JIT-AGENT: SCALING HARNESS INTELLIGENCE VIA JUST-IN-TIME HARNESS EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Agent capability is not determined by the model alone. The agent harness, encompassing memory management, planning strategy, action protocol, and tool/skill orchestration, can dominate the contribution of the underlying foundation model. Yet harness design remains manual, task-specific, and fundamentally unscalable. We present JIT-Agent, a harness intelligence model trained to synthesize task-adaptive agent harnesses on the fly for arbitrary off-the-shelf agentic LLMs. We formalize the agent harness as a composable, machine-generatable artifact governed by a fixed four-module protocol, and train JIT-Agent to *customize harnesses* for a given task at hand, *repair harnesses* for stable and reliable execution, and *self-evolve* by distilling performance signals from an expanding archive of prior harness configurations. Equipped with JIT-Agent as a harness helper, DeepSeek-V4-Flash surpasses GPT-5.6 on DeepSearchQA ($+9.1$) and OdysseyBench ($+4.3$), while the already strong GLM-5.2 gains up to $+20.2$ points. Across controlled evaluations, JIT-Agent-generated harnesses are performance-competitive with mature agent runtimes such as OpenCode and Claude Code and consistently improve multi-scale model families of DeepSeek V4, Mimo-V2.5, and Qwen3.6. To our knowledge, JIT-Agent is the first model purpose-built for just-in-time harness generation, establishing harness intelligence as a trainable, transferable, and compounding dimension of agent capability orthogonal to model scaling.

1 INTRODUCTION

Agent harness matters. The capability of an LLM agent is jointly determined by two tightly coupled factors: the *foundation model* that produces reasoning and actions, and the *agent harness* that situates this model in a closed-loop execution environment (Meng et al., 2026; Zhou et al., 2026; Ning et al., 2026). The harness decides what history is retained, how local intent is formed, which tools or skills are exposed, how actions are executed, and when verification or recovery is triggered (Meng et al., 2026; Zhou et al., 2026). A strong model may fail when placed behind the wrong memory, planner, or action protocol; a strong harness, in turn, can only unlock capability when the model can understand and follow it (Lee et al., 2026c; Lin et al., 2026). Thus, agent intelligence is not a property of model weights alone, but of the model–harness pair (Zhou et al., 2026).

Ahead-of-Time (AOT) Harness. This observation has motivated a growing line of work on test-time harness optimization (Lee et al., 2026c; Lin et al., 2026). Recent systems optimize harness code, prompts, tools, memories, skills, or control policies from trajectories and feedback, yielding strong improvements in coding, terminal-use, web interaction, and general agentic domains (Lee et al., 2026c; Lin et al., 2026). Despite their differences, many of these methods share an *Ahead-of-Time* (AOT) assumption: the harness is treated as a durable artifact to be optimized over an experience stream, with the hope that the resulting artifact will generalize across future tasks, domains, or model versions (Lee et al., 2026c; Lin et al., 2026). This is a powerful paradigm when the deployment distribution is stable and homogeneous (Lin et al., 2026; Zhang et al., 2025). However, it still asks the optimization loop to precompile a broadly useful harness before seeing the exact structure of each future problem (Lee et al., 2026c).

Just-in-Time (JIT) Harness. We instead turn to a ***Just-in-Time* (JIT) view of harness construction**. Different tasks plainly require different harness priors. Wide-search tasks may benefit from parallel evidence exploration (Qin et al., 2025); terminal tasks often favor a lean serial ReAct loop (Yao et al., 2023; Merrill et al., 2026); deep-research tasks require working memory over retrieved evidence (Hu et al., 2025); and NL2Repo-style coding tasks are naturally mediated by a filesystem that stores

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

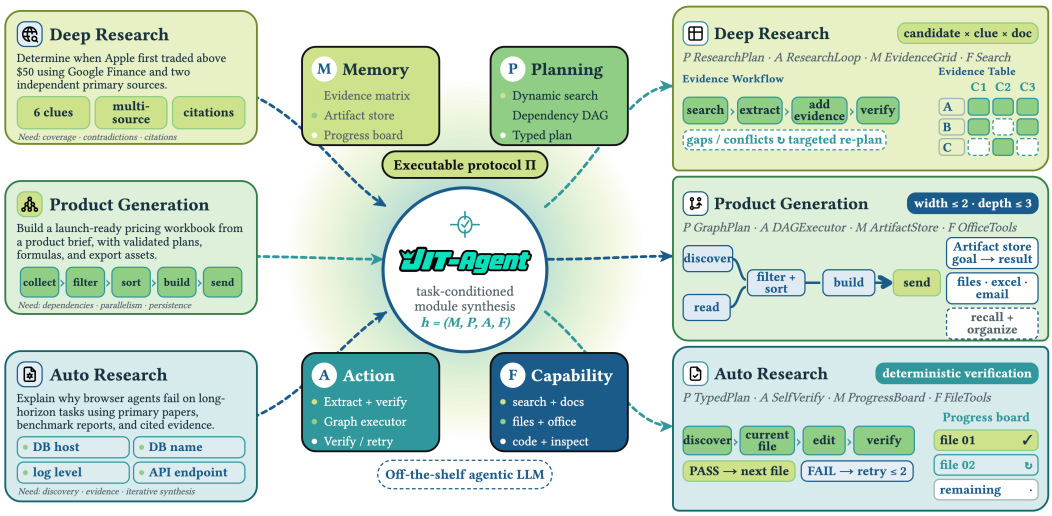

Figure 1: **Overview of JIT-Agent.** Given a task, JIT-Agent composes a problem-specific agent harness by instantiating (rather than simply combining) four modules: memory, planning, action, and capability. Different task structures therefore induce distinct executable protocols.

patches, tests, traces, and repository state (Yang et al., 2024; Ning et al., 2026). In short, the appropriate harness is not only domain-dependent, but instance-dependent (Meng et al., 2026; Zhou et al., 2026). Optimizing a single AOT harness across such heterogeneous demands is cumbersome: one must search a large design space, accumulate enough trajectories, and then hope that the resulting scaffold matches the next task (Lee et al., 2026c; Lin et al., 2026). We ask whether another possibility is viable: **Model-as-a-Harness**, where a trained meta-agent generates a task-specific harness on the spot, and an arbitrary agentic LLM executes under that harness.

JIT-Agent. We propose JIT-Agent, a compact meta-agent for just-in-time harness generation and evolution. At inference time, JIT-Agent receives a task specification, a protocol, an executable tool/skill registry, and a small retrieved context of prior harnesses, and emits an executable harness tailored to the task at hand. The resulting harness wraps an off-the-shelf agentic LLM for execution. As feedback and traces accumulate, JIT-Agent can revise the harness and update a harness archive, enabling test-time harness evolution while the generator itself remains fixed. Training such a system raises three challenges: ❶ **adaptivity**, namely how to match the generated harness to the task; ❷ **reliability**, namely how to ensure executable behavior and recover when synthesis fails; and ❸ **evolvability**, namely how to turn execution feedback into stronger future harnesses. Together, these three requirements define what we call **harness intelligence**:

Harness Intelligence

Harness intelligence is the capacity to construct and refine the operational scaffold through which a model acts. It has three defining properties: **adaptivity**, matching the harness to the task and backbone; **reliability**, producing executable behavior and recovering when synthesis fails; and **evolvability**, turning execution feedback into stronger future harnesses. Intelligence about the harness begins where competence within a fixed harness ends.

JIT-Agent realizes harness intelligence through a three-stage training recipe over a protocol-induced harness space (Meng et al., 2026; Zhou et al., 2026). First, we factor an agent harness into a four-tuple of *memory*, *planning*, *action*, and *capability-orchestration* modules, and build HARNESSFACTORY, a unified codebase of representative scaffolds under this common interface (Hu et al., 2025; Erdogan et al., 2025; Yao et al., 2023; Xu & Yan, 2026). Following the view that *code carries the harness* (Ning et al., 2026; Pan et al., 2026), this representation makes harness generation feasible: JIT-Agent generates structured executable modules rather than unconstrained agent programs. ❶ **Stage I** then teaches task-conditioned customization from teacher-generated protocol-compliant examples, giving the generator adaptivity over the shared design space. Second, ❷ **Stage II** converts failed generations into bounded repair trajectories, turning compiler errors, interface mismatches, and runtime failures into supervision for reliable recovery. Third, ❸ **Stage III** introduces *Evolutionary Group-Decoupled*

Policy Optimization (Evo-GDPO), which optimizes JIT-Agent to propose harnesses that overtake the current archive frontier while separately normalizing reward, latency, and cost. This supplies evolvability by turning test-time harness improvement into a trained capability rather than an external search heuristic. Through this pipeline, we train **JIT-Agent-27B**, a **harness intelligence model** that can customize, repair, and evolve task-specific harnesses, shifting harness engineering from AOT artifacts to native JIT synthesis.

Experiment Findings. Across nine benchmarks, JIT-Agent consistently strengthens its underlying backbones: DeepSeek-V4-Flash surpasses GPT-5.6 by 9.1 points on DeepSearchQA and 4.3 on OdysseyBench, while GLM-5.2 gains up to 20.2 points. The generated harnesses also remain competitive with mature runtimes and transfer across DeepSeek V4, Mimo-V2.5, and Qwen3.6.

Contributions. (I) We formulate *Model-as-a-Harness* and define *harness intelligence* as the learned ability to construct, repair, and evolve an agent's operational scaffold. (II) We instantiate this paradigm with JIT-Agent, which combines a four-module executable protocol, HARNESSFACTORY, and a three-stage customization–repair–evolution pipeline, and (III) validate that JIT-generated harnesses improve diverse backbones while remaining competitive with strong production harnesses.

2 RELATED WORK

Harness Engineering and Modularization. Harnesses have diversified across coding and general-purpose runtimes such as Claude Code, Codex, Hermes, and OpenClaw (Anthropics, 2025; OpenAI, 2025; Nous Research, 2026; OpenClaw Contributors, 2026), embodied systems (Zhang et al., 2026c; Lee et al., 2026b), search agents (Apodex Team, 2026; Qin et al., 2025), and memory-centric systems (Chhikara et al., 2025; Hu et al., 2026; Zhang et al., 2025). Despite domain-specific implementations, they share an operational layer that governs memory, planning and control, action verification, and tool/skill orchestration, making capability a property of the model–harness pair (Meng et al., 2026; Zhou et al., 2026; Pan et al., 2026). Modular representations expose these choices through explicit interfaces, allowing components to be substituted and optimized independently (Chen et al., 2026b; Ning et al., 2026). We therefore use the compact executable factorization $\mathbf{h} = (\mathbf{M}, \mathbf{P}, \mathbf{A}, \mathbf{F})$ for memory, planning, action, and capability orchestration.

Harness Optimization. With explicit interfaces, harness construction becomes optimization over executable artifacts rather than a fixed engineering choice (Khattab et al., 2023; Yuksekgonul et al., 2024; Hu et al., 2024b; Shang et al., 2024). Recent methods extend optimization from prompts and workflow graphs to complete harness code, commonly using a diagnose–propose–validate loop in which traces guide edits and validation determines retention (Zhang et al., 2024; Lee et al., 2026c; Lin et al., 2026; Chen et al., 2026b). Recursive, hierarchical, and online variants improve this loop, but still primarily optimize a persistent artifact against accumulated experience and are therefore ahead-of-time relative to a new task instance (Lee et al., 2026a; Zhou, 2026; Nie et al., 2026b; Shao et al., 2026). JIT-Agent instead amortizes search into a trained generator that constructs a task-conditioned harness just in time; Table 8 summarizes this distinction.

3 UNIFIED HARNESS CODEBASE

3.1 MODULARIZED HARNESS DESIGN SPACE

An agent harness is the operational layer that turns a foundation model into a closed-loop agent (Meng et al., 2026; Zhou et al., 2026): it determines what from prior interaction is retained (Hu et al., 2025), how intermediate directives are formed (Cao et al., 2025), which external capabilities are exposed at each stage (Xu & Yan, 2026; Yang et al., 2026), and how control advances. Harness generation is therefore defined over programs rather than unconstrained text. We distinguish the raw generation space $\mathcal{G}$, syntactically valid harnesses $\mathcal{H}^{\text{syn}}$, protocol-compliant harnesses $\mathcal{H}_{\mathbf{\Pi}}$, and their executable subset $\mathcal{H}_{\mathbf{\Pi}}^{\text{exec}}$, with $\mathcal{G} \supseteq \mathcal{H}^{\text{syn}} \supseteq \mathcal{H}_{\mathbf{\Pi}} \supseteq \mathcal{H}_{\mathbf{\Pi}}^{\text{exec}}$. The fixed protocol $\mathbf{\Pi}$ specifies module schemas and interfaces, their lifecycle, validation rules, and the shared execution semantics. It removes incidental variation in language and runtime while retaining the principal operational choices that distinguish existing harnesses (Wu et al., 2023; Microsoft, 2024; LangGraph, 2025; Luo et al., 2025). JIT-Agent makes these choices explicit through four interoperable modules: *how history is compressed*, *how local intent is formed*, *how tools and skills are orchestrated*, and *how control advances*. Let $\boldsymbol{\tau}$ be a task, π_ψ a frozen backbone executor, and $\mathcal{C}_\tau$ the capability registry available (*e.g.*, tools, APIs, skills) to that task. Running a harness $\mathbf{h}$ with π_ψ induces a closed-loop trajectory

$$\boldsymbol{\xi} \sim \text{Rollout}(\boldsymbol{\tau}, \pi_\psi, \mathbf{h}, \mathcal{C}_\tau; \mathbf{\Pi}) = \big(\mathbf{s}_1, e_1, \mathbf{o}_1, \ldots, \mathbf{s}_T, e_T, \mathbf{o}_T\big), \tag{1}$$

where $\mathbf{s}_t \in \mathcal{S}$ is the maintained controller state, $e_t \in \mathcal{A} = \mathcal{U} \sqcup \mathcal{Y}$ is either an executable call from $\mathcal{U}$ or a terminal output from $\mathcal{Y}$, $\mathbf{o}_t \in \mathcal{O}$ is the resulting observation, and T is the protocol- or budget-bounded stopping time. Our key assumption is that every $\mathbf{h} \in \mathcal{H}_{\mathbf{\Pi}}$ admits:

$$\mathbf{h} = \big(\mathbf{M}, \mathbf{P}, \mathbf{A}, \mathbf{F}\big) \in \mathfrak{M} \times \mathfrak{P} \times \mathfrak{A} \times \mathfrak{F}, \tag{2}$$

where $\mathbf{M}$, $\mathbf{P}$, $\mathbf{A}$, and $\mathbf{F}$ denote memory, planning, action, and capability-orchestration modules, and $\mathfrak{M}$, $\mathfrak{P}$, $\mathfrak{A}$, and $\mathfrak{F}$ are their protocol-compatible implementation spaces. The tuple follows this conceptual decomposition; at runtime, its dependency order is $\mathbf{M} \to \mathbf{P} \to \mathbf{F} \to \mathbf{A}$. All modules operate against the same frozen π_ψ, whose common runtime dependence is suppressed below. The protocol maintains both the immutable event history $\boldsymbol{\xi}_{<t}$ and the mutable controller state $\mathbf{s}_t$, which interact as

$$\begin{aligned}
\mathbf{v}_t &= \mathbf{M}(\boldsymbol{\xi}_{<t}, \mathbf{s}_t) \in \mathcal{V}, && \text{history} \to \text{view},\\
\mathbf{d}_t &= \mathbf{P}(\boldsymbol{\tau}, \mathbf{s}_t, \mathbf{v}_t) \in \mathcal{D}_{\text{dir}}, && \text{view} \to \text{local directive},\\
\mathcal{C}_t &= \mathbf{F}(\mathcal{C}_\tau, \mathbf{s}_t, \mathbf{v}_t, \mathbf{d}_t) \subseteq \mathcal{C}_\tau, && \text{directive-conditioned capability orchestration},\\
(\mathbf{s}_{t+1}, e_t) &= \mathbf{A}(\mathbf{s}_t, \boldsymbol{\tau}, \mathbf{v}_t, \mathbf{d}_t, \mathcal{C}_t) \in \mathcal{S} \times \mathcal{A}, && \text{control update and action emission},
\end{aligned} \tag{3}$$

where $\mathcal{V}$ and $\mathcal{D}_{\text{dir}}$ are the view and directive spaces. The registry $\mathcal{C}_\tau$ contains callable tools (*e.g.*, bash tools, APIs, and MCPs (Sarkar & Sarkar, 2025)) as well as higher-level agent skills (Xu & Yan, 2026; ClaudeCode, 2025); a harness without an explicit planner remains type-consistent through the null directive $\mathbf{d}_\emptyset \in \mathcal{D}_{\text{dir}}$ returned by $\mathbf{P}_\emptyset$. Thus, $\mathbf{M}$ constructs a view of realized history, $\mathbf{P}$ forms a local directive, $\mathbf{F}$ activates relevant capabilities, and $\mathbf{A}$ updates the controller state and emits the next action. The shared kernel $\text{Exec} : \mathcal{U} \times 2^{\mathcal{C}_\tau} \to \mathcal{O}$ sets $\mathbf{o}_t = \text{Exec}(e_t; \mathcal{C}_t)$ for $e_t \in \mathcal{U}$ and $\mathbf{o}_t = \perp$ for $e_t \in \mathcal{Y}$, where $\perp \in \mathcal{O}$ is the terminal null observation; it then updates $\boldsymbol{\xi}_{\le t} = \boldsymbol{\xi}_{<t} \oplus (\mathbf{s}_t, e_t, \mathbf{o}_t)$, with $\oplus$ denoting event append. Execution starts from $\boldsymbol{\xi}_{<1} = \emptyset$ and a protocol-defined $\mathbf{s}_1 \in \mathcal{S}$, and terminates when $e_t \in \mathcal{Y}$, yielding $y = e_T \in \mathcal{Y}$. This factorization maps otherwise heterogeneous programs into comparable coordinates in $\mathfrak{M} \times \mathfrak{P} \times \mathfrak{A} \times \mathfrak{F}$: canonical ReAct uses full-history memory, no explicit planner, a ReAct loop, and the full capability registry; engineered variants such as Codex and OpenCode add compact memory and todo-based planning (OpenAI, 2025; OpenCode-AI, 2025); and recursive systems such as ROMA, AOrchestra, and RLM instantiate subproblem memory, decomposition, recursive action, and routed capabilities (Alzu'bi et al., 2026; Ruan et al., 2026). Given this modularized design space, we next introduce HARNESSFACTORY, whose role is to test the expressiveness of $\mathcal{H}_{\mathbf{\Pi}}$ and furnish diverse source material for JIT-Agent's meta-harness design.

3.2 HARNESSFACTORY

Under the shared protocol $\mathbf{\Pi}$ and a common kernel, HARNESSFACTORY re-implements 13 representative scaffolds spanning memory, planning, aggregation, hierarchical control, and recursive execution; Table 1 gives the full inventory. These implementations form the protocol-compatible seed bank $\mathcal{B}_0$ of size $K_0 = 13$. The bank serves a dual role: its seeds anchor Stage-I synthesis, while later states supply the prior population against which new designs are evaluated. After n evolution rounds, $\mathcal{B}_0$ grows into $\mathcal{B}_n \supseteq \mathcal{B}_0$, whose entries associate each retained harness with its task and observed reward, latency, and cost; this evolution is formalized in Section 4.1.

Table 1: Seed bank $\mathcal{B}_0$: 13 hand-written harnesses instantiating the four-module protocol $\mathbf{\Pi}$. Each row is a complete harness; columns follow the conceptual order of memory, planning, action, and capability orchestration.

Harness	♣ Memory $\mathbf{M} \in \mathfrak{M}$	♦ Planning $\mathbf{P} \in \mathfrak{P}$	♥ Action $\mathbf{A} \in \mathfrak{A}$	♠ Capability $\mathbf{F} \in \mathfrak{F}$
REACT [67]	FullHistory	No explicit planner	ReAct	Full registry
PLAN-AND-EXECUTE [12]	FullHistory	Linear roadmap	ReAct	Full registry
RESUM [62]	ReSum memory	No explicit planner	ReAct	Full registry
FLASH-SEARCHER [52]	FullHistory	DAG planning	ReAct	Full registry
GAM [64]	GAM retrieval	DAG	ReAct	Full registry
MEMOBRAIN [51]	Reasoning graph	No explicit planner	Marker-guided	Full registry
AGGAGENT [29]	Isolated rollout	No explicit planner	Multi-rollout	Full registry
OAGENT [78]	Coordinator history	No explicit planner	Ensemble voting	Full registry
AGENTFOLD [68]	AgentFold memory	DAG	ReActFold	Full registry
HIAGENT [20]	Hierarchical memory	No explicit planner	ReAct	Full registry
DEEPAGENT [30]	Three-tier memory	No explicit planner	Marker-guided	Tool search
ROMA [1]	Context isolation	Atomizer + DAG	Recursive	Full registry
AORCHESTRA [54]	Context isolation	Atomizer + DAG	Recursive	Agent Delegation

4 METHOD

4.1 TRAINING PIPELINE

Training follows the inference-time lifecycle of JIT-Agent: synthesize a task-conditioned harness, recover unstable generations, and then improve from stronger archive states online. Let $\boldsymbol{\tau} \sim \mathcal{D}_{\text{task}}$ denote a training task, $\mathcal{C}_\tau$ its capability registry, and $\mathcal{E}_\tau$ a small reference context from the harness

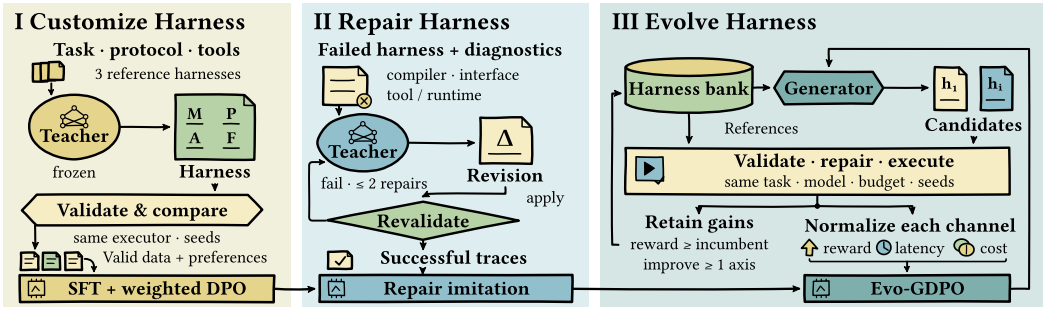

Figure 2: **Training pipeline of JIT-Agent.** Stage I learns task-conditioned harness customization. Stage II converts failed harnesses and execution diagnostics into bounded repair trajectories. Stage III performs online evolution by comparing candidate harnesses against the incumbent bank, optimizing decoupled reward, latency, and cost advantages, and retaining frontier-improving designs.

bank. The generation context is $\mathbf{c}_\tau = (\boldsymbol{\tau}, \mathbf{\Pi}, \mathcal{C}_\tau, \mathcal{E}_\tau)$, from which p_θ generates a harness $\mathbf{h} \in \mathcal{G}$. The frozen executor π_ψ is DeepSeek-V4-Flash-Preview throughout training and evaluates task reward, latency, and monetary cost, while $\mathrm{Valid}_{\mathbf{\Pi}}(\mathbf{h}; \boldsymbol{\tau}, \pi_\psi, \mathcal{C}_\tau) \in \{0, 1\}$ checks protocol and execution validity and returns structured diagnostics upon failure. The three stages optimize this generator through offline supervision, repair-trajectory learning, and online policy improvement, respectively. See Appendix A.5 for training parameters.

Stage I: Data Preparation. Stage I uses a frozen, stronger teacher q_ϕ to synthesize task-adapted harnesses under the fixed four-module protocol. For each task, $\mathcal{E}_\tau$ contains three scaffolds sampled from the task-type-matched subset $\mathcal{B}_0^{(d(\boldsymbol{\tau}))}$, where $d(\boldsymbol{\tau})$ denotes the task type. The teacher receives the resulting context $\mathbf{c}_\tau$, and its generation is retained only after protocol validation and execution checks. Training datasets are specified in Appendix A.2.

Stage I: Training Setup. We first apply standard supervised fine-tuning to accepted teacher generations, teaching JIT-Agent to map a task and its reference context directly to a protocol-compliant harness. Since executable harnesses can still differ substantially in quality and efficiency, we then compare candidates under the same backbone and evaluation seeds. A preference is retained only when reward improves, neither efficiency axis degrades, and at least one efficiency gain is strict:

$$\begin{aligned} \mathbf{h}^+ \succ_\tau \mathbf{h}^- \iff r^+ > r^- \ \wedge\ \ell^+ \le \ell^- \ \wedge\ \kappa^+ \le \kappa^- \ \wedge\ \big(\ell^+ < \ell^- \ \vee\ \kappa^+ < \kappa^-\big), \\ \Delta_{\mathrm{val}}(\boldsymbol{\tau}; \mathbf{h}^+, \mathbf{h}^-) = \alpha_r (r^+ - r^-) + \alpha_\ell [\ell^- - \ell^+]_+ + \alpha_\kappa [\kappa^- - \kappa^+]_+, \end{aligned} \tag{4}$$

where $(r^\pm, \ell^\pm, \kappa^\pm)$ are repeated-rollout averages for $(\mathbf{h}^+, \mathbf{h}^-)$, $[x]_+ = \max(x, 0)$, and $\alpha_r, \alpha_\ell, \alpha_\kappa \ge 0$ weight the reward, latency, and cost gaps. The resulting pairs supervise a standard DPO objective, weighted by Δ_{val} and anchored to the frozen Stage-I SFT checkpoint. Combined with generation imitation, this objective preserves protocol-valid structure while favoring harnesses that are both more effective and more efficient.

Stage II: Data Preparation. Stage I optimizes for protocol compliance, but some generated harnesses still fail static or runtime validation. Rather than discarding them, Stage II pairs each failed harness $\widetilde{\mathbf{h}}^{(0)}$ with a structured diagnostic $\mathbf{g}^{(0)}$ that records compiler errors, interface mismatches, tool-call failures, or runtime exceptions. At repair round k, the teacher proposes a structured revision $\Delta^{(k+1)}$ from patch space $\mathcal{P}$; Apply updates the harness deterministically, after which validation produces the next diagnostic. We retain only trajectories that become executable within two rounds:

$$\begin{aligned} \widetilde{\mathbf{h}}^{(k+1)} &= \mathrm{Apply}\Big(\widetilde{\mathbf{h}}^{(k)}, \Delta^{(k+1)}\Big), \qquad k \in \{0, 1\}, \\ K^\star &= \min\Big\{k \in \{1, 2\} : \mathrm{Valid}_{\mathbf{\Pi}}(\widetilde{\mathbf{h}}^{(k)}; \boldsymbol{\tau}, \pi_\psi, \mathcal{C}_\tau) = 1\Big\}. \end{aligned} \tag{5}$$

Supervision is constructed only when $K^\star$ exists, focusing the corpus on realistic, locally recoverable failures rather than those requiring wholesale redesign.

Stage II: Training Setup. Stage II trains JIT-Agent by teacher-forced imitation of the successful revisions $\{\Delta^{\star(k+1)}\}_{k=0}^{K^\star-1}$. Each revision is conditioned on $\mathbf{c}_\tau$ and the complete preceding harness–diagnostic history $\{(\widetilde{\mathbf{h}}^{(j)}, \mathbf{g}^{(j)})\}_{j=0}^{k}$. Because $K^\star \le 2$, the model learns the short-horizon regime

needed at deployment: use execution feedback to make a small number of high-leverage revisions that restore protocol-valid execution.

Stage III: Data Preparation. Stage III treats *test-time harness evolution* as a trainable capability: JIT-Agent learns to propose harnesses that surpass prior designs and then become stronger references for subsequent tasks. We term the resulting online objective **Evolutionary Group-Decoupled Policy Optimization (Evo-GDPO)**, inspired by (Liu et al., 2026a). At online round n, we sample a task $\boldsymbol{\tau}$, retrieve prior high-quality designs $\mathcal{E}_{\tau,n}$ from the current bank $\mathcal{B}_n$, and form $\mathbf{c}_{\tau,n} = (\boldsymbol{\tau}, \mathbf{\Pi}, \mathcal{C}_\tau, \mathcal{E}_{\tau,n})$. The model samples $G > 1$ candidates independently from the current rollout policy; candidates and retrieved designs are evaluated under the same frozen executor π_ψ, budget, and seeds. The highest-reward retrieved harness, with ties broken by lower latency and then lower cost, supplies the incumbent statistics (b_r, b_ℓ, b_κ).

Stage III: Training Setup. Every candidate is validated before execution; one that remains invalid after bounded repair receives the minimum task reward, while its repair latency and cost remain part of the measured efficiency. For executable candidate i, let r_i, $\bar{\ell}_i$, and $\bar{\kappa}_i$ denote reward, mean latency, and mean monetary cost over repeated rollouts. The reward channel is primary, whereas efficiency gains activate only when incumbent reward is preserved:

$$R_i^{\text{rew}} = r_i + \lambda_{\text{evo}}\,[r_i - b_r]_+,\ R_i^{\text{lat}} = \mathbb{I}[r_i \geq b_r]\,[b_\ell - \bar{\ell}_i]_+,\ R_i^{\text{cost}} = \mathbb{I}[r_i \geq b_r]\,[b_\kappa - \bar{\kappa}_i]_+, \quad (6)$$

where $\lambda_{\text{evo}} \geq 0$ controls the improvement bonus and $\mathbb{I}[\cdot]$ is the indicator function. Evo-GDPO normalizes the three channels independently within each candidate group before aggregation, preventing their scales from overwhelming one another:

$$\begin{aligned} A_i^m &= \frac{R_i^m - \text{mean}(\{R_j^m\}_{j=1}^G)}{\text{std}(\{R_j^m\}_{j=1}^G) + \varepsilon_{\text{num}}}, \qquad m \in \{\text{rew}, \text{lat}, \text{cost}\}, \\ A_i^\Sigma &= w_{\text{rew}} A_i^{\text{rew}} + w_{\text{lat}} A_i^{\text{lat}} + w_{\text{cost}} A_i^{\text{cost}}, \quad w_{\text{rew}} > w_{\text{lat}} + w_{\text{cost}}. \end{aligned} \quad (7)$$

The nonnegative weights sum to one, and the stated inequality makes task reward dominant; $\varepsilon_{\text{num}} > 0$ stabilizes normalization. We further normalize A_i^Σ across the training batch and use it in a standard clipped PPO objective with a token-level KL penalty to the frozen Stage-II checkpoint. Thus, unlike standard group-relative optimization, Evo-GDPO rewards candidates for overtaking prior designs and, when reward is preserved, for doing so more efficiently. After optimization, we update the harness bank conservatively: a candidate is retained only when it Pareto-dominates the incumbent—reward is at least as high, latency and cost are no greater, and at least one comparison is strict. During training, this feedback updates both θ and $\mathcal{B}_n$; at deployment, θ remains frozen.

4.2 Inference Architecture

JIT-Agent supports two inference modes: *static inference* and *streaming inference*; they differ in whether experience is discarded after the task or retained to support subsequent tasks.

Static inference. In this mode, we introduce a lightweight form of test-time scaling: JIT-Agent generates $N = 3$ harnesses in parallel, selects one by length-normalized log-probability, and executes only the selected harness. This increases candidate diversity without increasing the number of environment rollouts. The selected harness follows the same validation and bounded-repair procedure described above; Appendix A.3 details the candidate-selection options.

Streaming inference. Streaming inference mode is designed to carry useful experience forward across a sequence of tasks. For the n-th task $\boldsymbol{\tau}_n$, JIT-Agent retrieves from the current bank $\mathcal{B}_n$, generates and selects a harness $\mathbf{h}_n^\dagger$, and produces one scored rollout. The resulting environment feedback is then used only to determine whether this experience should update the bank:

$$\begin{aligned} \boldsymbol{\xi}_n \sim \text{Rollout}(\boldsymbol{\tau}_n, \pi_\psi, \mathbf{h}_n^\dagger, \mathcal{C}_{\tau_n}; \mathbf{\Pi}), \mathbf{m}_n = \text{Eval}(\boldsymbol{\xi}_n), \\ \mathcal{B}_{n+1} = \text{Update}_{\text{III}}(\mathcal{B}_n; \boldsymbol{\tau}_n, \mathbf{h}_n^\dagger, \mathbf{m}_n), \mathcal{E}_{\tau_{n+1},n+1} = \text{Retrieve}(\boldsymbol{\tau}_{n+1}; \mathcal{B}_{n+1}), \end{aligned} \quad (8)$$

where $\mathbf{m}_n = (r_n, \bar{\ell}_n, \bar{\kappa}_n)$ contains reward, mean latency, and mean monetary cost. Following the Stage-III retention rule, $\text{Update}_{\text{III}}$ admits a harness only after it Pareto-dominates the incumbent on the current task and on mean statistics over five recalled historical tasks; otherwise, $\mathcal{B}_n$ remains unchanged. The retained harness becomes a potential reference for later tasks. Streaming inference therefore transfers prior experience through the evolving harness bank, without injecting terminal reward feedback into the current rollout or updating model parameters. See Appendix A.4 for the streaming evaluation rules.

Table 2: Main results across nine agentic benchmarks. Rows prefixed by JIT-Agent use the same backbone as their vanilla counterpart but replace its default scaffold (ReAct) with a JIT-generated harness. Scores are reported on a 0–100 scale; higher is better. Top-1, Top-2, and Top-3 results in each column are highlighted in soft green, yellow-green, and yellow, respectively; Top-1 is bolded.

Model	Deep Research			Daily Work		Planning		Workspace	
	BC+	DSQA	xBench	AgentIF	PinchBench	Shop	Travel	Office	Odyssey
Qwen3.7-Plus	70.5	78.0	75.0	59.9	80.5	75.2	52.1	56.6	67.7
GLM-5.2	72.0	89.2	76.0	63.0	87.0	78.2	62.8	63.0	75.3
DeepSeek-V4-Flash	68.1	76.2	70.0	58.4	81.7	59.1	54.8	61.0	71.0
DeepSeek-V4-Pro	71.4	72.4	79.0	56.5	61.1	71.1	55.2	62.0	72.0
Kimi K2.7 Code	73.6	87.8	79.0	57.3	76.1	72.8	56.9	65.8	70.0
GPT-5.6	76.9	76.0	81.0	68.0	84.2	83.7	**84.9**	65.3	68.7
Gemini 3.1 Pro	67.5	75.8	83.0	60.1	81.0	77.4	70.7	60.6	74.0
Gemini 3.5 Flash	75.0	88.0	85.0	64.0	74.2	76.2	50.3	63.3	78.0
JIT-Agent + GLM-5.2	**78.0**	**93.9**	**88.0**	**69.9**	**93.3**	83.4	83.0	**68.4**	**78.7**
JIT-Agent + DeepSeek-V4-Flash	74.0	85.1	82.0	63.8	92.9	**83.9**	61.3	63.4	73.0

5 Experiments and Analysis

5.1 Experiment Setup

Evaluation Benchmarks. We evaluate JIT-Agent on nine benchmarks spanning four task types. **Deep Research** includes BrowseComp-Plus (Chen et al., 2025b) (accuracy), DeepSearchQA (Gupta et al., 2026) (answer F1), and xBench-DeepSearch (xBench-DS) (Chen et al., 2025a) (accuracy); **Daily Work** includes AgentIF-Oneday (Chen et al., 2026a) (normalized weighted rubric score) and PinchBench (PinchBench, 2026) (average score); **Planning** includes the DeepPlanning-Shopping (cart match rate) and DeepPlanning-Travel (composite constraint score) subsets (Zhang et al., 2026b); and **Workspace** includes OfficeBench (Wang et al., 2024) and OdysseyBench (Wang et al., 2025) (task success rate for both). See Appendix A.1 for benchmark details.

Backbone Models. JIT-Agent is trained based on Qwen3.6-27B (see Appendices A.2 and A.5 for training data/parameters). We primarily instantiate JIT-Agent with GLM-5.2 (Z.ai, 2026) and DeepSeek-V4-Flash-Preview (DeepSeek-AI, 2026). Qwen3.6 and Mimo-V2.5 series are also tested.

Baselines. We use two complementary baseline groups. First, the broad comparison includes vanilla **Qwen3.7-Plus** (Qwen Team, 2026), **GLM-5.2**, **DeepSeek-V4-Flash/Pro** (both preview version), **Kimi K2.7 Code** (Moonshot AI, 2026), **GPT-5.6-Sol** (OpenAI, 2026), and **Gemini 3.1 Pro/3.5 Flash** (Google DeepMind, 2026a;b). Second, holding the backbone fixed, we compare JIT-Agent with five reusable agent harnesses: Claude Code (Anthropics, 2025), Codex (OpenAI, 2025), OpenCode (OpenCode-AI, 2025), Hermes (Nous Research, 2026), and NanoBot (HKUDS, 2026). The first group measures end-to-end competitiveness, while the second isolates the contribution of harness design from that of model weights.

5.2 Main Results

JIT-generated harnesses improve all 18 matched backbone–benchmark pairs and lead eight of nine benchmarks in Table 2. Average scores rise from 74.1 to 81.8 on GLM-5.2 (+7.8) and from 66.7 to 75.5 on DeepSeek-V4-Flash (+8.8), with the largest gains on DeepPlanning-Shopping for DeepSeek-V4-Flash (59.1 → 83.9, +24.8) and DeepPlanning-Travel for GLM-5.2 (62.8 → 83.0, +20.2). JIT-Agent + GLM-5.2 ranks first on seven benchmarks, including 93.9 on DeepSearchQA and 69.9 on AgentIF, while JIT-Agent + DeepSeek-V4-Flash leads DeepPlanning-Shopping and exceeds DeepSeek-V4-Pro on every benchmark by 8.7 points on average. These results show that task-adaptive harnesses consistently raise the capability of a fixed backbone.

5.3 Comparison with Advanced Agent Harnesses

Across the six fixed-backbone settings in Table 3, JIT-Agent leads in performance in four and achieves the lowest token consumption and API cost in all six. It exceeds the strongest fixed harness by 4.7 points on DeepSearchQA and 4.0 on xBench-DS with DeepSeek-V4-Flash, and by 7.0 on xBench-DS and 2.9 on AgentIF with Qwen3.6-Flash. Per-case cost falls by 14.9–54.1% relative to the cheapest fixed alternative (36.0% on average); on DeepSeek-V4-Flash xBench-DS, tokens drop from 527K to 212K and cost from $0.075 to $0.039 as performance rises from 78.0 to 82.0. Task-conditioned scaffolds thus combine competitive performance with more efficient execution.

Table 3: Controlled comparison with advanced agent harnesses. For each fixed backbone, we report task performance (Perf.), average token consumption per case in thousands (#Tokens (K)), and task-execution API cost per case in USD. Performance is better when higher; tokens and cost are better when lower. The best value within each backbone group is bolded.

Backbone	Harness	DeepSearchQA			xBench-DS			AgentIF		
		Perf. ↑	#Tokens (K)	Cost ↓	Perf. ↑	#Tokens (K)	Cost ↓	Perf. ↑	#Tokens (K)	Cost ↓
DeepSeek-V4-Flash	Claude Code	79.6	625	$0.088	75.0	559	$0.079	**66.9**	808	$0.114
	Codex	77.8	760	$0.107	70.0	680	$0.096	58.5	870	$0.123
	OpenCode	75.9	1,832	$0.258	65.0	1,157	$0.159	48.1	950	$0.135
	Hermes	69.9	1,157	$0.163	72.0	1,254	$0.177	60.3	1,000	$0.142
	NanoBot	80.4	924	$0.131	78.0	527	$0.075	53.1	1,034	$0.147
	JIT-Agent	**85.1**	**506**	**$0.066**	**82.0**	**212**	**$0.039**	63.8	**486**	**$0.097**
Qwen3.6-Flash	Claude Code	72.8	710	$0.140	58.0	650	$0.128	55.4	900	$0.177
	Codex	68.5	980	$0.193	52.0	874	$0.172	34.2	839	$0.170
	OpenCode	64.1	1,968	$0.384	44.0	1,300	$0.256	38.7	961	$0.217
	Hermes	60.8	1,319	$0.256	36.0	1,155	$0.227	49.7	1,184	$0.223
	NanoBot	**74.2**	892	$0.197	63.0	597	$0.119	43.5	950	$0.187
	JIT-Agent	70.3	**464**	**$0.095**	**70.0**	**300**	**$0.069**	**58.3**	**394**	**$0.078**

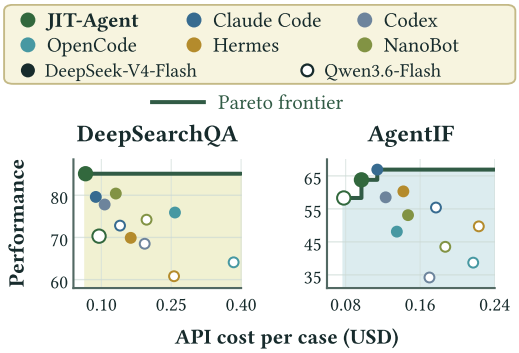

Figure 3: **Cost–performance frontiers on DeepSearchQA and AgentIF.** Color identifies the harness; filled/open circles denote DeepSeek-V4-Flash/Qwen3.6-Flash. Axes show execution API cost per case (USD; x) and performance (y). The green step line marks the global Pareto frontier; shading indicates dominated pairings.

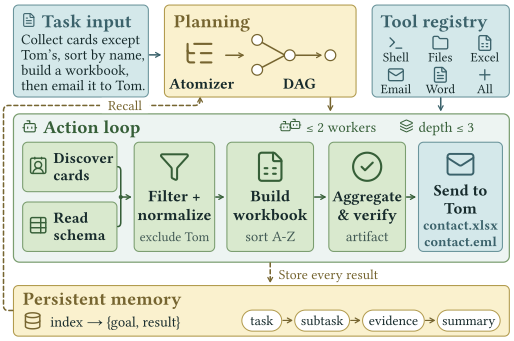

Figure 4: **Palimpsest: graph-planned artifact execution.** `GraphPlanPlanning` maps the contact-processing request to a DAG; `GraphPlanAction` executes ready nodes with $\leq$ 2 workers and depth $\leq$ 3. `GraphPlanMemory` retains reusable artifacts and reasoning state.

Cost–Performance Pareto Frontiers JIT-generated harnesses reach the cost–performance frontier on both DeepSearchQA and AgentIF (Figure 3). On DeepSearchQA, DeepSeek-V4-Flash + JIT-Agent achieves 85.1 at $0.066 per case, surpassing NanoBot by 4.7 points at 49.6% lower cost. On AgentIF, Qwen3.6-Flash + JIT-Agent achieves 58.3 at $0.078 per case, outperforming all fixed harnesses while reducing cost by 54.1% relative to the cheapest baseline under the same backbone. These costs cover task execution. Harness generation adds only a small computational overhead: less than 10% of task-execution time, with total output across all three candidates and repairs amounting to approximately 4–5% of task-execution tokens (Appendix C.2).

5.4 Framework Analysis

Training Dynamics & Ablation Study. We analyze the training stages of JIT-Agent in detail, comparing the model before and after each stage under the same DeepSeek-V4-Flash executor (Table 4). Stage-I DPO raises Valid@0 from 75.4% to 88.6% and reduces cost from $0.090 to $0.083 (7.8%), while improving both task scores. Stage-II repair training raises Valid@1 to 98.3%: one repair recovers approximately 65% of initially invalid harnesses, compared with 14% before repair training, converting otherwise failed runs into valid executions. Stage-III RL achieves the highest DSQA/AgentIF scores (85.1/63.8) at the lowest cost ($0.077), while increasing mean code size from 673 to 710 lines. Total code tokens also grow from about 5.25K after SFT to 5.40K after RL, consistent with learning more elaborate harnesses while improving task performance.

Table 4: **Training dynamics across successive checkpoints.** DSQA and AgentIF denote DeepSearchQA and AgentIF-Oneday scores on a 0–100 scale, with DeepSeek-V4-Flash as executor. Valid@0 and Valid@1 report protocol and execution validity (%) before repair and after at most one repair round. Each module reports mean code tokens and lines; task-execution API cost is in USD per case.

Checkpoint	DSQA	AgentIF	Valid@0	Valid@1	Memory		Planning		Capability		Action		Cost
					Tokens	Lines	Tokens	Lines	Tokens	Lines	Tokens	Lines	
Stage I–SFT	78.9	60.3	75.4	79.1	1,290	170	771	102	501	70	2,688	329	0.090
Stage I–SFT+DPO	79.5	60.9	88.6	90.2	1,210	165	769	105	503	72	2,703	335	0.083
Stage II	83.4	61.1	95.2	98.3	1,223	168	762	105	493	70	2,719	330	0.081
Stage III	85.1	63.8	94.7	97.9	1,382	180	758	103	515	76	2,740	351	0.077

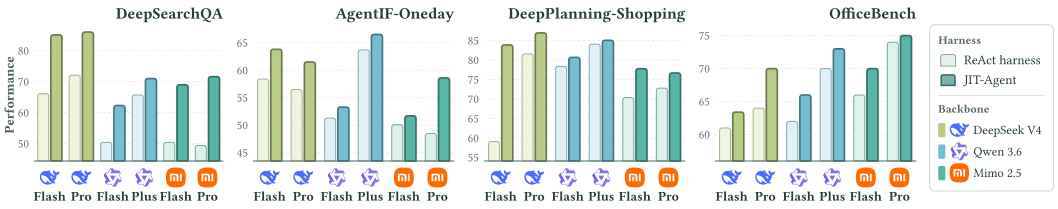

Figure 5: **JIT-Agent's generalization across model families.** Within each panel, paired bars compare the same backbone under a fixed ReAct harness and a JIT-generated harness, with six backbones grouped by model family and scores shown above the bars. DeepSearchQA uses a 100-example subset; the others use 50.

Generalization across Model Pairs. To test transfer across model families and scales, we compare ReAct with JIT-generated harnesses on six fixed backbones from DeepSeek V4, Qwen3.6, and Mimo-V2.5 across four benchmarks. Figure 5 shows that JIT-Agent wins all 24 matched comparisons by 7.6 points on average, with family-level gains of 10.2, 4.0, and 8.6 points, respectively. DeepSearchQA benefits most (+15.2 on average), including +22.2 for Mimo-V2.5-Pro and +19.0 for DeepSeek-V4-Flash, while DeepPlanning-Shopping reaches a +24.8 gain. All training executions use DeepSeek-V4-Flash-Preview (Appendix A.2), so the consistent gains on Qwen3.6 and Mimo-V2.5 directly demonstrate generalization to executor families unseen during training.

5.5 Visualization of Generated Harnesses

To inspect the generated structures, Figure 4 presents PALIMPSEST, a harness for discovering contact cards, filtering and normalizing records, constructing a workbook, and delivering it by email. It compiles these dependencies into a DAG: discovery precedes filtering, workbook construction waits for normalized records, and delivery follows artifact verification. `GraphPlanAction` executes ready nodes with bounded width and depth, while `GraphPlanMemory` preserves goals, evidence, and artifacts for downstream nodes. This task-shaped coordination of planning, memory, and execution goes beyond prompt variation and turns the request into a verifiable production pipeline; Appendix D shows complementary structures for recursive delegation, context folding, gated tools, phase-conditioned execution, evidence tracking, numerical state, and workspace repair.

6 Conclusion

We introduced JIT-Agent, a dedicated model for synthesizing, repairing, and evolving task-conditioned agent harnesses at inference time. By expressing the harness through a composable four-module protocol and training the generator through customization learning, repair supervision, and Evo-GDPO, JIT-Agent turns harness construction from a manually engineered, ahead-of-time artifact into a learned capability. Across deep research, daily work, planning, and workspace tasks, JIT-generated harnesses consistently strengthen their underlying backbones, compete with frontier models and advanced fixed harnesses, and improve cost–performance trade-offs. These results establish harness intelligence, *i.e.*, the capacity to adapt the operational scaffold through which a model acts, as a trainable and transferable source of agent capability beyond model weights alone.

AI USE STATEMENT

In this work, we used generative AI tools, including OpenAI Codex, to support literature discovery, research ideation and execution, manuscript drafting, figure preparation, and code development. Additional uses included translation, writing refinement, and improvements to methodological descriptions and mathematical notation. The use of LLMs to synthesize task-adapted harnesses and construct repair supervision is described in Section 4.1. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including text, claims, code, and figures produced with the aid of generative AI.

ETHICS STATEMENT

We have not identified distinctive ethical risks arising from this work beyond those commonly associated with LLM-based agent systems, including unreliable outputs, bias, privacy concerns, and potential misuse. These considerations remain relevant when deploying JIT-generated harnesses in real-world agent systems.

REPRODUCIBILITY STATEMENT

Our anonymous repository at https://anonymous.4open.science/r/jit provides HARNESSFACTORY, the JIT-Agent inference engine, and our data engine. We will release training scripts and model weights upon acceptance. The harness protocol, training pipeline, and inference architecture are described in Sections 3, 4.1 and 4.2, respectively. See Appendix A for benchmark details, training data curation, candidate selection, streaming evaluation rules, and training parameters.

REFERENCES

Salaheddin Alzu'bi, Baran Nama, Arda Kaz, Anushri Eswaran, Weiyuan Chen, Sarvesh Khetan, Rishab Bala, Tu Vu, and Sewoong Oh. Roma: Recursive open meta-agent framework for long-horizon multi-agent systems, 2026. URL https://arxiv.org/abs/2602.01848.

Anthropics. GitHub - anthropics/claude-code: Claude Code is an agentic coding tool that lives in your terminal, understands your codebase, and helps you code faster by executing routine tasks, explaining complex code, and handling git workflows - all through natural language commands. — github.com. https://github.com/anthropics/claude-code, 2025. [Accessed 04-01-2026].

Apodex Team. Apodex-1.0: Self-evolving heavy-duty solver. https://www.apodex.com/blog/apodex-1.0, 2026.

Pengfei Cao, Tianyi Men, Wencan Liu, Jingwen Zhang, Xuzhao Li, Xixun Lin, Dianbo Sui, Yanan Cao, Kang Liu, and Jun Zhao. Large language models for planning: A comprehensive and systematic survey, 2025. URL https://arxiv.org/abs/2505.19683.

Kaiyuan Chen, Yixin Ren, Yang Liu, Xiaobo Hu, Haotong Tian, Tianbao Xie, Fangfu Liu, Haoye Zhang, Hongzhang Liu, Yuan Gong, Chen Sun, Han Hou, Hui Yang, James Pan, Jianan Lou, Jiayi Mao, Jizheng Liu, Jinpeng Li, Kangyi Liu, Kenkun Liu, Rui Wang, Run Li, Tong Niu, Wenlong Zhang, Wenqi Yan, Xuanzheng Wang, Yuchen Zhang, Yi-Hsin Hung, Yuan Jiang, Zexuan Liu, Zihan Yin, Zijian Ma, and Zhiwen Mo. xbench: Tracking agents productivity scaling with profession-aligned real-world evaluations, 2025a. URL https://arxiv.org/abs/2506.13651.

Kaiyuan Chen, Qimin Wu, Taiyu Hou, Tianhao Tang, Xueyu Hu, Yuchen Hou, Bikun Li, Chengming Qian, Guoyin Wang, Haolin Chen, Haotong Tian, Haoye Zhang, Haoyu Bian, Hongbing Pan, Hongkang Zhang, Hongyi Zhou, Jiaqi Cai, Jiewu Rao, Jiyuan Ren, Keduan Huang, Lucia Zhu Huang, Mingyu Yuan, Naixu Guo, Qicheng Tang, Qinyan Zhang, Shuai Chen, Siheng Chen, Ting Ting Li, Xiaoxing Guo, Yaocheng Zuo, Yaoqi Guo, Yinan Wang, Yinzhou Yu, Yize Wang, Yuan Jiang, Yuan Tian, Yuanshuo Zhang, Yuxuan Liu, Yvette Yan Zeng, Zenyu Shan, Zihan Yin, Xiaobo Hu, Yang Liu, Yixin Ren, and Yuan Gong. Agentif-oneday: A task-level instruction-following benchmark for general ai agents in daily scenarios, 2026a. URL https://arxiv.org/abs/2601.20613.

Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, Yuan Xie, Heng Qu, Kun Shao, and Jian Luan. Harnessx: A

composable, adaptive, and evolvable agent harness foundry, 2026b. URL https://arxiv.org/abs/2606.14249.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025b. URL https://arxiv.org/abs/2508.06600.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory, 2025. URL https://arxiv.org/abs/2504.19413.

ClaudeCode. Agent Skills - Claude Code Docs — code.claude.com. https://code.claude.com/docs/en/skills, 2025. [Accessed 03-01-2026].

DeepSeek-AI. Deepseek-v4 preview release. https://api-docs.deepseek.com/news/news260424/, 2026.

Lutfi Eren Erdogan, Nicholas Lee, Sehoon Kim, Suhong Moon, Hiroki Furuta, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. Plan-and-Act: Improving planning of agents for long-horizon tasks. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 15419–15462. PMLR, 2025. URL https://proceedings.mlr.press/v267/erdogan25a.html.

Tianyu Fan and Chao Huang. HELIX: Model-harness co-evolution for recursive self-improvement, 2026. URL https://arxiv.org/abs/2608.13951.

Runnan Fang, Yuan Liang, Xiaobin Wang, Jialong Wu, Shuofei Qiao, Pengjun Xie, Fei Huang, Huajun Chen, and Ningyu Zhang. Memp: Exploring agent procedural memory, 2025. URL https://arxiv.org/abs/2508.06433.

Google DeepMind. Gemini 3.1 pro model card. https://deepmind.google/models/model-cards/gemini-3-1-pro, 2026a.

Google DeepMind. Gemini 3.5 flash model card. https://deepmind.google/models/model-cards/gemini-3-5-flash/, 2026b.

Nikita Gupta, Riju Chatterjee, Lukas Haas, Connie Tao, Andrew Wang, Chang Liu, Hidekazu Oiwa, Elena Gribovskaya, Jan Ackermann, John Blitzer, Sasha Goldshtein, and Dipanjan Das. Deepsearchqa: Bridging the comprehensiveness gap for deep research agents, 2026. URL https://arxiv.org/abs/2601.20975.

HKUDS. Nanobot: Lightweight open-source ai agent. https://github.com/HKUDS/nanobot, 2026.

Chuanrui Hu, Xingze Gao, Zuyi Zhou, Dannong Xu, Yi Bai, Xintong Li, Hui Zhang, Tong Li, Chong Zhang, Lidong Bing, and Yafeng Deng. Evermemos: A self-organizing memory operating system for structured long-horizon reasoning, 2026. URL https://arxiv.org/abs/2601.02163.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. Hiagent: Hierarchical working memory management for solving long-horizon agent tasks with large language model, 2024a. URL https://arxiv.org/abs/2408.09559.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024b.

Yuyang Hu, Shichun Liu, Yanwei Yue, Guibin Zhang, Boyang Liu, Fangyi Zhu, Jiahang Lin, Honglin Guo, Shihan Dou, Zhiheng Xi, Senjie Jin, Jiejun Tan, Yanbin Yin, Jiongnan Liu, Zeyu Zhang, Zhongxiang Sun, Yutao Zhu, Hao Sun, Boci Peng, Zhenrong Cheng, Xuanbo Fan, Jiaxin Guo, Xinlei Yu, Zhenhong Zhou, Zewen Hu, Jiahao Huo, Junhao Wang, Yuwei Niu, Yu Wang, Zhenfei Yin, Xiaobin Hu, Yue Liao, Qiankun Li, Kun Wang, Wangchunshu Zhou, Yixin Liu, Dawei Cheng, Qi Zhang, Tao Gui, Shirui Pan, Yan Zhang, Philip Torr, Zhicheng Dou, Ji-Rong Wen, Xuanjing Huang, Yu-Gang Jiang, and Shuicheng Yan. Memory in the age of ai agents, 2025. URL https://arxiv.org/abs/2512.13564.

Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. MemoHarness: Agent harnesses that learn from experience, 2026. URL https://arxiv.org/abs/2607.14159.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

LangGraph. GitHub - langchain-ai/langgraph: Build resilient language agents as graphs. — github.com. https://github.com/langchain-ai/langgraph, 2025. [Accessed 03-01-2026].

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement, 2026a. URL https://arxiv.org/abs/2607.15524.

Sanghoon Lee, Jiyeong Chae, and Kyung-Joon Park. Harness engineering for physical ai: Robot middleware is the harness layer, 2026b. URL https://arxiv.org/abs/2606.09416.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses, 2026c. URL https://arxiv.org/abs/2603.28052.

Yoonsang Lee, Howard Yen, Xi Ye, and Danqi Chen. Agentic aggregation for parallel scaling of long-horizon agentic tasks, 2026d. URL https://arxiv.org/abs/2604.11753.

Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, and Zhicheng Dou. Deepagent: A general reasoning agent with scalable toolsets, 2026. URL https://arxiv.org/abs/2510.21618.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, and Yu-Gang Jiang. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses, 2026. URL https://arxiv.org/abs/2604.25850.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Peter Belcak, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, Yejin Choi, Jan Kautz, and Pavlo Molchanov. Gdpo: Group reward-decoupled normalization policy optimization for multi-reward rl optimization, 2026a. URL https://arxiv.org/abs/2601.05242.

Zewen Liu, Zhan Shi, Yisi Sang, Bing He, Minhua Lin, Tianxin Wei, Dakuo Wang, Benoit Dumoulin, Wei Jin, and Hanqing Lu. Adaptive auto-harness: Sustained self-improvement for agentic system deployment on open-ended task streams, 2026b. URL https://arxiv.org/abs/2606.01770.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. AutoHarness: Improving llm agents by automatically synthesizing a code harness, 2026. URL https://arxiv.org/abs/2603.03329.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

Qianyu Meng, Yanan Wang, Liyi Chen, Qimeng Wang, Chengqiang Lu, Wei Wu, Yan Gao, Yi Wu, and Yao Hu. Agent harness for large language model agents: A survey. *Preprints*, April 2026. doi: 10.20944/preprints202604.0428.v2. URL https://www.preprints.org/manuscript/202604.0428/v2.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff,

Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Microsoft. GitHub - ag2ai/ag2: AG2 (formerly AutoGen): The Open-Source AgentOS. https://github.com/ag2ai/ag2, 2024. [Accessed 31-08-2025].

Moonshot AI. Kimi k2.7 code: Open-source agentic coding model. https://www.kimi.com/resources/kimi-k2-7-code, 2026.

Jun Nie, Yonggang Zhang, Qianshu Cai, Yiu-ming Cheung, Xinmei Tian, and Bo Han. EvolveNet: Collaborative harness evolution for agent self-improvement, 2026a. URL https://arxiv.org/abs/2608.04968.

Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. TTHE: Test-time harness evolution, 2026b. URL https://arxiv.org/abs/2607.08124.

Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.

Nous Research. Hermes agent. https://github.com/NousResearch/hermes-agent, 2026.

OpenAI. Codex | AI Coding Partner from OpenAI — openai.com. https://openai.com/codex/, 2025.

OpenAI. Gpt-5.6: Frontier intelligence that scales with your ambition. https://openai.com/index/gpt-5-6/, 2026.

OpenClaw Contributors. Openclaw: Your own personal ai assistant. https://github.com/openclaw/openclaw, 2026.

OpenCode-AI. GitHub - opencode-ai/opencode: A powerful AI coding agent. Built for the terminal. — github.com. https://github.com/opencode-ai/opencode, 2025.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory, 2025. URL https://arxiv.org/abs/2509.25140.

Linyue Pan, Lexiao Zou, Shuo Guo, Jingchen Ni, and Hai-Tao Zheng. Natural-language agent harnesses. *arXiv preprint arXiv:2603.25723*, 2026.

PinchBench. Best Models by Success Rate | PinchBench — pinchbench.com. https://pinchbench.com/, 2026. [Accessed 11-04-2026].

Hongjin Qian, Zhao Cao, and Zheng Liu. Memobrain: Executive memory as an agentic brain for reasoning, 2026. URL https://arxiv.org/abs/2601.08079.

Tianrui Qin, Qianben Chen, Sinuo Wang, He Xing, King Zhu, He Zhu, Dingfeng Shi, Xinxin Liu, Ge Zhang, Jiaheng Liu, Yuchen Eleanor Jiang, Xitong Gao, and Wangchunshu Zhou. Flash-searcher: Fast and effective web agents via dag-based parallel execution, 2025. URL https://arxiv.org/abs/2509.25301.

Qwen Team. Qwen3.7-plus: Multimodal agent intelligence. https://qwen.ai/home, 2026.

Jianhao Ruan, Zhihao Xu, Yiran Peng, Fashen Ren, Zhaoyang Yu, Xinbing Liang, Jinyu Xiang, Yongru Chen, Bang Liu, Chenglin Wu, Yuyu Luo, and Jiayi Zhang. Aorchestra: Automating sub-agent creation for agentic orchestration, 2026. URL https://arxiv.org/abs/2602.03786.

Anjana Sarkar and Soumyendu Sarkar. Survey of llm agent communication with mcp: A software design pattern centric review, 2025. URL https://arxiv.org/abs/2506.05364.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. *arXiv preprint arXiv:2410.06153*, 2024.

Shuai Shao, Kangning Zhang, Qingyao Li, Shijian Wang, Hao Wang, Wenxiang Jiao, Yuan Lu, Yi Guo, Weiwen Liu, and Weinan Zhang. Harness-R1: Learning to edit executable runtime harnesses from agent failure trajectories, 2026. URL https://arxiv.org/abs/2608.02276.

Weixuan Wang, Dongge Han, Daniel Madrigal Diaz, Jin Xu, Victor Rühle, and Saravan Rajmohan. Odysseybench: Evaluating llm agents on long-horizon complex office application workflows, 2025. URL https://arxiv.org/abs/2508.09124.

Zilong Wang, Yuedong Cui, Li Zhong, Zimin Zhang, Da Yin, Bill Yuchen Lin, and Jingbo Shang. Officebench: Benchmarking language agents across multiple applications for office automation, 2024. URL https://arxiv.org/abs/2407.19056.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. Autogen: Enabling next-gen llm applications via multi-agent conversation framework, August 01, 2023 2023.

Siwei Wu, Jincheng Ren, Yizhi Li, Haau-Sing Li, Chengran Yang, Yuxuan Zhang, Weicheng Gu, Jian Yang, Riza Batista-Navarro, Chuanyi Zhang, Xianglong Liu, Ming Zhou, Bryan Dai, and Chenghua Lin. ModularRSI: Modular and generalizable recursive harness self-improvement, 2026a. URL https://arxiv.org/abs/2609.14857.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. Resum: Unlocking long-horizon search intelligence via context summarization, 2026b. URL https://arxiv.org/abs/2509.13313.

Renjun Xu and Yang Yan. Agent skills for large language models: Architecture, acquisition, security, and the path forward, 2026. URL https://arxiv.org/abs/2602.12430.

B. Y. Yan, Chaofan Li, Hongjin Qian, Shuqi Lu, and Zheng Liu. General agentic memory via deep research, 2025. URL https://arxiv.org/abs/2511.18423.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024. URL https://arxiv.org/abs/2405.15793.

Xiaofang Yang, Lijun Li, Heng Zhou, Tong Zhu, Xiaoye Qu, Yuchen Fan, Qianshan Wei, Rui Ye, Li Kang, Yiran Qin, Zhiqiang Kou, Daizong Liu, Qi Li, Ning Ding, Siheng Chen, and Jing Shao. Toward efficient agents: Memory, tool learning, and planning, 2026. URL https://arxiv.org/abs/2601.14192.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. Agentfold: Long-horizon web agents with proactive context management, 2025. URL https://arxiv.org/abs/2510.24699.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text, 2024. URL https://arxiv.org/abs/2406.07496.

Z.ai. Glm-5.2: Built for long-horizon tasks. https://z.ai/blog/glm-5.2, 2026.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems, 2025. URL https://arxiv.org/abs/2512.18746.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation, October 2024. URL http://arxiv.org/abs/2410.10762. arXiv:2410.10762.

Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. HarnessCompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses, 2026a. URL https://arxiv.org/abs/2608.01918.

Yinger Zhang, Shutong Jiang, Renhao Li, Jianhong Tu, Yang Su, Lianghao Deng, Xudong Guo, Chenxu Lv, and Junyang Lin. Deepplanning: Benchmarking long-horizon agentic planning with verifiable constraints, 2026b. URL https://arxiv.org/abs/2601.18137.

Yixian Zhang, Huanming Zhang, Feng Gao, Xiao Li, Zhihao Liu, Chunyang Zhu, Jiaxing Qiu, Yuchen Yan, Jiyuan Liu, Wenhao Tang, Zhengru Fang, Yi Nie, Changxu Wei, Yu Wang, Wenbo Ding, and Chao Yu. Harness vla: Steering frozen vlas into reliable manipulation primitives via memory-guided agents, 2026c. URL https://arxiv.org/abs/2607.08448.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, Congming Zheng, Jiachen Zhu, Zeyu Zheng, Zhuosheng Zhang, Xingyu Lou, Changwang Zhang, Zhihui Fu, Jun Wang, Weiwen Liu, Jianghao Lin, and Weinan Zhang. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering, 2026. URL https://arxiv.org/abs/2604.08224.

Tailin Zhou. Hierarchical self-improvement: A framework for task-specific evolvable agent harnesses, 2026. URL https://arxiv.org/abs/2608.08466.

He Zhu, Tianrui Qin, King Zhu, Heyuan Huang, Yeyi Guan, Jinxiang Xia, Yi Yao, Hanhao Li, Ningning Wang, Pai Liu, Tianhao Peng, Xin Gui, Xiaowan Li, Yuhui Liu, Yuchen Eleanor Jiang, Jun Wang, Changwang Zhang, Xiangru Tang, Ge Zhang, Jian Yang, Minghao Liu, Xitong Gao, Jiaheng Liu, and Wangchunshu Zhou. Oagents: An empirical study of building effective agents, 2025. URL https://arxiv.org/abs/2506.15741.

A Experimental Details

A.1 Benchmark Detail

Table 5 summarizes the evaluated cases, primary metrics, and scoring procedures. Counts refer to the test sets used to evaluate JIT-Agent. DeepPlanning-Travel uses the Chinese task set, and OdysseyBench-Plus denotes the plus subset referred to as OdysseyBench in the main text. All scores are scaled to 0–100; higher is better.

Table 5: **Benchmark evaluation summary.** Case counts refer to the main evaluation of JIT-Agent.

Task type	Benchmark	Cases	Primary metric	Evaluation procedure
Deep research	BrowseComp-Plus	200	Accuracy	Official answer evaluation.
	DeepSearchQA	200	Answer F1	LLM-based answer matching; binary correctness for single answers and precision/recall/F1 for answer sets.
	xBench-DeepSearch	100	Accuracy	LLM-as-judge answer evaluation.
Daily work	AgentIF-Oneday	104	Normalized weighted rubric score	Official rubrics and score aggregation; text-only LLM judging of answers and workspace artifact contents.
	PinchBench	20	Average score	Official benchmark scorer.
Planning	DeepPlanning-Shopping	120	Cart match rate	Deterministic programmatic cart evaluation; no LLM judge.
	DeepPlanning-Travel	120	Composite constraint score	LLM parsing of the plan into JSON, followed by deterministic checks of commonsense and personalized constraints.
Workspace	OfficeBench	295	Task success rate	Deterministic checks of generated files and final workspace state.
	OdysseyBench-Plus	300	Task success rate	Task-level evaluation of final workspace artifacts.

Execution settings and analysis subsets. The execution model uses temperature 0. The maximum agent-step budget is 40, except for OdysseyBench-Plus (50). The per-call output limit is 32,000 tokens, except for DeepPlanning-Travel (64,000). The model-pair analysis in Figure 5 uses 100 DeepSearchQA cases and 50 cases each from AgentIF-Oneday, DeepPlanning-Shopping, and OfficeBench; these subsets are distinct from the main evaluation counts in Table 5.

Benchmark scoring and reporting. We follow the benchmark authors' published scoring criteria and report one aggregate score per system and benchmark. For example, the official AgentIF-OneDay and OfficeBench papers likewise present a single aggregate rubric score or task success rate for each evaluated system and setting (Chen et al., 2026a; Wang et al., 2024). We therefore consider this reporting format reasonable and consistent with their published practice. Our scores aggregate one scored execution per case under the shared evaluation settings above.

A.2 Training Data Curation

Our training and test task instances are strictly disjoint, with zero overlap. We collect training harnesses from 2,617 task instances across the five sources in Table 6. OdysseyBench-Neo and OdysseyBench-Plus are two distinct, non-overlapping sets defined by the benchmark authors (Wang et al., 2025); we use Neo exclusively for training and Plus exclusively for evaluation. The 300 BrowseComp-Plus training tasks are also completely disjoint from the 200 BrowseComp-Plus tasks used to evaluate JIT-Agent. No test task is included in training-harness collection or model training.

Table 6: **Task sources for training-harness collection.** Counts refer to source task instances before harness sampling and filtering.

Source	Training tasks
BrowseComp-Plus	300
Monaco	1,315
WebWalkerQA	200
OdysseyBench-Neo	302
ClawGym	500
Total	**2,617**

Harness sampling and filtering. We use DeepSeek-V4-Pro-Preview as the teacher for training-harness collection, independently sampling three JIT harness candidates for each task instance. We first discard candidates that fail execution or encounter runtime errors. We then filter by task outcome: for binary success/failure judgments, we discard unsuccessful candidates; for graded scores in $[0, 1]$, we discard the single lowest-scoring candidate among those that passed the execution checks. The remaining candidates provide the teacher demonstrations for Stage-I supervised fine-tuning (Section 4.1).

Generalization to unseen executors. All execution trajectories used to train JIT-Agent are generated exclusively by DeepSeek-V4-Flash-Preview. DeepSeek-V4-Pro-Preview serves as the harness-generation teacher, while DeepSeek-V4-Flash-Preview is the sole training executor. Consequently, GLM-5.2, Mimo-V2.5, and Qwen3.6 are strictly unseen, out-of-distribution (OOD) executors: none supplies execution trajectories or feedback during training. JIT-Agent improves all nine GLM-5.2 benchmark scores in Table 2 and every Mimo-V2.5 and Qwen3.6 comparison in Figure 5. These gains demonstrate consistent generalization from a single training executor to unseen executor families.

A.3 Harness Candidate Selection

For each task $\boldsymbol{\tau}$, we independently sample $N = 3$ harness candidates $\{\mathbf{h}_i\}_{i=1}^{3}$ from $p_\theta(\cdot \mid \mathbf{c}_\tau)$, using the shared context defined in Section 4.1. We support two selection modes, both applied before task execution.

Length-normalized log-probability. Let $(h_{i,1}, \ldots, h_{i,L_i})$ be the output-token sequence of candidate $\mathbf{h}_i$. We score each candidate by its mean conditional log-probability:

$$s_i = \frac{1}{L_i} \sum_{t=1}^{L_i} \log p_\theta(h_{i,t} \mid \mathbf{c}_\tau, h_{i,<t}), \qquad i^\star = \arg\max_{i \in \{1,2,3\}} s_i. \tag{9}$$

The selected harness is $\mathbf{h}^\dagger = \mathbf{h}_{i^\star}$. Averaging over output tokens puts candidates of different code lengths on the same per-token scoring scale; shared input-context tokens are excluded. **We use this selection mode in our experiments.**

Model self-selection. Alternatively, the same generator receives $\mathbf{c}_\tau$ together with the three candidate harnesses and returns the index of its preferred design. It judges task suitability and protocol consistency directly from the candidate code. In either mode, only the selected harness proceeds to validation, bounded repair, and task execution, as described in Section 4.2.

A.4 Streaming Evaluation Rules

Feedback setting. Using execution feedback to improve subsequent tasks is established practice in test-time agent evolution. Mem^p updates procedural memory using benchmark rewards (Fang et al., 2025), and MemEvolve evolves memory architectures using task-outcome and efficiency feedback (Zhang et al., 2025). ReasoningBank similarly learns from successes and failures (Ouyang et al., 2025). In our streaming setting, the evaluator supplies only a terminal success/failure reward

r_n as task-quality feedback; latency $\bar{\ell}_n$ and monetary cost $\bar{\kappa}_n$ are measured runtime statistics. This feedback arrives after the scored rollout and guides bank updates for subsequent tasks.

Two-stage bank admission. After a 20-task warm-up, we instantiate $\mathrm{Update}_{\mathrm{III}}$ in Equation (8) with a current-task check followed by historical replay. Let $\mathbf{h}_n^{\mathrm{ref}}$ be the retrieved incumbent selected by the Stage-III rule: highest reward, with ties broken by lower latency and then lower cost. Write $\mathbf{m}_{n,j}$ and $\mathbf{b}_{n,j}$ for the $(r, \bar{\ell}, \bar{\kappa})$ statistics of $\mathbf{h}_n^{\dagger}$ and $\mathbf{h}_n^{\mathrm{ref}}$ on task $\boldsymbol{\tau}_j$, respectively, with $\mathbf{m}_{n,n} = \mathbf{m}_n$. For $\mathbf{m} = (r, \bar{\ell}, \bar{\kappa})$ and $\mathbf{b} = (b_r, b_\ell, b_\kappa)$, Pareto dominance is

$$\begin{aligned} \mathbf{m} \succ_{\mathrm{P}} \mathbf{b} \quad \Longleftrightarrow \quad & r \geq b_r \ \wedge \ \bar{\ell} \leq b_\ell \ \wedge \ \bar{\kappa} \leq b_\kappa \\ & \wedge \ \big(r > b_r \ \vee \ \bar{\ell} < b_\ell \ \vee \ \bar{\kappa} < b_\kappa\big). \end{aligned} \tag{10}$$

Only if $\mathbf{m}_n \succ_{\mathrm{P}} \mathbf{b}_{n,n}$ do we recall five completed tasks, indexed by $\mathcal{I}_n \subset \{1, \ldots, n-1\}$ with $|\mathcal{I}_n| = 5$, and replay both harnesses on them. Paired comparisons use the same frozen executor π_ψ, task budgets, and evaluation conditions. Define the historical means $\widehat{\mathbf{m}}_n = \frac{1}{5}\sum_{j \in \mathcal{I}_n} \mathbf{m}_{n,j}$ and $\widehat{\mathbf{b}}_n = \frac{1}{5}\sum_{j \in \mathcal{I}_n} \mathbf{b}_{n,j}$. Admission requires dominance on the current task and on these five-task averages:

$$\mathcal{B}_{n+1} = \begin{cases} \mathcal{B}_n \cup \{\mathbf{h}_n^{\dagger}\}, & \mathbf{m}_n \succ_{\mathrm{P}} \mathbf{b}_{n,n} \ \wedge \ \widehat{\mathbf{m}}_n \succ_{\mathrm{P}} \widehat{\mathbf{b}}_n, \\ \mathcal{B}_n, & \text{otherwise.} \end{cases} \tag{11}$$

The reported task performance comes from the original scored rollout. Admission checks determine the bank used for future tasks, with generator parameters θ fixed throughout the stream.

A.5 Training Parameters

Table 7 summarizes the settings for SFT, DPO, and Evo-GDPO. For Evo-GDPO, each rollout batch contains 16 tasks with $G = 4$ candidate harnesses per task, yielding 64 candidates for one optimizer update.

Table 7: **Training parameters.** A dash denotes a setting that is not applicable.

Parameter	SFT	DPO	Evo-GDPO
Optimization			
Learning rate	10^{-4}	5×10^{-6}	10^{-5}
Epochs	2	2	1 PPO epoch per batch
Per-device micro batch size	1	1	1
Gradient accumulation steps	2	4	8 (with 8-way DP)
Warmup	Ratio 0.03	Ratio 0.03	5 optimizer updates
Learning-rate scheduler	Cosine; minimum $0.1\times$ initial LR	Cosine	Constant after warmup
Training precision	BF16	BF16	BF16
DPO β	—	0.1	—
DPO loss type	—	`sigmoid`	—
RPO α	—	0.5	—
Sequence limits and candidate generation			
Maximum sequence length (tokens)	65,536	65,536	40,960 (prompt + generation)
Maximum prompt length (tokens)	—	—	24,576
Maximum generation length (tokens)	—	—	16,384
Tasks per rollout batch	—	—	16
Candidates per task G	—	—	4
Effective batch per optimizer update	—	—	64 candidates
Optimizer updates per rollout batch	—	—	1
Temperature / top-p / top-k	—	—	1.0 / 1.0 / unrestricted

Continued on the next page.

Table 7 (continued). Training parameters.

Parameter	SFT	DPO	Evo-GDPO
Evo-GDPO objective and archive validation			
KL penalty	—	—	Enabled; coefficient 0.01
KL estimator	—	—	`low_var_kl`
Loss aggregation	—	—	Mean over tokens within each harness, then mean over harnesses
Reward / latency / cost weights	—	—	0.8 / 0.1 / 0.1
Incumbent improvement bonus λ_{evo}	—	—	0.5
Numerical stabilizer ε_{num}	—	—	10^{-6}
Maximum repair attempts	—	—	2
Archive admission re-evaluation	—	—	At least 3 paired executions

For each Evo-GDPO task, all four candidates use the same reference harnesses and executor. The prompt budget includes the protocol, capability descriptions, reference harnesses, and necessary feedback; the generation budget is reserved for harness code. Each candidate batch is trained for one PPO epoch with a single optimizer update. With eight data-parallel workers and one candidate per device micro batch, eight gradient accumulation steps give an effective batch of 64 candidates; accumulation is recalculated when the parallel configuration changes to preserve this batch size. Repair latency and cost are included in evaluation. Before archive admission, each candidate and the incumbent are re-evaluated with at least three paired executions under matched conditions.

B Harness Optimization Paradigms

Table 8 organizes representative harness-optimization methods by when the harness is constructed and which parts of improvement are learned. Existing approaches primarily search for a durable harness ahead of deployment or edit that artifact online, whereas JIT-Agent amortizes construction into a trained generator that synthesizes an instance-specific harness and learns both repair and evolution. The comparison is structural rather than a ranking of empirical performance: it clarifies which capabilities are native to the optimization procedure and which remain external search operations.

Concurrent work. Several recent studies pursue harness optimization and evolution concurrently with JIT-Agent. These include TTHE (Nie et al., 2026b), MemoHarness (Huang et al., 2026), Recursive Harness Self-Improvement (Lee et al., 2026a), HarnessCompass (Zhang et al., 2026a), Harness-R1 (Shao et al., 2026), EvolveNet (Nie et al., 2026a), Hierarchical Self-Improvement (Zhou, 2026), HELIX (Fan & Huang, 2026), and ModularRSI (Wu et al., 2026a).

Table 8: **Harness optimization paradigms.** *Construction* distinguishes harnesses found by ahead-of-time search, ahead-of-time harnesses subsequently edited with test-time feedback, and harnesses generated just in time for each task. *Instance synthesis*, *Harness model*, *Learned repair*, and *Online evolution* indicate whether a method directly synthesizes an instance-specific harness, trains the generator, learns repair from failed execution trajectories, and continues improving after deployment, respectively.

Method	Construction	Instance synthesis	Harness model	Learned repair	Online evolution
AutoHarness (Lou et al., 2026)	AOT (search)	✗	✗	✗	✗
Meta-Harness (Lee et al., 2026c)	AOT (search)	✗	✗	✗	✗
AHE (Lin et al., 2026)	AOT (search)	✗	✗	✗	✗
Adaptive AH (Liu et al., 2026b)	AOT (test-time editing)	✗	✗	✗	✓
TTHE (Nie et al., 2026b)	AOT (test-time editing)	✗	✗	✗	✓
RHI (Lee et al., 2026a)	AOT (test-time editing)	✗	✗	✗	✓
Harness-R1 (Shao et al., 2026)	AOT (test-time editing)	✗	✓	✓	✓
JIT-Agent (ours)	**JIT**	✓	✓	✓	✓



Figure 6: **Streaming test-time harness evolution across task streams.** Cumulative accuracy (top), per-task API cost (middle), and per-task tool calls (bottom) on DeepPlanning-Shopping, DeepPlanning-Travel, and OfficeBench. Dashed curves represent Static JIT, where task-specific harness generations are independent, while solid curves represent Streaming JIT, which continuously incorporates execution feedback as new tasks arrive. Shaded regions show local variation around the streaming trajectories. Streaming JIT finishes with higher cumulative accuracy on all three benchmarks, while API-cost and tool-use trajectories remain task-dependent and of broadly similar scale.

C Additional Results

This section complements the aggregate comparisons in the main paper with the full streaming evaluation and a decomposition of harness-generation overhead. The streaming evaluation tests whether completed tasks can improve the harness bank for later tasks without changing the generator parameters.

C.1 Test-Time Harness Evolution

JIT-Agent is designed not only to generate a harness once, but also to improve its archive from execution feedback. We compare *Static JIT*, where harness generations are independent, with *Streaming JIT*, which retrieves and updates harnesses throughout the evaluation stream.

Figure 6 shows that Streaming JIT finishes above the static variant on DeepPlanning-Shopping, DeepPlanning-Travel, and OfficeBench. Across all three streams, the cumulative-accuracy advantage emerges as execution feedback accumulates and remains positive through the end of evaluation. The accompanying cost and tool-call traces suggest that the endpoint gains are not uniformly coupled to larger interaction budgets. This pattern is consistent with the objective of Evo-GDPO, which retains harnesses only when they advance the archive frontier.

C.2 Harness Generation Overhead

Harness generation is a small fraction of the computation devoted to solving each task. Table 9 reports mean per-rollout statistics on DeepSearchQA and AgentIF-Oneday with DeepSeek-V4-Flash as executor, separating harness generation and repair from task execution. The costs in Figure 3 and tables 3 and 4 cover task-execution API usage.

Table 9: **Harness-generation and task-execution statistics.** All values are means per rollout with DeepSeek-V4-Flash as executor.

Metric per rollout	DeepSearchQA	AgentIF-Oneday
JIT generation time (s)	52	58
Selected harness length (tokens)[1]	**5,424**	**7,790**
Total JIT output tokens, including repair	18,107	24,526
Cumulative task-execution time (s)[2]	**552.80**	**583.62**
Model API time (s)	373.42	402.25
Tool-call time (s)	179.38	181.36
Task input tokens	464,742	439,349
Task output tokens	42,139	47,427
Task total tokens	**506,880**	**486,776**
Execution steps	**22.83**	**17.63**
Execution attempts	1.07	1.20

[1]Token length of the selected harness artifact; total JIT output tokens sum all three candidates and repair outputs.
[2]Execution time accumulated across task-execution attempts within a rollout, comprising model API and tool-call time.

Generation takes 52 and 58 seconds, compared with 552.80 and 583.62 seconds of cumulative task execution, respectively: ratios of 9.4% and 9.9%. Across all three candidates and repairs, JIT emits 18,107 and 24,526 output tokens, equivalent to 3.6% and 5.0% of the corresponding task-execution input-plus-output totals. AgentIF elicits longer harnesses and more execution attempts than DeepSearchQA, while synthesis remains modest relative to the ensuing execution. This breakdown complements the cost–performance results in the main text: a compact, task-specific program governs a substantially larger budget of model and tool interactions.

D Additional Generated Harness Visualizations

This appendix complements the representative case in Section 5.5 with nine additional harnesses generated under the same four-module protocol. The examples span bounded recursive research, constraint-heavy shopping and travel, phased web production, clue-driven research, evidence-sensitive retrieval, numerical analysis, and workspace manipulation. They are not variants of a single execution template: JIT-Agent selects a state representation and control regime suited to the structure of each request, then instantiates planning, action, memory, and capability orchestration around that choice. Collectively, the cases make task-conditioned harness customization visible at the level of execution semantics rather than prompt wording.

Deep research as bounded recursive delegation. TRAPDOOR addresses a multi-hop identity question whose clues span a historical quotation, a university merger, a memoir, and a research paper. Here a fixed DAG would commit too early to an uncertain evidence path, so the generated harness uses `DynamicDecomposer` to form and revise sub-questions and augments the research tool policy with a synthetic `delegate` capability. A `delegate` call opens a private research subagent with its own memory, research-only tools, and a five-step budget; the returned answer re-enters the parent loop as an ordinary observation. In parallel, `FactGraphMemory` extracts compact key–value facts from the growing history, allowing the parent to integrate evidence across branches without inheriting every subagent trace. Thus, the harness makes recursive research a runtime primitive specifically for a task whose uncertainty is resolved through branching evidence collection.

Figures D.2 and D.3 illustrate two task-specific ways to control long-horizon execution through memory. ORIGAMI must assemble a seasonal wardrobe under coupled stock, delivery, size, review-count, and rating constraints. `ROMAPlanning` decomposes the request into dependent product

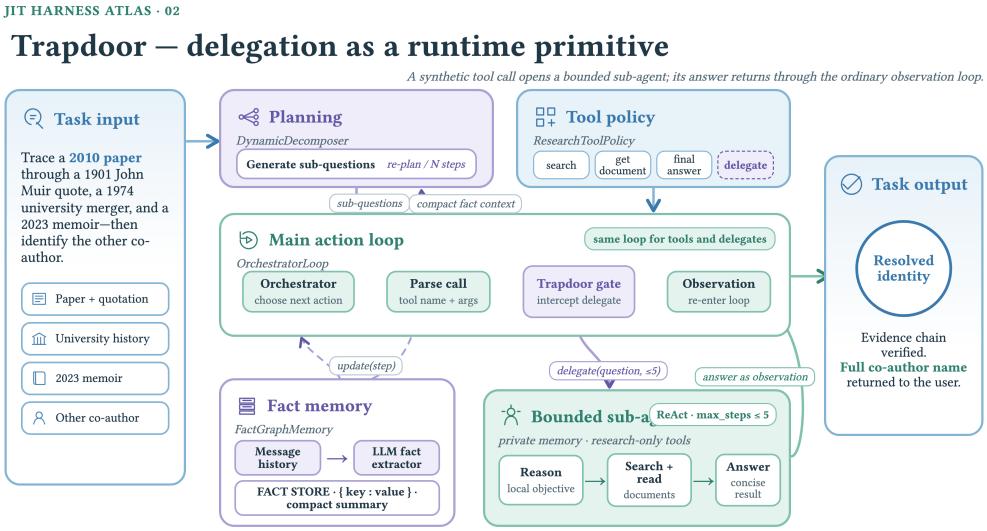

Figure D.1: **Trapdoor: bounded research behind a delegated tool call.** A synthesized `delegate` capability is intercepted by `OrchestratorLoop`, which runs a bounded subagent and writes extracted facts into `FactGraphMemory`.

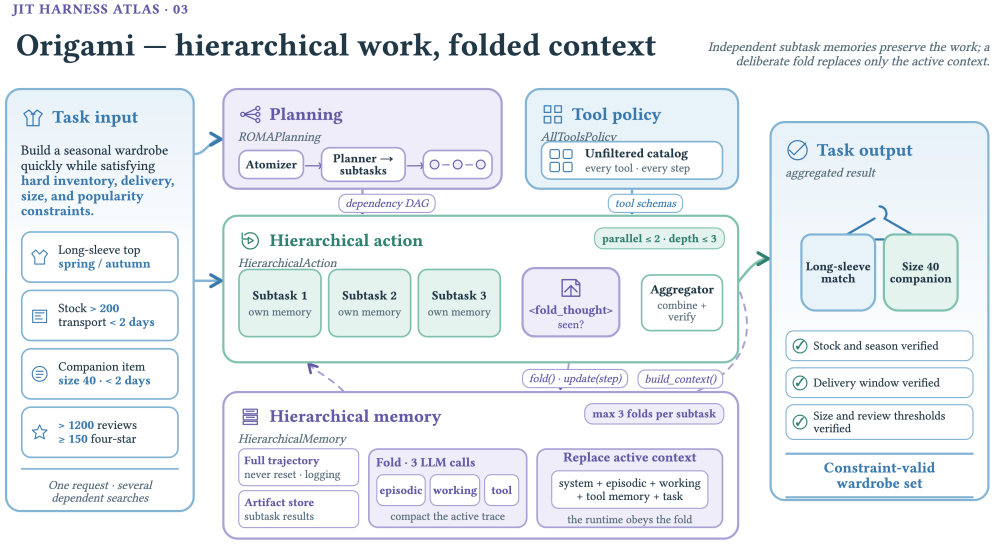

Figure D.2: **Origami: hierarchical work with folded context.** `ROMAPlanning` creates isolated subtasks; `HierarchicalMemory` retains their trajectories and artifacts while `fold_thought` replaces only the active working context before aggregation.

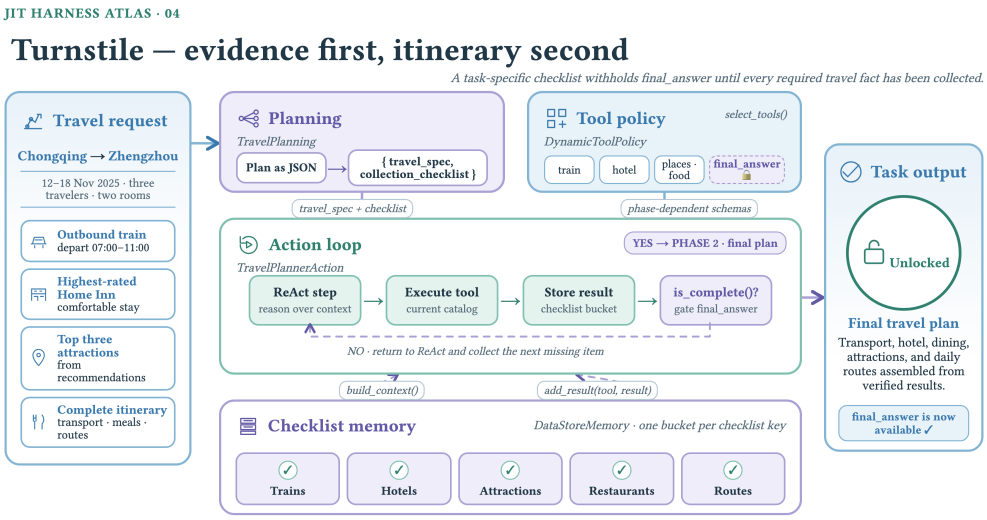

Figure D.3: **Turnstile: evidence first, itinerary second.** `TravelPlanning` emits a travel specification and checklist, `DataStoreMemory` tracks the required evidence buckets, and `DynamicToolPolicy` exposes `final_answer` only after `is_complete()` succeeds.

searches, `HierarchicalAction` runs at most two branches in parallel to bounded depth, and each branch retains its own trajectory and artifacts. When a branch grows long, `fold_thought` replaces only its active working context; completed subtask results remain available for the final constraint check. The harness therefore spends context on the unresolved portion of a combinatorial shopping task without discarding earlier product evidence.

TURNSTILE, by contrast, serves a travel request that specifies dates, passengers, rooms, a departure window, a highest-rated hotel, attractions, meals, and daily routes. The generated `TravelPlanning` module compiles these obligations into a typed `travel_spec` and collection checklist, and `DataStoreMemory` assigns a dedicated bucket to each evidence class. `DynamicToolPolicy` exposes search tools for the missing bucket but withholds `final_answer` until `is_complete()` succeeds. Here memory is not used primarily to compress a long trace; it acts as an executable coverage contract that prevents itinerary synthesis before the required travel facts have been collected.

Figures D.4 and D.5 show how explicit state can continuously reshape later decisions. GEARBOX is generated to find Chinese restaurants within a specified driving radius and publish the results as an interactive one-page guide. Its `PhaseAction` is the sole writer of a shared *search–collect–build* register; the transition to construction occurs only after enough structured restaurant records have been gathered. Both `PhaseToolPolicy` and `PhaseAwareMemory` read that register, switching from web discovery, to record inspection, to code and publication while compressing observations into the schema needed by the current phase. The model thus creates a phase machine because the task changes modality from evidence acquisition to artifact construction.

PEGBOARD instead tackles an identification problem defined by six biographical and film clues. It represents progress as a candidate $\times$ clue matrix in which every extracted claim retains its document identifier. Empty cells and contradictions drive `DynamicResearchPlanning` toward targeted searches, while a sufficiently supported row triggers a separate verification phase that reopens cited documents and resolves conflicts. Unlike a generic research transcript, this state makes coverage and source provenance directly actionable. The contrast with GEARBOX shows that generated state may be a small phase variable or a task-shaped evidence structure, depending on what must govern the next action.

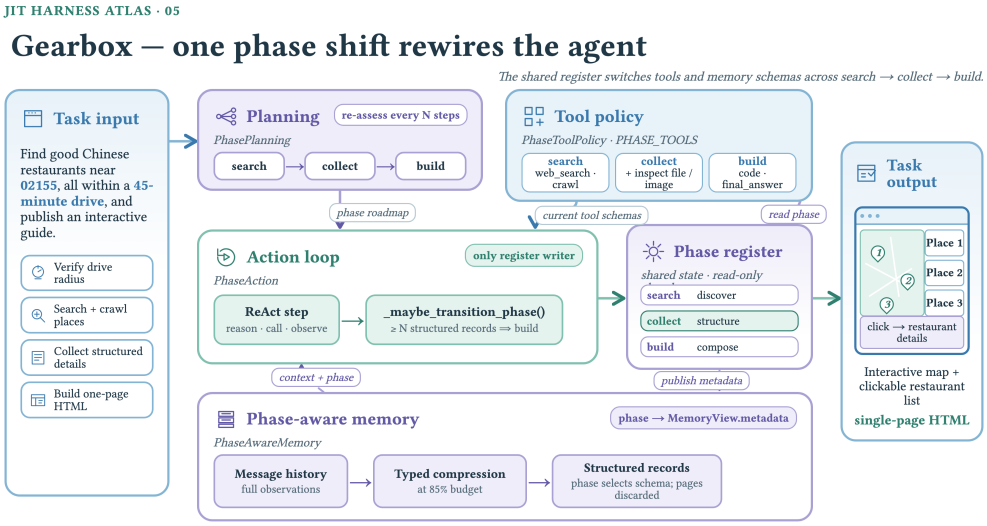

Figure D.4: **Gearbox: one phase shift rewrites the agent.** `PhaseAction` is the sole writer of a shared phase register; `PhaseToolPolicy` and `PhaseAwareMemory` read that state to switch both exposed capabilities and typed memory schemas.

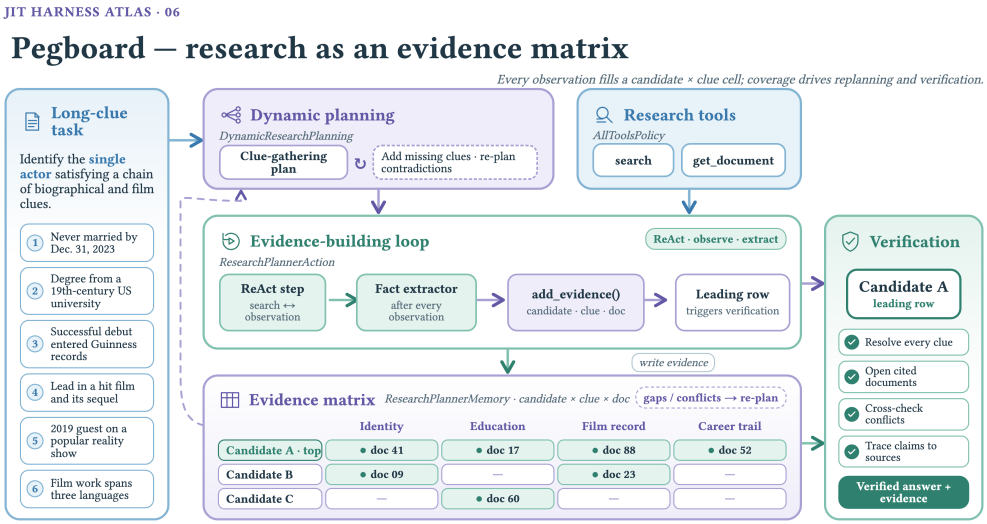

Figure D.5: **Pegboard: research as an evidence matrix.** Every observation is extracted into a candidate × clue cell with a document identifier; matrix coverage drives both `DynamicResearchPlanning` and the transition to source-grounded verification.

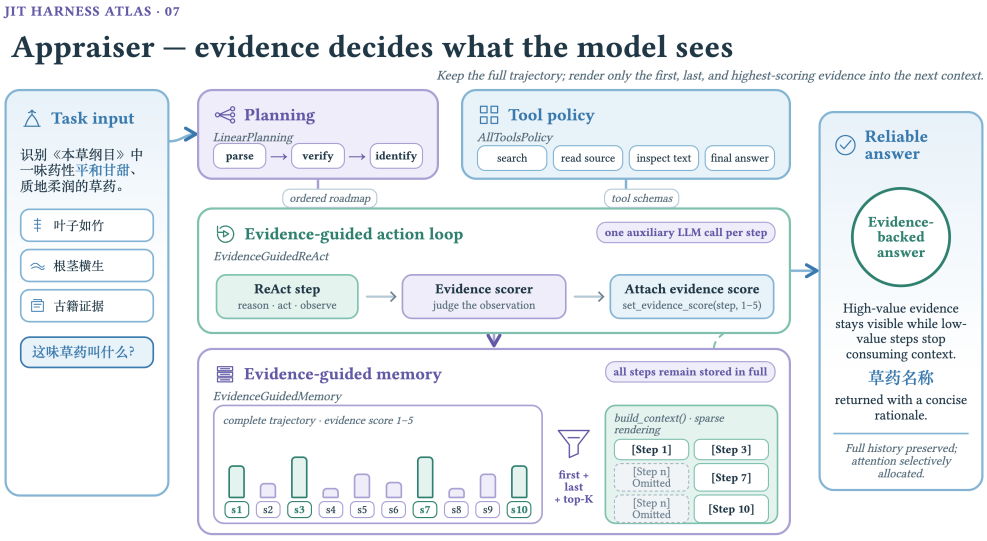

Figure D.6: **Appraiser: evidence decides what the model sees.** `EvidenceGuidedReAct` scores every step, while `EvidenceGuidedMemory` stores the full run but renders only the first, last, and top-K observations into `build_context()`.

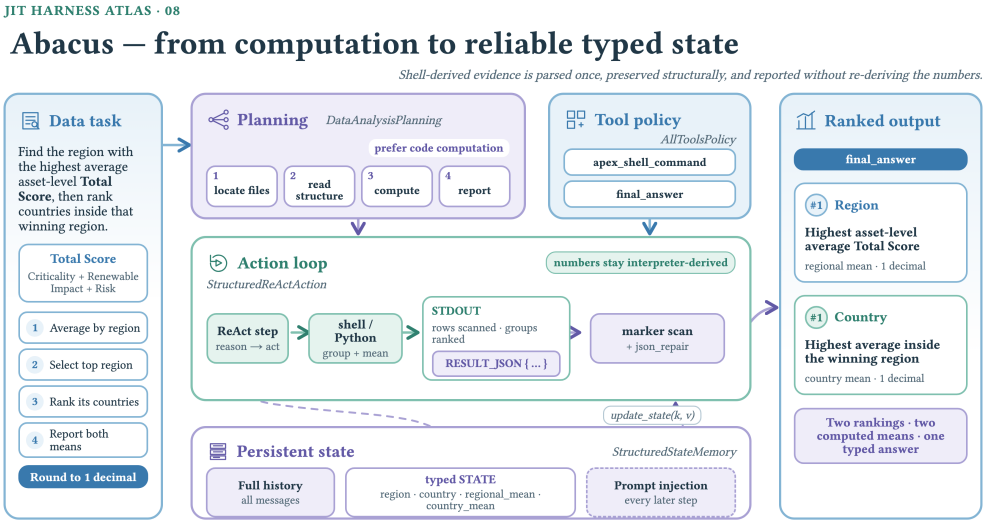

Figure D.7: **Abacus: from computation to reliable typed state.** `StructuredReActAction` executes shell or Python code, extracts a `RESULT_JSON` payload from standard output, and updates `StructuredStateMemory` without asking the model to re-derive computed values.

Figures D.6 and D.7 customize what subsequent reasoning receives from prior execution. APPRAISER answers a classical materia-medica identification question by reconciling botanical traits with textual evidence. After every search or reading step, an auxiliary scorer assigns the observation an evidence value; `EvidenceGuidedMemory` keeps the complete trajectory for auditability but renders only the first, last, and top-K observations into the next context. Low-value exploration therefore stops competing with the passages most useful for identification, while no evidence is erased from persistent history. This harness is tailored to a retrieval task in which the main bottleneck is evidence salience rather than tool choice.

ABACUS is generated for a numerical analysis request that first compares regional averages and then ranks countries within the winning region. The harness deliberately moves aggregation out of language-model reasoning: `StructuredReActAction` invokes shell or Python, extracts a marked `RESULT_JSON` payload from standard output, repairs its syntax if necessary, and writes the values into `StructuredStateMemory`. Region, country, regional mean, and country mean are then injected as typed state into every later step, so the final answer reports interpreter-derived quantities instead of re-deriving them from prose. Both cases preserve a full audit trail, but each renders the working view around its task's dominant reliability requirement—evidence selection for one and numerical fidelity for the other.

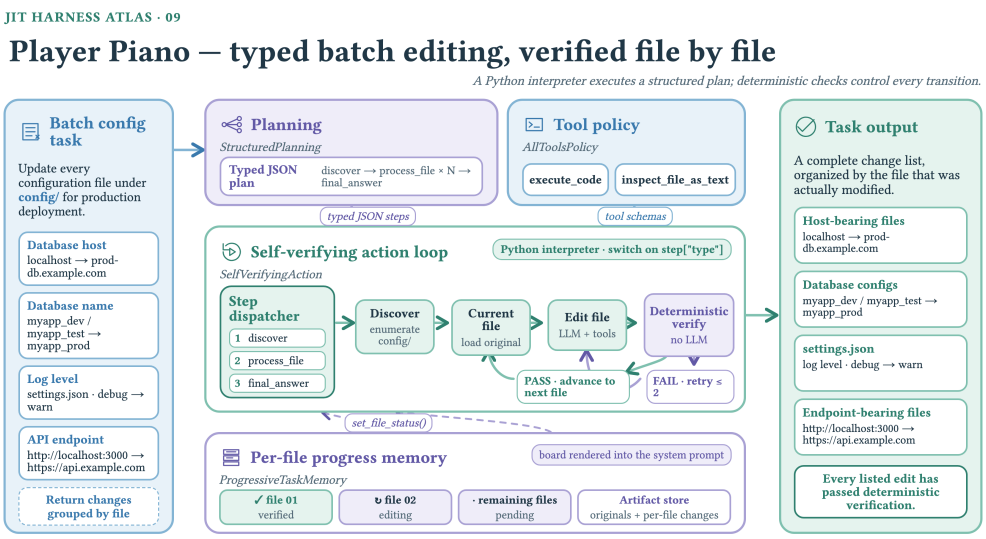

Figure D.8: **Player Piano: typed batch editing, verified file by file.** A Python dispatcher interprets `StructuredPlanning` steps, `SelfVerifyingAction` checks each edit without an LLM, and `ProgressiveTaskMemory` maintains a per-file board and artifact store.

Figures D.8 and D.9 specialize workspace execution at different scales by controlling when actions enter persistent history. PLAYER PIANO handles a batch migration of configuration files, including database, logging, and API-endpoint changes. `StructuredPlanning` emits typed *discover*, *process-file*, and *final-answer* steps that a Python dispatcher executes; after each edit, a deterministic non-LLM check decides whether to advance or retry. `ProgressiveTaskMemory` renders a per-file board into the system prompt and retains both originals and accepted changes, preventing a large batch from losing track of which files have actually passed verification.

MULLIGAN is generated for a shorter document task: extract a student identifier from a DOCX file and rename that same file. Its linear plan needs no hierarchy or context compression, so the harness retains full successful history and exposes only Word and file operations while hiding unrelated email, spreadsheet, and PDF tools. If an action fails, the loop appends the error locally and regenerates the same step for at most two attempts; only a successful action is committed before the plan advances. The two harnesses therefore share a concern for reliable workspace mutation but realize it differently: one uses typed batch progress and per-file verification, whereas the other uses narrow tool exposure

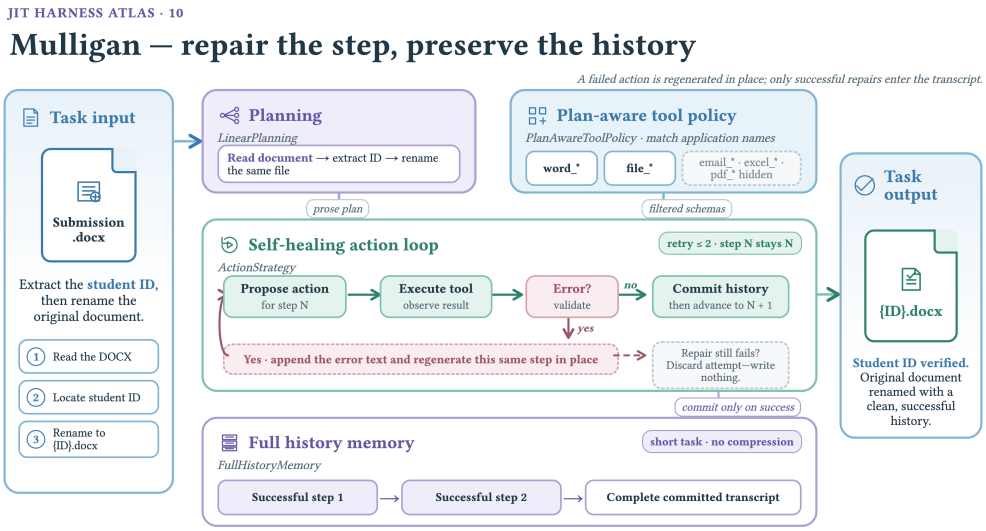

Figure D.9: **Mulligan: repair the step, preserve the history.** `PlanAwareToolPolicy` exposes only plan-relevant file and document tools; failed actions are regenerated at the same step, and `FullHistoryMemory` receives only the successful execution.

and transactional step repair. This task-dependent difference is precisely the behavior sought from JIT-Agent: reliability mechanisms are generated to fit the structure and failure surface of the request rather than imposed as one universal scaffold.