OpenSquilla: Token-Efficient Agent = Models + Routing Harness

Token Rhythm Technology Co., Ltd.

2026-08-12

ChinaRxiv

Abstract

OpenSquilla is an open-source, agent-native, learnable harness whose starting point can be summarized as Token-Efficient Agent = Models + Routing Harness: agent capabilities come from scheduling and allocating a pool of heterogeneous models rather than binding to a single powerful model. On end-to-end agent benchmarks, its local step-level routing reduces cost by 88.9% while maintaining 99.96% of the task quality of a fixed flagship model; multi-model fusion routing achieves higher quality on deep research tasks at 31% of the cost of the strongest single-model baseline (Fable 5). These capabilities are built on a microkernel-style design: the multi-model pool, tools, memory, skills, channels, scheduling, and sandbox policies can all be integrated in a pluggable manner, thereby separating stable execution mechanisms from evolvable policies. OpenSquilla has three core competitive advantages: 1. Single-model routing reduces cost: it formalizes model selection as a step-level routing operator conditioned on harness state. In cost-saving mode, it dispatches the cheapest model with sufficient capability for each execution step, while coordinating with runtime mechanisms such as inference budgets, prompt caching, and on-demand skill injection, reducing end-to-end cost by multiples with almost no loss in task quality. 2. Multi-model fusion routing improves effectiveness: in high-accuracy mode, it assembles complementary multi-model combinations and fuses their candidate outputs, achieving higher task effectiveness at a cost below that of a strong single model and advancing both the quality and cost frontier on deep research tasks. 3. System mechanisms for a learnable harness: the Meta-Skill orchestration layer enables the agent to learn to retrieve, filter, combine, and evolve skills, and to distill users' historical collaboration into auditable new workflows; the persistent memory system maintains editable, retrievable, forgettable, and auditable long-term context for cross-session collaboration; and safety governance defines boundaries for real tool execution through tiered policies, runtime isolation, and audit loops. These three components respectively provide capability orchestration, cross-session state, and execution boundaries, enabling the system to become better with use while remaining auditable and controllable. Comparative analysis shows that the value of OpenSquilla lies not in stacking a single large model, but in improving the agent capability obtainable per unit cost through agentic routing (single-model routing for cost savings and multi-model fusion routing for effectiveness gains), Meta-Skill orchestration, long-term memory, and controlled tool boundaries.

Full Text

[Token Rhythm logo] [OpenSquilla logo]

OpenSquilla: Token-Efficient Agent = Models + Routing Harness

Token Rhythm Technology Co., Ltd.

https://github.com/opensquilla/opensquilla

(a) PinchBench (single-model routing)

(b) DRACO (multi-model fusion)

Figure 1 | Core results of OpenSquilla. (a) PinchBench: the gray bars are an OpenClaw fixed flagship Opus-4.7 direct run, and the orange bars are the OpenSquilla multi-level routing configuration; quality is maintained at 99.96% (92.51 vs. 92.55), while cost is reduced by 88.9% ($0.69 vs. $6.23). (b) DRACO: the gray bars are the strongest single model, Fable 5, and the orange bars are the multi-model fusion configuration in this paper; quality exceeds the baseline by 1.0 point (60.82 vs. 59.80), and per-task cost decreases by 68.9% ($0.3766 vs. $1.2122). In each subfigure, the left axis is task score and the right axis is cost.

Abstract

OpenSquilla is an open-source Agent-native learnable Harness. Its starting point can be summarized as **Token-Efficient Agent = Models + Routing Harness**: Agent capability comes from scheduling and allocating a pool of heterogeneous models, rather than being bound to a single strong model. On the end-to-end Agent benchmark, its local progressive routing reduces cost by 88.9% while preserving 99.96% of the task quality of a fixed flagship model; on deep-research tasks, multi-model fusion routing achieves higher quality than the strongest single-model baseline (Fable 5) at 31% of the cost. These capabilities are built on a microkernel-style design: the multi-model pool, tools, memory, skills, channels, scheduling, and sandbox strategies are all pluggable, thereby separating a stable execution mechanism from evolvable strategies. OpenSquilla has three core competitive advantages. 1. Single-model routing saves cost: model selection is formalized as a progressive routing operator conditioned on the harness state. In money-saving mode, each execution step is dispatched to a model that is capable enough and cheapest, and is coordinated with runtime mechanisms such as inference budgets, prompt caching, and skill injection on demand, reducing end-to-end cost by multiples with almost no loss in task quality. 2. Multi-model fusion routing improves effectiveness: in high-accuracy mode, complementary multi-model combinations are assembled and their candidate outputs are fused, achieving better task performance than a strong single model at lower cost, while advancing the quality–cost frontier on deep-research tasks. 3. System mechanisms for a learnable Harness: the Meta-Skill orchestration layer enables the Agent to learn to retrieve, filter, combine, and evolve skills, and condenses user-history collaboration into auditable new workflows;

the persistent-memory system maintains editable, searchable, forgettable, and auditable long-term context for cross-session collaboration; and safety governance defines the boundaries of real tool execution through hierarchical policies, runtime isolation, and audit loops. Together, these respectively provide capability orchestration, cross-session state, and execution boundaries, allowing the system to remain auditable and controllable while "getting better the more it is used." Comparative analysis shows that the value of OpenSquilla does not lie in

stacking a single large model; rather, it lies in using agentic routing (single-model routing to reduce cost, multi-model fusion routing to improve effectiveness), Meta-Skill orchestration, long-term memory, and controlled tool boundaries to increase the Agent capabilities obtainable per unit cost.

1. Introduction

With the rapid improvement of large language model (LLM) capabilities [27], LLM-based agents have shown enormous potential in code generation, knowledge question answering, task automation, and other domains. To move Agents from experimental prototypes toward large-scale production deployment, academia and the open-source community have produced a series of Agent frameworks and runtime systems, such as LangChain [5], OpenClaw [33], and Hermes Agent [25]. However, when building production-grade Agent applications oriented toward high-frequency interaction, long-period collaboration, and real tool execution, existing frameworks generally face three core challenges:

Cost out of control. Complex tasks usually require multiple rounds of dynamic planning and long-context interaction, causing Token consumption to grow in a stepwise manner with task complexity. If a system is bound for a long time to a single strong model, even low-difficulty requests such as simple confirmation, format conversion, and short-text retrieval must pay the price of a flagship model; if the system only relies on static rules or keyword-based diversion, it is easy to mistakenly sink complex tasks to weak models, resulting in failed retries. The lack of a localized, auditable, and reversible model-routing mechanism is a key reason why Agent costs are difficult to adaptively scale with task difficulty.

The performance ceiling of a single model. As models become increasingly specialized—each having strengths in coding, long-context recovery, reasoning, tool invocation, and low-latency response—even with sufficient budget, a single strong model is not optimal across different execution states such as evidence retrieval, analytical synthesis, and code modification; task performance is capped by the capability of the strongest single model. Simply replacing the model with a stronger one cannot exploit the complementary strengths of a model pool: existing frameworks generally lack a systematic mechanism that dispatches multiple complementary models at key states and verifies and fuses their candidate outputs.

Harness is difficult to learn and govern. Real user tasks span multiple stages, including retrieval, analysis, tool invocation, clarification, auditing, and final synthesis. However, traditional Skills only encapsulate single-item capabilities and rely on models to plan on the fly, making it difficult to stably reuse successful paths or to distill new workflows from historical collaboration. The memory mechanisms of most frameworks are limited to conversation-context stacking or full-vector databases, making it difficult to distinguish long-term facts, short-term tasks, audit evidence, and outdated noise; they may also carry historical noise and latent prompt injection into future contexts for a long time. At the same time, when Agents perform file reads/writes, Shell commands, and network requests, risks have expanded from improper textual answers to real changes in the host environment. Relying only on model self-restraint or one-time approvals cannot cover compound attack paths such as tool-return injection, memory contamination, and continuous probing; fully relying on heavy containers also increases deployment complexity. In other words, the Harness must be learnable, auditable, and governable simultaneously across three dimensions: capability orchestration, cross-session memory, and runtime security boundaries.

If we view this problem in the broader evolution of AI systems, the dominant paradigm from 2023 to 2025 has been to optimize model parameters around datasets, pretraining compute, and the Scaling Law, thereby forming a set of foundation-model pools with strong capabilities but different cost structures, $\mathcal{M} = \{M_1, \dots, M_n\}$, such as GPT [32], Claude [2], GLM [43], Kimi [34], and DeepSeek [38]. After entering the Agent era, system value is no longer determined solely by the ranking capability of a single model, but by whether multiple foundation models, tools, memories, skills, and security boundaries can be organized into a stable task-execution system. In other words, the basic form of an Agent can be abstracted as

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

Under the paradigm described by Eq. (1), if an Agent is to truly assist humans or even replace humans in completing part of their work, its value function should simultaneously consider task value, success rate, and Token cost-performance. This paper expresses it as a Harness optimization problem that improves intelligence per Token (Intelligence/Token):

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

where $\mathcal{L}_T(a)$ and $P(a)$ respectively denote the normalized loss and normalized execution price of the Agent on task T. Their product is a rough cost-performance characterization—the lower the loss and the lower the price, the better the objective. g_t is the model gate on the t-th subtask, responsible for

is responsible for selecting an appropriate model from the model pool $\mathcal{M}$; s_t is the Harness operation on the selected model, including prompt organization, retrieval-augmented generation (RAG), Skill injection, memory recall, tool strategies, and safety constraints. Here, $\circ$ denotes the combination of the model choice $g_t(\mathcal{M})$ and the Harness operation s_t, rather than numerical multiplication.

Under this framework, the three types of challenges described above correspond respectively to key failure modes in the objective: cost runaway means that g_t always selects the strongest model and $P(a)$ cannot scale with task difficulty; the ceiling effect of a single model means that g_t dispatches only one model at each step, and $\mathcal{L}_T(a)$ is bounded by the capability of the strongest single model; the difficulty of learning and governing the Harness means that s_t remains confined to static prompts or isolated Skills, lacks cross-session state, and the executable boundary of $g_t \circ s_t$ is unclear.

To address the above pain points, this paper designs and implements **OpenSquilla**—an Agent-native learnable Harness. The Harness here is not merely a wrapper for model calls, but a system-level scaffolding that carries task input, model routing, Skill orchestration, tool execution, state persistence, permission approval, and evaluation observability: on each subtask t, g_t selects a model, while s_t adjusts the prompt, RAG, skills, memory, and tool boundaries. Its core design follows the modern operating-system principle of "mechanism and policy separation" : the core engine retains only turn lifecycle, state-machine progression, event streams, tool boundaries, and finalization semantics; external capabilities such as multi-model pools, toolsets, memory stores, message channels, safety sandboxes, Meta-Skills, and schedulers are connected through standardized registries, adapters, or services. This architecture both improves extensibility and provides clear boundaries for fault isolation, capability evolution, permission control, and end-to-end observability.

From the perspective of technical features, OpenSquilla' s main contributions include the following three aspects:

- **Single-model routing reduces cost:** model selection is formalized as a step-level routing operator conditioned on harness state, uniformly characterizing the quality–cost Pareto frontier of Agent execution. Single-model routing ($k = 1$) matches capabilities by dispatching, for each execution step, the model that is sufficiently capable and cheapest, greatly reducing cost under the premise of nearly no loss in quality. The open-source cold-start implementation SquillaRouter combines four classes of text-model types, reasoning budgets, and prompt strategies, completes hybrid feature classification and routing locally without requiring additional remote classification calls; meanwhile, through Prompt Cache continuity, stable/dynamic prompt segmentation, and on-demand Skill loading, it reduces token overhead caused by repeated prefixes and irrelevant Skill descriptions.
- **Multi-model fusion routing improves performance:** in the high-

accuracy regime on the same frontier ($k > 1$), the router is based on a set of proposers that are mutually complementary in task-image assembly, and a state-aware aggregator fuses candidate outputs; complementary regularization terms guide the collection's failure modes to decorrelate, enabling combinations of cheaper models to achieve higher task performance at a cost lower than that of a strong single model, and automatically degenerating into single-model routing in routine states.

- **System mechanisms for a learnable Harness—Meta-Skill orchestration, persistent memory, and safety governance:** at the orchestration layer, a protocolized orchestration mechanism is introduced over community and local multi-source Skills, telling the Agent how to retrieve, filter, combine, and even evolve Skills, organizing complex tasks into an inspectable task graph of composition, routing, checkpoint, user clarification, and final synthesis. Meta-Skill Creator further generates auditable proposals from users' historical collaboration trajectories, allowing the system to gradually align with users during use. At the memory layer, a long-term context lifecycle is designed that integrates Markdown source files, local full text and semantic hybrid recall, MMR (maximal marginal relevance) re-ranking, optional temporal decay, Session Flush, and Memory Dream, emphasizing editability, reconstructability, auditability, and rollback, rather than simple accumulation of conversation logs. At the safety layer, Standard / Strict / Locked three-tier strategies are constructed, combined with permission approval, denial ledgers, expired-output cache cleanup, path and network constraints, and platform capabilities such as Bubblewrap (Linux) / Seatbelt profile generation (macOS), establishing runtime isolation boundaries for tool invocation without imposing a dependency on Docker. These three respectively provide capability orchestration, cross-session state, and execution boundaries for s_t, jointly supporting the learnability and controllability of the Harness.

Table 1 compares **OpenSquilla** with two representative open-source Agent frameworks of the same category along multiple dimensions. The emphasis of this table is not to claim absolute superiority on all dimensions, but to show how OpenSquilla builds a differentiated system design around the Harness goals of being "Agent-native, learnable, low-cost, and governable."

2. Agentic Routing: Harness-Native Model Routing

The traditional Agent instantiates the decomposition in Equation (1) as follows: a base model $\bar{m}$ is fixed offline, and every step of the harness calls the same model; all optimization occurs on the harness side—prompt, tools, context management, and recovery strategies all revolve around

Table 1 | System-orientation comparison of OpenSquilla and similar Agent Harnesses

Dimension	OpenSquilla	OpenClaw	Hermes Agent
Architectural form	Agent-native learnable microkernel Harness: separates the stable execution mechanism from pluggable, evolvable strategies, and continuously sediments workflows through use	Runs with local-first personal assistant orientation, integrating channels, tools, and conversational capabilities	A personal Agent system centered on long-term memory, skill generation, and multi-backend deployment
Cost control	Intelligent model routing, inference budgets / prompt-word strategy stratification, Prompt Cache continuity maintenance, and on-demand skill loading	Mainly relies on model configuration and tool flows; routing capability is not a core claim	Provides backend and execution-strategy selection, but lacks an open local ML routing closed loop
Learnable orchestration	Meta-Skill protocol layer, which sediments historical collaboration trajectories and multi-Skills into auditable workflow capabilities	Mainly relies on fixed tool/plugin capabilities; workflow evolution is not the core abstraction	Emphasizes skill generation, but orchestration governance and the harness boundary depend on its main-loop design
Memory system	Hierarchical memory, hybrid semantic retrieval, optional time-table attenuation, and Memory Dream consolidation	Local memory and conversational state combining files/databases	Emphasizes long-term memory and conversational retrieval; concrete implementation depends on its runtime design

Dimension	OpenSquilla	OpenClaw	Hermes Agent
Security governance	Three-tier security policy, denial ledger, cache invalidation, and platform sandbox collaboration	Reduces risk by means of tool permissions, runtime environment, and external isolation	Constrains high-risk operations through approval, execution environments, and task policies
Observability	Through diagnostic logs, read-only replay, and session export, records routing, cache, compression, tools, and cost events	Relies on conversation logs and replay of tool traces	Relies on conversation retrieval, skill traces, and runtime-log replay

This fixed model call parameter. In the language of this chapter, this type of Agent has no routing operator: model selection is a constant; the actual cost structure is determined by $\bar{m}$, and the only balancing means between quality and cost is to switch the entire Agent to another model. Once multiple base models appear in the model pool $\mathcal{M}$, this constant becomes a stepwise decision variable. Agentic routing replaces the fixed $\bar{m}$ with a state-conditioned routing operator g: at each execution step, it decides which model, or which group of model actions, should be used next, thereby turning the quality–cost tradeoff into an explicit optimization objective. A traditional Agent degenerates into the constant route $g \equiv \{\bar{m}\}$. Figure 2 summarizes the two deployment mechanisms of this operator: in economy mode, the router dispatches a single best-matched model (subsection 2.2); in high-accuracy mode, it dispatches multiple complementary models and aggregates their outputs (subsection 2.3). This chapter formalizes the operator and its objective, serving as the theoretical basis for the g_t side of objective (2); the runtime cost mechanisms coordinated with routing (derivation of inference budgets and prompt strategies, prompt caching, and skill filtering) are discussed in Section 3.

2.1. Problem Definition and Optimization Objective

Agentic routing is step-level model (or model-set) selection conditioned on the harness state during Agent execution. Let q be the user task, $\mathcal{M}$ the candidate model pool, and h_t the harness state at the t-th decision step. In this report, a step refers to one model-call decision; a turn refers to a user-visible interaction round and consists of one or more steps; the subtask subscript t in objective (2) is taken by step, and the iterative control signals in Section 3 correspond

Figure 2. Overview of the two working mechanisms of Agentic routing. The Agent harness provides execution state to the router, which selects from the shared model pool; the user-selected deployment mode determines the working mechanism: in cost-effective mode, the router dispatches a single best-matching model; in high-accuracy mode, it dispatches multiple complementary proposers and fuses their outputs with an aggregator. Verification and cost signals score each decision, and the generated records flow back to improve the router.

Figure 1: Figure 2. Overview of the two working mechanisms of Agentic routing. The Agent harness provides execution state to the router, which selects from the shared model pool; the user-selected deployment mode determines the working mechanism: in cost-effective mode, the router dispatches a single best-matching model; in high-accuracy mode, it dispatches multiple complementary proposers and fuses their outputs with an aggregator. Verification and cost signals score each decision, and the generated records flow back to improve the router.

to views of these step-level decisions at the turn granularity. h_t includes not only current observations, but also compressed and original context, control-loop state, available actions, artifact state, tool history, recovery state, and verification signals.

We do not define routing as a single cost-minimization problem. Existing model routing and cascade systems show that heterogeneous model pools can improve the cost–quality tradeoff of LLM services, but they usually operate at the static prompt or query layer [6, 11, 22, 26]; recent agentic routing benchmarks and routers instead push this decision toward execution trajectories [40, 48]. Unlike static shunting that relies on hard-coded rules and LLM routing that only performs semantically aware model selection, this paper adopts a harness-native perspective. We take the formalization one step further: the router can maintain quality at lower cost, combine multiple cheaper specialized models to surpass a strong single-model baseline, and invest more budget in high-risk states to improve task success rate. Therefore, we define a unified quality–cost frontier for harness-native Agent execution.

Figure 2 | Overview of the two working mechanisms of Agentic routing. The Agent harness provides the execution state to the router, which selects from the shared model pool; the deployment mode selected by the user determines the working mechanism: in cost-effective mode, the router dispatches a single best-matching model, while in high-accuracy mode it dispatches multiple complementary proposers and fuses their outputs through an aggregator. Verification and cost signals score each decision, and the generated records flow back to improve the router.

We model routing as an operator g: conditioned on the current harness state h_t, it selects a model set $S_t = g(\mathcal{M} \mid h_t)$ from the model pool $\mathcal{M}$ for the next execution step, and, when multiple models are selected, simultaneously provides

an ensemble strategy. The router optimizes the Pareto frontier between task loss and cost:

$$\min_{g} \left(\mathbb{E}_{q\sim\mathcal{D},\,\tau\sim g}[\ell(\tau)],\ \mathbb{E}_{q\sim\mathcal{D},\,\tau\sim g}[C(\tau)] \right), \qquad S_t = g(\mathcal{M} \mid h_t), \qquad 1 \le |S_t| = k \le K, \tag{3}$$

where the objective is minimized in the Pareto sense. $\ell(\tau) = 1 - R_{\text{task}}(\tau)$ is the task loss of the final trajectory, and $C(\tau) = \sum_{t=1}^{T} \sum_{m\in S_t} c(m, h_t)$ is the actually incurred cost. $k = |S_t|$ is the number of models appearing simultaneously at step t, constrained by the upper bound K. Taking $k = 1$ yields single-model routing (subsection 2.2), in which g dispatches exactly one model at each step; taking $k > 1$ yields multi-model routing (subsection 2.3), in which g dispatches multiple models together with an ensemble strategy. Equivalently, scanning the scalarized objective $\mathbb{E}_{\tau\sim g}[\ell(\tau) + \lambda C(\tau)]$ over $\lambda \ge 0$ can trace the same frontier; when deployment requires it, latency can enter the objective as an optional third axis. Equation (3) is a special case of objective (2) in the routing scenario of this paper: with the harness operation s_t fixed, only the model-selection dimension varies; g is therefore instantiated step by step as g_t, while $\ell(\tau)$ and $C(\tau)$ respectively play the roles of $\mathcal{L}_T(a)$ and $\mathbf{P}(a)$.

Interpreting Eq. (3): a router g is better if and only if it reduces task loss, reduces cost, or achieves both. The key point is that a multi-model action ($k > 1$) is not necessarily more expensive than a single-model action, because S_t may combine several inexpensive or specialized models whose total cost is lower than one call to an expensive flagship model. Different operating points on the frontier answer different questions: the low-cost point asks how much money the harness minimally needs to preserve task quality, whereas the high-accuracy point invests more budget in difficult, high-risk, or weakly recoverable states. This paper instantiates this frontier along the dimensions of the models and model sets used by routing actions: the action outputs the model set S_t, and, when $k > 1$, attaches an aggregation strategy. To isolate the model-allocation component, this paper fixes the surrounding harness strategy and evaluates only how far model and model-set selection can advance the frontier, so that the reported gains can be attributed to capability allocation on a fixed execution substrate.

The loss is defined at the task level rather than the single-step level. Agent execution is recoverable: a locally weak step can be corrected later, whereas a locally cheap step may create downstream recovery costs. This perspective is consistent with treating reasoning, actions, tool use, memory, and feedback as intertwined execution processes in Agent work [30, 31, 35, 39, 41]. Therefore, the supervision signal should not assume that every step is ···

there are independent correctness labels; the natural reward is final or derived from verification, so g must learn from trajectories rather than from an isolated prompt. The trajectory field

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

defines the schema of harness-native routing-record data: each record stores the routing action separately from the label given afterward by the environment —the action is the model set S_t dispatched at the time, while the labels are the final result y, the single-step output z_t, the cost c_t, and the verification signal. This separation means that the trajectory is not an object to be imitated: a clumsy routing decision, together with its negative outcome, is likewise an effective training sample. In this report, the open-sourced LightGBM router is the cold-start strategy that initiates this data flow; subsequent router-model iterations (subsection 2.4) use accumulated routing-trajectory data to improve future routing decisions.

2.2. Single-Model Routing

We first instantiate the frontier under the single-model setting. Taking $k = 1$ in Eq. (3), the routing operation is restricted to a single model:

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

The router still optimizes the same task-level frontier, but its online action becomes: dispatching one model, selected from a heterogeneous model pool, for the next harness step.

We call this perspective **capability matching**: the harness issues capability requirements conditioned on the state; the model pool supplies capabilities that differ in price, latency, context length, tool-call reliability, and failure modes; the single-model router selects the least expensive model whose capability is sufficient to cover the current state after risk adjustment. The model pool is therefore a collection of specialized models, and under uncertainty the router matches the most suitable model to the needs of each state. What the router does is not merely classify prompts, but allocate model capability along the harness execution trajectory: a model that appears inexpensive will become costly if its failure triggers retries, tool repair, context reconstruction, or downstream verification recovery.

Let the single-model routing state be

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

where q is the original user task and h_t is the current harness state. h_t contains not only the visible prompt, but also context pressure, artifact state, tool history, recovery state, recent routing decisions, and verification signals. The required capability is therefore not a fixed attribute of the user query, but a trajectory-dependent requirement induced by the current execution state.

Operationally, single-model routing is the greedy application of Eq. (3) at $k = 1$ to the current step. To make this step-level decision easier to estimate and optimize than a purely final signal, we add a one-round reward $\rho_t = \rho(h_t, S_t, z_t) \in [0, 1]$ to score the immediate output z_t of the routing action: it may be a verifiable reward (unit tests pass, tool call succeeds, schema or safety constraints are satisfied), or it may be an LLM-as-a-judge [45] score for that step' s output. Fixing the weight λ in Eq. (3), the router dispatches the single model with the minimum expected remaining-trajectory task loss and cost, offset by the immediate reward:

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

Here $\mathcal{M}_t \subseteq \mathcal{M}$ is the current candidate model set after admission filtering and deployment-availability constraints; the weight $\beta \geq 0$ trades off, in the immediate single-round reward, against trajectory-level loss $\ell(\tau)$ and cost $C(\tau)$; when $\beta = 0$, it reverts to the purely final objective of Eq. (3). Keeping ℓ and C at the trajectory level preserves the semantics of recoverable execution—a model that is superficially cheap but subsequently triggers retries, tool repair, context reconstruction, or verification recovery will still raise the downstream cost $C(\tau)$, while a model whose capability is insufficient to cover the current state will raise the task loss $\ell(\tau)$. The single-round reward merely densifies the supervisory signal, providing step-level feedback to the router, alleviating credit assignment and accelerating optimization. The intuition of capability matching—using the heterogeneous capability supply in the pool to match the capability requirements of the state condition, while accounting for the residual risk of irreversible or weakly recoverable actions—is an interpretation of how $\ell(\tau)$, $C(\tau)$, and ρ_t respond to model choice, rather than another independent optimization problem. The key point is unchanged: a model whose failures create expensive recovery is not cheap, and a model whose capabilities do not match the state requirements is not strong.

The image of required capabilities is multidimensional. Routine control, formatting of short tool calls, low-risk context bookkeeping, or cacheable response generation may require only lightweight models; vague planning, difficult code modification, long-context debugging, and recovery after verification failure may require ···

Complex, safety-sensitive decisions or final product generation may require stronger models. The same user task may therefore require different models at different steps: context pressure makes long-context reliability more important than the original reasoning score; recent tool errors make action-format robustness more important than leaderboard performance; verifier failure raises the capability needed for the next recovery step. This is precisely the difference between harness-native single-model routing and ordinary query-level routing: model selection depends not only on what the user asked, but also on where the harness currently is in its execution trajectory.

The single-model router open-sourced in OpenSquilla realizes this matching perspective through a lightweight cold-start mechanism. It executes a four-stage pipeline: admission filtering, which sends clearly trivial and low-risk states to an inexpensive fast path; demand construction, which builds a coarse-grained capability profile from the task and harness state; risk pricing, which adjusts that demand using context pressure, tool failure, verifier results, recovery state, action irreversibility, and downstream error-correction risk; and capability matching, which binds the adjusted demand to a specific model or tier through the deployment registry. This decomposition deliberately differs from a simple model-ranking pipeline: it separates "what the next harness step needs" from "which model in the current deployment can satisfy it." In the open-source implementation, the first three stages are implemented by inexpensive deterministic filters and lightweight classifiers: demand construction scores mixed features such as prompt length, language, code form, keywords and semantic embeddings, and identifies task signals such as code blocks, attachments, long context, and high-risk prompts; the final matching stage is completed over the remaining candidates by a LightGBM ranker [16], in which multimodal requests such as images are imposed as a hard constraint on an independent capability type—only models with the corresponding input capability are bound, and they are not mixed with text tiers. The entire pipeline executes locally at millisecond scale; routing itself consumes no LLM tokens and does not send user requests to a third-party classification service. LightGBM is useful but not fundamental: it is a cold-start matching engine rather than the matching problem itself—the latter is jointly defined by the harness state, model pool, deployment registry, verifiers, and runtime accounting. The harder learning problem is to infer, from harness-native trajectories, the capability needs under state conditions, the risk of under-routing, and the expected recovery cost that equation (7) only implicitly depicts through $\ell(\tau)$ and $C(\tau)$.

This positioning is important for the open-source boundary. The LightGBM router is the first practical point on the single-model frontier: it is efficient at cold start, routes by step quickly, is auditable and therefore open-sourceable, and is simple enough to improve from log execution. It should not be read as the final router model, but as an open seed that starts the data stream for routing trajectories. The router model should be viewed as an iterative matching policy: each batch of routing-trajectory data updates the same group of latent objects that define single-model routing—demand construction, risk adjustment, and capability matching. States and selected models are recorded as actions; final outcomes, verifier events, recovery costs, actual costs, and latency are supplied by the environment. These records allow subsequent generations of router models to learn when an open-source strategy under-routes, when it pays unnecessarily for excessive capability, and when changes in the model pool or harness cause the deployment profile of a certain model to drift. Under the single-model setting, router-model iteration is evaluated in money-saving mode: each generation must assign a model to the current step while maintaining task quality and reducing unnecessary calls to expensive models. The learning

objective is improvement of outcomes rather than imitation of the previous-generation router, consistent with the logged-feedback and off-policy-learning perspective in contextual bandit evaluation [8]: a rough LightGBM decision is useful feedback rather than toxic supervision, because labels are provided by environmental signals such as verification, recovery, actual cost, and latency. In this sense, single-model routing is not model selection in the narrow serving-layer sense, but model-capability allocation toward outcomes under uncertainty in the harness state.

2.3. Multi-Model Fusion Routing

Multi-model fusion routing is an instantiation of equation (3) under $k > 1$: the router g dispatches a set of models rather than a single model. Given the harness state h_t and a candidate pool $\mathcal{M}_t \subseteq \mathcal{M}$ that has likewise been prefiltered, the routing operation returns a proposal set and a combination strategy:

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

where the set of proposer models P_t generates candidate outputs, and the aggregation strategy a_t merges the candidates into a single executable result; $\mathcal{G}$ includes voting, verifier-based selection, judge models, and fallback strategies. When the aggregator itself is a model call, the selected set in equation (3) is

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

where m_t^{agg} is the aggregation model selected as part of a_t; for rule-based, voting, or verifier aggregators, $S_t = P_t$. At this point, the cost $C(\tau)$ in equation (3) counts each proposal call and aggregation overhead, while latency is tracked separately as an overhead indicator.

As in the single-model case, the same frontier is instantiated step by step. Let $\ell(u \mid h_t) = \mathbb{E}_{\tau\sim g}[\ell(\tau) \mid u_t = u, h_t]$, $C(u \mid h_t) = \mathbb{E}_{\tau\sim g}[C(\tau) \mid u_t = u, h_t]$, and $\rho(u \mid h_t) = \mathbb{E}_{\tau\sim g}[\rho_t \mid u_t = u, h_t]$ denote, respectively, the expected task loss, cost, and one-step reward of routing action u under state h_t. For the multi-model action $u_t = (P_t, a_t)$, the router minimizes the scalarized objective in Eq. (3), and augments it with a complementarity regularizer and one-step reward:

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

The first two terms are the scalarized frontier objective of the current-step greedy application of Eq. (3) when $k > 1$; the third term $-\alpha V(P \mid h_t)$ is a complementarity regularizer with weight α; the fourth term $-\beta\rho((P, a) \mid h_t)$ is the extension of the one-step reward in Eq. (7) to an ensemble action, where a

verifier or LLM judge acts on the aggregated output. The single-model rule in Eq. (7) is the special case $u_t = (\{m\}, \emptyset)$, in which case V disappears.

Cost C is cumulatively charged for proposed calls and aggregation overhead, so a larger set is penalized according to how much it actually costs. The complementarity term $V(P \mid h_t)$ rewards proposal sets that are related to failure modes. We implement $V(P \mid h_t)$ with a marginal diminishing-return proxy based on historical error decorrelation, state-conditional branching, and verifier feedback, supporting efficient greedy subset construction. In principle, the task loss ℓ already prefers such sets—proposers that reduce relevant failures reduce expected loss, whereas redundant models that share the training distribution and failure modes raise cost without reducing loss. However, because ℓ is trajectory-level, off-policy, and noisy, while the choice of P is combinatorial, the explicit $-\alpha V$ regularizer makes subset search easier to optimize and generalize, guiding exploration toward complementary sets rather than collapsing to the existing preference of a logging policy. It is an inductive bias over the agent objective and does not change the frontier of Eq. (3): α is taken to be small and is annealed to zero as the loss estimate becomes accurate, so that improvements in the estimator recover the true cost–quality frontier; the router is never rewarded merely for diversity that does not reduce loss. This complementarity term is precisely what distinguishes multi-model routing from top-k selection. The scale constraint $|P| \leq K$ inherits from Eq. (3); when required by deployment, a latency budget enters as an optional third axis.

The two operating mechanisms are two interpretations of Eq. (10) under different cost weights λ, both compared against a strong single-model reference action $u_t^{\text{ref}} = (\{m_t^{\text{ref}}\}, \emptyset)$. When λ is small, the minimizer tends toward accuracy-seeking ensembles: in states where single-model errors are costly and hard to recover—final answer generation, complex code editing, irreversible tool actions, safety-sensitive decisions, long-context synthesis, or low router confidence—it spends additional calls to push loss below the reference ($\ell(u_t \mid h_t) < \ell(u_t^{\text{ref}} \mid h_t)$), paying higher cost within the single-step budget B_t. A typical division of labor is: one model gives the main proposal, another searches for errors from a different perspective, and a third provides inexpensive plausibility checks. When λ is large, the same minimizer tends toward cost-saving ensembles: a combination of medium- and low-cost models replaces an expensive reference, keeping loss within tolerance ε ($\ell(u_t \mid h_t) \leq \ell(u_t^{\text{ref}} \mid h_t) + \varepsilon$) while strictly spending less ($C(u_t \mid h_t) < C(u_t^{\text{ref}} \mid h_t)$); when the loss is also strictly below the reference, the same combination is simultaneously accuracy-improving. Both conditions are derived from Eq. (10), not added as new objectives. The economics of multi-model routing is to seek a better cost–quality operating point among a strong single model, a cheap single model, and combinations of cheaper models.

We implement multi-model routing as a two-stage process. The first stage is task-image-driven subset selection: the routing model first computes a task image φ_t from (q, h_t), summarizing signals such as task type, difficulty, required capabilities, risk level, verifiability, cost and latency tolerance, and routing uncertainty.

This image is a compact representation of the state-conditioned quantities in the above objective, not a new environment state. Given φ_t and the candidate pool $\mathcal{M}_t$, the selector picks a proposal subset P_t whose capabilities match the image and whose members can provide complementary evidence. Additional proposers are scored by their fit to the task image, their own cost and latency image, expected marginal value, verifier compatibility, uncertainty reduction, and complementarity with the already selected models. A valuable proposer is not merely strong in isolation, but fills the capability or perspective required by the task image, such as long-context evidence recovery, code synthesis, adversarial critique, or low-cost error checking. Selection is therefore image-conditioned subset construction rather than top-k ranking, and preferences differ by mechanism: accuracy-seeking ensembles prefer reliability gains in key states, while cost-saving ensembles prefer quality retention at lower total cost. The second stage is state-aware aggregation: each proposer $m \in P_t$ independently generates a candidate r_m, and the aggregator produces the final result

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

The strategy a_t is likewise conditioned on the task image and on the strongest verification signals observable by the harness: executable signals in software tasks such as unit tests, static analysis, or patched application; schema, permission, and safety constraints in tool tasks; and rubric scoring, consistency, factuality, and citation checks in open research tasks. a_t is not a post-processing module but part of the routing action: the router simultaneously decides who proposes and how to fuse. Under the cost-saving mechanism, the aggregator itself must be cost-sensitive—an expensive aggregator above cheap proposers would consume the savings on the proposal side—so a_t is a lightweight judge, rule-based or verifier selector, or a low-cost model.

model aggregator; the refinement mechanism can then use a stronger aggregator when the improvement in task risk, verification accuracy, and quality is sufficient to cover its cost.

Each integration step is also a data-production operation: the candidates r_m from each proposer provide comparison results from multiple models at the same decision point (q, h_t), supplying factual counterfactual-contribution data needed for learning heterogeneous strategies. The two-stage design preserves the cost–quality trade-off of a single model: the system degenerates to single-model routing in routine states, selectively introduces complementary proposers in states where the failure cost of a single model is high, and in cost-sensitive states replaces a strong single model with a cheaper combination of models—reducing cost to a given quality level while expanding the quality frontier.

2.4. Router Model Iteration: Implementation and Evolution

The router is not deployed as a single model, but as a sequence of strategies $g^{(0)}, g^{(1)}, g^{(2)}, \ldots$ that share the objective in (3) yet differ in representation and training data. The zeroth generation $g^{(0)}$ is the open-source LightGBM ranker of subsection 2.2: a shallow cost-quality ranker with manual features, wrapped externally by four inexpensive pipeline stages—admission filtering, requirement construction, risk adjustment, and capability matching [16]. It is not intended to be the final router, but rather the cheapest policy that can run at every step and initiate the routing-trajectory data stream. Every subsequent generation $g_\theta^{(r)}$ is a learned model trained on this data stream. This section describes how such a generation of policy is constructed, trained, and rolled online.

Architecture. A router model iteration is decomposed into three components. The state encoder maps the harness state h_t—the task and instructions, compressed and original context, artifact and tool history, verifier outputs, recovery state, and recent routing history—to a state vector, replacing the manual features of $g^{(0)}$. The model encoder maps each candidate model $m \in \mathcal{M}_t$ to an image embedding, constructed from its model card and observed outcome history: price, latency, context length, reliability of tool calls, and success rate by task family. Because candidates are described by images rather than fixed output classes, newly released models can be scored directly from their image without retraining the label space. The prediction head estimates, for each candidate action $u = (S, a)$ under state h_t, the expected task loss $\hat{\ell}(u \mid h_t)$ and expected cost $\hat{C}(u \mid h_t)$. The router then acts strictly according to the rules in (7) and (10), selecting $\arg\min_u[\hat{\ell}(u \mid h_t) + \lambda \hat{C}(u \mid h_t)]$.

Training. Each generation minimizes, on the accumulated routing-trajectory corpus $\mathcal{D}^{(r)}$, the frontier objective in (3):

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

This expectation is an off-policy estimate: because each record is produced by an earlier policy $g^{(r)}$ rather than by uniform sampling over actions, the estimator reweights logged outcomes with inverse-propensity or doubly robust correction, so that the update targets outcome improvement rather than imitation of the logging policy [8]. No additional reward, recovery, or risk terms are needed: recovery costs and downstream mismatches are as described in subsection 2.2 and subsection 2.3; they enter the objective through trajectory-level $C(\tau)$ and $\ell(\tau)$, while λ selects the operating point on the frontier.

Coverage. Coverage is treated as a first-class citizen in the routing-data design. Logs are naturally densest for selected actions, so routing trajectories introduce controlled exploration—uncertainty-triggered escalation, and occasional alternative-model or oracle replay on sampled state subsets—to improve coun-

terfactual coverage, enabling $g^{(r+1)}$ to learn in which states $g^{(r)}$ mismatched and in which states it paid extra for capability.

Evolution. The policy sequence forms a ladder: a generation is promoted only when it pushes the frontier in (3) farther than the previous generation. The open-source seed $g^{(0)}$ is a LightGBM ranker over manual features; the first learned generation $g^{(1)}$ replaces manual features with a trained state encoder and prediction head, is fit offline on the initial corpus, and remains gated behind the inexpensive admission filter of $g^{(0)}$; the next generation $g^{(2)}$ adds model-encoder images for generalization to new models and counterfactual data supplied by exploration, and can be distilled into a small fast router to satisfy step-level latency budgets; later generations $g^{(r)}$ are continuously updated as the corpus grows and the model pool changes. A candidate generation is released only when it reduces actual cost under equal task reward, or improves task reward under a fixed budget; otherwise deployment rolls back to the previous generation. Consistency with $g^{(r-1)}$ has never been the objective—the movement of the quality–cost frontier is.

Deployment. Because the router runs at every step, its own cost and latency are also part of the trade-off. Deployment maintains two layers: the cheap admission filter and $g^{(0)}$-style gates handle clearly trivial or clearly difficult states, while the heavier learned router is invoked only on states that are uncertain, high-value, weakly recoverable, or worth paying for better matching decisions. This keeps the router always on the favorable side of its own cost–quality trade-off. This phased path also defines the open-source boundary: the $g^{(0)}$ seed can be inspected, trained locally, and is useful for cold starts, so it can be open-sourced; subsequent generations rely on accumulated routing-trajectory data and on changing ···

···models supply images and continuous frontier evaluations; therefore this is an evolving OpenSquilla strategic path rather than a stable product. Convergence of the router sequence is not a prerequisite for deployment: each promotion uses the measured frontier movement as its benchmark, while the routed trajectory data stream continuously provides the data needed for improvement as the model pool and harness evolve.

3. Intelligent Cost Control

3.1. Two-Layer Cost-Control Framework

The actual load of LLM agents exhibits two types of structural heterogeneity: the high variance of single-round difficulty and the high redundancy of cross-round content. A short confirmation and a cross-file code revision may differ in difficulty by more than an order of magnitude; a single-tier strategy—whether always using the strongest model or always using the weakest—cannot simultaneously account for quality and sustainable cost [6, 26]. Meanwhile, within the same conversation, the system prompt, tool definitions, and stable context are highly repetitive at the token level, and naive retransmission makes "cost $\propto$

number of rounds" hold. Therefore, OpenSquilla decomposes cost control into two complementary problems: single-round cost must be tiered by difficulty, and cross-round repetition must be identified and amortized. Among them, the routing problem of "selecting the model tier according to difficulty" has already been fully delineated by the agentic routing framework in Section 2—SquillaRouter, i.e., the cold-start strategy $g^{(0)}$ that initiates the routed trajectory data stream. Locally, it completes a four-stage pipeline of access filtering, requirement construction, risk adjustment, and capability matching, and outputs the routing tier, confidence, and decision-element information; this chapter does not revisit the routing mechanism itself. From the perspective of objective (2), the focus of this chapter is: under the premise that $g_t(M)$ has already been given by the router, how to further reduce $P(a)$: the decision layer derives inference budgets and prompt strategies from the routing-element information, while the runtime layer reduces repeated tokens and irrelevant context through operations s_t such as cache, prompt, and Skill injection; at the same time, the system retains the single-model direct-connection mode for reproducing experiments and isolating "model capability" from "framework-scheduling contribution" during evaluation.

Figure 3 presents the two-layer architecture corresponding to this idea. In each round, the decision layer gives discrete control signals (m_t, r_t, p_t): the text-model type $m_t \in \{c0, c1, c2, c3\}$ (four capability tiers from lightweight to flagship; multimodal capabilities such as images are treated as independent types) is selected by the single-model router from Section 2, while the inference budget r_t and prompt-word strategy p_t are derived from the same routing-element information in §3.2. The runtime layer interfaces with the decision layer and consists of Prompt Cache continuity and skill filtering, respectively handling cross-round stable-prefix reuse and injectable element-information clipping. The two layers converge at the upstream LLM call: the decision layer reduces "the marginal cost to be spent in the current round," while the runtime layer removes the "already spent" and "unnecessary" portions from the next round' s context.

Figure 3 | Text in the diagram

- **User input** (current message, recent history, previous-round output and usage, task feature f_t)
  - **Decision layer** (**Section 2**, **§3.2**)
    Agentic Routing local routing kernel $g^{(0)}$
    Four text-model types: $\{c0, c1, c2, c3\}$
    Outputs route class, confidence, and metadata
    Derives inference budget and prompt-word strategy
    Output: (m_t, r_t, p_t)
  - **Runtime layer** (**§3.3–§3.4**)
    Skill Filter: $\mathcal{T} \to \mathcal{F}_t$, selected as needed
    Explicit naming, trigger description, and dependency gating
    Prompt Cache: stable base + dynamic tail
    cache continuity affected by provider / tool / attachment
    Output: $\mathcal{F}_t$, (BASE, DYNAMIC), K_t

- **Upstream LLM call**: m_t, r_t, prompt p_t, inject $\mathcal{F}_t$, cache key

Figure 3 | OpenSquilla two-layer cost-control framework. The decision layer gives the text-model type, inference budget, and prompt-word strategy in each round; the runtime layer concurrently performs skill filtering and Prompt Cache continuity maintenance, outputting an injectable skill set $\mathcal{F}_t$ as well as a prompt structure in the form of stable base / dynamic tail. The two layers converge at the upstream LLM call, jointly reducing the single-round marginal cost and cross-round repetition cost.

Table 2 compares the four types of mechanisms involved in the two-layer framework horizontally by their scope of action. This table is an index for Section 2 and §3.2–§3.5; when readers encounter a specific mechanism in different subsections, they can refer back to this table to confirm its position within the overall two-layer framework.

Table 2 | Horizontal comparison of four cost mechanisms in the two-layer framework. The "failure fallback" column describes the fallback behavior when each mechanism is unavailable.

Mechanism	Layer of action	Time scale	Main observable signals	Failure fallback
Model-tier routing by m_t (Section 2)	Decision layer / single turn	Local millisecond level	routed tier, confidence, metadata	Default middle tier (c1)
Derivation of reasoning budget and prompt policy (r_t, p_t)	Decision layer / single turn	Local execution	reasoning tier, policy, metadata	Conservative default budget and policy
Prompt caching	Runtime layer / cross-turn	Amortized over the entire session	cache token, cache invalidation, base/tail changes	Price according to cache miss
Skill filtering	Runtime layer / system level	Single turn, scaled with $\|T\|$	Hit skills, eligibility, inspect / meta runs	Explicit skill names or no skill injection

3.2. Derivation of Reasoning Budget and Prompt Policy

The tier m_t is selected locally by the single-model router described in Section 2, and only answers "which tier of model to call." OpenSquilla also derives

a reasoning budget and response policy from the same routing metadata, in order to answer "whether deeper reasoning is needed" and "how much prompt is appropriate." This allows the system to continue adjusting cost within the same model tier: for example, low-difficulty requests may use a lighter reasoning budget and shorter prompts; debugging, long context, strict formatting, or high-risk operations use more conservative response policies, and delegate execution risk to permissions and the sandbox layer for further adjudication.

Let ψ_r denote the reasoning-derivation function and ψ_p the policy-derivation function. Then

$$r_t = \psi_r(m_t, \pi_t, f_t, H_t), \qquad p_t = \psi_p(m_t, \pi_t, f_t, H_t). \tag{13}$$

Here, π_t is the tier distribution output by the router, f_t denotes task features parsed from keywords, code blocks, attachments, and high-risk prompts, and H_t is a finite history window. The two derivation functions are not directly exposed as user-visible "chain-of-thought text," but are instead mapped to reasoning budgets, response lengths, format strictness, or prompt variants acceptable to the model interface. In other words, what OpenSquilla controls is the reasoning budget and response policy, rather than writing the model' s internal reasoning process verbatim into the context.

The value of this design is that it extends cost control from "one-dimensional model selection" to "three-dimensional budget allocation" : tier determines the model unit price, reasoning determines reasoning tokens and latency, and policy determines dynamic prompt length and formatting constraints. Together, the three constitute a controllable surface for end-to-end Agent cost.

3.3. Cross-Turn Mechanism: Prompt Caching

In addition to single-turn tier selection, Agent cost is also affected by repeated prefixes across turns. Prompt caching exploits the reuse of stable-prefix KV-cache on the provider side; the responsibility of the harness is to keep that prefix stable. OpenSquilla organizes prompts into a stable base and a dynamic tail: the base mainly contains the system identity, stable constraints, and tool definitions; the tail carries the current request, runtime context, recall history, attachment summaries, and tool results. Let γ_t be the reusable proportion of the stable prefix in turn t, and let δ be the provider' s discount factor for cache-read tokens. Then the input-side cost can be written as

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

This expression does not depend on the pricing details of any specific provider; it only states that, when the stable prefix remains stable and the provider supports cache billing, repeated system prompts and tool definitions need not be paid for at full price in every turn.

In engineering practice, cache continuity is best-effort rather than absolutely guaranteed. Changes in model tier, provider switching, tool-definition changes, new attachments entering the context, long-context compression, or changes in the dynamic skill set may all reduce the reuse rate. Therefore, the system records cache-hit and failure events in diagnostic logs, and exposes the relationship between cached tokens and total tokens in a session-level cost view, so that caching gains can be reviewed rather than remaining only a configuration-layer claim.

3.4. Cross-Turn Mechanism: Skill Filtering

The skill system is another link by which OpenSquilla reduces prompt inflation. The system presets and allows the expansion of multi-source skills, covering common scenarios such as coding, version collaboration, document generation, summarization, and external information acquisition. If every turn injected descriptions of all skills into the system prompt, the richer the skill library became, the longer the stable prefix would be and the higher the cost. Therefore, OpenSquilla adopts a strategy of "loading relevant guidance on demand" : the model does not see the full set of skills by default; instead, at runtime, candidates are selected according to the task intent, explicit skill names, trigger descriptions, and the installation environment.

Formally, let the full set be $\mathcal{T}$, and let the set injected in the current turn be $\mathcal{F}_t$. Then

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

where msg_t is the current user message, and ctx_t includes the runtime environment, available dependencies, tool permissions, workspace boundaries, and the user' s explicit intent. The filtering process contains two types of constraints: first, availability gating—for example, skills whose dependencies are missing or that are only for demonstration are unavailable; second, relevance matching, which matches the user' s natural-language objective with skill descriptions, trigger words, and the Meta-Skill structure. The system also provides the ability to inspect skill structures, enabling users and developers to understand the step boundaries of a given Meta-Skill.

Skill filtering and the prompt strategy p_t form a nested relationship: p_t determines the form of the prompt (compressed / neutral / sufficient), while skill filtering determines the scale of what can be injected into the content pool. In objective (2), this kind of on-demand injection is precisely the core component of s_t: it does not change the selected model itself, but affects the trade-off between task success rate and prompt cost by selecting which capability descriptions the current model should see. The advantage of decoupling the two is that growth in the size of the skill library mainly affects the filtering side, rather than forcing every turn to bear the token cost of the full set of skill descriptions. At runtime, the system further exposes key decisions through skill visibility, structure

inspection, execution traces, and the proposal-management interface, turning skill injection from "implicit prompt magic" into inspectable system behavior.

3.5. Failure Modes and Operations Observability

Soft failure semantics When predictive core-loading fails, routing asset validation fails, or a one-time classification throws an error, the router returns the default mid-tier model and attaches a diagnosable cause, rather than silently choosing the cheapest tier. This strategy keeps unknown difficulty within a mid-tier safety interval, reducing failure retries and quality collapse caused by misrouting.

Runtime-layer degradation A stable prompt prefix may change during a conversation (for example, when new tools, attachments, or skills enter the context), which lowers cache continuity; vendor or model switching can also reduce cache benefits. On the skill-system side, when dependencies are missing or availability gates are not passed, relevant skills do not enter the injection set; users can still confirm their availability through the skill-visibility and structure-inspection interfaces. When the memory or semantic-search backend is unavailable, the system preserves keyword / full-text retrieval paths, avoiding interruption of the main turn due to the absence of optional components.

Diagnosis and replay OpenSquilla provides unified diagnosis, conversation export, and read-only replay capabilities. Diagnostic logs record model communication, SquillaRouter decisions, prompt cache hits and misses, context compression, tool-output compression, channel delivery, and cost / latency anomalies; read-only replay does not re-execute tools, but instead generates a readable trajectory from decision logs. Thus, cost control is not only an online strategy, but also has offline review and attribution-assessment capability.

At this point, OpenSquilla' s intelligent cost control can be summarized as a two-layer system in which the decision layer and runtime layer cooperate orthogonally: in each turn, the decision layer decides "which tier of model to call / how much reasoning budget to use / what response strategy to adopt," where the model tier is given by the agentic routing of Section 2; the runtime layer removes "stable prefixes already spent" and "skill descriptions that need not be injected" from the next-turn context. All key decisions enter diagnostic, cost-statistics, and read-only-replay views; when subcomponents are unavailable, the system degrades to interpretable paths such as the default mid-tier model, single-model direct connection, or keyword retrieval. Thus, cost control becomes the engineering implementation of the $P(a)$ side of objective (2): g_t reduces the unit price of models through agentic routing, while s_t reduces reasoning and repeated token costs through reasoning-budget derivation, cache continuity, and on-demand skill injection, jointly promoting improvement in Intelligence/Token. Subsequent experiments will show the cost–capability trade-off formed by this design on end-to-end Agent benchmarks.

4. System Mechanisms of a Learnable Harness: Meta-Skill, Persistent Memory, and Safety Governance

After Section 2 and Section 3 answer "which model to call and how much to spend," this chapter introduces the three system mechanisms that support a learnable Harness, corresponding respectively to the three dimensions of s_t in objective (2): the Meta-Skill orchestration layer determines how capabilities are organized and evolved; the persistent-memory system provides a cross-session state substrate; and the safety-governance framework delineates the executable boundary for $g_t \circ s_t$. Together, the three ensure that while the Harness "gets better with use," it remains auditable, rollbackable, and controllable.

4.1. Meta-Skill: Skill Orchestration and System Evolution

In traditional Agent architectures, Skills encapsulate single capabilities such as summarization, retrieval, analysis, writing, and tool invocation. As bundled, workspace, and community-sourced Skills continue to expand, the problem the system must solve is no longer "whether there is a capability," but "how to let the Agent retrieve, filter, combine, and evolve these capabilities" : for complex tasks spanning stages, requiring conditional judgments and multiple rounds of interaction, it is difficult to achieve stable reuse by relying only on ad hoc planning by a single Skill or model. For this reason, OpenSquilla defines Meta-Skill as a declarative orchestration mechanism positioned above Skills—an abstraction of workflows that performs composition, routing, dependency scheduling, failure substitution, user clarification, and final aggregation across multiple Skills. Composition decomposes complex tasks into a semantically explicit task graph with dependencies and data flow; for example, report generation is organized into requirement understanding, structural planning, chapter generation, and consistency summarization, with different Skills responsible for each stage. Routing makes execution branches explicit: entry routing selects a Meta-Skill from the user request; in-process routing selects subsequent branches based on intermediate results; capability routing chooses, among candidate Skills, the capability suitable for the current context; and recovery routing continues from a suspension point after the user supplements information. Its key principle is that the model is responsible for understanding intent, while runtime is responsible for constrained execution—the overall task structure, step order, branch paths, and output strategies are controlled by declarative workflows, thereby embedding the flexibility of large models into a verifiable, reusable, and observable engineering framework. Expressed in the language of objective (2), Skills extend the capability set of s_t, and Meta-Skill makes the mode of combining s_t perceptible to the task type.

The Meta-Skill Creator further solves "how the system learns new organizational patterns from historical collaboration." The capability growth of traditional skill systems relies on manual development and is difficult to cover the long-tail tasks of real scenarios; yet in practice, many complex tasks repeatedly form similar multi-Skill collaboration patterns. If these patterns cannot be sedimented, the

model must re-plan every time. The Creator observes, from historical execution traces, the co-occurrence, ordering, and dependency relationships among Skills. Its evolution process is divided into four stages: observation, recording user goals, multi-Skill invocation chains, intermediate results, branch choices, and output quality; abstraction, extracting from repeated traces stable composition, routing, input requirements, and output strategies; validation, checking the structural completeness of candidate workflows, trigger boundaries, routing stability, output quality, and risk; and sedimentation, saving validated candidates as Meta-Skill proposals, which enter the skill system after review or conservative activation. It must be emphasized that what the Creator generates is a proposal rather than a capability that goes directly online: once a workflow is solidified, it will influence how the system handles similar tasks in the future; without governance, accidental paths, overly broad trigger conditions, or high-risk operations may be sedimented as long-term behavior. Therefore, candidates must pass structural checks, positive and negative sample validation, risk assessment, and necessary human review. Thus, Skills provide atomic capabilities, Meta-Skills provide orchestration capabilities, and the Creator provides the generation and evolution capabilities for orchestration patterns; together, the three transform s_t from fixed prompt/RAG operations into a learnable orchestration strategy that adapts to user scenarios, which is precisely the core meaning of a "learnable Harness."

4.2. Persistent Memory System

When an Agent shifts from single-turn question answering to long-period collaboration, user preferences, project facts, historical decisions, and reuse patterns need to remain valid across sessions; however, complete dialogue logs, tool returns, and temporary instructions cannot indiscriminately enter future contexts, otherwise this will cause noise accumulation, residual outdated facts, and persistent prompt-injection risks. OpenSquilla therefore designs memory as an independent long-term state layer rather than "vector store + full conversation log," and follows two boundaries: memory is separated from skills—Skills describe how the Agent completes tasks, while Memory describes what the Agent should continue to know in the future; long-term memory is separated from audit records—reusable facts are organized as curated memory, while complete conversations and raw turn records are retained as audit or derivative retrieval material and do not directly contaminate ordinary recall paths. In objective (2), the memory system is the cross-session state substrate of s_t: it enables Harness operations to perceive historical facts and successful paths, rather than reorganizing the task from a blank context in every round.

In terms of system architecture, persistent memory is abstracted as a four-layer lifecycle (Table 3). The source layer defines which materials can become memory: editable Markdown long-term-memory files, virtual retrieval documents derived from persisted sessions, and private turn capture that by default does not enter ordinary indexing. The indexing layer uses a local relational database as its

base, hashing memory content for deduplication and splitting it into line-based ...

fragments in the number range, and builds a full-text index and an optional vector index; changes to the embedding backend, model, or dimensions trigger safe index rebuilding, avoiding erroneous mixing of vectors from different embedding spaces. Before each retrieval, the recall layer performs pre-query synchronization to ensure index consistency, and then conducts hybrid recall and reranking. The consolidation-governance layer manages memory evolution through Session Flush, Memory Dream, and cleanup strategies. Semantically, a local-first strategy is adopted: by default, quantized embedding inference is completed on the local machine, long-term memory is not sent to third-party services by default, and when the local backend is unavailable the system degrades to pure full-text retrieval.

Table 3 | Division of responsibilities among the four layers of the OpenSquilla persistent memory system

Layer	System role	Main responsibilities	Degradation semantics
Source layer	Editable memories, session fragments, private transcripts	Distinguish long-term facts, historical session evidence, and audit raw records, avoiding direct contamination of ordinary recall by raw streams.	Preserve files or session records, but do not enter ordinary retrieval.
Index layer	Local full-text and semantic indexes	Split memory sources into retrievable fragments, build a full-text index and an optional vector index, and maintain consistency between the embedding cache and indexes.	When vectors are unavailable, degrade to full-text retrieval.

Layer	System role	Main responsibilities	Degradation semantics
Recall layer	Hybrid retrieval and reranking	Synchronize indexes before querying, fuse semantic recall and lexical recall, and improve stability with temporal decay, MMR, and lexical floor protection.	Preserve keyword recall capability.
Consolidation-governance layer	Flush, Dream, and cleanup strategies	Distill long conversations into candidate memories, raise long-term facts through evidence gatekeeping, and perform TTL, desensitization, rollback, and audit.	Use dry-run / archive-only or skip writing.

The objective of the recall layer is to return the most relevant and as non-redundant as possible memory fragments within a limited prompt budget, using a hybrid strategy of vector semantic recall and lexical recall:

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

The vector channel captures semantically similar historical facts, while the lexical channel ensures precise hits on proper nouns, paths, dates, error codes, and project terminology; when the vector backend is unavailable, set $w_v = 0$, $w_l = 1$, automatically degrading to pure full-text retrieval. On top of this, there are three stabilization mechanisms: lexical floor protection allows strongly matched full-text fragments to enter the final candidates, avoiding semantic scores suppressing explicit keyword hits; date-type memories undergo exponential time decay with a 30-day half-life, while root-level `MEMORY.md` and non-date-type long-term memories are regarded as evergreen and do not participate in decay, so that temporary tasks naturally sink while stable facts remain visible; MMR reranking (relevance–diversity weight $\lambda_{\text{MMR}} = 0.7$, using inter-fragment Jaccard similarity to measure redundancy) constrains result diversity and avoids near-duplicate fragments occupying the recall budget.

The sedimentation side consists of two levels of mechanisms. Session Flush, before session reset, archiving, or context compression, distills long conversations into candidate memories such as facts, preferences, decisions, procedures, and to-do items, and writes them into contextual storage through controlled writing; when LLM extraction times out or the result cannot be safely parsed, it degrades to archiving only the raw material—the evidence can be retained for audit, but low-confidence summaries are not promoted to long-term facts. Memory Dream is periodically triggered offline consolidation: each run scans only incremental candidates, skips private directories and high-risk texts, and raises items according to a weighted evidence-score gate based on occurrence frequency, positive/negative signal balance, source credibility, and cross-date consolidation. It does not permit the model to rewrite the entire memory base, but instead generates constrained incremental modifications (insert / merge / skip). Before modifying the root-level index, it creates a backup, and each run writes replayable receipts. On the write side, writing a memory is treated as a long-term context-governance operation: the target must lie within a controlled memory-source path; before writing, desensitization, injection scanning, and capacity checks are performed; keys, typical prompt injection, and invisible control characters are blocked; snapshot and rollback semantics are adopted; TTL and capacity cleanup do not touch `MEMORY.md`, bootstrap files, or private directories, and a single cleanup operation sets no deletion upper bound. Compared with a simple "full history + vector recall" design, this design emphasizes source boundaries, index rebuildability, recall interpretability, and consolidation auditability, allowing s_t to accumulate and reuse experience across the scale of session time without being hijacked for long periods by historical noise, expired information, or malicious content.

4.3. Security Governance

The security risks of tool-type Agents are not limited to "non-compliant answers" in model text: an apparently ordinary tool call may delete workspace files, overwrite configurations, leak keys, access network resources, or write malicious instructions into long-term memory and candidate workflows and then trigger them again in subsequent sessions. Drawing on the attack surfaces examined in existing Agent security evaluations, this paper broadly groups attacks into six categories (the classification and abbreviations are defined in this paper): direct and indirect prompt injection (DPI / IPI), tool-return

injection (TRI), memory contamination and memory extraction (MPI / MEX), and adversarial delegation induction (ADI)—the first three types are biased input sources, the middle two involve persistent state, and the last originates from the adversarial boundary between user intent and execution privileges. This partitioning means that Agent safety cannot rely only on a prompt-level policy; instead, it must introduce least privilege, system-level isolation, audit closure, and state cleanup at runtime. The security design of OpenSquilla follows four principles: least privilege by default, where each tool call obtains only

the file, network, and process capabilities needed to complete the current task; separation of policy and execution, where the model may propose an action plan, but whether to execute it is decided at runtime according to risk prompts, path scope, and approval state; auditability of refusals, where all refused actions are cumulatively entered into a denial ledger by operation fingerprint; and non-reuse of state, where tool outputs that are refused or expired cannot be cited by subsequent steps. From objective (2), security governance is a domain restriction on executable (g_t, s_t) combinations: the stronger s_t is, the more precise its runtime authorization boundary must be.

At runtime, a security tier is selected dynamically according to the source of the operation, scope of effect, and potential impact. It determines five dimensions: trusted-source coverage, network requirements, writes outside the workspace, crossing of trust boundaries, and high-impact operations. Table 4 presents the three-tier policy and its runtime actions. This tiering converts "whether to trust the model' s judgment" into "what system capabilities to permit at runtime" : instructions from external webpages, emails, or tool return values, even if semantically reasonable, do not automatically obtain file-write or networking privileges; actions involving keys, configuration, deletion, overwriting, and cross-directory writes are raised to STRICT or LOCKED, and are jointly constrained by the sandbox and approval. In other words, after routing, a $g_t \circ s_t$ combination enters the real execution path only when permission, safety, and budget predicates are all satisfied.

Table 4 | Three-tier security policy and runtime actions

Level	Trigger condition	Runtime action
STANDARD	Trusted source, reads/writes within the working directory, no network, or low-impact operation	Execute directly, and record the tool summary and output fingerprint
STRICT	Untrusted input, reading across working directories, writing to the host path, tool return values carrying instructions	Enable sandbox isolation; restrict mounting and output; trigger user approval when necessary
LOCKED	Crossing trust boundaries, network access, deleting/overwriting key files, credential-related or irreversible operations	Enforce human review; use the strictest sandbox configuration; after consecutive refusals, suspend autonomous execution

System-level isolation adopts a default-deny policy on systems that support namespaces or platform sandboxes (Bubblewrap-based on Linux, Seatbelt profile generation on macOS): only the working directory and the host system directories necessary at runtime are exposed into the sandbox in read-only mode; network access and process capabilities are disabled by default; tool output is constrained by size and timeout; and path mounting undergoes normalized validation to prevent cross-directory access and path traversal. Because isolation capabilities differ across platforms, the sandbox state is explicitly incorporated into runtime diagnostics: when complete system-level isolation is unavailable, the system degrades to a combined line of defense consisting of a permission profile, workspace constraints, sensitive-path protection, interactive approval, denial ledger, and invalidation of tool outputs—the paper regards this as controlled degradation rather than complete isolation. On the audit and state-cleanup side, after the denial ledger reaches a threshold through consecutive refusals, autonomous execution is automatically suspended to prevent the Agent from probing the security policy by repeatedly modifying commands, paths, or parameters. The expired-output cache mechanism immediately clears the tool-output cache with the same fingerprint after a refusal decision, blocking detours such as "first obtain sensitive output in a low-risk context, then read the previous output." On the input side, skill descriptions, tool returns, and external content are uniformly encapsulated in an envelope and escaped; external sources in long-term memory and model-generated content are treated as low-trust data to avoid commands in memory entries being mistakenly elevated to system-level rules.

In security evaluation, the results cannot be described solely by a sweeping attack success rate (ASR): semantic obedience to attacks, real harm supported by evidence in logs and file traces, and state changes that actually occur in the executable sandbox are three distinct endpoints, and should be reported separately with clear denominators and raw counts. Artificially constructed hard-biased attack sets are interpreted only as relative performance under stress testing. Accordingly, OpenSquilla' s security claims are limited to runtime protection and auditable execution boundaries: through the three-tier policy, system-level sandboxing, denial ledger, and expired-output cleanup, it reduces the probability that tool-type Agents cause actual harm; prompt-only defenses and complex strategy packs are used only as diagnostic components, and are not described as always-superior production-grade defenses.

5. System Architecture

OpenSquilla' s system architecture decomposes the AI Agent Harness into a clear microkernel and user-space capabilities. The microkernel is responsible for a stable turn lifecycle, an explicit state machine, concurrency control, and event streams; user-space capabilities are responsible for multi-model pools, tools, memory,

memory, skills, Meta-Skill channels, scheduling, and sandboxing. This division

is precisely the systematic realization of Agent = base model pool $\mathcal{M}$ + Harness: the microkernel provides a stable execution skeleton, while $g_t(\mathcal{M})$ and s_t are injected into the turn lifecycle as replaceable, learnable, and governable policies.

The advantages of this architecture are reflected in three aspects. First, capability expansion does not destroy the core execution path: new model providers, message channels, tools, skills, Meta-Skills, or memory strategies can be connected through boundaries. Second, failures are localized: model failures, tool rejections, channel anomalies, memory degradation, or skill proposals that fail to pass can all be handled within the corresponding boundary. Third, system observability is stronger: a complex Agent behavior can be decomposed into stage records such as input, capability parsing, prompt assembly, context governance, orchestration, streaming execution, and finalization. Thus, OpenSquilla forms an Agent-native microkernel harness suitable for long-cycle, multi-entry, high-risk tool invocation and continuous skill evolution scenarios.

5.1. Design Principle: Separation of Mechanisms and Policies

OpenSquilla' s system architecture adopts the microkernel idea. The "microkernel" here does not mean that the system has few capabilities; rather, it means that the Harness core retains only the minimal and stable execution mechanisms: session-level task orchestration, state-machine advancement, streaming event distribution, tool-invocation boundaries, error-termination semantics, and lifecycle management. Policy capabilities such as model selection, tool collections, memory retrieval, skill injection, Meta-Skill orchestration, channel sending and receiving, scheduled tasks, and sandbox isolation are connected outside the core through protocolized interfaces, registries, adapters, or services.

This design continues the operating-system idea of "separating mechanisms and policies." At runtime, the core does not directly hard-code a particular model provider, a particular channel protocol, or a certain memory strategy; instead, it provides stable execution timing and failure boundaries, while external capabilities evolve independently around these boundaries. A single Agent turn can be abstracted as the following composition:

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

Here, input normalization, capability parsing, prompt assembly, context governance, orchestration scheduling, streaming execution, and final persistence constitute the stable kernel path; model routing, skill filtering, Meta-Skill orchestration, memory recall, security policies, and cost accounting are injected into the corresponding stages as replaceable policies. Specifically, $F_{\text{capability}}$ and F_{prompt} carry the selection of $g_t(\mathcal{M})$ and the assembly of s_t, $F_{\text{orchestrate}}$ carries the structuring of Meta-Skill for s_t, and F_{finalize} settles the trajectory of this

round as the input for future s_t. This decomposition enables the system to elevate the process of "model generating text" into a controllable, observable, recoverable, and evolvable task-execution process.

5.2. Phased Turn Execution

In early implementations, many Agent frameworks tend to concentrate input processing, model selection, prompt concatenation, tool invocation, history writing, and error handling in a long main loop. This pattern is suitable for rapid prototyping, but in production systems it causes two problems: first, any capability expansion may affect the entire execution path; second, boundaries such as errors, cancellations, compression, tool invocations, and persistence are difficult to test independently.

OpenSquilla divides a single turn into several stable stages. Each stage only receives clearly defined inputs, produces clearly defined outputs, and accesses external dependencies through narrow interfaces. Table 5 gives an abstract view of phased execution.

The advantage of phased design is that it establishes "extensibility" on clear boundaries rather than relying on global variables or implicit invocation order. Model routing mainly affects prompt assembly and provider selection; the memory system mainly participates in prompt assembly and finalization; security policies mainly act on tool scheduling and sandbox boundaries; cost accounting is uniformly recorded in the finalization stage. Different capabilities exchange necessary information through stage outputs, thereby reducing cross-module coupling.

5.3. Explicit State Machine and Streaming Event Model

Within a turn, OpenSquilla does not regard the model call as a one-shot blocking RPC, but models it as an explicit state machine. A typical lifecycle includes:

$$\text{IDLE} \rightarrow \text{THINKING} \rightarrow \text{STREAMING} \rightarrow (\text{TOOL\_CALLING} \rightarrow \text{THINKING})^{*} \rightarrow \text{DONE/ERROR}.$$

Table 5 | Staged Abstraction of the OpenSquilla Turn Execution Path

Stage	Core question	Technical role
Input normalization	What is the current request?	Unify Web, CLI, channel, and scheduled-task entrances into runtime messages, semantic inputs, attachments, and source metadata.
Capability parsing	What can be used in this turn?	Parse model provider, tool visibility, tool-policy context, and runtime capability boundaries.
Prompt assembly	What should the model see?	Compose identity prompts, tool definitions, skills, memory, routing metadata, and cache boundaries.
Context governance	How does history enter this turn?	Load conversation history, perform necessary compression, and prevent history, attachments, and tool results from exceeding the context budget.
Orchestration scheduling	How should the task organize capabilities?	Determine workflow steps and recovery paths according to Meta-Skill, skill matching, user clarification, and checkpoint state.
Streaming execution	How do the model and tools alternate?	Drive model streaming output, tool invocation, tool-result injection, route rollback, and mid-course error handling.
Finalization	How does this turn conclude?	Persist assistant output, tool fragments, errors, memory captures, and token and cost statistics.

This state machine supports multiple types of streaming events, such as model text deltas, start of tool invocation, tool-parameter deltas, end of tool invocation, tool results, heartbeats, completion events, and error events.

The core value of this design is that it transforms the uncertainty in the Agent execution process into observable states. Long tasks may involve multiple tool invocations; the provider may report an error midway; the user may cancel the task; a tool may return an excessively large result; and output may trigger context compression. If these events are hidden only inside synchronous calls or exception stacks, it is difficult for the system to perform stable recovery. An explicit event model allows the runtime to apply policies separately for each class of event, such as retry, downgrade, compression, persistence of partial results, or generation of terminal errors.

Therefore, the Agent kernel of OpenSquilla is not a simple ReAct loop, but an event-driven state machine oriented toward production operation. It preserves the interaction paradigm of "reasoning–action–observation," while incorporating tool execution, streaming output, budget control, cancellation recovery, and error auditing into a unified runtime.

5.4. Unified Tool-Scheduling Boundary

Tool invocation is the part of an Agent system most prone to loss of control: a tool name and parameters generated by the model may correspond to file writes, command execution, network access, message sending, or external API calls. An important design in OpenSquilla is that tool functions are not exposed directly to the model for invocation; instead, a unified scheduling boundary is established between the model and the tools.

This boundary contains four key steps. First, the tool name must pass registry-table verification; unregistered tools are not executed. Second, tool invocation enters a policy chain, where the system determines whether execution is permitted by combining the invocation source, permission context, tool visibility, budget, and security rules. Third, actual execution occurs in the request-level context, so that the tool can perceive the current conversation, Agent, workspace, and invocation source. Fourth, tool results uniformly enter the finalization layer, with execution status, truncation status, artifact metadata, and error envelopes attached.

In set notation, the set of tools actually executable in this turn is not the global tool set $\mathcal{U}$, but a subset filtered by context:

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

where V denotes the visibility constraint, P denotes permission, security, and budget constraints, c_t is the context of this turn, and a_t is the action proposed by the model. This design separates "what the model wants to call" from "what

the runtime permits it to call," and is the basis on which security sandboxing, permission control, and cost budgets can take effect.

5.5. Cross-Interface Consistency and the Boundary of Pluggable Capabilities

The extensibility of OpenSquilla mainly comes from three kinds of boundaries: model providers, external channels, and capability plugins.

Model providers are connected through a unified streaming interface. Different providers only need to convert their own APIs into text deltas, tool calls, completion events, and error events; the core state machine does not need to be aware of differences in underlying protocols. This allows model routing, fallback, and cost statistics to operate over provider abstractions.

External channels are connected through a unified message model. Channel adapters are responsible for converting the entry points of different platforms into standard input messages, and for converting system outputs back into platform messages. The Harness core only handles normalized session IDs, Agent IDs, senders, attachments, and source information, without needing to care about platform-specific fields. Thus, changes in channel semantics can be repaired at the adapter and entry-normalization layers, without rewriting the Agent kernel.

Skills, Meta-Skill, and memory are connected as prompt, orchestration, and context capabilities. The skill system provides reusable task knowledge; Meta-Skill provides an inspectable way to organize workflows; and the memory system provides cross-session state. None of the three changes the core state machine; instead, they inject or deposit information during prompt assembly, orchestration scheduling, and finalization. This design ensures that long-term capability growth does not cause the complexity of the kernel to increase linearly, and also enables the Harness to keep learning new task-organization patterns within governance boundaries. As s_t evolves from static configuration into a learning policy driven by Meta-Skill Creator, Memory Dream, and diagnostic replay, the microkernel boundary ensures that this evolution remains auditable, replayable, and governable.

5.6. Concurrency Model: Serial Within a Session, Parallel Across Sessions

A production-grade Agent framework must simultaneously handle multiple entry points and long-running tasks. If all requests are globally serialized, the system throughput will be dragged down by a single slow task; if all requests are fully parallelized, multi-turn inputs within the same session may interleave writes to history, memory, and cost statistics, causing state disorder. OpenSquilla adopts a concurrency model of "serial within a session, parallel across sessions."

Let s denote a session, and let t_i^s denote the i-th turn in that session. Then the system expects:

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

That is, ordering is maintained within the same session, while parallelism is allowed between different sessions. At runtime, the system distinguishes between long-execution locks and short write locks: long-execution locks protect the turn life cycle of the same session, while short write locks only protect transient writes to the transcript and session state. In this way, model streaming output, tool execution, and waiting for external responses do not occupy the state write lock for a long time, reducing the risk of deadlocks and queue blocking.

This concurrency model also supports the practical requirements of channel-based Agents. Message channels may trigger multiple inputs within a short period of time, and scheduled tasks and sub-Agents may also run concurrently. Through queue limits, cancellation semantics, timeout semantics, and lifecycle events, the system can maintain controllable backpressure under high-concurrency entry points, rather than allowing tasks to accumulate without bound.

5.7. Lifecycle Governance and Fault Isolation

The value of the microkernel architecture lies not only in ease of extension, but also in fault isolation. When the system starts, the composition root is responsible for constructing configuration, session storage, model selection, tool registration, memory management, skill loading, scheduling, search, and channel-management dependencies. When the system shuts down, it stops background producers in dependency order, and then closes memory, sessions, and storage resources. This ordering avoids race conditions in which background tasks are still writing while the underlying database or memory index has already been closed.

Fault isolation follows the same idea. An unavailable model provider returns a stable termination event; tool privilege violations return structured rejection results; channel startup failure does not block other channels; when the memory vector backend is unavailable, the system can degrade to full-text retrieval; and failure to load a skill does not destroy the main state machine. Table 6 summarizes these boundaries.

In actual implementation, malformed or illegal tool-call histories should be intercepted before the model-provider boundary, rather than waiting for the remote model interface to return an unrecoverable error and then handling it. Similarly, semantic repairs such as workflow handoff, Meta-Skill clarification, and channel-reply context should be placed in entry normalization, capability orchestration, and adapter layers, rather than intruding into the core state machine.

Table 6 | Main Isolation Boundaries in the OpenSquilla Architecture

Boundary	Design Objective	Degradation Mode
Model-provider boundary	Isolate differences among model providers, errors, and fallback strategies.	When no provider is available, return a stable error; fallback may be performed according to policy.
Tool boundary	Isolate model intent from real-system actions.	When unauthorized or unavailable, return a structured rejection envelope.
Memory boundary	Isolate long-term state from execution of the current turn.	When vectors are unavailable, retain the full-text recall.
Channel boundary	Isolate different external messaging platforms.	Failure of a single channel does not affect Web, CLI, or other channels.
Scheduler boundary	Isolate background tasks from interactive sessions.	Task failures record execution results and do not damage the main service.
Sandbox boundary	Isolate tool execution from host-system resources.	When the backend is unavailable, explicitly degrade and record the risk.

5.8. Engineering Observability

The architecture of OpenSquilla pursues not only modular decomposition, but also emphasizes observability. A single complex Agent behavior is decomposed and recorded into stages such as input, capability parsing, prompt assembly, context governance, streaming execution, and finalization. The final event includes not only text output, but also tool fragments, artifacts, error types, model, provider, cached tokens, cost sources, and memory-related metadata. Thus, the system can review the key decisions of a turn, rather than retaining only a final natural-language answer.

Observability is especially important for a microkernel architecture. Because model routing, tool policies, memory recall, skill filtering, and sandbox isolation are distributed across different boundaries, without unified traces and decision logs it is difficult for the system to determine whether a failure originated from model capability, tool rejection, context overflow, memory mismatch, or channel adaptation. Through staged recording, OpenSquilla transforms complex Agent

behavior into engineering objects that are locatable, replayable, and regressively testable.

6. Experiments

To validate the end-to-end performance of OpenSquilla as an Agent-native learnable harness on real tasks, we evaluated it on standard Benchmarks commonly used in industry and compared it with representative Agent frameworks/harnesses. The experimental design corresponds to objective (2): using the single-model direct-run setting with an external harness as a baseline that provides an approximately fixed $\bar{g}_t$, then using OpenSquilla to force the contribution of the single model to the microkernel and s_t to be isolated, and finally using OpenSquilla multi-file routing to measure the impact of jointly optimizing g_t and s_t on the cost–capability Pareto frontier. In addition, the single-model routing ($k = 1$) special case of agentic routing described in Section 2 corresponds to §6.1, while the special experiments for multi-model fusion routing ($k > 1$) are given in §6.5.

6.1. PinchBench

PinchBench[1] is an evaluation benchmark for the real-task capabilities of code-oriented Agents, emphasizing "end-to-end practical outcomes" rather than synthetic isolated capability tests. This work uses a fixed version of the public task set and evaluator; 25 tasks cover six typical Agent workflows: file operations, data processing, web search, creative output, tool use, and memory recall, while also examining tool-call accuracy, multi-step reasoning chains, and the ability to handle ambiguities in real tasks. The grader for each task uses automated validation, LLM-as-a-judge, or a combination of both, and outputs a score in the interval $[0, 1]$; the total score in the table is the cross-task average score multiplied by

above 100.

To isolate the two factors of "model capability" and "framework contribution," under a unified supplier pricing path we compare three Agent frameworks: OpenClaw, Hermes Agent, and OpenSquilla. The first two are external harness baselines that run directly with a single model, respectively covering a flagship model and a high cost-performance model. OpenSquilla reports both forced single-model runs (the same model completes the full framework but routing is disabled, to measure the effect relative to direct single-model runs during microkernel execution) and multi-tier routing (representing two cost orientations: "flagship-model fallback" and "low-cost-model priority"). Token usage and cost are accumulated tier by tier according to the same price list.

Table 7 summarizes the total scores and total costs of the three frameworks under different models (pools). Overall, under the same flagship model Claude

[1] https://github.com/pinchbench/skill

Opus-4.7 (abbreviated below as Opus-4.7), OpenSquilla obtains the highest score, 93.85, improving by about +1.3 pp and +1.2 pp over OpenClaw (92.55) and Hermes Agent (92.65), respectively, while the cost instead drops from $6.23 / $6.66 to $4.84. This shows that, without sacrificing quality, microkernel runtime operations such as cache / Skill and s_t can already compress costs. Multi-tier routing forms a clearer cost frontier: the tier combination that bottoms out with Opus-4.7 stays close to direct Opus at a basic score of 92.51, while reducing the cost to$0.69; the low-cost combination with GLM-5.1 as the top tier still reaches a total score of 90.48, with the cost further reduced to $0.13. From objective (2), this means that $P(a)$ drops sharply while $\mathcal{L}_T(a)$ is basically stable, verifying the joint value of routing g_t together with cache / Skill and s_t operations.

Table 7 | PinchBench 25 task comparison table. Comparison of the total scores and costs of the three frameworks OpenClaw, OpenSquilla, and Hermes Agent under different models (pools). OpenRouter Auto is the official automatic routing service baseline of OpenRouter; its results were measured on the routing-specific path (PinchBench 1.2.1; an average per-task cost of $0.1204 is converted over 25 tasks into total cost), and are not directly comparable with the other rows.

Framework	Model (pool)	Total score	Cost ($)
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

In addition to the above overall harness comparison, we also conducted a separate specialized evaluation of the single-model routing described in Section 2 on PinchBench. This group of experiments differs from the main

path of Table 7 and is not directly comparable: PinchBench version 1.2.1 was used; the model pool was mainly the Opus-4.8 generation (the router gradually selects among Opus-4.8, GLM-5.2, and DeepSeek-V4 Flash); costs are counted by the average per-task charge; and the harness, task split, and scoring protocol are kept consistent within the same path. Under this path, with a score of 93.14, the routing strategy is almost tied with the OpenClaw fixed Opus-4.8 baseline (93.35; quality retention rate 99.77%), while the per-task cost drops from \$0.2224 to $0.0204, a reduction of about 10.9\times$. OpenSquilla with fixed Opus-4.8 obtains 94.33 points at \$0.1649. As a query-level routing baseline, OpenRouter Auto (OpenRouter' s official automatic routing service, running in the OpenClaw harness) obtains 88.10 points at $0.1204 per task (the converted total cost has already been included in Table 7). Relative to this baseline, the routing str
cheaper, showing that stepwise routing conditioned on the harness state is significantly superior to routing services that look only at the query.

6.2. ClawMark

ClawMark[2] is a comprehensive evaluation set of 100 questions for vertical-industry Agent tasks. The tasks cover 13 real business domains, including clinical assistant, content operations, e-commerce, EDA, executive assistant, human resources, insurance, investment analysis, news, legal assistant, product management, real estate, and research assistant. Unlike benchmarks that only examine question answering or code repair, ClawMark places more emphasis on multi-step information processing, table/document reading, visual-material understanding, and structured input in industry scenarios.

output and business constraints. Each task is assigned a score in $[0, 1]$ by an independent grader; in this paper, tasks with scores greater than or equal to 0.5 are marked as completed.

To isolate the two factors of "model capability" and "framework contribution," under a unified vendor pricing interface we compare **OpenClaw** with **OpenSquilla**: **OpenClaw** uses GLM-5.1 single-model direct run as the baseline; **OpenSquilla** reports both forced single-model execution (the same model runs through the full framework but routing is disabled) and multi-tier routing (a cost-side tier combination). To ensure consistent entry points for different systems when reading images, PDF pages, and screenshots, all configurations use the same artificial visual parsing pipeline: visual materials are first converted into textual observations and then handed to the corresponding agent; this vision model is not expanded as a routing tier in the table.

Table 8 summarizes the total scores and average per-task costs of the two frameworks under different models (pools). Overall, **OpenSquilla** with GLM-5.1 forced single-model execution obtains 77.0 points, leading all configurations, an improvement of +5.8 pp over the GLM-5.1 direct run of **OpenClaw** (71.2).

[2] https://github.com/evolvent-ai/ClawMark

The routing configuration is closer to a cost-side Pareto point: the combination using DeepSeek-V4 Pro as the middle tier obtains 70.6 points, close to the GLM-5.1 direct run, while reducing the per-task cost from $0.62 to $0.39. This result shows that, in industry-scenario tasks, s_t under a fixed model can improve task-completion quality, while after enabling g_t the system can move along the cost side, forming a more controllable $\mathcal{L}_T(a) - P(a)$ trade-off.

Table 8 | Comparison on the 100 ClawMark tasks. Comparison of the total scores and costs of OpenClaw and OpenSquilla under different models (pools). All scores are percentages (×100); cost is the average billed cost per task.

Framework	Model (pool)	Total score	Cost ($)
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 is a multilingual SWE-bench-style benchmark for evaluating agent harnesses. It contains 350 repair tasks from real GitHub Issues, covering 8 programming languages and 43 repositories. Its evaluation protocol fixes the task set, Docker runtime container, Prompt template, maximum number of rounds, and timeout upper bound, leaving only the harness implementation as the controlled variable; thus, differences at the harness level are separated from LLM capability and task difficulty. All harnesses share the same task Prompt, evaluation image, baseline submission, working directory, maximum number of interaction rounds, wall-clock timeout, and single-instance one-time execution constraints; candidate patches are uniformly cached and extracted by the benchmark runner, and then judged by the evaluator as one of three states: Resolved / Unresolved / Empty Patch. The resolution rate in the table is the proportion of Resolved instances among the 350, and cost is the average billed cost per task (converted from the official list Cost / 350 interface).

To isolate the two factors of "model capability" and "framework contribution," under the unified protocol above we compare three types of systems: (i) **OpenClaw** single-model direct run—Opus-4.7, GLM-5.2, GLM-5.1, and Qwen3.7-Max, four mainstream models, each independently complete all 350 tasks, serving as model-capability baselines; (ii) **OpenSquilla** forced single-model—the same model runs through the full **OpenSquilla** framework but with routing disabled, selecting the two capability tiers GLM-5.2 and GLM-5.1 [42] to measure the pure framework effect; (iii) **OpenSquilla** multi-tier routing—using the three-tier combination {DeepSeek-V4 Flash, GLM-5.1, GLM-5.2}, with the local

Figure 4. ClawSWEBench cost–resolve-rate view.

Figure 2: Figure 4. ClawSWEBench cost–resolve-rate view.

router described in Section 2 dynamically assigning tiers for each task, measuring the impact of low-tier sinking and top-tier fallback on the cost–resolution-rate frontier.

Table 9 summarizes the resolution rates and average per-task costs of the configurations on the 350 tasks. Overall, **OpenSquilla** with GLM-5.2 forced single-model obtains the highest resolution rate of 278/350 (79.4%), improving by +5.1 pp over the GLM-5.2 direct run of **OpenClaw** (260/350, 74.3%), and surpassing the strongest **OpenClaw** single model Opus-4.7 (77.1%) at about 31% of the cost ($0.95 vs $3.09); GLM-5.1 entering the framework likewise gains +1.4 pp, indicating that **OpenSquilla**' s tool boundaries and dialogue governance can stably amplify code-repair capability without changing the model. Multi-tier routing gives a more favorable

---

^3 https://claw-swe-bench.github.io/

Figure 4 | ClawSWEBench cost–resolve-rate view. The colored circles are OpenClaw single-model direct-run baselines; the colors correspond to model families (blue: Opus, teal: GLM, yellow: Qwen). The orange markers (connected by a line) are the two operating points of **OpenSquilla**—the star is forced single-model GLM-5.2 (the highest-quality point), and the square is multi-tier routing (the lowest-cost point). The cost axis is on a logarithmic scale.

the cost-side Pareto point: a resolve rate of 259/350 (74.0%) is essentially tied with OpenClaw' s GLM-5.2 direct-run baseline, while the cost is compressed to half of the latter ($0.44 vs $0.87). The routing distribution shows that 207 tasks (59%) are sunk to the low-cost DeepSeek-V4 Flash, resolving 143 of them (69.1%); 139 tasks float up to top-tier GLM-5.2, resolving 113 (81.3%); and the remaining 4 tasks fall to mid-tier GLM-5.1—i.e., the router leaves most of the budget to difficult problems that truly require top-tier capability. From objective formula (2), under a fixed model, s_t directly lowers $\mathcal{L}_T(a)$ on mid- and high-tier models, while g_t halves $P(a)$ with almost no loss in resolve rate; together, the two expand the cost–capability frontier for the code-repair scenario. Figure 4 gives the corresponding cost–resolve-rate view: neither of the two operating points of **OpenSquilla** is dominated by any OpenClaw baseline—the forced single-model GLM-5.2 occupies the highest-quality point, while multi-tier routing stands at the far left of the frontier with the lowest cost.

Table 9 | ClawSWEBench 350-task comparison table. Comparison of the resolve rate and cost of OpenClaw and **OpenSquilla** under different models (pools). Cost is the average charged cost per task (computed from the official list price divided by 350); the resolve rate of Opus-4.7 is taken from the official

leaderboard.

Framework	Model (pool)	Resolve rate (%)	Cost ($)
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 is a cross-domain deep-research benchmark containing 100 complex research tasks, scored by a grader along five dimensions: factual accuracy, completeness, objectivity, presentation quality, and citation quality. Its long execution trajectories, in which evidence retrieval, analytical synthesis, and citation generation are intertwined, pose challenges for harness context governance and model-capability allocation. Compared with common Q&A or coding benchmarks, DRACO better reflects the end-to-end performance of Agents on long-horizon research tasks.

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

We follow the comparative logic of the preceding three subsections, isolating two types of factors— "model capability" and "harness contribution" —under a unified task set and evaluation protocol: OpenClaw takes fixed Opus-4.8 single-model direct running as the external harness baseline; OpenSquilla reports both forced single-model performance (also using Opus-4.8 to run the full harness but disabling routing) and multi-file routing ({DeepSeek-V4 Pro, GLM-5.2, Opus-4.8}, progressively allocated by the local router described in Section 2). Costs are computed as the average billing cost per task.

Table 10 summarizes the total score and cost of each configuration on 100 tasks. Overall, OpenSquilla' s forced single-model setting slightly exceeds OpenClaw' s Opus-4.8 direct run, with 52.36 points versus 52.13, while cost drops from $1.1420 to $0.6559—showing that the framework' s own tool boundary and context management already make a positive contribution on deep-research tasks. After enabling multi-file routing, the system retains 99.94% of the strong-model configuration' s quality (52.33 vs. 52.36), while cost further drops to $0.3729 (43.15% lower than the strong-model configuration, and 67.3% lower cumulatively than the OpenClaw direct run). Notably, token usage in the routing configuration is instead slightly higher than in the fixed configuration—the savings

do not come from shortening prompts, but from changing the price composition of model calls: parts of the execution trace for which capability is sufficient are sunk to cheaper models. This result, together with the single-model routing ablation in §6.1, jointly supports the capability-matching view in subsection 2.2: model capability can be selectively allocated inside the harness according to state, without treating every execution state as requiring the strongest model.

Table 10 | DRACO 100-task comparison. Total score and cost of OpenClaw and OpenSquilla under different model pools. The total score is the average score across tasks; cost is the average billing cost per task; tokens are input plus output (thousands); the routing threshold is set to 0.95.

Framework	Model (pool)	Total score	Cost ($)	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. Multi-model fusion routing experiments

The preceding subsections compare OpenSquilla with external frameworks from the overall harness perspective; this subsection conducts a dedicated evaluation of the multi-model fusion routing ($k > 1$) described in subsection 2.3, examining whether a complementary proposer set and aggregation can move the quality-cost frontier relative to the strong single-model baseline. All configurations in this group of experiments—including each single-model baseline and the fusion configurations—run in the same OpenSquilla harness, ensuring the execution substrate is consistent. DRACO is the main benchmark, because deep-research tasks often benefit from the strengths of complementary models: one model may retrieve better evidence, another may synthesize more coherent analysis, and a third may produce more reliable citations. The selected configuration uses DeepSeek-V4, GLM-5.2, Gemini 3 Flash, and Qwen3.7 as proposers, GLM-5.2 as the aggregator, and adds random sampling to the inexpensive Gemini and Qwen proposers.

Table 11 reports single-model baselines and the selected multi-model configuration on DRACO (default DuckDuckGo search). The strongest quality point among single models is Fable 5, averaging 59.80 points over the 94 tasks it executed, at a per-task cost of $1.2122; the multi-model configuration in this paper achieves 60.82 points over the full 100 tasks, at an average cost of $0.3766 —improving quality by 1.02 points and reducing cost by 68.9% relative to Fable 5. This is precisely the money-saving substitution mechanism predicted by the frontier formalization in Eq. (10): complementary inexpensive proposers

plus an aggregator can outperform the strongest single-model baseline simultaneously on both the quality and monetary-cost dimensions. As the expected cost of ensemble strategies, this operating point consumes more tokens and wall-clock time; latency is the deployment axis of the main text and can be controlled through parallel proposer execution, early stopping, and stricter proposer budgets. Compared with mixture-of-agents baselines, Hermes MoA (preset virtual model provider; reference models first generate private suggestions, and a fixed aggregation model performs the action [24]) obtains 59.55 points at $0.4460, while the configuration in this paper is better on both dimensions (60.82, $0.3766). After switching the search provider to Brave (Table 12), the substitution effect is stronger: the configuration in this paper obtains 64.09 points, with a per-task cost of only $0.1218; relative to Fable 5 (62.06, $1.3241), it improves by 2.03 points and reduces cost by 90.8%; relative to Opus-4.8 (59.11, $1.6177), it improves by 4.98 points and reduces cost by 92.5%; relative to GPT-5.5 (53.28, $0.8407), it improves by 10.81 points and reduces cost by 85.5%. On the already near-saturated short-task benchmark PinchBench, the corresponding working mechanism is instead cost-efficient quality preservation: the multi-model configuration maintains approximately the same quality as the strongest single model, Opus-4.8, with 94.31 points versus 94.33 (a gap of only 0.03 points), while cost drops from $0.1649 to $0.1349 (−18.2%); relative to GPT-5.5 (93.73, $0.0963), this configuration is 0.58 points higher in quality, but more costly.

Table 11 | Comparison on DRACO (default DuckDuckGo search) between single-model baselines and the selected multi-model fusion routing configuration. All configurations are run in the OpenSquilla harness. Cost is averaged per task; tokens are counted in thousands. Fable 5 completed 94/100 tasks, and its metrics are averaged over the completed tasks; all other configurations completed all 100 tasks.

Method	Total score	Cost ($)	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	49.34	0.0432	99.5
Gemini 3 Flash	40.79	0.0117	9.5
Multi-model fusion routing (this paper)	60.82	0.3766	579.7

Table 12 | Comparison on DRACO (Brave search) between single-model baselines and the selected multi-model fusion routing configuration. All configurations are run in the OpenSquilla harness. Cost is averaged per task; tokens are counted in thousands (visible tokens plus inference tokens). Because safety filter-

Figure 5: DRACO cost-score plot

Figure 3: Figure 5: DRACO cost-score plot

ing blocked 7 tasks, Fable 5 completed only 93/100 tasks, and each of its metrics is averaged over the 93 completed tasks; all other configurations completed all 100 tasks.

Method	Total score	Cost ($)	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
Multi-model fusion routing (this paper)	64.09	0.1218	500.1

The above results use a preselected multi-model configuration: the proposer set is fixed before each run. Further, we couple the agentic router with the aggregation stage: at each run, the routing strategy dynamically assembles the proposer set, while GLM-5.2 is fixed as the aggregator. On DRACO (default DuckDuckGo search), we evaluate three routing-strategy operating points: *control* (the current routing strategy), *diversity-heavy* (the complementarity term in weighted formula (10), with a larger α), and *quality-heavy* (emphasizing predicted single-model quality, with a lighter cost weight λ). Table 13 reports the three sets of results, from which two observations can be drawn. First, diversity-heavy is the optimal routing strategy (60.31 vs. 59.18 for control), and the cost is slightly lower—direct evidence for the positive effect of the complementarity regularizer: steering the router toward more relevant proposer sets can improve aggregation quality without increasing overhead. Second, quality-heavy trades a small quality gap (59.93) for the lowest cost ($0.2582), providing a more economical operating point on the same frontier. The best dynamic-assembly group (diversity-heavy, 60.31, $0.3172) is close to the manually selected configuration in Table 11 (60.82, $0.3766), with 15.8% lower cost and no need to manually select the proposer set.

Table 13 | Dynamic assembly integration on DRACO: the router assembles the proposer set at runtime, with GLM-5.2 fixed as the aggregator. All configurations are run in the OpenSquilla harness. Cost is averaged per task; tokens are counted in thousands.

Routing strategy	Total score	Cost ($)	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

Figure 5 | Cost-score view of single-model routing and multi-model

fusion results on DRACO. The colored dots are fixed single-model baselines, with colors corresponding to model families (blue: Opus; green: GPT-5.5; purple: Fable 5); the orange square and star (connected by a line) are respectively the single-model routing and multi-model fusion methods in this paper; the cost axis is on a logarithmic scale.

Figure 5 summarizes the single-model and multi-model fusion results into a cost-score view on DRACO (excluding the OpenClaw harness baseline and baselines that are cheaper than the single-model routing method in this paper). The two operating points of this work dominate the strongest baselines: multi-model fusion obtains a higher total score than Opus-4.8 and Fable 5 at far lower cost, while single-model routing stays near the quality point of Opus-4.8 at a small fraction of the cost. On PinchBench, the strong baselines are already close to saturation, and single-model routing is the cheapest operating point; the integrated configuration reaches nearly the quality point of Opus-4.8 at even lower cost. Overall, agentic routing achieves the frontier shift in the sense of Equation (3) on both benchmarks: the $k = 1$ regime provides cost-side compression, while the $k > 1$ regime simultaneously advances quality and cost on deep-research-type tasks.

7. Related Work

7.1. Agent Systems

From a macro-level perspective, the main line of development from 2023 to 2025 has been to optimize model parameters by expanding the Scaling Law, data scale, and training compute, forming a foundation-model pool M represented by GPT [32], Claude [2], GLM [43], Kimi [34], DeepSeek [38], and others. In the Agent era, the key question shifts from "how to train a stronger single model" to "how to learn to schedule and constrain a model pool," that is, how to organize M as **Agent = Base Models + Harness**. The objective Equation (2) of this paper is precisely to characterize the value of the Harness under this shift: selecting models through g_t, and organizing prompts, RAG, tools, memory, and safety boundaries through s_t, thereby improving Intelligence/Token.

The core idea of large language model Agent systems is that models no longer merely generate text, but explicitly maintain intermediate states during task execution, call external tools, observe environmental feedback, and dynamically revise subsequent actions accordingly. ReAct [41] is one representative early paradigm in this direction. It alternates reasoning traces and actions, enabling models to obtain information from the external environment in question answering, fact verification, and interactive decision-making tasks, and to adjust existing plans during execution. Compared with pure prompt-engineering methods, ReAct abstracts the "reasoning–acting–observation" loop into a reusable interaction pattern, laying the conceptual foundation for later tool-calling Agent frameworks.

At the engineering implementation level, LangChain [5] wraps models, prompt

templates, tool interfaces, retrievers, vector databases, and external APIs into composable standardized components, significantly lowering the engineering barrier to building LLM applications and Agent prototypes. As the complexity of Agent applications evolves from simple chained calls to long-range, recoverable, and controllable workflows, LangGraph [17] further models the Agent execution process as a directed graph, emphasizing durable execution, human-in-the-loop collaboration, branch control, and long-term state management. LlamaIndex [20], in turn, takes data connection and RAG as its entry point and has gradually expanded into an Agentic application framework for document processing, index construction, and workflow orchestration.

architecture, and is especially suitable for scenarios that require close coupling of private data, information retrieval, and tool invocation.

Multi-agent systems are another important research branch. AutoGen [36] organizes multiple conversational Agents into a group-chat-style collaborative session structure: each Agent can combine LLM reasoning, human input, and tool execution, and collaboration is carried out through natural-language dialogue, thereby decomposing complex tasks into specialized Agents with different roles, permissions, or tool sets. CrewAI [7] likewise adopts a role-based collaboration approach, using the abstractions of Crews and Flows to organize autonomous Agents and event-driven workflows, which better matches enterprise-level automation and process orchestration requirements. A common feature of such systems is that they extend a single model call into multi-round, multi-role, multi-tool collaborative execution; at the same time, however, they introduce a series of engineering challenges, such as maintaining state consistency, suppressing error propagation, permission control, and system observability.

Recent open-source personal Agent systems place greater emphasis on local execution, cross-channel interaction, and real tool execution. For example, OpenClaw [33] integrates gateways, sessions, tools, and multi-channel message entry points into a local-first personal assistant system, supporting file operations, Shell execution, browser automation, scheduled tasks, and multi-Agent routing. Hermes Agent [25] further emphasizes, in similar form, long-term memory, dynamic skill generation, conversation retrieval, scheduled tasks, and multi-backend deployment. nanobot [29] attempts to provide long-running Agents, chat channels, memory mechanisms, and the Model Context Protocol (MCP) with smaller and more readable core code. Viewed comprehensively, the above work shows that Agent systems are evolving from the ReAct-style "reasoning–action" prompting paradigm toward a complete Harness infrastructure covering model routing, tool abstraction, state management, Skill orchestration, persistent memory, security sandboxing, and observability.

7.2. Efficient Agent Technologies

Work on end-to-end inference efficiency for LLMs and Agents can be roughly divided into three categories according to the optimization target: on the context

side, it addresses the quadratic growth of attention memory and computation with sequence length; on the input side, it focuses on redundant Tokens in the Prompt; and on the model side, it amortizes invocation costs at the two granularities of requests and Tokens by "using a small model first and a large model as the fallback."

Context side: **KV Cache compression and sparse attention.** This line of work targets two bottlenecks in long-context processing: KV cache memory grows linearly with sequence length but is large in absolute size, while attention computation grows as $O(N^2)$ with sequence length. The common objective is to substantially reduce these two types of overhead without losing generation quality. StreamingLLM [37] observes that the earliest few tokens in long streaming inference play the role of an *attention sink*, and that retaining only the sink and the KV of a sliding window is sufficient for stable generation. H2O [44] further dynamically retains the Heavy Hitters with the highest accumulated attention scores. SnapKV [19] moves the decision time to before generation and selectively compresses the KV cache according to the attention patterns inside the prompt. PyramidKV [4] performs pyramidal allocation along the layer dimension, reflecting the funnel-shaped structure of information leakage across attention layers. In addition to "which caches to discard," KIVI [21] starts from numerical precision and gives an asymmetric 2-bit post-training quantization for the KV cache. MInference [13] identifies three patterns in attention matrices—A-shape, vertical-slash, and block-sparse—and uses sparse attention kernels to accelerate long-context prefill.

Input side: **Prompt compression and Token-count reduction.** Context-side work targets existing contextual KV storage and attention computation, whereas input-side work goes one level higher and directly reduces the Prompt length seen by the LLM. LLMLingua [12] uses a small model to estimate the perplexity of each token and performs coarse-to-fine compression of the prompt at both sentence and token granularities. LLMLingua-2 [28] trains a task-agnostic prompt compressor through data distillation, enabling faithful and rapid compression for various downstream tasks. LongLLMLingua [14] builds on this by introducing key-information density estimation and positional-bias compensation for long-context scenarios, alleviating the problem that key information in long documents is "compressed and diluted." In the direction of soft compression, Gisting [23] trains an LLM to compress reusable prompts into cacheable gist tokens, so that subsequent uses of the same prompt can be reused directly from the cache. ICAE [10] trains a contextual autoencoder to further compress long contexts into short soft prompts.

Model side: **small-large model collaboration and dynamic routing.** The core idea of this direction is not to hand every request to the same model, but to route it, according to granularity, to models of different scales. At the token level, Speculative Decoding [18] accelerates single-pass decoding by having a draft small model generate and a large model verify, compressing the original N-step serial generation into near-one-step parallel verification. Medusa [3]

attaches multiple sets of parallel decoding heads to the main model, eliminating the need for an independent draft model. At the request level, FrugalGPT [6] learns an LLM cascading strategy and jointly selects model combinations among invocation cost, answer quality, and early-stopping decisions. AutoMix [1] estimates the confidence of small-model answers with a small number of self-verification samples, and then a POMDP router decides whether to escalate to a larger model. RouteLLM [26] uses preference pairs to supervise and train a router, making binary classification decisions between stronger and weaker models.

The above three categories of work all focus on the local efficiency of a single LLM inference pass, corresponding respectively to the local optimization of $P(a)$ or s_t in objective (2): KV cache and prompt compression reduce input-side tokens; dynamic routing lowers the unit price of model calls; and cascade strategies trade off quality against early stopping. When the evaluation target rises from LLM to Agent, HAL [15] incorporates cost and token usage into a standardized agent evaluation protocol across models, scaffolds, and benchmarks; SWE-Effi [9] further treats token and time budgets as resource constraints and re-evaluates the effectiveness of agent systems under a fixed budget. Together, they make end-to-end inference efficiency a first-class quantitative objective alongside solve rate. The distinction of OpenSquilla is that it does not treat these techniques as isolated optimizations, but instead incorporates them into a learnable Harness, enabling g_t and s_t to work jointly during end-to-end execution across subtasks.

8. Conclusion and Discussion

This paper introduced OpenSquilla, an open-source agent-native learnable harness for real tool execution, multi-entry interaction, and long-horizon collaboration. Unlike the system route that simply pursues "stronger models," OpenSquilla decomposes agent capability into four classes of system resources: routable model budgets, evolvable Meta-Skill schedules, governable long-term memory, and auditable tool boundaries. It then brings these resources into a unified turn lifecycle through a microkernel-style execution path. From a formal perspective, these four classes of resources are precisely the engineering decomposition of $g_t(\mathcal{M})$ and s_t in objective (2): agentic routing and runtime cost mechanisms jointly optimize $P(a)$ (multi-model fusion routing can still simultaneously improve $\mathcal{L}_T(a)$ and $P(a)$ on deep research tasks), Meta-Skill and memory improve the learnability of s_t, and safety governance constrains the executable scope of $g_t \circ s_t$. Experimental results show that, on end-to-end tasks such as PinchBench, ClawMark, ClawSWEBench, and DRACO, OpenSquilla can improve the cost curve while maintaining competitive task performance. This is manifested in particular by a cost–capability Pareto optimization jointly formed by local routing, on-demand injection of prompt cache continuity techniques, and harness-level observability.

Three cost-level issues in the above results merit further discussion: why multi-model fusion can achieve lower cost with more models, how cross-model routing

interacts with prompt-cache continuity, and the overhead of the router itself.

Multi-model fusion achieving lower cost with more models seems contradictory, but in fact it stems from the composition of cost: cost equals token usage multiplied by unit price, and the unit price of flagship models is usually one to two orders of magnitude higher than that of mid- and low-tier models. Therefore, a combination consisting of several low-price proposers and a mid-tier aggregator can still have a total cost far below that of a single flagship-model trajectory, even if its token usage is several times that of a single model. On DRACO, the per-task token usage of the selected fusion configuration is 579.7K (only 93.7K in Fable 5), yet its cost is only \$0.3766, about 30% of Fable 5 (\$1.2122) (Table 11). As for flagship-level quality that low-price models themselves cannot reach, it is compensated for by mechanism design: the complementarity regularization term in equation (10) encourages the failure modes of the selected proposers to be mutually decorrelated, so that an aggregator conditioned on verification signals guarantees that the final result is no worse than the best proposer.

Of course, cross-model routing also introduces new overhead: prompt caching on the provider side is isolated by model. Once models are switched, the previously cached stable prefix immediately becomes invalid and must be fully billed again. In response, the system imposes constraints at three levels. First, the router optimizes trajectory-level cost $C(\tau)$; the prefix-reloading overhead required by switching has already been included, so switching occurs only when the expected price gain exceeds the cache loss, equivalent to a hysteresis mechanism that suppresses repeated oscillation between tiers. Second, routing decisions are aligned with the task being executed: in multi-turn conversations within the same session, models are not switched every turn; instead, the same tier is maintained within the same task stage, allowing the prefix cache to remain continuously hit. Finally, the stable-base/dynamic-tail prompt organization (subsection 3.3) compresses the prefix that must be repaid to a minimum; cache-hit and invalidation events are both recorded in logs for auditing. The DRACO results confirm that, after these two effects offset each other, the net benefit remains positive: although the token usage of the routing configuration is slightly higher than that of the fixed configuration, cost still drops by 43% (Table 10), indicating that the benefit from the model unit-price structure is sufficient to cover cache losses. Further explicitly incorporating cache state into routing decisions —i.e., cache-aware routing—is a natural direction for subsequent iterations of the router model.

Finally, there is the overhead of the router itself. The router must run at every step, and its overhead must be significantly smaller than the savings it brings in order to be self-governing. The open-source $g^{(0)}$ is a local LightGBM pipeline that completes one decision in milliseconds, consumes no LLM tokens, and sends no data to remote services; relative to one model call, its marginal cost can be ignored. Although the subsequent learning cost is heavier, the router itself remains a small model (for example, at the 4B scale), still quite inexpensive compared with calls to the large models being routed, and the router decision

emits very few tokens. In deployment, the two-tier architecture (subsection 2.4) is still maintained: states that are clearly trivial or clearly difficult are handled directly by a low-overhead admission filter; the learned router is enabled only on uncertain, high-value, or weakly recoverable states. At the same time, the promotion-threshold design uses measured frontiers for the router' s own overhead and latency as the criterion, thereby mechanistically ensuring that the router' s overhead does not, in turn, offset the savings it creates.

Overall, the core claim of OpenSquilla is that the competitiveness of production-grade Agents should not be determined solely by model leaderboards, but instead should be jointly determined by task completion rate under unit cost, learnable Skill orchestration capability, long-term context-governance capability, and real-world tool-execution safety. From the era of the Scaling Law to the era of Agents, the competitive focus is shifting from training a stronger $\mathcal{M}$ to designing a better Harness; OpenSquilla is an open-source exploration under this paradigm shift, using a learnable Harness to improve Intelligence/Token. As users continue to use it, the Harness can precipitate preferences, processes, memory, and safety boundaries into auditable assets, drawing ever closer to users' ways of working on similar tasks. OpenSquilla is open source under the Apache 2.0 license, with code hosted at https://github.com/opensquilla/opensquilla. Community contributions and feedback are welcome.

Appendix

A. Contributors

Core contributors: Han Kai, He Wei, Zhou Hang, Liu Tingchen, Jiao Qingrui, Tian Yuchuan, Zheng Mengyu, Zong Yingjie, Liu Runke, Zhang Shuo, Shu Meng, Cheng Siyang, Song Liuyang, Zhang Hongbo, Fan Jiayu, Hu Hailin, Wang Yunhe

Contributors: Guo Tianyu, Kuang Xiang, Li Hongguang, Li Yulong, Li Zhexiang, Liu Xi, Liu Junyi, Liu Yishan, Pei Zehua, Wang Keya, Wu Yihong, Xie Zhaokai

B. Glossary

Most of the core terms in this report come from English literature. Table 14 gives the Chinese–English correspondences for the core terms most commonly used in the main text.

Table 14 | Chinese–English Glossary

Chinese term	Original English
agent	agent
large language model	large language model (LLM)

Chinese term	Original English
token	token
harness	harness
learnable harness	learnable harness
routing	routing
singleton routing	singleton routing
multi-model ensemble routing	multi-model ensemble routing
proposer	proposer
aggregator	aggregator
off-policy learning	off-policy learning
prompt injection	prompt injection
MMR re-ranking	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. LLMLingua: 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. LongLLMLingua: 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. LLMLingua-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/5a7c947568c1b1328ccc523 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. H_2O: 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.

Note: Figure translations are in progress. See original paper for figures.