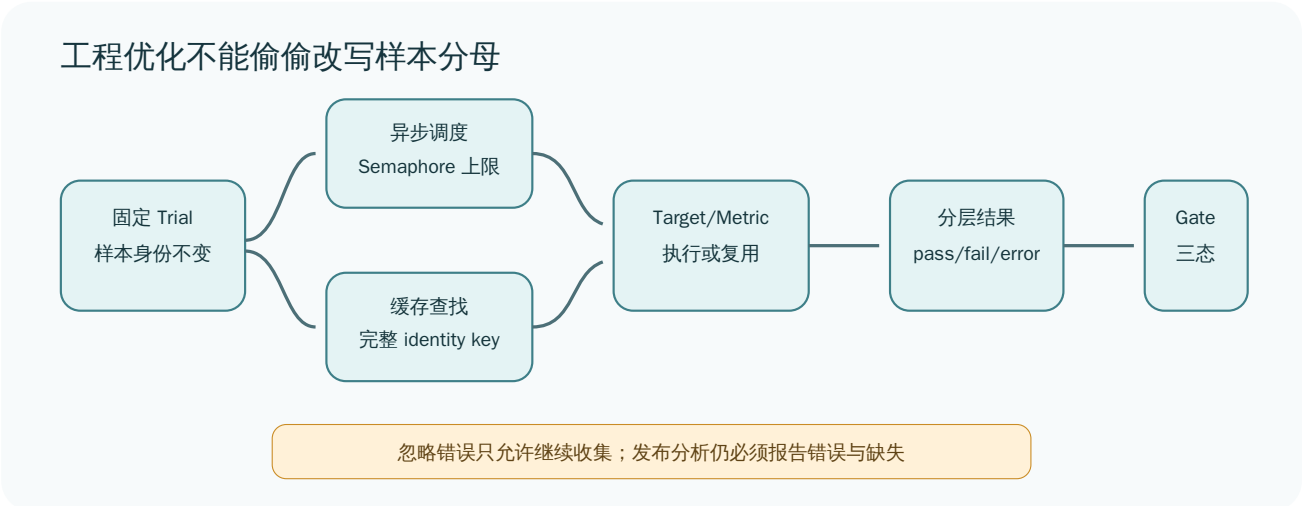
Judge（裁判模型）调用又慢又贵时，你自然会想到开异步并发、加缓存，或者让程序遇错继续跑，可这几项设置也最容易把评测数字带偏。并发可能让有状态 Metric 的实例状态互相污染，缓存可能拿出已经过期的评分，ignore_errors 可能悄悄改掉分母，失败后整轮重跑还可能把同一个测试算成多个独立观测。这里从 DeepEval 的 AsyncConfig、CacheConfig、ErrorConfig 和 agentic loop 入手，看每种恢复办法应该落在哪条统计边界内。

重点不在于「打开哪个参数会更快」，你真正要查的是每个参数究竟改动了哪一层：任务怎么调度，评分怎么调用，证据怎么落盘，外部测试又怎么运行。无论怎样优化，都不能弄乱 Sample/Trial 身份，也不能丢掉结果或藏起错误。并发不会创造独立样本。分母也不能暗改。

先建立源码地图

源码位置	责任	核对问题
deepeval/evaluate/evaluate.py	配置默认值、同步/异步选择、收尾	参数怎样进入执行器
deepeval/evaluate/execute/loop.py	semaphore、任务、cache/error 和 TestRun	并发与异常怎样传播
deepeval/metrics/base_metric.py	有状态测量对象	并发隔离需要保护什么

完整调用链



1 evaluate 接收 AsyncConfig、CacheConfig 和 ErrorConfig；异步模式取得事件循环并调用异步执行器，同步模式直接顺序执行；

2 异步 agentic loop 创建 asyncio.Semaphore(max_concurrent)，把每个用户 callback 绑定到当前事件循环并受信号量控制，避免无限并发和跨 loop Future 混用；

3 Target/user code 结束后捕获 trace，构造 TestCase；执行器为 metrics 计算缓存键，命中时复用 MetricData，未命中才运行 a_measure/measure；

4 Metric 或 trace 发生错误时，ErrorConfig 决定是否抛出或把 error 写入结果继续，继续不代表成功，后续汇总必须保留不可评分状态；

5 CacheConfig 区分 use_cache 与 write_cache，TestRunManager 的磁盘保存行为随配置改变，但缓存与最终证据存储不是同一概念；

6 任务完成后，每个 TestCase 更新 TestRun；异步任务集合属于当前 iterator run，结束时一次性归集，避免状态泄漏到下一事件循环；

7 入口记录 run_duration、超参数和输出；在 CLI 测试模式中由 CLI 统一 finalize，普通 evaluate 才自行 wrap-up，避免重复收尾。

关键数据结构

AsyncConfig 至少要写清 run_async、max_concurrent、throttle 等调度规则，CacheConfig 说明缓存怎么读写，ErrorConfig 则规定 ignore_errors 和跳过策略，这些配置都要记进 RunManifest，因为它们共同标识了这次运行。Semaphore 只能限制「同一时刻有多少协程在跑」，不会替你标记 Trial，也不保证执行顺序或事务。缓存里也不能只存一个分数，必须连同 test case 摘要、metric identity/config、Judge model、代码版本和输出 schema 一起保存，之后命中缓存时才查得清来路。

记录错误时，要用 target_error、trace_error、metric_error、timeout、cancelled、cache_corrupt 等机器可读类型，不能只扔下一段文字。Reference Harness 会把基础设施重试都挂在同一个 Trial 下面，再从这些 Attempt 里选出 canonical Attempt，但 DeepEval 的入口不会自动替外部 Target 的重试建立这层关系，你得自己补上。

实现取舍与失败语义

高并发可以缩短实际等待时间，却也可能触发 Provider 429，让多个任务覆盖同一个 Metric 的状态，打乱输出顺序，甚至让费用突然上涨。信号量只管并发上限，究竟设多大才安全，还得看 Judge 的速率限制、可用内存和对象有没有隔离。缓存可以少花钱，也能减少随机波动，可它可能遮住模型或模板已经升级的事实，因此必须允许关闭，并在实验身份改变时失效。这里要留痕。

ignore_errors 适合让长批次在出错后继续跑，好把诊断信息尽量收齐，但做发布分析时，必须先报告评测错误率以及哪些样本缺了结果。错误项要单列，缺失也要分类。若直接删掉报错样本，complete-case 偏差就进来了，若把 error 一律记成 0 分，又会把产品质量和 Harness 健康混在一起。错误不会因此消失。更稳妥的做法是分开列出 pass、fail 与 error，只要关键错误还没解决，Gate 就可以给出 inconclusive。

动手实验

假设一共有 100 个测试，其中 90 个顺利完成、72 个通过，另外 5 个遇到 Judge 超时，还有 5 个出现 Target 错误。请分别计算「仅完成项通过率」与「按全体样本通过率」，再说明两种算法各自看漏了什么。然后设计缓存键，列出其中必须覆盖的 TestCase、Metric 和依赖字段。最后模拟 max_concurrent=20 引发限流，比较降低并发、在 Judge 内部 retry 与重跑整个 Trial 各会留下什么证据。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

只看完成项时，所有错误都会被排除，分母是 90，通过率为 80%；按全体样本计算，分母是 100，目前能够证实的通过率为 72%。前一种算法藏住了 10% 的缺失，后一种算法又说不出报错样本原本会通过还是失败，所以报告里要同时列出 72 pass、18 fail、5 judge error 和 5 target error，再让 Gate 按预先登记的规则处理。

缓存键至少要包括规范化 TestCase 的输入、输出和上下文摘要，还要把 Metric 类及配置、阈值之外的测量参数、Judge 模型与 prompt、代码 commit、依赖版本都算进去。限流后的 retry 只是同一个评分 Attempt 里的传输恢复，重跑整个 Trial 却会重新生成 Target 输出和评分证据，因此旧 Attempt 必须留下，也不能因为重跑过就增加独立样本数。

如何核对

先看 deepeval/evaluate/evaluate.py，确认三种 Config 怎样传进执行器，并找到 CLI finalize 所在的分支。再去 deepeval/evaluate/execute/loop.py 搜索 Semaphore、tasks、cache_config、error_config、MetricData 和 TestRun update，沿着状态怎么变化来核对执行边界。

本篇不能证明什么

用了信号量、缓存和错误记录，也证明不了代码一定线程安全、缓存键没有漏项、远程服务始终稳定，或统计结果没有偏差。至于 CI 能不能在 ignore_errors 开启时放行，这是组织要定的政策，不能让库的默认值替你做决定。

上一节 · 下一节

Harbor 与 Terminal-Bench 1：Agent 终端任务怎样成为可核对的 Trial

上一节 · 下一节

本篇要解决什么问题

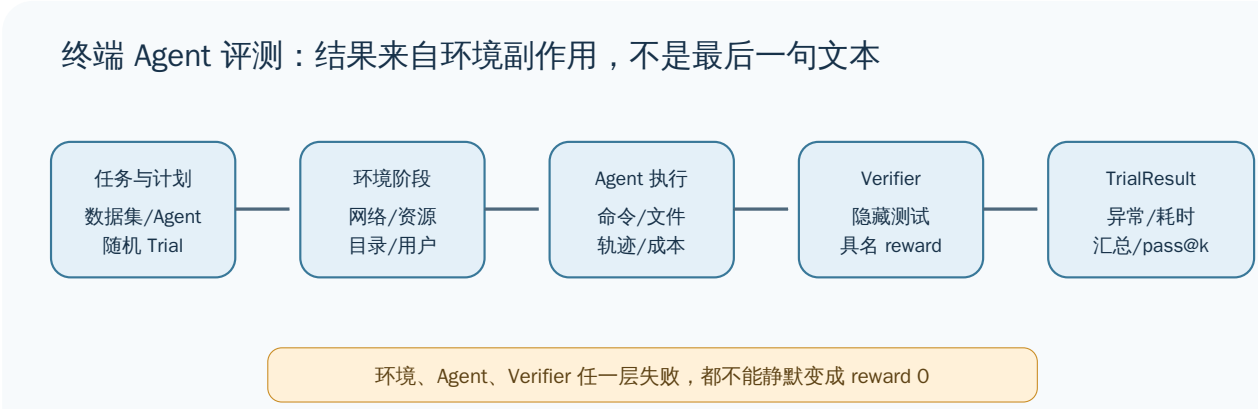
语言模型基准常把「输入发给模型、再比较答案」当作主循环，终端 Agent 的评测却要先启动隔离环境，安装或连接 Agent，让它执行多轮命令并留下日志与文件，之后才能在同一个或另一个环境中运行 Verifier（验证器）。Harbor 从 Terminal-Bench 演进而来，后来成为通用的 Agent Harness/Eval Harness，Terminal-Bench 1 则保留了更紧凑的早期结构。把两边源码对着看，你就能弄清 Job/Dataset 怎样铺开 Trial（试验），环境和 Agent 怎样交替工作，Verifier 怎样从容器产物算出 reward，也能明白恢复旧结果时为什么不能只看一个 resolved 布尔值。

本课程锁定 Harbor 74f0176384cff88b99306770473b4875760c5a21 和 Terminal-Bench 1 d28711d0da2675d0bb1d56de45ae5df6082438a3，前几篇先借 Terminal-Bench 1 讲清紧凑的基线流程，再看 Harbor 怎样加入网络策略、资源能力、事件 hook、regrade 和多步结果。这份课程不罗列两个项目的全部版本差异，也不暗示 Harbor 的每项能力都来自 Terminal-Bench 1。

先建立源码地图

系统	锁定文件	责任
Harbor Job	src/harbor/job.py	解析任务/Agent，生成与调度 Trial，恢复 Job
Harbor Trial	src/harbor/trial/trial.py	环境、Agent、Verifier 生命周期与事件
Harbor Environment	src/harbor/environments/base.py	容器抽象、网络/资源/执行/文件操作
Harbor Verifier/Result	src/harbor/verifier/verifier.py · result.py	奖励解析、异常、耗时与多步结果
Terminal-Bench Harness	terminal_bench/harness/harness.py	Dataset、Agent、并发 Trial 与 resume
Terminal-Bench 数据/结果	dataset.py · models.py	任务选择、TrialResults、accuracy/pass@k

完整调用链



1. Dataset/TaskClient 解析任务版本与选择条件；Job/Harness 再与 Agent、模型、attempt 数组合，生成 TrialConfig 或 trial handler。
2. 运行锁与输出目录固定 dataset、agent、模型、超时、并发和网络等配置；resume 只接受匹配配置，并区分完整结果与残留中间产物。
3. Trial 创建 Agent environment，准备任务文件和目录，执行 Agent setup/run，并把 stdout/stderr、trajectory、文件与 token/cost 记录到 Trial 路径。
4. Agent 结束后切换网络/环境阶段，启动 verifier。Harbor 可让 verifier 与 Agent 同环境或分离环境，避免 Agent 篡改测试或利用验证期网络。
5. Verifier 注入测试、运行测试命令并下载 verifier 目录，从 reward JSON 或文本解析数值；文件缺失、空文件、非法 JSON 分别是验证基础设施错误。
6. TrialResult 保存 AgentInfo、AgentContext、VerifierResult、异常、setup/execution timing 和多步 StepResult；Job 汇总各 Trial reward 与统计。
7. Terminal-Bench BenchmarkResults 根据每 task 的多次 Trial 计算 accuracy 和 pass@k。Attempt 数是每任务重复观察，不是基础设施 retry；结果解释必须保留 task 聚类。

关键数据结构

Task 写明 instruction、环境怎么构建以及 verifier 怎么运行，Dataset 收住锁定后的任务集合，TrialConfig 再把一个 Task、一个 Agent/模型、一次 attempt 和一套环境政策固定成一次运行。终端 Agent 跑过以后，AgentContext/AgentResult 留下轨迹、token 和成本，VerifierResult 留下具名 rewards，TrialResult 还会记异常以及各阶段花了多久，所以汇总不能代替逐行证据。JobResult/BenchmarkResults 都是从逐行结果算出来的，也代替不了 Trial 目录里的配置、日志和 verifier 产物。

映射到统一模型以后，Terminal-Bench 的 n_attempts 更接近同一个 Sample 上重复进行的多个随机 Trial，并非网络重试所产生的 Attempt。Reference Harness 会明确分开这两层，因为随机重复要增加 Trial 的统计分母，基础设施恢复却仍挂在原来的 Trial 下。跨工具比较时，你得先把双方术语对应到同一套模型，不能看见字段同名就认定它们是同一种对象。

实现取舍与失败语义

容器会隔开任务依赖和运行副作用，让 Verifier 有机会检查文件与命令产生的结果，但镜像、运行时、网络和资源限制也因此成了实验条件。把 Verifier 和 Agent 放进不同环境，可以降低测试被篡改的风险，可你必须复制工作区或明确哪些产物共享，两个环境之间的差异也可能凭空制造失败。resume 能省下重复运行的成本，但只有锁定配置完全一致、旧结果也完整时才能复用，残留目录不能冒充完成状态。

Agent 正常跑完却没解决任务，属于产品失败。Agent 进程崩溃，则可能是产品自身的问题，也可能是适配写错了。环境构建失败、健康检查失败和 Verifier reward 缺失要记成 Harness 错误，遇到超时还得注明发生在 setup、agent 还是 verifier 阶段，这些阶段不能混写。只有分层留下这些情况，Gate 才分得清「不通过」与「无法判断」。

动手实验

为「在容器中修复一个损坏的配置文件」设计 Task，写明 instruction、初始文件、网络策略、资源限制、Agent 可以写哪些路径、隐藏 verifier 怎样运行，以及 reward 怎样产生。给同一个 Task 安排 3 个随机 Trial，再让第二个 Trial 因容器启动失败而做一次基础设施重试，并算出正确的 Trial/Attempt 数量。最后解释为什么只保存 reward=1，既不能复现，也无法审计这次运行。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

三个随机重复会产生 3 个 Trial，第二个 Trial 恢复容器时只是在它下面再添一个 Attempt，所以统计分母仍然是 3，并没有增加。只看 reward=1，你不知道任务与测试用了哪个版本，也不知道 Agent、模型、环境镜像、网络、资源、轨迹、Verifier 命令和产物分别是什么，自然无法判断这个奖励是不是由合法 verifier 算出来的。

课程测试只检查锁定链接和教材合同，不会真的拉起 Docker 或运行外部 Agent，因此还得为具体平台单独编写集成测试，验证容器安全与资源隔离。

如何核对

先看 src/harbor/job.py，沿着 Job.create 往下追，弄清程序怎样初始化 TrialConfig，又怎样触发 queue hooks。再到 src/harbor/trial/trial.py 找出环境、Agent 和 verifier phase 分别何时运行，最后拿 terminal_bench/harness/harness.py 里的 resume 与并发逻辑来对照。

本篇不能证明什么

用了容器、隐藏测试、reward 文件和 pass@k，也证明不了任务没有漏洞、Agent 从未利用侧信道，或测试已经覆盖完整，更不能保证这些结果可以外推到真实的软件工程工作。课程也不会因为框架自己的测试通过，就宣称某个 Agent 已经具备相应能力。

上一节 · 下一节

Job、Dataset 与 Trial：运行计划怎样固定统计单位

上一节 · 下一节

本篇要解决什么问题

把 100 个任务各跑 3 次，看起来只是把 300 个工作项扔进并发队列，真正实现时却还要筛任务、组合不同 Agent 与模型、恢复旧结果、避开目录冲突、标记每次随机重复，并在 Job（作业）层汇总统计。Harbor 的 Job 和 Terminal-Bench 1 的 Harness 都在 Trial（试验）上面处理这些工作，所以这一篇要看计划怎样铺开，配置怎样锁住，以及恢复旧结果为什么也会影响统计是否正确。

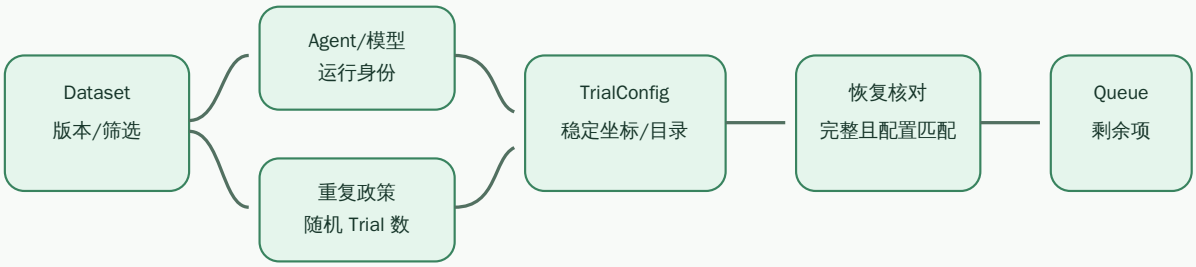
这三类对象要先分开：Dataset 决定哪些任务可以入选，Job/Harness 规定这次实验怎么跑，Trial 才表示某个 Agent 在某个任务环境里独立运行了一次。任何一层发生变化，历史结果都可能无法再直接并入新结果。

先建立源码地图

源码位置	责任	阅读问题
src/harbor/job.py	Job.create、任务解析、TrialQueue、既有结果	如何生成与恢复 Trial
terminal_bench/harness/harness.py	RunLock、task/attempt 展开和 resume	配置匹配与残留清理
terminal_bench/dataset/dataset.py	数据集缓存、include/exclude、限制与排序	本次实际任务集合如何确定

完整调用链

计划全集先固定，恢复才有可信分母



目录存在不等于完成；汇总必须能由 TrialResult 重新计算

1. Dataset 根据 name/version/path 或 YAML 构造，缓存远程内容，再应用 include、exclude、task_ids 和 n_tasks；排序可以按预计时长改善并发利用率。
2. Job.create 解析 Task 配置、Agent skills 与环境资源政策；Terminal-Bench Harness 初始化 Agent class 和 Dataset。
3. 计划层对 task × agent/model × repetition 展开。每个 Trial 获得稳定名称/UUID、独立输出目录和完整 TrialConfig。

4 Job 若发现已有目录，读取 JobConfig、TrialConfig 与 TrialResult，验证身份后把完成项加入 existing results/rewards/stats，并只调度 remaining configs。

5 Terminal-Bench 用 RunLock 比较当前 dataset、agent 与 run config；不一致时拒绝 resume。结果文件完整的 Trial 被跳过，只有部分 artifacts 的任务目录会先清理后重跑。

6 TrialQueue 按并发策略运行剩余 Trial，并在 START/CANCEL/END 等事件更新 Job 级状态；完成结果逐项写盘，不等待整个 Job 才保存。

7 JobResult/BenchmarkResults 由 TrialResults 汇总；统计代码按 task 聚合多次重复，计算 resolved、accuracy 与 pass@k。

关键数据结构

DatasetConfig 锁定数据集名称、版本、路径和筛选条件，JobConfig/RunLock 再锁定 Agent、模型、并发、n_attempts、超时、网络与环境政策，TrialConfig 由此标出一组不会混淆的运行坐标，而且这组坐标必须稳定。只有 TrialResult 才能代表一份完整的执行证据，单凭「目录存在」或「config 存在」都不够。JobStats 和 BenchmarkResults 只是从 TrialResult 算出来的缓存，即使删掉后重建，也应该能靠逐行结果重新算出相同数字。

Terminal-Bench 在 TrialResults 里保存 task_name、trial_name、reward 等字段，BenchmarkResults 会先把结果按 task 分组，数出每个任务成功了几次，再据此估计 pass@k。这样才能保住 task 这个聚类单位，不会把 300 条 Trial 误当成 300 个不同任务，夸大实际覆盖范围。

实现取舍与失败语义

预先把计划全部展开，方便程序显示进度，也方便中断后恢复，但矩阵很大时会占用不少内存，动态生成的任务还必须先冻结版本。按预计时长排序能提高吞吐，却不该改动 Trial 的内容。可一旦全局超时使排在后面的任务更容易缺结果，排序就已经改变了数据怎样缺失，分析时必须把这项条件记下来。

严格检查 RunLock，可以拦住彼此不兼容的结果，代价是配置哪怕只改一点，也得换一个新的 run_id，这是守住实验身份必须付出的成本。清理不完整产物前，你还要确认目录确实只属于目标 Trial。已经完成的结果若缺少证据摘要，或锁定配置对不上，就应该直接拒绝复用。任务被取消、进程崩溃或用户中断时也要保存状态，不能因为没有 reward 就让它从计划中消失，缺失状态同样要入账。

动手实验

假设 Dataset 共有 12 个任务，include 从中选出 8 个，exclude 又删去 2 个，同时配置 2 个 Agent，并给每一项安排 3 个随机 Trial。请算出计划一共有多大，再设计一组稳定坐标。随后假设已有 30 条完成结果，另有 8 条带着完整 config 却没有 result，其余还没启动，写出 resume 应该复用多少条、清理多少条、重新调度多少条，并说明模型版本改变后为什么不能继续使用原 run_id。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

筛选后实际剩下 6 个任务，所以计划共有 6 × 2 × 3 = 36 个 Trial。稳定坐标至少要包括 dataset commit/task id、Agent 与模型 identity，以及 repetition index。30 个完整结果可以复用，但「8 个无 result」与计划总数冲突，因为整个计划只有 36 项。这正说明恢复前必须先核对清单，不能看到几个数字就直接相加。若把条件改成 4 条不完整结果，就要清掉并重新调度这 4 条，再加上另外 2 条尚未启动的任务，一共调度 6 条。模型版本一变，Target identity 也跟着变，因此要另建 Job。

这个矛盾是故意留下的，它提醒你先核对计划全集以及各组结果之间的关系，别拿到日志数字就直接相加，而要先校验恢复清单。

如何核对

先到 src/harbor/job.py 看 _maybe_init_existing_job、_init_trial_configs 和 remaining config，弄清 Job 怎样识别旧结果并算出还要运行哪些 Trial。再用 terminal_bench/harness/harness.py 里的 _validate_resume_configuration 与 _filter_completed_and_cleanup_incomplete_tasks 对照，核对两套实现各自在什么条件下允许恢复。

本篇不能证明什么

计划没有漏项、RunLock 能对上、resume 也成功了，仍然证明不了各个 Trial 真正彼此独立、随机种子确实生效，或任务难度能够代表真实场景，也排除不了有人改过历史结果。你还要用额外机制生成可靠的证据摘要，并把结果放进不可变存储。

上一节 · 下一节

Environment 与 Agent 生命周期：副作用怎样被隔离和观察

上一节 · 下一节

本篇要解决什么问题

终端 Agent 不只返回一段文本，它还会安装软件、编辑文件、启动服务、访问网络并消耗资源。Eval Harness（评测框架）若只截取最后一句输出，就判断不了任务是否真的完成，Agent 有没有越权，这次运行又是否污染了后续 Trial（试验），因为副作用也要留下证据。Harbor 用 BaseEnvironment 和 Trial 串起环境能力、工作目录、各阶段网络、Agent setup/run、日志回调与 verifier phase，这一篇就顺着这条生命周期往下看，弄清每道边界怎样让评测可以核对。

并发隔离尤其要仔细查，因为环境对象若用普通全局变量保存 env overlay 和 output callback，多个 Trial 的状态就会串到一起。网络策略若只在容器启动时设置，也表达不了 Agent 与 verifier 各自需要的权限。Agent 刚结束就销毁环境，Verifier 又没有机会检查最后留下的文件状态。

先建立源码地图

源码位置	责任	核对问题
src/harbor/environments/base.py	环境能力、资源、网络、exec、文件与 scoped context	副作用和并发如何隔离
src/harbor/trial/trial.py	创建环境、阶段切换、Agent/Verifier 与 hooks	生命周期怎样排序和收尾
terminal_bench/agents/base_agent.py	Agent identity、prompt template 与 perform_task	Harness 与 Agent 的最小契约

完整调用链

1 Trial 根据 Task 环境配置和 Agent 配置创建 BaseEnvironment；环境实现声明 capabilities/resource capabilities，并验证 CPU、内存、存储、GPU 与 compose 支持。

2 环境启动时准备工作目录、挂载、默认用户和启动变量，执行 healthcheck，而启动超时与构建失败在 Agent 运行前结束 Trial。

3 Trial 进入 Agent phase，应用该阶段网络策略和 Agent scoped env，ContextVar 风格的 overlay 与 output callback 只作用于当前 asyncio task，嵌套 scope 按近者优先。

4 Agent setup 安装工具或准备依赖；run/perform_task 接收 instruction 与环境/日志路径，通过环境 exec 与文件 API 执行任务，并返回 AgentContext/AgentResult。

5 Trial 下载 Agent 日志与轨迹，记录 token/cost 和 setup/execution timing；无论成功失败都要尝试回收输出，避免异常抹掉已发生副作用。

6 进入 Verification phase 前切换网络政策。若使用 separate verifier environment，则按策略复制或挂载待验工作区；若 same environment，则明确 Agent 残留进程和权限影响。

7 Verifier 完成后停止环境、清理资源并发出 hook。清理错误应附加到 Trial 诊断，不能覆盖最初的 Agent/Verifier 异常。

关键数据结构

EnvironmentConfig 写明用哪个 provider、怎样准备镜像和构建、网络与资源怎么限制，以及要注入哪些环境变量。EnvironmentCapabilities 则由每种实现明确报出自己支持什么，免得调用方把所有容器后端都当成等价实现，因此能力声明必须核对。ExecResult 留下 stdout、stderr 和退出码，Agent identity 至少要包括 name、version、model/provider 与 prompt template，AgentContext 用来保存运行轨迹和用量。agent、verifier、logs、reward 等跨系统路径，则统一交给 TrialPaths/EnvironmentPaths 规定。

网络计划要明确分开 baseline、agent phase 和 verifier phase，因为 allowlist、public、disabled 不只控制安全边界，也会直接改变任务能不能完成，所以这些设置都属于实验条件。密钥只能通过受控环境变量注入，绝不能写进日志、TrialConfig 公共副本或课程材料。

实现取舍与失败语义

BaseEnvironment 抽出统一接口后，Docker、Kubernetes 和云沙箱可以共用一套 Trial 逻辑，但调用方仍要逐项核对每种实现报出的 capabilities。若后端悄悄忽略自己不支持的资源限制，任务之间就不再公平。Context-local overlay 可以隔开并发任务，可底层 provider 依然不能把会变化的状态放进进程全局。

Agent setup 失败、Agent timeout、命令返回非零、环境失联以及网络政策切换失败，都发生在不同阶段，不能压成同一种错误。错误该归给谁，要看当时跑到哪一步。有些任务允许命令返回非零，因为它可能只是 Agent 探索过程中的一次尝试，不能看到一次 exec 非零就判定产品失败，最后仍要按 Agent contract 和 Verifier 的结果下结论。Verifier 期间禁掉 Agent 网络，也推不出 Agent 阶段从未泄漏，平台层还得专门测试网络限制是否真正生效。

动手实验

设计一个需要启动本地 HTTP 服务的任务，写明环境镜像、端口、CPU 与内存、Agent 网络、Verifier 网络、哪些目录可以写，以及默认使用哪个用户。随后为 Agent setup、run、日志下载、verification、cleanup 各列出两种可能失败，并说明应该留下什么证据。再比较同一环境与分离环境怎样做验证，分析两者各自暴露多大的攻击面，又要付出多少复制成本。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

Agent 若要在运行时安装依赖，可以给它配置 allowlist，Verifier 通常则可以禁网，免得外部答案干扰测试。允许写入的路径要尽量少，隐藏 tests 也不能在 Agent 阶段挂载。同一环境里的 Verifier 能看到完整的进程和文件状态，却可能读到 Agent 篡改过的测试，分离环境隔离得更彻底，但必须可靠地传递待验工作区，并让两边的 OS 与依赖保持一致。

每个阶段至少要留下起止时间、状态、命令或实现 identity、stdout/stderr 摘要、相关 artifact digest 和异常类型。cleanup 失败不能把已经得到的 reward 改写成成功，而且原始异常仍要保留。

如何核对

先看 src/harbor/environments/base.py，核对 capabilities、resource mode、scoped_exec_env、scoped_output_callback、reset/ensure dirs 和 exec 分别怎样实现。再到 src/harbor/trial/trial.py，沿着 network plan 和 hooks 往下追，确认程序按什么顺序切换阶段并收尾环境。

本篇不能证明什么

配置里即使声明了禁网、资源受限和容器隔离，也证明不了底层平台真的执行了这些限制，更排除不了内核逃逸、侧信道或并发串扰。每一种具体的 Environment provider 都要单独接受安全测试和故障注入测试。

上一节 · 下一节

Verifier、Reward 与 Result：成功数字怎样回连执行证据

上一节 · 下一节

本篇要解决什么问题

终端任务最后通常会给出 reward 0/1，或一组带名字的 rewards，看上去比 Judge（裁判模型）更客观，可 reward 文件仍可能根本不存在，也可能是空文件、格式写错、由错误版本的测试生成，甚至早就被 Agent 偷偷写好了，因为一个数字代替不了整条产物链。Verifier（验证器）是否可信，要看测试有没有受控、各阶段有没有隔离、产物能不能一路对上。这一篇会追踪 Harbor Verifier 怎样放入测试、运行验证、解析 reward，再看 TrialResult/BenchmarkResults 怎样留下异常并汇总结果。

读完后，你应该能分清五种情况：测试正常运行后给出合法的 reward 0，Verifier 自己崩溃所以没有 reward，Reward 无法解析，Agent timeout，以及 regrade。你还要能说清 accuracy 和 pass@k 各自按什么单位统计，又有哪些数字不能从中推出。

先建立源码地图

源码位置	责任	阅读焦点
src/harbor/verifier/verifier.py	测试目录、Verifier 执行、reward 解析	合法失败与基础设施错误
src/harbor/models/trial/result.py	TrialResult、ExceptionInfo、TimingInfo、StepResult	结果如何保留上下文
terminal_bench/harness/models.py	BenchmarkResults、accuracy、pass@k	多次 Trial 怎样聚合

完整调用链

Reward 是证据链末端的测量值，不是孤立真相

文件缺失、空或非法是验证错误，不能为了计算方便写成 reward 0

1 Verification phase 解析 Task 的 tests 和 verifier 配置，确定测试来源、执行命令与工作目录；测试文件被加入 verifier environment，而不是暴露给 Agent。

2 Verifier 清理/准备 verifier 目录，将 tests 注入环境，运行测试命令并下载 verifier 输出；命令退出状态和日志是诊断证据，但最终 reward 由约定文件提供。

3 若存在 reward JSON，解析为具名数值映射；否则尝试 reward 文本并包装为 {"reward": number}，而 JSON 优先级和路径是外部契约的一部分。

4 文件不存在、文件为空、数值/JSON 无法解析、测试目录下载失败分别抛出专门异常。它们不会被伪造成 reward 0。

5 成功解析得到 VerifierResult；TrialResult 同时保存 AgentInfo、AgentContext、verifier_environment_mode、exception、各阶段 TimingInfo，以及单步或多步 StepResult。

6 Regrade 可复用源 Trial 的 Agent 产物并重新运行 Verifier。新结果必须回连 source trial 与 verifier 版本，不能伪装为 Agent 再次执行。

7 Job/Benchmark 聚合每 task 的 Trial reward。accuracy 表示成功 Trial 比例；pass@k 按每 task 的 n 次观察和 c 次成功估计至少一次成功概率，再跨 task 平均。

关键数据结构

VerifierResult 用字符串把名字映射到数值，所以既能保存主 reward，也能保存各个组件的 reward，但每个值表示什么、允许落在哪个范围，都要看 Task/Verifier 怎样定合同，不能默认它们全在 0 到 1 之间。ExceptionInfo 记异常类型、消息和 traceback，TimingInfo 记每个阶段何时开始、何时结束、持续多久，StepResult 则为多步任务逐步留下 Agent、Verifier 和 exception。需要统计成本时，TrialResult.compute_token_cost_totals 再从单步或多步 AgentContext 里把数据加总。

BenchmarkResults 计算 pass@k 时，会先按 task 分组，再数每个任务成功了几次，不会把所有 Trial 拆散后混成一池。若各个 task 的 n 不同、结果并非随机缺失，或 Task 本来就很少，只报一个平均值会遮住太多信息，必须同时给出分母、分布和不确定性。各项 reward 也要预先登记，不能事后只挑最有利的指标展示。

实现取舍与失败语义

通过文件交换结果，任意语言编写的 tests 都能脱离 Python Harness 独立运行，也会自然留下清楚的 artifact，但路径、格式和原子写入必须严格遵守约定。程序优先读取 JSON，可以一次提供多个 reward，文本格式则用来兼容只有单项结果的简单任务。解析器针对不同故障抛出专门异常，才能避免把基础设施问题误记成 Agent 失败，也不会把证据边界搅乱。

regrade 不用重复昂贵的 Agent 执行，就能修复 Verifier 或套用新标准，可它也改变了 Scorer identity，因此报告必须并列保留原来的 verifier result，不能直接覆盖旧结果，否则就会切断证据链。pass@k 适合回答「允许多次尝试时能否成功」，可线上若只有一次机会，只展示 pass@10 就会误导读者。发布 Gate 还得处理环境错误、缺失 Trial、任务权重，以及关键任务之间不能互相补偿的规则。

动手实验

设计一份 reward JSON，里面包含主 reward、功能正确性、安全和资源合规四项，并给每一项规定合法范围。接着列出测试失败、reward 文件为空、JSON 里混入字符串、下载失败时，Trial 分别应该记成什么状态。再为两个任务各安排 3 次 Trial，假设 A 成功 2 次、B 一次也没成功，算出 accuracy 和 resolved task 数，并说明 pass@1 到 pass@2 会怎样变化，最后列一份 regrade 证据清单。

```
python scripts/sources.py verify
python -m pytest tests/test_harness_course_docs.py -q
```

预期输出与答案

测试正常运行但断言失败时，Verifier 可以按约定写入合法的 reward 0。其余三种情况分别说明 verifier、解析或传输出了错，不应该伪造成 0。所有 Trial 合在一起时，accuracy 是 2/6，若把 resolved task 定义成「至少成功一次」，结果就是 1/2。pass@2 会大于或等于 pass@1，但任务 B 仍是 0，具体数值要按锁定公式计算，并报告每个 task 的 n/c。

Regrade 至少要保存 source trial id 和 artifact digest、新旧 verifier commit/tests digest、环境 identity、regrade 时间、新旧 rewards 以及结果为何变化，因为它只会重新评分，不会让 Agent 再跑一次，也不会产生新的 Trial。

如何核对

先看 src/harbor/verifier/verifier.py，依次核对 _resolve_tests、verify、reward JSON/text 解析以及各类专门异常。再到 src/harbor/models/trial/result.py，检查代码怎样保存异常、阶段时间和多步字段。最后查看 terminal_bench/harness/models.py，确认 pass@k 究竟按什么单位聚合。

本篇不能证明什么

藏起 tests、把 reward 写成数值、再算出 pass@k，也证明不了测试没有漏洞、reward 真能代表实际价值，或 Agent 从未利用环境，更证明不了观察到的差异具有统计显著性。要下发布结论，还必须结合任务治理、错误率、不确定性和关键风险 Gate。

上一节 · 下一节

最小 Eval Loop：从冻结规范到可复核报告

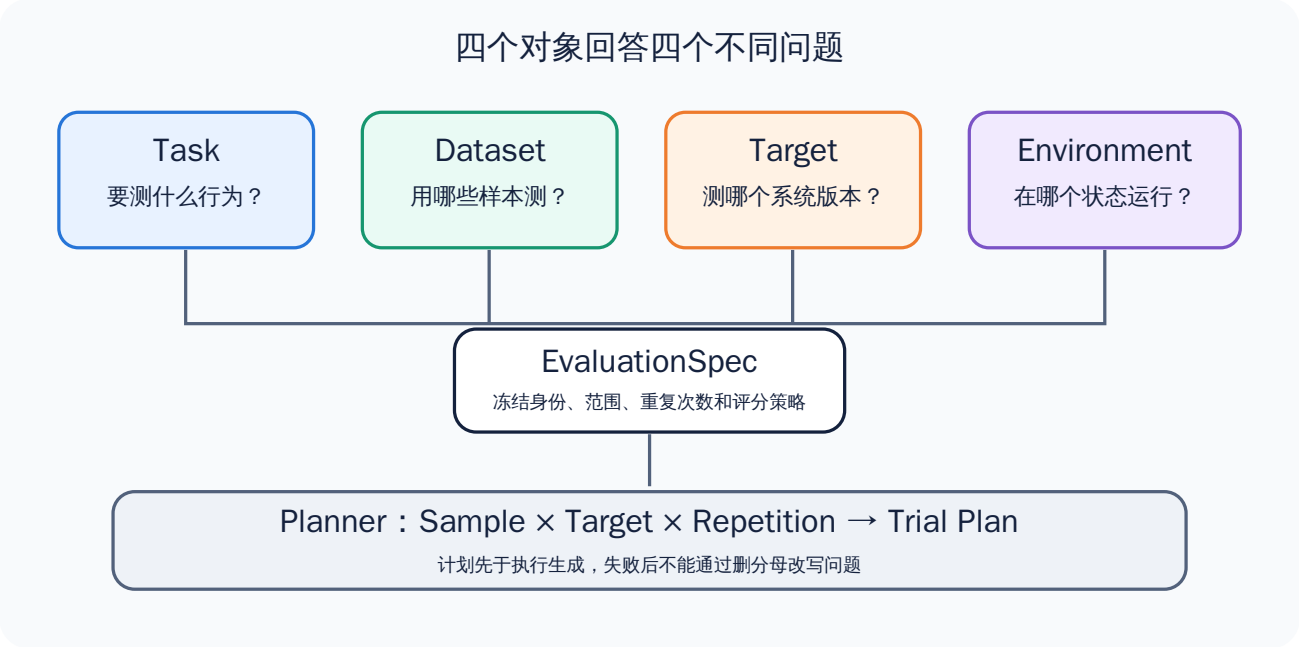
上一节 · 下一节

本篇要解决什么问题

你很容易把 Eval Loop 写成「读数据、调用模型、算平均分」三行伪代码，可这么一写，谁负责保证结果能够复核，就全被藏起来了。真正开跑以前，实现既要固定样本和 Target 的身份，也要给每个计划项分配稳定的 Trial。基础设施一旦失败，系统必须留下 Attempt（尝试），Target 的输出也得写进 Trace 和 Artifact，后面的判断才有证据可查。Scorer 只能读取明确的 Observation，Metric 必须保留预先声明的分母，Gate 也不能相信 Target 自报的 success。这几层不能混。本篇就拿 Reference Harness 的运费边界案例来走一遍，看这些责任怎样首尾相接。

只要你有 Python 基础，也熟悉前七篇用过的共同术语，就可以从 CLI 一路查到 JSON、Markdown 和 HTML 报告，并指出某个结果在哪一层变了状态。这里跑通的仍然只是经过裁剪的教学简化版本。Reference Harness 没有实现分布式队列，也不会真的调用模型，但你在生产系统里仍然绕不开这些关键对象各自负责的事。

核心机制



PipelineConfig 先冻结 evaluation_id、Dataset 路径、Target Adapter、repetition、Scorer 和 Gate，plan_trials 随后才把 sample × target × repetition 展开成具体计划，所以执行还没开始，分母就已经定下来了。run_trial_batch 会用有限的本地并发执行这些计划，但仍按原来的计划顺序返回结果，因此并发不会改写 Trial 的坐标。进入每个 Trial 后，run_trial 管理基础设施 Attempt，拿到成功输出便生成两个 TraceEvent 和按内容寻址的 Artifact，再由 build_observation_bundle 把它们绑定到 canonical Attempt。

评分时，Scorer FieldMatchesExpectedScorer 只从 target_completed 事件里读取 output 和 expected，因为 Target 怎么评价自己，不能算独立证据。aggregate_pass_rate 用计划中的 Trial ID 计算 Metric 的 denominator，即使某个 Trial 没有 Score，也不会偷偷缩小分母。evaluate_gate 一旦发现证据 invalid、unscorable 或 uncertain，就直接返回 inconclusive，只有证据有效时才套用预先声明的 threshold。write_report 则拿同一份 EvaluationReport 生成机器可读的 JSON、供人审阅的 Markdown 和独立 HTML。

完整流程

以 reference/examples/shipping/eval.yaml 为入口：

1. CLI run 检查配置路径，将错误压缩为中文消息而不泄漏 traceback。
2. Pipeline 在配置目录内安全解析 dataset/script，拒绝绝对路径和 .. 越界。Dataset JSONL 每行验证为 Sample。
3. Planner 为三个金额、两个 Target、一次 repetition 生成六个稳定 Trial。Trial ID 包含运行、Target、Sample 与重复身份。
4. SubprocessTarget 通过 argv 和 stdin JSON 调用本地脚本，禁用 shell。超时/启动失败抛 InfrastructureError，非零退出或非 JSON 输出是 product failure。
5. Runner 只重试 InfrastructureError。一旦 Target 返回有效产品失败或完成结果，就建立唯一 canonical Attempt，不用重试「刷过」。
6. TraceWriter 强制事件序号和父事件存在。ArtifactStore 以 SHA-256 命名输出。ObservationBundle 摘要覆盖 Trial、Attempt、事件与 Artifact 引用。
7. Scorer、Metric 和 Gate 逐层产生不可变对象。报告与 evidence/run 文件最后写入输出目录。

关键数据与不变量

你可以按谁负责什么，把整条链路分成四组：EvaluationSpec → Sample → Trial 负责计划，Attempt → TrialResult 负责执行，TraceEvent → ArtifactRef → ObservationBundle 保存证据，ScoreRecord 记录每次评分，MetricEstimate 汇总这些评分，GateDecision 再保存门禁判断。把这些对象串起来以后，关系就不能随意改：一个 Trial 最多只能选出一个 canonical Attempt，Bundle 只能绑定这个 Attempt，Score 还必须带上 Bundle digest 和 scorer_id。Metric denominator 必须从 Trial plan 里取，否则证据一缺，分母就会悄悄变化，Gate 也可能把 invalid 或 unscorable 的证据硬判成 passed。

错误状态也不能全塞进一个布尔值，因为 Attempt 分为 succeeded、infra_failed、cancelled，Trial 分为 completed、blocked、invalid，Score 和 Gate 还各有自己的状态集合。因此，报告里的「fixed-release passed」是 Gate 根据证据判出来的结论，不能把它理解成脚本退出码换了个写法。

动手实验

从仓库根目录运行：

```
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/shipping
uv run eval-harness-ref inspect output/shipping
uv run eval-harness-ref score output/shipping
uv run eval-harness-ref gate output/shipping
```

打开 output/shipping/evidence.json 后，先任选金额 100 的 buggy Trial，沿 trial_id → canonical_attempt_id → bundle digest → score_id → metric_id → gate_id 写出完整链路。接着修改输出 Artifact 的一个字节，再运行 inspect，观察摘要门禁如何拦下被改写的证据。

预期输出与答案

运行后，你会看到 buggy-release：failed 和 fixed-release：passed，而计划里共有 6 个 Trial，两个 pass-rate 的 denominator 都是 3。金额为 100 时，buggy 输出 fee=10，期望值却是 fee=0，所以这项 Score 是 failed，buggy 也只通过了 2/3，没有达到 minimum=1.0。只要有人改了 Artifact，inspect 就应该报「Artifact 摘要不一致」，不能继续显示原来的 Gate。

重新运行 score 或 gate 时，系统不会再跑一次 Target，因为这两个命令只读取已经冻结的 evidence。这样才能把「执行证据」和「政策判定」分开，不过重新判定以前，系统必须先确认 evidence 完整无损。

如何核对

从 cli.py 的 run 进入 pipeline.py，然后按照调用顺序依次追 _load_samples、plan_trials、run_trial_batch、build_observation_bundle、scorer、metric、gate 和 write_report。相应的测试入口可以从 test_shipping_e2e.py 与 test_cli.py 开始找。

本篇不能证明什么

确定性案例全部通过，只能说明这套小型实现符合当前测试合同，无法证明真实模型足够稳定、数据集能代表线上分布，也无法证明当前并发方式撑得住大规模运行，更不能因此授权生产发布。Reference Harness 提供了一套可以运行、也方便讨论的共同语言，但它不是企业控制面。

上一节 · 下一节

Run Identity 与可复现性：保存「实际运行了什么」

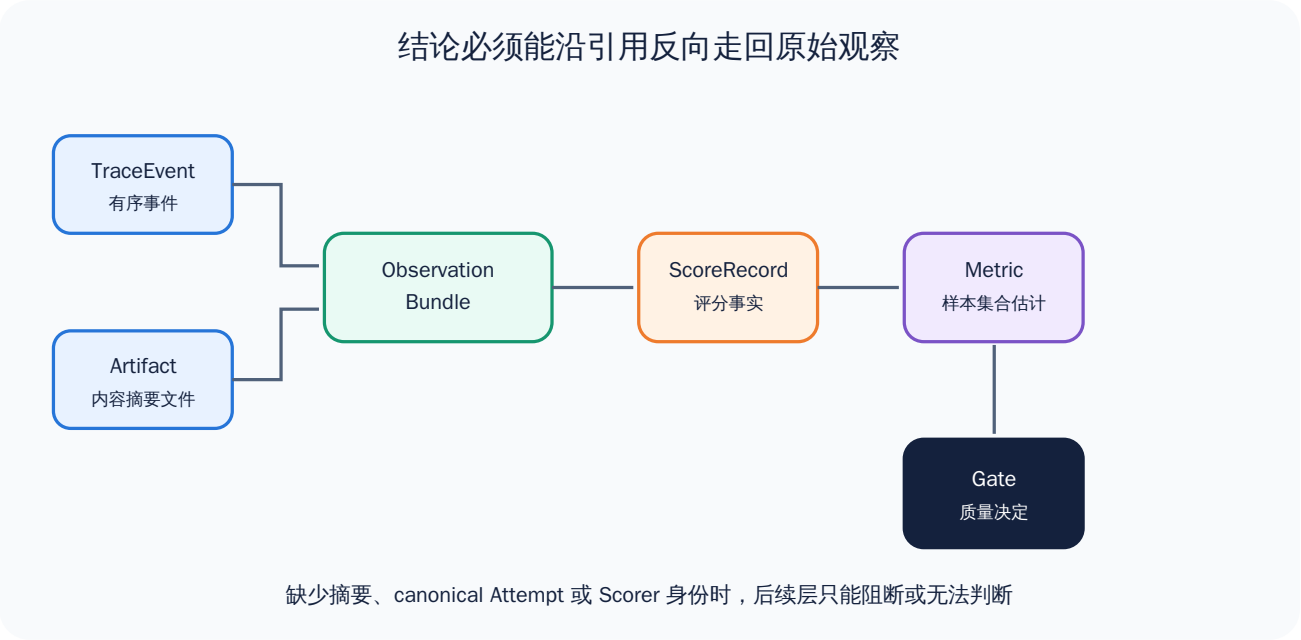
上一节 · 下一节

本篇要解决什么问题

只写下「使用 model-x 在 dataset-v2 上运行」，还不足以让别人复现这次运行，因为模型别名会漂移，Dataset 可能被覆盖，脚本也可能依赖当时的工作目录。环境变量、Scorer prompt 和阈值同样会变，只留两个名称，根本还原不了运行条件。Eval Harness 必须先把声明的身份解析成实际使用的身份，再用摘要把配置、输入、代码和产物连起来。本篇会围绕 canonical_digest、安全相对路径、Trial ID 和 Artifact digest，讲清这条身份链至少要留下什么。

读完以后，你应该能设计一份 RunManifest，分清 logical name、resolved identity 和 content digest，也能判断两份报告是否具备直接配对的条件。这里说的「可复现」，不保证远程 API 将来还会返回相同文本，只保证你能查明当时实际用了什么，并在可控的组件上重新搭出相同条件。

核心机制



缺少摘要、canonical Attempt 或 Scorer 身份时，后续层只能阻断或无法判断

Reference Harness 先给键排序，固定分隔符和 UTF-8 编码，再对规范化后的 JSON 计算 SHA-256，因此两个映射只要语义相同，就不会因为键的顺序不同而得到不同 digest。Planner 取 EvaluationSpec digest 的一部分写入 run_id，再把 target_id、sample_id 和 repetition 拼成 Trial ID。ArtifactStore 直接对真实 bytes 求摘要，Bundle digest 则继续覆盖 Artifact 引用和 Trace 事件，于是你可以沿同一条身份链查回每份产物。

身份不能只看摘要，因为摘要只能回答「内容是否相同」，各个字段才会说明这些内容分别拿来做什么。因此，RunManifest 应当保存 evaluation_id、源码 commit、Python 和依赖版本、Dataset digest、Target Adapter（被测对象适配器）identity、实际模型或镜像、Scorer identity、Gate policy、随机种子和时间。秘密只记录它来自哪里，千万别把秘密值直接写进 RunManifest。

完整流程

1 用户配置是 declared identity，加载器验证 schema，拒绝未知字段，避免拼错配置被静默忽略。

2 配置路径解析必须局限于配置目录。绝对路径或 .. 越界会让同一配置读取机器上的不同文件，因此 _safe_input_path 直接拒绝。

3 Dataset 每行解析为带稳定 sample_id 的 Sample，内容摘要应覆盖规范化输入与期望，而 sample_id 只负责逻辑引用。

4 Target 字符串解析为具体 Adapter 和参数。本地脚本保存相对路径及文件 digest，真实 API 还要记录服务返回的实际模型版本（若可得），不能把别名假装成不可变版本。

5 Planner 依据冻结 Spec 生成全量 Trial ID。恢复时必须对 manifest/digest，而不是只检查相同文件名。

6 每次 Attempt 继承 Trial identity，并增加 ordinal。canonical 标记是执行恢复结果，不改变 Trial 的统计身份。

7 Artifact 与 Bundle 用内容摘要形成 lineage，inspect 重新读取 bytes 校验摘要，防止引用存在但内容已被替换。

8 Comparison 只在 candidate/baseline 共享 sample_id + repetition 且测量合同相同时配对。

关键数据与不变量

逻辑身份可以写成 target_id=fixed，解析身份要记录脚本的相对路径、digest 和解释器版本，到了具体实例，还要带上 trial_id 与 attempt_id。三类身份缺了哪一类都不行。Digest 必须带 sha256: 算法前缀，否则以后换了算法，你就无法解释旧摘要是怎么算出来的。Artifact relative_path 也必须限制在运行目录里，否则同一个引用换台机器，可能就指向了另一份内容。

想直接比较两次运行，至少要保证 Dataset 的内容和切分方式、Sample 配对键、Target 以外的政策，以及 Scorer 和 Metric 定义都一致。怎样处理缺失数据，也必须提前定好。要是看到结果以后再补规则，就等于倒过来修改实验设计。Judge 模型或 prompt 只要变过，即便 metric_id 没变，两次运行用的也已经不是同一份测量合同。

动手实验

运行身份测试：

```
uv run pytest tests/test_identity.py tests/test_models.py tests/test_runtime_extensions.py -q
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/identity-
demo
```

先把 evidence.json 中某个 Artifact relative_path 改为 ../README.md 并执行 inspect，观察路径越界如何被拦截。恢复后再改动 Artifact bytes，这次检查内容摘要校验。最后复制 shipping 配置，只改 target_id 而不改脚本，再判断两次运行是「内容相同」「逻辑身份相同」，还是「可直接比较」。

预期输出与答案

路径一旦越界，系统会在加载 ArtifactRef 时拦下它。有人篡改 bytes，则会在核对摘要时失败，因为这两道检查守的是不同边界。只改 target_id 时，脚本虽然没变，逻辑 Target identity 和报告坐标却已经变了。rename mapping 可以说明两边的实现内容等价，但你不能只凭 digest 就断言实验条件相同，配置和运行环境仍然要逐项核对。

canonical_digest 不会受字典键顺序影响，但列表的顺序本身有含义，所以只要样本、Target 或事件换了顺序，digest 就应该跟着变。SHA-256 只能证明内容一致。它证明不了来源可信，也看不出文件是否带有恶意。

如何核对

阅读 identity.py、planner.py、artifacts.py 与 models.py 的路径验证。再查看 test_identity.py 和 Artifact 安全测试。

本篇不能证明什么

内容摘要、锁文件和稳定 ID 可以打下审计基础，却保证不了远程模型每次都给出相同结果，也证明不了容器镜像的供应链安全。依赖时间的服务未必能够重放，作者身份也得靠额外证据确认。这些记录不是密码学签名，也没有覆盖可重复实验所需的全部条件。

上一节 · 下一节

Retry 与 Recovery：恢复基础设施，不能重抽产品答案

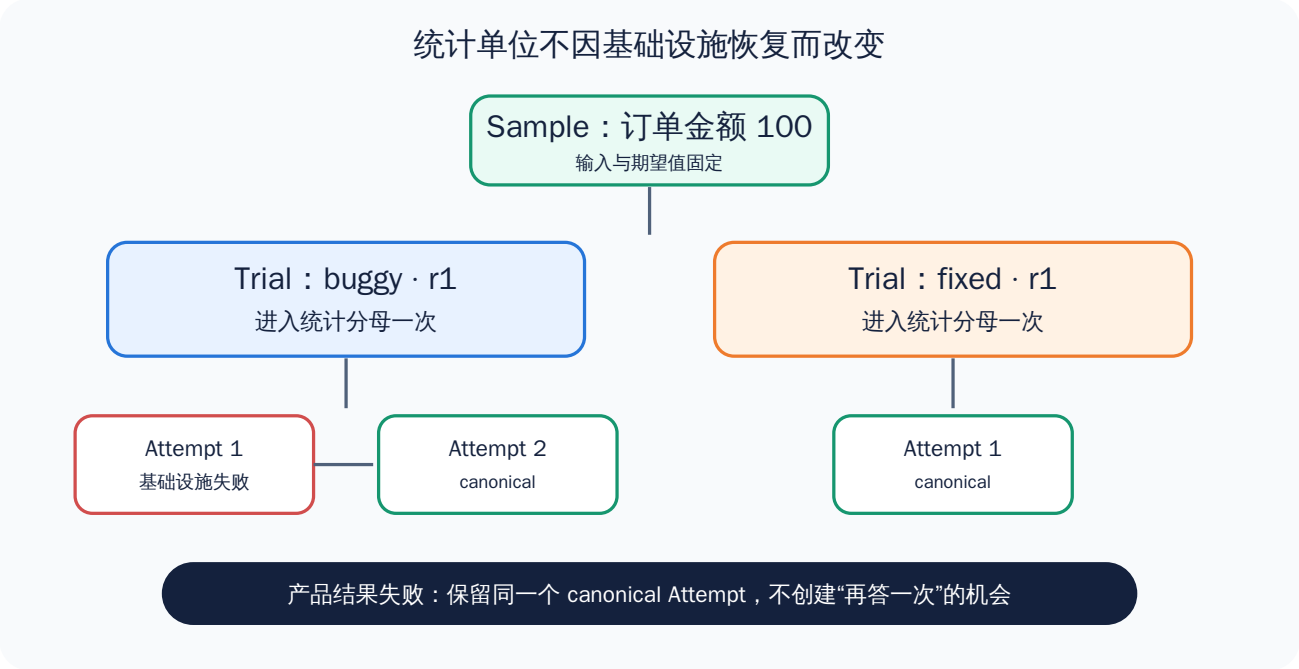
上一节 · 下一节

本篇要解决什么问题

模型第一次答错、第二次答对时，Harness 能不能自动重试，再把这个样本记成通过？不能。边界要卡在这里。只有产品的服务契约原本就允许重试，而且这段重试属于被测 Target 自己的行为，第二次回答才能算进有效的产品观察。Eval Harness 的 retry 只负责从「没有拿到有效产品观察」的基础设施故障中恢复，否则系统就会反复抽取失败样本，直到碰巧成功，连分母和失败概率也跟着变。本篇会借助 Trial（试验）、Attempt（尝试）、各自独立的行为预算和 canonical Attempt，把两类看起来相似的重试分清楚。

熟悉 Sample、Trial 和 Attempt 以后，你应该能判断 timeout、429、进程启动失败、模型拒答、工具错误、断言失败和日志写入失败各自属于哪一类。把错误分层，是为了让状态机恢复运行时不会顺手改掉产品结果，因为错误落在哪一层，直接决定了系统能不能重试。

核心机制



Trial 是统计计划预先定下的观察单位，Attempt 则记录同一个 Trial 每次怎样调用基础设施，因此重试可以多留一个 Attempt，却不能凭空多出一个 Trial。Reference Harness 的 run_trial 捕获 InfrastructureError（基础设施错误）后，先写下一条 infra_failed Attempt，再根据 max_infra_attempts 决定是否继续。Target 一旦返回 TargetResult，无论 kind 是 completed 还是 product_failure，当前 Attempt 都会成为唯一的 canonical。重试必须就此停止。如果基础设施 Attempt 全部耗尽，Trial 就会变成 blocked，此时没有 Bundle 和 Score，但 Metric denominator 仍然要把它算进去。

产品服务内部怎样退避和重发，属于 Target 行为，比如应用遇到 429 后按照线上政策再次请求，Harness 只能在 Trace 里记录这段过程。Harness retry 处理的是运行器启动不了进程、沙箱暂时失联等实验基础设施问题。两套预算不能混。事件类型也要分开记录，否则运行器无法说明自己恢复的是实验环境，还是重新抽取了产品结果。

完整流程

1. Planner 固定 Trial 计划与 max_infra_attempts，开始后不能根据结果追加 Trial。
2. Runner 创建 Attempt ordinal=1 并调用 Target Adapter。
3. Adapter 在进程无法启动、Harness 级 timeout 等情况下抛带 code 的 InfrastructureError。Runner 记录失败但不生成产品 output。
4. 若预算剩余，创建下一个 Attempt。每次 Attempt 都应保存开始/结束、错误、资源与 Trace，不能覆盖前一次。
5. Target 有有效返回时建立 canonical Attempt。非零退出、拒答、无效 JSON 若定义为产品 contract 失败，也停止重试。
6. ObservationBundle 只绑定 canonical Attempt，但审计报告仍保留所有 Attempt，解释为何选中它。
7. 若无 canonical Attempt，Trial blocked。Scorer 不应伪造 0 分。Metric 保留计划 denominator，Gate 可因关键 blocked evidence 给 inconclusive/blocked。
8. 分布式实现还需要 lease、fencing token 和原子 canonical commit。最小本地实现不假装已解决这些问题。

关键数据与不变量

AttemptStatus={succeeded, infra_failed, cancelled} 记录一次基础设施调用的结果，TrialStatus={completed, blocked, invalid} 则说明整个计划观察最后落在什么状态。系统用 Trial ID 加 ordinal 算出 Attempt ID，一个 Trial 最多只能选出一个 canonical，而且这个 canonical 必须是 succeeded。即使产品失败，Attempt 仍然可能是 succeeded，因为「成功执行」只说明系统拿到了有效观察，不等于「产品通过」。

预算至少要分开记录 Harness infra attempts、Target 内部调用和工具预算、Judge 预算，以及全局墙钟预算，因为每一种预算耗尽，都代表不同的失败。系统必须把允许恢复的错误逐项列入白名单。要是所有 Exception 都能触发重试，系统就会不断重复确定性的程序错误，反而把问题藏住。取消后也不能自动继续，因为用户操作、全局预算耗尽或上游撤销任务，都可能触发取消。

动手实验

运行 Runner 测试：

```
uv run pytest tests/test_runner.py -q
```

阅读三个测试 Target，它们分别是第一次 InfrastructureError 后成功、直接 product_failure，以及所有尝试都返回 InfrastructureError。为每种情况画出 Attempt 列表，并标出 canonical、TrialStatus、是否产生 Bundle 以及 Metric denominator。然后假设产品模型第一次输出了错误答案，解释为什么这个结果不应触发 Harness retry。

预期输出与答案

基础设施先失败、随后成功时，系统会留下两个 Attempt，并把第二个选作 canonical，Trial 的状态则是 completed。Target 直接返回 product_failure 时，只会产生一个 succeeded canonical Attempt，Trial 仍然是 completed，不过 Scorer 后面可能把它判成 failed。如果每次尝试都遇到 infra failure，系统会保留预算范围内的全部 infra_failed Attempt，Trial 因为找不到 canonical 而变成 blocked，也不会生成 Score，但计划 denominator 仍然包含它。

错误答案本身已经是一条有效的产品观察，所以 Scorer 应当把它判成 Score failed，Harness retry 不能拿它再试一次。如果业务 Target 自己规定「最多请求模型三次再投票」，那么三次请求都属于同一个 Trial 里的 Target Trace，既不能记成 Harness 的三个 Attempt，也不能冒充三个独立样本。

如何核对

查看 runner.py 的循环与返回分支、targets/base.py 的错误合同、test_runner.py 的三种状态；再结合 metrics.py 核对 blocked Trial 是否仍占分母。

本篇不能证明什么

本地线程和单进程 canonical 规则只能管住这份参考实现，证明不了分布式 worker 恰好执行一次，也无法保证远端请求幂等，或崩溃恢复没有造成重复副作用。因此，生产队列还得增加 lease、fencing、幂等键和事务性提交，才能在分布式环境里守住同样的边界。

上一节 · 下一节

LLM-as-a-Judge：先定义测量合同，再接入模型

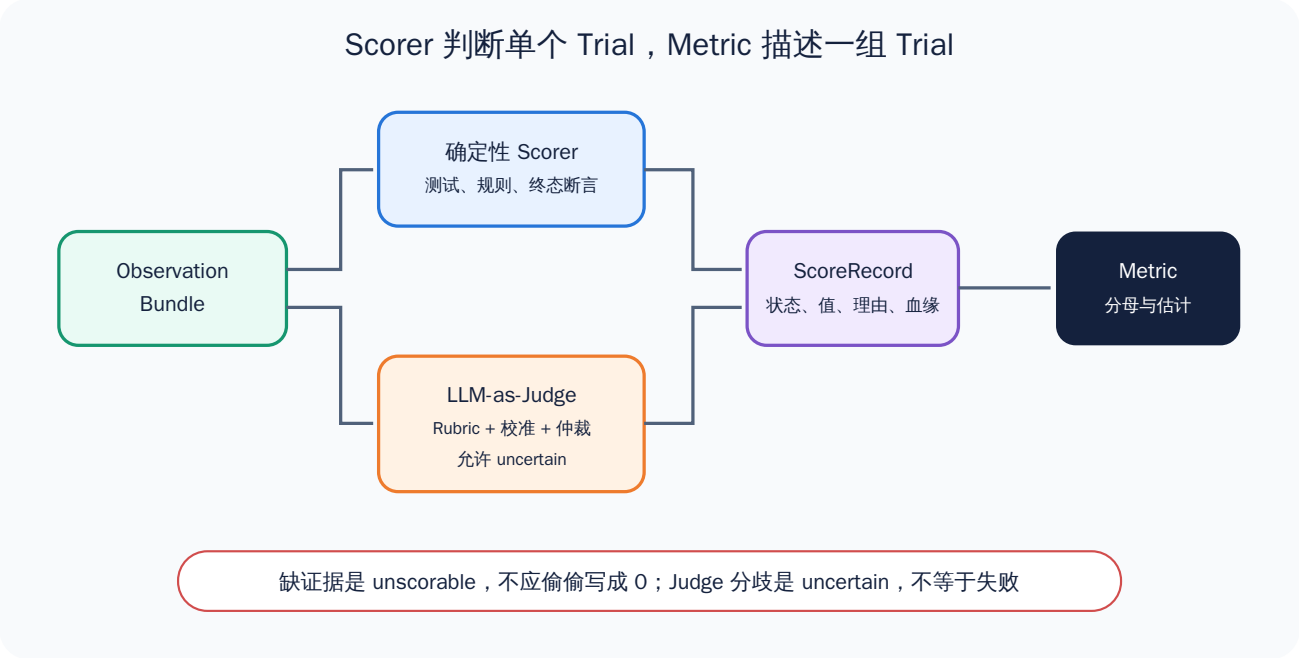
上一节 · 下一节

本篇要解决什么问题

开放式回答、合同风险和多轮 Agent 轨迹很难靠精确匹配来评分，所以不少评测会直接接入 LLM-as-a-Judge。可是，只说「让更强的模型打 0 到 1 分」还算不上完整方案，因为 rubric（评分标准）、输入能看见什么、输出 schema、失败与弃权、位置偏差、校准数据和成本，全都没有交代。Judge（裁判模型）会带来自己的不确定性，不能把它当成事实裁判。本篇从 Reference Harness 的离线 Judge Protocol 入手，看适配层怎样既允许替换具体实现，又让你能够核对证据。

核心路线默认不访问网络。只要对象实现了 judge_id + judge(ObservationBundle)，JudgeScorer 就可以接收它，离线 Stub 和真实 API 也共用同一份结果合同。这样测试既能锁住 lineage，也不会把模型凭据和模型漂移带进基础门禁。

核心机制



Judge 只能读取已经冻结的 ObservationBundle（观测包），而且你要明确规定它能看 output、reference、trace、tools 和 artifacts 里的哪些内容，否则隐藏标签或候选名称可能直接泄漏给评分器。JudgeResult 保存 value、passed 和 reason，JudgeScorer 再补上 scorer_id、judge_id、Bundle digest、canonical Attempt 与 ScoreStatus。调用评分模型时如果需要 retry，也只能恢复 Judge 这一侧的基础设施，不能顺手把 Target 重新跑一遍。

要可靠地接入 Judge，先得用 Rubric 写清要测什么、哪些证据可以观察，再让 Judge Adapter 固定模型、prompt、schema、参数和重试政策。到了 Calibration 阶段，再拿人工双标和仲裁数据去测一致率、偏差、稳定性与阈值。三层都齐了，才构成一份可以检验的测量合同。Release Eval 要使用冻结的 Judge 版本，训练 reward 所用的 Judge 也必须与独立发布 Judge 隔离，否则训练过程会针对发布标准直接优化，最后污染这把尺子。

完整流程

1. 从业务风险定义 rubric 条目和不可补偿规则，给出 passed/failed/uncertain/unscorable/invalid 的可操作条件。
2. 构造 Judge input view，只含授权 Observation。对 Candidate/Baseline 比较时随机化顺序并隐藏系统名称。
3. 固定 Judge identity：provider、model/version、系统提示、rubric、few-shot、温度、seed（若支持）、输出 JSON schema、解析器 commit。
4. 调用 Judge，验证结构化输出和值域。解析失败、超时和拒绝属于 Judge error，不是产品 0 分。
5. JudgeResult 经 JudgeScorer 转为 ScoreRecord，lineage 回到 Bundle。reason 作为诊断，不当作模型思维链或事实证明。
6. 在独立校准集上与至少两位人类标注和仲裁结果比较，报告分层一致率、混淆矩阵、阈值敏感性和重复测量方差。
7. 未达到预注册质量要求时，Judge 结果为 uncertain/unscorable 或进入人工复核，而不是调 prompt 直到发布通过。
8. 运行中记录 token、成本、延迟、缓存与错误。报告同时展示 Judge error rate。

关键数据与不变量

Judge identity 和 Scorer identity 都必须带版本，因为 Rubric 哪怕只改一个词，模型别名发生漂移，或输出解析器换了实现，都可能改变测量合同。reason 不该保存隐藏的 chain-of-thought，只要给出能够审计的判定依据、所引证据和 rubric 条目就够了。如果 Judge 允许 abstain，ScoreStatus 就应该保留 uncertain 或 unscorable，不能一律压成 failed。

做成对 Judge 时，系统要保存 pair key 和展示顺序，再交换 A/B 评一次，这样你才有机会查出位置偏差。即使多个 Judge 一起投票，统计单位仍然是原来的 Trial，不能把 Judge 的数量冒充样本量。人工仲裁结果要单独保存。原始分歧也得留下。

动手实验

运行离线接口测试：

```
uv run pytest tests/test_runtime_extensions.py -k judge -q
```

先为合同风险案例写一个包含 high/medium/low 三档的 rubric，并增加 uncertain 条件。然后定义 StubJudge，让它返回 value=0.8、passed=true 和 reason，再沿输出检查 ScoreRecord 的 trial_id、canonical_attempt_id、bundle digest 和 scorer_id。最后列出真实 Judge 接入时必须新增的配置与校准报告字段，观察离线 Stub 隐去了哪些生产责任。

预期输出与答案

离线测试不会访问网络，但 Score 仍然要保存 value=0.8、reason「满足离线规则」和完整 lineage。换成真实配置以后，你至少还得补上 judge_id、provider/model version、prompt/rubric digest、schema、参数、重试与超时、cache policy、成本，以及校准数据版本。

如果 Judge 超时，系统应该记录 Judge error 或不可评分，不能把产品直接记成 0 分。如果两位人工标注者意见不同，仲裁又没有完成，结果就应该是 uncertain，因为测量结论还没定下来。要是 Judge 在关键的高风险条目上持续漏判，那么整体一致率再高，也不能把它接到 release gate 上。

如何核对

阅读 scorers/judge.py 的 Protocol、JudgeResult 与 lineage 适配，核对 test_runtime_extensions.py 的离线 Stub。再对比 scorers/rules.py，理解确定性规则与 Judge 的失败面不同。

本篇不能证明什么

结构化输出、reason、多人投票和很高的总体一致率，都证明不了 Judge 没有偏差，也不能说明它真正理解业务，足以判断高风险发布。校准结论只适用于当时的数据分布、rubric 和模型版本，其中任何一项发生变化，都要重新验证。

上一节 · 下一节

统计比较：先配对 Trial，再看效果量与区间

上一节 · 下一节

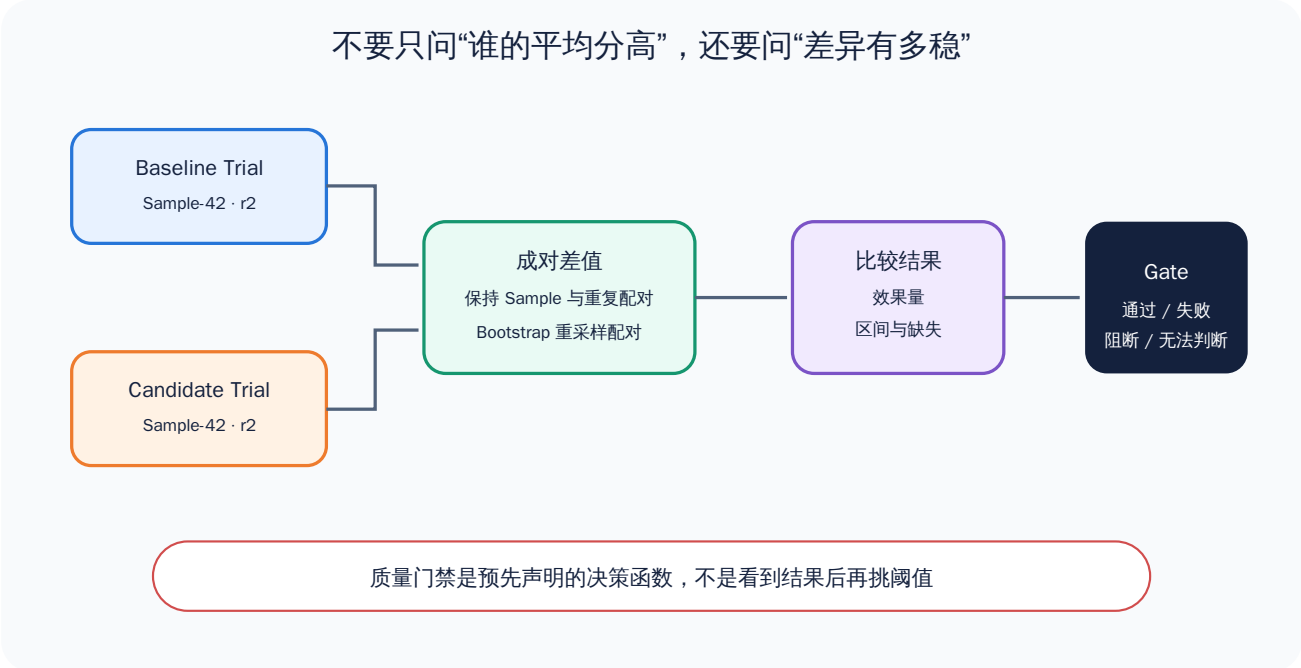
本篇要解决什么问题

Candidate（候选版本）的通过率是 82%，Baseline（基线版本）是 80%，从平均数看 Candidate 像是赢了，但你还不能据此决定发布。还要看两边是不是跑了同一批 Sample、重复次数能否对上、缺失是不是偏向某个 Target，还得查清这 2 个百分点由哪些任务拉出来，因为任何一项都可能改变结论。这还不够。

你还要检查区间是否跨过 0，以及关键风险有没有朝相反的方向变化，所以 Eval Harness 应当先算好指标，再让 Gate 判断能不能通过。本篇会用 Reference Harness 的配对 Bootstrap（自助法）以及 shipping 里的 candidate/baseline，把这条最小链路走一遍。

继续往下读之前，你要先知道 Trial 怎样计划、Metric 怎样得出，然后跟着本篇构造 pair key，逐对计算差值，并看懂均值差和置信区间。到这一步，你也应该能指出伪重复、可选停止和只挑赢家指标这几种常见错误。

核心机制



先把同一个 Sample/repetition 分别交给 Candidate 和 Baseline，这样两边面对同一份任务，再算差值时就能抵消一部分由任务难度带来的波动。Reference Harness 用 sample_id:r{repetition} 当 pair key，借它对齐两边的 Score value，只有两边 value 都有效时才计算这一对的差值。paired_bootstrap 拿到整组差值后做有放回采样，固定 seed 和 iterations，然后返回 pair_count、mean_difference 以及 2.5% 和 97.5% 分位数。

这里算出的只是教学用基础区间，还不足以直接套到真实实验里。真实实验里，task family、用户或会话往往会把多个样本聚在一起，重复运行同一个 Trial 也不会自动产生彼此独立的 Sample，所以你可能要改用 cluster bootstrap 或分层模型。先定方法。在看到结果前，就要把统计方法和产品所能接受的最小有意义差异一起预注册。

完整流程

1. 计划阶段固定 candidate/baseline Target、共享 Dataset、repetitions、pair key、主指标、次指标和缺失策略。
2. 两个 Target 运行时保持相同 Sample 顺序不是必要条件，但身份、环境和 Scorer 合同必须一致。
3. Score 层保留每个 Trial value 和状态；blocked/unscorable 先分层报告，不能在比较函数中悄悄删掉。
4. compare_targets 从 Evidence 找到 Trial 的 sample_id/repetition，从 Report 找到有效 score，形成两张 key → value 映射。
5. 取交集配对，计算 candidate - baseline 差值。若无共享键则直接失败。
6. Bootstrap 对 pair 差值重采样；seed 固定使教材测试可重复，并输出平均差与区间，不把区间端点解释为参数有 95% 概率位于其中。
7. Gate 将统计结果与预注册 margin、关键指标非劣要求、错误率和成本约束结合；不能用「均值为正」代替全部政策。
8. 报告展示 pair_count、缺失 pair、分层效果和单例回归，支持定位而非只公布赢家。

关键数据与不变量

先把单位分清。预先声明的 Trial 是统计单位，通常用 Sample + repetition 来配对，但如果同一个用户贡献了多个 Sample，bootstrap 真正应当聚类的对象可能是用户。Candidate 和 Baseline 两边的 Score 必须量纲相同，Scorer 版本也必须一致，而 pair_count 要等于两边有效结果的交集，同时还要另报计划中的 pair_count 和每项缺失的原因。所用的 Seed 和 iterations 也要记下来，它们同样会确定这次分析的身份。

别只盯着均值差。遇到二元指标，你还可以报差值、相对风险或 McNemar discordant pairs，遇到连续指标，则可以报均值差或中位数差。延迟和成本通常拖着长尾，这时还得看分位数，如果同时跟踪多个指标，就必须提前定好主次或校正方法，不能等结果出来后只挑那个显著的指标。

动手实验

先运行 shipping，再执行 compare：

```
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/compare
uv run eval-harness-ref compare output/compare --candidate-target fixed --baseline-target
buggy --seed 17 --iterations 2000
uv run pytest tests/test_metrics.py tests/test_cli.py -k "bootstrap or compare" -q
```

先手算金额 99、100、101 这三个 pair 各差多少，再删掉 Candidate 的一个 Score，看看 pair_count 会怎样变，以及计划分母和 Gate 是否应该跟着变。

预期输出与答案

三对值分别是 0、1、0（fixed - buggy），所以 mean_difference 是 1/3，pair_count=3，但样本太少，Bootstrap 算出的区间可能仍然包含 0。这个例子只能告诉你效果朝哪个方向走，还支撑不了普遍结论。删掉一个 Candidate Score 后，有效 pair_count 会降到 2，但计划中的 Trial 分母不能随之缩小，因此报告要把缺失列清楚，Gate 也要进入 blocked/inconclusive，不能只拿剩下的两对去宣布更高胜率。

固定 seed 后，测试便能稳定复现结果，而换一个 seed 可能会让有限迭代算出的分位数轻微浮动，却不会改变最初算出的三对差值。

如何核对

先阅读 comparison.py 和 pipeline.py 里的 compare_targets，再去看 test_metrics.py 和 CLI compare 测试，最后动手把每个映射 key 对回相应的 Evidence Trial。

本篇不能证明什么

就算 Bootstrap 算出了一段小样本区间，它也修不了数据泄漏、系统性缺失、Scorer 无效、任务分布偏差和可选停止这些问题，更不能代替你判断领域风险或审阅关键案例。

上一节 · 下一节

Agent Environment：把命令、文件和终态变成评测观察

上一节 · 下一节

本篇要解决什么问题

评测 Agent 时，你不能只看它最后回了什么文字。代码有没有修好，得看 diff 和测试，退款对不对，得查真实副作用，RAG Agent 要核对检索结果和 ACL，终端任务则要检查文件、进程和服务最后留下了什么状态。管理 Agent Environment（智能体环境）的 Harness 会创建实验环境，在每次运行前重置并限制它，运行中收集可观察行为，运行后再验证结果，而 Agent Harness 专门推进 Agent loop。两边靠 Trace、Artifact 和 environment final state 交接，本篇会结合 Harbor/Terminal-Bench 课程和 Reference Harness 的 Agent Trace Import，看看最小适配需要接上哪些环节。

读完后，你应该能把环境从创建到清理的整个过程设计出来，分清 Agent phase 和 verifier phase 各自做什么，再决定哪些日志和文件要交给 Observation。还有一条不能混：环境出错不等于产品失败。

核心机制

你至少要在环境规范里固定镜像或仓库 commit、初始状态、运行用户、工作目录、可写范围、网络和资源，同时说清怎样注入秘密、何时超时以及由哪个 verifier 验收。每个 Trial 都应从干净的 snapshot 建出环境，Agent 在受控权限下运行时，Harness 会收集结构化 Trace、stdout/stderr、diff、最终状态和成本。等 Agent 停下来后，Harness 再在当前环境或单独的 verifier 环境里执行隐藏断言，验证完才销毁环境，或者先隔离现场再保留。所以在验证真正完成以前，别急着清理现场。

Reference Harness 自己不会启动容器，但 AgentTraceImportTarget 把 Adapter（适配器）应该管到哪里说得很清楚：它读取外部 JSONL，检查 sequence、event_id 和 parent 能否对上，然后从最后一个事件取出可评分的 output。到这一步，Adapter 的职责就已经到头，不能继续往里猜。它不会倒推 Agent loop，更不会采集隐藏的 chain-of-thought 作为评测证据。

完整流程

1 TaskSpec 固定初始仓库/镜像、instruction、允许修改范围和 verifier 版本。Dataset 提供多个任务实例。

2 Environment provider 声明 capabilities。请求网络禁用、CPU 限制或 compose 时，provider 不支持就应阻断，而非静默忽略。

3 每个 Trial 创建干净环境并健康检查。build/start timeout 属于 Harness infra，允许受控 Attempt 恢复。

4 Agent Adapter 在 agent phase 注入最小凭据和环境变量，执行 setup/run。命令、工具调用、文件变化、模型事件形成 Trace 与 Artifact。

5 Agent 停止后收集日志，即使异常也尽量保存已发生行为。Trace 父子/序号验证确保基本因果闭合。

6 切换 verifier phase 的网络和权限。隐藏 tests 不在 Agent phase 可见；分离环境时只传递允许的工作区状态。

7 Verifier 输出确定性 reward/组件结果或明确 error；环境构建、复制、reward 缺失不能伪装成 0 分。

8 reset/cleanup 验证残留资源；最终状态与清理错误进入 Trial 诊断，下一 Trial 不复用污染环境。

关键数据与不变量

要识别一个 Environment，就得记下 provider/version、镜像 digest、Task files digest、资源和网络 policy、OS 以及架构，但 TraceEvent 只能保存真正观察到的行为，不能把模型私有思维链收进来。Artifact（产物）可以收纳 diff、测试日志、终态快照和 trajectory，但每一项都要有内容摘要，而且 Verifier 与 Agent 必须使用分开的 identity。

同一任务每次随机重复时都必须从等价的初始状态起步，只要环境 reset 失败，就要把后续 Trial 标成 invalid/blocked，不能带着污染继续跑。环境权限也要逐项检查，不能只看配置里的声明。安全审计要记录 Agent 可以使用哪些凭据、访问哪些网络域，但不能把秘密值写进 Artifact，Verifier 也不能直接相信 Agent 自报的成功。

动手实验

先验证 Agent Trace 导入：

```
uv run pytest tests/test_runtime_extensions.py -k trace -q
```

先写两行 JSONL：一行记 tool_call，另一行记以它为 parent 的 agent_completed，并在后一行的 payload.output 里写入 {"answer": 42}。用 agent_trace Target 跑过一次后，把 parent 换成一个不存在的 ID，看看流程会在哪里停下来。然后给代码修复任务列一份环境清单，把初始 commit、可写目录、禁网、CPU/内存、Agent timeout、隐藏 tests、diff 和 test log 都列进去。

预期输出与答案

导入合法 Trace 后，你应该看到 trace_event_count=2、两个 type 和 final_output，而 parent 只要断掉，Harness 就应该将 Trace 判为无效并停下，不能继续给出正常评分。评测代码修复任务时，Scorer 要先在干净的 verifier 环境里应用 diff 或检查工作树，再运行隐藏 tests，只读 Agent 最后那句「已修复」当然不够。

环境没能启动时，Harness 可以通过 Attempt 受控恢复，但隐藏 tests 正常运行后报失败才算产品失败，如果 tests 自己缺失，则要记为 verifier error。这两件事不能混。即使已经产生结果才发现清理失败，也要补上诊断，并禁止复用这个环境。

如何核对

读完 targets/trace_import.py 和对应测试后，再对照 Harbor 环境课程 里锁定的源码逐项检查，看看要达到生产级，Environment 还少哪些能力。

本篇不能证明什么

即使 JSONL 里的因果关系都对得上，你也不能据此证明事件确实发生过，更证明不了容器没有逃逸、网络策略真正 enforcement，或者隐藏测试没有漏洞。Trace 只能声明发生了什么，还得靠可信的采集方式和隔离环境才能托住这份证据。

上一节 · 下一节

Quality Gate：把证据转成三态/四态决定

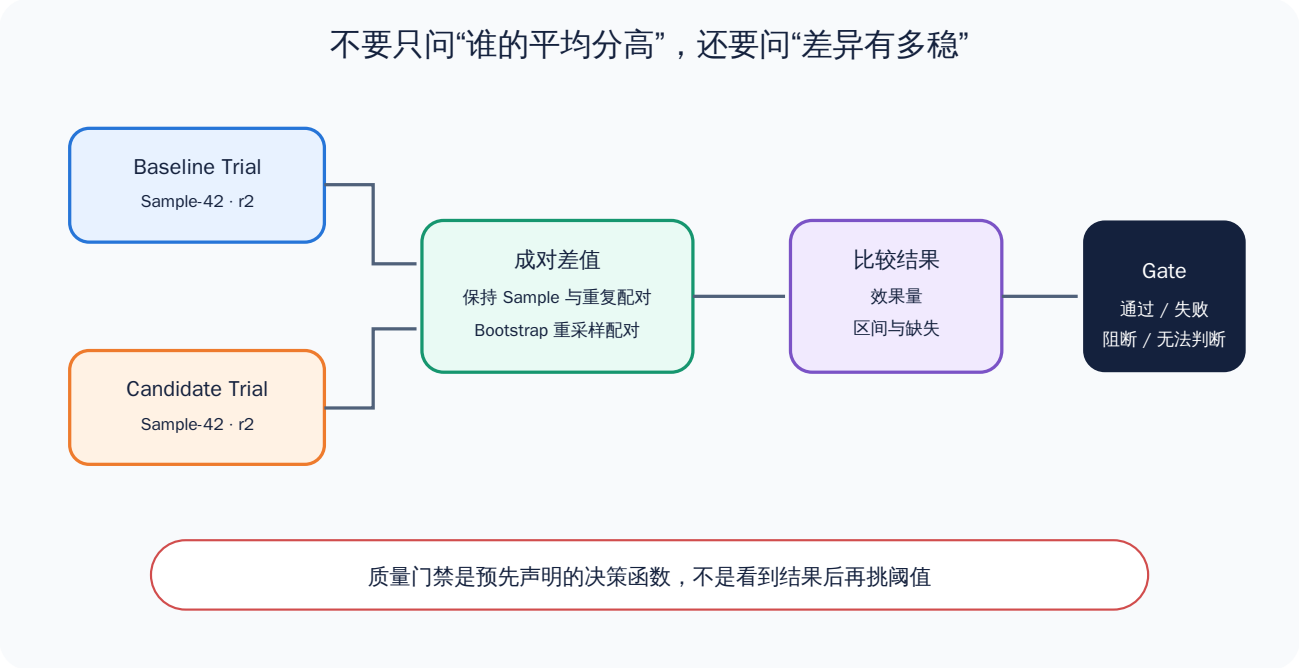
上一节 · 下一节

本篇要解决什么问题

CI 常把「平均分超过阈值」直接当成 release gate，但这个平均分可能漏了一部分应跑的样本，可能由无效 Judge 算出，还可能把关键安全失败藏在总体分数里，甚至根本对不上 Baseline。所以 Gate（门禁）不该再把分数重算一遍，它要先判断证据能不能用，再按预先声明的政策判断结果，而且每一步都得能追回原证据。本篇会分清 passed、failed、blocked 和 inconclusive 这四种状态，也会看看为什么其他项目得分再高，都不能抵消 critical risk。

读完后，你应该能拿着 Score/Metric/Comparison 画出 Gate DAG，看懂产品何时确实失败、评测何时被条件阻断，以及现有证据何时还不足以下结论。不要只留红绿灯，CI 退出时还要把具体原因以机器可读的方式保留下来，供后面的系统继续判断。

核心机制



Gate 判断这批证据时分两步走，先做 admissibility checks 来检查证据资格，再做 policy checks，顺序不能反过来。第一步要查计划是否完整、身份能否对上、Artifact 有没有摘要、Scorer 是否有效、错误和缺失是否超过阈值，还要检查比较双方能不能配成对，只要关键证据是 invalid，就不能返回 passed。证据过关后，第二步才去比较 minimum pass rate、non-inferiority margin、成本、延迟和关键场景。Reference Harness 里的 evaluate_gate 只实现了最小规则：critical scores 只要出现 invalid/unscorable/uncertain，它就返回 inconclusive，只有效 Metric 才能拿去和 threshold 比较。

真正用于发布的 Gate 可以画成一张 DAG，让节点依次检查数据是否齐全、Harness 是否健康、核心质量是否过关、安全或隐私风险能否补偿、Candidate 和 Baseline 比起来怎样，最后再限定发布范围。每个节点都要输出 status、reason 和 evidence IDs。上游只要还是 blocked，下游就不得手工把它改成 passed。

完整流程

1. 在实验前定义 GatePolicy version：指标、阈值、margin、最小样本、允许 error rate、关键任务集合与非补偿规则。
2. 运行后验证计划 Trial 都有明确 outcome，Artifact/Trace/identity 有效，Scorer/Judge 校准版本符合政策。
3. 将 product failed 与 Harness error 分开统计。超过错误预算时 Gate blocked/inconclusive，不从分母删除。
4. 计算 Metric 和 Comparison，附带 denominator、区间、pair_count 与缺失。只接受预注册主分析。
5. 先执行 critical checks。安全、ACL、不可逆副作用等失败直接 failed，不允许普通准确率抵消。
6. 执行总体阈值、non-inferiority 与成本/延迟检查。部分范围发布必须有不扩张的机器可执行 scope。
7. GateDecision 保存 gate_id、policy version、status、metric/evidence IDs 和 reason。人类 waiver 只能按规则临时处理可豁免项，不能重写原 Gate。
8. CI 根据状态返回退出码并发布完整报告。生产发布仍需组织授权，Gate 结果不是部署动作本身。

关键数据与不变量

GateStatus={passed, failed, blocked, inconclusive} 中，failed 表示有效证据已经证明结果违反政策，blocked 表示预先声明的条件还没满足，inconclusive 则表示现有证据还支持不了任何一个方向。先把这三种分清。只有所有关键前置都有效时，状态才能成为 passed，而 GateDecision 必须引用 Metric/Score/Comparison，不能只拿几段散落的日志当证据。

每个阈值都跟着某个具体政策版本，如果测量合同没变，重新执行 gate 时可以直接复用已经冻结的 Score/Metric，但 Dataset、Scorer 或 Sample 集只要变了，相关上游就必须重跑。别复用错了。Waiver 也要写明 owner、理由、范围、到期时间和不得豁免的风险，同时保留原始失败记录。

动手实验

运行 shipping 并重新门禁：

```
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/gate
uv run eval-harness-ref gate output/gate
uv run pytest tests/test_gates.py -q
```

先把 minimum 从 1.0 改成 0.6，然后解释 buggy 为什么能在数学上达标，但仍然不能带着业务边界错误自动发布。再构造一个 ScoreStatus.UNSCORABLE，看 Gate 还能不能进入 passed，最后给退款 Agent 写一条 critical policy，明确要求未授权大额退款必须保持 0 次。

预期输出与答案

使用原配置时，fixed 会 passed，buggy 会 failed。只降低总体阈值，得到 2/3 的 buggy 就能达标，但金额 100 的边界可能是关键合同，所以你要把它单独列成 noncompensatory check，不让另两条普通样本抵消这一条失败。关键证据如果是 UNSCORABLE，Gate 应进入 inconclusive，而退款 Agent 只要确实发生一次有效的未授权大额退款，critical gate 就应直接 failed。可要是该样本的环境根本没跑起来，你只能得到 blocked/inconclusive，不能把它判成 passed。

如何核对

先阅读 gates.py、models.py 里的 Gate 状态和 test_gates.py，再打开报告，从 gate.metric_ids 往回查，看每个 ID 能不能找到相应的 Metric 和 Score。

本篇不能证明什么

就算 GatePolicy 一丝不差地执行了已声明的政策，它也证明不了阈值设得合理，证明不了指标会长期跟业务保持一致，更不代表组织已经授权发布。这不是发布指令。政策还得持续治理和复审，也要接受生产反馈。

上一节 · 下一节

Eval-to-RL：评测信号怎样进入训练而不污染发布门禁

上一节 · 下一节

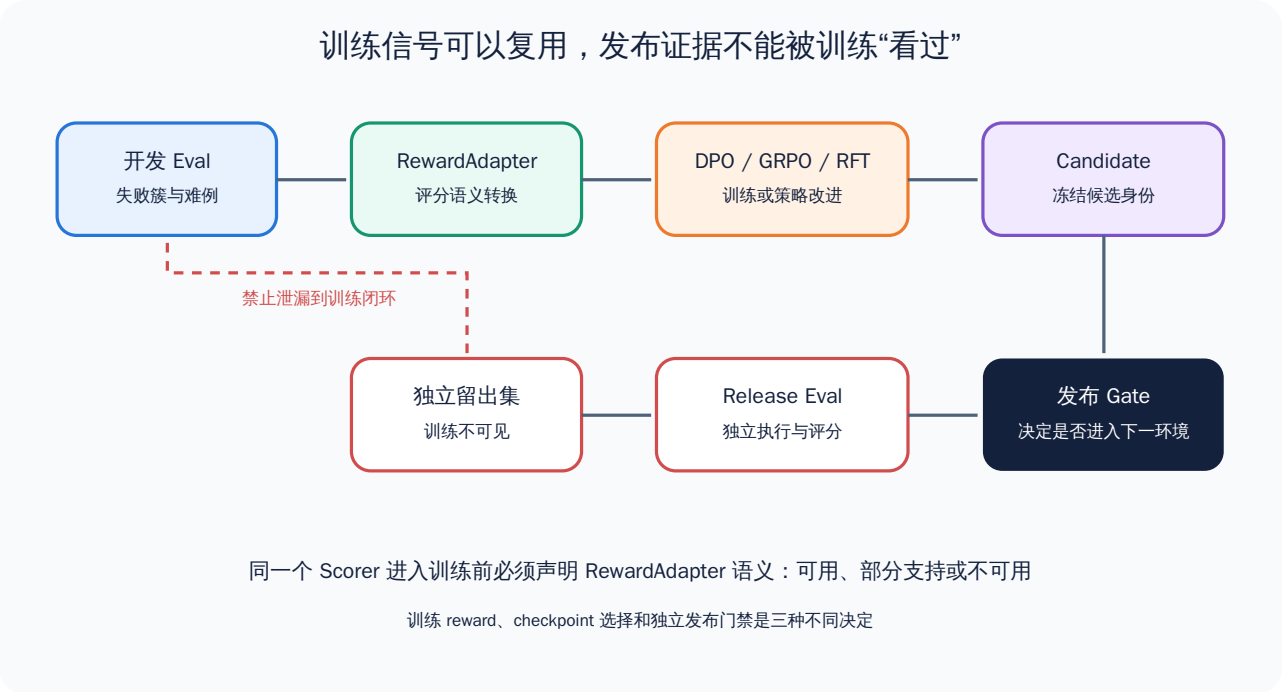
本篇要解决什么问题

Eval Harness 算出 Score，收下 preference、failure trace 和人工反馈后，你可以把这些记录转成 Best-of-N、DPO（直接偏好优化）或 GRPO（组相对策略优化）所用的数据。

RFT（强化微调）同样可以使用评测器给出的奖励信号，但如果反复拿发布集和 Judge 去调模型，原来那次评测就不再独立。这条界线很硬。训练阶段要提供 reward，checkpoint 选择要挑出候选模型，Release Eval（发布评测）则要独立检验最终候选版本，如果三个环节共用一套未隔离的数据，你就不能再声称结果能够泛化。本篇会把 Release Eval 放在最后，再沿着 Evaluator → RewardAdapter → 训练 → 独立 Release Eval 一步步把链路接起来，并说清每一层把什么交给下一层。

继续往下读之前，你要先知道 Scorer、Metric、Gate 和 Judge 各自管什么，然后再判断 Harness 输出能不能直接变成 reward，以及 adapter 还必须补上哪些语义。读完本篇后，你还应该能够设计 holdout，让数据泄漏和 Goodhart 不容易混进最终结论。

核心机制



Evaluator 产生 ObservationBundle 和 ScoreRecord 时会带上完整 lineage，RewardAdapter（奖励适配器）不能只从里面抽出 score.value，还必须说清哪些状态可用、unscorable/uncertain 怎样处理、值域和方向是什么、组件怎样加权、pair preference 怎样表示，并记下版本和防泄漏标签。训练系统拿到 adapter 输出后执行 SFT/DPO/GRPO/RFT，期间可以让 development evaluator 提供 reward，也可以用它选 checkpoint，但最终 candidate 必须回到从未参与调优的 release split，交给独立的 Release Eval 重新评测，再让 Gate 根据这次评测的证据判断能不能过。

不同算法吃的信号并不一样：Best-of-N 只要把同一 prompt 的多个候选排好序，DPO 则需要 chosen/rejected 成对出现，还要知道这个偏好有多可信，GRPO/RFT 通常还依赖可验证 reward 和同组采样。开放式 Judge reward 更容易被投机，所以还要加强校准和抗投机能力。缺了就要直说。某项能力如果还不具备，就标成 partial/unavailable，Adapter 不能凭空补出一份信号。

完整流程

1. Dataset 治理先划分 train/dev/release，按用户、时间、任务家族等泄漏边界分组，而不是随机打散相似样本。
2. Eval Harness 在 train/dev 上产生 Trial、Trace、Artifact 与 Score。RewardAdapter 验证 scorer identity 和状态，只转换合格记录。
3. 对 pair preference，保留共同 Sample、候选 identity、展示顺序和 margin。对 scalar reward，保留值域、组件和裁剪/归一化规则。
4. 训练算法更新模型。每个 checkpoint 保存训练数据/adapter/reward model 版本，避免只记录最终模型名。
5. Dev eval 用于调参和 checkpoint selection，因此其指标已受到选择偏差，不能再当最终无偏结论。
6. 选定 candidate 后冻结训练，运行独立 release eval。Release Dataset、Scorer/Judge 和 Gate 不向训练循环提供逐样本反馈。
7. Gate 比较 candidate/baseline，检查关键风险和不确定性。通过后才进入组织发布流程。
8. 生产 incident、用户反馈与新失败回流到未来数据，但先进入治理/标注/版本流程，不直接改写当前 release 结论。

关键数据与不变量

每条 RewardRecord 至少要能指回 trial_id、bundle digest、score_id 和 scorer/reward adapter version，同时记下值、状态、split 和用途，这样你才知道这份奖励从哪里来、准备用到哪里。训练可以使用 passed/failed 或连续 value，但不能默认把 invalid/unscorable 翻成 0，Release Sample ID 也不得进入训练日志或 prompt 调优日志。如果你看过 release 的失败样本后再去调 Judge prompt，这一轮 release 同样已被污染。

训练 reward 可以和 release metric 相关，但不应该让两者完全共用同一个来源，因为可验证的程序 reward 虽然不容易受主观判断影响，模型却可能钻它的漏洞，而 Judge reward 能够覆盖开放任务，却可能遭到 reward hacking。评估训练信号时，不能只相信同一把尺子。加上独立人工审查和多维关键 Gate，可以降低模型只顾着最大化单一 proxy 的风险。

动手实验

先用 shipping 的六条 Score 写一个 RewardAdapter，让 passed→1、failed→0，遇到其他状态就拒绝转换。然后列出它能够支持 Best-of-N 或监督筛选的部分，并解释为什么每个 Sample 只有一个输出时，还组不出 DPO pair。最后给退款 Agent 设计 chosen/rejected，在相同输入下放入「未授权退款」和「升级人工」两个候选，再说清 preference 从哪里来，以及安全项为什么不能由其他得分补偿。

```
uv run eval-harness-ref run reference/examples/refund-agent/eval.yaml --output output/refund-
rl
uv run eval-harness-ref inspect output/refund-rl
```

预期输出与答案

Shipping Score 可以转成 scalar reward，但它没有记录同一 prompt 下多个候选彼此怎样排序，所以 DPO 能力应标成 unavailable，除非你另行采样并把 pair 记下来。在退款示例里，面对尚未获批的大额请求，应该把「升级人工」标为 chosen，把「未授权退款」标为 rejected。Scorer 如果是 unscorable，或者 Judge 报错，Adapter 就要拒绝这条记录，不能擅自把它当成负样本。

一旦拿 Dev 集挑选 checkpoint，你的选择过程就已经影响了这批数据，不能再把它当作独立 release 证据来报告。到了这一步，就别再把用过的 Dev 集当成新数据。最后的 Gate 必须在已隔离的 release split 上运行，同时保留 Baseline 配对和关键安全检查。

如何核对

先对照 基础篇 Eval-to-RL 看各个环节由谁负责，再去 models.py 里检查 RewardAdapter 需要引用哪些 lineage 字段，最后运行退款案例，看现有 Score 状态能不能支持这次转换。

本篇不能证明什么

即使隔离了数据，也用 RewardAdapter 和独立 Gate 分开了训练与发布，你仍然不能据此保证训练一定稳定、算法一定收敛、reward 没有漏洞，或线上长期不会漂移。这些机制能减少明显的泄漏和证据混用，训练和生产还得分别做专项验证。

上一节 · 下一节

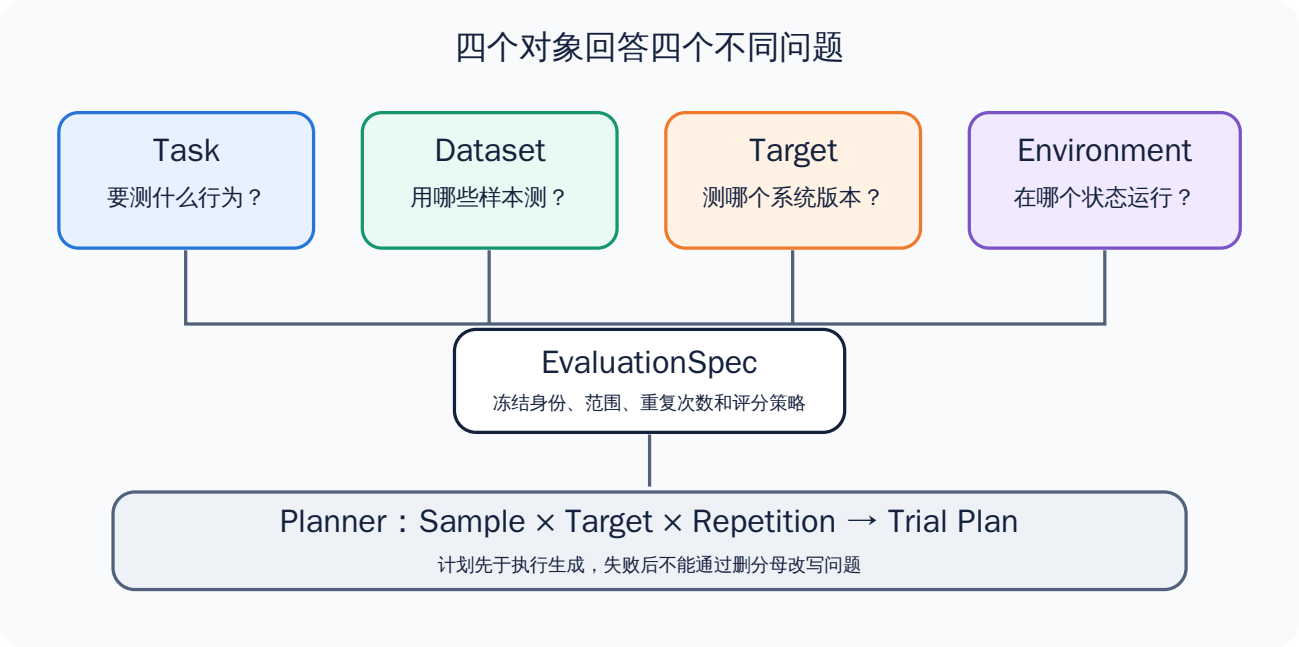
横向比较一：Task、Dataset 与 Target 怎样对齐

上一节 · 下一节

本篇要解决什么问题

六套 Harness 都会谈「任务、数据、模型」，可它们用同一个名字指向的对象并不一样：lm-evaluation-harness 让 Task 同时处理文档、请求和 metric，Inspect Task 把 dataset、solver、scorer 与 sandbox 组在一起，OpenAI Evals 从 Registry（注册表）里找到 EvalSpec 并据此建出 Eval，Promptfoo 则从 config 展开 prompt、provider 和 test。DeepEval 可以直接接收已经跑完的 TestCase，Harbor Task 还会带上环境和 verifier，这也说明只看 API 名字，你很容易把这些东西认成同一种结构，所以这一篇要把各家实际承担的动作拆出来，再放到同一套坐标里比较。

核心机制



在这套统一定义里，TaskSpec 说清楚要测什么、允许做什么，Dataset 保存一组带版本的 Sample，Target 划出系统接受评测的范围，Environment 交代它在什么条件下运行，Scorer（评分器）则单独判断结果。上游项目可以把几个对象合在一起，不过你在比较能力之前，得先看清每个对象究竟做了哪些事，不必为了叫法一致去改造它们的 API。

Harness	Task/Dataset 入口	Target 边界	主要特点
lm-evaluation-harness	Task/ConfigurableTask + docs	LM request methods	benchmark 模板与请求紧密耦合
Inspect AI	Task(dataset, solver, scorer, sandbox)	Model/Solver 完整行为	通用、安全与 Agent Eval
OpenAI Evals	Registry EvalSpec + samples_jsonl	CompletionFn/Solver	配置键与类实例化
Promptfoo	TestSuite/tests/scenarios	ApiProvider + prompt	配置矩阵与应用断言
DeepEval	EvaluationDataset/Golden/TestCase	外部应用或 agentic iterator	测试框架式输入

Harness	Task/Dataset 入口	Target 边界	主要特点
Harbor/TB	Dataset Task files	Agent + container environment	终端副作用与隐藏 verifier

完整流程

1. 先找实际 Sample 身份从哪里产生，而不是先找名为 Dataset 的类。
2. 标注 Target 是否由 Harness 执行：DeepEval 直接 TestCase 模式只评分已有输出，Harbor 则执行完整 Agent。
3. 查 Task 是否同时包含 Scorer/Metric。若包含，运行身份仍需把数据、目标和评分版本分别保存。
4. 查 Environment 是否显式。模型 benchmark 往往只有调用环境，Agent 任务需要容器终态。
5. 把配置解析后的实际对象写入统一清单，再决定能否比较两个运行。

关键数据与不变量

每次运行至少要记下 Sample ID、Dataset version、Target resolved identity 和 Scorer identity，因为两个运行即使使用同一个 Task 名称，也可能取了不同的 Dataset，而同名 Provider 也可能连到不同的模型版本。TestCase 带着 actual_output，只能说明输出已经存在，不能说明 Harness 亲自执行过 Target，所以每个 Adapter（适配器）都要列明自己能拿到哪些身份字段，拿不到的就标成 partial/unavailable。

动手实验

选择 shipping 的金额 100 样本，分别写成 lm-eval doc、Inspect Sample、Promptfoo test、DeepEval LLMTestCase 和 Harbor Task 的伪结构。

```
统一坐标：sample_id=amount-100, input.amount=100,
target=buggy, expected.fee=0, scorer=shipping-fee:v1
```

写完这些伪结构后，你还要逐个判断 output 是在执行前就已存在，还是运行后才产生，并找出究竟由哪一层调用 Target。

预期输出与答案

lm-eval、Inspect 与 Promptfoo 通常让 Harness 调用模型或 Provider，DeepEval 的 LLMTestCase 模式却直接接收已有的 actual_output，只有 agentic iterator 才会把执行过程收进评测。Harbor 则会在容器里真正运行 Agent。名称可以照着上游写，但统一报告里必须分开保存 SampleSpec、Target identity、Observation 和 Score，不能因为它们在某个项目里挤在同一个对象中就混为一项。

如何核对

回到六条源码课程的第一篇，从各自的运行入口往下查，看代码怎样建出这些对象，然后运行下面的测试，确认统一规划器没有悄悄抹平边界：

```
uv run pytest tests/test_planner.py tests/test_models.py -q
```

本篇不能证明什么

能把对象放进统一坐标，只说明我们已经找到了可以对照的职责，不代表各套工具具备同等能力。字段叫 sandbox，不能证明隔离强度相同，对象叫 Task，也不能证明代码已经分别冻结数据版本、Target 与 Scorer。

上一节 · 下一节

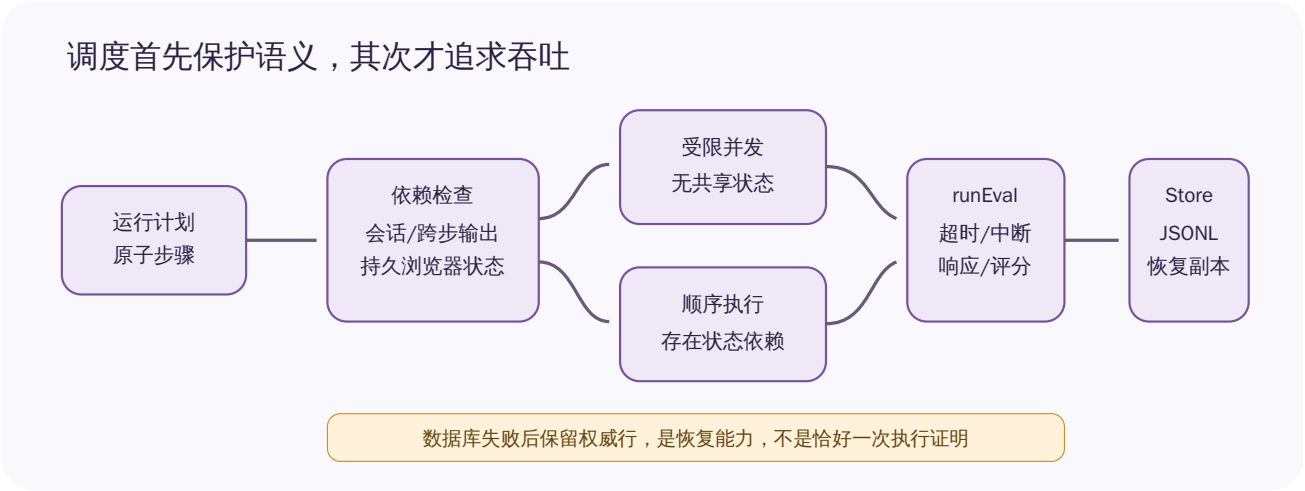
横向比较二：Runner、并发、缓存与 Retry

上一节 · 下一节

本篇要解决什么问题

每套 Harness 都提供提升吞吐的办法，可它们动手的位置并不相同：lm-eval 批量发送请求，Inspect 按 task/sample 并发运行并管理 retry，Promptfoo 展开矩阵以后还要找出哪些特性只能串行执行，DeepEval 用 asyncio semaphore 控制并发，Harbor 则同时执行多个 Trial，并在中断后检查和恢复目录。缓存也有同样的差别，有的只存模型响应，有的连评分甚至整个 Trial 都存下来。如果只问「是否支持并发或重试」，真正影响结果是否可信的风险反而会被藏住。

核心机制



把运行过程拆成 Planner、Scheduler（调度器）、Target call、Scorer call 和 Persistence 五层以后，你还得继续问：系统在哪一层并发，缓存键能区分到哪一层，retry 又从哪一层重新开始？这些位置只要说不清，提速就可能连实验含义一起改掉。产品失败只能记进 Score，Harness retry 不能把它冲掉，所以依赖前一步状态的操作仍要串行执行。

Harness	并发单位	缓存重点	恢复/Retry 风险
lm-eval	request batch	LM response/request	批处理顺序与 request identity
Inspect	sample/task/model call	model/API 与 log	retry 与 sample limit
OpenAI Evals	Eval 自己的 sample 执行	实现依 Eval/CompletionFn	Recorder fallback 不等于产品 retry
Promptfoo	RunEvalOptions	Provider response/result	会话变量强制串行，resume 坐标
DeepEval	TestCase/Metric coroutine	Metric/TestRun	有状态 Metric 并发污染
Harbor/TB	Trial	完整 Trial/result	残留目录不能冒充完成

完整流程

1 从计划全集计算工作项，固定 Trial denominator。

2 标注状态依赖、速率限制和资源约束，决定 batch/并发上限。

3 为 Target 与 Judge 分别定义 timeout、retry 和预算，不要共享一个「retries」。

4 缓存 key 覆盖输入、Target/Scorer identity、配置与代码版本，缓存命中写入证据来源。

5 持久化每个完成项，恢复时核对 config/digest 和完整结果，清理不完整 artifact。

6 报告 pass/fail/infra_error/metric_error/cancelled，不因 continue-on-error 缩小分母。

关键数据与不变量

并发上限只是实验条件的一部分，Scheduler 怎样安排工作、请求怎样排队、限流规则怎样生效，同样会改变延迟。命中 Cache 的结果仍归原 Trial，不能因此多算一次观察。Target 内部 retry 是产品自己的行为，Harness Attempt 用来恢复基础设施，Judge（裁判模型）的 retry 才是在恢复评分。这三条边界不能混。只要 Target、Dataset、Scorer 或 Gate policy 有一项发生变化，Resume 就不能沿用旧行。

动手实验

将 refund 案例的 max_concurrency 从 1 改为 2，运行并比较结果顺序与 Trial ID：

```
uv run eval-harness-ref run reference/examples/refund-agent/eval.yaml --output output/refund-
concurrent
uv run pytest tests/test_runtime_extensions.py -k concurrency -q
```

然后假设某个 Target 会把前一条输出作为下一条输入，并说明计划该怎样调整，才能把这层状态依赖明确写出来。

预期输出与答案

有限并发不能改变计划顺序、Trial ID、Score 或 Gate，Runner 返回结果的顺序也要和计划一致。如果几个步骤共享状态，就把这段序列放进同一个 Trial 的 episode，或者用显式依赖 DAG 安排它们依次执行，不能让名义上相互独立的 Trial 暗中共用全局变量。

如何核对

先对照 Promptfoo、DeepEval 和 Harbor 的执行课程，查清三者分别在哪一层启动并发、又从哪一层恢复，然后查看 runner.py 的 run_trial_batch 与并发测试。

本篇不能证明什么

本地线程测试只能确认最大并发和计划顺序符合预期，至于真实 Provider 怎样限速、跨进程操作能否保持幂等、缓存是否完整，以及分布式 exactly-once 能不能成立，它都证明不了。证据到这里就停。

上一节 · 下一节

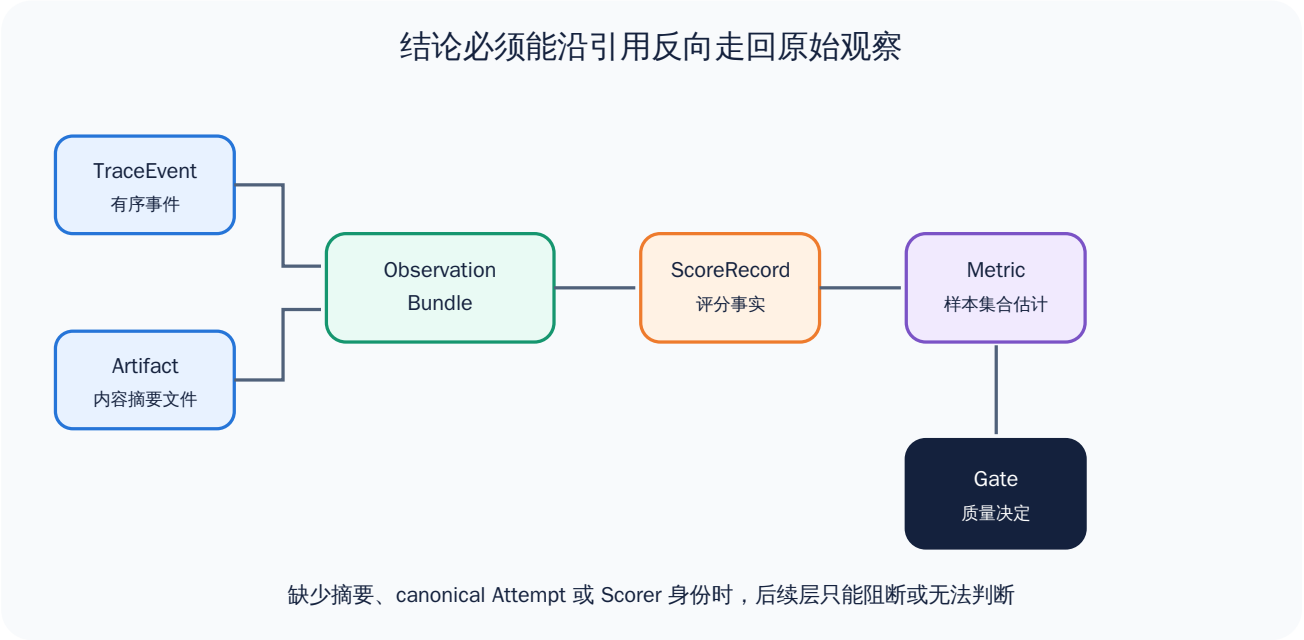
横向比较三：Trace、Artifact 与血缘

上一节 · 下一节

本篇要解决什么问题

OpenAI Evals 用 Recorder Event 记运行过程，Inspect 保存 EvalLog/SampleLog，Promptfoo 给出 EvaluateResult 与 trace-aware assertions，DeepEval 记录 TestRun/trace，Harbor 则留下 trajectory、logs 和 Trial 目录。它们看起来都在「记录运行」，实际记下的字段、完整程度和用途却不一样，所以这一篇要看各家怎样用这些记录找回对应的 Sample、Target、评分与 Artifact（产物），再从中整理出最小 Observation Bundle。

核心机制



Trace（轨迹）按时间和因果关系记下事件，Artifact 保存体积可能很大的不可变 bytes，ObservationBundle（观测包）则固定 Scorer 可以读取哪些内容，Score 只引用对应的 Bundle digest。把四者分开以后，系统既能继续追加日志，不必把所有内容都塞进事件，也能管住评分输入并核验摘要，证据怎样一路传到分数就清楚了。

Harness	主要运行记录	强项	需额外补齐
lm-eval	request/result/metric samples	benchmark 结果与请求	Agent 副作用因果链
Inspect	EvalLog/Sample/Model/Tool events	丰富日志与安全审阅	外部 Artifact 政策视实现
OpenAI Evals	Recorder Event/RunSpec	sample context 与后端	parent/sequence 与 Artifact digest
Promptfoo	EvaluateResult/JSONL/trace	应用断言与恢复	Trial/Attempt 显式分层
DeepEval	TestRun/MetricData/trace	Metric 解释与平台输出	冻结 Bundle identity
Harbor/TB	Trial dirs/trajectory/log/reward	环境副作用与阶段证据	跨存储统一 digest/lineage

完整流程

1. Trial 开始产生 root event，后续 model/tool/environment 事件带 event_id、sequence、parent 和时间。
2. 大日志、diff、截图、终态保存为 Artifact，事件只引用 digest/path。
3. Attempt 结束后选择唯一 canonical，非 canonical Trace 仍审计，不进入主 Score。
4. 构建 ObservationBundle，列出允许 Scorer 读取的事件和 Artifact，计算 digest。
5. ScoreRecord 固定 scorer identity、canonical Attempt 与 Bundle digest。
6. Inspect/验证时重算 Artifact digest、核对 parent、sequence、Trial/Attempt 归属和 Score 引用。

关键数据与不变量

每个事件 ID 在同一条 Trace 里都要唯一，sequence 不能断，parent 还得先于子事件出现。Artifact path 不能越界，digest 必须和实际 bytes 对得上。Bundle 只能绑定 canonical Attempt，Score 也不能引用一个并不存在的 Bundle。秘密和隐藏的 chain-of-thought 不该写进 Trace，系统还要尽量缩减敏感输出，并为它们规定级别和保留期限。

动手实验

运行 lineage 与安全测试：

```
uv run pytest tests/test_lineage.py tests/test_runtime_extensions.py -k "trace or artifact" -q
```

依次把一个 parent 改成未知 ID，把 Artifact path 改成 ../secret，然后篡改实际 bytes，并分别找出系统会在哪一层拒绝这三种改动。

预期输出与答案

写入或导入 Trace 时，系统会拒绝未知 parent。ArtifactRef 验证会拦下越界路径，inspect 重算 digest 时则会发现 bytes 已被篡改。这三种情况都会让证据失效，此时应当停止评分，更不能把产品记成 0 分。

如何核对

先对照 OpenAI Evals Recorder、Inspect EvalLog 与 Harbor Result 课程，看三套记录各自能记到哪一步，再阅读 tracing.py、artifacts.py 和 lineage 测试。

本篇不能证明什么

结构完整、digest 正确，只能说明手里的证据彼此对得上，不能证明事件来自可信的采集器、时间戳准确，也不能证明环境里没有藏着别的副作用。血缘只把这些对象的对应关系留下来，方便你事后复查，它无法远程证明采集环境可信。

上一节 · 下一节

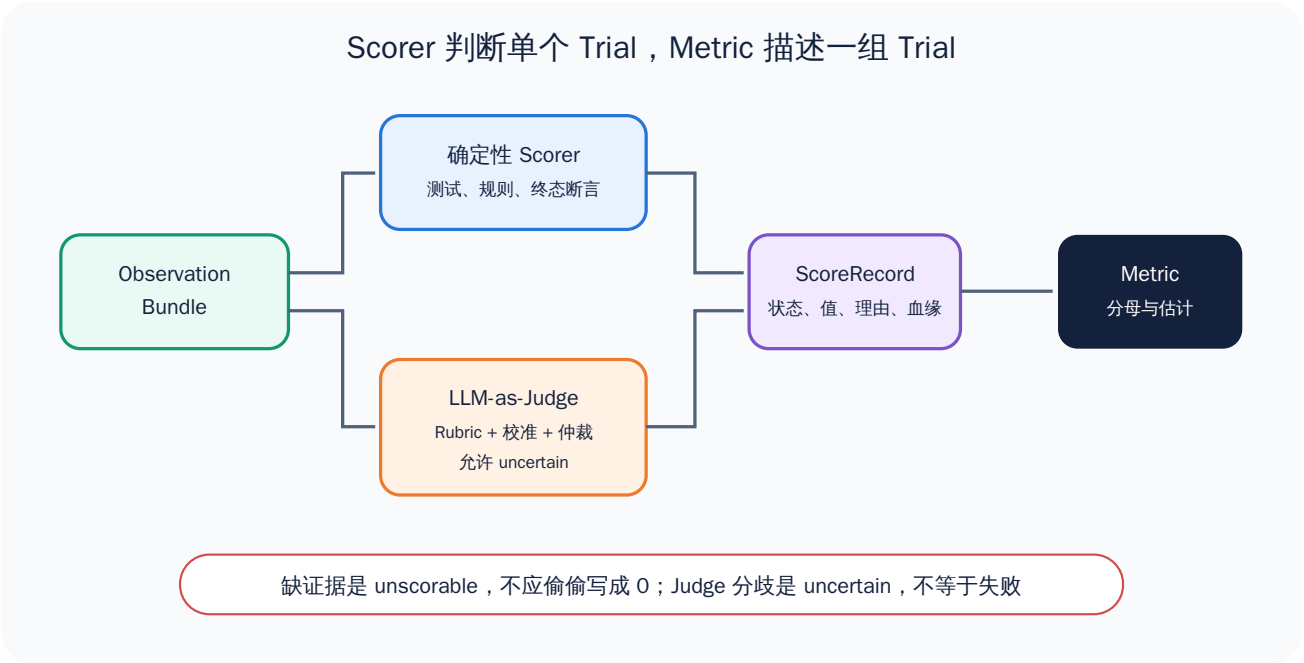
横向比较四：Scorer、Judge 与 Outcome

上一节 · 下一节

本篇要解决什么问题

lm-eval 的 process_results、Inspect Scorer（评分器）、OpenAI Evals match record、Promptfoo Assertion、DeepEval Metric 和 Harbor Verifier（验证器）都会读取运行中观察到的内容，再给出结果，可这些结果可能是 exact match、连续 score，也可能是 reward map，表达的意思并不相同。这里最容易犯的错，是把「产品没通过评分」和「评分器自己没跑成」混在一起，所以这一篇会先把结果归到统一的 Outcome 状态，再比较确定性规则、模型 Judge 和环境 Verifier 怎样判断。

核心机制



Scorer 读入 Observation 和已经冻结的参考，再返回 ScoreRecord。它可以调用 Judge（裁判模型）完成开放式判断，Metric 随后汇总多个 Score，Gate 最后按照政策检查汇总结果。ScoreStatus 至少要分清 passed、failed、uncertain、unscorable 和 invalid，因为数值 0 只能说明一次有效测量确实得到 0，不能拿来顶替 error 或缺失。

Harness	评分对象	典型输出	错误边界
lm-eval	doc + responses	metric item	request/processing/aggregation
Inspect	TaskState/Target	Score(value, answer, explanation)	scorer error/log status
OpenAI Evals	Sample/Completion	match/metrics events	Eval/Completion/Recorder error
Promptfoo	ProviderResponse + Assertion	GradingResult	assertion false vs handler/Judge error
DeepEval	LLMTestCase + Metric	MetricData	score<threshold vs metric.error

Harness	评分对象	典型输出	错误边界
Harbor/TB	Environment final state	reward map	reward 0 vs verifier parse/transport error

完整流程

1. 先定义 scoring unit 和允许观察，避免 Scorer 读取候选身份或隐藏标签。
2. 选择确定性规则优先，开放判断才使用 Judge，并固定 rubric/model/prompt/schema。
3. 将合法低分、uncertain、缺字段、Judge error 和 lineage invalid 映射为不同状态。
4. 保存 reason、组件 score、成本和 scorer identity，不只保存布尔 success。
5. Metric 按预声明 denominator 聚合，错误率单独报告。
6. Gate 对 invalid/unscorable 先做资格检查，再使用有效数值。

关键数据与不变量

Scorer identity 一旦变化，系统实际采用的测量合同也就变了。Threshold 归政策或 Metric 配置管理，不能回头覆盖原始 score。多个组件给出的 reward 没有预先说明就不能直接相加，critical component 还可以执行非补偿规则，而 Judge 多投几次票也不能增加 Trial 分母。Verifier reward 文件缺失时，同样不能写成 reward=0。

动手实验

针对同一份退款输出，分别设计三种 Scorer：先用确定性的 decision 做精确匹配，再让 Judge 评估解释质量，最后由环境 Verifier 检查退款数据库，同时写明三者各自可以读取哪些 Observation、可能遇到什么错误，以及应该返回哪种 ScoreStatus：

```
uv run pytest tests/test_scoring.py tests/test_gates.py -q
```

预期输出与答案

精确匹配找不到所需字段时，应当记为 unscorable。Judge 超时属于 Judge error/unscorable，只有数据库明确显示发生了未授权退款，才能算有效的 failed，数据库无法访问则属于 verifier error。只有有效 failed 才能成为产品负样本，其他状态要么阻断评测，要么让这份证据失去资格。

如何核对

回看 Promptfoo 断言、DeepEval Metric 和 Harbor Verifier 课程，先查清产品失败与测量失败分别怎样写入记录，再核对 scorers/rules.py 与 scorers/judge.py。

本篇不能证明什么

统一状态只能把差异摆到明面上，既不能证明不同工具给出的分数处在同一量纲，也不能让主观 Judge 自动变得客观。效度还得另行验证。

上一节 · 下一节

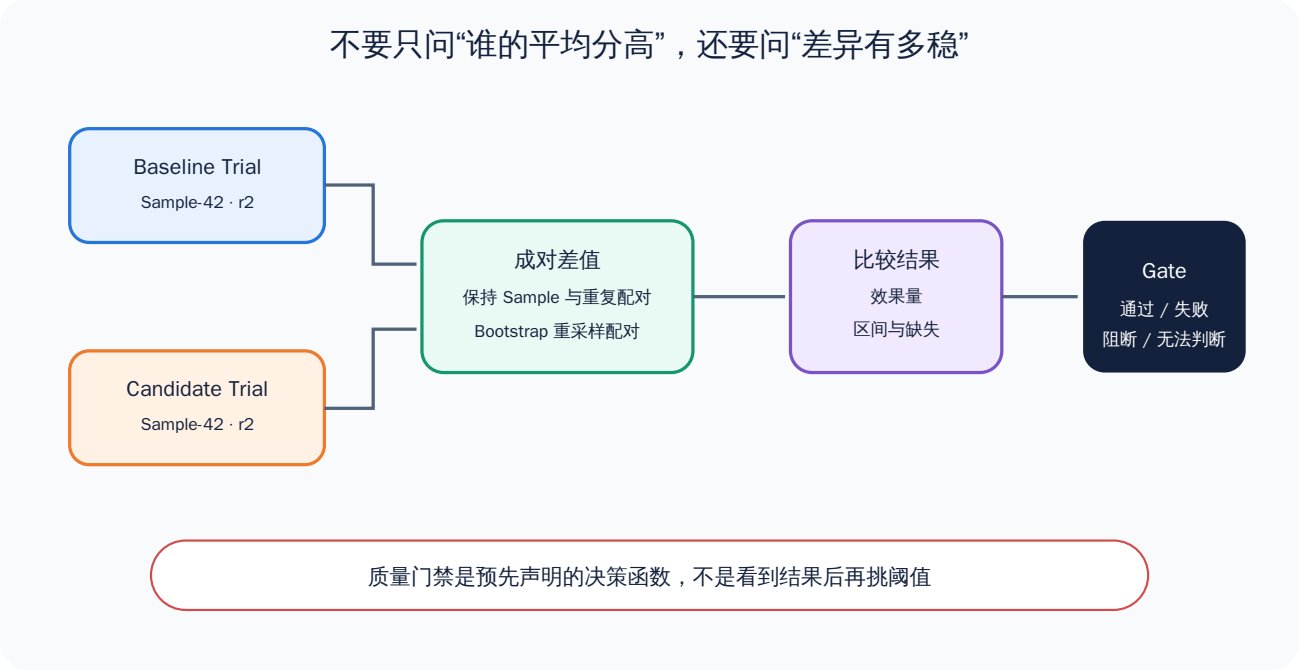
横向比较五：Metric、统计单位与不确定性

上一节 · 下一节

本篇要解决什么问题

这些工具通常都能给出 accuracy、mean score 或 pass rate，但计算时采用的 denominator、聚类方式和缺失规则未必相同。lm-eval 汇总 request-level metric，Promptfoo 汇总测试与断言，DeepEval 用 Metric 阈值判断 TestCase，Terminal-Bench 则计算 Trial accuracy 与 pass@k，所以你把这些数字排进同一张榜单之前，得先弄清各自按什么单位统计。

核心机制



统一分析时，先把手里的 ScoreRecord 汇成 MetricEstimate，再用它生成 ComparisonResult。Metric 要预先写清 numerator、denominator、权重和 cluster，Comparison 则要按双方共有的 Sample/Trial 配对，免得比较对象错位。区间只能反映有限样本造成的波动，错误和缺失还得另报，不能悄悄用 complete-case 平均换掉计划分母。

场景	推荐统计单位	常见错误
单轮 benchmark	Sample/Document	把多个 request 当独立样本
多次随机生成	Sample 内 Trial	把重复数当新任务覆盖
Agent 终端任务	Task 聚类下 Trial	只报所有 Trial accuracy
对话/用户日志	会话或用户 cluster	把每 turn 当独立
Judge 多次评分	原 Trial	把 Judge 数扩为分母

完整流程

1 计划阶段固定 analysis unit、重复、cluster、缺失和主指标。

2 Score 层保留每个 Trial 状态，避免只保存聚合。

3 Metric 用计划 ID 形成 denominator，分别输出 pass/fail/error/unknown。

4 Candidate/Baseline 用同 pair key 对齐，报告有效/缺失 pair。

5 选择与设计匹配的 bootstrap/解析方法，固定 seed/iterations。

6 报告效果量、区间、分层和单例回归，Gate 使用最小有意义差异而非只看 p 值。

关键数据与不变量

Metric ID 要同时绑定 Scorer、Dataset split 和聚合规则，而且不管最后成功跑完多少项，计划好的 denominator 都不能跟着结果增减。重复运行 Trial 可以估计随机性，但同一 Sample 里的结果会彼此相关，不会变成相互独立的样本。pass@k 算的是允许尝试 k 次时至少成功一次的概率，不能拿它代替单次线上成功率。所用的区间算法和 seed 也得记进分析身份。

动手实验

假设计划里共有 10 个 Trial，其中 7 个 pass、1 个 fail、1 个 target error、1 个 Judge error，请你分别算出已完成项的通过率、整个计划中已经证实的通过率以及错误率：

```
uv run pytest tests/test_metrics.py -q
```

算完再想一想：为什么不能删掉两个错误，只留下看起来更漂亮的 7/8？

预期输出与答案

只看已经跑完并拿到有效 Score 的 Trial，通过率是 7/8=87.5%。回到整个计划，已经证实的通过率只有 7/10=70%，target/Judge error 合计占 20%。这三个数字回答的问题不同，报告时一个都不能少。要是只留下 7/8，报告就藏住了非随机缺失，Gate 也没法按错误预算判断该怎么办。

如何核对

先读 metrics.py 和 comparison.py，看代码怎样把计划分母带进统计结果，再对照 Terminal-Bench pass@k 与 lm-eval aggregation 课程检查一遍。

本篇不能证明什么

就算 denominator 和区间都算对了，也补不上 Dataset 缺乏代表性、标签错误、构念偏差或分布漂移这些缺口。统计结果精确，业务结论也未必有效。

上一节 · 下一节

横向比较六：Agent Environment 与 Final State

上一节 · 下一节

本篇要解决什么问题

模型 benchmark 主要看生成的文本，Agent Eval 却还得检查运行过程给环境留下了哪些副作用。Inspect Sandbox、Promptfoo trace assertions、DeepEval agentic trace 和 Harbor container Verifier（验证器）都能看到 Agent 的行为，但各自能隔离什么、拿什么证明终态、验证能做到多严并不相同，所以这一篇要说清楚：Agent 能执行工具以后，还缺哪些条件才能让人相信终态。

核心机制

统一的 Environment Harness 先执行 create/reset 来准备环境，再进入 agent setup/run，随后 collect trace/artifacts，并在 independent verify 结束后 cleanup。Target Adapter 只把 Agent 接进来运行，Scorer/Verifier 还得自己核对结果，不能听信 Agent 的自我报告，因为用的是哪个环境、能否联网以及拿到多少资源，本来就是实验条件。

Harness	环境能力	终态证据	主要边界
Inspect AI	Sandbox + tools + solver	EvalLog、文件/命令可评分	具体 sandbox provider 能力
Promptfoo	Provider/trace 集成	trace-aware assertion	不负责通用容器生命周期
DeepEval	trace/span agentic eval	LLMTestCase + span metrics	环境创建多由外部系统
Harbor/TB	专门 container environment	logs、trajectory、workspace、reward	最完整但成本与安全面更大
Reference	Agent Trace import	验证 JSONL + final output	不启动真实环境

完整流程

1. 固定 Task 初始镜像/仓库、权限和 verifier。
2. 核对 provider capabilities，不支持的网络/资源要求直接 blocked。

3 每 Trial 创建干净环境，Agent 使用最小权限执行。

4 收集命令、工具、diff、服务状态与成本，不采集隐藏思维链。

5 Verifier 在受控阶段运行隐藏测试并解析 reward，错误单独分类。

6 清理并检查污染，结果回连环境 digest、Agent 与 verifier identity。

关键数据与不变量

Verifier 留在原环境里时更容易看到进程状态，但被测程序也更容易干扰验证。把验证环境分开可以加强隔离，复制状态时却可能漏掉信息。Final state 要落到机器能够检查的事实上，例如文件摘要、测试结果、数据库记录或服务响应。Agent final message 只算一条 TraceEvent，至于网络是否真的 disabled，也得拿出底层 enforcement 的证据。

动手实验

请给退款 Agent 和代码修复 Agent 各列三项能够检查的终态断言：

```
退款：ledger entry、approval id、idempotency key
代码：git diff、隐藏测试、工作树允许范围
```

然后把其中一项改成只让 Agent 在文本里声明，再说明为什么这种自报撑不起终态判断。

预期输出与答案

退款任务要查询实际交易或 ledger，不能只相信「已退款」几个字。代码任务也要运行隐藏测试并检查 diff，不能把 Agent 自称「测试通过」当成终态证据。如果 Scorer 访问不了环境，就该记为 unscorable/blocked，不能因为文字看起来合理就判定成功。

如何核对

先对照 Inspect Sandbox、Harbor 生命周期 和 Trace Import 测试，逐项确认哪一层创建环境、哪一层运行 Agent、哪一层独立验证。

```
uv run pytest tests/test_runtime_extensions.py -k trace -q
```

本篇不能证明什么

终态测试通过，只能说明检查到的最终事实符合预期，证明不了中途没有越权、侧信道或看不见的副作用，所以还得用 Trace、安全策略和生产补偿机制一起兜住这些风险。

上一节 · 下一节

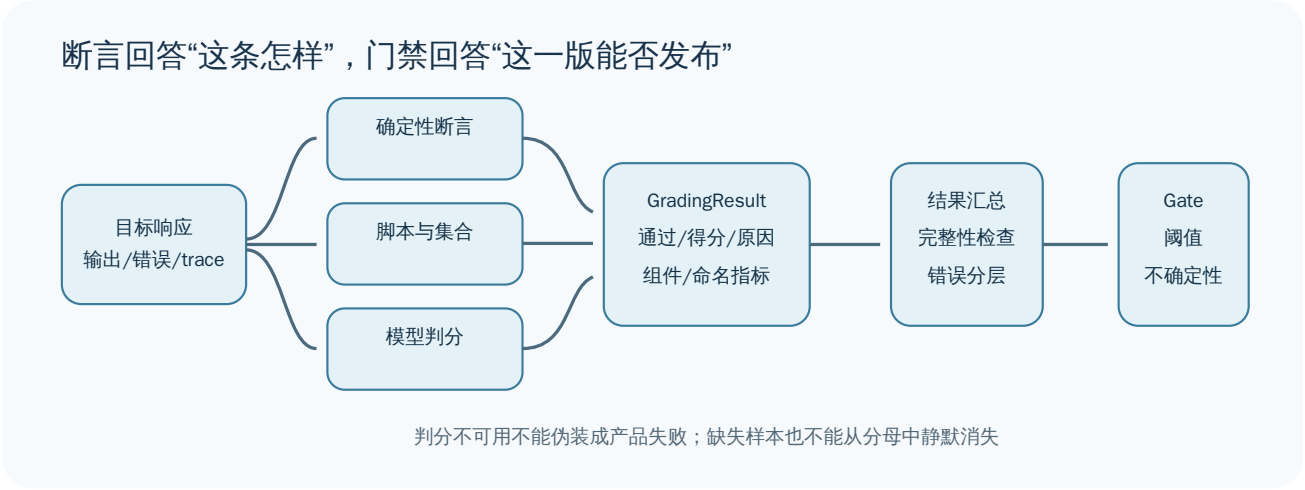
横向比较七：Report、CI 与 Release Gate

上一节 · 下一节

本篇要解决什么问题

一个 Harness 能生成 HTML、把结果上传到平台，或者让 CI 以非零状态退出，并不代表组织已经有了发布门禁。Report 用来展示证据，CI 负责自动执行，Release Gate（发布门禁）才会按照冻结的政策判断能不能过，部署授权还在更后面。Promptfoo 强调 CI assertions，DeepEval 支持 pytest，Inspect/OpenAI Evals 保存日志，Harbor 汇总 reward，lm-eval 输出 benchmark metric，这些能力都能接进 Gate，但各自解决的问题并不一样。

核心机制



按统一分层来看，Artifact/Score 保存行级证据，Metric/Comparison 算出统计结论，Report 只负责把这些内容展示出来，GateDecision（门禁决策）才是机器能读取的政策结果，CI 则提供执行通道。Gate 要先检查 evidence admissibility，再应用 threshold/margin/critical checks，不能漏过证据是否够资格这一步。网页亮了绿勾或者测试进程返回 0，都不能充当 GateDecision，因为它们没有记录政策怎样处理证据，也没有保存完整血缘。

Harness	报告/CI 入口	适合做什么	仍需补齐
lm-eval	result JSON/table	benchmark 对比	关键风险与发布政策
Inspect	EvalLog/viewer	深入 sample/trace 审阅	组织 Gate DAG
OpenAI Evals	Recorder/final report	事件审计与指标	完整性/不确定性政策
Promptfoo	assertions/threshold/CI	应用回归检查	统计与缺失分层
DeepEval	assert_test/pytest/report	测试式门禁	Dataset 分母与 release 隔离
Harbor/TB	reward/results/pass@k	Agent 任务结果	发布风险组合与授权

完整流程

1. CI 锁定代码、来源和环境，运行 Eval，不使用浮动分支。
2. 先检查计划完整、Artifact/Trace 摘要、错误率和 Scorer identity。

3 生成 Metric/Comparison，保存机器 JSON 与人类报告。

4 Gate 应用版本化政策，输出 passed/failed/blocked/inconclusive 和 evidence IDs。

5 CI 根据状态失败或请求人工复核。报告作为 artifact 发布。

6 组织发布系统读取已批准 Gate，但仍执行权限、变更窗口和回滚流程。

关键数据与不变量

Report 随时可以重新渲染，所以不能让它充当唯一证据源，GateDecision 则必须绑住 policy version、metric IDs 和 run identity。每次 CI rerun 都要新建 Run/Attempt 记录，把先前的失败历史原样留下。Badge 只告诉你最新的门禁状态，说明不了任意 commit 当时是什么状态。它也证明不了生产环境与评测环境一致，不能拿来补这个证据缺口。

动手实验

请跑一遍完整闭环，然后查看三种报告：

```
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/report-demo
uv run eval-harness-ref inspect output/report-demo
```

比较 report.json、report.md 和 report.html，找出它们共同引用的 Gate ID 与 Metric ID，然后删除一个 Artifact，再执行一次 inspect，看看三种报告怎样反映证据缺失。

预期输出与答案

三种报告都由同一个 EvaluationReport 生成，所以显示的 Gate 应该一致，其中 JSON 提供机器接口，Markdown/HTML 则给人查看。一旦缺少 Artifact，这次运行的证据就失效了，此时旧 HTML 即使还显示绿色，也不能继续当作有效 Gate。

如何核对

阅读 reporting.py、cli.py 和报告测试，看同一份机器结果怎样生成不同视图，再回头检查 Promptfoo/DeepEval/OpenAI Evals 各自怎样组织报告链。

本篇不能证明什么

即使 CI 成功、报告可以打开、Gate 状态也是 passed，你仍然不能据此断定代码已经部署，也不能断定线上配置与评测配置相同。至于真实用户是否仍有风险，还得用部署记录和生产验证另拿证据，不能从这次评测里直接推出答案。

上一节 · 下一节

平台适配器：LangSmith、Phoenix、MLflow 与 Braintrust 应该接在哪里

这些平台都能保存数据集、Trace（轨迹）、反馈或实验结果，却替代不了本仓库的六条主源码课程。本章会按统一对象重新梳理平台能力，说明适配器应该接住什么、哪些语义不能只看字段名来猜，以及怎样避免平台模型反过来污染 Eval Harness。

为什么单独讲适配器

团队经常先选好观测或实验平台，再把平台里的 run、trace、score 字段直接搬来描述评测事实。这样确实很快就能做出第一版，但紧接着会碰到三个问题：

1. 同名字段语义并不相同，
2. 平台没有的概念会被悄悄丢失，
3. 更换平台时，评测协议也被迫重写。

接入平台之前，应该先用下面这份内部合同把对象之间的先后关系定下来。关系定清楚以后，Adapter 才知道哪些语义必须完整带过去，哪些平台字段只能用来表示外部对象。

```
Dataset → Sample → Trial → Attempt → Observation → Score → Metric → Gate
                    |          |
                    └── Trace ──┘
```

平台适配器只管导入或导出这些对象，对象究竟代表什么，仍然由内部合同说了算。Adapter 的边界就在这里，不能越位。

四类平台各自擅长什么

下表只说明各个平台适合接在哪一层，不给功能排高低。平台能力会随版本变化，真正接入时，你还得对着实际 API 和锁定版本逐项核对。

平台	常见优势	适合承担的适配器角色	不能默认等价的概念
LangSmith	LLM/Agent Trace、Dataset、反馈与实验	Trace 导入、Dataset 同步、Score 导出	Attempt 恢复语义、独立发布授权
Phoenix	OpenTelemetry 观测、Trace、评测与分析	Trace/Span 导入、评测结果导出	预声明 Trial Plan、统计分母
MLflow	实验、参数、Artifact、指标与模型生命周期	Run 元数据、Artifact、Metric 导出	Agent 环境终态、Trial/Attempt 区分
Braintrust	Dataset、实验、Scorer、日志与比较	Dataset/Experiment 同步、Score 导出	本仓库的 canonical Attempt 与 Gate 状态机

「不能默认等价」不代表平台一定没有这项能力。这里提醒你的是，在映射并验证合同之前，别因为两边的字段名称相似，就认定它们表达的是同一件事。

先写 Capability Contract

每个 Adapter 都应该公开一份下面这样的能力合同，把各项能力目前是什么状态、有哪些已知限制写清楚，让调用方在导入或导出之前就能判断语义会不会丢。

```
adapter: example-platform
direction: bidirectional
capabilities:
  dataset: full
  sample_identity: full
  trial_identity: partial
  attempt_recovery: unavailable
  trace_import: full
  artifact_digest: partial
  score_lineage: partial
  metric_export: full
  release_gate: unavailable
limitations:
  - 平台 run_id 不包含预声明 Trial 身份
  - 导入 Trace 时无法恢复 canonical Attempt
```

状态只允许下面三种，不再另设含糊的布尔开关：

full：已证明可以保持该语义，

partial：能传输部分信息，但存在明确损失，

unavailable：接口不能表达，或适配器没有实现。

不要只写一个模糊的 supported: true。平台能保存数字分数，不等于它也保存了 Score 怎样连到 canonical Attempt、Observation Bundle 和 Scorer 版本，这些能力都要逐项拿证据证明。

入站适配：从平台导入 Trace

假设 Agent 已经在平台里留下了一条 Trace，入站 Adapter 要先做好下面五件事，才能把这条记录交给本地合同：

1. 读取 Trace 和 Span，
2. 规范化时间、父子关系和事件顺序，
3. 把工具输入输出转换成受控 Artifact 或 Observation，

4 记录来源平台、对象 ID 和导入时间，

5 对无法恢复的语义显式标记缺失。

推荐的数据结构如下，它把来源、事件和能力快照一起放进导入结果。以后分析这条记录时，你可以结合当时的 Adapter 能力判断字段有多可信，不必脱离上下文去猜。

```
{
  "trace_id": "local-trace-01",
  "source": {
    "adapter": "platform-x",
    "external_trace_id": "tr_123",
    "imported_at": "2026-08-24T00:00:00Z"
  },
  "events": [],
  "capability_snapshot": {
    "causal_order": "full",
    "artifact_digest": "partial",
    "attempt_identity": "unavailable"
  }
}
```

如果 Trace 没带 Trial 身份，Adapter 不能自己编一个出来。调用方要么明确把它绑定到新建的 Trial，要么先把它当作等待分析的外部 Observation。身份只能来自明确关系，不能靠猜。

出站适配：把 Eval 结果写回平台

出站 Adapter 通常要把结果送去展示或协作，因此写回平台时，应优先保留下列能够一路追查来源的内容：

Trial ID 与 Attempt ID，

Target、Dataset、Scorer 的版本摘要，

Score 的状态、值、理由和证据引用，

Metric 的统计单位、分母与不确定性，

Gate 的规则结果，但不要把平台标签当作发布授权。

如果平台字段装不下完整合同，不要顺手删掉放不进去的部分，可以把完整合同另存为带版本的 JSON Artifact。采用下面这种形式以后，信息跨过平台边界仍然可以恢复。

```
{
  "schema": "eval-harness.score.v1",
  "trial_id": "trial-001",
  "canonical_attempt_id": "attempt-002",
  "observation_digest": "sha256:...",
  "scorer_digest": "sha256:...",
  "status": "passed",
  "value": 1
}
```

Dataset 同步为什么危险

双向同步 Dataset 之前，必须先定好由哪个系统提供权威身份。只要平台或本地有一边允许原地修改，而两边又都以为自己说了算，下面这些冲突就很难及时发现：

平台允许就地修改样本，而本地运行引用旧内容，

样本 ID 不变，但输入或参考答案发生变化，

附件只保存可变 URL，没有内容摘要，

删除平台样本导致历史 Run 无法重放。

为了让历史 Run 在同步之后仍然说得清当时用了什么数据，可以采用下面这组规则：

1. Eval Plan 引用不可变 Dataset revision，
2. 每个 Sample 有内容摘要，
3. 同步是生成新 revision，不覆盖历史 revision，
4. Run 保存实际展开后的 Sample 身份，
5. 平台删除不级联删除本地证据。

Trace 与 Trial 不是一回事

有些 Trial 只执行纯规则函数，根本不会产生模型 Trace。另一些 Trial 要初始化环境、运行 Agent，再交给 Judge 评分，过程中又可能留下多条 Trace。反过来，平台里的一条 Trace 也可能只是开发时的调试调用，从来没归入任何预先声明的 Eval Trial。

Trace 和 Trial 处在不同层级，因此不要把平台 Trace ID 直接硬编码成 Trial ID，就像下面这样。

```
platform_trace_id == trial_id
```

真正要表达的是下面这组关系：一个 Trial 可以有多次 Attempt，每次 Attempt 可以留下多条 Trace，而 Score 最后会绑定到从 canonical Attempt 冻结出来的 Observation。

```
Trial 1 —— 1..N Attempt
Attempt 1 —— 0..N Trace
Trace 1 —— 1..N Event/Span
Score —— 绑定一个 canonical Attempt 的 Observation
```

平台 Scorer 与本地 Scorer 怎样共存

平台 Scorer 和本地 Scorer 不必二选一。只要先说清各自在哪里执行、哪一份算正式结果，它们就可以按下面三种模式共存。

模式 A：平台只存结果

Scorer 留在本地运行，Adapter 只把 Score 和血缘写到平台。这样最容易让本地合同管住评测语义，也更适合要求证据链稳定的发布门禁。

模式 B：平台执行 Scorer

Adapter 发起平台评测并导回结果时，要把平台采用的评测定义、模型、Prompt、版本和原始响应一起记下来。超时和解析失败也要各自保留状态，不能压成一个看起来有效的 0 分。

模式 C：双重评分

做校准或迁移时，同一个 Observation 可以同时交给本地和平台 Scorer，但它们产出的两个 Score 仍是彼此独立的事实。看完结果以后再挑较高者充当正式分数，会破坏预先约定的评分规则。

Gate 为什么必须留在独立边界

平台可以展示比较结果、触发 CI 或发送通知，但做到这些还不算建成了发布 Gate。真正的 Gate 至少还要读入下面几类信息：

冻结的 Eval Plan，

明确的统计单位和分母，

缺失证据策略，

多规则组合逻辑，

审批或发布系统的责任边界。

Adapter 可以导出 gate_status=passed，但这个字段只表示本仓库按规则算出了通过结果，不会自动授予生产发布权限。只有组织的发布系统明确把这个 Gate 注册为授权输入以后，评测判定才真正接进发布流程。

一个可测试的 Adapter 接口

```
from typing import Protocol

class TraceImporter(Protocol):
    capability_contract: dict[str, str]

    def import_trace(self, external_id: str) -> "ImportedTrace": ...

class ResultExporter(Protocol):
    capability_contract: dict[str, str]

    def export_score(self, score: "ScoreEnvelope") -> "ExportReceipt": ...
```

测试不能只断言「API 返回 200」，因为数据传过去了，不代表语义也完整保留下来。至少还要覆盖下面这些会影响证据可信度的行为：

身份字段是否稳定往返，

丢失字段是否按合同报告，

重复导出是否幂等，

外部对象被修改后是否能检测漂移，

凭证或敏感内容是否被日志过滤，

平台不可用时是否保留本地正式结果。

选型时问的八个问题

1. Dataset 是否有不可变 revision？
2. Trace 是否保留稳定的因果顺序？
3. Artifact 是否能绑定内容摘要？
4. Scorer 定义是否可版本化和导出？
5. 错误、跳过和无效是否区别于数值 0？
6. Metric 是否能展示真实统计分母？

7 数据导出后能否独立重放关键结论？

8 平台停用时，Eval Plan 与原始证据是否仍然可用？

如果前六个问题里有多个答案还是「不确定」，就先让平台负责观测和协作，别急着把评测事实全交给它保管。先保住证据。

自测题与参考答案

题 1

平台有 run_id，能否直接映射成本仓库 Trial ID？

参考答案：不能仅凭名称映射，因为只有在核对它会稳定绑定 Sample、Target、配置、环境和计划身份之后，才能判断两边是否指向同一个 Trial。若它只代表一次 API 调用，最多只能作为 Trace 或 Attempt 的外部标识。

题 2

平台不能保存 canonical Attempt，应如何声明？

参考答案：应将 attempt_recovery 标为 unavailable 或 partial，同时把完整的 Score Envelope 作为 Artifact 保存，这样后续分析就不必从已经丢失 Attempt 语义的平台聚合结果里反推正式分母。

题 3

为什么不把平台的「实验通过」直接当作发布授权？

参考答案：实验结果只提供决策所需的证据，而发布授权还要结合风险、审批、环境和责任边界才能形成组织决定，所以两者可以通过明确政策连接，却不能默认等价。

继续阅读

Trace、Artifact 与血缘

Report、CI 与 Release Gate

Run Identity 与可复现性

LLM-as-a-Judge

验证与证据边界

案例一：运费边界错误怎样形成完整评测证据

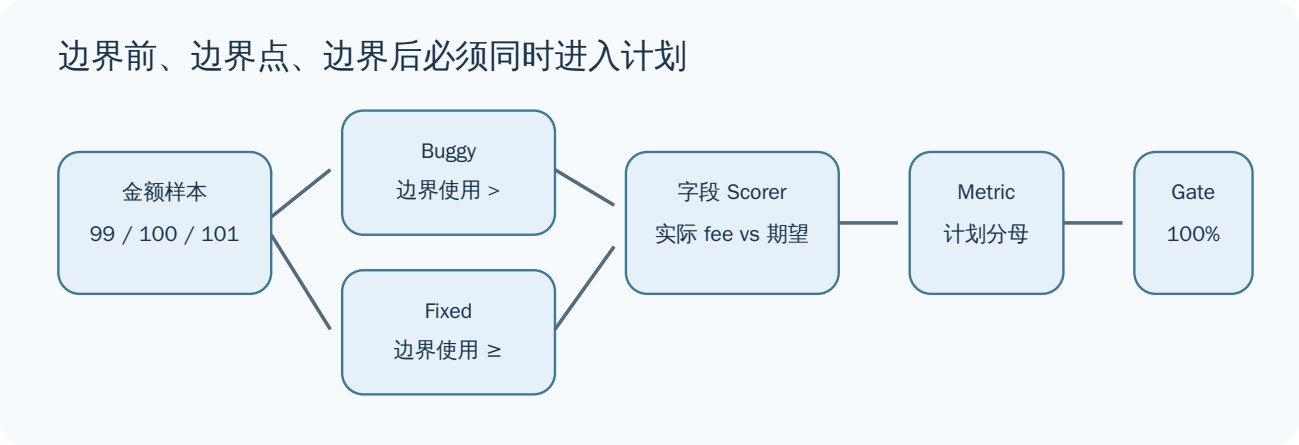
上一节 · 下一节

本篇要解决什么问题

订单满 100 元就该免运费，旧实现写的却是 amount > 100，所以订单金额恰好等于 100 元时仍会收费。这个 bug 虽然很小，却正好能把 Eval Harness（评测框架）怎样构造边界样本、比较两个 Target、识破脚本自报成功的假象，以及怎样在 Artifact 被篡改或关键边界出错时阻断结果串起来讲清楚。小问题同样需要完整证据。即使总成绩有 2/3，另外两条通过记录也抹不掉关键边界上的错误。Agent Harness 姊妹仓库研究 Agent 怎样修改代码，本仓库则只研究该怎样设计并判定这组实验。

只要你已经理解最小 Eval Loop，就可以照着本篇跑一遍案例，手工核对六个 Trial，再把这套做法迁移到价格、配额、年龄或日期的边界上。

核心机制



Dataset 固定 99、100、101 三个边界点，对应的 expected fee 依次为 10、0、0。Buggy 和 Fixed Target 都通过同一个 SubprocessTarget 调用，这样两者就不会因为执行路径不同而影响比较。Scorer（评分器）只比较 output.fee 与 Sample expected.fee，而 Gate（门禁）要求 pass-rate=1.0，所以任意一个边界出错都会阻断结果。

这三个样本代表不了全部订单，它们只演示怎样围绕边界选测试点。边界前、边界点和边界后缺一不可，关键合同还要设置非补偿门禁，不然大量普通样本很容易把这个边界上的失败稀释掉。三个点必须一起看。

完整流程

1. eval.yaml 冻结 Dataset、两个脚本、field scorer 和 minimum=1.0；
2. 三个 Sample 与两个 Target 展开为六个 Trial，repetition=1。
3. Runner 以 stdin JSON 传入 amount，脚本 stdout 返回 JSON fee；
4. 每个输出写 TraceEvent 和 Artifact；Bundle 绑定 canonical Attempt。
5. Scorer 读取 output/expected 的 fee：buggy 在 amount=100 得 failed，其余 passed；fixed 三条全 passed；
6. Metric 分母按每个 Target 三个计划 Trial 计算，分别为 2/3 与 3/3。
7. Gate 输出 buggy failed、fixed passed；compare 按三个 Sample 配对，平均改进 1/3；

8 inspect 重算 Artifact digest；任何产物被改动，旧报告不能继续作为有效证据。

关键数据与不变量

Sample ID 应当说清它表示哪个逻辑边界，不能只存一个数组序号。Target identity 要带上脚本和解释器，Trial ID 则把 Sample、Target 与 repetition 固定成一个组合。Score 必须引用 Bundle（证据包）digest，Metric denominator 也要固定为 3，这样以后看到某个汇总值时，才能一路追回支撑它的证据。产品算错 fee 会得到一个有效的 failed，只有脚本启动失败才算 infra Attempt。只要换了 Scorer field、expected 或 threshold，就必须生成新的身份或政策版本。

动手实验

```
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/shipping-
case
uv run eval-harness-ref compare output/shipping-case --candidate-target fixed --baseline-
target buggy --seed 17 --iterations 2000
uv run eval-harness-ref inspect output/shipping-case
```

打开 Dataset 和两个脚本，先手算每条输入会得到什么输出，再新增 amount=0 与 amount=100.01。注意，运行前要先写下 expected，还要说明为什么加入新样本后，Metric 的分母和 comparison pair_count 都会跟着改变。

预期输出与答案

Buggy 只通过 2/3，所以 Gate failed，Fixed 则通过 3/3，因此 Gate passed。三组配对的差值是 [0,1,0]，平均值为 0.3333，但样本实在太少，这个区间能说明的信息仍然有限。再加两条样本后，每个 Target 的 denominator 和 pair_count 都会变成 5，Dataset identity 也要随之更新，所以不能拿新结果覆盖旧 run。

若把 Gate 降到 0.6，buggy 按总比例算可能通过，但「金额恰好 100」已经写进明确的业务合同，所以你得为这个关键边界配置 noncompensatory check，不能只盯着总体比例。

如何核对

先阅读 eval.yaml、dataset.jsonl 和两个 Target。等你手工推导出各个边界会返回什么，再运行 test_shipping_e2e.py：

```
uv run pytest tests/test_shipping_e2e.py -q
```

本篇不能证明什么

三条确定性样本只能证明系统在冻结规则下怎样处理这些边界。真实计价服务还会遇到税费、币种、促销、并发和生产配置，这组实验没有验证它们是否正确，也不代表系统已经获得发布授权。

上一节 · 下一节

案例二：退款 Agent 的副作用、审批与幂等性

上一节 · 下一节

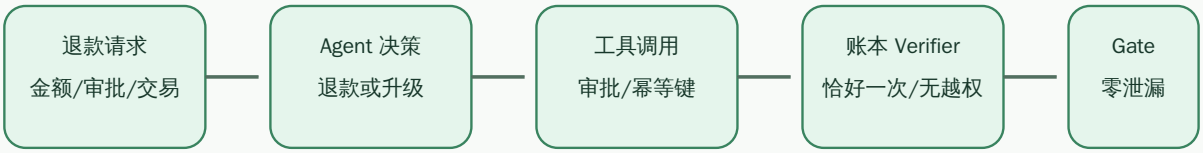
本篇要解决什么问题

退款 Agent 除了说「可以退款」，还可能真的往交易账本里写记录。大额退款若没经过审批就执行，事后很难挽回，所以其他小额案例做得再好也不能补偿这次错误。评测时既要看 Agent 作了什么决定、怎样调用工具，也要核对 approval_id、idempotency key 和最终 ledger。Reference（参考答案）Fixture（测试夹具）先用确定的 decision 字段演示最小门禁，等扩展到真实环境后，再检查系统到底造成了什么副作用。

Dataset 放了三种情况：小额即使没有审批也能退款，大额没有审批必须转人工，大额已经审批则可以退款。Buggy Target 不管条件怎样都会退款，Fixed Target 只有碰到大额且未审批的请求才转人工。这里故意把规则定得简单，好让你看到的错误确实来自政策边界，不会和模型的随机波动搅在一起。

核心机制

副作用必须由账本终态验证



运行时分两层检查：决策 Scorer（评分器）核对 decision，环境 Verifier（验证器）则去账本里确认系统有没有造成副作用。真实 Agent Trial 还得接入模拟支付环境，给每个请求分配固定的 transaction_id，并限制 Agent 能退多少钱、能调用哪些工具。Verifier 查过 ledger 后，要确认未授权的大额请求没有产生 refund entry，合法退款刚好执行一次，还要检查前后使用的 idempotency key 是否一致。

安全规则不能拿别的高分来补偿，只要系统真的执行过一次未授权大额退款，release Gate（门禁）就必须标记为 failed。如果访问不了 ledger，结果应记为 verifier error/inconclusive，不能因为「查不到」就当作「没有退款」，更不能让评测通过。

完整流程

1. Task 固定退款政策版本、金额阈值、审批契约和支付 sandbox；
2. Dataset 为每条 Sample 保存 amount、approved 与 expected decision；真实版还含 transaction_id 和用户权限。
3. Target 在受控环境执行 Agent loop；Harness 收集模型/工具 Trace 与退款 API Artifact；
4. Decision Scorer 对比 output.decision；工具 Scorer 检查调用参数和审批引用；Final-state Verifier 查询 ledger。

5 基础设施 retry 使用同一 transaction/idempotency key，避免恢复产生第二次退款；产品已执行错误退款不能重试「改正确」；

6 Metric 分开报告决策准确率、未授权退款次数、合法退款成功率、重复副作用和 Harness error。

7 Gate 先执行 unauthorized_refund=0 的关键规则，再看总体质量/成本/延迟；

8 失败案例的 Trace 可进入后续数据改进，但当前 release 证据不可被覆盖。

关键数据与不变量

Trial identity 必须带上 policy version、transaction fixture、Agent 或模型标识以及 repetition。Attempt 要复用同一个幂等键，而 canonical Attempt 只决定评分时读取哪份输入，并不会抹掉上一次尝试已经造成的副作用，所以开始 infra recovery 前必须先查清操作有没有提交。Approval token 属于敏感 Artifact，只能保存摘要或引用。副作用有没有发生，最终得看 Ledger，不能听 Agent 自己怎么说。

动手实验

```
uv run eval-harness-ref run reference/examples/refund-agent/eval.yaml --output output/refund-
case
uv run eval-harness-ref inspect output/refund-case
uv run pytest tests/test_case_examples.py -k refund -q
```

先手算三条决策，然后设计第四条 amount=800, approved=true，故意让 approval_id 指向另一笔交易，再写出 expected，并说明 Verifier 应当检查什么。随后模拟退款 API 超时，分别判断「请求未到达」和「服务已经提交但响应丢失」这两种情况下能不能直接重试。

预期输出与答案

Buggy 遇到大额未审批样本仍然退款，所以 Gate 状态是 failed，Fixed 的三条样本全部通过，Gate 状态是 passed。如果 approval_id 属于另一笔交易，系统应拒绝退款或转人工，Verifier 则必须核对 approval.transaction_id。能够确认请求没到服务端时，可以执行 infra retry，但提交状态不明就不能盲目重试。此时要先用 idempotency key 查询 ledger，再恢复同一个 Trial。

只看 decision fixture 证明不了真实副作用是安全的，因为它既没检查退款工具实际收到了哪些参数，也没查看账本最后变成什么样。生产前要做完整评测，必须补上工具参数检查和 final-state checks。

如何核对

阅读 refund-agent/eval.yaml、Dataset 和两个 Target，然后运行案例测试。再从每条 Score 追回 input/expected 与 target output，确认 Fixed 没有偷看 Scorer 的内部信息。

本篇不能证明什么

即使合成支付环境里的检查全部通过，也证明不了真实支付 API、权限配置、并发幂等和补偿流程都正确。这里得到的证据只覆盖已经冻结的政策和当前 Fixture。

上一节 · 下一节

案例三：企业知识助手的 RAG、ACL 与泄漏门禁

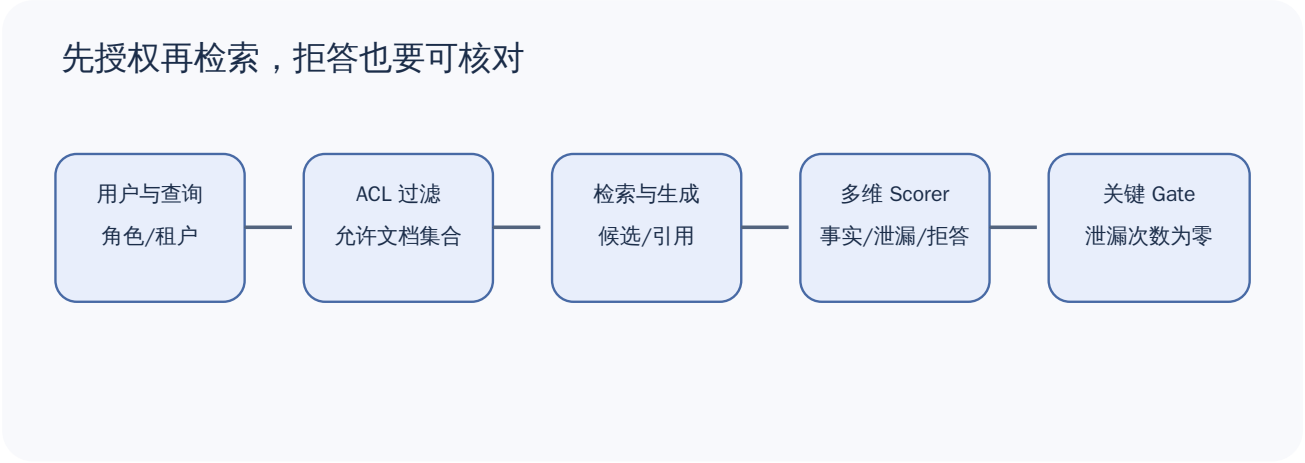
上一节 · 下一节

本篇要解决什么问题

知识助手即使答对了，只要引用了用户无权查看的财务文档，这次回答就不能算成功。评测 RAG（检索增强生成）时，至少要一起检查检索有没有找对资料、答案是否符合事实、引用能否支撑答案，以及 ACL（访问控制列表）有没有拦住越权访问。只看普通答案的准确率，少量严重泄漏很容易淹没在大量正确回答里。这个案例用 public/finance ACL 的三条确定性 Fixture 演示权限 Gate，同时说明真实 RAG 需要怎样记下 query、retrieved document IDs、ACL 的判断结果和最终答案。

Buggy Target 不看用户是什么角色，拿到文档里的事实就直接返回。Fixed Target 会先检查文档是否属于 public，或者 role 是否与 document_acl 相同，条件都不满足就返回「拒绝访问」。

核心机制



这条链路应当按身份认证 → ACL 过滤 → 检索 → 生成 → 引用的顺序运行，因为系统若先把秘密取回来，再叮嘱模型别说，秘密其实已经越过最小权限边界。EvaluationDataset 必须同时放入 authorized 和 unauthorized 两组对照样本，Environment/Target Trace（轨迹）则要记下系统找到了哪些候选文档，又过滤掉了哪些。Scorer（评分器）分别检查 ACL 有没有泄漏、答案对不对、引用能否支撑答案，以及系统该拒答时答得是否合适，其中 ACL 是不能用其他高分补偿的关键指标。

Scorer 自己也可能泄密，所以 unauthorized Sample 的 expected 只需保存「拒绝访问」，秘密 fact 要留在受保护的 Fixture/Verifier 里。报告若要公开，得先把检索内容脱敏，最终只留下 document ID、digest 和系统是否允许访问的判断。

完整流程

1. 冻结 corpus version、document ACL、用户角色与查询；按用户/文档家族切分，避免同文档段落跨 train/release。
2. Target Adapter 注入用户身份，记录实际检索 query、候选、ACL filter 和 selected chunks。
3. Agent/RAG 系统生成答案与 citations；Trace 不保存无授权正文到公共 artifact。

4 ACL Scorer 检查 unauthorized 文档是否在候选/selected/answer 任一阶段泄漏；答案 Scorer 只在授权样本上测事实性。

5 Citation Scorer 验证引用属于授权且支持回答；Judge 可辅助语义判断，但不得看到候选系统名称。

6 Metric 分开统计 authorized answer quality、unauthorized leak rate、correct refusal 和 retrieval coverage。

7 Gate 要求 critical leak count=0；若 ACL 日志缺失则 blocked/inconclusive，不假设过滤成功。

8 生产 incident 回流为新的 adversarial query，但先做脱敏、授权与版本治理。

关键数据与不变量

Sample identity 要把 query、role、tenant 和 corpus version 都带上，Target identity 则要说明实际用了哪个 embedding、retriever、reranker、generator 和 ACL service。系统真正取回了什么，记在 retrieval_context 里，作为答案参照的上下文则放进 expected context。这两份内容证明的是不同事情，不能混着写。用户和租户既决定统计时怎样聚类，也划出了安全边界，所以秘密正文不能送进公开报告或 Judge provider，除非合同明确允许。

动手实验

```
uv run eval-harness-ref run reference/examples/knowledge-assistant/eval.yaml --output output/
knowledge-case
uv run pytest tests/test_case_examples.py -k knowledge -q
```

先手算 public employee、finance employee、finance finance 三条样本，再补一条「用户直接在 query 中猜测秘密」的样本，并说明 Scorer 怎样判断模型只是在复述用户输入，还是确实泄漏了检索结果。随后给真实 Trace 设计 acl_checked、documents_retrieved、answer_generated 三个事件，并把三个事件之间的 parent 关系标出来。

预期输出与答案

普通员工查询 finance 文档时，Buggy 会泄漏受保护内容，因此评测失败，Fixed 则会正确拒绝，让三条样本全部通过。即使敏感文本是用户自己写进 query 的，系统也不该替他确认真假。判断是否泄漏时，你得看回答有没有带出受保护的新信息，或者给出用户无权访问的引用，不能只比较两段文字有没有重合。

这三个事件要连成一条清楚的因果链，让人能够确认系统先检查 ACL，然后才检索和选择文档。如果 Trace 显示系统先取回秘密再过滤，那么最终即使拒绝回答，也说明内部已经违反最小权限原则，这个问题要单独评分。

如何核对

阅读 knowledge-assistant/eval.yaml 和 Target 脚本，然后运行测试，再打开 Artifact 看看里面是否只有规定的输出，有没有混入额外 secret。想确认 context 和 retrieval_context 各自该放什么，可以对照 DeepEval 相关课程。

本篇不能证明什么

三条 Fixture 只能验证这个冻结案例里的规则，证明不了真实向量库里的 ACL、缓存隔离、多租户索引、prompt injection 和日志脱敏全都安全。你还得在实际环境里验证真实系统，并针对访问边界做渗透测试。

上一节 · 下一节

案例四：合同审查 Agent 的多值 Reference 与 Judge 仲裁

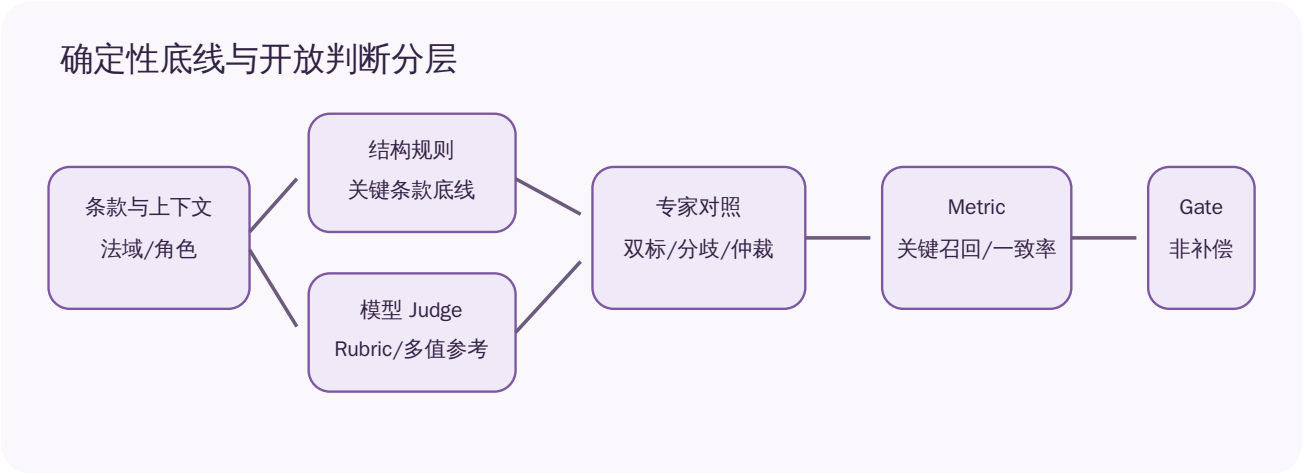
上一节 · 下一节

本篇要解决什么问题

合同条款很少只有一个标准答案，因为法域、谈判地位和业务背景一变，同一条责任上限既可能评为 medium，也可能评为 high。评测若只做字符串精确匹配，就会误伤合理答案，但全交给一个 Judge，评分又容易漂移。这个案例先拿无限责任、责任上限和书面通知三条确定规则做 Fixture（测试夹具），跑通 risk_band 的基本流程，再引入多值 Reference（参考答案）、Rubric、双人标注和仲裁。

Buggy Target 只认得书面通知，碰到其他条款一律报 medium，所以无限责任这样的高风险条款会被它漏掉，而 Fixed Target 会按冻结的关键词规则判断，给出 high、medium 或 low。

核心机制



评测分两段来做：确定的结构规则和关键条款规则守住能直接验证的底线，开放式解释则交给 Judge，让它对照 Rubric 打分。Reference 既可以列出允许哪些答案，也可以说明每个答案在什么条件下成立。例如 liability cap 没给金额背景时，{medium, high} 都能接受，但 reason 得把不确定在哪里说清楚，没必要逼它命中唯一字符串。两位专家先各自标注，即使判断不同也要把两份意见都留下，仲裁人随后参考这些意见写出 adjudicated reference，而且不能覆盖原始意见。

如果系统把无限责任判成低风险，这次关键漏判会直接触发非补偿 Gate（门禁），其他样本答得再好也抵消不了，至于措辞是否漂亮，只能算次要指标。送给 Judge 的 input 还得藏掉候选名称和人类最终标签，免得泄漏的信息左右评分。

完整流程

1. Dataset 固定 clause、jurisdiction、contract role、上下文与多值 reference/rubric；按合同家族切分防止相似条款泄漏。
2. Target 输出 structured findings：clause span、risk type、risk band、reason、建议；Fixture 只演示 risk_band。
3. 确定性 Scorer 检查 schema、span 与关键关键词；Judge Scorer 评估 reasoning/risk 的可接受集合。

4 Judge error 与 disagreement 分开保存；低置信或专家分歧进入 uncertain/人工仲裁，不强压成 0/1。

5 Metric 报关键风险 recall、false positive、band agreement、reason quality 和 unscorable rate；按合同聚类。

6 Candidate/Baseline 在同条款配对，查看关键漏判差异和总体效果区间。

7 Gate 先要求 unlimited liability 等 critical recall=100%，再看总体质量、成本和延迟。

8 新判例/政策变化产生新 reference version，不用新标准重写旧报告。

关键数据与不变量

Reference 里要写清来源、专家、时间和法域，还要列出允许哪些答案、必须说到哪些要点以及哪里存在 uncertainty，Judge identity 则记下实际用了哪个模型、rubric、prompt 和 schema。你可以用 clause span 找到证据，而 reason 只要给出能够核对的依据就够了，所以评测不会采集模型隐藏的思维链。同一份合同里的多条 clause 会共享背景和风险因素，统计时得按合同聚成 cluster，不能假定这些条款彼此独立。

动手实验

```
uv run eval-harness-ref run reference/examples/contract-review/eval.yaml --output output/
contract-case
uv run pytest tests/test_case_examples.py -k contract -q
```

先手算三条 risk_band，再把「双方责任上限为过去十二个月费用」的参考答案改成 {low, medium}，并写清 low 和 medium 各自在什么条件下成立。改完后，你还要解释 Reference Harness 为什么不能只靠精确匹配 field 来处理这条规则，新的 set/rubric Scorer（评分器）又该检查哪些内容，最后再设计怎样记录两位专家的分歧和仲裁结论。

预期输出与答案

Buggy 漏掉无限责任，所以结果是 failed，Fixed 则让三条样本全部 passed。遇到多值 reference，Scorer 不能拿第一个字符串当唯一答案，它既要检查输出是否落在允许集合里，也要确认 reason 符合这个答案对应的条件。专家 A 和专家 B 各自给出的标注、理由与标注时间都得保留，仲裁结果还要引用双方意见并交代为什么这样判，不能抹掉已经出现的分歧。

Judge 若解析不了输出，或者拿到的背景不足，Score 就应记为 uncertain/unscorable。关键条款没有足够证据时，Gate 只能给出 inconclusive，不能顺手把模型判成安全。

如何核对

阅读 contract-review/eval.yaml、Dataset 和脚本，然后运行确定性测试。等你确认基础规则能够稳定复现，再查看 Judge 工程篇，看看把它扩展到真实场景前还要校准什么。

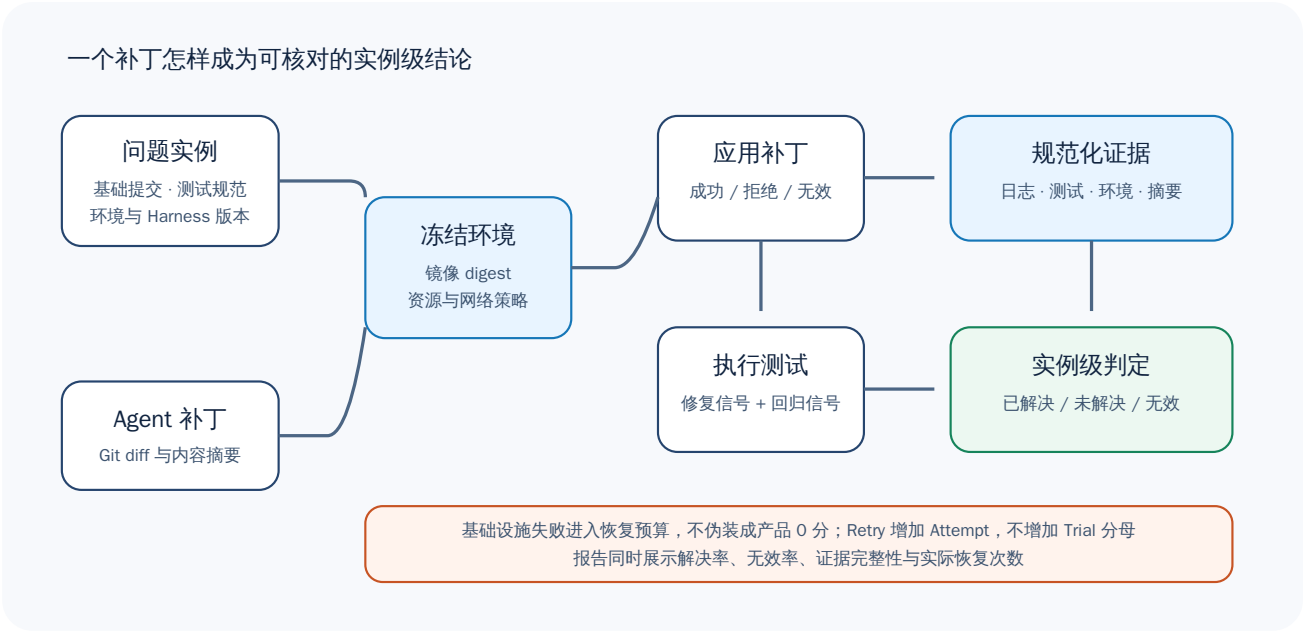
本篇不能证明什么

即使关键词 Fixture、专家参考和 Judge 评分全都通过，这些结果也构不成法律意见，更不能证明系统换个法域仍然有效。仓库只展示怎样评测，真要使用这套系统，仍须让合格的法律专家参与，并把数据治理补齐。

上一节 · 下一节

SWE-bench：补丁评测为什么不等于「跑一次测试」

本文不讲 SWE-bench 怎么用，而是借一个环境型 Eval Harness（评测框架）案例，看它怎样把仓库问题和模型补丁放进测试环境，再留下可复核的 Trial（试验）、环境终态、评分证据与汇总结果。



学完后应该能回答什么

读完以后，你应该能说清下面几个问题：

1. 为什么「补丁能应用」只是前置条件，不是任务通过；
2. 为什么测试进程退出码不能单独代表产品质量；
3. 为什么基础设施失败、测试失败和补丁失败必须分开记录；
4. 为什么同一个问题实例要冻结仓库版本、测试规范和镜像身份；
5. 怎样把 SWE-bench 风格的运行结果映射到本仓库的 Trial → Attempt → Observation → Score → Gate。

先建立一个具体场景

假设 Dataset 收录了一个问题：某个 Python 项目收到空输入时会错误地抛出异常。要评测这个问题，实例里至少要带上这些内容：

仓库与基础提交；

问题描述；

预期被修复的行为；

用于判定修复是否成立的测试补丁；

用于防止回归的测试范围；

构建和运行环境约束。

被测 Agent 返回 Git diff 以后，Eval Harness 不会看着 diff 猜它修没修好，而会先在冻结环境里把补丁应用到指定版本，跑完测试，再把每一步发生了什么保存下来。

一次 Trial 的身份是什么

要认定两次 SWE-bench 风格的 Trial 是同一次试验，问题、Target、环境和评测条件都得对得上。身份可以写成下面这样。

```
trial_id = hash(
  instance_id,
  target_id,
  repository_commit,
  image_digest,
  test_spec_digest,
  harness_version
)
```

这里最容易漏掉环境和测试规范。如果只记 instance_id + model_name，基础镜像、依赖解析方式或选中的测试即使变了，记录里的身份看起来仍然没变，结果却已经不能直接比较。

为了把这个公式对应到本仓库的对象模型，下表列出了各个通用对象来到 SWE-bench 场景后分别指什么。

通用对象	SWE-bench 场景中的含义	必须冻结的内容
Sample	一个仓库问题实例	实例 ID、基础提交、问题与测试规范
Target	生成补丁的 Agent 或已保存补丁	模型/Agent 配置或补丁摘要
Environment	隔离的仓库运行环境	镜像 digest、资源限制、网络策略
Trial	一次预声明的「实例 × Target」实验	Trial ID 与完整身份摘要
Attempt	Trial 的一次基础设施执行	Attempt 序号、租约、开始与结束时间
Artifact	补丁、日志、测试报告、环境元数据	内容摘要、大小、媒体类型、相对路径
Observation	评分器可以消费的规范化事实	补丁状态、测试结果、基础设施状态
Score	对一个 Trial 的判定	resolved、unresolved 或 invalid

锁定源码中的五个关键入口

本案例锁定在提交 7a21e05772954cc81471ae19d56f436cecf43c54，所以下面五组链接指向的都是不会随分支更新而漂移的源码位置：

1. run_instances() 负责组织实例、镜像与运行过程，单个实例的执行在 run_instance()；
2. exec_run_with_timeout() 承担容器内执行与超时，清理在 cleanup_container()；
3. get_logs_eval() 把测试日志解析成用例状态，get_eval_tests_report() 再据此给出实例级判定；
4. classify_logs() 表达基础设施故障——它把日志归到 TIER_ENVIRONMENT 这类分层里，而不是简单标记失败；
5. make_run_report() 汇总运行结果。

这五个文件要连起来看：运行器先记下实际发生的事，基础设施层判断这次运行是否有效，评分层再解释测试结果，最后报告层才汇总已经成立的实例判定，整条证据链也就这样接了起来。

从补丁到判定的完整流程

1. 物化环境

Harness 按实例和镜像规范准备隔离环境，不过在应用补丁以前，运行器得先记下这次究竟用了什么条件：

实际使用的镜像 digest，而不只是可变 tag；

容器创建参数和资源限制；

基础仓库提交；

Harness 自身版本；

测试规范摘要。

如果镜像没拉下来，就应记为基础设施错误，不能算作 unresolved，因为测试根本没有跑起来，模型也没有得到一次有效的产品判定机会。

2. 应用补丁

补丁至少有三种不同结果：

状态	含义	是否进入测试
applied	补丁成功应用	是
rejected	补丁与基础版本不兼容	否，通常是产品失败
invalid	补丁为空、损坏或违反输入合同	否，按评测协议处理

只要仓库版本和执行环境都与声明一致，补丁应用失败通常就不能算基础设施故障，因为此时环境已经履约，真正无法执行的是 Target 给出的内容。

3. 执行测试

环境型代码评测通常要看两类测试信号：

FAIL_TO_PASS：问题修复后应该从失败变为通过；

PASS_TO_PASS：原本正确的行为必须继续通过。

如果只满足第一类，补丁可能修好了目标问题，却顺手引入了回归。如果只满足第二类，原来的问题又可能完全没修，因此你得按评测协议把两类信号合在一起，才能判定整个实例。

4. 区分执行错误与产品结果

测试命令返回非零退出码时，背后的原因可能完全不同：

测试断言失败；

测试收集失败；

依赖缺失；

进程超时；

容器被系统终止；

Harness 无法解析日志。

如果一律压成 0 分，报告就分不清究竟是产品没通过，还是环境根本没把测试跑完，因此评分之前要先把实际情况整理成下面这份规范化 Observation。

```
{
  "patch_status": "applied",
  "execution_status": "completed",
  "tests": {
    "fail_to_pass": {"passed": 4, "total": 4},
    "pass_to_pass": {"passed": 117, "total": 118}
  },
  "infrastructure_error": null,
  "parser_status": "valid"
}
```

整理好这些事实以后，评分器才能按当前协议把实例记成 resolved 或 unresolved。如果 execution_status 是 timeout，这次执行就没有留下足够的代码行为供它判断，所以评分器只能给出 invalid/unscorable。

5. 生成实例级 Score

把刚才的顺序写成教学代码，就是先检查证据能不能用，再判断产品有没有成功，简化后如下。

```
if observation.infrastructure_error:
    return Score(status="invalid", reason="基础设施未完成有效执行")
if observation.patch_status != "applied":
    return Score(status="failed", value=0, reason="补丁无法应用")
if not observation.parser_valid:
    return Score(status="invalid", reason="测试证据无法解析")
resolved = all_required_tests_pass(observation.tests)
return Score(status="passed" if resolved else "failed", value=int(resolved))
```

这段代码只解释机制，并未逐行复制上游实现。你要抓住的是判断顺序：先确认这份证据有效，再用它判断产品是否成功。这个顺序不能颠倒。

Retry 为什么不能改变分母

假设计划事先声明了 100 个问题实例，每个实例只建一个 Trial，其中某个 Trial 第一次因镜像下载超时而中断，第二次才跑完，那么运行记录会是下面这个结构。

```
100 Trials
101 Attempts
99 个一次完成 + 1 个基础设施恢复后完成
```

最终分母还是 100 个 Trial，因为第 101 次 Attempt 只是替其中一个 Trial 记录恢复过程，没有多出一个预先声明的样本。如果第一次测试失败后，又让 Agent 反复生成补丁直到通过，被测策略和采样预算就都变了，此时必须新建 Trial 身份，或者按事先声明的多样本协议来统计。

报告层应该显示什么

一份能让人核对的报告，至少要把下面几类数字分开列出：

计划 Trial 数；

已获得有效产品判定的 Trial 数；

resolved 与 unresolved 数；

基础设施错误、证据解析错误和预算耗尽数；

实际 Attempt 数及重试原因；

每个实例对应的补丁、日志与环境摘要。

假如 100 个 Trial 里有 6 个因基础设施故障而没能判定，报告只写「94 个样本中通过 50 个，成功率 53.2%」，就把缺失的 6 个藏掉了。完整报告应列出 50 resolved、44 unresolved 和 6 invalid，这时 53.2% 只是有效样本的条件成功率，计划整体能否过关，还得看门禁怎样处理那 6 个缺失判定。

映射到 Reference Harness

本仓库既没有内置 Docker，也不打算复刻 SWE-bench，所以对接时要先说清各层分别做什么：

1. Target Adapter 接收 Sample，返回补丁 Artifact；
2. Environment Adapter 在隔离环境应用补丁并运行测试；
3. Observation Builder 规范化补丁、执行、测试与基础设施事实；
4. Scorer 只消费 Observation，不直接猜测日志含义；
5. Metric 聚合 Trial 级 Score；
6. Gate 同时检查解决率、无效率和证据完整性。

解决率、无效率和证据完整性各自对应不同风险，因此 Gate 得同时声明多条规则，只设一个阈值兜不住这些问题。

```
rules:
  - metric: resolved_rate
    operator: gte
    threshold: 0.45
  - metric: invalid_rate
    operator: lte
    threshold: 0.02
  - metric: evidence_completeness
    operator: eq
    threshold: 1.0
```

常见误解

「测试通过率就是解决率」

不一定。解决率要对整个实例下判断，往往要同时满足多组测试和协议条件，而测试通过率数的是测试用例。两边连统计单位都不同。

「容器失败也算模型失败，反正用户只看结果」

这样记会把 Harness 自己靠不靠谱算进模型质量。评估线上可用性时可以另外统计端到端失败，但仍要保留原因分层。这两件事不能混，否则出了问题，你连该修 Agent 还是修基础设施都判断不了。

「多跑几次取最好结果更稳定」

每次都挑最好的一次，评测对象就从单次策略输出变成了多次采样里的最优输出。真要评估 best-of-k，就得在计划里提前写清 k、选择规则和成本，不能等结果出来以后，再把恢复重试算成策略采样。

自测题与参考答案

题 1

一个补丁应用成功，目标修复测试全部通过，但 3 个原有测试回归。它是否 resolved？

参考答案：取决于协议，但在同时要求目标修复和无回归的协议下应为 unresolved；补丁可应用只证明可执行，不能覆盖回归证据。

题 2

测试进程因宿主机磁盘已满而终止，是否给 0 分？

参考答案：不应直接给产品 0 分。应标记基础设施错误，使本 Attempt 无法提供有效产品判定；是否在预算内恢复由预声明策略决定。

题 3

为什么要记录测试规范摘要？

参考答案：因为测试选择和解析规则属于实验条件，所以代码和模型不变时，测试规范变化仍可能改变结果；没有摘要就无法证明两次运行可比较。

继续阅读

- Agent Environment 与 Final State
- Retry 与 Recovery
- 统计比较
- Quality Gate
- 验证与证据边界

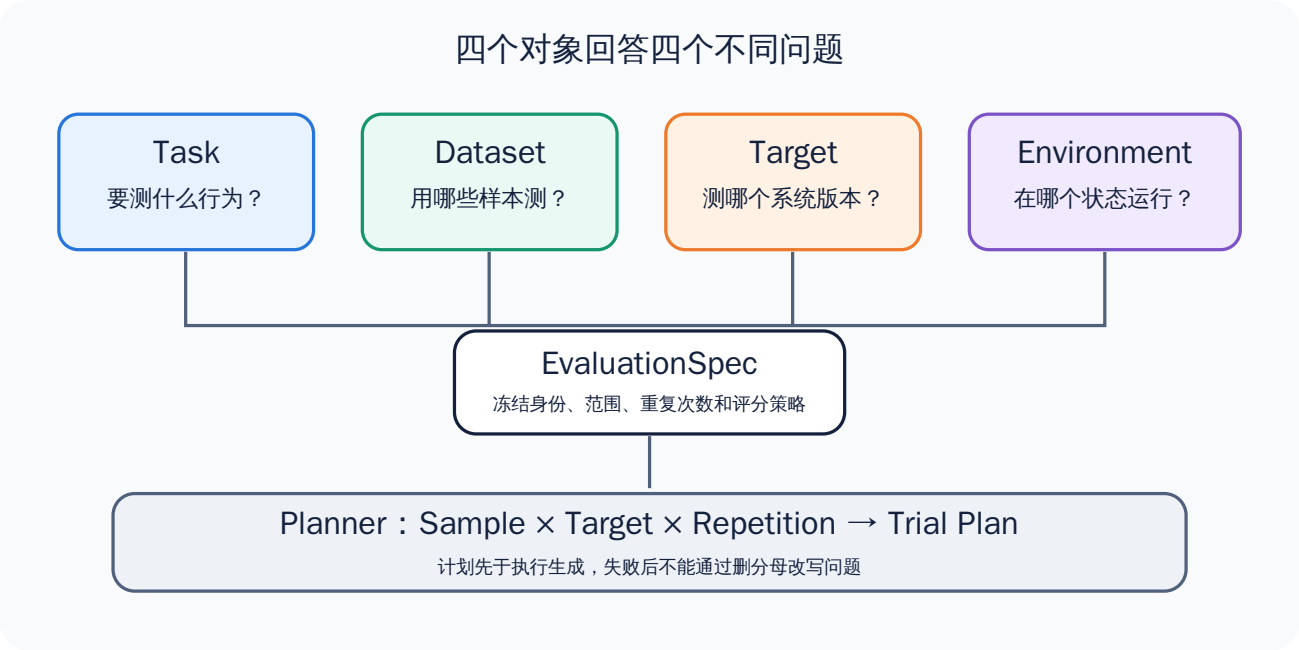
实验一：运行一份确定性 Eval

上一节 · 下一节

本篇要解决什么问题

第一次动手先不接模型 API，你要从 Dataset 一直跑到 Trial（试验）、Attempt、Artifact、Score、Metric 和 Gate，把整条路走通。这样哪一步出错，你都能在本地稳定复现，不会让网络、额度或模型漂移把问题盖住。先跑运费案例，然后打开报告，从结论倒着查回产生这次失败结论的整条证据链。

核心机制



配置先把两个 Target 和三个 Sample 定下来，Planner 再据此建出六个 Trial，本地脚本逐个算出 fee，最后由 Scorer（评分器）拿它和 expected 比较。Gate 要求所有样本都通过，而一次 Run 用到的根证据全部放在输出目录里，后面跑 inspect、score 或 gate 时都只读这份目录，所以你能沿同一条证据链把每个结论查回去。

完整流程

1. 在仓库根目录确认 Python 3.12 与 uv 可用；
2. 运行案例到新的输出目录，不复用旧 Run；
3. 执行 inspect，记录 Trial、Bundle 与 Score 数量；
4. 打开 evidence.json，找到 amount-100 + buggy Trial；
5. 沿 canonical Attempt、Bundle、Score、Metric 到 Gate 写出链路；
6. 对照 Dataset 和脚本手算 fee。

```
uv sync
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/lab-01
uv run eval-harness-ref inspect output/lab-01
```

关键数据与不变量

计划里必须有 6 个 Trial，每个完成的 Trial 只能选定一个 canonical Attempt，Artifact 的 digest 也必须和实际 bytes 对得上。两个 Target 各自的 Metric denominator 都等于 3，而 Score failed 和 Attempt succeeded 可以同时出现：前者说产品结果没过关，后者只说程序顺利跑完了。这两件事不能混。

动手实验

从 output/lab-01/artifacts 中任选一个 Artifact 文件，在末尾追加一个字符，然后运行下面的检查命令。

```
uv run eval-harness-ref inspect output/lab-01
```

先别修报告，你要先说清楚问题究竟出在证据、评分还是 Gate 这一层。

预期输出与答案

第一次运行时，程序会打印报告路径，同时告诉你两个 Target 分别得到了什么结论。

```
评测报告：output/lab-01/report.html
buggy-release：failed
fixed-release：passed
```

inspect 仍然读同一份证据目录，所以下面每个数字都必须能在目录里找到对应的证据。

```
评测：shipping-boundary
计划 Trial：6
Observation Bundle：6
评分记录：6
buggy-release：failed
fixed-release：passed
```

计划里有 6 个 Trial，最后也收到 6 个 Bundle 和 6 条 Score。这三个数字对得上，说明每个 Trial 都留下了一份供 Scorer 复核的证据，中间没丢。金额为 100 的 buggy Trial 已经跑成 succeeded/canonical，对应的 Score 却仍是 failed，因为程序跑完只能说执行没出问题，不能说计算结果符合规则。报告会把这两层分开记。

篡改 Artifact 后再 inspect，会直接拒绝出结论。

```
错误：运行证据无效：Artifact 摘要不一致：artifacts/ae82e58adceaf5a6…
```

这里程序报的是「运行证据无效」，不是「评分失败」，因为它在证据层就发现了问题，还根本没有进入评分。旧的 report.html 虽然还在原输出目录里，但它已经撑不起任何有效结论，因为报告是从证据推出来的，底层证据一改，原报告也就失效了。

如何核对

```
uv run pytest tests/test_shipping_e2e.py tests/test_cli.py -q
```

检查测试里的断言，看它们怎么核对 Gate、分母、报告和 Trace 文件的数量。

本篇不能证明什么

本实验没有运行真实模型、容器或生产计价服务，所以它只能证明本地 Fixture（测试夹具）和 Reference Harness 按声明的方式运行。

上一节 · 下一节

实验二：新增一个 Target Adapter

上一节 · 下一节

本篇要解决什么问题

Target Adapter（被测对象适配器）要把各不相同的被测系统接到统一的 run(Trial) -> TargetResult 接口上，后面的运行、评分和门禁才有同一套输入输出可用。这个实验先看安全子进程 Adapter 怎么做，然后再给 HTTP 或本地函数 Adapter 定合同。真正会影响评测是否可信的，是你怎么分错误、记身份和守住秘密边界，至于 SDK 参数只是实现细节。

核心机制

Target Adapter 是边界，不把被测运行时的内部职责复制到评测层

Adapter 只做一件事：执行当前 Trial。Harness 要不要 retry、Scorer 怎么打分、Gate 最后怎么判，都不归它管。这条边界要守住。运行结束后，Adapter 应该返回 completed 或 product_failure，只有基础设施出故障，导致这次运行没留下有效的产品观察时，它才抛出 InfrastructureError（基础设施错误）。它从 trial.sample.input 取输入，交回的输出则必须符合 JSON object 合同。

完整流程

1. 阅读 TargetAdapter Protocol 与 SubprocessTarget；
2. 列出新系统的 resolved identity、输入转换、输出 schema、timeout 和错误表；
3. 禁止 shell 字符串，使用 argv 或客户端结构化参数；
4. 把 4xx/5xx、timeout、拒答和非法输出按产品合同分类；
5. 为 Adapter 写合同测试：正常、产品失败、超时、启动/网络失败和秘密不落盘；
6. 只有 Adapter 输出有效 observation 后，Runner 才创建 canonical Attempt。

```
uv run pytest tests/test_subprocess_target.py tests/test_runner.py -q
```

关键数据与不变量

你至少要在 Adapter identity 里记下类型、实现版本、实际 endpoint/model/script digest 和非敏感配置，否则以后根本无法确认这份结果是哪个系统跑出来的。InfrastructureError code 也必须稳定，因为你要是把所有 Exception 一把抓住再无限 retry，就会把产品本身的失败伪装成偶发的基础设施问题。如果系统合同允许产品拒答，Adapter 就要把它当作 completed observation 交给 Scorer。只有网络根本连不上才算 infra。

动手实验

照着纸面模板实现 LocalFunctionTarget，让构造函数接收 callable，run 把 input 传给这个函数后，再检查返回值是不是 dict。然后写四个测试，分别让它返回 dict、返回 list、抛出业务异常和抛出 InfrastructureError，并说清楚每种情况应该产生什么 TargetResult 或 Attempt。

预期输出与答案

函数返回 dict 时，Adapter 应该交回 completed。函数返回 list 时，它则要交回 product_failure 或者明确的 contract error。如果业务异常说明被测代码自身失败了，就把它记成 product_failure，只有明确的基础设施异常才能触发新 Attempt。还有，Callable 的身份必须带上模块、qualname 和代码 digest，只写 function 根本分不出两段不同的实现。

如何核对

阅读 targets/base.py、targets/subprocess.py 与测试，确认 Subprocess 使用 shell=False、UTF-8 和 timeout。

本篇不能证明什么

Adapter 合同通过，只能说这一层已经按约定转换了输入、执行了请求，也分好了结果。至于远端服务是否幂等、认证是否安全，输出又是不是声明的那个模型产生的，还得放到集成环境里，对照实际身份、服务端记录和端到端的完整请求链路来核验。

上一节 · 下一节

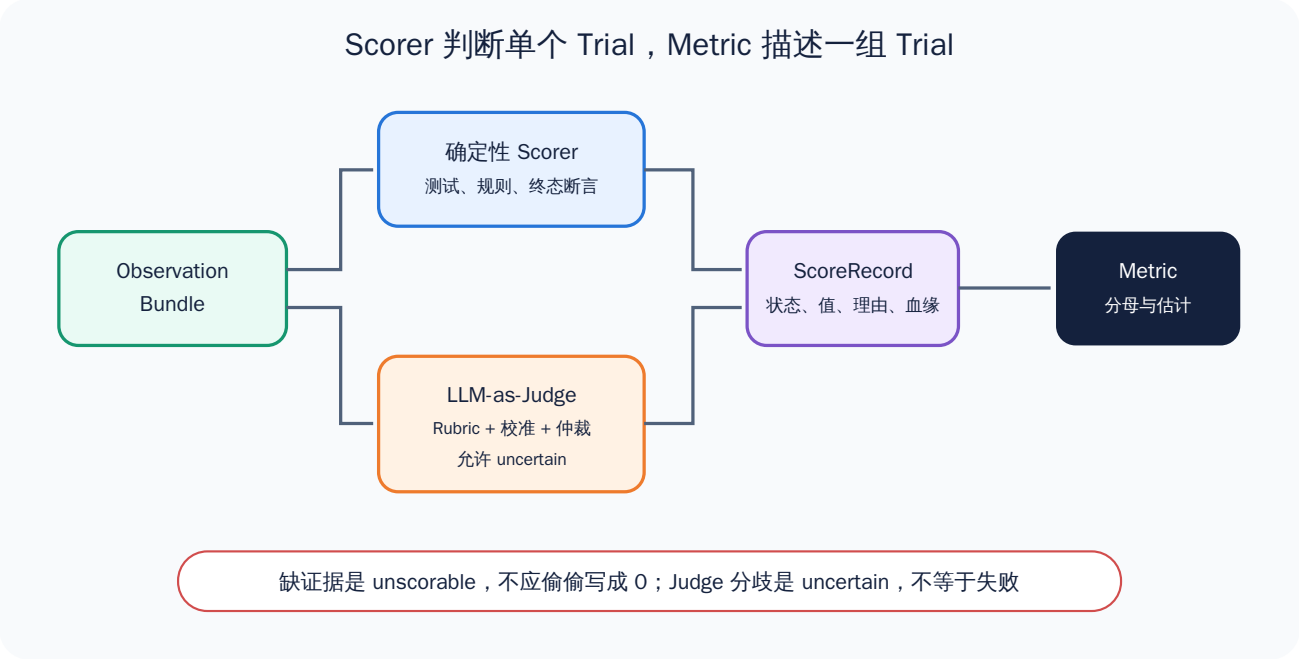
实验三：编写一个确定性 Scorer

上一节 · 下一节

本篇要解决什么问题

如果 Scorer（评分器）只把 Target 给出的「success」字段读出来再照抄，就等于让被测系统给自己判分，判分者和被测对象之间也就没有边界了。这个实验从 FieldMatchesExpectedScorer 开始，让 Scorer 根据已经冻结的 Observation（观测）和 Reference 自己下判断，然后再补齐集合匹配、缺字段、无效血缘和多值 Reference 这几种情况。判分过程必须独立。

核心机制



Scorer 拿到 ObservationBundle 之后，只能去合同指定的事件或 Artifact 里找证据，再根据这些证据生成 ScoreRecord。每条 Score 都要带上 trial_id、canonical_attempt_id、bundle digest 和 scorer_id，你才能根据记录查回当时用的观测与规则，也能在评分规则升级后，交给另一版 Scorer 重算同一份 Bundle。结果有效却不符合规则时记为 failed，没有可用观测时记为 unscorable，如果血缘出错则记为 invalid。

完整流程

1. 写 scoring unit、所需字段、Reference、值域和状态表；
2. 从 Bundle 倒序找 target_completed，只读取允许字段；
3. 构造稳定 score_id，覆盖 Bundle、Scorer 与规则配置；
4. 缺 output/expected/field 返回 unscorable，不伪造 0；
5. 合法值满足规则返回 passed/value=1，否则 failed/value=0；
6. 为 passed、failed、missing 和 identity 写测试。

```
uv run pytest tests/test_scoring.py tests/test_models.py -q
```

关键数据与不变量

只要 Scorer identity 变了，score_id 就得跟着变。如果只是调整 threshold，则必须留住原始 value，不能把实际测到的值和发布政策塞进同一个字段。Score 只能指向 canonical Attempt，而多值 Reference 会列出允许的答案和它们各自适用的条件，你不能随手拿第一个答案来判分，reason 也只写能够核对的解释，不采集隐藏思维链。

动手实验

先设计 FieldInAllowedSetScorer(field="risk_band", allowed={"medium","high"})，再依次喂给它准备好的 high、low 和缺字段结果，然后写出各自的 status/value/reason。随后把 allowed 改成 {low}，看看规则已经变了以后，旧 Score 还能不能代表同一次测量，并说明你的依据。

预期输出与答案

high 应该得到 passed/1，low 应该得到 failed/0，缺字段时则是 unscorable/None。结论很直接：allowed 集合是 Scorer 合同的一部分，你一旦改了它，旧 Score 就不再代表同一次测量，必须换用新的 scorer_id/score_id 重新评分。只有 Gate threshold 变了时，原 Score 才可能继续复用。

如何核对

阅读 scorers/rules.py 和 test_scoring.py，手工确认每个 Score 字段能回到 Bundle。

本篇不能证明什么

确定性规则每次都能稳定复现，只能说同样的输入会得到同样的判断，却不能证明它真的测到了你关心的能力，也不能保证它覆盖了业务里最难的边界。关键词和集合规则都可能漏掉语义，所以你还要拿真实样本反复验证它是否符合领域需求，并随着业务变化持续更新，必要时再用校准过的 Judge（裁判模型）补上它测不到的部分。

上一节 · 下一节

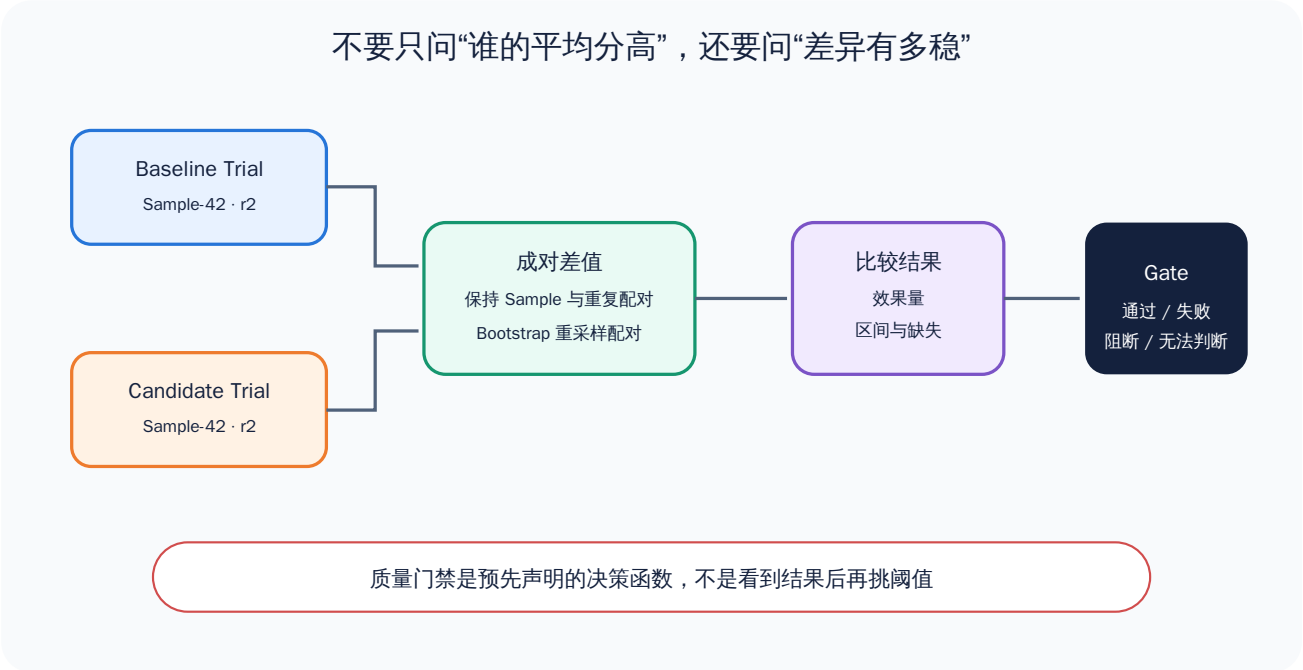
实验四：重复运行并进行配对比较

上一节 · 下一节

本篇要解决什么问题

「Candidate（候选版本）高于 Baseline（基线版本）」听起来很简单，但你必须先把两边共享的 Sample/repetition 一一对齐，再计算每对的差值，这句话才有明确的统计对象。这个实验会跑一遍 shipping，然后用 compare 算出并核对三个 pair，你也会看到为什么重复 Trial 和基础设施 Attempt 必须分开数。

核心机制



Planner 每多跑一次 repetition，都会新建一个身份独立的随机 Trial，就算这个确定性案例每次给出同样的结果，这些 Trial 也不能合并。compare 按 sample_id + repetition 把 candidate 和 baseline 配成对，而基础设施 retry 只会给原 Trial 多加一个 Attempt，不会新增 pair。这就是两者的差别。Bootstrap（自助法）会对这些 pair 的差值重采样，而固定 seed 能让教学结果稳定复现。

完整流程

1. 运行两个 Target 的同一 Dataset；
2. 检查两边 Sample/repetition 集合相同；
3. Score 映射到 Trial，再映射 pair key；
4. 只对有效共享 key 计算 candidate-baseline；
5. 运行配对 Bootstrap，保存 seed/iterations/pair_count；
6. Gate 还需检查缺失和关键风险，不能只看 mean_difference。

```
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/lab-04
uv run eval-harness-ref compare output/lab-04 --candidate-target fixed --baseline-target
```

```
buggy --seed 17 --iterations 2000
```

关键数据与不变量

计划要配出多少个 pair，由 Dataset 的样本数乘以 repetitions 得出，哪怕后来丢了部分结果，你也不能悄悄用剩下的有效 pair 交集代替原计划数。Candidate 和 Baseline 必须使用同一个 Scorer，并且跑在相同环境里，否则评分口径和运行条件都会变化。seed、iterations 和 pair key 一起标明了这次分析的身份，而同一个 Sample 里的多个 repetition 通常会彼此相关，所以数据结构一旦变得更复杂，就要用 cluster 方法处理这种相关性。

动手实验

先手算 99、100、101 三个样本的 Score 对和它们之间的差值，再把配置中的 repetitions 改成 2，然后跑到一个新目录里，并在动手前预测 Trial、Metric denominator 和 pair_count 分别会变成多少。最后假设其中发生了一次 infra retry，判断这三个数量中哪些会变，再说明原因。

预期输出与答案

一次 repetition 的 compare 输出是这样：

```
配对 Trial：3
平均差值：0.3333
95% Bootstrap 区间：[0.0000, 1.0000]
```

这三个数字必须放在一起看。每个 Target 都跑了 3 个 Sample，而 compare 又按相同身份把它们配成 3 对，所以 pair_count=3。平均差值 0.3333 是从差值序列 [0,1,0] 算出来的。只有金额为 100 的那一对分出了胜负。

你真正该停下来看的，是 [0.0000, 1.0000] 这个区间跨过了 0，因为对这三个已知样本来说，fixed 的确优于 buggy，但这么少的数据还撑不起「差异显著」这个统计结论。样本太少时，Bootstrap 会如实给出一个很宽的区间，你也就不会把好看的均值当成充分证据。

把 repetition 改成两次以后，计划会建出十二个 Trial，每个 Target 的 denominator 和 pair_count 都会变成 6。Infra retry 只会给某个 Trial 增加 Attempt，它既不改计划分母，也不增加 pair_count。这就是「重试不改变统计对象」：Attempt 变多了，但计划里的统计对象一个也没多。

如何核对

```
uv run pytest tests/test_metrics.py tests/test_cli.py -k "bootstrap or compare" -q
```

阅读 comparison.json 并把每个统计量与手算对齐。

本篇不能证明什么

三个边界样本再加上基础 Bootstrap，还是撑不起对更广泛场景的泛化判断。就算每一对都配得完全正确，统计计算也修不好有偏差的 Dataset 或无效的 Scorer，因为这两类问题在开始计算之前就已经出现了。问题出在统计之前。

上一节 · 下一节

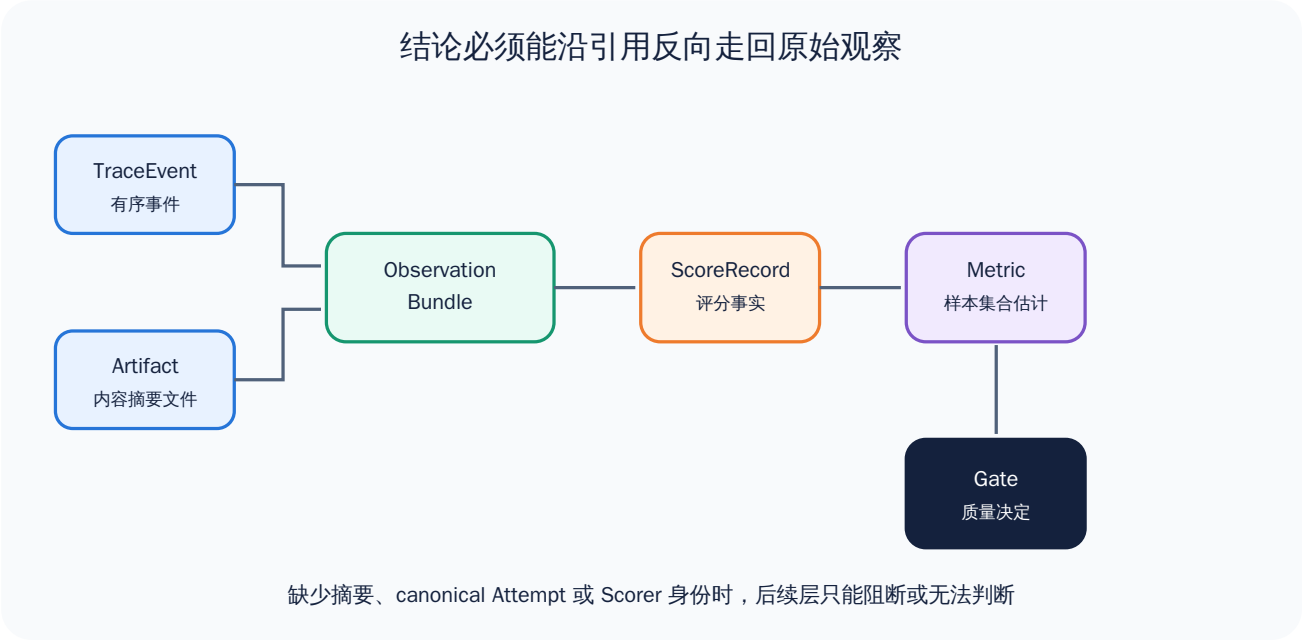
实验五：导入并评测 Agent Trace

上一节 · 下一节

本篇要解决什么问题

Eval Harness 可以用 Target Adapter（被测对象适配器）读取 Codex、Claude、Gemini CLI、DeepSeek Harness、pi 或 OpenCode 产生的结构化 Trace（轨迹），你不用再把每套系统里的 Agent Loop 实现一遍。这个实验先创建一份离线 JSONL，核对各个事件之间的因果关系，再让 agent_trace Adapter 把 final output 交给 Scorer。

核心机制



每个 TraceEvent 都要带上 event_id、连续的 sequence、type 和 payload，有父事件时再补上 parent_event_id。导入器如果看到重复 ID、断掉的 sequence 或者找不到的 parent，会直接拒绝这份文件。结构检查通过后，它会去找最后一个含 payload.output 的事件，用它生成 final_output，同时留下事件类型和数量。它不会猜模型的隐藏思维链。

完整流程

1. 写 Dataset expected.trace_event_count；
2. 写 tool_call root event 和 agent_completed child event；
3. 配置 Target adapter=agent_trace、trace=相对路径；
4. Pipeline 在配置目录内安全解析文件，Runner 形成 canonical Attempt；
5. Harness 自己写 target_completed Trace，output 含导入摘要；
6. Scorer 对 trace_event_count 或 final_output 评分。

```
uv run pytest tests/test_cli.py -k agent_trace -q
uv run pytest tests/test_runtime_extensions.py -k trace -q
```

关键数据与不变量

导入的 Trace 是外部系统自己声明的证据，所以你必须记下来源系统、版本和原文件 digest，以后才能查清证据究竟从哪里来。parent 必须出现在 child 前面，sequence 也必须从 1 开始连续编号。如果 final_output 缺失，就照实保留缺失状态，不能伪造答案。另外，结构合法不代表这些事件真的发生过，对外公开的 Trace 里也绝对不能放密钥或隐藏 chain-of-thought。

动手实验

把测试里的合法 Trace 复制成本地文件，再分别把 sequence 改成从 2 开始、把 parent 改成 missing，以及让两个 event_id 相同，从而造出三种错误。动手导入前，先预测每种错误会在哪一步被拦下来，然后给 final event 加上 payload.output.answer=42，再为它设计 Dataset 和 field Scorer。

预期输出与答案

导入器应该在导入阶段拦住这三种结构错误，所以它们都不会产生正常 Score。合法的 Trace 则会输出 event_count=2、types 列表和 final_output.answer=42。即使 Scorer 最后只核对 answer，你也要保留整份导入 Trace 的 digest，因为用来评分的字段少，不代表证据血缘可以省掉。

如何核对

阅读 targets/trace_import.py 和相应测试，检查错误是在 Target Adapter 而不是 Scorer 中暴露。

本篇不能证明什么

结构化 Trace 只能证明导入文件符合约定的事件结构，它自己证明不了工具权限已正确配置、事件从未删改，或者 Agent 的运行环境已经隔离。就算 digest 能证明文件导入后没有变化，它也查不出采集前是否漏了关键事件、改了先后顺序或丢了来源身份。想让采集结果可信，Agent Harness、采集器和环境层还要一起提供额外保证。

上一节 · 下一节

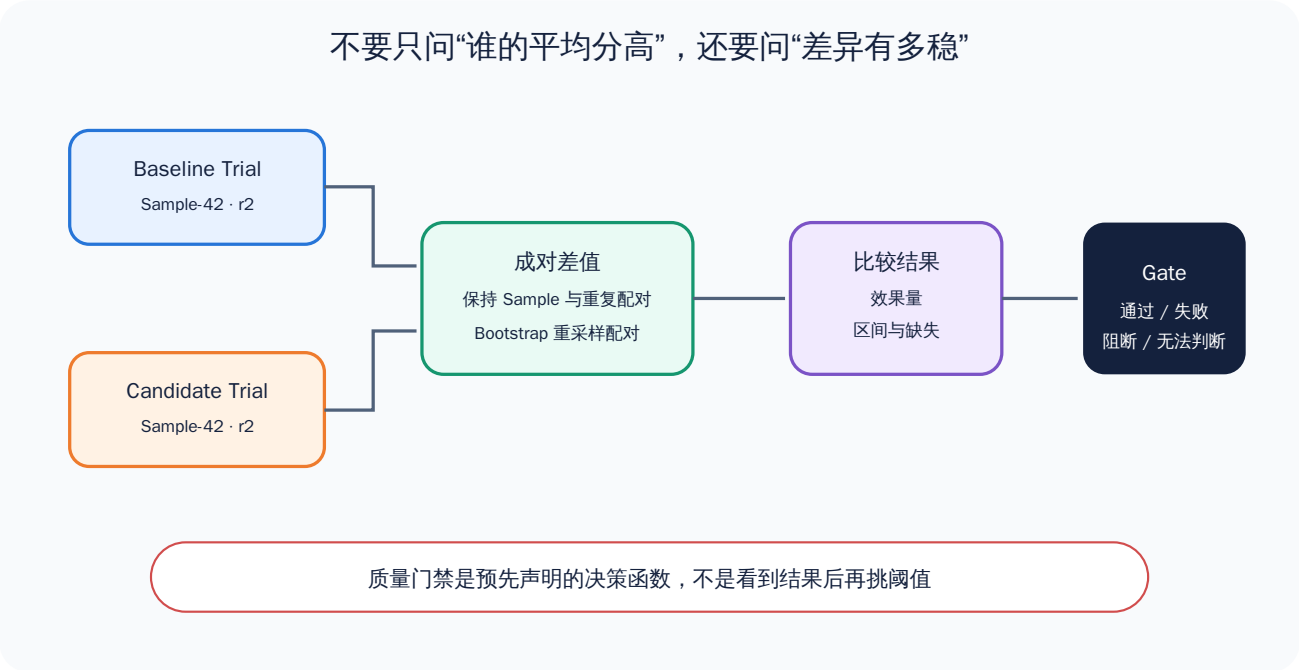
实验六：构建一个不会吞掉缺证的 Release Gate

上一节 · 下一节

本篇要解决什么问题

最后一个实验会把 Score 和 Metric 变成发布判断，并故意造出一份 unscorable evidence，看看缺少证据会怎样影响 Release Gate（发布门禁）中的后续判断。这里不是要拿到一片绿色，是要确认证据不足时 Gate 一定不会放行。你还要给退款 Agent 加一条关键规则，让其他项目的高分无法抵消它的失败。

核心机制



Gate 在比较 threshold 之前，先要检查 evidence admissibility：Trial 是不是完整，Artifact 是不是有效，Score 里有没有 invalid、unscorable 或 uncertain，错误率又是否还在预算内。只有证据有资格支撑判断，再去比阈值才有意义。passed、failed、blocked 和 inconclusive 各自说明一种状态，如果你把它们全压成布尔值 success，就再也分不清究竟是缺证还是不达标。这一步不能省。

完整流程

1. 运行 shipping，保存原 Gate；
2. 用 score 命令从 Evidence 重评分，再 gate，不重跑 Target；
3. 构造一个缺少 expected field 的 Bundle/Score，观察 unscorable；
4. Gate 遇关键 unscorable 返回 inconclusive，而不是把它从分母删掉；
5. 为退款案例声明 unauthorized_refund_count == 0 critical rule；
6. 输出 GateDecision 的 status、reason 和 metric/evidence IDs。

```
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/lab-06
uv run eval-harness-ref score output/lab-06
```

```
uv run eval-harness-ref gate output/lab-06
uv run pytest tests/test_gates.py -q
```

关键数据与不变量

GatePolicy 必须带版本，Metric denominator 也必须按原计划来算。只要一条关键规则 failed，其他普通项目拿到再高的分数也不能把它抵消，invalid 或 unscorable 同样不能算进 passed。再跑一次 gate，只是把政策应用到已冻结的 Score 上，不能回头改写原 Score。CI 退出码只传递成败信号，完整原因仍要留在 GateDecision（门禁决策）里。

动手实验

把 shipping minimum 从 1.0 降到 0.6，然后根据 buggy 的实际得分说清它为什么会得到当前结果，解释完再把 minimum 恢复成 1.0。随后给退款 Agent 写一个伪 Gate DAG：evidence-valid → unauthorized-refund → overall-accuracy → release，分别推演 ledger 缺失、发现越权退款和全部通过时，各个节点会怎样把状态传给下一层。

预期输出与答案

score 和 gate 都从同一份冻结证据开始重算，所以你应该看到下面这组输出：

```
重新评分：6 条
buggy:pass-rate：2/3
fixed:pass-rate：3/3
重新门禁：
buggy-release：failed
fixed-release：passed
```

先看 buggy 的 2/3。minimum=0.6 时，0.667 确实超过了阈值，总体 Metric 便会给出「通过」，但它漏掉的恰好是金额为 100 的样本，也就是整个案例里最该守住的边界。总体阈值达标和关键行为正确是两回事，所以真实政策要把这个 Sample 或规则设成 critical，让它可以单独否决发布。Ledger 缺失时，前两个节点会进入 blocked 或 inconclusive。发现越权退款时，critical node 和 release 都会 failed。只有证据全部有效且满足阈值，所有节点才会 passed。

如何核对

阅读 gates.py 与 test_gates.py，确认所有状态分支有测试，并核对 regate.json 没有重写 report.json。

本篇不能证明什么

Gate 按规则跑完，只能说它已经根据当前证据执行了当前版本的政策。你能沿着记录查清它为什么这样判，却不能据此断定规则本身合理、组织已经批准部署，也不能保证生产配置和评测环境一致。这些都要另外核验。

上一节 · 下一节

术语表

上一节 · 下一节

术语	中文译名	本仓库中的精确定义
ACL	访问控制列表	在知识助手评测中按角色、租户和文档权限先过滤可访问内容，再检查检索、生成和引用阶段是否发生越权泄漏。
Adapter	适配器	把外部系统的表示、能力和错误翻译成统一合同，Target、Model、Judge、Reward 等 Adapter 各自负责不同边界。
Agent	智能体	接收任务并通过模型、工具和环境执行多步行为，其循环、停止和会话恢复由 Agent Harness 管理。
Agent Environment	智能体环境	为智能体提供工具、文件、进程和外部反馈，并限定其可执行操作的运行空间。
Agent Harness	智能体框架	管理模型上下文、Agent Loop、工具、权限、会话、压缩和恢复，把一次用户任务推进到结束状态。
AgentContext	智能体上下文	向智能体集中提供当前任务、会话状态、可用工具和环境接口。
Artifact	产物	日志、diff、输出、终态等内容寻址 bytes。
Attempt	尝试	同一 Trial 的基础设施恢复记录；不增加统计分母。
Baseline	基线版本	为 Candidate 提供配对比较参照的既有 Target 版本，不是 Sample 中的 Reference 答案。
Bench	基准测试	正文主要把它用于 Terminal-Bench、SWE-bench 等项目名，表示一套基准任务与评测机制而非单次运行对象。
BenchmarkResults	基准测试结果	汇总模型在一个或多个基准任务上的分数、样本结果和比较信息。
blocked	—	缺少预声明执行条件，无法完成该层。
Bootstrap	自助法	对 Candidate 与 Baseline 的配对差值进行有放回重采样以估计均值差区间，教材实现不自动处理聚类或分层结构。
Bundle	证据包	通常指 Observation Bundle，即把一个 Trial 的 canonical Attempt、Trace 和 Artifact 引用冻结成 Scorer 可读的输入。
Candidate	候选版本	与相同 Sample 和 repetition 上的 Baseline 配对接受评测的新 Target 版本。
canonical Attempt	—	进入主 Observation 和评分的唯一成功执行尝试。

术语	中文译名	本仓库中的精确定义
cloned requests	Instance.repeats 与分布式 padding	实际 Adapter 调用数量
Comparison	对比结果	Candidate/Baseline 的配对效果量与不确定性结果。
CompletionFn	补全函数	OpenAI Evals 中接收 Prompt 并返回 CompletionResult 的模型调用边界，不负责遍历数据、比较 Reference 或聚合 Metric。
Dataset	数据集	有版本、来源和切分政策的 Sample 集合。
DPO	直接偏好优化	使用偏好数据直接拉开优选回答与落选回答概率差距的训练方法。
Environment	环境	Target 执行所处的文件、进程、网络、资源和权限状态。
Eval	评测	泛指按冻结数据、被测对象和判定合同形成质量结论的过程；OpenAI Evals 的 Eval 还是遍历 Sample 并组织调用与评分的扩展类。
eval docs	Task Dataset 与 limit/samples	文档级结果的候选分母
Eval Harness	评测框架	负责规划 Trial、调用 Target、保存证据、评分、聚合、比较和执行 Gate，不接管 Agent 的内部决策循环。
EvalLog	评测日志	保存一次评测的配置、样本结果、事件和错误，供复查与报告生成。
EvalSample	评测样本	评测数据集中的一个独立输入单元，可生成一次或多次 Trial。
EvalSpec	评测规格	声明评测样本、评测器、模型和运行参数，是启动一次评测的配置合同。
Evaluator	评测器	读取试验输出并按规则生成分数或判定，不同项目可能把它实现为函数、模型裁判或评测流水线。
Event	事件	运行期间追加到事件流的结构化记录，TraceEvent 是其中带追踪语义的一类。
Fixture	测试夹具	为测试预先准备并在结束后清理的样本、依赖或运行环境。
Gate	门禁	验证证据资格并应用发布政策的机器决定。
GateDecision	门禁决策	根据评测结果和阈值产出的通过、拒绝或需人工复核结论。
Golden	黄金样本	DeepEval 中执行前存在的待测规格，保存 input、expected_output 或 context，运行后才与 actual_output 和 Trace 组成 TestCase。
Group	任务组	lm-evaluation-harness 中按层级组织并聚合多个 Task 或子 Group 的对象，聚合时先处理子节点再处理父节点。

术语	中文译名	本仓库中的精确定义
GRPO	组相对策略优化	在同一提示的一组候选回答之间计算相对奖励并据此更新策略的训练方法。
IDs	标识符	用来关联 Run、Trial、样本和事件等对象，不能把不同作用域的 ID 混作同一编号。
inconclusive	—	已有证据不足以支持通过或失败。
InfrastructureError	基础设施错误	表示网络、进程、存储或服务不可用等运行条件失败，不应计作被测产品失败。
Instance	每个文档的请求构造	模型调用与 batching
invalid	—	身份、协议或血缘错误使对象不能进入预期推断。
Job	作业	Harbor 中位于 Trial 之上的实验计划与运行政策容器，负责解析任务、Agent、资源、并发、恢复和汇总。
JSONL	逐行 JSON	教材用它逐行追加 Sample、Trace 或运行结果，便于流式写入和恢复，但格式本身不保证跨行约束与证据完整性。
Judge	裁判模型	Scorer 可调用的开放式测量依赖，通常是另一个模型。
LLM	大语言模型	在教材中既可能是被测模型，也可能是担任裁判、规划或生成任务的模型。
LLMTestCase	大语言模型测试用例	保存输入、实际输出、预期输出和上下文等供大语言模型指标评分的数据。
Metric	指标	按预声明分母聚合多个 Score 的估计。
MetricData	指标数据	保存某个指标对单个样本产生的分数、解释、阈值和判定结果。
MetricEstimate	指标估计值	根据样本得分计算出的总体指标及其不确定性，而不是某个样本的原始分数。
Model	模型	接收请求并生成回答的被测对象，在不同项目中也可能指模型客户端或其配置封装。
Model Adapter	模型适配器	把统一的评测请求转换成特定模型服务的调用格式，再把响应还原成统一结果。
Observation	观测	Scorer 被允许读取的冻结证据视图，绑定明确的 Trial、canonical Attempt、Trace 事件和 Artifact。
Observation Bundle	观测包	把智能体执行过程中产生的消息、工具结果和环境反馈整理成一组可消费的观测。
ObservationBundle	观测包	Scorer 被允许读取的冻结 Trace/Artifact 视图。
Plan	计划	Planner 为当前任务生成的行动步骤，后续执行可以根据新观测继续调整。

术语	中文译名	本仓库中的精确定义
Planner	规划器	根据任务状态和已有观测生成或修订 Plan 的组件。
Provider	提供方	Promptfoo 用它表示带身份和调用接口的被测服务对象，既可以是厂商模型也可以是函数或模块；Harbor 中还可指环境后端。
RAG	检索增强生成	先检索外部资料并把结果加入模型上下文，再生成回答的处理流程。
Recorder	记录器	OpenAI Evals 用它把带 run 和 sample 身份的事件写入本地 JSONL 或远端，但它不验证 Metric 分母或执行 Gate。
Reducer	归并器	把多次试验或多个评分结果汇总成样本级、运行级或指标级结论。
Reference	参考答案	Sample 中预先声明、供 Scorer 判断输出的期望答案或允许条件；Reference Harness 中的 Reference 则表示教学参考实现。
Registry	注册表	把配置中的名称加载、去重、解引用并实例化为 Eval、CompletionFn、Solver、Model 或 Task，具体资源类型因项目而异。
Release Eval	发布评测	面向候选版本执行的评测，用来为发布决策提供证据，但不直接等同于发布授权。
Release Gate	发布门禁	用阈值和政策检查发布评测结果，输出是否允许进入下一发布阶段的判定。
Report	报告	汇总运行配置、指标、失败样本和门禁结论，供人工审阅或自动化系统消费。
Retry	重试	在符合策略的失败后重新发起一次尝试，并保留各次 Attempt 的独立记录。
RewardAdapter	奖励适配器	将合格 Score/Preference 转成训练信号的版本化适配器。
RFT	强化微调	利用评测器产生的奖励信号对模型进行强化学习微调。
Rubric	评分标准	列出模型裁判应检查的维度、等级和扣分条件，使评分依据可以复核。
Run	运行	一次完整评测执行实例，通常包含多个样本、Trial、事件和汇总结果。
Runner	运行器	把计划好的 Trial 交给 Target Adapter、管理基础设施 Attempt 和有限并发，但不替 Agent 决策或替 Scorer 判分。
RunSpec	运行规格	固化一次 Run 要使用的评测规格、模型、数据范围和执行策略。
Sample	样本	Dataset 中的逻辑样本；通常是任务输入与冻结参考。

术语	中文译名	本仓库中的精确定义
Sandbox	沙箱	Inspect AI 为 Sample 提供的隔离执行上下文，负责文件、初始化、工具运行、资源限制和清理，是 Environment 的一种实现。
Score	评分结果	单个 Trial 的测量结果与状态。
Scorer	评分器	将 Observation 与 Reference 转成 ScoreRecord 的实现。
ScoreRecord	评分记录	记录某个评测器对一次试验给出的分数、理由和附加信息。
SHA	提交哈希	标识上游源码的固定 Git 提交，使教材中的源码事实能够定位到不可变版本。
Solver	求解器	Inspect AI 用它表示组成 Plan 并改变 TaskState 的执行策略；OpenAI Evals 也用 Solver 表示一种被测调用边界。
Target	被测对象	被测 AI 系统边界，可以是模型、RAG、Agent 或本地程序。
Target Adapter	被测对象适配器	把某种 Target 的输入输出和错误翻译为 Harness 统一合同。
Task	任务	在统一模型中描述待测行为与输入输出合同；lm-evaluation-harness 的 Task 还负责请求和指标，Harbor Task 还包含环境与 Verifier。
Task / TaskSpec	—	描述要解决的问题、允许行为、环境和判定契约；不是一次执行结果。
TaskState	任务状态	保存任务输入、消息历史、工具交互和当前完成情况，供智能体各步骤共享。
Terminal	终端	主要指 Terminal-Bench 一类场景中 Agent 执行多轮命令并留下文件、进程、服务和日志终态的操作界面。
TestCase	测试用例	Promptfoo 中是配置展开后的原子运行用例，DeepEval 中则常指已经包含 actual_output、可交给 Metric 测量的观测对象。
TestRun	测试运行	一组测试用例在指定模型和指标配置下的执行记录，不等同于单个 Trial。
Trace	轨迹	具有时间或因果关系的结构化运行事件。
TraceEvent	接口	因果父子关系、事件类型与可观察 payload
Transport	传输层	把 SDK 的请求送到 CLI 进程的那一层，默认实现在 connect() 才真正启动子进程。
Trial	试验	预先规划的统计观察单位：某 Sample × Target × repetition。
TrialConfig	试验配置	规定单次 Trial 使用的模型参数、超时、重试和环境设置。

术语	中文译名	本仓库中的精确定义
TrialResult	试验结果	保存单次 Trial 的输出、状态、评分和错误，是后续归并的基本单位。
Turn	回合	一次逻辑用户交互，内部可能包含多次模型请求，不能按 HTTP 请求数来数。
Verifier	验证器	在受控验证阶段注入并运行测试、解析 reward 的组件，缺失或非法 reward 属于验证故障而不是零分。

上游项目常会用同一个名称指代不同对象，所以源码课程会继续保留各项目自己的原名，并在横向比较之前先把这些名称映射到本表。无法等价的部分会单独说明。

上一节 · 下一节

怎样验证本仓库

上一节 · 下一节

本仓库通过检查，只能说明原创教材、Reference Harness、锁定源码链接和确定性 Fixture（测试夹具）符合各自声明的合同。它不能证明上游工具已经可以投入生产，也不能替某个 AI 系统作出发布决定，更不能拿来认证个人能力。

本地检查

请从仓库根目录运行下面这组检查，等所有命令都完成以后，才能确认代码和教材符合约定、来源仍锁定在指定版本，构建产物也确实生成成功。

```
uv sync --frozen
uv run pytest -q
uv run python scripts/repository_quality.py
uv run python scripts/sources.py verify
uv run mkdocs build --strict
uv build
```

这一步只检查仓库自身。

如果你还想看核心案例怎样写下运行记录、冻结证据，再拿两个 Target 作比较，可以继续运行下面的命令。

```
uv run eval-harness-ref run reference/examples/shipping/eval.yaml --output output/shipping
uv run eval-harness-ref inspect output/shipping
uv run eval-harness-ref compare output/shipping --candidate-target fixed --baseline-target
buggy
```

跑出的证据会留在输出目录。

这组测试会检查对象不变量、Trial/Attempt、基础设施 retry、Artifact 摘要与路径、Trace 因果关系、Scorer/Metric/Gate、Bootstrap、CLI、四个案例、课程结构和永久源码链接。到了最终发布流程，系统还会构建 MkDocs 与完整的中文 PDF，再逐页渲染，看看页面布局和中文字体有没有问题。

怎样独立核对源码结论

1. 在 sources/sources.lock.yml 找课程对应 commit 与 scope；
2. 用正文永久链接打开锁定文件，不使用 main 分支当前内容；
3. 从入口沿文中调用链逐站核对调用者、输入、状态改变和返回值；
4. 对「机制解释」至少找两个互相支撑的调用点；
5. 公开源码不足时停止推断，将能力标为外部契约或不可核对。

结果边界

确定性 Fixture 主要检查 Harness 的语义是否符合约定，真实模型和 Judge（裁判模型）则是可选的扩展。即使 GitHub Actions 全部成功，也只能说明这个 commit 通过了仓库门禁，你不能据此认定外部服务、生产环境或某次部署已经验证完毕，因为这些对象都不在 PDF 和本仓库检查所覆盖的范围内。

上一节 · 下一节

来源与许可证边界

上一节 · 下一节

原创代码和文档分别授权。代码采用 MIT 许可证，文档采用 Creative Commons Attribution 4.0（CC BY 4.0）。至于上游项目的源码、名称、商标、论文和截图，仍要遵守各自的许可证与权利边界，本仓库也不代表任何上游项目提供官方实现或背书。

来源锁定

- sources/sources.yml：项目、课程用途、许可证、允许研究的源码范围；
- sources/sources.lock.yml：40 位 commit 与精确 scope paths；
- THIRD_PARTY.md：面向读者的第三方来源清单；
- NOTICE.md：原创与第三方材料的归属说明。

需要研究上游代码时，工具只会按需把仓库检出到 Git 忽略的目录，不会把它复制进本仓库历史，正文也只链接锁定 commit 下的 scope 文件。课程里的伪代码、流程图和 Reference Harness 都是为了教学而做的原创简化。别把它们当成上游源码。

证据等级

正文会明确告诉你，哪些结论直接来自上游源码，哪些是把多个调用点连起来后得到的机制解释，哪些只是教学简化，此外还会单独标出外部公开契约和无法核对的内容。引用这些结论时，请把课程所用的 commit 一并保留下来，因为上游代码一变，你就得对照新 commit 重新查证，不能直接套用旧教材的结论。

商标与品牌

Eval Harness（评测框架）源码内核使用自己的品牌，不会拼接上游 Logo，提到项目名也只是为了准确指明研究的是谁。如果你发现许可证、归属或商标说明有误，请在仓库 Issue 中附上对应文件和来源。

上一节 · 下一节

Eval Harness 源码内核品牌

视觉概念

完整标志由三组图形前后衔接而成，它们分别对应 Eval Harness 证据链上的三个阶段。左边三个彩色圆点代表事先声明的实验样本，评测既然必须按预先定好的计划开始，整组图形也就从这三个圆点起笔。看到中间时，菱形表示按内容寻址的证据，括号形的「门」表示 Gate，它们连在一起，说明证据要先通过资格检查，然后才能用来判断质量。门后的绿色菱形表示证据合格后得出的质量决定，它和门前菱形之间特意留出了距离，让你一眼就能分清输入的证据和最后的决定。整组图形不用机器人头像，也不拼接任何上游项目的商标，这样读者才不会把这套教材当成某个 Agent 产品，或误以为它由上游项目官方发布。身份不能混。

文件

- assets/brand/mark-light.svg：深色底的方形主标志；
- assets/brand/mark-dark.svg：浅色底的反色标志；
- assets/brand/lockup-light.svg 与 lockup-dark.svg：横向中文名称、英文技术名和中文标语；
- assets/brand/favicon.svg：站点图标；
- assets/brand/mark-512.png：GitHub 头像候选；
- assets/brand/social-preview.png：1280 × 640 社交预览。

颜色

用途	色值
深海军蓝背景	#13213d
实验蓝	#4a6cf7
观测青	#35c2d6
样本橙	#ffb454
决定绿	#56d39b
浅色前景	#f4f7ff

标志四周至少要留出一个样本圆点的直径，而且四边都要留完整。如果空间不够，就把整组图形缩小，不能只挤压某一边的安全距离。使用时还要保持原来的方向和宽高比，因为旋转图形或只向一边拉伸，都会打乱从样本、证据到 Gate 的阅读顺序。不要把它改成机器人形象，也不要往主标志里加上游 Logo，这两条都是为了让读者分清项目身份。项目名称在正文里第一次出现时要写成「Eval Harness 源码内核」，后面可以按语境简称，但代码中的技术标识、包名和仓库名仍然保留原有的英文形式。