OpenSquilla: Token-Efficient Agent = Models + Routing Harness

TokenRhythm Technologies

https://github.com/opensquilla/opensquilla

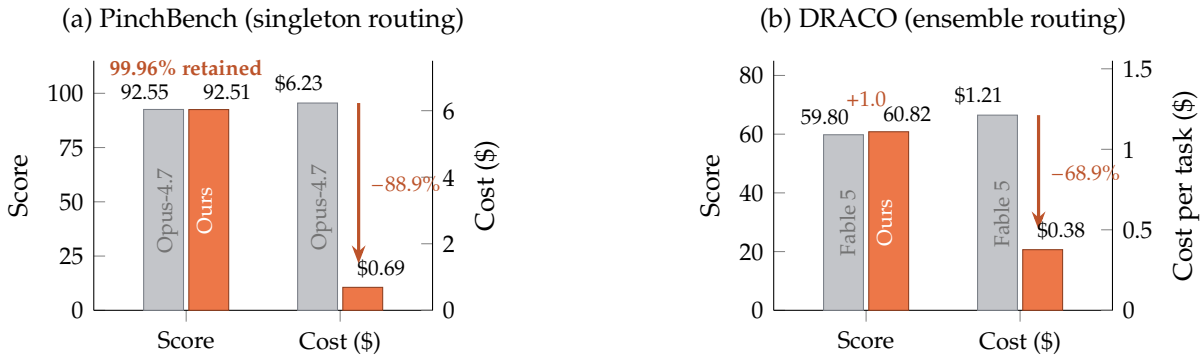

Figure 1 | **Headline results of OpenSquilla.** (a) PinchBench: gray bars are OpenClaw running the fixed flagship Opus-4.7, orange bars are OpenSquilla with multi-tier routing; quality is retained at 99.96% (92.51 vs 92.55) while cost drops by 88.9% ($0.69 vs $6.23). (b) DRACO: gray bars are the strongest single model Fable 5, orange bars are our multi-model ensemble; quality exceeds the baseline by 1.0 points (60.82 vs 59.80) while per-task cost drops by 68.9% ($0.3766 vs $1.2122). In each panel the left axis reports task score and the right axis cost.

Abstract

OpenSquilla is an open-source, agent-native learnable harness built on one premise: Token-Efficient Agent = Models + Routing Harness—agent capability comes from allocating a pool of heterogeneous models, not from binding to a single strong one. On end-to-end agent benchmarks, its local step-level router preserves 99.96% of fixed-flagship task quality while cutting cost by 88.9%, and its multi-model ensemble routing outscores the strongest single-model baseline (Fable 5) on deep-research tasks at 31% of that baseline's cost. Underneath is a microkernel design: model pools, tools, memory, skills, channels, schedulers, and sandbox policies all plug in through standardized boundaries, separating a stable execution mechanism from evolvable policy. OpenSquilla offers three core competitive advantages. (1) Singleton routing cuts cost: model selection is formalized as a step-level routing operator conditioned on the harness state; in the cost-effective mode the router selects the cheapest sufficiently capable model for each execution step, coupled with runtime mechanisms such as reasoning-budget derivation, prompt caching, and on-demand skill injection, reducing end-to-end cost severalfold while preserving task quality almost unchanged. (2) Multi-model ensemble routing improves quality: in the high-accuracy mode the router assembles complementary model sets and fuses their

candidate outputs, exceeding strong single-model baselines in task quality at lower cost and pushing the quality–cost frontier on deep-research tasks. (3) System mechanisms for a learnable harness: the Meta-Skill orchestration layer teaches the agent to retrieve, filter, compose, and evolve skills, distilling user collaboration history into reviewable new workflows; the persistent memory system maintains an editable, retrievable, forgettable, and auditable long-term context for cross-session collaboration; and security governance draws the boundary for real tool execution with tiered policies, runtime isolation, and an audit loop. Together they provide capability orchestration, cross-session state, and execution boundaries, letting the system improve with use while remaining reviewable and controllable. Comparative analysis shows that the value of OpenSquilla lies not in stacking a single large model, but in raising the agent capability obtainable per unit cost through agentic routing (singleton routing for cost, multi-model ensemble routing for quality), Meta-Skill orchestration, long-term memory, and governed tool boundaries.

1. Introduction

With the rapid progress of large language models (LLMs) [27], LLM-based agents have shown great potential in code generation, knowledge question answering, and task automation. To move agents from experimental prototypes to large-scale production, academia and the open-source community have produced a series of agent frameworks and runtimes such as LangChain [5], OpenClaw [33], and Hermes Agent [25]. When building production-grade agents for high-frequency interaction, long-horizon collaboration, and real tool execution, however, existing frameworks face three core challenges.

Runaway cost. Complex tasks typically require multi-turn dynamic planning and long-context interaction, so token consumption grows in steps with task complexity. A system permanently bound to a single strong model pays flagship prices even for low-difficulty requests such as short confirmations, format conversion, and short-text retrieval; a system that relies only on static rules or keyword dispatch easily mis-routes complex tasks to weak models and pays for failed retries. The lack of a local, auditable, degradable model-routing mechanism is the key reason agent cost fails to scale adaptively with task difficulty.

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

A harness that is hard to learn and govern. Real user tasks span retrieval, analysis, tool calls, clarification, review, and final synthesis, yet traditional skills encapsulate single capabilities: relying on ad-hoc model planning makes successful paths hard to reuse and new workflows hard to distill from collaboration history. Most frameworks' memory mechanisms are limited to stacking session context or dumping full transcripts into a vector store, unable to separate long-term facts, short-term tasks, audit evidence, and stale noise, and prone to carrying historical

The agentic-routing component of this report (Section 2) was previously presented as a standalone paper, *Agentic Routing: The Harness-Native Data Flywheel*; this report incorporates that work and extends it into the full OpenSquilla system.

noise and latent prompt injection into future context. Meanwhile, when an agent performs file writes, shell commands, and network requests, risk expands from inappropriate text to real state changes on the host; model self-restraint or one-shot approvals cannot cover compound attack paths such as tool-return injection, memory poisoning, and repeated probing, while heavyweight containers add deployment complexity. In short, a harness must be simultaneously learnable, auditable, and governable along three dimensions: capability orchestration, cross-session memory, and runtime security boundaries.

Viewed against the broader evolution of AI systems, the dominant paradigm of 2023–2025 optimized model parameters around datasets, pre-training compute, and scaling laws, producing a pool of capable base models with heterogeneous cost structures $\mathcal{M} = \{M_1, \ldots, M_n\}$, e.g., GPT [32], Claude [2], GLM [44], Kimi [34], and DeepSeek [39]. In the agent era, system value is no longer determined solely by a single model's leaderboard capability, but by whether multiple base models, tools, memory, skills, and safety boundaries can be organized into a stable task-execution system. The basic form of an agent can thus be abstracted as

$$\text{Agent} \triangleq \mathcal{M} + \text{Harness}. \tag{1}$$

Under the paradigm of Equation (1), an agent that truly assists or partially replaces human work should be valued jointly by task value, success rate, and token efficiency. We state this as a harness-optimization problem that maximizes intelligence per token (Intelligence/Token):

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

where $\mathcal{L}_T(a)$ and $P(a)$ denote the agent's normalized loss and normalized execution price on task T; their product is a coarse cost-effectiveness measure—lower loss and lower price are both better. Here g_t is the model gate on the t-th subtask, selecting a suitable model from the pool $\mathcal{M}$; s_t is the harness operation applied to the selected model, including prompt organization, retrieval-augmented generation (RAG), skill injection, memory recall, tool policy, and safety constraints; and $\circ$ denotes the composition of the model selection $g_t(\mathcal{M})$ with the harness operation s_t, not numerical multiplication.

Within this framework, the three challenges map to distinct failure modes of the objective: runaway cost means g_t always takes the strongest model and $P(a)$ cannot scale with task difficulty; the single-model quality ceiling means g_t selects only one model per step, so $\mathcal{L}_T(a)$ is bounded by the strongest single model; and a hard-to-learn, hard-to-govern harness means s_t remains a static prompt or isolated skill, lacks cross-session state, and the executable boundary of $g_t \circ s_t$ is unclear.

To address these challenges, we design and implement OpenSquilla, an agent-native learnable harness. Here the harness is not a thin model-call wrapper but the system-level scaffold that carries task input, model routing, skill orchestration, tool execution, state persistence, permission approval, and evaluation observability: on each subtask t, g_t selects the model and s_t adjusts the prompt, RAG, skills, memory, and tool boundaries. The core design borrows the separation of mechanism and policy from modern operating systems: the core engine keeps only the turn lifecycle, state-machine progression, event streaming, tool boundaries, and finalization semantics, while extensible capabilities—model pools, toolsets, memory stores, message channels, security sandboxes, Meta-Skills, and schedulers—attach through standardized registries, adapters, or services. This architecture improves extensibility and provides clean boundaries for fault isolation, capability evolution, permission control, and end-to-end observability.

From a technical standpoint, the main contributions of OpenSquilla are threefold:

- Singleton routing for cost savings: building on our harness-native agentic-routing framework [35], which formalizes model selection as a step-level routing operator conditioned on the harness state, singleton routing ($k = 1$) selects, by capability matching, the cheapest sufficiently capable model for each execution step, cutting cost substantially with almost no quality loss. The open cold-start implementation, SquillaRouter, combines four text-model tiers, reasoning budget, and prompt policy, performing mixed-feature classification routing locally without extra remote classification calls; prompt-cache continuity, stable/dynamic prompt segmentation, and on-demand skill loading further reduce the token overhead of repeated prefixes and irrelevant skill descriptions.
- Multi-model ensemble routing for quality gains: in the high-accuracy region of the same frontier ($k > 1$), the router assembles complementary proposer sets from a task profile and fuses candidate outputs through a state-aware aggregator; the complementarity regularizer of the framework steers the sets toward decorrelated failure modes, so compositions of cheaper models achieve higher task quality at lower cost than a strong single model, and the system degenerates to singleton routing at routine states.
- System mechanisms of the learnable harness—Meta-Skill orchestration, persistent memory, and security governance: at the orchestration layer, a protocolized mechanism over community and local skills teaches the agent to retrieve, filter, compose, and evolve skills, organizing complex tasks into inspectable task graphs of composition, routing, checkpoints, user clarification, and final synthesis, while the Meta-Skill Creator generates reviewable proposals from user collaboration history so the system gradually fits the user; at the memory layer, a long-term context lifecycle fuses Markdown source files, local full-text and semantic hybrid recall, maximal marginal relevance (MMR) re-ranking, optional temporal decay, Session Flush, and Memory Dream, emphasizing editability, rebuildability, auditability, and rollback rather than accumulating transcripts; at the security layer, Standard / Strict / Locked tiers combine permission approval, a denial ledger, stale-output cache invalidation, path and network constraints, and platform capabilities such as Bubblewrap (Linux) and Seatbelt profile generation (macOS), establishing runtime isolation for tool calls without mandating Docker. The three 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 along multiple dimensions. The point of the table is not to claim absolute superiority everywhere, but to show how OpenSquilla builds a differentiated system design around the harness goals of "agent-native, learnable, low-cost, governable."

2. Agentic Routing

A traditional agent binds the decomposition of Equation (1) to one foundation model chosen offline: every step of the harness calls the same model, and the only lever on the quality–cost trade-off is to migrate the whole agent to a different model. As soon as the pool $\mathcal{M}$ holds many base models, that offline constant turns into a decision the harness can make at every step. Agentic routing is exactly this move: a state-conditioned operator g picks, per execution step, the model—or the set of models—that should act next, and the classical agent reappears as the degenerate router that always returns the same singleton. The framework was introduced in our companion report [35]; this section restates it in the form the rest of this paper builds on,

Table 1 | System orientation of OPENSQUILLA versus comparable agent harnesses

Dimension	OpenSquilla	OpenClaw	Hermes Agent
Architecture	Agent-native learnable microkernel harness: stable execution mechanism separated from pluggable, evolvable policy; workflows accumulate with use	Local-first personal-assistant runtime integrating channels, tools, and session capabilities	Personal agent system centered on long-term memory, skill generation, and multi-backend deployment
Cost control	Intelligent model routing, tiered reasoning depth / prompt policy, prompt-cache continuity, and on-demand skill loading	Centered on model configuration and tool flows; routing is not a core claim	Backend and execution-policy choices, but no public local ML routing loop
Learnable orchestration	Meta-Skill protocol layer distilling collaboration history and multi-skill workflows into reviewable assets	Fixed tool/plugin capabilities; workflow evolution is not a core abstraction	Emphasizes skill generation, but orchestration governance depends on its main-loop design
Memory system	Layered memory, hybrid semantic retrieval, optional temporal decay, and Memory Dream consolidation	File/database-backed local memory and session state	Emphasizes long-term memory and session retrieval; implementation tied to its runtime
Security governance	Three-tier policies with denial ledger, cache invalidation, and platform sandboxes	Tool permissions, runtime environment, and external isolation	Approvals, execution environments, and task policies for high-risk operations
Observability	Diagnostic logs, read-only replay, and session export covering routing, cache, compression, tool, and cost events	Session logs and tool-trace review	Session retrieval, skill traces, and run logs

and describes how OPENSQUILLA deploys it in two user-selectable regimes—a cost-effective mode that fields one best-fit model per step, and a high-accuracy mode that fields several complementary models and fuses their outputs (Figure 2).

2.1. Problem Definition and Objective

Let q denote the user task and $\mathcal{M}$ the candidate pool. At decision step t the harness exposes a state h_t that aggregates far more than the visible prompt: the current observation, raw and compressed context, the status of the control loop, the admissible actions, artifact state, tool history, whether the agent is recovering from a failure, and any verification signals. Query-level routers and cascades have shown that heterogeneous pools improve the economics of LLM serving [6, 11, 22, 26], and recent agentic benchmarks push the decision toward execution traces [41, 48]. The harness-native formulation goes further: instead of classifying a static prompt, the router observes where execution currently stands, and may preserve quality more cheaply,

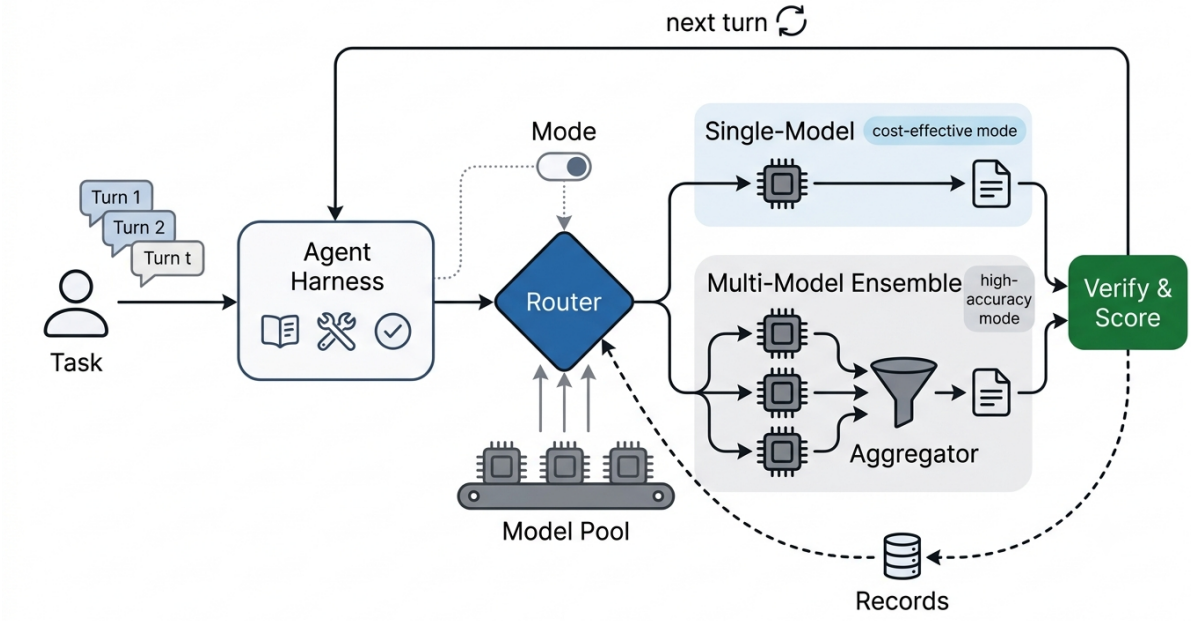

Figure 2 | **The two operating regimes of agentic routing** (reproduced from our companion report [35]). The agent harness supplies the execution state to the router, which selects from a shared model pool. The user-selected mode determines the regime: in the cost-effective mode the router fields a single best-fit model, while in the high-accuracy mode it fields several complementary proposers whose outputs are fused by an aggregator. Verification and cost signals score each decision, and the resulting records feed back to improve the router.

assemble inexpensive specialists that beat a strong single model, or deliberately overspend at states where failure is hard to undo.

Formally, the router is an operator g that, given h_t, selects a model set for the next step and optimizes the Pareto front between task loss and realized cost:

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

with $\ell(\tau) = 1 - R_{\text{task}}(\tau)$ the loss of the completed trajectory, $C(\tau) = \sum_t \sum_{m\in S_t} c(m, h_t)$ its bill, and K a cap on how many models may act at once. Choosing $k = 1$ yields singleton routing (subsection 2.2); $k > 1$ yields multi-model ensemble routing, in which the action additionally carries an aggregation policy (subsection 2.3). Sweeping λ in the scalarized form $\mathbb{E}[\ell(\tau) + \lambda C(\tau)]$ traces the same frontier, and latency can join as a third axis where a deployment needs it.

Two consequences of this formulation deserve emphasis. First, the loss attaches to the trajectory, not the step: agent execution is recoverable, so a locally weak step may be repaired later while a locally cheap step may generate expensive repairs downstream—a view consistent with work that treats reasoning, acting, tool use, memory, and feedback as one interleaved process [30, 31, 36, 40, 42]—and the router must therefore learn from trajectories rather than isolated prompts. Second, fielding several models is not automatically more expensive than fielding one, because a set of low-priced specialists can bill less than a single frontier call. In the notation of Equation (2), g is the per-step realization of g_t and $(\ell(\tau), C(\tau))$ instantiate $(\mathcal{L}_T(a), P(a))$ with s_t held fixed; throughout this report the surrounding harness policies are frozen during routing evaluation, so measured gains are attributable to capability allocation

alone.

Every routing decision is also logged. The trajectory fields

$$(q,\ h_t,\ S_t,\ z_t,\ c_t,\ y) \tag{4}$$

define the schema of harness-native **arena records**: each record stores the action (the fielded set S_t) separately from the labels the environment supplies afterwards—the step output z_t, the cost c_t, the final outcome y, and verification signals. Because of this separation, trajectories are never imitation targets: a poor routing decision, together with its negative outcome, remains a valid training sample. The open router of subsection 2.2 is the cold-start policy that opens this record stream; subsection 2.4 describes how later router generations consume it.

2.2. Singleton Routing

With $k = 1$ the action degenerates to a single choice $S_t = \{m_t\}$. We read this regime as **capability matching**: each state issues a demand for capability; the pool offers supplies at different prices, latencies, context windows, tool-call reliability, and failure modes; and the router buys the cheapest supply that—after adjusting for risk—covers the demand. Two symmetries anchor the intuition: a model whose failures trigger retries, tool repair, or context reconstruction was never actually cheap, and a model whose strengths do not line up with what the current state needs is not effectively strong. The routing state is $x_t = (q, h_t)$; since h_t carries context pressure, artifact state, recovery status, and recent decisions, the demanded capability is a property of the execution position, not of the user query.

Because purely terminal feedback is sparse, the framework attaches a **per-turn reward** $\rho_t = \rho(h_t, u_t, z_t) \in [0, 1]$ that scores the immediate product of the decision—a passing unit test, a well-formed tool call, a satisfied schema or safety constraint, or an LLM-judge score. The router then solves, greedily at each step,

$$m_t = \arg\min_{m \in \mathcal{M}_t} \ \mathbb{E}_{\tau \sim g}[\ \ell(\tau) + \lambda\, C(\tau) - \beta\, \rho_t \mid S_t = \{m\},\ h_t\ ], \tag{5}$$

where $\mathcal{M}_t \subseteq \mathcal{M}$ is the candidate set surviving admission and availability filtering, and $\beta \geq 0$ trades the dense step-level signal against the trajectory-level terms ($\beta = 0$ recovers the terminal objective). Keeping ℓ and C at the trajectory level preserves the recoverable-execution semantics, so the per-turn reward densifies supervision—easing credit assignment—without changing what is optimized.

What a step demands is multi-dimensional and drifts along the trajectory. Bookkeeping, short tool-call formatting, and cacheable responses tolerate lightweight models; ambiguous planning, difficult code edits, recovery after a failed verifier, safety-sensitive decisions, and final artifact generation call for stronger ones. Context pressure can promote long-context reliability above raw reasoning scores, and a recent tool error can promote action-format robustness above leaderboard rank. This dependence on execution position is precisely what separates harness-native singleton routing from query-level routing.

SquillaRouter, the open singleton router shipped with OpenSquilla, implements capability matching as a lightweight cold-start pipeline of four passes: *order admission* diverts clearly trivial, low-risk states to cheap fast paths; *demand construction* scores mixed features of the prompt—length, language, code shape, keywords, and semantic embeddings—and recognizes task signals such as code blocks, attachments, long context, and high-risk phrasing; *risk pricing* adjusts the demand for context pressure, tool failures, verifier outcomes, recovery status, and

action irreversibility; and *capability matching* binds the priced demand to a deployed model or tier via the registry, with a LightGBM ranker over the surviving candidates [16]. Multimodal requests such as images form an independent capability type under a hard constraint: they bind only to models with the matching input capability and are never mixed with text tiers. The full pipeline runs locally in milliseconds, consumes no LLM tokens, and never forwards the user request to a third-party classifier. The LightGBM stage is the cold-start engine, not the essence of the mechanism—the harder learning problem, inferring state-conditioned demand and expected recovery cost from harness-native trajectories, belongs to the router iterations of subsection 2.4.

2.3. Multi-Model Ensemble Routing

With $k > 1$ the routing action returns a proposer set together with a policy for combining candidates,

$$g(\mathcal{M}_t \mid h_t) = (P_t, a_t), \qquad P_t \subseteq \mathcal{M}_t, \quad a_t \in \mathcal{G}, \tag{6}$$

where each proposer in P_t independently generates a candidate and the aggregation policy a_t—drawn from a family $\mathcal{G}$ of judge models, verifier-based selectors, voting rules, and fallbacks—fuses them into one executable result. If the aggregator is itself a model call, the fielded set of Equation (3) becomes $S_t = P_t \cup \{m_t^{\text{agg}}\}$; otherwise $S_t = P_t$. The bill $C(\tau)$ then counts every proposer call plus aggregation overhead, and latency is tracked separately.

Writing $\ell(u \mid h_t)$, $C(u \mid h_t)$, and $\rho(u \mid h_t)$ for the expected loss, cost, and per-turn reward of an action u at state h_t, the ensemble router solves

$$(P_t, a_t) = \arg \min_{P \subseteq \mathcal{M}_t,\ a \in \mathcal{G}} \Big[\ell\big((P,a) \mid h_t\big) + \lambda\, C\big((P,a) \mid h_t\big) - \alpha\, V\big(P \mid h_t\big) - \beta\, \rho\big((P,a) \mid h_t\big) \Big], \tag{7}$$

of which the singleton rule (5) is the special case $u_t = (\{m\}, \varnothing)$. The new ingredient is the complementarity term $V(P \mid h_t)$, which pays proposer sets whose members err in *uncorrelated* ways; we realize it as a diminishing-returns surrogate over historical error decorrelation, state-conditioned disagreement, and verifier feedback, which admits greedy subset construction. In principle the loss ℓ already prefers complementary sets—redundant models sharing training distributions raise cost without lowering loss—but ℓ is noisy, off-policy, and trajectory-level while the choice of P is combinatorial, so the explicit term serves as an inductive bias that keeps the subset search from collapsing onto the logging policy's favorites. The weight α is kept small and annealed toward zero as loss estimates sharpen, so the true frontier is recovered and diversity is never rewarded for its own sake.

Reading Equation (7) at a light cost weight gives the **accuracy-seeking** regime: extra calls are spent exactly where a single-model error is costly and hard to undo—final answer generation, complex code modification, irreversible tool actions, safety-sensitive decisions, long-context synthesis, or low router confidence—with a typical division of labor of one primary proposer, one adversarial checker, and one cheap sanity check. A heavy cost weight gives the **cost-saving** regime: mid- and low-priced models jointly replace an expensive reference within a quality tolerance while billing strictly less; when the composition also beats the reference, it improves quality and cost at once. The implementation is two-stage. Stage one is profile-driven subset selection: a task profile φ_t computed from (q, h_t) summarizes task type, difficulty, required capabilities, risk, verification availability, and cost tolerance, and proposers are scored by fit to the profile, marginal value, verification compatibility, and complementarity with those already chosen. Stage two is state-aware aggregation, $\hat{r}_t = a_t(\{(m, r_m)\}_{m \in P_t}, h_t)$, conditioned on the strongest verification signal the harness can observe—unit tests and static analysis for software tasks, schema and permission constraints for tool tasks, rubric judging and citation checks

for open-ended research. In the cost-saving regime the aggregator itself must be cheap, or it consumes the proposer-side savings. The system degenerates to singleton routing at routine states, so ensembles are fielded only where they pay for themselves; each ensemble step also yields multi-model outcomes on the same decision point, feeding the counterfactual coverage that off-policy learning needs.

2.4. Router-Model Iterations

The deployed router is not one model but a ladder of policies $g^{(0)}, g^{(1)}, g^{(2)}, \ldots$ sharing the objective of Equation (3). The seed $g^{(0)}$ is the four-pass LightGBM pipeline of subsection 2.2—deliberately the cheapest policy able to run at every step, whose job is to open the arena-record stream. A learned generation $g_\theta^{(r)}$ factorizes into a *state encoder* replacing the hand-engineered features, a *supply encoder* that embeds each candidate's model card and observed outcome history—so a newly released model can be scored from its profile without retraining a label space—and a *prediction head* estimating $\hat{\ell}(u \mid h_t)$ and $\hat{C}(u \mid h_t)$ for candidate actions, acted on by exactly the rules of Equations (5) and (7). Training minimizes the frontier objective on the accumulated corpus,

$$\theta^{(r+1)} = \arg\min_\theta \; \widehat{\mathbb{E}}_{\mathcal{D}^{(r)}} \big[\ell(\tau) + \lambda \, C(\tau) \big], \tag{8}$$

where the expectation is corrected off-policy with inverse-propensity or doubly-robust reweighting, so the target is outcome improvement rather than imitation of the logging policy [8]. Coverage is maintained by controlled exploration—uncertainty-triggered escalation and occasional alternative-model replay on sampled states—so that $g^{(r+1)}$ can learn where $g^{(r)}$ under-routed or overpaid. A candidate generation is promoted only when it measurably moves the frontier—lower realized cost at equal task reward, or higher reward under a fixed budget—and otherwise rolls back; agreement with its predecessor is never the goal. Deployment stays two-tier: cheap admission filters settle clearly trivial or clearly hard states, and the heavier learned router runs only where a better decision is worth its overhead. The $g^{(0)}$ seed is open-sourced because it is inspectable, locally trainable, and useful for cold start; later generations depend on the accumulated records and evolving model-supply profiles, and are served as an evolving policy path. In this report, $g^{(0)}$ is the router deployed in the cost-control stack of Section 3 and in the experiments of Section 6.

3. Intelligent Cost Control

3.1. Two-Layer Cost-Control Framework

The real workload of an LLM agent exhibits two kinds of structural heterogeneity: high variance in per-turn difficulty and high repetition across turns. A short confirmation and a cross-file code revision can differ in difficulty by well over an order of magnitude, so a single-tier policy—always strongest or always cheapest—cannot balance quality and sustainable cost [6, 26]. Meanwhile, the system prompt, tool definitions, and stable context of a session are highly repetitive at the token level; naive retransmission makes cost proportional to turn count. OpenSquilla therefore decomposes cost control into two complementary problems: per-turn cost must be tiered by difficulty, and cross-turn repetition must be recognized and amortized. The routing problem of "choosing a model tier by difficulty" is already fully characterized by the agentic routing framework of Section 2—SquillaRouter is its cold-start policy $g^{(0)}$ that starts the routing-record stream, locally running the four passes of admission filtering, demand construction, risk adjustment, and capability matching, and emitting the routed tier, confidence, and decision

metadata; this section does not repeat the routing mechanism itself. From the perspective of Equation (2), this section addresses how to further reduce $P(a)$ once $g_t(\mathcal{M})$ is provided by the router: the decision layer derives the reasoning budget and prompt policy from the routing metadata, while the runtime layer removes repeated tokens and irrelevant context through s_t operations such as caching, prompt segmentation, and skill injection. The system also retains a single-model passthrough mode for reproducing experiments and isolating "model capability" from "framework contribution" in evaluations.

Figure 3 shows the corresponding two-layer architecture. The **decision layer** emits discrete control signals (m_t, r_t, p_t) each turn: the text-model tier $m_t \in \{c0, c1, c2, c3\}$ (four capability tiers from lightweight to flagship; multimodal capabilities such as vision are handled as independent types) is selected by the singleton router of Section 2, while the reasoning budget r_t and prompt policy p_t are derived from the same routing metadata in §3.2. The **runtime layer** is orthogonal to the decision layer and consists of prompt-cache continuity and skill filtering, handling stable-prefix reuse across turns and pruning of injectable metadata, respectively. The two layers meet at the upstream LLM call: the decision layer shrinks the marginal cost of the current turn, while the runtime layer removes what has already been paid for and what need not be paid for from the next turn's context.

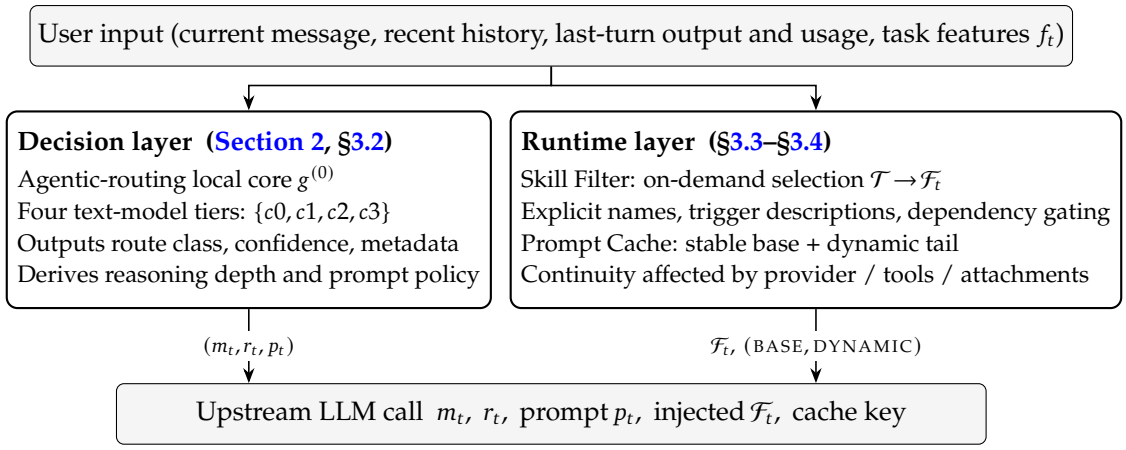

Figure 3 | The OpenSquilla two-layer cost-control framework. The decision layer emits the text-model tier, reasoning budget, and prompt policy each turn; the runtime layer performs skill filtering and prompt-cache continuity maintenance in parallel, producing the injectable skill set $\mathcal{F}_t$ and a stable-base / dynamic-tail prompt structure. The two layers meet at the upstream LLM call, jointly reducing per-turn marginal cost and cross-turn repeated cost.

Table 2 compares the four cost mechanisms of the two-layer framework by scope. The table serves as an index for Section 2 and §3.2–§3.5; readers can return to it to place each mechanism within the overall framework.

3.2. Deriving the Reasoning Budget and Prompt Policy

The tier m_t, selected locally by the singleton router of Section 2, only answers "which tier of model to call." OpenSquilla additionally derives a reasoning budget and a response policy from the same routing metadata, answering "is deeper reasoning needed" and "how complete should the prompt be." This lets the system keep adjusting cost within a fixed model tier: low-difficulty requests can use a lighter reasoning budget and shorter prompts, while debugging, long-context, strict-format, or high-risk operations use a more conservative response policy, with execution

Table 2 | The four cost mechanisms of the two-layer framework. "Fallback" describes the degradation behavior when a mechanism is unavailable.

Mechanism	Layer	Time scale	Observable signals	Fallback
Model-tier routing m_t (Section 2)	Decision / per turn	Local, milliseconds	routed tier, confidence, metadata	Default mid tier (c1)
Reasoning-budget and prompt-policy derivation (r_t, p_t)	Decision / per turn	Local execution	reasoning tier, policy, metadata	Conservative defaults
Prompt caching	Runtime / cross turn	Amortized over the session	cache tokens, cache invalidation, base/tail changes	Priced at cache miss
Skill filtering	Runtime / system level	Per turn, scales with $\lvert\mathcal{T}\rvert$	matched skills, eligibility, inspect / meta runs	Explicit skill name or no injection

risk further adjudicated by the permission and sandbox layers.

Let ψ_r denote the reasoning derivation function and ψ_p the policy derivation function; then

$$r_t = \psi_r(m_t, \pi_t, f_t, H_t), \qquad p_t = \psi_p(m_t, \pi_t, f_t, H_t). \tag{9}$$

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

The value of this design is that it extends cost control from one-dimensional model selection to three-dimensional budget allocation: the tier determines the model's unit price, reasoning determines reasoning tokens and latency, and the policy determines dynamic prompt length and format constraints. Together they form the controllable surface of end-to-end agent cost.

3.3. Cross-Turn Mechanism: Prompt Caching

Beyond per-turn tier selection, agent cost is also driven by repeated prefixes across turns. Prompt caching exploits provider-side KV-cache reuse over a stable prefix; the harness's contribution is to keep that prefix stable. OpenSquilla organizes the prompt 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, recalled history, attachment summaries, and tool results. Let γ_t be the reusable fraction of the stable prefix at turn t and δ the provider's discount factor for cache-read tokens; the input-side cost can be written as

$$\tilde{c}_t^{\text{in}} \;=\; c_m(m_t)\,\big((1-\gamma_t)n_t^{\text{base}} \;+\; \gamma_t\,\delta\,n_t^{\text{base}} \;+\; n_t^{\text{tail}}\big). \tag{10}$$

The formula does not depend on any provider's specific pricing; it only states that when the stable prefix stays stable and the provider supports cache billing, the repeated system prompt and tool definitions need not be re-paid at full price every turn.

In practice, cache continuity is best-effort rather than guaranteed. Model-tier changes, provider switches, tool-definition changes, new attachments entering the context, long-context compression, or dynamic skill-set changes can all reduce the reuse rate. The system therefore records cache hits and invalidations in diagnostic logs, and exposes the ratio of cached tokens to

total tokens in the session-level cost view, so cache benefits can be audited rather than merely claimed at the configuration layer.

3.4. Cross-Turn Mechanism: Skill Filtering

The skill system is OpenSquilla's other lever against prompt bloat. The system ships with, and allows extension of, multi-source skills covering coding, version collaboration, document generation, summarization, and external information access. If every turn injected the full set of skill descriptions into the system prompt, a richer skill library would mean a longer stable prefix and higher cost. OpenSquilla instead loads relevant guidance on demand: the model does not see the full skill set by default; the runtime selects candidates based on task intent, explicit skill names, trigger descriptions, and the installation environment.

Formally, with the full set $\mathcal{T}$ and the injected set $\mathcal{F}_t$ for the current turn,

$$\mathcal{F}_t = \text{Filter}(\mathcal{T}, \text{msg}_t, \text{ctx}_t), \qquad |\mathcal{F}_t| \ll |\mathcal{T}|, \tag{11}$$

where msg_t is the current user message and ctx_t includes the runtime environment, available dependencies, tool permissions, workspace boundaries, and explicit user intent. Filtering applies two kinds of constraints: availability gating—skills with missing dependencies or demo-only status are excluded—and relevance matching, which aligns the user's natural-language goal with skill descriptions, trigger words, and Meta-Skill structure. The system also provides skill-structure inspection so users and developers can understand the step boundaries of a Meta-Skill.

Skill filtering nests with the prompt policy p_t: p_t determines the *shape* of the prompt (compressed / neutral / full), while skill filtering determines the *size* of the injectable content pool. In Equation (2), this on-demand injection is a core component of s_t: it does not change the selected model, but by choosing which capability descriptions the current model sees, it shapes the trade-off between task success and prompt cost. Decoupling the two means that skill-library growth mainly affects the filtering side rather than forcing every turn to pay the token cost of the full description set. The runtime further exposes key decisions through skill visibility, structure inspection, execution traces, and proposal-management interfaces, turning skill injection from implicit prompt magic into inspectable system behavior.

3.5. Failure Modes and Operational Observability

Soft-failure semantics When the predictor core fails to load, routing assets fail validation, or a single classification throws, the router returns the default mid-tier model with a diagnosable reason instead of silently choosing the cheapest tier. This keeps unknown difficulty in the mid-tier safety zone and reduces failure retries and quality collapse caused by mis-downgrades.

Runtime-layer degradation Changes to the stable prefix mid-session (new tools, attachments, or skills entering the context) reduce cache continuity, as do provider or model switches. On the skill side, missing dependencies or failed availability gating keep skills out of the injected set; users can still verify availability through the visibility and structure-inspection interfaces. When the memory or semantic retrieval backend is unavailable, the system retains the keyword / full-text retrieval path so the main turn is never interrupted by an optional component.

Diagnostics and replay OpenSquilla provides unified diagnostics, session export, and read-only replay. Diagnostic logs record model communication, SquillaRouter decisions, prompt-

cache hits and invalidations, context compression, tool-output compression, channel delivery, and cost/latency anomalies; read-only replay reconstructs a readable trace from decision logs without re-executing tools. Cost control is thus not only an online policy but also supports offline review and regression evaluation.

In sum, OpenSquilla's intelligent cost control is a two-layer system in which the decision and runtime layers cooperate orthogonally: the decision layer decides each turn "which tier of model / how much reasoning budget / which response policy," with the model tier supplied by the agentic routing of Section 2; the runtime layer removes the already-paid stable prefix and unneeded skill descriptions from the next turn's context. All key decisions flow into diagnostics, cost statistics, and read-only replay; when subcomponents are unavailable, the system degrades along interpretable paths such as the default mid tier, single-model passthrough, or keyword retrieval. Cost control thereby becomes the engineering realization of the $P(a)$ side of Equation (2): g_t reduces the model unit price through agentic routing, and s_t reduces reasoning and repeated-token cost through budget derivation, cache continuity, and on-demand skill injection, jointly raising Intelligence/Token. The experiments later show the cost–capability trade-offs this design achieves on end-to-end agent benchmarks.

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

After Section 2 and Section 3 answer "which model to call and how much to spend," this section presents the three system mechanisms that support the learnable harness, corresponding to three dimensions of s_t in Equation (2): the Meta-Skill orchestration layer determines how capabilities are organized and evolved, the persistent memory system provides the cross-session state substrate, and the security governance framework draws the executable boundary of $g_t \circ s_t$. Together they ensure the harness improves with use while remaining reviewable, revertible, and controllable.

4.1. Meta-Skill: Skill Orchestration and System Evolution

In traditional agent architectures, a skill encapsulates a single capability—summarization, retrieval, analysis, writing, or tool invocation. As bundled, workspace, and community skills proliferate, the question shifts from "does the system have a capability" to "how does the agent retrieve, filter, compose, and evolve those capabilities": complex tasks that span stages, require conditional branching, and involve multi-turn interaction cannot be reliably reused through ad-hoc planning by a single skill or model. OPENSQUILLA therefore defines a Meta-Skill as a declarative orchestration mechanism above skills—a workflow abstraction that performs composition, routing, dependency scheduling, failure substitution, user clarification, and final synthesis over multiple skills. Composition decomposes a complex task into a task graph with clear semantics, dependencies, and data flow; for example, report generation is organized into requirement understanding, structure planning, per-chapter generation, and consistency synthesis, with different skills serving each stage. Routing makes execution branches explicit: entry routing selects the Meta-Skill from the user request, in-flow routing chooses subsequent branches from intermediate results, capability routing picks the skill suited to the current context among candidates, and recovery routing resumes from the pause point after the user supplies missing information. The key principle: the model understands intent, and the runtime constrains execution—the overall task structure, step order, branch paths, and output policy are controlled by the declarative workflow, embedding the flexibility of large models in a verifiable,

reusable, observable engineering frame. In the language of Equation (2), skills extend the capability set of s_t, and Meta-Skills make the composition of s_t task-type-aware.

The Meta-Skill Creator further answers "how does the system learn new ways of organizing capabilities from collaboration history." Capability growth in traditional skill systems relies on manual development and struggles to cover the long tail of real scenarios, while in practice many complex tasks repeatedly form similar multi-skill collaboration patterns; without distillation, each occurrence requires fresh model planning. The Creator observes co-occurrence, ordering, and dependency relations among skills in historical execution traces, and evolves in four stages: **observation**, recording user goals, multi-skill call chains, intermediate results, branch choices, and output quality; **abstraction**, distilling stable composition, routing, input requirements, and output policies from repeated traces; **validation**, checking a candidate workflow's structural integrity, trigger boundaries, routing stability, output quality, and risk; and **consolidation**, saving validated candidates as Meta-Skill proposals that enter the skill ecosystem after review or conservative activation. Crucially, the Creator produces proposals rather than live capabilities: once a workflow is frozen it shapes how the system will handle similar tasks, and without governance, accidental paths, overly broad triggers, or high-risk operations could be consolidated into long-term behavior; candidates must therefore pass structural checks, positive/negative example validation, risk assessment, and, where needed, human review. Skills provide atomic capabilities, Meta-Skills provide orchestration, and the Creator provides the generation and evolution of orchestration patterns; together they evolve s_t from fixed prompt / RAG operations into a learnable orchestration policy that adapts to the user's scenarios—the core meaning of a "learnable harness."

4.2. Persistent Memory System

As agents move from single-shot Q&A to long-horizon collaboration, user preferences, project facts, past decisions, and reusable output formats must remain valid across sessions, while full transcripts, tool returns, and transient instructions must not enter future context indiscriminately—otherwise noise accumulates, stale facts linger, and prompt injection persists. OPENSQUILLA therefore designs memory as an independent long-term state layer rather than "a vector store plus full session logs," with two boundaries: memory is separated from skills—skills describe how the agent accomplishes tasks, memory describes what the agent should keep knowing; and long-term memory is separated from audit records—reusable facts are curated, while full sessions and raw turn records are retained as audit or derived-retrieval material that does not pollute ordinary recall. In Equation (2), the memory system is the cross-session state substrate of s_t: it lets harness operations perceive historical facts and successful paths instead of reorganizing every task from a blank context.

Architecturally, persistent memory is a four-layer lifecycle (Table 3). The source layer restricts what can become memory: editable Markdown long-term memory files, virtual retrieval documents derived from persisted sessions, and private turn captures excluded from ordinary indexing by default. The index layer, built on a local relational database, deduplicates memory by content hash, splits it into line-numbered fragments, and maintains a full-text index and an optional vector index; changes to the embedding backend, model, or dimensionality trigger a safe rebuild so vectors from different embedding spaces are never mixed. The recall layer synchronizes the index before each query and then performs hybrid recall and re-ranking. The consolidation-and-governance layer manages memory evolution through Session Flush, Memory Dream, and cleanup policies. Semantic representation is local-first: quantized embedding inference runs on the local machine by default, and long-term memory is not sent to third-party

services; when the local backend is unavailable, the system degrades to pure full-text retrieval.

Table 3 | The four layers of the OPENSQUILLA persistent memory system

Layer	Role	Responsibilities	Degradation
Source	Editable memory, session fragments, private transcripts	Separate long-term facts, session evidence, and raw audit records; keep raw transcripts out of ordinary recall.	Files and session records are kept but not indexed for ordinary retrieval.
Index	Local full-text and semantic indexes	Split sources into retrievable fragments, maintain full-text and optional vector indexes, embedding cache, and index consistency.	Falls back to full-text retrieval when vectors are unavailable.
Recall	Hybrid retrieval and re-ranking	Sync the index before each query; fuse semantic and lexical recall with temporal decay, MMR, and a lexical floor.	Keyword recall is retained.
Consolidation & governance	Flush, Dream, and cleanup policies	Distill long sessions into candidate memory, promote long-term facts through evidence gating, and run TTL, redaction, rollback, and audit.	Dry-run / archive-only, or skip writes.

The recall layer aims to return the most relevant, least redundant fragments within a bounded prompt budget, fusing semantic and lexical recall:

$$s_{\text{hyb}}(q,d) = w_v\, s_{\text{vec}}(q,d) + w_l\, s_{\text{lex}}(q,d), \qquad w_v = 0.7,\ w_l = 0.3, \tag{12}$$

where the vector channel captures semantically similar facts and the lexical channel guarantees exact hits on proper nouns, paths, dates, error codes, and project terminology; when the vector backend is unavailable, w_v=0, w_l=1 recovers pure full-text retrieval. Three stabilizers sit on top: a lexical floor admits strong full-text matches into the final candidates so semantic scores cannot suppress explicit keyword hits; dated memories decay exponentially with a 30-day half-life, while the root-level `MEMORY.md` and non-dated long-term memories are treated as evergreen, letting transient tasks sink naturally while stable facts remain visible; and MMR re-ranking (relevance–diversity weight $\lambda_{\text{MMR}} = 0.7$, with redundancy measured by inter-fragment Jaccard similarity) constrains result diversity so near-duplicates do not crowd the recall budget.

Consolidation has two levels. Session Flush distills long sessions into candidate memories—facts, preferences, decisions, procedures, to-dos—before session reset, archival, or context compression, saving them through a restricted write context; when LLM extraction times out or the result cannot be parsed safely, it degrades to archiving raw material only—the system would rather keep auditable evidence than promote low-confidence summaries to long-term facts. Memory Dream is periodic offline consolidation: each run scans only incremental candidates, skips private directories and high-risk text, and gates promotion by a weighted evidence score over occurrence frequency, positive/negative signal balance, source credibility, and cross-date consolidation; promotion never lets the model rewrite the memory store wholesale, but generates constrained incremental edits (insert / merge / skip), backs up the root index before modification, and writes a replayable receipt for every run. On the write side, memory writing is treated as long-term context governance: targets must lie within controlled memory source paths; redaction, injection scanning, and capacity checks run before writing, blocking secrets, typical prompt injection, and invisible control characters; writes use snapshot-and-rollback semantics; and TTL and capacity cleanup never touch `MEMORY.md`, bootstrap files, or private directories, with a per-sweep deletion cap. Compared with naive "full history + vector recall,"

this design emphasizes source boundaries, index rebuildability, recall interpretability, and auditable consolidation, letting s_t accumulate and reuse experience across sessions without being hijacked by historical noise, stale information, or malicious content.

4.3. Security Governance

The security risk of tool-using agents goes beyond "non-compliant answers" in text: a seemingly ordinary tool call can delete workspace files, overwrite configuration, leak secrets, access network resources, or write malicious instructions into long-term memory and candidate workflows that re-trigger in later sessions. Drawing on the attack surfaces examined in existing agent-security evaluations, we group attacks into six families (the taxonomy and its abbreviations are defined for this report): direct and indirect prompt injection (DPI / IPI), tool-return injection (TRI), memory poisoning and memory extraction (MPI / MEX), and ambiguous delegation induction (ADI)—the first three concern input sources, the middle two involve persistent state, and the last stems from ambiguity between user intent and execution permissions. This taxonomy implies that agent security cannot rely on prompt-level policy alone; it requires least privilege, system-level isolation, an audit loop, and state cleanup at runtime. OPENSQUILLA follows four principles: **least privilege by default**—each tool call receives only the file, network, and process capabilities the current task requires; **separation of policy and execution**—the model may propose an action plan, but whether it executes is decided by the runtime from risk hints, path scope, and approval status; **auditable denial**—all denied actions accumulate by action fingerprint into a denial ledger; and **non-reusable state**—denied or stale tool outputs cannot be referenced by later steps. From the perspective of Equation (2), security governance is a domain restriction on the executable (g_t, s_t) pairs: the more powerful s_t becomes, the more precise its runtime authorization boundary must be.

The runtime selects a security tier dynamically from the operation's source, scope, and potential impact, judged along five dimensions: trust of source, network need, writes outside the workspace, trust-boundary crossing, and high-impact operations; Table 4 lists the three tiers and their runtime actions. This tiering converts "whether to trust the model's judgment" into "which system capabilities the runtime grants": instructions from external webpages, emails, or tool returns do not automatically gain file-write or network permissions even when semantically plausible; actions involving secrets, configuration, deletion, overwriting, or cross-directory writes are escalated to STRICT or LOCKED and constrained jointly by the sandbox and approvals. In other words, a routed $g_t \circ s_t$ pair enters the real execution path only when the permission, security, and budget predicates are all satisfied.

Table 4 | Three-tier security policy and runtime actions

Tier	Trigger conditions	Runtime actions
STANDARD	Trusted source, reads/writes within the workspace, no network or high-impact operation	Execute directly; record tool summary and output fingerprint
STRICT	Untrusted input, cross-workspace reads, writes to host paths, instructions embedded in tool returns	Enable sandbox isolation, restrict mounts and output, trigger user approval when necessary
LOCKED	Trust-boundary crossing, network access, deletion/overwrite of critical files, credential-related or irreversible operations	Mandatory human review with the strictest sandbox profile; autonomous execution pauses after repeated denials

On systems that support namespaces or platform sandboxes (Bubblewrap on Linux; Seatbelt profile generation on macOS), system-level isolation adopts deny-by-default: only the necessary runtime and workspace are exposed, host directories enter the sandbox read-only, network and process capabilities are off by default, tool output is bounded by size and timeout, and path mounts pass normalization checks against cross-directory access and traversal. Isolation capability differs across platforms, so the sandbox state is surfaced in runtime diagnostics: when full system-level isolation is unavailable, the system degrades to a combined defense of permission profiles, workspace constraints, sensitive-path protection, interactive approvals, the denial ledger, and tool-output invalidation—treated as controlled degradation, not full isolation. On the audit and state-cleanup side, the denial ledger pauses autonomous execution after consecutive denials reach a threshold, preventing the agent from probing policy by rewriting commands, paths, or parameters; the stale-output cache immediately invalidates cached tool outputs with the same fingerprint once a denial is issued, closing the bypass of "obtain sensitive output in a low-risk context, then read the last output"; on the input side, skill descriptions, tool returns, and external content are uniformly enveloped and escaped, and external or model-generated content in long-term memory is treated as low-trust data so memory entries are never promoted into system-level rules.

For security evaluation, results should not be reported as a single blanket attack success rate (ASR): semantic compliance with an attack, evidence-backed real harm in logs and file traces, and actual state changes in an executable sandbox are three different endpoints and should be reported separately with explicit denominators and raw counts; hand-crafted hard-biased attack sets should be interpreted only as relative performance under stress. Accordingly, OPENSQUILLA's security claims are scoped to runtime protection and auditable execution boundaries: the three-tier policy, system-level sandboxes, the denial ledger, and stale-output invalidation reduce the probability of real harm from tool-using agents, while prompt-only defenses and complex policy bundles are treated as diagnostic components rather than universally superior production defenses.

5. System Architecture

The OPENSQUILLA architecture splits the AI agent harness into a clean microkernel and user-space capabilities. The microkernel owns the stable turn lifecycle, an explicit state machine, concurrency control, and event streaming; user-space capabilities own the evolvable policies—model pools, tools, memory, skills, Meta-Skills, channels, schedulers, and sandboxes. This division is the systems-level realization of Agent = base-model pool $\mathcal{M}$ + Harness: the microkernel provides the stable execution skeleton, while $g_t(\mathcal{M})$ and s_t are injected into the turn lifecycle as replaceable, learnable, governable policies.

The advantages are threefold. First, capability extension never breaks the core execution path: new model providers, message channels, tools, skills, Meta-Skills, or memory policies attach through boundaries. Second, faults are local: model failures, tool denials, channel anomalies, memory degradation, or rejected skill proposals are handled within their respective boundaries. Third, observability improves: one complex agent behavior decomposes into recorded stages—input, capability resolution, prompt assembly, context governance, orchestration, streaming execution, and finalization. OPENSQUILLA thus forms an agent-native microkernel harness suited to long-horizon, multi-channel, high-risk tool-calling, and continuously evolving skill scenarios.

5.1. Design Principle: Separating Mechanism from Policy

The architecture adopts the microkernel idea. "Microkernel" does not mean few capabilities; it means the harness core keeps only the minimal, stable execution mechanisms: session-level task orchestration, state-machine progression, streaming event dispatch, tool-call boundaries, error-termination semantics, and lifecycle management. Policy capabilities—model selection, toolsets, memory retrieval, skill injection, Meta-Skill orchestration, channel I/O, scheduled tasks, and sandbox isolation—attach outside the core through protocolized interfaces, registries, adapters, or services.

This continues the operating-system principle of separating mechanism from policy: the core runtime does not hard-code any model provider, channel protocol, or memory policy; it provides stable execution ordering and fault boundaries around which external capabilities evolve independently. An agent turn can be abstracted as the composition

$$\mathcal{T} = F_{\text{finalize}} \circ F_{\text{stream}} \circ F_{\text{orchestrate}} \circ F_{\text{context}} \circ F_{\text{prompt}} \circ F_{\text{capability}} \circ F_{\text{input}}. \tag{13}$$

Input normalization, capability resolution, prompt assembly, context governance, orchestration, streaming execution, and final persistence form the stable kernel path; model routing, skill filtering, Meta-Skill orchestration, memory recall, security policy, and cost accounting are injected into the corresponding stages as replaceable policies. Concretely, $F_{\text{capability}}$ and F_{prompt} carry the selection of $g_t(\mathcal{M})$ and the assembly of s_t, $F_{\text{orchestrate}}$ carries the Meta-Skill structuring of s_t, and F_{finalize} consolidates the turn's trajectory as input to future s_t. This decomposition elevates "a model generating text" into "a controllable, observable, recoverable, evolvable task-execution process."

5.2. Staged Turn Execution

Many early agent frameworks concentrate input handling, model selection, prompt concatenation, tool calls, history writes, and error handling in one long main loop. This suits rapid prototyping but hurts production systems in two ways: any capability extension can perturb the whole execution path, and boundaries such as errors, cancellation, compression, tool calls, and persistence are hard to test independently.

OpenSquilla splits a turn into stable stages. Each stage receives well-defined inputs, produces well-defined outputs, and accesses external dependencies through narrow interfaces. Table 5 gives the abstract view.

The benefit of staging is that extensibility rests on clear boundaries rather than global variables or implicit call order. Model routing mainly affects prompt assembly and provider selection; the memory system participates in prompt assembly and finalization; security policy acts on tool dispatch and sandbox boundaries; cost accounting lands uniformly at finalization. Capabilities exchange only the necessary information through stage outputs, reducing cross-module coupling.

5.3. Explicit State Machine and Streaming Event Model

Within a turn, OpenSquilla does not treat a model call as one blocking RPC; it models it as an explicit state machine. A typical lifecycle is

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

The state machine supports streaming events including model text deltas, tool-call start, tool-argument deltas, tool-call end, tool results, heartbeats, completion, and errors.

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 entries into runtime messages, semantic input, attachments, and source metadata.
Capability resolution	What can this turn use	Resolve model providers, 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 history enters this turn	Load session history and compress as needed so history, attachments, and tool results respect the context budget.
Orchestration	How the task organizes capabilities	Decide workflow steps and recovery paths from Meta-Skills, skill matching, user clarification, and checkpoint state.
Streaming execution	How model and tools interleave	Drive streaming model output, tool calls, tool-result reinjection, routing replay, and mid-flight error handling.
Finalization	How the turn is persisted	Persist assistant output, tool fragments, errors, memory capture, and token and cost statistics.

The core value of this design is converting the uncertainty of agent execution into observable state. A long task may go through many tool calls; a provider may fail mid-stream; the user may cancel; a tool may return oversized results; output may trigger context compression. If these events hide inside synchronous calls or exception stacks, stable recovery is hard; an explicit event model lets the runtime apply per-event policies—retry, degrade, compress, persist partial results, or emit a terminal error.

The OPENSQUILLA agent kernel is therefore not a bare ReAct loop but a production-oriented, event-driven state machine: it retains the reasoning–acting–observation interaction paradigm while bringing tool execution, streaming output, budget control, cancellation recovery, and error auditing into one runtime.

5.4. A Unified Tool-Dispatch Boundary

Tool calling is the part of an agent system most prone to losing control: a tool name and arguments produced by the model may correspond to file writes, command execution, network access, message sending, or external API calls. A key design decision in OPENSQUILLA is that tool functions are never exposed directly to the model; a unified dispatch boundary sits between model and tools.

The boundary has four steps. First, the tool name must pass registry lookup; unregistered tools never execute. Second, the call enters a policy chain that combines call source, permission context, tool visibility, budget, and safety rules to decide whether execution is allowed. Third, actual execution happens in a request-scoped context so the tool can perceive the current session, agent, workspace, and call source. Fourth, tool results enter the finalization layer uniformly, annotated with execution status, truncation status, artifact metadata, and error envelopes.

In set notation, the tools actually executable this turn are not the global toolset $\mathcal{U}$ but a context-filtered subset:

$$\mathcal{U}_t = \{u \in \mathcal{U} \mid V(u, c_t) = 1 \wedge P(u, a_t, c_t) = 1\}, \tag{14}$$

where V denotes visibility constraints, P denotes permission, safety, and budget constraints, c_t is the turn context, and a_t is the action the model proposes. Separating "what the model wants to call" from "what the runtime allows" is the foundation on which the security sandbox, permission control, and cost budgeting take effect.

5.5. Cross-Entry Consistency and Pluggable Capability Boundaries

Extensibility comes mainly from three boundaries: model providers, external channels, and capability plugins.

Model providers attach through a unified streaming interface. A provider only needs to translate its API into text-delta, tool-call, completion, and error events; the core state machine never perceives protocol differences. Model routing, fallback, and cost accounting therefore run on top of the provider abstraction.

External channels attach through a unified message model. Channel adapters convert platform-specific entries into standard input messages and convert system output back into platform messages. The harness core handles only normalized session identifiers, agent identifiers, senders, attachments, and source information. Changes in channel semantics are fixed in adapters and input normalization, never by rewriting the agent kernel.

Skills, Meta-Skills, and memory attach as prompt, orchestration, and context capabilities. Skills provide reusable task knowledge, Meta-Skills provide inspectable workflow organization, and memory provides cross-session state; none of them changes the core state machine—they inject or consolidate information at the prompt-assembly, orchestration, and finalization stages. Long-term capability growth thus does not linearly inflate kernel complexity, and the harness keeps learning new task-organization patterns within governance boundaries. As s_t evolves from static configuration into a learning policy driven by the Meta-Skill Creator, Memory Dream, and diagnostic replay, the microkernel boundary keeps that evolution reviewable, replayable, and governable.

5.6. Concurrency Model: Serial within a Session, Parallel across Sessions

A production agent framework must handle multiple entries and long tasks at once. Global serialization lets one slow task throttle the system; full parallelism lets multiple turns of the same session interleave writes to history, memory, and cost statistics, corrupting state. OPENSQUILLA adopts "serial within a session, parallel across sessions."

With s a session and t_i^s its i-th turn, the system maintains

$$t_i^s \prec t_{i+1}^s, \qquad t_i^{s_a} \parallel t_j^{s_b} \quad (s_a \neq s_b), \tag{15}$$

i.e., order within a session and parallelism across sessions. The runtime distinguishes a long execution lock from short write locks: the former protects a session's turn lifecycle, the latter only critical writes to the transcript and session state. Streaming output, tool execution, and waits on external responses thus never hold state write locks for long, reducing deadlock and head-of-line blocking.

The model also serves channel-based agents: message channels can fire multiple inputs in a short window, and scheduled tasks and sub-agents may run concurrently. Queue caps, cancellation semantics, timeout semantics, and lifecycle events keep backpressure controlled under high-concurrency entry instead of letting tasks pile up without bound.

5.7. Lifecycle Governance and Fault Isolation

The value of a microkernel is not only easy extension but fault isolation. At startup, the composition root constructs configuration, session storage, model selection, tool registries, memory management, skill loading, scheduling, search, and channel management; at shutdown, background producers stop first in dependency order, then memory, sessions, and storage close. This ordering avoids races where background tasks still write while the underlying database or memory index has closed.

Fault isolation follows the same idea: an unavailable model provider returns a stable terminal event; an over-privileged tool call returns a structured denial; a channel that fails to start does not block other channels; when the memory vector backend is unavailable the system falls back to full-text retrieval; a skill that fails to load never corrupts the main state machine. Table 6 summarizes these boundaries.

Table 6 | Main isolation boundaries in the OPENSQUILLA architecture

Boundary	Design goal	Degradation
Model provider	Isolate provider differences, errors, and fallback policies.	Stable error when no provider is available; policy-driven fallback.
Tool	Isolate model intent from real system actions.	Structured denial envelope when over-privileged or unavailable.
Memory	Isolate long-term state from current-turn execution.	Full-text recall retained when vectors are unavailable.
Channel	Isolate external message platforms from each other.	One channel's failure does not affect web, CLI, or other channels.
Scheduler	Isolate background tasks from interactive sessions.	Failed tasks record results without harming the main service.
Sandbox	Isolate tool execution from host resources.	Explicit degradation with recorded risk when backends are unavailable.

In practice, malformed or invalid tool-call histories should be intercepted before the model-provider boundary rather than after the remote API returns an unrecoverable error. Likewise, semantic repairs such as workflow handoff, Meta-Skill clarification, and channel reply context belong in input normalization, capability orchestration, and adapters—never inside the core state machine.

5.8. Engineering Observability

The architecture pursues observability, not just modularization. A complex agent behavior is decomposed into recorded stages—input, capability resolution, prompt assembly, context governance, streaming execution, and finalization. Final events carry not only text output but tool fragments, artifacts, error types, model, provider, cached tokens, cost provenance, and memory-related metadata, so the system can reconstruct the key decisions of a turn rather than only its final natural-language answer.

Observability matters especially for a microkernel: with model routing, tool policy, memory

recall, skill filtering, and sandbox isolation distributed across boundaries, a unified trace and decision log is the only way to tell whether a failure came from model capability, tool denial, context overflow, memory mismatch, or channel adaptation. Staged records turn complex agent behavior into an engineering object that can be located, replayed, and regression-tested.

6. Experiments

To validate the end-to-end behavior of OpenSquilla as an agent-native learnable harness on real tasks, we evaluate on widely used benchmarks against representative agent frameworks / harnesses. The design mirrors Equation (2): single-model direct runs in external harnesses provide baselines approximating a fixed g_t; OpenSquilla forced single-model runs isolate the contribution of the microkernel and s_t; and OpenSquilla multi-tier routing measures how the joint optimization of g_t and s_t moves the cost–capability Pareto frontier. In addition, the dedicated evaluations of the agentic routing framework itself—singleton ($k = 1$) and multi-model ensemble ($k > 1$) routing—appear in §6.4 and §6.5, following the protocols of our companion report [35].

6.1. PinchBench

PinchBench[1] is a benchmark for the real-task capability of code-oriented agents, emphasizing end-to-end practical outcomes over synthetic isolated abilities. We use a fixed version of the public task set and grader; the 25 tasks cover six typical agent workflows—file operations, data processing, web retrieval, creative output, tool use, and memory recall—while probing tool-call accuracy, multi-step reasoning chains, and ambiguity handling. Each task's grader combines automated checks and/or LLM-as-a-judge, producing a score in $[0, 1]$; the reported total is the cross-task mean times 100.

To separate model capability from framework contribution, we compare three agent frameworks under a unified provider pricing scheme: OpenClaw, Hermes Agent, and OpenSquilla. The first two are external-harness baselines running single models directly, covering a flagship and a cost-effective model. OpenSquilla reports both forced single-model runs (the same model through the full framework with routing disabled, measuring the microkernel's effect relative to direct runs) and multi-tier routing (two cost regimes: flagship-backstopped and low-cost-first). Token usage and cost accumulate per call under the same price list.

Table 7 summarizes total scores and costs. Under the same flagship model Claude Opus-4.7 (hereafter Opus-4.7), OpenSquilla achieves the highest score of 93.85, about +1.3pp and +1.2pp over OpenClaw (92.55) and Hermes Agent (92.65), while cost drops from \$6.23 / \$6.66 to \$4.84—the microkernel runtime plus cache / skill s_t operations already compress cost without sacrificing quality. Multi-tier routing yields a sharper cost frontier: the Opus-4.7-backstopped tier combination scores 92.51, nearly matching the Opus direct run, at \$0.69; the low-cost combination topped by GLM-5.1 still reaches 90.48 at \$0.13. In terms of Equation (2), $P(a)$ drops dramatically while $\mathcal{L}_T(a)$ stays essentially stable, validating the joint value of routing g_t and cache / skill s_t operations.

Beyond the harness-level comparison, we also probed the singleton router itself on PinchBench, under a separate protocol that is not directly comparable to the rows above: PinchBench version 1.2.1, an Opus-4.8-generation pool (Opus-4.8, GLM-5.2, DeepSeek-V4 Flash), and costs averaged per task. There, the routed policy held 99.77% of the fixed Opus-4.8 baseline's quality

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

Table 7 | PinchBench 25-task comparison of OpenClaw, OpenSquilla, and Hermes Agent across models (pools). The OpenRouter Auto row is a query-level routing baseline measured under a separate routing-specific protocol (PinchBench 1.2.1; its $0.1204 average per-task cost is scaled to a 25-task total) and is not directly comparable to the other rows.

Framework	Model (pool)	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

(93.14 vs 93.35) while spending $0.0204 per task against $0.2224—roughly a 10.9× reduction—and OpenSquilla with Opus-4.8 fixed reached 94.33 at $0.1649. Against OpenRouter Auto, a query-level routing service run in the OpenClaw harness (88.10 at $0.1204 per task; scaled total in Table 7), the routed policy was 5.04 points better at roughly 5.9× lower cost: query-only routing does not see the execution state that step-level routing exploits.

6.2. ClawMark

ClawMark[2] is a 100-task evaluation suite for domain-specific agent tasks spanning 13 real business domains, including clinical assistance, content operations, e-commerce, EDA, executive assistance, HR, insurance, investment analysis, news, legal assistance, product management, real estate, and research assistance. Unlike benchmarks limited to Q&A or code repair, ClawMark stresses multi-step information processing, table/document reading, visual-material understanding, structured output, and business-constraint compliance. Each task's grader outputs a score in $[0, 1]$; we count tasks scoring at least 0.5 as completed.

To separate model capability from framework contribution, we compare OpenClaw and OpenSquilla under a unified provider pricing scheme: OpenClaw runs GLM-5.1 directly as the baseline; OpenSquilla reports both a forced single-model run (the same model through the full framework with routing disabled) and multi-tier routing (a cost-side tier combination). To keep the image/PDF/screenshot entry consistent across systems, all configurations use the same manual visual-parsing procedure: visual material is first converted into text observations before being handed to the agent, and the vision model is not expanded as a routing tier in the tables.

Table 8 summarizes total scores and average per-task cost. Overall, OpenSquilla with forced single-model GLM-5.1 achieves 77.0, leading all configurations and improving +5.8pp over OpenClaw's GLM-5.1 direct run (71.2); the routing configuration sits closer to the cost-side Pareto point, reaching 70.6—close to the GLM-5.1 direct run—while cutting per-task cost from $0.62 to $0.39. On industry-scenario tasks, then, s_t under a fixed model raises completion quality, and enabling g_t moves the system along the cost side, yielding a more controllable $\mathcal{L}_T(a)$–$P(a)$ trade-off.

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

Table 8 | ClawMark 100-task comparison of OpenClaw and OpenSquilla across models (pools). Scores are on a 100-point scale (×100); costs are average billed cost per task.

Framework	Model (pool)	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 agent harnesses, containing 350 repair tasks from real GitHub issues across 8 programming languages and 43 repositories. Its protocol fixes the task set, Docker containers, prompt templates, maximum turns, and timeout caps, leaving the harness implementation as the only controlled variable, thereby separating harness-level differences from LLM capability and task difficulty. All harnesses share the same task prompt, evaluation image, base commit, working directory, maximum interaction turns, wall-clock timeout, and single-execution constraint; candidate patches are staged and extracted uniformly by the benchmark runner and judged by the evaluator as Resolved / Unresolved / Empty Patch. Resolve rate is the fraction of Resolved instances out of 350, and cost is average billed cost per task (converted from the official leaderboard's Cost/350).

Under this unified protocol we compare three system classes to separate model capability from framework contribution: (i) **OpenClaw single-model direct runs**—Opus-4.7, GLM-5.2, GLM-5.1, and Qwen3.7-Max each complete all 350 tasks independently as capability baselines; (ii) **OpenSquilla forced single-model**—the same model through the full OpenSquilla framework with routing disabled, at two capability tiers (GLM-5.2 and GLM-5.1 [43]), measuring the pure framework effect; (iii) **OpenSquilla multi-tier routing**—the three-tier pool {DeepSeek-V4 Flash, GLM-5.1, GLM-5.2}, dynamically tiered per task by the local router of Section 2, measuring the effect of low-tier delegation and top-tier backstopping on the cost–resolve-rate frontier.

Table 9 summarizes resolve rates and average per-task cost. Overall, OpenSquilla with forced single-model GLM-5.2 achieves the highest resolve rate of 278/350 (79.4%), +5.1pp over OpenClaw's GLM-5.2 direct run (260/350, 74.3%), and surpasses OpenClaw's strongest single model Opus-4.7 (77.1%) at about 31% of its cost ($0.95 vs $3.09); GLM-5.1 also gains +1.4pp inside the framework, showing that OpenSquilla's tool boundaries and session governance reliably amplify code-repair capability without changing the model. Multi-tier routing yields a superior cost-side Pareto point: 259/350 (74.0%), essentially matching OpenClaw's GLM-5.2 direct run, at half its cost ($0.44 vs $0.87). The routing distribution shows 207 tasks (59%) delegated to the inexpensive DeepSeek-V4 Flash (143 resolved, 69.1%), 139 escalated to the top tier GLM-5.2 (113 resolved, 81.3%), and 4 to the mid tier GLM-5.1—the router reserves most of the budget for problems that genuinely need top-tier capability. In terms of Equation (2), s_t under a fixed model directly lowers $\mathcal{L}_T(a)$ on mid/high-tier models, while g_t halves $P(a)$ with almost no loss in resolve rate, jointly extending the cost–capability frontier of code repair. Figure 4 shows the corresponding cost–resolve-rate view: neither OpenSquilla operating point is dominated by any OpenClaw baseline—forced single-model GLM-5.2 holds the top-quality point, and multi-tier routing anchors the leftmost, lowest-cost end of the frontier.

[3] https://claw-swe-bench.github.io/

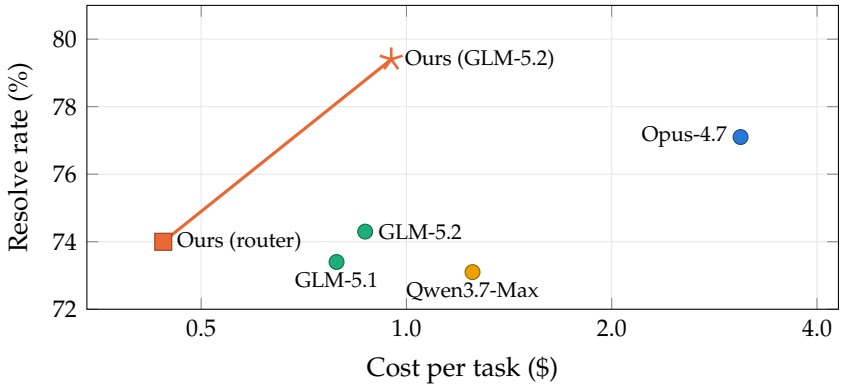

Figure 4 | Cost–resolve-rate view on ClawSWEBench. Colored dots are OpenClaw single-model baselines, colored by model family (blue Opus, teal GLM, yellow Qwen); orange markers joined by a line are the two OPENSQUILLA operating points—the star is forced single-model GLM-5.2 (top-quality point) and the square is multi-tier routing (lowest-cost point). Cost is on a log scale.

Table 9 | ClawSWEBench 350-task comparison of OpenClaw and OPENSQUILLA across models (pools). Costs are average billed cost per task (converted from the official Cost/350); the Opus-4.7 resolve rate 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 of 100 complex research tasks, graded along five dimensions: factual accuracy, completeness, objectivity, presentation quality, and citation quality. Its long execution trajectories—interleaving evidence retrieval, analytical synthesis, and citation generation—stress both the harness's context governance and its model-capability allocation, so DRACO reflects an agent's end-to-end behavior on long-horizon research work better than typical Q&A or code benchmarks.

We keep the comparison logic of the previous three subsections, separating model capability from framework contribution under a unified task set and scoring protocol. OpenClaw runs a fixed Opus-4.8 single model directly as the external-harness baseline; OPENSQUILLA reports both a forced single-model run (the same Opus-4.8 through the full framework with routing disabled) and multi-tier routing over {DeepSeek-V4 Pro, GLM-5.2, Opus-4.8}, tiered step-by-step by the local router of Section 2. Costs are average billed cost per task.

Table 10 summarizes scores and costs over the 100 tasks. The forced single-model config-

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

uration edges past OpenClaw's direct run (52.36 vs 52.13) while nearly halving cost ($0.6559 vs $1.1420), so the framework's tool boundaries and context governance already contribute positively on deep-research work. Turning routing on keeps 99.94% of that quality (52.33 vs 52.36) and cuts cost to $0.3729—another 43.15% below the forced configuration and 67.3% below the OpenClaw direct run cumulatively. The token column is the interesting part: the routed run consumes slightly *more* tokens than the fixed one, so the savings cannot come from shorter prompts; what changes is the price composition of the calls, with the affordable segments of each trajectory sunk to cheaper models. Together with the ClawSWEBench routing distribution (Table 9), this supports the capability-matching reading of subsection 2.2: capability can be allocated state-by-state inside the harness instead of treating every state as needing the strongest model.

Table 10 | DRACO 100-task comparison of OpenClaw and OpenSquilla across models (pools). Scores are cross-task means; costs are average billed cost per task; tokens are input plus output in thousands; the routing threshold is 0.95.

Framework	Model (pool)	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 Ensemble Routing Experiments

The previous subsections compare OpenSquilla with external frameworks from a holistic harness perspective; this subsection is the dedicated evaluation of the multi-model ensemble routing ($k > 1$) of subsection 2.3, asking whether complementary proposer sets plus aggregation can move the quality–cost frontier past strong single-model baselines. Every configuration in this group—single-model baselines included—runs in the same OpenSquilla harness, so the execution substrate is held fixed. DRACO is the primary benchmark because deep-research tasks reward complementary strengths: one model retrieves better evidence, another synthesizes more coherent analysis, a third produces more reliable citations. The selected configuration uses DeepSeek-V4, GLM-5.2, Gemini-3 Flash, and Qwen3.7-Max as proposers with GLM-5.2 as aggregator, plus extra stochastic samples for the two cheapest proposers.

Table 11 reports the default-provider (DuckDuckGo search) results. The strongest single model, Fable 5, averages 59.80 at $1.2122 over the 94 tasks it completed; the ensemble reaches 60.82 over the full 100 at $0.3766—1.02 points higher and 68.9% cheaper. This is the cost-saving regime of subsection 2.3 realized at benchmark scale: complementary mid-priced proposers plus an aggregator dominate the best single model on both axes, paying instead in tokens and wall-clock time, which parallel proposer execution, early stopping, and per-proposer budgets can control. The mixture-of-agents preset Hermes MoA [24]—reference models advise privately while a fixed aggregator acts—lands at 59.55 and $0.4460, below our configuration on both axes. Swapping the search provider to Brave (Table 12) amplifies the substitution: 64.09 at $0.1218 per task, i.e., +2.03 points and −90.8% cost against Fable 5, +4.98 and −92.5% against Opus-4.8, +10.81 and −85.5% against GPT-5.5. Near-saturated PinchBench shows the parity mode instead: the ensemble ties the strongest single model (94.31 vs 94.33 for Opus-4.8) at 18.2% lower cost, and sits 0.58 above GPT-5.5 at higher cost.

Table 11 | DRACO (default DuckDuckGo search): single-model baselines versus the selected multi-model ensemble routing configuration. All configurations run in the OPENSQUILLA harness. Costs are per-task averages; tokens in thousands. Fable 5 completed 94/100 tasks and its metrics are averaged over completed tasks; all other configurations completed all 100.

Method	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-Max	49.34	0.0432	99.5
Gemini-3 Flash	40.79	0.0117	9.5
Multi-model ensemble routing (ours)	60.82	0.3766	579.7

Table 12 | DRACO (Brave search): single-model baselines versus the selected multi-model ensemble routing configuration. All configurations run in the OPENSQUILLA harness. Costs are per-task averages; tokens in thousands (visible plus reasoning tokens). Fable 5 completed 93/100 tasks because safety filtering blocked 7; its metrics are averaged over the 93 completed tasks; all other configurations completed all 100.

Method	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 ensemble routing (ours)	64.09	0.1218	500.1

The results above fix the proposer set in advance per run. We additionally let the agentic router assemble the set at run time, holding only GLM-5.2 fixed as aggregator, and evaluate three routing-policy operating points on default-provider DRACO: *control* (the current policy), *diversity-heavy* (a larger α on the complementarity term of Equation (7)), and *quality-heavy* (a lighter cost weight λ). Table 13 yields two observations. First, diversity-heavy wins (60.31 vs 59.18 for control) at slightly lower cost—behavioral evidence that what improves aggregated quality is steering subsets toward decorrelated failures, not merely adding models. Second, quality-heavy buys the cheapest point ($0.2582) at a small quality gap (59.93). The best dynamic group comes within 0.51 of the hand-picked configuration of Table 11 at 15.8% lower cost, with no manual proposer curation at all.

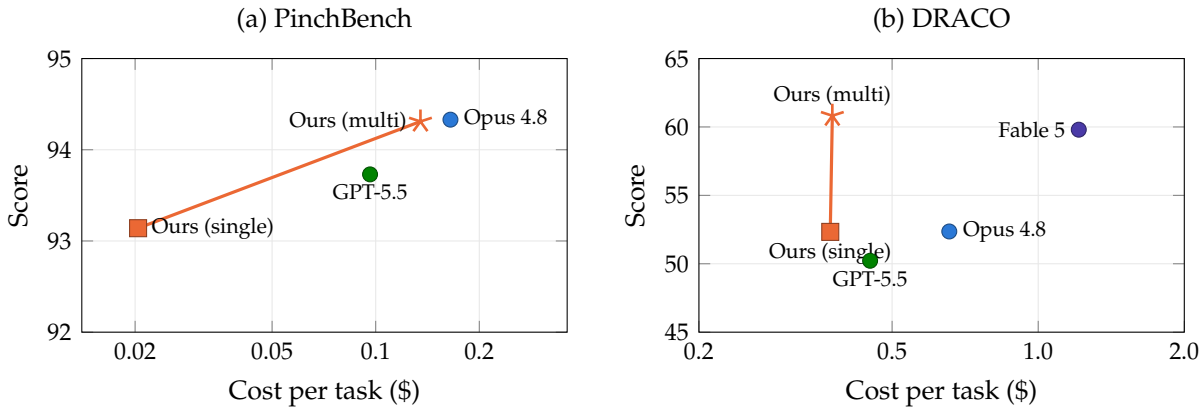

Figure 5 | Cost–score view of the singleton-routing and multi-model ensemble results on PinchBench and DRACO. Colored dots are fixed single-model baselines, colored by model family (blue Opus, green GPT-5.5, violet Fable 5); the orange square and star, joined by a line, are our singleton router and multi-model ensemble. Cost is on a log scale.

Table 13 | Routed ensembles on DRACO: the router assembles the proposer set at run time, with GLM-5.2 fixed as the aggregator. All configurations run in the OPENSQUILLA harness. Costs are per-task averages; tokens in thousands.

Routing policy	**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 collects the singleton and ensemble results into one cost–score view per benchmark (omitting the OpenClaw baseline and baselines cheaper than our singleton router). On DRACO both of our operating points dominate the strongest baselines—the ensemble outscores Opus-4.8 and Fable 5 at far lower cost, and the singleton router matches the Opus-4.8 quality point at a fraction of the price. On PinchBench, where strong baselines are close to saturation, the singleton router anchors the cheapest point and the ensemble nearly reaches the Opus-4.8 quality point while spending less. Both regimes of Equation (3) thus move the frontier: $k = 1$ compresses the cost side, and $k > 1$ advances quality and cost together on deep-research tasks.

7. Related Work

7.1. Agent Systems

At the paradigm level, the main line of 2023–2025 optimized model parameters through scaling laws, data scale, and training compute, producing a base-model pool $\mathcal{M}$ represented by GPT [32], Claude [2], GLM [44], Kimi [34], and DeepSeek [39]. In the agent era, the key question shifts from "how to train a stronger single model" to "how to learn to schedule and constrain the model pool"—that is, how to organize $\mathcal{M}$ as Agent = Base Models + Harness. Our objective (2) characterizes the harness's value under exactly this shift: selecting models through g_t and organizing prompts, RAG, tools, memory, and safety boundaries through s_t to raise Intelligence/Token.

The core idea of LLM agent systems is that the model no longer merely generates text: it explicitly maintains intermediate state, calls external tools, observes environment feedback, and revises its plan during execution. ReAct [42] is a representative early paradigm, interleaving reasoning traces with actions so the model can gather information from the environment in question answering, fact verification, and interactive decision tasks, adjusting its plan mid-execution. Compared with pure prompt engineering, ReAct abstracts the reasoning–acting–observation loop into a reusable interaction pattern, laying the conceptual groundwork for tool-calling agent frameworks.

At the engineering level, LangChain [5] packages models, prompt templates, tool interfaces, retrievers, vector databases, and external APIs into composable standardized components, markedly lowering the barrier to building LLM applications and agent prototypes. As agent applications evolved from simple chains to long-horizon, recoverable, controllable workflows, LangGraph [17] models agent execution as a directed graph, emphasizing durable execution, human-in-the-loop, branch control, and long-term state management. LlamaIndex [20] enters from data connection and RAG, gradually expanding into an agentic application framework for document processing, index construction, and workflow orchestration, especially where private data, retrieval, and tool calls are tightly coupled.

Multi-agent systems are another important branch. AutoGen [37] organizes conversational agents into group-chat-style collaboration in which each agent combines LLM reasoning, human input, and tool execution, decomposing complex tasks across specialized agents with different roles, permissions, or toolsets. CrewAI [7] likewise adopts role-based collaboration, organizing autonomous agents and event-driven processes through Crews and Flows abstractions closer to enterprise automation. The common trait is extending a single model call into multi-turn, multi-role, multi-tool coordinated execution—at the cost of new engineering challenges in state consistency, error-propagation suppression, permission control, and observability.

Recent open-source personal agent systems emphasize local operation, cross-channel interaction, and real tool execution. OpenClaw [33] integrates gateways, sessions, tools, and multi-channel message entries into a local-first personal assistant supporting file operations, shell execution, browser automation, scheduled tasks, and multi-agent routing; Hermes Agent [25] further emphasizes long-term memory, dynamic skill generation, session retrieval, scheduled tasks, and multi-backend deployment on a similar form; nanobot [29] pursues a smaller, more readable core providing long-running agents, chat channels, memory, and the Model Context Protocol (MCP). Overall, agent systems are evolving from ReAct-style reasoning–acting prompting into full harness infrastructure covering model routing, tool abstraction, state management, skill orchestration, persistent memory, security sandboxes, and observability.

7.2. Efficient Agent Techniques

Work on end-to-end LLM and agent inference efficiency falls roughly into three classes by optimization target: the context side tackles the quadratic blow-up of attention memory and compute with sequence length; the input side targets redundant tokens in the prompt; and the model side amortizes call cost at the request and token granularity with a "small model first, large model backstop" pattern.

Context side: KV-cache compression and sparse attention. This line addresses two bottlenecks of long-context processing: KV-cache memory grows linearly but is large in absolute terms, and attention compute grows as $O(N^2)$. StreamingLLM [38] observes that the earliest tokens act as *attention sinks* in long streaming inference, so keeping only the sink and sliding-

window KV suffices for stable generation; H2O [45] dynamically retains Heavy Hitters with the highest cumulative attention scores; SnapKV [19] moves the decision before generation, compressing the KV cache selectively from intra-prompt attention patterns; PyramidKV [4] allocates pyramid-style along layers, reflecting the information funnel across attention layers. Beyond which cache entries to drop, KIVI [21] approaches from numerical precision with tuning-free asymmetric 2-bit KV-cache quantization; MInference [13] identifies A-shaped, vertical-slash, and block-sparse attention patterns and accelerates long-context prefill with sparse attention kernels.

Input side: prompt compression and token reduction. While context-side work targets the KV storage and attention compute of existing context, input-side work goes one level up and shrinks the prompt the LLM sees. LLMLingua [12] uses a small model to estimate token perplexity and compresses prompts coarse-to-fine at sentence and token granularity; LLMLingua-2 [28] distills a task-agnostic prompt compressor that compresses faithfully and quickly across downstream tasks; LongLLMLingua [14] adds key-information density estimation and positional-bias compensation for long contexts, mitigating dilution of key information in long documents. On the soft-compression side, Gisting [23] trains the LLM to compress reusable prompts into cacheable gist tokens for later reuse; ICAE [10] trains an in-context autoencoder that compresses long context into short soft prompts.

Model side: large–small model cooperation and dynamic routing. The core idea is to stop giving every request to the same tier of model and instead spread work across model sizes by granularity. At the token level, speculative decoding [18] accelerates single decoding with draft-generate/verify, compressing N serial steps into near-one-step parallel verification; Medusa [3] attaches parallel decoding heads to the main model, removing the separate draft model. At the request level, FrugalGPT [6] learns LLM cascades that jointly choose model combinations across call cost, answer quality, and early stopping; AutoMix [1] estimates small-model answer confidence via few-shot self-verification and lets a POMDP router decide whether to escalate; RouteLLM [26] trains routers with preference-pair supervision to make binary strong/weak decisions.

All three classes focus on local efficiency of single LLM inference, corresponding to local optimization of $P(a)$ or s_t in Equation (2): KV-cache and prompt compression reduce input tokens, dynamic routing reduces model unit price, and cascades trade quality against early stopping. As evaluation rises from LLM to agent, HAL [15] brings 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, re-evaluating agent effectiveness under fixed budgets. Together they make end-to-end inference efficiency a first-order metric alongside resolve rate. OpenSquilla differs in not treating these techniques as isolated optimizations: it folds them into a learnable harness so g_t and s_t act in concert across subtasks in end-to-end execution.

8. Conclusion and Discussion

This report introduced OpenSquilla, an open-source agent-native learnable harness for real tool execution, multi-channel interaction, and long-horizon collaboration. Unlike system paths that simply chase stronger models, OpenSquilla decomposes agent capability into four system resources—routable model budget, evolvable Meta-Skill orchestration, governable long-term memory, and auditable tool boundaries—and brings them into a unified turn lifecycle through a microkernel execution path. Formally, these resources are the engineering decom-

position of $g_t(\mathcal{M})$ and s_t in Equation (2): agentic routing and the runtime cost mechanisms jointly optimize $P(a)$ (with multi-model ensemble routing additionally advancing $\mathcal{L}_T(a)$ and $P(a)$ together on deep-research tasks), Meta-Skills and memory raise the learnability of s_t, and security governance constrains the executable range of $g_t \circ s_t$. Experiments on end-to-end tasks including PinchBench, ClawMark, ClawSWEBench, and DRACO show that OPENSQUILLA improves the cost curve while maintaining competitive task performance, driven by local routing, prompt-cache continuity, on-demand skill injection, and harness-level observability that together yield cost–capability Pareto improvements.

Three cost-related aspects of these results warrant further discussion: why the multi-model ensemble costs less than a single strong model, how cross-model routing interacts with prompt-cache continuity, and what the router itself costs.

The apparent paradox of the ensemble results—more models, yet less money—dissolves once cost is read as tokens times unit price. Frontier models are priced one to two orders of magnitude above mid-tier models, so several cheap proposers plus a mid-tier aggregator can consume several times more tokens and still bill a fraction of a single frontier trajectory: on DRACO the selected ensemble spends 579.7K tokens per task against Fable 5's 93.7K, yet costs \$0.3766 against \$1.2122 (Table 11). What the cheaper calls cannot buy—frontier-level quality—is recovered structurally rather than purchased: the complementarity regularizer of Equation (7) steers proposer sets toward decorrelated failure modes, and the verification-conditioned aggregator keeps a result at least as good as the best proposer.

Routing across models does, however, carry a cost of its own at the cache layer: provider-side prompt caches are isolated per model, so every switch forfeits the cached stable prefix and re-pays it at full price. This penalty is bounded rather than ignored. The trajectory-level cost $C(\tau)$ that the router optimizes already includes the re-prefill cost of a switch, so a switch happens only when the expected price improvement exceeds the cache loss—in effect a hysteresis against oscillating between tiers; in practice routing decisions align with the task being executed rather than with individual turns—within a multi-turn session the model is not switched every turn but held stable over a task phase—so the cached prefix keeps hitting; and the stable-base / dynamic-tail organization (subsection 3.3) keeps the re-paid prefix small while logged cache events keep the net effect auditable. The DRACO numbers confirm that the balance is positive: the routed configuration uses slightly more tokens than the fixed one yet still cuts cost by 43% (Table 10). Making cache state an explicit input to the routing decision—cache-aware routing—is a natural next step for the router-model iterations.

The last piece of overhead is the router itself, which runs at every step and must therefore stay on the useful side of its own quality–cost trade-off. The released $g^{(0)}$ is a local LightGBM pipeline: it completes in milliseconds, consumes no LLM tokens, and sends nothing to remote services, so its marginal cost per decision is negligible against a model call. Later learned generations are heavier, but they remain small models (on the order of 4B parameters), far cheaper to invoke than the large models being routed and emitting only a handful of output tokens per decision; deployment nonetheless stays two-tier (subsection 2.4)—cheap admission filters resolve clearly trivial or clearly hard states, and the learned router is invoked only on uncertain, high-value, or weakly recoverable states where a better decision is worth its overhead—and why the promotion gate measures frontier movement with the router's own cost and latency included, so router overhead can never silently erode the savings it produces.

Overall, the central claim of OPENSQUILLA is that the competitiveness of production agents should not be determined by model leaderboards alone, but jointly by task completion per unit cost, learnable skill orchestration, long-term context governance, and the safety of real tool

execution. From the scaling-law era to the agent era, the focus of competition is shifting from training a stronger $\mathcal{M}$ to designing a better harness; OpenSquilla is an open-source exploration of raising Intelligence/Token with a learnable harness under this paradigm shift. With continued use, the harness consolidates preferences, procedures, memory, and safety boundaries into reviewable assets, growing ever closer to how each user works. OPENSQUILLA is open-sourced under Apache 2.0 at https://github.com/opensquilla/opensquilla; community contributions and feedback are welcome.

Appendix

A. Contributors

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

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

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] TokenRhythm Technologies. Agentic routing: The harness-native data flywheel. *arXiv preprint arXiv:2607.11399*, 2026.

[36] 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.

[37] 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.

[38] G. Xiao, Y. Tian, B. Chen, et al. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*, 2024.

[39] 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.

[40] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, volume 37, pages 50528–50652, 2024. URL https://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

[41] 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.

[42] 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.

[43] Z.ai. GLM-5.1: Open-weight agentic coding model. https://huggingface.co/zai-org/GLM-5.1, 2026.

[44] 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.

[45] 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.

[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.