

OpenSquilla: Token-Efficient Agent = Models + Routing Harness

基元律动科技有限公司

<https://github.com/opensquilla/opensquilla>

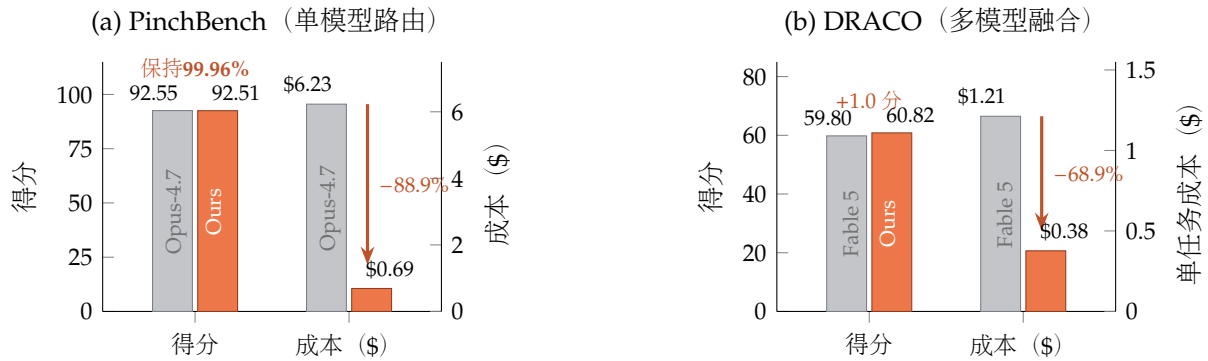


Figure 1 | **OPENSQUILLA**的核心结果。(a) PinchBench: 灰色柱为OpenClaw 固定旗舰Opus-4.7 直跑, 橙色柱为OPENSQUILLA多档路由配置; 质量保持99.96% (92.51 vs 92.55), 成本下降88.9% (\$0.69 vs \$6.23)。(b) DRACO: 灰色柱为最强单模型Fable 5, 橙色柱为本文多模型融合配置; 质量超出基线1.0分 (60.82 vs 59.80), 单任务成本下降68.9% (\$0.3766 vs \$1.2122)。每个子图左轴为任务得分、右轴为成本。

Abstract

OpenSquilla 是一个开源的Agent 原生可学习Harness, 其出发点可以概括为Token-Efficient Agent = Models + Routing Harness: Agent 能力来自对一池异构模型的调度分配, 而非绑定单一强模型。在端到端Agent 基准上, 其本地步级路由在保持固定旗舰模型99.96% 任务质量的同时将成本降低88.9%; 多模型融合路由在深度研究任务上以最强单模型基线 (Fable 5) 31% 的成本取得更高质量。这些能力构建在微内核式设计之上: 多模型池、工具、记忆、技能、通道、调度与沙箱策略均可插拔式接入, 从而将稳定执行机制与可演化策略分离。OpenSquilla 具备三项核心竞争优势: 1、单模型路由节省成本: 将模型选择形式化为以harness 状态为条件的步级路由算子, 在省钱模式下为每个执行步派出能力足够且最便宜的模型, 并与推理预算、提示缓存与技能按需注入等运行时机制联动, 在几乎不损失任务质量的前提下成倍降低端到端成本。2、多模型融合路由提升效果: 在高精度模式下装配互补的多模型组合并融合其候选输出, 以低于强单模型的成本取得更高任务效果, 在深度研究类任务上同时推进质量与成本前沿。3、可学习Harness 的系统机制: Meta-Skill 编排层让Agent 学会检索、筛选、组合并演化技能, 并把用户历史协作沉淀为可审查的新 workflow; 持久记忆系统为跨会话协作维护可编辑、可检索、可遗忘、可审计的长期上下文; 安全治理以分级策略、运行时隔离与审计闭环为真实工具执行划定边界。三者分别提供能力编排、跨会话状态与执行边界, 使系统在“越用越好”的同时保持可审查与可控。对比分析表明, OpenSquilla 的价值不在于

堆叠单一大模型，而在于以agentic routing（单模型路由省成本、多模型融合路由提效果）、Meta-Skill 编排、长期记忆与受控工具边界提升单位成本可获得的Agent 能力。

1. 引言

随着大语言模型（LLM）能力的快速提升 [27]，基于LLM 的智能体（Agent）已在代码生成、知识问答、任务自动化等领域展现出巨大潜力。为了将Agent 从实验性原型推向大规模生产落地，学术界与开源社区涌现出了LangChain [5]、OpenClaw [33]、Hermes Agent [25] 等一系列Agent 框架与运行时系统。然而，在构建面向高频交互、长周期协作与真实工具执行的生产级Agent 应用时，现有框架普遍面临三类核心挑战：

成本失控。 复杂任务通常需要多轮动态规划与长上下文交互，导致Token 消耗随任务复杂度呈阶梯式增长。若系统长期绑定单一强模型，简单确认、格式转换、短文本检索等低难度请求也会支付旗舰模型价格；若系统只依赖静态规则或关键词分流，则又容易把复杂任务误下沉到弱模型而造成失败重试。缺乏本地化、可审计、可退化的模型路由机制，是Agent 成本难以随任务难度自适应伸缩的关键原因。

单模型效果天花板。 随着模型日益专长化——编码、长上下文恢复、推理、工具调用与低延迟响应各有所长——即便预算充足，单一强模型在证据检索、分析综合、代码修改等不同执行状态上也并非处处最优，任务效果被最强单模型的能力所封顶。仅靠换用更强的模型无法利用模型池的互补强项：现有框架普遍缺少在关键状态派出多个互补模型、并对其候选输出进行验证与融合的系统机制。

Harness 难以学习与治理。 真实用户任务跨越检索、分析、工具调用、澄清、审校与最终综合等多个阶段，但传统Skill 只封装单项能力，依赖模型临场规划难以稳定复用成功路径、也难以从历史协作中沉淀新的 workflow；多数框架的记忆机制局限于会话上下文堆叠或全量向量库，难以区分长期事实、短期任务、审计证据与过期噪声，还可能把历史噪声和潜在prompt injection 长期带入未来上下文；同时，Agent 执行文件读写、Shell 命令与网络请求时，风险已从文本层面的不当回答扩展为对宿主环境的真实状态改变，仅靠模型自我约束或一次性审批难以覆盖工具返回注入、记忆污染与连续试探等复合攻击路径，完全依赖重型容器又会增加部署复杂度。换言之，Harness 需要在能力编排、跨会话记忆与运行时安全边界三个维度上同时做到可学习、可审计、可治理。

如果把这一问题放在更宏观的AI 系统演进中看，2023–2025 年的主导范式是围绕数据集、预训练算力与Scaling Law 优化模型参数，进而形成一组能力强、成本结构各异的基础模型池 $\mathcal{M} = \{M_1, \dots, M_n\}$ ，例如GPT [32]、Claude [2]、GLM [43]、Kimi [34]、DeepSeek [38] 等。进入Agent 时代后，系统价值不再只由单个模型的榜单能力决定，而取决于能否把多个基础模型、工具、记忆、技能与安全边界组织成稳定的任务执行系统。换言之，Agent 的基本形态可以抽象为

$$\text{Agent} \triangleq \mathcal{M} + \text{Harness}. \quad (1)$$

在式 (1) 所刻画的范式下，Agent 若要真正辅助人甚至替代人完成部分工作，价值函数应同时考虑任务价值、成功率与Token 性价比。本文将其实表述为提升每Token 智能（Intelligence/Token）的Harness 优化问题：

$$\begin{aligned} \min_{s,g} \quad & \mathcal{L}_T(a) P(a) \\ \text{s.t.} \quad & a = \text{Agent}(g_1(\mathcal{M}) \circ s_1, \dots, g_t(\mathcal{M}) \circ s_t), \end{aligned} \quad (2)$$

本报告的agentic routing 部分 (Section 2) 曾以独立论文 *Agentic Routing: The Harness-Native Data Flywheel* 的形式发布；本报告将该工作纳入并扩展为完整的OPENSQUILLA系统。

其中 $\mathcal{L}_T(a)$ 与 $P(a)$ 分别表示Agent在任务 T 上归一化的损失与归一化的执行价格，二者相乘是一种粗略的性价比刻画——损失越低、价格越低则目标越优； g_t 是第 t 个子任务上的模型gate，负责从模型池 $\mathcal{M}$ 中选择合适模型； s_t 则是Harness对所选模型的操作，包括prompt组织、检索增强生成（RAG）、Skill注入、记忆召回、工具策略与安全约束等。这里 $\circ$ 表示模型选择 $g_t(\mathcal{M})$ 与Harness操作 s_t 的组合，而非数值相乘。

在此框架下，前述三类挑战分别对应目标式中的关键失效模式：成本失控意味着 g_t 恒取最强模型， $P(a)$ 无法随任务难度伸缩；单模型效果天花板意味着 g_t 每步只派出单个模型， $\mathcal{L}_T(a)$ 被最强单模型的能力所限；Harness难以学习与治理则意味着 s_t 仍停留在静态prompt或孤立Skill、缺少跨会话状态，且 $g_t \circ s_t$ 的可执行边界不清晰。

为破解上述痛点，本文设计并实现了OPENSQUILLA——一个Agent原生的可学习Harness。这里的Harness不只是模型调用wrapper，而是承载任务输入、模型路由、Skill编排、工具执行、状态持久化、权限审批与评测可观测性的系统级脚手架：每个子任务 t 上， g_t 选择模型， s_t 调整prompt、RAG、技能、记忆与工具边界。其核心设计借鉴现代操作系统的“机制与策略分离”原则：核心引擎只保留turn生命周期、状态机推进、事件流、工具边界和最终化语义；而多模型池、工具集、记忆库、消息通道、安全沙箱、Meta-Skill与调度器等外延能力，则通过标准化registry、adapter或service接入。该架构既提升了扩展能力，也为故障隔离、能力演化、权限控制和端到端可观测性提供了清晰边界。

从技术特性的角度来看，OpenSquilla的主要贡献包括以下三个方面：

- 单模型路由节省成本：将模型选择形式化为以harness状态为条件的步进路由算子，统一刻画Agent执行的质量-成本Pareto前沿；单模型路由（ $k=1$ ）按能力匹配为每个执行步派出能力足够且最便宜的模型，在几乎不损失质量的前提下大幅降低成本。开源冷启动实现SquillaRouter结合四档文本模型类型、推理预算与提示词策略，在本地完成混合特征分类路由，不需要额外远端分类调用；同时通过Prompt Cache连续性、稳定/动态提示分段与技能按需加载，降低重复前缀与无关技能说明带来的token开销。
- 多模型融合路由提升效果：在同一前沿的高精度区间（ $k>1$ ），路由器基于任务画像装配互补的proposer集合，并由状态感知聚合器融合候选输出；互补性正则项引导集合的失效模式去相关，使更便宜模型的组合能以低于强单模型的成本取得更高任务效果，并在例行状态自动退化为单模型路由。
- 可学习Harness的系统机制——Meta-Skill编排、持久记忆与安全治理：在编排层，于社区与本地多来源Skills之上引入协议化编排机制，告诉Agent如何检索、筛选、组合乃至演化Skills，将复杂任务组织为composition、routing、checkpoint、用户澄清与最终综合的可检查任务图，Meta-Skill Creator进一步从用户历史协作轨迹中生成可审查proposal，使系统在使用过程中逐渐贴合用户；在记忆层，设计融合Markdown源文件、本地全文与语义混合召回、MMR（maximal marginal relevance）重排、可选时间衰减、Session Flush与Memory Dream的长期上下文生命周期，强调可编辑、可重建、可审计与可回滚，而不是简单累积会话日志；在安全层，构建Standard / Strict / Locked三档策略，结合权限审批、否决账本、过期输出缓存清理、路径与网络约束，以及Bubblewrap（Linux）/ Seatbelt profile生成（macOS）等平台能力，在不强制依赖Docker的前提下为工具调用建立运行时隔离边界。三者分别为 s_t 提供能力编排、跨会话状态与执行边界，共同支撑Harness的可学习性与可控性。

表1将OPENSQUILLA与两个代表性的同类开源Agent框架进行了多维度对比。该表的重点不是宣称所有维度绝对优越，而是展示OpenSquilla如何围绕“Agent原生、可学习、低成本、可治理”的Harness目标建立差异化系统设计。

Table 1 | OPENSQUILLA与同类Agent Harness的系统取向对比

维度	OpenSquilla	OpenClaw	Hermes Agent
架构形态	Agent 原生可学习微内核Harness: 稳定执行机制与可插拔、可演化策略分离, 随使用持续沉淀 workflow	本地优先个人助理运行时, 集成渠道、工具与会话能力	以长期记忆、技能生成和多后端部署为核心的个人Agent系统
成本控制	智能模型路由、推理预算/提示词策略分级、Prompt Cache 连续性保持以及技能按需加载	以模型配置与工具流程为主, 路由能力不是核心主张	提供后端与执行策略选择, 但缺少公开的本地ML路由闭环
可学习编排	Meta-Skill 协议层, 将历史协作轨迹和多Skills沉淀为可审查 workflow 能力	以固定工具/插件能力为主, 工作流演化不是核心抽象	强调技能生成, 但编排治理与harness 边界依赖其主循环设计
记忆系统	分层记忆、混合语义检索、可选时间衰减以及Memory Dream 巩固	文件/数据库结合的本地记忆与会话状态	强调长期记忆与会话检索, 具体实现依赖其运行时设计
安全治理	三档安全策略, 否决账本、缓存失效与平台沙箱协同	借助工具权限、运行环境和外部隔离降低风险	以审批、执行环境和任务策略约束高风险操作
可观测性	通过诊断日志、只读回放与会话导出记录路由、缓存、压缩、工具与成本事件	依赖会话日志与工具轨迹复盘	依赖会话检索、技能轨迹和运行日志复盘

2. Agentic Routing: Harness 原生的模型路由

传统Agent 对分解式 (1) 的实例化方式是: 离线固定一个基础模型 $\bar{m}$, harness 的每一步都调用同一个模型, 全部优化发生在harness 一侧——prompt、工具、上下文管理与恢复策略都围绕这个固定模型调参。用本章的语言说, 这类Agent 没有路由算子: 模型选择是一个常量, 实际成本结构由 $\bar{m}$ 决定, 质量与成本之间的唯一权衡手段是把整个Agent 换到另一个模型上。一旦模型池 $\mathcal{M}$ 中出现多个基础模型, 这个常量就变成了逐步决策变量。Agentic routing 把固定的 $\bar{m}$ 替换为状态条件路由算子 g : 在每个执行步上决定接下来由哪个模型或哪组模型行动, 从而把质量-成本权衡变成显式优化目标; 传统Agent 则退化为常值路由 $g \equiv \{\bar{m}\}$ 。图 2 概览了该算子的两种部署机制: 省钱模式下路由器派出单个最适配模型 (subsection 2.2), 高精度模式下派出多个互补模型并聚合其输出 (subsection 2.3)。本章形式化该算子及其目标, 作为目标式 (2) 中 g_t 一侧的理论基础; 与路由协同的运行时成本机制 (推理预算与提示策略派生、提示缓存与技能过滤) 见Section 3。

2.1. 问题定义与优化目标

Agentic routing 是Agent 执行过程中以harness 状态为条件的步级模型 (或模型集合) 选择。设 q 为用户任务, $\mathcal{M}$ 为候选模型池, h_t 为第 t 个决策步的harness 状态。本报告中, 步 (step) 指一次模型调用决策, 轮 (turn) 指一次用户可见的交互回合、由一个或多个步组成; 目标式 (2) 的子任务下标 t 按步取值, Section 3 中的逐轮控制信号即这些步级决策在轮粒度上的视图。 h_t 不仅包含当前观测, 还包含压缩与原始上下文、控制循环状态、可用动作、产物 (artifact) 状态、工具历史、恢复状态与验证信号。

我们不把路由定义为单一的成本最小化问题。已有的模型路由与级联系统表明, 异构模型池能改善LLM 服务的成本-质量折中, 但它们通常作用在静态prompt 或query 层面 [6, 11, 22, 26]; 近期的agentic 路由基准与路由器则把这一决策推向执行轨迹 [40, 48]。与依赖硬编码规

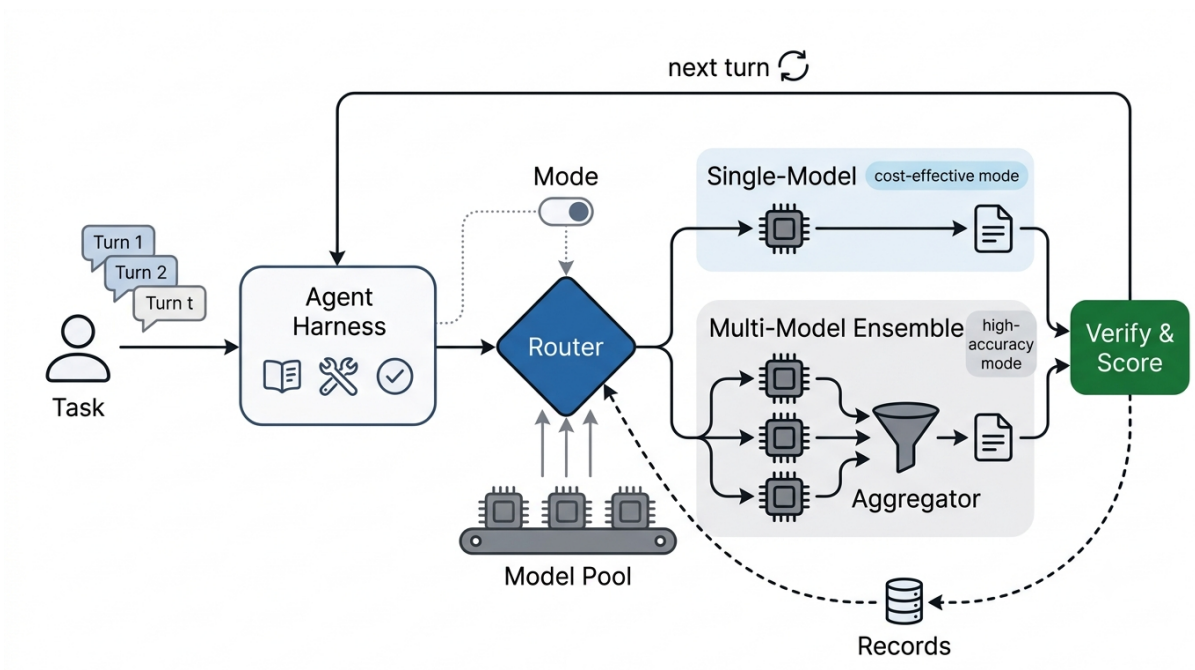


Figure 2 | **Agentic routing** 两种工作机制概览。Agent harness 将执行状态提供给路由器，路由器从共享模型池中进行选择；用户选定的部署模式决定工作机制：省钱模式（cost-effective）下路由器派出单个最适配模型，高精度模式（high-accuracy）下派出多个互补proposer并由聚合器融合其输出。验证与成本信号为每次决策打分，产生的记录回流用于改进路由器。

则的静态分流以及仅做语义感知模型选择的LLM路由不同，本文采取harness原生视角，我们的形式化再前进一步：路由器既可以在更低成本下保持质量，也可以组合多个更便宜的专长模型去超越强单模型基线，还可以在高风险状态上投入更多预算以提升任务成功率。因此我们为harness原生的Agent执行定义统一的质量-成本前沿。

我们把路由建模为算子 g ：以当前harness状态 h_t 为条件，从模型池 $\mathcal{M}$ 中为下一执行步选出模型集合 $S_t = g(\mathcal{M} | h_t)$ ，并在选出多个模型时同时给出组合策略。路由器优化任务损失与成本之间的Pareto前沿：

$$\min_g \left(\mathbb{E}_{q \sim \mathcal{D}, \tau \sim g} [\ell(\tau)], \mathbb{E}_{q \sim \mathcal{D}, \tau \sim g} [C(\tau)] \right), \quad S_t = g(\mathcal{M} | h_t), \quad 1 \leq |S_t| = k \leq K, \quad (3)$$

其中目标对按Pareto意义最小化， $\ell(\tau) = 1 - R_{\text{task}}(\tau)$ 是最终轨迹的任务损失， $C(\tau) = \sum_{t=1}^T \sum_{m \in S_t} c(m, h_t)$ 是实际发生的成本， $k = |S_t|$ 是第 t 步同时上场的模型数，受上限 K 约束。取 $k = 1$ 即得单模型路由（subsection 2.2）， g 每步恰好派出一个模型； $k > 1$ 即多模型路由（subsection 2.3）， g 派出多个模型并附带组合策略。等价地，对标量化目标 $\mathbb{E}_{\tau \sim g} [\ell(\tau) + \lambda C(\tau)]$ 在 $\lambda \geq 0$ 上扫描可以刻画同一条前沿；当部署有需要时，延迟作为可选的第三条轴进入目标。式(3)是目标式(2)在本文路由场景下的特例：固定Harness操作 s_t 、只沿模型选择维度变化， g 即逐步实例化的 g_t ，而 $\ell(\tau)$ 与 $C(\tau)$ 分别扮演 $\mathcal{L}_T(a)$ 与 $P(a)$ 的角色。

解读式(3)：一个路由器 g 更优，当且仅当它降低任务损失、降低成本，或两者兼得。关键在于，多模型动作（ $k > 1$ ）并不必然比单模型动作更贵，因为 S_t 可以组合若干便宜模型或专长模型，其总成本低于一次昂贵旗舰模型调用。前沿上不同的工作点回答不同的问题：低成本点问的是harness最便宜能以多少钱保住任务质量，高精度点则在困难、高风险或弱可恢复状态上投入更多预算。本文沿路由动作的模型与模型集合维度实例化这条前沿：动作给出模型集合 S_t ，并在 $k > 1$ 时附带聚合策略。为了把模型分配这一分量单独隔离出来，本文固定周边的harness策略，只评测模型与模型集合选择能把前沿推进多远，从而使所报告的收益可归因于

固定执行底座上的能力分配。

损失定义在任务层而非单步层。**Agent** 执行是可恢复的：局部较弱的一步可以在后续被修正，而局部便宜的一步可能制造下游恢复成本。这一视角与把推理、行动、工具使用、记忆与反馈视为交织执行过程的**Agent** 工作一致 [30, 31, 35, 39, 41]。因此监督信号不应假设每一步都有独立的正确性标签；自然的奖励是终局的或由验证派生的， g 必须从轨迹而非孤立prompt 中学习。轨迹字段

$$(q, h_t, S_t, z_t, c_t, y) \quad (4)$$

定义了harness 原生路由轨迹数据（routing records）的模式：每条记录把路由动作与环境事后给出的标签分开存储——动作是当时派出的模型集合 S_t ，标签则是最终结果 y 、单步产出 z_t 、成本 c_t 与验证信号。这一分离意味着轨迹不是被模仿的对象：一次糟糕的路由决策连同其负向结果同样也是有效训练样本。在本报告中，开源的LightGBM 路由器是启动这一数据流的冷启动策略，后续的router 模型迭代（subsection 2.4）则利用累积的路由轨迹数据改进未来的路由决策。

2.2. 单模型路由

我们首先在单模型设定下实例化前沿。在式 (3) 中取 $k = 1$ ，路由操作被限制为单个模型：

$$S_t = g(\mathcal{M} \mid h_t) = \{m_t\}. \quad (5)$$

路由器仍优化同一条任务级前沿，但其在线动作变为：从异构模型池中为下一个harness 步恰好派出一个模型。

我们把这一视角称为能力匹配（capability matching）：harness 发出状态条件的能力需求；模型池提供价格、延迟、上下文长度、工具调用可靠性与失效模式各异的能力供给；单模型路由器选择能力足以覆盖当前状态、且经风险调整后最便宜的模型。模型池因而是一组专长模型的集合，路由器在不确定性下为每个状态的需求匹配最合适的模型。路由器做的不只是给prompt 分类，而是在harness 的执行轨迹中分配模型能力：一个表面便宜的模型，若其失败引发重试、工具修复、上下文重建或下游验证恢复，就会变得昂贵。

设单模型路由状态为

$$x_t = (q, h_t), \quad (6)$$

其中 q 是原始用户任务， h_t 是当前harness 状态。 h_t 不仅包含可见prompt，还包含上下文压力、artifact 状态、工具历史、恢复状态、近期路由决策与验证信号。所需能力因此不是用户query 的固定属性，而是当前执行状态诱导出的、依赖轨迹的需求。

操作上，单模型路由是式 (3) 在 $k = 1$ 下于当前步的贪心应用。为了让这一步级决策比纯终局信号更易估计与优化，我们加入单轮奖励 $\rho_t = \rho(h_t, S_t, z_t) \in [0, 1]$ ，对路由动作的即时产出 z_t 打分：它可以是可验证奖励（单元测试通过、工具调用成功、schema 或安全约束满足），也可以是LLM-as-a-judge [45] 对该步输出的评分。固定式 (3) 的权重 λ ，路由器派出使剩余轨迹的期望任务损失与成本最小、并由即时奖励抵减的单个模型：

$$m_t = \arg \min_{m \in \mathcal{M}_t} \mathbb{E}_{\tau \sim g} [\ell(\tau) + \lambda C(\tau) - \beta \rho_t \mid S_t = \{m\}, h_t]. \quad (7)$$

这里 $\mathcal{M}_t \subseteq \mathcal{M}$ 是经准入过滤与部署可用性约束裁剪后的当前候选模型集合；权重 $\beta \geq 0$ 在即时单轮奖励与轨迹级损失 $\ell(\tau)$ 、成本 $C(\tau)$ 之间折中； $\beta = 0$ 时退回式 (3) 的纯终局目标。 ℓ 与 C 保持轨迹级，保留了可恢复执行的语义——表面便宜但随后引发重试、工具修复、上下文重建或验证恢复的模型仍会抬高下游成本 $C(\tau)$ ，而能力不足以覆盖当前状态的模型会抬高任务损失 $\ell(\tau)$ ——单轮奖励只是把监督信号致密化，给路由器提供即时的步级反馈，缓解信用分配并加速优化。能力匹配的直觉——用池中异构的能力供给去匹配状态条件的能力需求，并计入不

可逆或弱可恢复动作的残余风险——是对 $\ell(\tau)$ 、 $C(\tau)$ 与 ρ_t 如何响应模型选择的一种解读，而非另一个独立的优化问题。要点不变：失败会制造昂贵恢复的模型不算便宜，能力与状态需求错配模型也不算强。

所需能力画像是多维的。例行控制、短工具调用格式化、低风险上下文簿记或可缓存的响应生成可能只需要轻量模型；含糊的规划、困难的代码修改、长上下文调试、验证失败后的恢复、安全敏感决策或最终产物生成则可能需要更强的模型。同一用户任务在不同步骤上因此可能需要不同模型：上下文压力会让长上下文可靠性比原始推理分数更重要；最近的工具错误会让动作格式稳健性比榜单成绩更重要；验证器失败会抬高下一恢复步所需的能力。这正是harness原生单模型路由与普通query级路由的区别：模型选择不仅取决于用户问了什么，还取决于harness当前处在执行轨迹的哪个位置。

OPENSQUILLA中开源的单模型路由器以轻量冷启动机制实现了这一匹配视角。它执行四段流水：准入过滤（admission filtering），把明显琐碎且低风险的状态送入便宜的快速通路；需求构建（demand construction），从任务与harness状态构建粗粒度能力画像；风险调整（risk pricing），用上下文压力、工具失败、验证器结果、恢复状态、动作不可逆性与下游纠错风险调整该需求；能力匹配（capability matching），通过部署registry把调整后的需求绑定到具体模型或档位。这一分解刻意强于单纯的模型排序流水线：它把“下一个harness步需要什么”与“当前部署的哪个模型能满足它”两个问题分开。在开源实现中，前三段由便宜的确定性过滤器与轻量分类器实现：需求构建对prompt的长度、语言、代码形态、关键词与语义嵌入等混合特征打分，并识别代码块、附件、长上下文与高风险提示等任务信号；最后的匹配段由LightGBM ranker在剩余候选上完成 [16]，其中图像等多模态请求作为独立能力类型施加硬约束——只绑定具备相应输入能力的模型，不与文本档位混用。全部流水在本地毫秒级执行，路由本身不消耗LLM token，也不会把用户请求发送给第三方分类服务。LightGBM有用但并不根本：它是冷启动的匹配引擎，而非匹配问题本身——该问题由harness状态、模型池、部署registry、验证器与运行时计费共同定义。更难的学习问题是从harness原生轨迹中推断状态条件的能力需求、下沉误配（under-routing）的风险，以及式 (7) 只通过 $\ell(\tau)$ 与 $C(\tau)$ 隐式刻画的期望恢复成本。

这一定位对开源边界很重要。LightGBM路由器是单模型前沿上第一个实用点：冷启动样本效率高、步级路由速度快、可检查因而可开源、且足够简单可从日志执行中改进。它不应被读作最终的路由器模型，而是启动路由轨迹数据流的开源种子。路由器模型应被视为迭代的匹配策略：每一批路由轨迹数据都更新定义单模型路由的同一组潜在对象——需求构建、风险调整与能力匹配。状态与所选模型作为动作被记录；最终结果、验证器事件、恢复成本、实际成本与延迟由环境提供。这些记录使后续router模型迭代能学到开源策略何时下沉误配、何时为不必要的能力多付了钱、何时因模型池或harness变化导致某模型的部署画像已经漂移。在单模型设定下，router模型迭代按省钱模式评测：每一代必须在保持任务质量的同时为当前步派出一个模型，并减少对昂贵模型的不必要调用。学习目标是结果改进而非模仿上一代路由器，与上下文bandit评估中的日志反馈与异策略学习视角一致 [8]：一个糟糕的LightGBM决策是有用的反馈而非有毒的监督，因为标签由验证、恢复、实际成本与延迟等环境信号提供。在这个意义上，单模型路由不是狭义serving层的模型选择，而是harness状态不确定性下面向结果的模型能力分配。

2.3. 多模型融合路由

多模型融合路由是式 (3) 在 $k > 1$ 下的实例化：路由器 g 派出一个模型集合而非单个模型。给定harness状态 h_t 与同前经前置过滤裁剪后的候选池 $\mathcal{M}_t \subseteq \mathcal{M}$ ，路由操作返回提议集合与组合策略：

$$g(\mathcal{M}_t | h_t) = (P_t, a_t), \quad P_t \subseteq \mathcal{M}_t, \quad a_t \in \mathcal{G}, \quad (8)$$

其中提议模型（proposer）集合 P_t 生成候选输出，聚合策略 a_t 把候选融合为单一可执行结果； $\mathcal{G}$ 包含投票、基于验证器的选择、judge模型与回退策略。当聚合器本身是一次模型调用

时，式 (3) 中的选中集合为

$$S_t = P_t \cup \{m_t^{\text{agg}}\}, \quad k = |S_t|, \quad (9)$$

其中 m_t^{agg} 是作为 a_t 一部分选出的聚合模型；对规则、投票或验证器聚合器则 $S_t = P_t$ 。此时式 (3) 的成本 $C(\tau)$ 计入每次提议调用与聚合开销，延迟单独作为开销指标跟踪。

与单模型情形一样，同一条前沿逐步实例化。记 $\ell(u | h_t) = \mathbb{E}_{\tau \sim g}[\ell(\tau) | u_t = u, h_t]$ 、 $C(u | h_t) = \mathbb{E}_{\tau \sim g}[C(\tau) | u_t = u, h_t]$ 、 $\rho(u | h_t) = \mathbb{E}_{\tau \sim g}[\rho(\tau) | u_t = u, h_t]$ 分别为路由动作 u 在状态 h_t 下的期望任务损失、成本与单轮奖励。对多模型动作 $u_t = (P_t, a_t)$ ，路由器最小化式 (3) 的标量化目标，并增广互补性正则项与单轮奖励：

$$(P_t, a_t) = \arg \min_{P \subseteq \mathcal{M}_t, a \in \mathcal{G}} \left[\ell((P, a) | h_t) + \lambda C((P, a) | h_t) - \alpha V(P | h_t) - \beta \rho((P, a) | h_t) \right]. \quad (10)$$

前两项是式 (3) 在 $k > 1$ 下于当前步贪心应用的标量化前沿目标；第三项 $-\alpha V(P | h_t)$ 是权重为 α 的互补性正则项；第四项 $-\beta \rho((P, a) | h_t)$ 是式 (7) 的单轮奖励向集成动作的推广，验证器或 LLM judge 作用于聚合后的输出。式 (7) 的单模型规则是特例 $u_t = (\{m\}, \emptyset)$ ，此时 V 消失。

成本 C 累计提议调用与聚合开销，因此更大的集合恰按其多花的钱被惩罚。互补性项 $V(P | h_t)$ 奖励失效方式去相关的提议集合。我们用基于历史错误去相关、状态条件分歧与验证器反馈的边际递减代理实现 $V(P | h_t)$ ，支持高效的贪心子集构造。原则上任务损失 ℓ 已经偏好这类集合——降低相关失效的提议者降低期望损失，而共享训练分布与失效模式的冗余模型只抬高成本不降低损失。但由于 ℓ 是轨迹级、异策略且带噪的，而 P 的选择是组合性的，显式的 $-\alpha V$ 作为正则项使子集搜索更易优化与泛化，把探索引向互补集合而不是塌缩到日志策略的既有偏好。它是代理目标上的归纳偏置，不改变式 (3) 的前沿： α 取小值并随损失估计变准逐渐退火至零，从而随估计器改进恢复真实的成本-质量前沿，路由器永远不会因不降低损失的多样性而获得奖励。这一互补性项正是多模型路由区别于 **top-k** 选择之处。规模上限 $|P| \leq K$ 继承自式 (3)；当部署有需要时，延迟预算作为可选第三轴进入。

两种工作机制是式 (10) 在不同成本权重 λ 下的两种读法，均与强单模型参考动作 $u_t^{\text{ref}} = (\{m_t^{\text{ref}}\}, \emptyset)$ 对照。 λ 较轻时，极小化子偏向求精集成 (**accuracy-seeking**)：在单模型错误代价高且难以恢复的状态——最终答案生成、复杂代码修改、不可逆工具动作、安全敏感决策、长上下文综合或路由器低置信——花费额外调用，把损失压到参考之下 ($\ell(u_t | h_t) < \ell(u_t^{\text{ref}} | h_t)$)，代价是在单步预算 B_t 内付出更高成本。典型分工是：一个模型给出主提议，另一个从不同视角找错，第三个提供便宜的合理性检查。 λ 较重时，同一极小化子偏向省钱集成 (**cost-saving**)：用中低成本模型的组合替换昂贵参考，在容差 ε 内保住损失 ($\ell(u_t | h_t) \leq \ell(u_t^{\text{ref}} | h_t) + \varepsilon$) 同时严格少花钱 ($C(u_t | h_t) < C(u_t^{\text{ref}} | h_t)$)；当损失也严格低于参考时，同一组合同时是提精度的。两个条件均由式 (10) 导出而非新增目标。多模型路由的经济学，就是在强单模型、便宜单模型与更便宜模型的组合三者之间寻找更优的成本-质量工作点。

我们把多模型路由实现为两阶段过程。第一阶段是任务画像驱动的子集选择：路由模型先从 (q, h_t) 计算任务画像 φ_t ，概括任务类型、难度、所需能力、风险水平、验证可得性、成本与延迟容忍度、路由不确定性等信号。该画像是上述目标中状态条件量的紧凑表示，而非新的环境状态。给定 φ_t 与候选池 $\mathcal{M}_t$ ，选择器挑出能力与画像匹配、成员提供互补证据的提议子集 P_t 。额外提议者按其对应任务画像的契合度、自身成本与延迟画像、期望边际价值、验证兼容性、不确定性削减以及与已选模型的互补性打分。一个有价值的提议者不只是孤立地强，而是补足任务画像所需的能力或视角，例如长上下文证据恢复、代码合成、对抗性批评或低成本查错。选择因此是画像条件的子集构造而非 **top-k** 排序，且偏好依机制而异：求精集成偏好关键状态上的可靠性增益，省钱集成偏好更低总成本下的质量保持。第二阶段是状态感知聚合：每个提议者 $m \in P_t$ 独立生成候选 r_m ，聚合器产出最终结果

$$\hat{r}_t = a_t(\{(m, r_m)\}_{m \in P_t}, h_t). \quad (11)$$

策略 a_t 同样以任务画像与 **harness** 可观测的最强验证信号为条件：软件任务中的单元测试、静态分析或补丁应用等可执行信号；工具任务中的 **schema**、权限与安全约束；开放研究任务中

的rubric 评审、一致性、事实性与引用检查。 a_t 不是后处理模块而是路由动作的一部分：路由器同时决定谁来提议与如何融合。在省钱机制下聚合器自身必须成本敏感——便宜提议者之上的昂贵聚合器会吃掉提议侧的节省——此时 a_t 是轻量judge、规则或验证器选择器、或低成本模型聚合器；求精机制则可在任务风险、验证难度与质量提升足以覆盖其成本时使用更强的聚合器。

每个集成步同时也是数据生产操作：各提议者的候选 r_m 提供了同一决策点 (q, h_t) 上多个模型的对照结果，为异策略学习所需的反事实覆盖贡献数据。两阶段设计保持了单模型的成本-质量折中：系统在例行状态退化为单模型路由，在单模型失败代价高的状态选择性引入互补提议者，在成本敏感状态用更便宜模型的组合替换强单模型——在扩展质量前沿的同时降低到达给定质量水平的成本。

2.4. Router 模型迭代：实现与演化

路由器不是以单个模型部署的，而是一列共享式 (3) 目标、但表示与训练数据各异的策略序列 $g^{(0)}, g^{(1)}, g^{(2)}, \dots$ 。第零代 $g^{(0)}$ 即subsection 2.2 的开源LightGBM ranker：手工特征馈入浅层成本-质量排序器，外层包裹准入过滤、需求构建、风险调整与能力匹配四段便宜流水 [16]。它刻意不是最终路由器，而是能在每一步运行、并启动路由轨迹数据流的最便宜策略。之后的每一代 $g^{(r)}$ 都是在该数据流上训练的学习模型，本节描述这样一代策略如何构建、训练与滚动上线。

架构。一次router 模型迭代分解为三个组件。状态编码器把harness 状态 h_t ——任务与指令、压缩与原始上下文、artifact 与工具历史、验证器输出、恢复状态与近期路由历史——映射为状态向量，替换 $g^{(0)}$ 的手工特征。模型编码器把每个候选模型 $m \in \mathcal{M}_t$ 映射为画像嵌入，由其model card 与观测到的结果历史构建：价格、延迟、上下文长度、工具调用可靠性与任务族成功率。由于候选以画像而非固定输出类别描述，新发布的模型可以直接从画像打分而无需重训标签空间。预测头对状态 h_t 下的候选动作 $u = (s, a)$ 估计期望任务损失 $\hat{\ell}(u | h_t)$ 与期望成本 $\hat{C}(u | h_t)$ ，路由器则严格按式 (7) 与 (10) 的规则行动，选择 $\arg \min_u [\hat{\ell}(u | h_t) + \lambda \hat{C}(u | h_t)]$ 。

训练。每一代在累积的路由轨迹语料 $\mathcal{D}^{(r)}$ 上最小化式 (3) 的前沿目标：

$$\theta^{(r+1)} = \arg \min_{\theta} \hat{\mathbb{E}}_{\mathcal{D}^{(r)}} [\ell(\tau) + \lambda C(\tau)]. \quad (12)$$

该期望是异策略估计：由于每条记录都由较早的策略 $g^{(r)}$ 而非动作上的均匀采样产生，估计器用逆倾向或双重稳健校正对日志结果重加权，使更新以结果改进而非模仿日志策略为目标 [8]。无需额外的奖励、恢复或风险项：恢复成本与下沉误配恰如subsection 2.2 与subsection 2.3 所述，通过轨迹级的 $C(\tau)$ 与 $\ell(\tau)$ 进入目标， λ 选定前沿上的工作点。

覆盖。覆盖被当作路由数据设计的一等公民。日志记录天然对被选动作观测最密，因此路由轨迹语料引入受控探索——不确定性触发的升档，以及在抽样状态子集上偶发的替代模型或oracle 重放——以改善反事实覆盖，使 $g^{(r+1)}$ 能学到 $g^{(r)}$ 在哪些状态下沉误配、在哪些状态为能力多付了钱。

演化。策略序列构成一架梯子，每一代只有在把式 (3) 的前沿推得比上一代更远时才被晋升。开源种子 $g^{(0)}$ 是手工特征上的LightGBM ranker；第一学习代 $g^{(1)}$ 用训练得到的状态编码器与预测头替换手工特征，在首批语料上离线拟合，仍闸控在 $g^{(0)}$ 的便宜准入过滤之后；下一代 $g^{(2)}$ 加入面向新模型泛化的模型编码器画像与探索补齐的反事实数据，并可蒸馏为小型快速路由器以满足步级延迟预算；后续各代 $g^{(r)}$ 随语料增长与模型池变化持续更新。候选代只有在等任务奖励下降低实际成本、或固定预算下提升任务奖励时才发布，否则部署回滚到上一代。与 $g^{(r-1)}$ 的一致率从来不是目标——质量-成本前沿的移动才是。

部署。由于路由器每步都在运行，其自身的成本与延迟也是折中的一部分。部署保持两层：便宜的准入过滤与 $g^{(0)}$ 式闸门解决明显琐碎或明显困难的状态，较重的学习路由器只在不

确定、高价值或弱可恢复、值得为更优匹配决策支付开销的状态上被调用。这使路由器始终停留在其自身成本-质量折中的有利一侧。这一分阶段路径同时划定了开源边界： $g^{(0)}$ 种子可检查、可本地训练、对冷启动有用，因此可以开源；后续各代依赖累积的路由轨迹数据、变化中的模型供给画像与持续的前沿评测，因此是一条演化中的OPENSQUILLA策略路径而非定稿产物。路由器序列的收敛不是部署的前提：每个晋升代都以测得的前沿移动为准绳，路由轨迹数据流则随模型池与harness 演化持续提供改进所需的数据。

3. 智能成本控制

3.1. 双层成本控制框架

LLM 智能体的实际负载呈现两类结构性异质：单轮难度的高方差与跨轮内容的高重复度。一句简短确认与一段跨文件代码修订在难度上相差可达数十倍以上，单一档位策略——无论恒用最强还是恒用最弱——都无法兼顾质量与可持续成本[6, 26]。同时，同一会话中的系统提示、工具定义与稳定上下文在token 维度上高度重复，朴素重传会让「成本 $\propto$ 轮数」成立。因此，OpenSquilla 将成本控制拆解为两个互补问题：单轮成本必须按难度分级，跨轮重复必须被识别并摊销。其中“按难度选择模型档位”这一路由问题已由Section 2 的agentic routing 框架完整刻画——SquillaRouter 即其中启动路由轨迹数据流的冷启动策略 $g^{(0)}$ ，在本地完成准入过滤、需求构建、风险调整与能力匹配四段流水，输出路由档位、置信度与决策元信息，本章不再重复路由机制本身。从目标式 (2) 的视角看，本章关注的是在 $g_t(\mathcal{M})$ 已由路由器给出的前提下，如何进一步降低 $P(a)$ ：决策层从路由元信息派生推理预算与提示策略，运行时层则通过cache、prompt 与Skill 注入等 s_t 操作降低重复token 和无关上下文；系统同时保留单模型直连模式，用于复现实验与在评测中隔离“模型能力”与“框架调度贡献”。

图 3 给出这一思想对应的双层架构。决策层在每一轮给出离散控制信号 (m_t, r_t, p_t) ：文本模型类型 $m_t \in \{c0, c1, c2, c3\}$ （从轻量到旗舰的四个能力档位，图像等多模态能力作为独立类型处理）由Section 2 的单模型路由器选出，推理预算 r_t 与提示词策略 p_t 则由§3.2 从同一份路由元信息派生。运行时层与决策层正交，由Prompt Cache 连续性与技能过滤组成，分别处理跨轮稳定前缀复用与可注入元信息裁剪。两层在上游LLM 调用处汇合：决策层减小“当轮要花的边际”，运行时层把“已花过的”和“不必花的”从下一轮上下文中剔除。

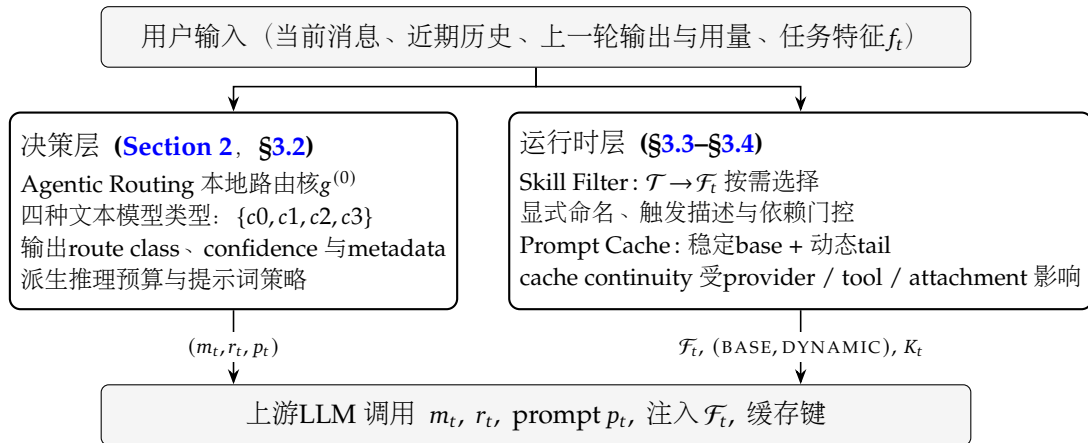


Figure 3 | OpenSquilla 双层成本控制框架。决策层每轮给出文本模型类型、推理预算与提示词策略；运行时层并行执行技能过滤与Prompt Cache 连续性维护，输出可注入技能集 $\mathcal{F}_t$ 以及稳定base / 动态tail 形式的提示结构。两层在上游LLM 调用处汇合，共同降低单轮边际成本与跨轮重复成本。

表 2 把双层框架涉及的四类机制按作用范围进行横向对比。该表是Section 2 与§3.2–§3.5 的索引，读者在不同小节中遇到具体机制时可回查本表以确认其在整个双层框架中的位置。

Table 2 | 双层框架内四类成本机制的横向对比。“失效退路”一列描述各机制不可用时的退化行为。

机制	作用层级	时间尺度	主要可观测信号	失效退路
模型档位路由 由 m_t (Section 2)	决策层/ 单轮	本地毫秒级	routed tier, confidence, metadata	默认中档tier (c1)
推理预算与提示策略 派生(r_t, p_t)	决策层/ 单轮	本地执行	reasoning 档位、policy、metadata	保守默认预算与policy
提示缓存	运行时层/ 跨轮	摊销到整会话	cache token、cache invalidation、base/tail 变化	按cache miss 计价
技能过滤	运行时层/ 系统级	单轮, 随 $ \mathcal{T} $ 缩放	命中技能、eligibility、inspect / meta runs	显式技能名或无技能注入

3.2. 推理预算与提示策略的派生

档位 m_t 由Section 2 所述的单模型路由器在本地选出，只回答“调用哪一档模型”。OpenSquilla 还会从同一份路由元信息中派生推理预算与响应策略，用以回答“是否需要更深推理”和“提示应当多充分”。这使系统可以在同一模型档位内继续调节成本：例如低难度请求可使用更轻的推理预算和更短提示；调试、长上下文、严格格式或高风险操作则使用更保守的响应策略，并把执行风险交给权限和沙箱层进一步裁决。

令 ψ_r 表示reasoning 派生函数， ψ_p 表示policy 派生函数，则

$$r_t = \psi_r(m_t, \pi_t, f_t, H_t), \quad p_t = \psi_p(m_t, \pi_t, f_t, H_t). \quad (13)$$

这里 π_t 是路由器输出的档位分布， f_t 是由关键词、代码块、附件与高风险提示解析出的任务特征， H_t 为有限历史窗口。两个派生函数不直接暴露为用户可见的“思维链文本”，而是映射为模型接口可接受的推理预算、响应长度、格式严格度或提示变体。换言之，OpenSquilla 控制的是推理预算与响应策略，而不是把模型内部推理过程原样写入上下文。

这种设计的价值在于把成本控制从“一维模型选择”扩展为“三维预算分配”：tier 决定模型单价，reasoning 决定推理token 与延迟，policy 决定动态提示长度和格式约束。三者共同构成端到端Agent 成本的可控面。

3.3. 跨轮机制：提示缓存

除单轮tier 选择外，Agent 成本还受到跨轮重复前缀的影响。提示缓存利用的是provider 侧对稳定前缀的KV-cache 复用，harness 的职责是保持该前缀稳定。OpenSquilla 将提示组织为稳定base 与动态tail：base 主要包含系统身份、稳定约束与工具定义；tail 承载当前请求、运行时上下文、召回历史、附件摘要与工具结果。设 γ_t 为第 t 轮稳定前缀的可复用比例， δ 为provider 对cache read token 的折扣系数，则输入侧成本可写为

$$\tilde{c}_t^{\text{in}} = c_m(m_t) ((1 - \gamma_t) n_t^{\text{base}} + \gamma_t \delta n_t^{\text{base}} + n_t^{\text{tail}}). \quad (14)$$

该式不依赖某个特定provider 的定价细节，只说明：当稳定前缀保持稳定且provider 支持缓存计时，重复系统提示与工具定义不必在每轮按全价重付。

工程上，缓存连续性是best-effort，而非绝对保证。模型档位改变、供应商切换、工具定义变化、新附件进入上下文、长上下文压缩或动态技能集合变化，都可能降低复用率。因此，系

统在诊断日志中记录缓存命中与失效事件，并在会话级成本视图中暴露缓存token与总token的关系，使缓存收益可以被复盘，而不是只停留在配置层主张。

3.4. 跨轮机制：技能过滤

技能系统是OpenSquilla降低prompt膨胀的另一条链路。系统预置并允许扩展多来源技能，覆盖编码、版本协作、文档生成、摘要和外部信息获取等常见场景。若每轮把全部技能说明都注入系统提示，技能库越丰富，稳定前缀越长，成本越高。因此OpenSquilla采用“按需加载相关指导”的策略：模型并不默认看到全集技能，而是由运行时根据任务意图、显式技能名、触发描述和安装环境选择候选。

形式化地，设全集为 $\mathcal{T}$ ，本轮注入集合为 $\mathcal{F}_t$ ，则

$$\mathcal{F}_t = \text{Filter}(\mathcal{T}, \text{msg}_t, \text{ctx}_t), \quad |\mathcal{F}_t| \ll |\mathcal{T}|, \quad (15)$$

其中 msg_t 为当前用户消息， ctx_t 包括运行环境、可用依赖、工具权限、工作区边界与用户显式意图。筛选过程包含两类约束：一是可用性门控，例如依赖缺失或仅用于演示的技能不可用；二是相关性匹配，即把用户自然语言目标与技能描述、触发词、Meta-Skill结构进行匹配。系统还提供技能结构检查能力，使用户和开发者能够理解某个Meta-Skill的步骤边界。

技能过滤与提示词策略 p_t 形成嵌套关系： p_t 决定提示的形态（压缩/中性/充分），技能过滤决定可注入内容池的规模。在目标式(2)中，这类按需注入正是 s_t 的核心组成：它不改变所选模型本身，却通过选择当前模型应看到哪些能力描述，影响任务成功率与prompt成本之间的权衡。两者解耦的好处是：技能库规模增长主要影响筛选侧，而不迫使每一轮都承担全量技能描述的token成本。运行时还通过技能可见性、结构检查、执行轨迹与proposal管理接口暴露关键决策，使技能注入从“隐式prompt魔法”变为可检查的系统行为。

3.5. 失败模式与运维可观测性

软失败语义 预测核加载失败、路由资产校验失败或单次分类抛错时，路由器返回默认中档模型，并附带可诊断原因，而不是静默选择最便宜档位。该策略把未知难度留在中档安全区间，降低因误降导致的失败重试和质量塌陷。

运行时层的退化 稳定提示前缀在会话中段变化（例如新工具、附件或技能进入上下文）会降低缓存连续性；供应商或模型切换也会造成缓存收益下降。技能系统侧，依赖缺失或可用性门控未通过时，相关技能不进入注入集合；用户仍可通过技能可见性与结构检查接口确认其可用性。记忆或语义检索后端不可用时，系统保留关键词/全文检索路径，避免因可选组件缺失而中断主turn。

诊断与回放 OpenSquilla提供统一的诊断、会话导出与只读回放能力。诊断日志记录模型通信、SquillaRouter决策、提示缓存命中与失效、上下文压缩、工具输出压缩、渠道投递以及成本/延迟异常；只读回放不重新执行工具，而是从决策日志生成可读轨迹。由此，成本控制不只是在线策略，也具备离线复盘与回归评估能力。

至此，OpenSquilla的智能成本控制可以概括为决策层与运行时层正交协同的双层系统：决策层每轮决定“调用哪一档模型/使用多强推理预算/采用何种响应策略”，其中模型档位由Section 2的agentic routing给出；运行时层把“已花过的稳定前缀”和“不必注入的技能说明”从下一轮上下文中剔除。所有关键决策都进入诊断、成本统计与只读回放视图；子组件不可用时退化为默认中档模型、单模型直连或关键词检索等可解释路径。由此，成本控制成为目标式(2)中 $P(a)$ 一侧的工程实现： g_t 通过agentic routing降低模型单价， s_t 通过推理预算派生、缓

存连续性与按需技能注入降低推理与重复token成本，共同推动Intelligence/Token的提升。后续实验将展示这种设计在端到端Agent benchmark上形成的成本-能力折中。

4. 可学习Harness的系统机制：Meta-Skill、持久记忆与安全治理

在Section 2与Section 3回答“调用哪个模型、花多少钱”之后，本章介绍支撑可学习Harness的三类系统机制，分别对应目标式(2)中 s_t 的三个维度：Meta-Skill编排层决定能力如何组织与演化，持久记忆系统提供跨会话状态基底，安全治理框架为 $g_t \circ s_t$ 的可执行范围划定边界。三者共同保证Harness在“越用越好”的同时保持可审查、可回滚与可控。

4.1. Meta-Skill: Skill编排与系统进化

传统Agent架构中，Skill封装摘要、检索、分析、写作、工具调用等单项能力。随着bundled、workspace与社区来源Skills不断扩展，系统要解决的不再是“有没有能力”，而是“如何让Agent检索、筛选、组合并演化这些能力”：跨阶段、需要条件判断和多轮交互的复杂任务，仅靠单个Skill或模型的临场规划难以稳定复用。为此，OPENSQUILLA将Meta-Skill定义为位于Skill之上的声明式编排机制——对多个Skill进行composition、routing、依赖调度、失败替代、用户澄清与最终汇总的工作流抽象。composition把复杂任务拆分为语义明确、有依赖与数据流的任务图，例如报告生成被组织为需求理解、结构规划、分章生成与一致性汇总，各阶段由不同Skill承担；routing则把执行分支显式化：入口routing从用户请求选择Meta-Skill，流程内routing依中间结果选择后续分支，能力routing在候选Skill中挑选适合当前上下文的能力，恢复routing在用户补充信息后从暂停点继续。其关键原则是：模型负责理解意图，运行时负责约束执行——任务的整体结构、步骤顺序、分支路径与输出策略由声明式工作流控制，从而把大模型的灵活性嵌入可验证、可复用、可观测的工程框架。用目标式(2)的语言表达，Skill扩充 s_t 的能力集合，Meta-Skill使 s_t 的组合方式对任务类型可感知。

Meta-Skill Creator进一步解决“系统如何从历史协作中学习新的组织方式”。传统技能系统的能力增长依赖人工开发，难以覆盖真实场景的长尾任务；而实践中许多复杂任务会反复形成相似的多Skill协作模式，若不能沉淀，每次仍需模型重新规划。Creator从历史执行轨迹中观察Skill之间的共现、顺序与依赖关系，其进化过程分为四个阶段：观察，记录用户目标、多Skill调用链、中间结果、分支选择与输出质量；抽象，从重复轨迹提炼稳定的composition、routing、输入需求与输出策略；验证，检查候选工作流的结构完整性、触发边界、routing稳定性、输出质量与风险；沉淀，将通过验证的候选保存为Meta-Skill proposal，经审查或保守启用后进入技能体系。需要强调，Creator生成的是proposal而非直接上线的能力：工作流一旦固化就会影响系统未来面对同类任务的执行方式，若缺少治理，偶然路径、过宽触发条件或高风险操作可能被沉淀为长期行为，因此候选必须通过结构检查、正反样例验证、风险评估与必要的人工审查。由此，Skill提供原子能力，Meta-Skill提供编排能力，Creator提供编排模式的生成与演化能力；三者合力使 s_t 从固定prompt / RAG操作演化为随用户场景自适应的可学习编排策略，这正是“可学习Harness”的核心含义。

4.2. 持久记忆系统

当Agent从单轮问答走向长周期协作时，用户偏好、项目事实、历史决策与复用格式需要跨会话持续有效，而完整会话流水、工具返回与临时指令又不能无差别进入未来上下文，否则会造成噪声累积、过期事实残留与持久化prompt injection风险。OPENSQUILLA因此把记忆设计为独立的长期状态层而非「向量库+全量会话日志」，并遵循两个边界：记忆与技能分离——Skill描述Agent如何完成任务，Memory描述Agent未来应继续知道什么；长期记忆与审计记录分离——可复用事实整理为curated memory，完整会话与原始turn记录保留为审计或派生检索材料，不直接污染普通召回路径。在目标式(2)中，记忆系统是 s_t 的跨会话状态基底：

它让Harness操作能够感知历史事实与成功路径，而不是每轮从空白上下文重新组织任务。

体系结构上，持久记忆抽象为四层生命周期（表3）。源层限定哪些材料可以成为记忆：可编辑的Markdown长期记忆文件、由已持久化会话派生的虚拟检索文档，以及默认不进入普通索引的私有turn捕获。索引层以本地关系数据库为底座，将记忆按内容哈希去重并切分为带行号范围的片段，建立全文索引与可选向量索引；嵌入后端、模型或维度变化会触发索引安全重建，避免不同嵌入空间的向量被错误混用。召回层在每次检索前执行查询前同步以保证索引一致，再做混合召回与重排。巩固治理层通过Session Flush、Memory Dream与清理策略管理记忆演化。语义表示采用本地优先策略：默认在本机完成量化嵌入推理，长期记忆不默认外传第三方服务；本地后端不可用时退化为纯全文检索。

Table 3 | OPENSQUILLA持久记忆系统的四层职责划分

层次	系统角色	主要职责	退化语义
源层	可编辑记忆、会话片段、私有转写	区分长期事实、历史会话证据与审计原始记录，避免原始流水直接污染普通召回。	保留文件或会话记录，但不进入普通检索。
索引层	本地全文与语义索引	将记忆源切分为可检索片段，建立全文索引与可选向量索引，并维护嵌入缓存和索引一致性。	向量不可用时退化为全文检索。
召回层	混合检索与重排	查询前同步索引，融合语义召回和词法召回，并用时间衰减、MMR与词法保底提升稳定性。	保留关键词召回能力。
巩固治理层	Flush、Dream与清理策略	将长会话蒸馏为候选记忆，通过证据门控提升长期事实，并执行TTL、脱敏、回滚与审计。	采用dry-run / archive-only或跳过写入。

召回层的目标是在有限prompt预算内返回最相关且尽量不重复的记忆片段，采用向量语义召回与词法召回的混合策略：

$$s_{\text{hyb}}(q, d) = w_v s_{\text{vec}}(q, d) + w_l s_{\text{lex}}(q, d), \quad w_v = 0.7, w_l = 0.3, \quad (16)$$

向量通道捕获语义相近的历史事实，词法通道保证专有名词、路径、日期、错误码与项目术语被精确命中；向量后端不可时令 $w_v=0$ 、 $w_l=1$ ，自动退化为纯全文检索。在此之上有三重稳定化机制：词法保底允许全文强匹配片段进入最终候选，避免语义分数压制明确的关键词命中；日期型记忆按30天半衰期做指数时间衰减，根级MEMORY.md与非日期型长期记忆视为evergreen不参与衰减，使临时任务自然下沉而稳定事实保持可见；MMR重排（相关性-多样性权重 $\lambda_{\text{MMR}} = 0.7$ ，以片段间Jaccard相似度度量冗余）约束结果多样性，避免近重复片段挤占召回预算。

沉淀侧由两级机制构成。Session Flush在会话重置、归档或上下文压缩前，把长会话蒸馏为事实、偏好、决策、流程、待办等候选记忆，并通过受限写入上下文保存；当LLM抽取超时或结果无法安全解析时退化为仅归档原始材料——宁可保留可审计证据，也不把低置信摘要提升为长期事实。Memory Dream是周期触发的离线巩固：每次只扫描增量候选，跳过私有目录与高风险文本，按出现频率、正负信号平衡、来源可信度与跨日期巩固度的加权证据分数门控提升；提升不允许模型重写整个记忆库，而是生成受约束的增量修改（插入/合并/跳过），修改根级索引前创建备份，每次运行写入可回放的收据。写入侧把记忆写入视为长期上下文治理操作：目标必须位于受控记忆源路径内，写入前执行脱敏、注入扫描与容量检查，拦截密钥、典型prompt injection与不可见控制字符，并采用快照与回滚语义；TTL与容量清理不触碰MEMORY.md、bootstrap文件与私有目录，单次清理设删除上限。与朴素的“全量历史+向量召回”相比，该设计更强调来源边界、索引可重建性、召回可解释性与巩固可审计性，使 s_t 能在跨会话时间尺度上积累并复用经验，而不被历史噪声、过期信息或恶意内容长期劫持。

4.3. 安全治理

工具型Agent 的安全风险不止于模型文本中的“违规回答”：一次看似普通的工具调用可能删除工作区文件、覆盖配置、泄漏密钥、访问网络资源，或把恶意指令写入长期记忆与候选workflow 并在后续会话中再次触发。借鉴现有Agent 安全评测所考察的攻击面，本文将攻击族概括为六类（该分类及缩写为本文自定义）：直接与间接提示注入（DPI / IPI）、工具返回注入（TRI）、记忆污染与记忆抽取（MPI / MEX）以及歧义委派诱导（ADI）——前三类偏输入来源，中间两类涉及持久状态，最后一类源于用户意图与执行权限边界之间的歧义。这一划分意味着Agent 安全不能只依赖prompt-level policy，而必须在运行时引入最小权限、系统级隔离、审计闭环与状态清理。OPENSQUILLA的安全设计遵循四个原则：默认最小权限，每次工具调用只获得完成当前任务所需的文件、网络与进程能力；策略与执行分离，模型可以提出行动计划，是否执行由运行时依风险提示、路径范围与审批状态决定；拒绝可审计，所有被拒动作按操作指纹累计形成否决账本；状态不可复用，被拒绝或过期的工具输出不可被后续步骤引用。从目标式 (2) 看，安全治理是对可执行(g_t, s_t) 组合的域限制： s_t 越强大，其运行时授权边界就需要越精确。

运行时按操作来源、作用范围与潜在影响动态选择安全档位，判定覆盖信任来源、网络需求、工作区外写入、信任边界跨越与高影响操作五个维度，表 4 给出三档策略及其运行时动作。这种分档把“是否相信模型判断”转换为“运行时允许什么系统能力”：来自外部网页、邮件或工具返回值的指令即使语义上看似合理，也不会自动获得写文件或联网权限；涉及密钥、配置、删除、覆盖与跨目录写入的动作被提升至STRICT 或LOCKED，由沙箱和审批共同约束。换言之，路由后的 $g_t \circ s_t$ 组合只有在权限、安全与预算谓词均满足时才进入真实执行路径。

Table 4 | 三档安全策略与运行时动作

级别	触发条件	运行时动作
STANDARD	信任源、工作目录内读写、无网络或高影响操作	直接执行，并记录工具摘要与输出指纹
STRICT	不信任输入、跨工作目录读取、写入宿主路径、工具返回携带指令	启用沙箱隔离，限制挂载与输出，必要时触发用户审批
LOCKED	跨信任边界、网络访问、删除/覆盖关键文件、凭据相关或不可逆操作	强制人审，使用最严格沙箱配置；连续拒绝后暂停自主执行

系统级隔离在支持命名空间或平台沙箱的系统上（Linux 上基于Bubblewrap，macOS 上生成Seatbelt profile）采用默认拒绝策略：只暴露必要运行时与工作目录，宿主系统目录以只读方式进入沙箱，网络访问与进程能力默认关闭，工具输出受大小与超时约束，路径挂载经过规范化校验以防跨目录访问与路径遍历。隔离能力存在平台差异，因此沙箱状态被显式纳入运行时诊断：完整系统级隔离不可用时，系统退化为权限profile、工作区约束、敏感路径防护、交互式审批、否决账本与工具输出失效的组合防线——本文将其视为受控降级而非完整隔离。审计与状态清理侧，否决账本在连续拒绝达到阈值后自动暂停自主执行，防止Agent 通过反复改写命令、路径或参数试探安全策略；过期输出缓存机制在拒绝判定后立即清除同指纹的工具输出缓存，封堵“先在低风险语境获得敏感输出、再读取上次输出”的绕过路径；输入侧对技能描述、工具返回与外部内容统一信封封装并转义处理，长期记忆中的外部来源与模型生成内容按低信任数据处理，避免记忆条目中的指令被误提升为系统级规则。

安全评测方面，结果不能只用一个笼统的攻击成功率（ASR）描述：语义上服从攻击、日志与文件轨迹中证据支持的真实伤害、以及可执行沙箱中真实发生的状态改变是三个不同端点，应分别报告并给出明确的分母与原始计数；人工构造的hard-biased 攻击集只解释为压力测试下的相对表现。据此，OPENSQUILLA的安全主张被限定在运行时防护与可审计执行边界上：通过三档策略、系统级沙箱、否决账本与过期输出清理降低工具型Agent 产生实际伤害的概率；prompt-only 防御与复杂策略包只作为诊断性组件，而不被表述为总是更优的生产级防

御。

5. 系统架构

OPENSQUILLA的系统架构把AI Agent Harness 拆成清晰的微内核与用户态能力。微内核负责稳定的turn 生命周期、显式状态机、并发控制和事件流；用户态能力负责多模型池、工具、记忆、技能、Meta-Skill、渠道、调度和沙箱等可演化策略。这一划分正是Agent = 基础模型池 $\mathcal{M}$ + Harness 的系统化实现：微内核提供稳定执行骨架， $g_t(\mathcal{M})$ 与 s_t 作为可替换、可学习、可治理的策略注入到turn 生命周期中。

这种架构的优势体现在三个方面。第一，能力扩展不会破坏核心执行路径，新的模型供应商、消息渠道、工具、技能、Meta-Skill 或记忆策略可以通过边界接入。第二，故障具有局部性，模型失败、工具拒绝、渠道异常、记忆退化或技能proposal 未通过都能在对边界内被处理。第三，系统可观测性更强，一次复杂Agent 行为可以被拆解为输入、能力解析、提示装配、上下文治理、编排执行、流式执行和最终化等阶段记录。由此，OPENSQUILLA形成了适合长周期、多入口、高风险工具调用和持续技能演化场景的Agent-native microkernel harness。

5.1. 设计原则：机制与策略分离

OPENSQUILLA的系统架构采用微内核思想。这里的“微内核”并不是指系统能力很少，而是指Harness 核心只保留最小且稳定的执行机制：会话级任务编排、状态机推进、流式事件分发、工具调用边界、错误终止语义和生命周期管理。模型选择、工具集合、记忆检索、技能注入、Meta-Skill 编排、渠道收发、定时任务与沙箱隔离等策略性能力，则通过协议化接口、注册表、适配器或服务形式接入核心之外。

这种设计延续了操作系统中的“机制与策略分离”思想。核心运行时不直接固化某个模型供应商、某个渠道协议或某种记忆策略，而是提供稳定的执行时序和故障边界；外部能力围绕这些边界独立演化。一次Agent turn 可以抽象为如下组合：

$$\mathcal{T} = F_{\text{finalize}} \circ F_{\text{stream}} \circ F_{\text{orchestrate}} \circ F_{\text{context}} \circ F_{\text{prompt}} \circ F_{\text{capability}} \circ F_{\text{input}}. \quad (17)$$

其中，输入归一化、能力解析、提示装配、上下文治理、编排调度、流式执行和最终持久化构成稳定内核路径；模型路由、技能过滤、Meta-Skill 编排、记忆召回、安全策略和成本统计则作为可替换策略注入到相应阶段。具体而言， $F_{\text{capability}}$ 与 F_{prompt} 承载 $g_t(\mathcal{M})$ 的选择与 s_t 的装配， $F_{\text{orchestrate}}$ 承载Meta-Skill 对 s_t 的结构化， F_{finalize} 则将本轮轨迹沉淀为未来 s_t 的输入。这个分解使系统能够把“模型生成文本”的过程提升为“可控、可观测、可恢复、可演化的任务执行过程”。

5.2. 阶段化Turn 执行

许多Agent 框架在早期实现中倾向于把输入处理、模型选择、prompt 拼接、工具调用、历史写入和错误处理集中在一个长主循环中。该模式适合快速原型，但在生产系统中会带来两个问题：第一，任一能力扩展都可能影响整条执行路径；第二，错误、取消、压缩、工具调用和持久化等边界难以独立测试。

OPENSQUILLA将一次turn 拆分为若干稳定阶段。每个阶段只接收明确定义的输入，产生明确定义的输出，并通过窄接口访问外部依赖。表 5 给出阶段化执行的抽象视图。

阶段化设计的优点在于，它把“可扩展”建立在清晰边界上，而不是依赖全局变量或隐式调用顺序。模型路由由主要影响提示装配和provider 选择；记忆系统主要参与提示装配与最终化；安

Table 5 | OPENSQUILLA Turn 执行路径的阶段化抽象

阶段	核心问题	技术作用
输入归一化	当前请求是什么	将Web、CLI、渠道和定时任务入口统一为运行时消息、语义输入、附件和来源元数据。
能力解析	本轮能使用什么	解析模型供应商、工具可见性、工具策略上下文和运行时能力边界。
提示装配	模型应该看到什么	合成身份提示、工具定义、技能、记忆、路由元数据与缓存边界。
上下文治理	历史如何进入本轮	加载会话历史，执行必要压缩，避免历史、附件和工具结果超过上下文预算。
编排调度	任务如何组织能力	根据Meta-Skill、技能匹配、用户澄清和checkpoint 状态决定 workflow 步骤与恢复路径。
流式执行	模型与工具如何交替	驱动模型流式输出、工具调用、工具结果回注、路由回放和中途错误处理。
最终化	本轮如何落盘	持久化assistant 输出、工具片段、错误、记忆捕获、token 与成本统计。

全策略主要作用在工具调度和沙箱边界；成本统计则在最终化阶段统一落盘。不同能力之间通过阶段输出交换必要信息，从而降低跨模块耦合。

5.3. 显式状态机与流式事件模型

在turn 内部，OPENSQUILLA并不把模型调用视作一次阻塞式RPC，而是把它建模为显式状态机。一个典型生命周期包括：

IDLE → THINKING → STREAMING → (TOOL_CALLING → THINKING)* → DONE/ERROR.

该状态机支持模型文本增量、工具调用开始、工具参数增量、工具调用结束、工具结果、心跳、完成事件和错误事件等多种流式事件。

这一设计的核心价值在于将Agent 执行过程中的不确定性转化为可观察状态。长任务可能经历多次工具调用，provider 可能中途报错，用户可能取消任务，工具可能返回过大结果，输出可能触发上下文压缩。若这些事件只隐藏在同步调用或异常栈中，系统很难做稳定恢复；显式事件模型则允许运行时对每类事件分别施加策略，例如重试、降级、压缩、持久化局部结果或生成终止错误。

因此，OPENSQUILLA的Agent 内核不是简单的ReAct 循环，而是一个面向生产运行的事件驱动状态机。它保留了“推理-行动-观察”的交互范式，同时将工具执行、流式输出、预算控制、取消恢复和错误审计纳入统一运行时。

5.4. 统一工具调度边界

工具调用是Agent 系统中最容易失控的部分：模型生成的一个工具名和参数，可能对应文件写入、命令执行、网络访问、消息发送或外部API 调用。OPENSQUILLA的一个重要设计，是没有让工具函数直接暴露给模型调用，而是在模型和工具之间建立统一调度边界。

该边界包含四个关键步骤。第一，工具名必须先经过注册表查验，未注册工具不会被执行。第二，工具调用进入策略链，结合调用来源、权限上下文、工具可见性、预算和安全规则判定是否允许执行。第三，实际执行发生在请求级上下文中，使工具能够感知当前会话、Agent、工作区和调用来源。第四，工具结果统一进入最终化层，附加执行状态、截断状态、产物元数据和错误信封。

用集合表示，本轮实际可执行工具集合不是全局工具集 $\mathcal{U}$ ，而是由上下文过滤后的子集：

$$\mathcal{U}_t = \{u \in \mathcal{U} \mid V(u, c_t) = 1 \wedge P(u, a_t, c_t) = 1\}, \quad (18)$$

其中 V 表示可见性约束， P 表示权限、安全和预算约束， c_t 是本轮上下文， a_t 是模型提出的动作。这个设计把“模型想调用什么”和“运行时允许调用什么”分离开来，是安全沙箱、权限控制和成本预算能够生效的基础。

5.5. 跨入口一致性与插件化能力边界

OPENSQUILLA的扩展性主要来自三类边界：模型供应商、外部渠道和能力插件。

模型供应商通过统一流式接口接入。不同provider只需将自身API转换为文本增量、工具调用、完成和错误事件，核心状态机无需感知底层协议差异。这使模型路由、fallback和成本统计可以运行在provider抽象之上。

外部渠道通过统一消息模型接入。渠道适配器负责把不同平台入口转为标准输入消息，并把系统输出转回平台消息。Harness核心只处理归一化后的会话标识、Agent标识、发送者、附件和来源信息，不需要关心平台特有字段。由此，渠道语义的变化可以在适配器和入口归一化层修复，而无需重写Agent内核。

技能、Meta-Skill与记忆作为prompt、编排和上下文能力接入。技能系统提供可复用任务知识，Meta-Skill提供可检查的工作流组织方式，记忆系统提供跨会话状态，三者都不改变核心状态机，而是在提示装配、编排调度和最终化阶段注入或沉淀信息。这样的设计保证了长期能力增长不会导致内核复杂度线性增加，也使Harness能够在治理边界内持续学习新的任务组织模式。随着 s_t 从静态配置演化为由Meta-Skill Creator、Memory Dream和诊断回放驱动的学习型策略，微内核边界确保这种演化保持可审查、可回放和可治理。

5.6. 并发模型：同会话串行，跨会话并行

生产级Agent框架必须同时处理多入口和长任务。若所有请求全局串行，系统吞吐量会被单个慢任务拖垮；若所有请求完全并行，同一会话内的多轮输入又可能交叉写入历史、记忆和成本统计，导致状态错乱。OPENSQUILLA采用“同会话串行、跨会话并行”的并发模型。

设 s 表示session， t_i^s 表示该session中第 i 个turn，则系统希望满足：

$$t_i^s \prec t_{i+1}^s, \quad t_i^{s_a} \parallel t_j^{s_b} \quad (s_a \neq s_b). \quad (19)$$

也就是说，同一session内保持顺序，不同session间允许并行。运行时区分长执行锁和短写锁：长执行锁保护同一会话的turn生命周期，短写锁仅保护transcript和session state的临界写入。这样，模型流式输出、工具执行和等待外部响应不会长期占用状态写锁，降低死锁和队头阻塞风险。

该并发模型也支持渠道型Agent的实际需求。消息渠道可能在短时间内触发多条输入，定时任务和子Agent也可能同时运行。通过队列上限、取消语义、超时语义和生命周期事件，系统可以在高并发入口下维持可控背压，而不是让任务无限堆积。

5.7. 生命周期治理与故障隔离

微内核架构的价值不只在扩展方便，更在于故障隔离。系统启动时，组合根负责构造配置、会话存储、模型选择、工具注册、记忆管理、技能加载、调度、搜索和渠道管理等依赖；系统关闭时，则按依赖顺序停止后台生产者，再关闭记忆、会话和存储资源。这个顺序避免了后台任务仍在写入，而底层数据库或记忆索引已经关闭的竞态问题。

故障隔离也遵循相同思路。模型供应商不可用会返回稳定的终止事件；工具越权会返回结构化拒绝结果；渠道启动失败不会阻塞其他渠道；记忆向量后端不可用时可退化为全文检索；技能加载失败不会破坏主状态机。表 6 总结了这些边界。

Table 6 | OPENSQUILLA架构中的主要隔离边界

边界	设计目标	退化方式
模型供应商边界	隔离模型供应商差异、错误和回退策略。	无可用供应商时返回稳定错误；可按策略回退。
Tool 边界	隔离模型意图与真实系统动作。	越权或不可用时返回结构化拒绝信封。
Memory 边界	隔离长期状态与当前turn 执行。	向量不可用时保留全文召回。
Channel 边界	隔离不同外部消息平台。	单渠道失败不影响Web、CLI 或其他渠道。
Scheduler 边界	隔离后台任务与交互式会话。	任务失败记录执行结果，不破坏主服务。
Sandbox 边界	隔离工具执行与宿主系统资源。	后端不可用时显式降级并记录风险。

在实际实现中，格式异常或不合法的工​​具调用历史应在模型供应商边界之前被拦截，而不是等远端模型接口返回不可恢复错误后再处理。类似地，工作流交接、Meta-Skill 澄清和渠道回复上下文等语义修复都应落在入口归一化、能力编排和适配器层，而不是侵入核心状态机。

5.8. 工程可观测性

OPENSQUILLA的架构并不只追求模块拆分，也强调可观测性。一次复杂Agent 行为会被分解为输入、能力解析、提示装配、上下文治理、流式执行和最终化等阶段记录。最终事件不仅包含文本输出，也包含工具片段、产物、错误类型、模型、供应商、缓存token、成本来源和记忆相关元数据。由此，系统可以复盘一次turn 的关键决策，而不是只保存一段最终自然语言回答。

可观测性对微内核架构尤其重要。由于模型路由、工具策略、记忆召回、技能过滤和沙箱隔离分布在不同边界上，如果没有统一的轨迹与决策日志，系统很难判断一次失败究竟来自模型能力、工具拒绝、上下文溢出、记忆失配还是渠道适配问题。通过阶段化记录，OPENSQUILLA将复杂Agent 行为转化为可定位、可回放、可回归测试的工程对象。

6. 实验

为了验证OpenSquilla 作为Agent-native learnable harness 在真实任务中的端到端表现，我们在业界常用的标准Benchmark 上进行了测评，并与代表性Agent 框架/ harness 进行对比。实验设计对应目标式 (2)：以外部harness 的单模型直跑提供近似固定 g_t 的基线，以OPENSQUILLA 强制单模型隔离微内核与 s_t 的贡献，再以OPENSQUILLA 多档路由测量 g_t 与 s_t 联合优化对成本-能力Pareto 前沿的影响。在此之外，Section 2 所述agentic routing 的单模型路由 ($k = 1$) 专项对照嵌入§6.1，多模型融合路由 ($k > 1$) 的专项实验见§6.5。

6.1. PinchBench

PinchBench¹ 是面向代码型Agent 真实任务能力的评测基准，强调“end-to-end practical outcomes”而非合成的孤立能力测试。本工作采用固定版本的公开任务集与评测器；25 条任务覆盖文件操作、数据处理、网页检索、创作输出、工具使用与记忆召回六类典型Agent workflows，同时考察工具调用准确性、多步推理链与现实任务的歧义处理能力。每条任务的grader 采用自动化校验、LLM-as-a-judge 或两者组合，输出[0,1] 区间分数；表中总分为跨任务平均分乘以100。

为隔离“模型能力”与“框架贡献”两类因素，我们在统一供应商计价口径下对照三个Agent 框架：OpenClaw、Hermes Agent 与OPENSQUILLA。前两者为单模型直跑的外部harness 基线，分别覆盖旗舰模型与高性价比模型。OPENSQUILLA 既报告强制单模型（同一模型走完整框架但禁用路由，用以测量微内核运行时相对模型直跑的影响），也报告多档路由（分别代表“旗舰模型兜底”与“低成本模型优先”两种成本取向）。Token 用量与成本按相同价目表逐次累计。

表 7 汇总三个框架在不同模型（池）下的总分与总成本。整体看，在相同旗舰模型Claude Opus-4.7（下文简称Opus-4.7）下，OPENSQUILLA 取得最高的93.85 分，较OpenClaw（92.55）与Hermes Agent（92.65）分别提升约+1.3pp 与+1.2pp，且成本反而从\$6.23 / \$6.66 降到\$4.84，说明微内核运行时与cache / Skill 等 s_t 操作在不牺牲质量的前提下已能压缩成本。多档路由则形成更清晰的成本前沿：以Opus-4.7 兜底的档位组合以92.51 分基本贴近Opus 直跑，同时把成本降到\$0.69；以GLM-5.1 作顶档的低成本组合总分仍达90.48，成本进一步降至\$0.13。从目标式 (2) 看，这意味着 $P(a)$ 大幅下降而 $\mathcal{L}_T(a)$ 基本稳定，验证了路由 g_t 与cache / Skill 等 s_t 操作的联合价值。

Table 7 | PinchBench 25 任务对照表。对比OpenClaw、OPENSQUILLA 与Hermes Agent 三个框架在不同模型（池）下的总分与成本。OpenRouter Auto 为OpenRouter 官方自动路由服务基线，其结果测于路由专项口径（PinchBench 1.2.1，单任务平均成本\$0.1204 按25 任务折算为总成本），与其余各行不直接可比。

框架	模型（池）	总分	成本(\$)
OpenClaw	Opus-4.7	92.55	6.23
OpenClaw	GLM-5.1	88.33	1.60
OpenClaw	OpenRouter Auto	88.10	3.01
Hermes Agent	Opus-4.7	92.65	6.66
OPENSQUILLA	Opus-4.7	93.85	4.84
OPENSQUILLA	{DeepSeek-V4 Flash, DeepSeek-V4 Flash, GLM-5.1, Opus-4.7}	92.51	0.69
OPENSQUILLA	{MiniMax M2.5 (free), DeepSeek-V4 Flash, DeepSeek-V4 Flash, GLM-5.1}	90.48	0.13

在上述harness 整体对照之外，我们在PinchBench 上另行开展了Section 2 所述agentic routing 的单模型路由专项评测。该组实验与表 7 主体口径不同、不直接可比：使用PinchBench 版本1.2.1，模型池以Opus-4.8 世代为主（路由器在Opus-4.8、GLM-5.2 与DeepSeek-V4 Flash 之间逐步选择），成本按单任务平均计费统计；同一口径内harness、任务划分与评分协议保持一致。该口径下，路由策略以93.14 分几乎持平OpenClaw 固定Opus-4.8 基线（93.35，质量保持率99.77%），单任务成本却从\$0.2224 降至\$0.0204，约10.9 $\times$ 的降幅；OPENSQUILLA 内固定Opus-4.8 则取得94.33 分、\$0.1649。作为query 级路由基线，OpenRouter Auto（OpenRouter 官方自动路由服务，运行于OpenClaw harness）取得88.10 分、单任务\$0.1204（折算总成本已列入表 7）；相对该基线，路由策略同时高出5.04 分并便宜约5.9 $\times$ ，说明以harness 状态为条件的步级路由显著优于仅看query 的路由服务。

¹<https://github.com/pinchbench/skill>

6.2. ClawMark

ClawMark² 是一个面向垂直行业Agent 任务的100 题综合评测集合，任务覆盖13 个真实业务域，包括临床助理、内容运营、电商、EDA、高管助理、人力资源、保险、投资分析、新闻、法律助理、产品管理、房地产与研究助理等场景。不同于只考察问答或代码修复的基准，ClawMark 更强调行业场景中的多步信息处理、表格/文档读取、视觉材料理解、结构化输出与业务约束遵循。每个任务由独立grader 给出[0,1] 区间分数，本文将分数大于等于0.5 的任务记为完成。

为隔离“模型能力”与“框架贡献”两类因素，我们在统一供应商计价口径下对照OpenClaw 与OPENSQUILLA 两个框架：OpenClaw 以GLM-5.1 单模型直跑作为baseline；OPENSQUILLA 既报告强制单模型（同一模型走完整框架但禁用路由），也报告多档路由（成本侧档位组合）。为保证不同系统读取图片、PDF 页面和截图的入口一致，所有配置均使用相同的人工视觉解析流程：视觉材料先转成本文观察，再交给对应agent；该视觉模型不作为表格中的路由档位展开。

表 8 汇总两个框架在不同模型（池）下的总分与单任务平均成本。整体看，OPENSQUILLA 以GLM-5.1 强制单模型取得77.0 分，领先全部配置，相对OpenClaw 的GLM-5.1 直跑（71.2）提升+5.8pp；路由配置则更接近成本侧Pareto 点：以DeepSeek-V4 Pro 作中档的组合以70.6 分接近GLM-5.1 直跑，同时把单任务成本从\$0.62 降至\$0.39。该结果说明，在行业场景任务上，固定模型下的 s_t 能提升任务完成质量，而启用 g_t 后系统可沿成本侧移动，形成更可控的 $\mathcal{L}_T(a)-P(a)$ 折中。

Table 8 | ClawMark 100 任务对照表。对比OpenClaw 与OPENSQUILLA 在不同模型（池）下的总分与成本。所有分数为百分制（ $\times 100$ ）；成本为单任务平均计费成本。

框架	模型（池）	总分	成本(\$)
OpenClaw	GLM-5.1	71.2	0.62
OPENSQUILLA	GLM-5.1	77.0	0.87
OPENSQUILLA	{MiniMax M2.5 (free), DeepSeek-V4 Pro, GLM-5.1, GLM-5.1}	70.6	0.39

6.3. ClawSWEBench

ClawSWEBench [46]³ 是面向Agent harness 评测的多语言SWE-bench 风格基准，包含350 条来自真实GitHub Issue 的修复任务，覆盖8 种编程语言与43 个仓库；其评测协议将任务集、Docker 运行容器、Prompt 模板、最大轮数与超时上限全部固定，仅留出harness 实现作为唯一受控变量，从而将harness 层面的差异从LLM 能力与任务难度中分离。所有harness 共享相同任务Prompt、评测镜像、基准提交、工作目录、最大交互轮数、墙钟超时与单实例一次执行约束；候选补丁由benchmark runner 统一暂存并抽取，再交由evaluator 判定Resolved / Unresolved / Empty Patch 三态。表中解决率为Resolved 实例占350 的比例，成本为单任务平均计费成本（由官方榜单Cost/350 口径折算）。

为隔离“模型能力”与“框架贡献”两类因素，我们在上述统一协议下对照三类系统：

(i) **OpenClaw** 单模型直跑——Opus-4.7、GLM-5.2、GLM-5.1 与Qwen3.7-Max 四个主流模型各自独立完成全部350 任务，作为模型能力基线；(ii) **OPENSQUILLA** 强制单模型——同一模型走完整OPENSQUILLA 框架但禁用路由，选取GLM-5.2 与GLM-5.1 [42] 两个能力档位，用以测量纯框架影响；(iii) **OPENSQUILLA** 多档路由——采用{DeepSeek-V4 Flash, GLM-5.1, GLM-5.2} 三档组合，由Section 2 所述本地路由器在每条任务上动态分档，测量低档下沉与顶档

²<https://github.com/evolvent-ai/ClawMark>

³<https://claw-swe-bench.github.io/>

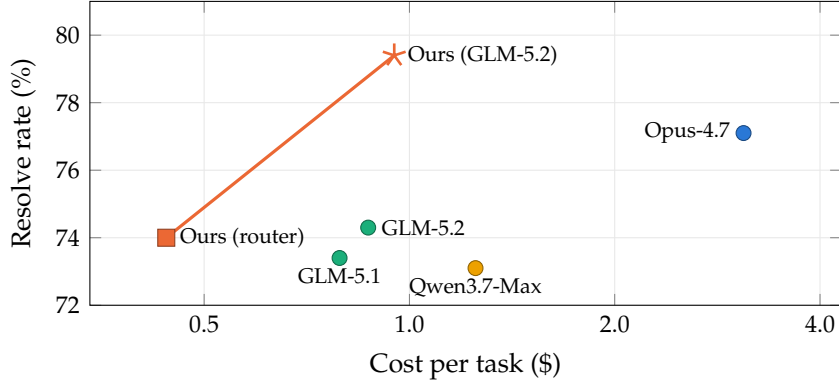


Figure 4 | ClawSWEBench 成本-解决率视图。彩色圆点为OpenClaw 单模型直跑基线，颜色对应模型系列（蓝色Opus、青绿色GLM、黄色Qwen）；橙色标记（以线相连）为OPENSQUILLA 的两个工作点——星形为强制单模型GLM-5.2（最高质量点），方块为多档路由（最低成本点）；成本轴为对数刻度。

兜底对成本-解决率前沿的影响。

表 9 汇总各配置在350 任务上的解决率与单任务平均成本。整体看，OPENSQUILLA 以GLM-5.2 强制单模型取得278/350（79.4%）的最高解决率，较OpenClaw 的GLM-5.2 直跑（260/350，74.3%）提升+5.1pp，且以约31%的成本（\$0.95 vs \$3.09）超过OpenClaw 最强单模型Opus-4.7（77.1%）；GLM-5.1 进入框架同样获得+1.4pp 提升，说明OPENSQUILLA 的工具边界与会话治理能在不更换模型的前提下稳定放大代码修复能力。多档路由则给出更优的成本侧Pareto 点：259/350（74.0%）的解决率与OpenClaw 的GLM-5.2 直跑基本持平，成本却压缩至后者的一半（\$0.44 vs \$0.87）。路由分布显示，207 条任务（59%）被下沉至低价的DeepSeek-V4 Flash 并解决其中143 条（69.1%），139 条上浮至顶档GLM-5.2 并解决113 条（81.3%），其余4 条落在中档GLM-5.1——即路由器把大部分预算留给了确实需要顶档能力的难题。从目标式 (2) 看，固定模型下的 s_t 在中高档模型上直接降低 $\mathcal{L}_T(a)$ ，而 g_t 在几乎不损失解决率的情况下将 $P(a)$ 减半，两者共同扩展了代码修复场景下的成本-能力前沿。图 4 给出对应的成本-解决率视图：OPENSQUILLA 的两个工作点均不被任何OpenClaw 基线支配——强制单模型GLM-5.2 占据最高质量点，多档路由则以最低成本立于前沿最左端。

Table 9 | ClawSWEBench 350 任务对照表。对比OpenClaw 与OPENSQUILLA 在不同模型（池）下的解决率与成本。成本为单任务平均计费成本（由官方榜单Cost/350 口径折算）；Opus-4.7 解决率取自官方榜单。

框架	模型（池）	解决率(%)	成本(\$)
OpenClaw	Opus-4.7	77.1	3.09
OpenClaw	GLM-5.2	74.3	0.87
OpenClaw	GLM-5.1	73.4	0.79
OpenClaw	Qwen3.7-Max	73.1	1.25
OPENSQUILLA	GLM-5.2	79.4	0.95
OPENSQUILLA	GLM-5.1	74.9	0.87
OPENSQUILLA	{DeepSeek-V4 Flash, GLM-5.1, GLM-5.2}	74.0	0.44

6.4. DRACO

DRACO [47]⁴ 是一个跨域深度研究基准，包含100个复杂研究任务，由grader沿事实准确性、完整性、客观性、呈现质量与引用质量五个维度打分；其证据检索、分析综合与引用生成交织的长执行轨迹，对harness的上下文治理与模型能力分配均构成挑战。与常见的问答或代码基准相比，DRACO更能反映Agent在长程研究型任务上的端到端表现。

我们沿用前三小节的对照逻辑，在统一任务集与评分协议下隔离“模型能力”与“框架贡献”两类因素：OpenClaw以固定Opus-4.8单模型直跑作为外部harness基线；OPENSQUILLA既报告强制单模型（同一Opus-4.8走完整框架但禁用路由），也报告多档路由（{DeepSeek-V4 Pro, GLM-5.2, Opus-4.8}，由Section 2所述本地路由器逐步分档）。成本按单任务平均计费统计。

表 10 汇总各配置在100任务上的总分与成本。整体看，OPENSQUILLA强制单模型以52.36分略超OpenClaw的Opus-4.8直跑（52.13），成本却从\$1.1420降至\$0.6559——说明框架自身的工具边界与上下文治理在深度研究任务上已带来正贡献。启用多档路由后，系统保持强模型配置99.94%的质量（52.33 vs 52.36），成本进一步降至\$0.3729（相对强模型配置再降43.15%，相对OpenClaw直跑累计降低67.3%）。值得注意的是，路由配置的token用量反而略多于固定配置——节省并非来自缩短prompt，而是改变了模型调用的价格构成：执行轨迹中能力足够的部分被下沉给更便宜的模型。该结果与§6.1的单模型路由专项对照共同支持subsection 2.2的能力匹配观点：模型能力可以在harness内部按状态选择性分配，而不必把每个执行状态都当作需要最强模型。

Table 10 | DRACO 100 任务对照表。对比OpenClaw与OPENSQUILLA在不同模型（池）下的总分与成本。总分为跨任务平均分；成本为单任务平均计费成本；token为输入加输出（千）；路由阈值设为0.95。

框架	模型（池）	总分	成本(\$)	Tokens (K)
OpenClaw	Opus-4.8	52.13	1.1420	54.2
OPENSQUILLA	Opus-4.8	52.36	0.6559	103.5
OPENSQUILLA	{DeepSeek-V4 Pro, GLM-5.2, Opus-4.8}	52.33	0.3729	108.6

6.5. 多模型融合路由实验

前述各小节从harness整体视角比较OPENSQUILLA与外部框架；本小节对subsection 2.3所述多模型融合路由（ $k > 1$ ）做专项评测，检验互补proposer集合与聚合能否相对强单模型基线移动质量–成本前沿。本组实验的全部配置——包括各单模型基线与融合配置——均运行在同一OPENSQUILLA harness中，保证执行底座一致。DRACO是主基准，因为深度研究任务往往受益于互补的模型强项：一个模型可能检索到更好的证据，另一个综合出更连贯的分析，第三个产出更可靠的引用。选定配置以DeepSeek-V4、GLM-5.2、Gemini 3 Flash与Qwen3.7-Max为proposer，GLM-5.2为聚合器，并对便宜的Gemini与Qwen proposer附加随机采样。

表 11 报告DRACO（默认DuckDuckGo搜索）上的单模型基线与选定多模型配置。单模型中最强的质量点是Fable 5，在其执行的94个任务上平均59.80分、单任务成本\$1.2122；本文多模型配置在完整100任务上取得60.82分，平均成本\$0.3766——相对Fable 5质量提升1.02分、成本降低68.9%。这正是式 (10) 前沿形式化所预言的省钱替代机制：互补的便宜proposer加聚合器可以在质量与货币成本两个维度上同时超过最强单模型基线。作为集成策略的预期代价，该工作点消耗更多token与墙钟时间；延迟是正交的部署轴，可通过proposer并行执行、早停与更严格的proposer预算控制。与mixture-of-agents类基线对比，Hermes MoA（预设式虚拟模型供应商，参考模型先产出私有建议、固定聚合模型执行动作 [24]）

⁴<https://huggingface.co/datasets/perplexity-ai/draco>

取得59.55分、\$0.4460，本文配置在两个维度上均更优（60.82，\$0.3766）。将搜索供应商换为Brave后（表12），替代效应更强：本文配置取得64.09分、单任务成本仅\$0.1218，相对Fable 5（62.06，\$1.3241）提升2.03分、成本降低90.8%，相对Opus-4.8（59.11，\$1.6177）提升4.98分、成本降低92.5%，相对GPT-5.5（53.28，\$0.8407）提升10.81分、成本降低85.5%。在已接近饱和的短任务基准PinchBench上，相应的工作机制则是成本高效的质量持平：多模型配置以94.31分几乎持平最强单模型Opus-4.8（94.33，差距仅0.03分），成本从\$0.1649降至\$0.1349（-18.2%）；相对GPT-5.5（93.73，\$0.0963），该配置质量高0.58分，但成本更高。

Table 11 | DRACO（默认DuckDuckGo搜索）上单模型基线与选定多模型融合路由配置的对照。所有配置均运行于OPENSQUILLA harness。成本为单任务平均；token以千计。Fable 5完成94/100任务，其指标按已完成任务平均；其余配置均完成全部100任务。

方法	总分	成本(\$)	Tokens (K)
Fable 5	59.80	1.2122	93.7
Opus-4.8	52.36	0.6559	103.5
GPT-5.5	50.22	0.4505	81.9
GLM-5.2	48.28	0.1214	116.8
DeepSeek-V4 Pro	50.32	0.1320	83.4
Kimi K2.7 Code	45.48	0.0676	86.3
Qwen3.7-Max	49.34	0.0432	99.5
Gemini 3 Flash	40.79	0.0117	9.5
多模型融合路由（本文）	60.82	0.3766	579.7

Table 12 | DRACO（Brave搜索）上单模型基线与选定多模型融合路由配置的对照。所有配置均运行于OPENSQUILLA harness。成本为单任务平均；token以千计（可见token加推理token）。Fable 5因安全过滤拦截7个任务仅完成93/100，其各项指标按已完成的93个任务平均；其余配置均完成全部100任务。

方法	总分	成本(\$)	Tokens (K)
Fable 5	62.06	1.3241	106.7
Opus-4.8	59.11	1.6177	257.7
GPT-5.5	53.28	0.8407	189.4
多模型融合路由（本文）	64.09	0.1218	500.1

上述结果使用预先选定的多模型配置：proposer集合在每次运行前固定。进一步地，我们把agentic路由器与聚合阶段耦合：每次运行时由路由策略动态装配proposer集合，只固定GLM-5.2作为聚合器，在DRACO（默认DuckDuckGo搜索）上评测三个路由策略工作点：control（当前路由策略）、diversity-heavy（加重式(10)的互补性项，取更大的 α ）与quality-heavy（加重预测的单模型质量，取更轻的成本权重 λ ）。表13给出三组结果，可得两个观察。第一，diversity-heavy是最优路由策略（60.31 vs control的59.18），且成本略低——这是互补性正则项的直接证据：把路由器引向去相关的proposer集合，可以在不增加开销的情况下提升聚合质量。第二，quality-heavy以微小的质量差距（59.93）换来最低成本（\$0.2582），给出同一前沿上更便宜的工作点。最优的动态装配组（diversity-heavy，60.31，\$0.3172）接近表11的手工选定配置（60.82，\$0.3766），成本低15.8%且无需人工挑选proposer集合。

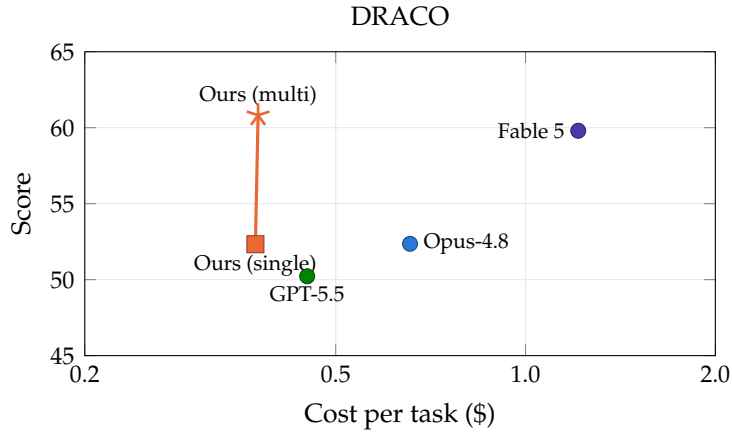


Figure 5 | DRACO 上单模型路由与多模型融合结果的成本-总分视图。彩色圆点为固定单模型基线，颜色对应模型系列（蓝色Opus、绿色GPT-5.5、紫色Fable 5）；橙色方块与星形（以线相连）分别为本文的单模型路由与多模型融合；成本轴为对数刻度。

Table 13 | DRACO 上的动态装配集成：路由器在运行时装配proposer 集合，GLM-5.2 固定为聚合器。所有配置均运行于OPENSQUILLA harness。成本为单任务平均；token 以千计。

路由策略	总分	成本(\$)	Tokens (K)
Control	59.18	0.3249	650.4
Diversity-heavy	60.31	0.3172	586.6
Quality-heavy	59.93	0.2582	721.3

图 5 把单模型与多模型融合结果汇总为DRACO 上的成本-总分视图（剔除OpenClaw harness 基线与比本文单模型路由更便宜的基线）：本文的两个工作点支配最强基线——多模型融合以远低于Opus-4.8 与Fable 5 的成本取得更高总分，单模型路由则以一小部分成本持平Opus-4.8 的质量点。PinchBench 上强基线已接近饱和，单模型路由是最便宜的工作点，集成配置则以更低成本几乎到达Opus-4.8 的质量点。综合来看，agentic routing 在两个基准上都实现了式 (3) 意义下的前沿移动： $k = 1$ 区间提供成本侧压缩， $k > 1$ 区间在深度研究类任务上同时推进质量与成本。

7. 相关工作

7.1. Agent系统

从宏观范式看，2023–2025 年的主线是通过Scaling Law、数据规模与训练算力扩展优化模型参数，形成以GPT [32]、Claude [2]、GLM [43]、Kimi [34]、DeepSeek [38] 等为代表的基础模型池 $\mathcal{M}$ 。Agent 时代的关键问题则从“如何训练更强的单个模型”转向“如何学习调度和约束模型池”，即如何把 $\mathcal{M}$ 组织为Agent = Base Models + Harness。本文的目标式 (2) 正是在这一转移下刻画Harness 价值：通过 g_t 选择模型，通过 s_t 组织prompt、RAG、工具、记忆与安全边界，从而提升Intelligence/Token。

大语言模型Agent 系统的核心思想在于：使模型不再仅生成文本，而是在任务执行过程中显式地维护中间状态、调用外部工具、观察环境反馈，并据此动态修正后续行动。ReAct [41] 是该方向具有代表性的早期范式之一，其将推理轨迹（reasoning trace）与行动（action）交替组织，使模型能够在问答、事实核验与交互式决策任务中借助外部环境获取信息，并在执行过

程中调整既定计划。相较于纯提示工程方法，ReAct 将“推理-行动-观察”（reasoning-acting-observation）循环抽象为一种可复用的交互模式，为后续工具调用型Agent 框架奠定了概念基础。

在工程实现层面，LangChain [5] 将模型、提示模板、工具接口、检索器、向量数据库与外部API 封装为可组合的标准化组件，显著降低了构建LLM 应用与Agent 原型的工程门槛。随着Agent 应用的复杂度由简单的链式调用演进至长程、可恢复且可控的工作流LangGraph [17] 进一步将Agent 执行过程建模为有向图（Directed Graph），强调持久化执行（durable execution）、人机协同（human-in-the-loop）、分支控制与长期状态管理。LlamaIndex [20] 则以数据连接与RAG为切入点，逐步扩展为面向文档处理、索引构建与 workflow编排的Agentic 应用框架，尤其适用于需将私有数据、信息检索与工具调用紧密耦合的场景。

多Agent 系统是另一条重要的研究分支。AutoGen [36] 将多个可对话Agent 组织为群聊式的协作会话结构，每个Agent 可结合LLM 推理、人类输入与工具执行，并通过自然语言对话进行协作，使复杂任务得以分解给具有不同角色、权限或工具集合的专门化Agent。CrewAI [7] 同样采用角色化协作思路，通过Crews 与Flows 抽象组织自治Agent 与事件驱动流程，更贴合企业级自动化与流程编排需求。此类系统的共同特征是将单次模型调用扩展为多轮、多角色、多工具的协同执行，但同时引入了状态一致性维护、错误传播抑制、权限控制与系统可观测性等一系列工程挑战。

近期的开源个人Agent 系统则更加强调本地化运行、跨渠道交互与真实工具执行。例如，OpenClaw [33] 将网关、会话、工具与多渠道消息入口整合为本地优先（local-first）的个人助理系统，支持文件操作、Shell 执行、浏览器自动化、定时任务以及多Agent 路由；Hermes Agent [25] 在类似形态上进一步突出长期记忆、技能动态生成、会话检索、定时任务与多后端部署；nanobot [29] 则尝试以更小、更具可读性的核心代码提供长时运行Agent、聊天渠道、记忆机制以及模型上下文协议（Model Context Protocol, MCP）等能力。综观上述工作可以发现，Agent 系统正由ReAct 式的“推理-行动”提示范式，逐步演化为涵盖模型路由、工具抽象、状态管理、Skill 编排、持久化记忆、安全沙箱与可观测性的完整Harness 基础设施。

7.2. 高效Agent技术

围绕LLM 与Agent 端到端推理效率的工作可按优化对象大致分为三类：上下文侧解决attention 的内存与计算随序列长度增长的二次膨胀；输入侧关心Prompt 中的冗余Token；模型侧则在请求与Token 两个粒度上以“小模型先做、大模型兜底”分摊调用成本。

上下文侧：**KV Cache** 压缩与稀疏注意力。这一线工作针对长上下文处理的两类瓶颈：KV cache 内存随序列长度线性增长但绝对量大，attention 计算则随序列长度呈 $O(N^2)$ 增长。共同目标是在不损失生成质量的前提下把上述两类开销实质降下来。StreamingLLM [37] 观察到长流式推理中最早的几个token 起到attention sink 的作用，仅保留sink 与滑动窗口的KV 即可稳定生成；H2O [44] 进一步动态保留attention 累积分数最高的Heavy Hitter；SnapKV [19] 把判定时机前移到生成之前，依据prompt 内部注意力模式选择性压缩KV cache；PyramidKV [4] 沿层维度做金字塔式分配，反映attention 的层间信息漏斗结构。除“丢哪些缓存”之外，KIVI [21] 从数值精度入手，给出KV cache 的非对称2 比特免微调量化；MInference [13] 则识别attention 矩阵中A 形、垂直-斜线与块稀疏三种模式，以稀疏注意力核加速长上下文prefill。

输入侧：**Prompt** 压缩与**Token** 数降低。上下文侧工作针对的是已有上下文的KV 存储与注意力计算，输入侧工作则更上一层、直接降低LLM 看到的Prompt 长度。LLMLingua [12] 利用小模型估算各token 的困惑度，对prompt 按句子与token 两级粒度做粗细压缩；LLMLingua-2 [28] 用数据蒸馏训练任务无关的prompt 压缩器，对各类下游任务都能做忠实的快速压缩；LongLLMLingua [14] 在此基础上针对长上下文场景引入关键信息密度估计与位置偏置补偿，缓解长文档中关键信息被“压缩稀释”的问题。在软压缩方向，Gisting [23] 训练LLM 把可重复使用的prompt 压缩为可缓存的gist token，使同一提示后续可从缓存直接重用；ICAE [10] 训

练上下文自编码器，将长上下文进一步压缩为短软提示。

模型侧：大小模型协同与动态路由。该方向的核心思路是不把每个请求都交给同一档模型，而是按粒度分摊到不同规模的模型上。在token级别，Speculative Decoding [18] 以draft小模型生成、大模型验证的方式加速单次解码，将原本N步串行生成压缩为接近一步的并行验证；Medusa [3] 在主模型上挂载多组并行解码头，省去独立的draft模型。在请求级别，FrugalGPT [6] 学习LLM级联策略，在调用成本、回答质量与早停判别之间联合选择模型组合；AutoMix [1] 用少样本自验证估计小模型答案的置信度，再由POMDP路由器决定是否上调至更大模型；RouteLLM [26] 则用偏好对监督训练路由器，在强弱模型间做二分类决策。

上述三类工作均聚焦在LLM单次推理的局部效率，分别对应目标式 (2) 中 $P(a)$ 或 s_t 的局部优化：KV cache 与prompt压缩降低输入侧token，动态路由降低模型调用单价，级联策略在质量与早停之间做权衡。当评测对象从LLM上升到Agent，HAL [15] 把cost与token用量纳入跨模型、跨scaffold、跨基准的标准化agent评测协议；SWE-Effi [9] 进一步将token与时间预算作为资源约束条件，重新评估agent系统在固定预算下的有效性。两者共同使端到端推理效率成为与解决率并列的一阶量化目标。OpenSquilla 的区别在于，它不把这些技术作为孤立优化，而是纳入可学习Harness，使 g_t 与 s_t 在跨子任务的端到端执行中协同发挥作用。

8. 结论与讨论

本文介绍了OPENSQUILLA，一个面向真实工具执行、多入口交互与长期协作的开源Agent-native learnable harness。与单纯追求“更强模型”的系统路线不同，OPENSQUILLA将Agent能力拆解为可路由的模型预算、可演化的Meta-Skill编排、可治理的长期记忆与可审计的工具边界四类系统资源，并通过微内核式执行路径把这些资源纳入统一turn生命周期。从形式角度看，这四类资源正是目标式 (2) 中 $g_t(\mathcal{M})$ 与 s_t 的工程分解：agentic routing 与运行时成本机制共同优化 $P(a)$ （多模型融合路由还能在深度研究任务上同时推进 $\mathcal{L}_T(a)$ 与 $P(a)$ ），Meta-Skill 与记忆提升 s_t 的可学习性，安全治理约束 $g_t \circ s_t$ 的可执行范围。实验结果表明，在PinchBench、ClawMark、ClawSWEbench 与DRACO等端到端任务中，OPENSQUILLA能在保持可竞争任务表现的同时改善成本曲线，尤其体现为本地路由、prompt cache continuity、技能按需注入与harness级可观测性共同形成的成本-能力Pareto优化。

上述结果中有三个成本层面的问题值得进一步讨论：多模型融合为何能以更多模型取得更低成本、跨模型路由如何与提示缓存连续性相互作用，以及路由器自身的开销。

多模型融合以更多模型取得更低成本，看似矛盾，实则源于成本的构成方式：成本等于token用量乘以单价，而旗舰模型的单价通常比中低档模型高出一到两个数量级。因此，由若干低价proposer与中档聚合器构成的组合，即便token用量数倍于单模型，总费用仍远低于一条旗舰模型轨迹：在DRACO上，选定融合配置的单任务token用量为579.7K（Fable 5仅93.7K），成本却只有\$0.3766，约为Fable 5（\$1.2122）的三成（表 11）。至于低价模型本身达不到的旗舰级质量，则依靠机制设计弥补：式 (10) 的互补性正则项促使所选proposer的失效模式相互去相关，以验证信号为条件的聚合器保证最终结果不劣于最优proposer。

当然，跨模型路由也引入了新的开销：provider侧的提示缓存按模型隔离，一旦切换模型，此前缓存的稳定前缀即告失效，须按全价重新计费。对此，系统从三个层面加以约束。首先，路由器优化的是轨迹级成本 $C(\tau)$ ，切换所需的前缀重填开销已计入其中，只有预期的价格收益超过缓存损失时才会发生切换，相当于一种抑制档位反复震荡的滞回机制；其次，路由决策与正在执行的任务对齐：同一会话的多轮对话中并非每轮都切换模型，而是在同一任务阶段内保持同一档位，前缀缓存因而持续命中；最后，稳定base / 动态tail的提示组织（subsection 3.3）将需要重付的前缀压缩到最小，缓存命中与失效事件均记录在日志中、可供核查。DRACO的结果印证了两相抵扣后收益为正：路由配置的token用量虽略多于固定配置，成本仍下降43%（表 10），说明模型单价结构上的收益足以覆盖缓存损失。进一步将缓存状态

显式纳入路由决策（即cache-aware routing），是router 模型后续迭代的一个自然方向。

最后是路由器自身的开销。路由器每一步都要运行，其开销必须明显小于其所带来的节省，方能自洽。开源的 $g^{(0)}$ 是本地LightGBM 流水线，毫秒级完成一次决策，不消耗任何LLM token，也不向远端服务发送数据，相对一次模型调用，其边际成本可以忽略。后续的学习代虽然更重，但其本身仍是小模型（例如4B 量级），相对被路由的大模型调用依然十分便宜，且路由决策的输出token 极少；部署上仍保持两层结构（subsection 2.4）：明显琐碎或明显困难的状态由低开销的准入过滤直接处理，学习路由器只在不确定、高价值或弱可恢复的状态上启用；同时，晋升门槛以计入路由器自身开销与延迟的实测前沿为准，从机制上保证路由器的开销不会反过来抵消其创造的节省。

总体而言，OPENSQUILLA的核心主张是：生产级Agent 的竞争力不应只由模型榜单决定，更应由单位成本下的任务完成率、可学习的Skill 编排能力、长期上下文治理能力和真实工具执行安全性共同决定。从Scaling Law 时代到Agent 时代，竞争焦点正在从训练更强的 M 转向设计更优的Harness；OpenSquilla 正是这一范式转移下，以可学习Harness 提升Intelligence/Token 的开源探索。随着用户持续使用，Harness 能够把偏好、流程、记忆与安全边界沉淀为可审查资产，在同类任务中越来越贴近用户工作方式。OPENSQUILLA以Apache 2.0 协议开源，代码托管于<https://github.com/opensquilla/opensquilla>，欢迎社区贡献与反馈。

附录

A. 贡献者

核心贡献者：韩凯、何为、周航、刘昕宸、焦庆锐、宗英杰、刘润科、张硕、田雨川、郑梦雨、舒梦、程思扬、宋柳扬、张鸿博、范佳宇、胡海林、王云鹤

贡献者：曹国良、郭天宇、匡翔、李宏广、李宇龙、李哲翔、刘曦、刘筱伊、刘怡杉、裴泽华、王科雅、吴益洪、谢兆凯

B. 术语对照表

本报告的多数核心术语源自英文文献，表 14 给出正文最常用核心术语的中英对应关系。

Table 14 | 中英术语对照表

中文术语	英文原文
智能体	agent
大语言模型	large language model (LLM)
词元	token
马具	harness
可学习马具	learnable harness
路由	routing
单模型路由	singleton routing
多模型融合路由	multi-model ensemble routing
提议模型	proposer
聚合器	aggregator
异策略学习	off-policy learning
提示注入	prompt injection
MMR 重排	maximal marginal relevance (MMR) re-ranking

References

[1] P. Aggarwal, A. Madaan, A. Anand, et al. AutoMix: Automatically mixing language models. *arXiv preprint arXiv:2310.12963*, 2024.

[2] Anthropic. System card: Claude opus 4.7. Anthropic Technical Report, April 2026. URL <https://www-cdn.anthropic.com/037f06850df7fbe871e206dad004c3db5fd50340/Claude%20opus%204.7%20System%20Card.pdf>. Accessed: 2026-06-24.

[3] T. Cai, Y. Li, Z. Geng, et al. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

[4] Z. Cai, Y. Zhang, B. Gao, et al. PyramidKV: Dynamic KV cache compression based on pyramidal information funneling. *arXiv preprint arXiv:2406.02069*, 2025.

[5] H. Chase et al. LangChain. <https://github.com/langchain-ai/langchain>, 2023. GitHub repository.

[6] L. Chen, M. Zaharia, and J. Zou. FrugalGPT: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176*, 2023. doi: 10.48550/arXiv.2305.05176. URL <https://arxiv.org/abs/2305.05176>.

- [7] CrewAI Inc. CrewAI. <https://github.com/crewAIInc/crewAI>, 2023. GitHub repository.
- [8] M. Dudik, J. Langford, and L. Li. Doubly robust policy evaluation and learning. *arXiv preprint arXiv:1103.4601*, 2011.
- [9] Z. Fan, K. Vasilevski, D. Lin, et al. SWE-Effi: Re-evaluating software AI agent system effectiveness under resource constraints. *arXiv preprint arXiv:2509.09853*, 2025.
- [10] T. Ge, J. Hu, L. Wang, et al. In-context autoencoder for context compression in a large language model. *arXiv preprint arXiv:2307.06945*, 2024.
- [11] Q. J. Hu, J. Bieker, X. Li, N. Jiang, B. Keigwin, G. Ranganath, K. Keutzer, and S. K. Upadhyay. RouterBench: A benchmark for multi-LLM routing system. *arXiv preprint arXiv:2403.12031*, 2024. URL <https://arxiv.org/abs/2403.12031>.
- [12] H. Jiang, Q. Wu, C.-Y. Lin, et al. LLMingua: Compressing prompts for accelerated inference of large language models. *arXiv preprint arXiv:2310.05736*, 2023.
- [13] H. Jiang, Y. Li, C. Zhang, et al. MInference 1.0: Accelerating pre-filling for long-context LLMs via dynamic sparse attention. *arXiv preprint arXiv:2407.02490*, 2024.
- [14] H. Jiang, Q. Wu, X. Luo, et al. LongLLMingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839*, 2024.
- [15] S. Kapoor, B. Stroebl, P. Kirgis, et al. Holistic agent leaderboard: The missing infrastructure for AI agent evaluation. *arXiv preprint arXiv:2510.11977*, 2025.
- [16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, 2017.
- [17] LangChain AI. LangGraph. <https://github.com/langchain-ai/langgraph>, 2024. GitHub repository.
- [18] Y. Leviathan, M. Kalman, and Y. Matias. Fast inference from transformers via speculative decoding. *arXiv preprint arXiv:2211.17192*, 2023.
- [19] Y. Li, Y. Huang, B. Yang, et al. SnapKV: LLM knows what you are looking for before generation. *arXiv preprint arXiv:2404.14469*, 2024.
- [20] J. Liu. LlamaIndex. https://github.com/jerryjliu/llama_index, 2022. GitHub repository.
- [21] Z. Liu, J. Yuan, H. Jin, et al. KIVI: A tuning-free asymmetric 2bit quantization for KV cache. *arXiv preprint arXiv:2402.02750*, 2024.
- [22] Y. Moslem and J. D. Kelleher. Dynamic model routing and cascading for efficient llm inference: A survey. *arXiv preprint arXiv:2603.04445*, 2026.
- [23] J. Mu, X. L. Li, and N. Goodman. Learning to compress prompts with gist tokens. *arXiv preprint arXiv:2304.08467*, 2023.
- [24] Nous Research. Hermes agent: Mixture of agents. <https://hermes-agent.nousresearch.com/docs/user-guide/features/mixture-of-agents>, 2026. Documentation.

- [25] NousResearch Team. Hermes agent. <https://github.com/NousResearch/hermes-agent>, 2026. Apache-2.0 License.
- [26] I. Ong, A. Almahairi, V. Wu, W.-L. Chiang, T. Wu, J. E. Gonzalez, M. W. Kadous, and I. Stoica. RouteLLM: Learning to route LLMs with preference data. *arXiv preprint arXiv:2406.18665*, 2024. doi: 10.48550/arXiv.2406.18665. URL <https://arxiv.org/abs/2406.18665>.
- [27] OpenAI. Learning to reason with llms, 2024. URL <https://openai.com/index/learning-to-reason-with-llms/>.
- [28] Z. Pan, Q. Wu, H. Jiang, et al. LLMingua-2: Data distillation for efficient and faithful task-agnostic prompt compression. *arXiv preprint arXiv:2403.12968*, 2024.
- [29] X. Ren. nanobot. <https://github.com/HKUDS/nanobot>, 2026. GitHub repository.
- [30] T. Schick, J. Dwivedi-Yu, R. Dessi, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language models can teach themselves to use tools. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL <https://openreview.net/forum?id=Yacmpz84TH>.
- [31] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023. URL https://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.
- [32] A. Singh, A. Fry, A. Perelman, A. Tart, A. Ganesh, A. El-Kishky, A. McLaughlin, A. Low, A. Ostrow, A. Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.
- [33] P. Steinberger. OpenClaw: A self-hosted multi-channel personal AI assistant gateway. <https://github.com/openclaw/openclaw>, 2025. Documentation: <https://docs.openclaw.ai/>.
- [34] K. Team, T. Bai, Y. Bai, Y. Bao, S. Cai, Y. Cao, Y. Charles, H. Che, C. Chen, G. Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.
- [35] G. Wang, Y. Xie, Y. Jiang, A. Mandlekar, C. Xiao, Y. Zhu, L. Fan, and A. Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.
- [36] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.
- [37] G. Xiao, Y. Tian, B. Chen, et al. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2024.
- [38] A. Xu, B. Lin, B. Xue, B. Wang, B. Xu, B. Wu, B. Zhang, C. Lin, C. Dong, C. Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.
- [39] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652, 2024. URL https://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

- [40] P. Yang, W. Chen, T. Yang, P. Feng, J. Xing, W. Guo, Y. Yao, Y. Han, H. Li, and X. Wang. Twinrouterbench: Fast static and live dynamic evaluation for realistic agentic llm routing. *arXiv preprint arXiv:2605.18859*, 2026.
- [41] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.
- [42] Z.ai. GLM-5.1: Open-weight agentic coding model. <https://huggingface.co/zai-org/GLM-5.1>, 2026.
- [43] A. Zeng, X. Lv, Z. Hou, Z. Du, Q. Zheng, B. Chen, D. Yin, C. Ge, C. Huang, C. Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.
- [44] Z. Zhang, Y. Sheng, T. Zhou, et al. H₂O: Heavy-hitter oracle for efficient generative inference of large language models. *arXiv preprint arXiv:2306.14048*, 2023.
- [45] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. P. Xing, H. Zhang, J. E. Gonzalez, and I. Stoica. Judging LLM-as-a-judge with MT-Bench and chatbot arena. In *Advances in Neural Information Processing Systems*, volume 36, 2023.
- [46] M. Zheng, K. Han, B. Li, H. Xu, Y. Tian, W. He, H. Zhou, J. Guo, H. Hu, L. Ma, C. Xu, G. Dai, L. Xia, Y. Wei, Y. Wang, and W. Yu. Claw-swe-bench: A benchmark for evaluating openclaw-style agent harnesses on coding tasks. *arXiv preprint arXiv:2606.12344*, 2026.
- [47] J. Zhong, H. Zhang, C. Southern, J. Yang, T. Wang, K. Jung, S. Zhang, D. Yarats, J. Ho, and J. Ma. Draco: A cross-domain benchmark for deep research accuracy, completeness, and objectivity. *arXiv preprint arXiv:2602.11685*, 2026.
- [48] P. Zhou, Z. Tang, Y. Ma, J. Tang, Y. Han, Z. Wan, F. Meng, W. Wang, B. Zhuang, W. Zhao, and Y. Yang. Agent-as-a-router: Agentic model routing for coding tasks. *arXiv preprint arXiv:2606.22902*, 2026.