Harness Updating Is Not Harness Benefit: Disentangling Evolution Capabilities in Self-Evolving LLM Agents

Anonymous ACL submission

Abstract

LLM agents are increasingly deployed as systems built around editable external harnesses, including prompts, skills, memories and tools, that shape task execution without changing model parameters. Harness self-evolution adapts such agents by updating these harnesses from execution evidence. Yet it remains unclear whether a model's *base capability* in task-solving predicts its capabilities in harness self-evolution: which models produce useful harness updates, and which actually benefit from them? We analyze two harness self-evolution capabilities: (i) *harness-updating*, the capability to produce useful persistent harness updates from execution evidence; (ii) *harness-benefit*, the capability to benefit from updated harnesses during task solving. Our analysis reveals two findings. First, *harness-updating is flat in base capability*: models from different capability tiers produce harness updates that lead to surprisingly similar gains; even Qwen3.5-9B's updates yields gains comparable to those of Claude Opus 4.6. Second, *harness-benefit is non-monotonic in base capability*: weak-tier models benefit little from updated harnesses, mid-tier models benefit most, and strong-tier models benefit less than mid-tier. We trace low gains at the weak tier to two failure modes: weak-tier models may fail to activate relevant harness artifacts, or activate them but fail to follow them faithfully. These findings suggest investing capability budget in the task-solving agent rather than the evolver, and targeting harness invocation and long-horizon instruction following in agent training. Our source code is publicly available at link.

1 Introduction

Large language models (LLMs) (Radford et al., 2018; Touvron et al., 2023) have become a general-purpose foundation for language understanding (Hendrycks et al., 2020), reasoning (Wang et al., 2025), and task solving (Zhou et al., 2025).

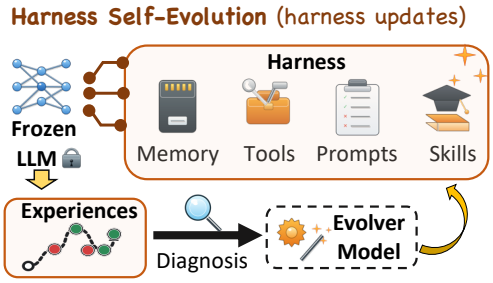

Figure 1: **Overview of harness self-evolution.**

Increasingly, they also power *agentic systems* that interact with external environments, call tools, operate software interfaces, and complete long-horizon tasks (Yang et al., 2024b; Merrill et al., 2026). In these settings, system behavior depends not only on the underlying model but also on an external *agent harness*: prompts (Wei et al., 2022), skills (Xia et al., 2026), memories (Yan et al., 2025), tools (Qin et al., 2024), etc., that shape how the model observes, reasons, acts, and recovers from errors. Improving an agentic system increasingly means refining not only the foundation model, but also the editable harness around it.

In current practice, harnesses are typically designed by hand. However, such manual design is brittle in deployment-time environments: task distributions shift, edge cases appear, and useful procedures are discovered only after the system interacts with real tasks. A natural response is to update the harness automatically from execution evidence: failures, feedback, trajectories, and successful procedures can be written back into the harness and reused on future tasks. We refer to this setting as *harness evolution* (Fig. 1): the model weights remain fixed, while the external agent harness is revised over time. Recent self-evolving agent methods (Madaan et al., 2023; Wu et al., 2025; Agrawal et al., 2026; Xia et al., 2026; Lin et al., 2026) pursue this approach across diverse harness components and have shown end-task im-

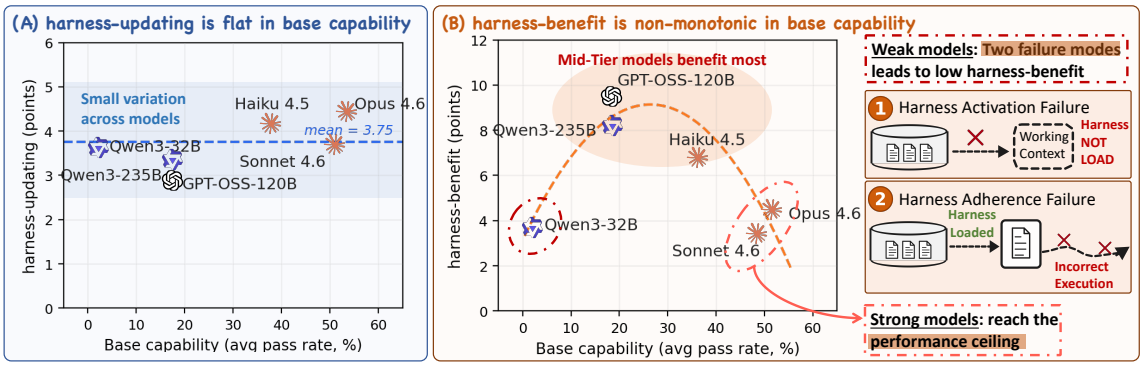

Figure 2: **Overview of our findings. (i)** *Harness-updating is flat in base capability*. Models across capability tiers produce harness updates that yield similar gains. **(ii)** *Harness-benefit is non-monotonic in base capability*. Mid-tier models benefit most, while weak-tier models benefit little due to failures in harness activation and adherence.

provements over non-evolving baselines. In these works, harness updates are typically produced by an LLM from execution evidence; we refer to this update role as the *evolver*.

Despite this rapid progress, evaluation of these methods still asks an end-to-end question: does a self-evolution method effectively improve agent performance? This question is important, but it hides the source of improvement. The gain may come from the *evolver* producing higher-quality harness updates, or from the task-solving agent using the updated harness more effectively during task solving. End-to-end scores cannot disentangle these contributions, leaving two practical questions open: *which models produce useful harness updates, and which models benefit most from them?*

To answer these questions, we analyze **two evolution capabilities** a model exercises in harness self-evolution across three agentic benchmarks and six LLMs: *harness-updating*, the capability to produce useful harness updates from execution evidence; and *harness-benefit*, the capability to benefit from updated harnesses during task solving. A model exercises harness-updating as the evolver, and harness-benefit as the task-solving agent. We conduct comprehensive experiments by pairing seven LLMs, spanning open-source and closed-source families across capability tiers, as agents and evolvers on three representative agentic benchmarks. Our analysis reveals two systematic decouplings between harness-evolution capabilities and *base capability*, namely, a model's task-solving capability without harness evolution (Fig. 2).

First, **harness-updating is flat in base capability**. When we fix the task-solving agent and vary the evolver model, models from different capability tiers produce harness updates that lead to surprisingly similar gains, and no evolver dominates across all substrates. Our case studies further show that even the Qwen3.5-9B evolver produces harness updates whose downstream gains match those of Claude Opus 4.6, despite a large gap in base capability.

Second, **harness-benefit is non-monotonic in base capability**. Models with mid-tier base-capability (e.g., GPT-OSS-120B) benefit most from updated harness, and strong-tier models (e.g., Claude Opus 4.6) reach the performance ceiling and benefit less. The weak-tier end, however, is not explained by the same ceiling argument: with the largest headroom above their base capability, models like Qwen3-32B might be expected to benefit most, yet they benefit the least. To understand why weak-tier models gain little, our in-depth analysis identifies two failure modes: (i) *Harness Activation Failure*: weak models often *fail to invoke* relevant harness artifacts (e.g., skills) during task-solving; and (ii) *Harness Adherence Failure*: even when the harness is loaded, weak models *fail to adhere* to it due to weak instruction-following over long-horizon tasks.

These findings translate into design guidance for harness self-evolution systems. *(i) Allocate capability budget to the agent, not the evolver*: the harness-updating gain is bounded above by a few points across all evolvers we test, so scaling up the evolver yields diminishing returns; the post-evolution score is determined primarily by the agent's harness-benefit. *(ii) Bake harness invocation into agent training*: weak-tier models often fail to load the harness at all (e.g., 25% for Qwen3-32B against $\approx 96\%$ for strong models), so harness invocation should be treated as a first-class learned skill. *(iii) Strengthen long-horizon instruction fol-*

lowing: even when loaded, weak-tier adherence decays across the trajectory over four times more steeply than strong agents', making sustained instruction following a second key target for downstream agent training.

2 Related Work

Harness engineering. An LLM agent combines a frozen backbone with an external *harness* that mediates reasoning, tool use, memory access, and environment interaction (Yao et al., 2022; Yang et al., 2024b; Ning et al., 2026). Recent work treats the harness as a first-class design object and differs mainly in artifact representation. *Prompts and instructions* encode guidance directly as natural language (Pan et al., 2026). *Skills* package reusable executable procedures into callable modules, as in Voyager (Wang et al., 2023). *Memory* stores cross-task experience for later retrieval, including compact strategy-level items (Ouyang et al., 2025). *Code* treats the harness as executable source that can be optimized by an agentic proposer (Lee et al., 2026). These works establish harnesses as editable agent state. Our work shifts the focus from how harnesses are represented to which models can update them effectively and which models can benefit from the updates.

Self evolution of LLM agents. Beyond *what* the harness contains, a complementary line asks how it is *updated* during task execution. Early systems operate at the episode or task level, through verbal self-reflection (Shinn et al., 2023) or iterative self-feedback (Madaan et al., 2023). More recent methods make persistent harness components the unit of self-evolution: GEPA (Agrawal et al., 2026) evolves prompts through trajectory-level reflection; SkillRL (Xia et al., 2026) recursively expands a skill library through reinforcement learning; ACE (Zhang et al., 2025) evolves contextual playbooks through structured generation, reflection, and curation; AWM (Wang et al., 2024) induces reusable workflows from successful trajectories; and EvolveR (Wu et al., 2025) connects offline strategy distillation with online retrieval. These methods show that writing execution experience into persistent harness components can improve downstream task performance. However, their evaluations typically report the end-to-end gain of a particular update procedure paired with a particular target agent on a particular substrate (Li et al., 2026; Jiang et al., 2026; Wei et al., 2025). Such scores conflate multiple sources of improvement: the model's base capability under the initial harness, the evolver's *harness-updating* capability in producing useful harness updates, and the model's *harness-benefit* capability to benefit from updated harnesses. Our work complements these methods with a controlled capability analysis: we vary the task-solving agent and the evolver independently, measure harness-updating and harness-benefit separately, and test whether either capability simply tracks base capability.

3 Harness-Evolution Capabilities

To explore the evolution capabilities in harness self-evolution, we consider harness self-evolution, which adapts an LLM agent by updating the external harness around a fixed model during task execution: the agent attempts a stream of tasks and the harness is updated based on the agent's execution evidence. In this section, we formalize the harness-evolution protocol and define two evolution capabilities: *harness-updating*, the ability to produce useful harness updates, and *harness-benefit*, the ability to benefit from updated harnesses.

3.1 Preliminaries: Harness State and Evolver

Agent harness. We use *agent harness* to denote the external, non-parametric context and infrastructure through which an LLM is deployed for task execution (Yao et al., 2022; Ning et al., 2026; Lee et al., 2026). Formally, at evolution step t, the LLM agent is defined as:

$$A_t = (f, H_t), \tag{1}$$

where f is the agent's model backbone and H_t is the harness state after step t. Following common harness self-evolution settings (Zhou et al., 2026; Lin et al., 2026), we keep f fixed and only update editable components of H_t (e.g., prompts, skills, memories), and fix other components such as tool interfaces and execution policies.

Evolver. An *evolver* is the update procedure that converts the agent's execution evidence into harness updates, where recent self-evolving agent systems (Yang et al., 2024a; Yuksekgonul et al., 2024; Xia et al., 2026; Agrawal et al., 2026) increasingly instantiate this procedure with LLM agents. Formally, given the previous harness H_{t-1} and the accumulated execution evidence $\mathcal{D}_t$ at step t, the evolver e proposes a harness update and applies it

to H_{t-1} to obtain the next harness:

$$\begin{aligned} \Delta H_t &= e(H_{t-1}, \mathcal{D}_t), \\ H_t &= \mathrm{Apply}(H_{t-1}, \Delta H_t). \end{aligned} \tag{2}$$

where Apply denotes the commit operation to apply ΔH_t to H_{t-1}.

3.2 Evolution Protocol

Following common harness self-evolution pipelines (Ouyang et al., 2025; Agrawal et al., 2026), we formalize the protocol as an iterative loop between task-solving and harness evolution. Starting from an initial harness H_0, the protocol iterates for T steps. At each step, the agent runs on a batch of tasks, collects execution evidence, and the evolver updates the harness for the next step. Formally, given an agent $A_{t-1} = (f, H_{t-1})$ and a task batch $\mathcal{X}_t$ at step t, A_{t-1} attempts to solve each task $x \in \mathcal{X}_t$ and output:

$$(\tau_{t,x}, y_{t,x}) = \mathrm{Solve}(A_{t-1}, x) \tag{3}$$

where $\tau_{t,x}$ is the execution trajectory and $y_{t,x}$ is the final output. The execution evidence $\mathcal{D}_t$ is then:

$$\mathcal{D}_t = \{(x, \tau_{t,x}, y_{t,x}) : x \in \mathcal{X}_t\}. \tag{4}$$

The evolver produces the updated harness H_t from H_{t-1} and $\mathcal{D}_t$ as in Eq. 2, yielding the next agent $A_t = (f, H_t)$. This loop repeats for T steps, producing the final harness H_T.

3.3 Capability Metrics

To analyze which models produce useful harness updates and which models benefit from them, we formally define three metrics to measure both harness-evolution capabilities (i.e., *harness-updating* and *harness-benefit*) along with each model's *base capability*.

Base Capability and Evolution Gain. Given a task set $\mathcal{X} = \bigcup_{t=1}^{T} \mathcal{X}_t$, the *base capability* of a model f is the task-solving performance of the initial agent $A_0 = (f, H_0)$ on $\mathcal{X}$:

$$M_{\text{base}}(f) = J_{\mathcal{X}}(f, H_0), \tag{5}$$

where $J_{\mathcal{X}}(f, H)$ is the scoring function that measures the performance of agent (f, H) on $\mathcal{X}$.

Given a model f and an evolver e, let $H_T^{(f,e)}$ denote the final harness produced after evolution with f as the agent and e as the evolver for T steps starting from H_0. We further define the *pairwise evolution gain* as the improvement of a specific agent–evolver pairing (f, e) over the agent's task-solving performance before evolution:

$$\Delta(f, e) = J_{\mathcal{X}}(f, H_T^{(f,e)}) - M_{\text{base}}(f). \tag{6}$$

Harness-updating Capability. The *harness-updating capability* of an evolver e is its ability to produce harness updates that improve agents' task-solving. Formally, this is defined as the mean pairwise gain across an anchor agent set $\mathcal{F}^\star$:

$$\Delta_{\text{update}}(e) = \frac{1}{|\mathcal{F}^\star|} \sum_{f \in \mathcal{F}^\star} \Delta(f, e). \tag{7}$$

Harness-benefit Capability. The *harness-benefit capability* of a model f is its maximum gain in task-solving performance from harness self-evolution. In practice, we estimate this as the maximum pairwise gain across a fixed anchor evolver set $\mathcal{E}^\star$:

$$\Delta_{\text{benefit}}(f) = \max_{e \in \mathcal{E}^\star} \Delta(f, e). \tag{8}$$

4 Experiments

In this section, we empirically analyze the two harness-evolution capabilities defined in Sec. 3. We present the evolver-side analysis of harness-updating capability in Sec. 4.2, and the agent-side analysis of harness-benefit capability in Sec. 4.3

4.1 Experimental Setup

Datasets. We evaluate on three representative agentic benchmarks: SWE-bench Verified (SWE) (Jimenez et al., 2024) for software engineering, MCP-Atlas (MCP) (Bandi et al., 2026) for tool use over real MCP servers, and SkillsBench (SB) (Li et al., 2026) for skill-based execution across diverse domains. More details are in Appendix A.1.

Models. We use seven LLM backbones, spanning open-source and closed-source families across capability tiers. For the agent-side analysis, we use six models: Claude Opus 4.6 (Anthropic, 2026a), Claude Sonnet 4.6 (Anthropic, 2026b), Claude Haiku 4.5 (Anthropic, 2025), Qwen3-235B-A22B and Qwen3-32B (Yang et al., 2025), and GPT-OSS-120B (Agarwal et al., 2025). For the evolver-side analysis, we use the same six models plus Qwen3.5-9B (Qwen, 2026), the smallest model in this paper, to test whether a substantially smaller open model can still produce useful harness updates.

Evaluation Protocol. We report the four metrics defined in §3.3: base capability $M_{\text{base}}(f_\theta)$, harness-updating gain $\Delta_{\text{update}}(e)$, harness-use capability

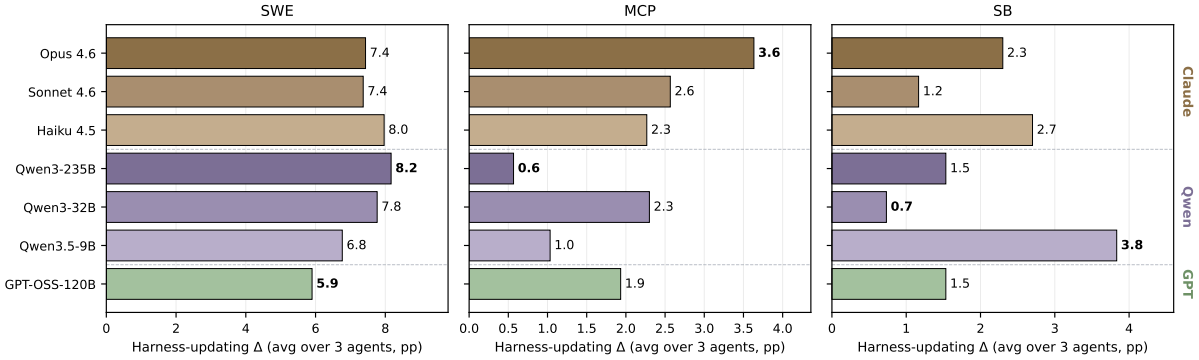

Figure 3: **Harness-updating capability (Δ_{update}) of each evolver.** Evolvers are grouped by model family (Claude, Qwen, GPT-OSS). The best and worst evolver, marked in bold within each panel, change with the benchmark.

$M_{\text{benefit}}(f_\theta)$, and the harness-use gain $\Delta_{\text{benefit}}(f_\theta)$. We use pass rate as the primary metric for $J_{\mathcal{X}}$ on three benchmarks. We consider an in-situ evaluation setting: each task in $\mathcal{X}_t$ is scored under H_{t-1} before its evidence is used to produce H_t. The final results are reported by aggregating per-task scores over the task stream. Further details are in Appendix A.3.

Implementation Details. We instantiate the evolution protocol in Sec. 3.2 with a fixed solve-evolve loop. For a fair comparison, we fix the prompt template for both agents and evolvers, along with the trajectory window, across all agent-evolver pairs; only the LLM backbone varies. All pairs within a benchmark start from the same initial harness H_0 and task stream $\mathcal{X}$, share the same evolution budget β and per-task turn limit. The evolvable components are skills for SWE-bench Verified and SkillsBench, and skills, prompts, and memories for MCP-Atlas. Other details such as evolution frameworks and prompt templates are in Appendix A.4.

4.2 Evolver-side Analysis

To understand how harness-updating capability varies across LLMs, we fix the task-solving agents and vary the evolver over the seven LLMs in Sec. 4.1. Specifically, we use three representative LLMs, Opus 4.6, Sonnet 4.6, and Qwen3-235B, as the anchor agents in $\mathcal{F}^\star$. For each evolver e, we report $\Delta_{\text{update}}(e)$, defined in Sec. 3.3, across the three benchmarks in Fig. 3. Full pass-rate results for all agent-evolver pairings are in Appendix B.1.

Finding 1: Harness-updating is flat in base capability. Fig. 3 shows two patterns: **(i)** *The spread of Δ_{update} across evolvers is narrow*. The gap between the best and worst evolver is at most 3.1 percentage points (pp) on any benchmark, and no model wins across benchmarks. Qwen3-235B illustrates this reshuffling: it leads on SWE (8.2 pp) but ranks last on MCP (0.6 pp). **(ii)** *Model scale is not predictive*. The smallest evolver, Qwen3.5-9B, posts the highest gain on SB (3.8 pp), exceeding both Opus 4.6 (2.3 pp) and Qwen3-235B (1.5 pp).

Case Study: the 9B evolver writes a skill procedurally isomorphic to Opus's. To understand the mechanism behind these comparable gains, we examine a representative SkillsBench task `flink-query` in detail. We fix the task-solving agent backbone at Opus 4.6 and compare its trajectories under three evolver conditions (Fig. 4): no evolver, Qwen3.5-9B as evolver, and Opus 4.6 as evolver. We observe that without an evolved skill, the agent omits the FINISH-event filter and fail to solve this task (scores 0.67); with a skill injected by either Qwen3.5-9B or Opus 4.6, the same agent solves the task successfully (score 1.0). Inspecting the two skills, we find they are procedurally isomorphic, prescribing the same sequence of steps and differing only in surface details of implementation and verbosity. The 9B open-source evolver thus reaches the same procedural content as the frontier evolver. Full details of the skill contents and analysis are in Appendix B.2.

Finding 2: Post-evolution score is dominated by models' base capability, not evolver identity. To understand the relative contribution of task-solving agents and evolvers to post-evolution performance, we plot the task-solving performances of three LLMs (Opus 4.6, Sonnet 4.6, Qwen3-235B) in $\mathcal{F}^\star$ under the updated harnesses from seven LLMs in Sec. 4.1 as the evolvers against each agents' base capability. Results on MCP-Atlas are shown in Fig. 5. We observe: **(i)** *Within-agent spread is much smaller than between-agent gap.* The within-

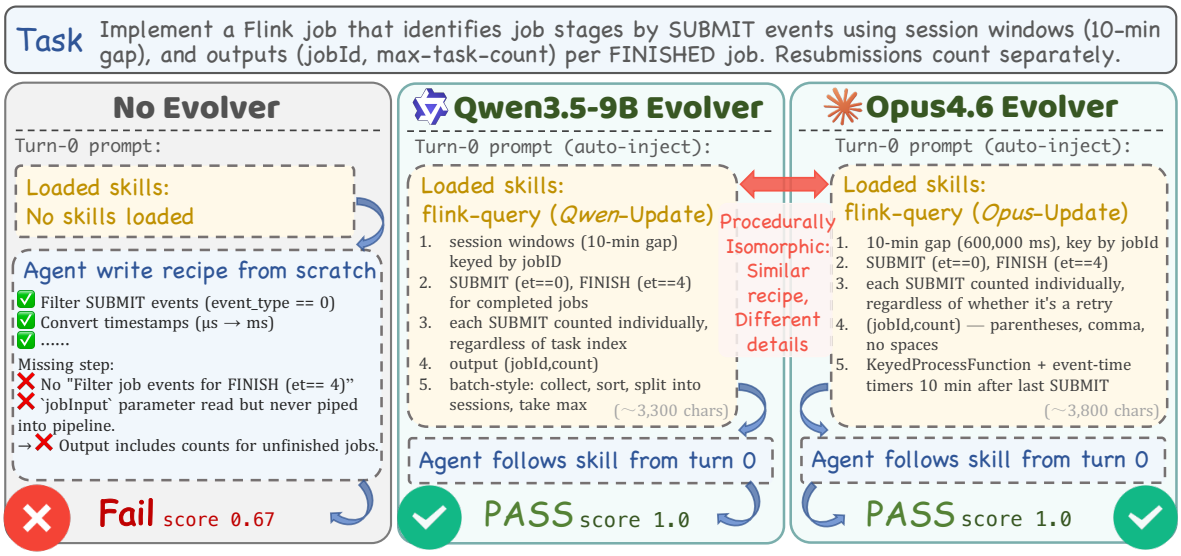

Figure 4: **Comparison of Harness updated by Qwen3.5-9B and Claude Opus 4.6.** Trajectory of an agent solving the SkillsBench `flink-query` task under three turn-0 conditions: no evolver (left, score 0.67), with a skill evolved by Qwen3.5-9B injected (center, score 1.0), and with a skill evolved by Opus 4.6 injected (right, score 1.0).

agent spread across seven evolvers is at most 5.1 pp (Qwen3-235B), small against the 36.0 pp gap between the Opus and Qwen3-235B base capabilities. The pattern persists on SWE and SB. **(ii)** *Extreme pairing still favors strong agents.* Even pairing the weakest anchor agent with its best-performing evolver against the strongest anchor agent with its worst-performing evolver, the strong agent still leads by 18.6 to 35.2 pp on every benchmark. Both patterns also persist on SWE and SB datasets (Appendix B.3). Post-evolution performance is therefore bottlenecked on the agent side, not the evolver side, motivating the agent-side analysis in Sec. 4.3.

Take-away. Allocate capability budget to the agent, not the evolver: (i) Δ_{update} varies by at most 3.1 pp across evolvers on any benchmark, and (ii) post-evolution score is dominated by the agent's base capability.

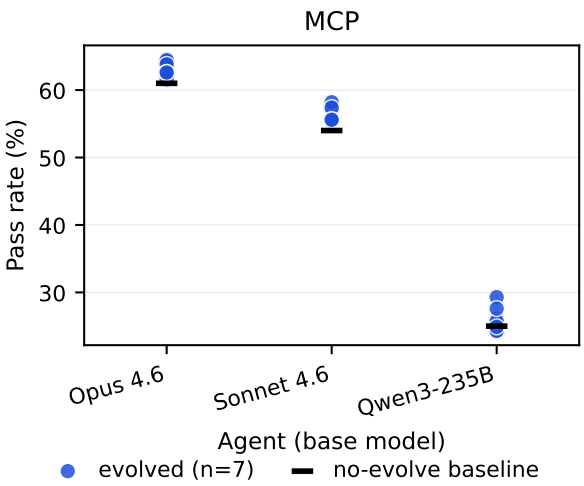

Figure 5: **MCP post-evolution scores**: for each anchor agent every blue dot is one of seven evolved scores and the black tick is the no-evolve baseline. Within-agent variation across evolvers is small relative to between-agent variation in base capability.

4.3 Agent-side Analysis

To understand how *harness-benefit* capability varies across LLMs, we fix the evolvers and vary the task-solving agent over the LLM backbones in Sec. 4.1: Opus 4.6, Sonnet 4.6, Haiku 4.5, Qwen3-235B, Qwen3-32B, and GPT-OSS-120B. We use Opus 4.6, Sonnet 4.6, and Qwen3-235B as the three anchor evolvers, denoted by $\mathcal{E}^\star$. For each agent f, we report $\Delta_{\text{benefit}}(f)$ as defined in Sec. 3.3.

Finding 1: Δ_{benefit} is non-monotonic in base capability. As shown in Tab. 1 and Fig. 6, Δ_{benefit} does not increase monotonically with base capability. On SWE, the gain peaks at Qwen3-235B (19.3 pp), while the weaker Qwen3-32B gains only 4.4 pp and the stronger Opus 4.6 gains only 2.6 pp. On MCP, the peak shifts to GPT-OSS-120B (7.0 pp), again with lower gains at both ends of the base-capability scale. This pattern has different explanations at the two ends of the capability scale. At the high-capability end, smaller gains are consistent with a ceiling effect: strong models already solve many tasks under the initial harness, leaving less room for further improvement. However, at the

Table 1: **Base pass rate (%) and Harness-benefit Δ_{benefit} (pp) per agent per benchmark.** Agents are sorted by SWE base ascending. Bold marks the column maximum of Δ_{benefit}.

	SWE		MCP		SB	
Model	Base	Δ	Base	Δ	Base	Δ
Qwen3-32B	3.6	4.4	3.6	1.0	0.0	5.8
Qwen3-235B	20.7	**19.3**	25.0	4.3	4.7	1.1
GPT-OSS-120B	26.2	15.8	28.0	**7.0**	0.0	7.0
Haiku 4.5	66.0	2.4	42.4	3.6	5.8	**15.1**
Sonnet 4.6	73.2	2.8	54.0	3.2	24.4	3.5
Opus 4.6	74.2	2.6	61.0	3.6	25.6	5.8

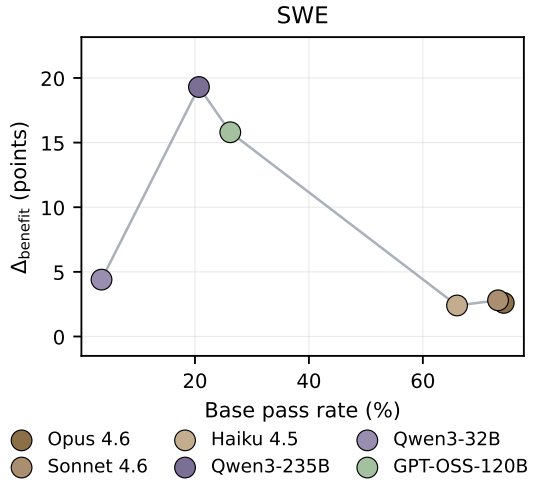

Figure 6: **Δ_{benefit} vs base pass rate on SWE for six agents**; line connects points in base-ascending order. MCP and SB analogues in Appendix C.2.

low-capability end, smaller gains reflect a different bottleneck, which we diagnose next.

Finding 2: Weak-tier models derive low Δ_{benefit} due to two failure modes. To understand why the weak-tier models with low base capabilities receive low Δ_{benefit}, we conduct an in-depth analysis on SkillsBench and identify two complementary failure modes: *harness activation* and *harness adherence*, which is illustrated in Fig. 7.

The first mode is *harness activation failure*: weak-tier models often fail to bring relevant harness artifacts, such as skills, into their working context. To quantify this on SkillsBench, we report each agent's *skill-load rate (SLR)*, the fraction of its trajectories in which it actively loads at least one skill into its context. Tab. 2 shows that the skill-load rate is near ceiling for Opus 4.6, Sonnet 4.6, and Qwen3-235B (0.957–0.961), but drops to 0.446 for GPT-OSS-120B and 0.251 for Qwen3-32B. The left panel of Fig. 7 illustrates this activation failure. Specifically, Qwen3-32B identifies the relevant skill, but embeds the loading request inside a broader action rather than issuing it as a standalone skill-loading action. The SkillsBench environment therefore does not treat it as a valid load request, so the skill body never enters context.

The second mode is *harness adherence failure*: even when relevant harness artifacts are loaded, weak-tier models often fail to follow their guidance faithfully during task solving. We quantify this failure with the *Harness-Following Rate* (HFR), computed over trajectories in which at least one skill is loaded. For each skill-loaded task-solving trajectory, an LLM judge determines whether the task-solving model follows the loaded skill's guidance. HFR is the fraction of skill-loaded trajectories judged as following the skill. Appendix C.3 provides details of the judge pipeline. Tab. 2 reports HFR together with two complementary metrics: *SLR*, which measures harness activation, and *pass-when-loaded (LPR)*, which measures the pass rate among that model's skill-loaded trajectories. We observe two patterns. **(i)** *Strong-tier models exhibit much higher harness adherence than weak-tier models.* Opus 4.6 reaches an HFR of 0.757, while Qwen3-32B reaches only 0.142. **(ii)** *Loading the harness is not sufficient for benefiting from it.* Qwen3-235B provides the cleanest separation between activation and adherence: its skill-load rate is 0.961, nearly identical to Opus 4.6, yet its HFR is only 0.350. Its pass-when-loaded rate mirrors this gap, at 0.022 compared with 0.177 for Opus 4.6. The `pg-essay-to-audiobook` case in the right panel of Fig. 7 illustrates this adherence failure. Qwen3-32B successfully loads the procedural skill, but treats the guidance as a ready-made script rather than a procedure to follow. After the first attempt fails, it terminates instead of trying the alternative steps prescribed by the skill. More details of the analysis are in Appendix C.1.

Diagnosis: Weak instruction following over long-horizon execution. To test whether harness adherence degrades as a trajectory unfolds, we conduct a phase-level adherence analysis. An LLM judge assigns a 0–1 adherence score at different execution stages, with details provided in Appendix C.4. We use Qwen3-32B, GPT-OSS-120B, and Opus 4.6 as representative weak-, mid-, and strong-tier models, respectively. Tab. 3 reports three representative phases, after harness loading, at the trajectory midpoint, and at final validation, with scores averaged over judged trajectories for each model. We observe that Qwen3-32B drops sharply from 0.52 af-

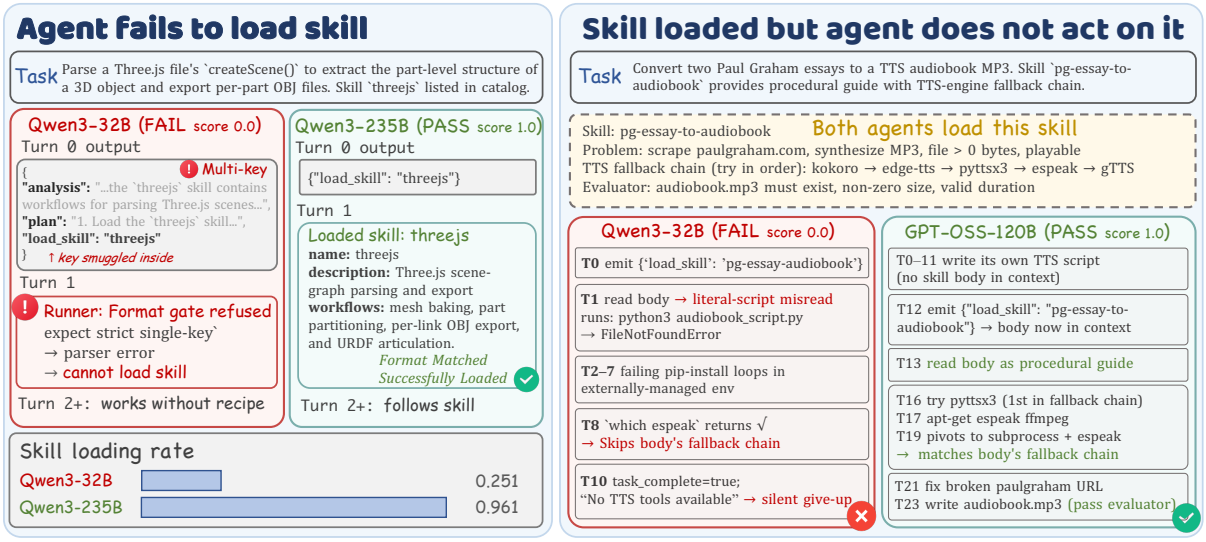

Figure 7: **Two failure modes for Qwen3-32B on SkillsBench.** Left (`threejs`): *Harness activation failure*. Harness invocation fails because a multi-key turn-0 action is rejected by the format gate, so the skill never reaches context. Right (`pg-essay-to-audiobook`): *Harness adherence failure*. the skill is loaded but read as a literal script; the agent skips the fallback chain and silently emits `task_complete`.

Table 2: **Per-model activation, adherence, and outcome metrics on SkillsBench.** *SLR*: fraction of a model's trajectories in which at least one skill is loaded into context. *HFR*: fraction of skill-loaded trajectories judged as following the loaded skill's guidance. *LPR*: pass rate among the model's skill-loaded trajectories. Models are sorted by base capability on SkillsBench.

Model	SLR	HFR	LPR
Qwen3-32B	*0.251*	*0.142*	0.023
GPT-OSS-120B	0.446	0.442	0.040
Haiku 4.5	0.794	0.600	0.099
Qwen3-235B	**0.961**	0.350	0.022
Sonnet 4.6	0.959	0.730	0.145
Opus 4.6	0.957	**0.757**	**0.177**

ter harness loading to 0.13 at final validation, while GPT-OSS-120B drops more moderately from 0.67 to 0.43. In contrast, Opus 4.6 remains stable, from 0.89 to 0.80. This graded drift suggests a long-horizon instruction-following bottleneck: weaker models progressively lose adherence as the trajectory unfolds, rather than merely misreading the harness at load time.

Take-away. Train agents for harness use along two axes will help evolution: (i) *Bake harness invocation into training.* Targets low harness-load rate at the weak end (0.25 for Qwen3-32B against ≈ 0.96 for strong agents); (ii) *Strengthen long-horizon instruction following.* Targets the trajectory-tail drift (weak-agent adherence drifts -0.34 from load to final, three times more than strong agents).

Table 3: **Models' Per-phase adherence score by base-capability tier.** Qwen3-32B, GPT-OSS-120B and Claude Opus 4.6 are used as the representative weak-tier, mid-tier and strong-tier models.

Trajectory Phase	Qwen3-32B (weak)	GPT-OSS (mid)	Opus 4.6 (strong)
Harness loaded	0.52	0.67	**0.89**
Mid turn	0.22	0.48	**0.79**
Final turn	0.13	0.43	**0.80**
drift (load → final)	-0.39	-0.24	**-0.09**

5 Conclusion

We analyze harness self-evolution by decomposing it into two model capabilities distinct from base capability: *harness-updating*, the capability to produce harness updates, and *harness-benefit*, the capability to benefit from updated harnesses during task solving. Across seven LLMs on three benchmarks, harness-updating is flat in base capability: models from various capability tiers produce updates that yield similar gains, and even Qwen3.5-9B evolver induces gains comparable to Claude Opus 4.6. In contrast, harness-benefit is non-monotonic in base capability: weak-tier models gain little, traced to two failure modes, failing to activate relevant harness artifacts or failing to follow them faithfully once activated. These findings motivate investing capability budget in the agent rather than the evolver, and targeting agent training at harness invocation and long-horizon instruction following.

6 Limitations

Our study focuses on harness self-evolution, where model weights remain fixed and adaptation occurs through updates to external harness artifacts. We do not evaluate parametric fine-tuning, reinforcement learning of model weights, or hybrid adaptation methods that combine weight updates with harness updates. Our model set is representative but not exhaustive: we include open-source and closed-source models across multiple capability tiers, but a broader model grid would further clarify how harness-updating and harness-benefit vary with model family, scale, training recipe, and deployment cost.

7 Ethics Statement

This work studies LLM agents that update persistent external harnesses from execution evidence. All experiments are conducted on benchmark tasks, and we do not collect or process private user data. However, harness self-evolution raises broader deployment concerns because updated harnesses may persist across future tasks. Incorrect lessons, unsafe tool-use rules, biased instructions, or sensitive information could be written into the harness and reused by later agents. In our evaluation, harness updates are logged, and evolvers are constrained from modifying evaluation scripts or updating model weights. These controls make the benchmark setting auditable, but they do not by themselves guarantee safety in open deployments. Real-world harness self-evolution systems should treat privacy, consent for data retention, update reversibility, auditability, and human oversight as first-class design requirements.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. 2025. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2026. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*.

Anthropic. 2025. Claude haiku 4.5 system card.

Anthropic. 2026a. Claude opus 4.6 system card.

Anthropic. 2026b. Claude sonnet 4.6 system card.

Chaithanya Bandi, Ben Hertzberg, Geobio Boo, Tejas Polakam, Jeff Da, Sami Hassaan, Manasi Sharma, Andrew Park, Ernesto Hernandez, Dan Rambado, et al. 2026. Mcp-atlas: A large-scale benchmark for tool-use competency with real mcp servers. *arXiv preprint arXiv:2602.00933*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

Sihang Jiang, Lipeng Ma, Zhonghua Hong, Keyi Wang, Zhiyu Lu, Shisong Chen, Jinghao Zhang, Tianjun Pan, Weijia Zhou, Jiaqing Liang, et al. 2026. Sea-eval: A benchmark for evaluating self-evolving agents beyond episodic assessment. *arXiv preprint arXiv:2604.08988*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. 2026. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*.

Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, et al. 2026. Skillsbench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670*.

Minhua Lin, Hanqing Lu, Zhan Shi, Bing He, Rui Mao, Zhiwei Zhang, Zongyu Wu, Xianfeng Tang, Hui Liu, Zhenwei Dai, et al. 2026. Position: Agentic evolution is the path to evolving llms. *arXiv preprint arXiv:2602.00359*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*.

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

Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. 2025. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*.

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

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2024. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *International Conference on Learning Representations*, volume 2024, pages 9695–9717.

Team Qwen. 2026. Qwen3.5: Accelerating productivity with native multimodal agents.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. 2018. Improving language understanding by generative pre-training.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Ruonan Wang, Runxi Wang, Yunwen Shen, Chengfeng Wu, Qinglin Zhou, and Rohitash Chandra. 2025. Evaluation of llms for mathematical problem solving. *arXiv preprint arXiv:2506.00309*.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024. Agent workflow memory. *arXiv preprint arXiv:2409.07429*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Tianxin Wei, Noveen Sachdeva, Benjamin Coleman, Zhankui He, Yuanchen Bei, Xuying Ning, Mengting Ai, Yunzhe Li, Jingrui He, Ed H Chi, et al. 2025. Evo-memory: Benchmarking llm agent test-time learning with self-evolving memory. *arXiv preprint arXiv:2511.20857*.

Rong Wu, Xiaoman Wang, Jianbiao Mei, Pinlong Cai, Daocheng Fu, Cheng Yang, Licheng Wen, Xuemeng Yang, Yufan Shen, Yuxin Wang, et al. 2025. Evolver: Self-evolving llm agents through an experience-driven lifecycle. *arXiv preprint arXiv:2510.16079*.

Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. 2026. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning. *arXiv preprint arXiv:2602.08234*.

Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z Pan, et al. 2025. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. *arXiv preprint arXiv:2508.19828*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. 2024a. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pages 12028–12068.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024b. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. 2024. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. 2025. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*.

Chenyu Zhou, Huacan Chai, Wenteng Chen, Zihan Guo, Rong Shan, Yuanyi Song, Tianyi Xu, Yingxuan Yang, Aofan Yu, Weiming Zhang, et al. 2026. Externalization in llm agents: A unified review of memory, skills, protocols and harness engineering. *arXiv preprint arXiv:2604.08224*.

Xiyuan Zhou, Xinlei Wang, Yirui He, Yang Wu, Ruixi Zou, Yuheng Cheng, Yulu Xie, Wenxuan Liu, Huan Zhao, Yan Xu, et al. 2025. Engibench: A benchmark for evaluating large language models on engineering problem solving. *arXiv preprint arXiv:2509.17677*.

A Experimental Setup Details

A.1 Dataset Details

We evaluate on three representative agentic benchmarks that cover complementary agent capabilities: long-horizon code repair with SWE-bench Verified, multi-server tool orchestration with MCP-Atlas, and skill-based execution across diverse domains with SkillsBench. Dataset statistics are in Tab. 4:

- **SWE-bench Verified (Jimenez et al., 2024).** This is a human-validated subset of SWE-bench containing 500 tasks drawn from real GitHub issues across 12 popular Python repositories. Each task provides a codebase snapshot and an issue description; the solver must produce a patch that resolves the issue. A task passes if its patch satisfies the hidden test suite associated with the issue. We use the full 500-task subset.

- **MCP-Atlas (Bandi et al., 2026).** This is a benchmark for multi-server tool-use competency over real Model Context Protocol servers. Each task is a natural-language request whose completion requires the solver to identify and orchestrate 3–6 tool calls across 36 real MCP servers exposing 220 tools. Scoring uses a claims-based rubric that awards credit per factual claim satisfied in the final answer; we report pass rate as the fraction of tasks for which all claims are satisfied. We use the 500-task public subset released by the authors.

- **SkillsBench (Li et al., 2026).** This is a 86-task benchmark spanning 11 domains (e.g., software, data analysis, document processing, audio synthesis) with a deterministic per-task verifier. Each task provides workspace files and a natural-language instruction; the agent must complete the task using the workspace and any skills available in its harness. The native benchmark ships with curated skills, but in our setup the no-evolution baseline starts from an empty skill set, and evolved cells use only the skills produced by the evolver from earlier in-situ tasks.

A.2 Models

We use seven LLM backbones, spanning open-source and closed-source families across capability tiers. The closed-source models are Claude Opus 4.6 (Anthropic, 2026a), Claude Sonnet 4.6 (Anthropic, 2026b), and Claude Haiku 4.5 (Anthropic, 2025). The open-source models are Qwen3-235B-A22B and Qwen3-32B (Yang et al., 2025), Qwen3.5-9B (Qwen, 2026), and GPT-OSS-120B (Agarwal et al., 2025).

For the agent-side analysis, we use the six LLMs (Opus 4.6, Sonnet 4.6, Haiku 4.5, Qwen3-235B-A22B, Qwen3-32B, GPT-OSS-120B) as task-solving agent backbones. For the evolver-side analysis, we use all seven models, including Qwen3.5-9B (the smallest model in our paper), to test whether a substantially smaller open model can still produce useful harness updates. Across all experiments we query each model through its official API or inference endpoint; no model weights are updated during evolution.

A.3 Metrics

Scoring function. For all four metrics in §3.3, we use pass rate as the scoring function $J_{\mathcal{X}}$: each task $x \in \mathcal{X}$ receives a per-task score from the benchmark's grader, and $J_{\mathcal{X}}$ is the mean over $\mathcal{X}$. Pass rates and average scores are reported in percent; gains are reported in percentage points.

Per-benchmark scoring. The scoring function $J_{\mathcal{X}}$ instantiates the standard grading procedure of each benchmark:

- **SWE-bench Verified** (Jimenez et al., 2024): per-task binary resolved score (1 if the submitted patch passes the designated fail-to-pass and pass-to-pass test suite, 0 otherwise). The mean over tasks is the standard pass rate.

- **MCP-Atlas** (Bandi et al., 2026): per-task claim-fulfillment score in $[0, 1]$, computed as the fraction of reference claims satisfied by the agent's final answer. We report both the strict pass rate (mean of binarized per-task scores) and the average claim-fulfillment score (mean of continuous per-task scores).

- **SkillsBench** (Li et al., 2026): per-task binary score averaged over 5 trials following Terminal-Bench (Merrill et al., 2026). We report the average score (mean across tasks and trials) as the primary metric.

For each benchmark, $J_{\mathcal{X}}$ in the metric definitions of §3.3 refers to the mean of these per-task scores aggregated over the task stream.

In-situ evaluation. We evaluate in an in-situ setting: the same task stream $\mathcal{X} = \bigcup_{t=1}^{T} \mathcal{X}_t$ that drives evolution also serves as the evaluation set. Concretely, at step t, each task $x \in \mathcal{X}_t$ is scored under

Table 4: **Dataset statistics.** N_b is the number of tasks; the rightmost column lists the static resources each task exposes to the agent.

Substrate	N_b	#Domains	Resources per task
SWE-bench Verified	500	12 repositories	Codebase snapshot, issue description, hidden test suite
MCP-Atlas	500	36 MCP servers	220 tools (shared across servers); 3–6 tool calls required per task
SkillsBench	86	11 task domains	Workspace files, deterministic verifier

the harness H_{t-1} at the time of its attempt; the score is locked in before $(\tau_{t,x}, y_{t,x})$ enters $\mathcal{D}_t$ and produces H_t. The pass rate of any individual task is thus not influenced by harness updates derived from that task itself.

A.4 Implementation Details

Evolvable Harness Component. The evolvable components differ by benchmark: SWE-Bench Verified and SkillsBench permit edits to the `skills/` directory only, while MCP-Atlas additionally permits edits to `prompts/system.md` and the `memory/` jsonl files. These permissions are passed to the same evolver system prompt as boolean flags; the prompt itself does not change.

Task-solving Agent Prompt Templates. Within each benchmark, all task-solving agents use the same system prompt; only the task-specific user prompt varies across tasks. For SWE-bench Verified, the solver prompt (Tab. 7) is an 828-byte procedural guide that scopes the agent to GitHub-issue patching and encourages minimal, focused edits. For MCP-Atlas, the solver prompt (Tab. 8) is a 1,309-byte API-agent guide that instructs the agent to satisfy task queries through tool calls and not ask the user for clarification. For SkillsBench, we follow the original setting (Li et al., 2026) to use no system prompt for the task-solving agent.

Evolver Prompt Template. The evolver system prompt is fixed across all benchmarks and LLM backbones, as shown in Tab. 9. The user message follows a fixed wrapper; the only changing fields are the cycle index, the workspace-permission block, and the canonicalized evidence payload. Write permissions are determined by the benchmark: SWE-bench Verified and SkillsBench allow edits to `skills/`, while MCP-Atlas allows edits to `prompts/` and `skills/` and appends to `memory/`. The `tools/` directory is read-only for all benchmarks.

Evolver	Opus 4.6	Sonnet 4.6	Qwen3-235B	Δ_{update}
SWE				
NONE	74.2	73.2	20.7	—
Opus 4.6	76.4	76.0	38.0	7.4
Sonnet 4.6	76.8	75.6	37.8	7.4
Haiku 4.5	77.8	74.8	39.4	8.0
Qwen3-235B	76.6	76.0	40.0	**8.2**
Qwen3-32B	76.2	75.4	39.8	7.8
Qwen3.5-9B	76.4	73.2	38.8	6.8
GPT-OSS-120B	75.2	75.6	35.0	*5.9*
MCP				
NONE	61.0	54.0	25.0	—
Opus 4.6	64.4	57.2	29.3	**3.6**
Sonnet 4.6	64.6	57.0	26.1	2.6
Haiku 4.5	64.4	58.2	24.2	2.3
Qwen3-235B	61.6	55.8	24.3	*0.6*
Qwen3-32B	63.8	57.4	25.7	2.3
Qwen3.5-9B	62.6	55.6	24.9	1.0
GPT-OSS-120B	62.6	55.6	27.6	1.9
SB				
NONE	25.6	24.4	4.7	—
Opus 4.6	30.2	27.9	3.5	2.3
Sonnet 4.6	29.1	25.6	3.5	1.2
Haiku 4.5	31.4	25.6	5.8	2.7
Qwen3-235B	31.4	22.1	5.8	1.5
Qwen3-32B	30.2	22.1	4.6	*0.7*
Qwen3.5-9B	26.7	31.4	8.1	**3.8**
GPT-OSS-120B	31.4	22.1	5.8	1.5

Table 5: **Pass rate (%) of each anchor agent under each evolver, across the three benchmarks.** The NONE row is the no-evolution baseline. Δ_{update} is the mean improvement of the three anchors over NONE, in percentage points. **Bold**: per-benchmark best evolver. *Italic*: per-benchmark worst evolver.

B Evolver-side Analysis Details in Sec. 4.2

B.1 Additional Results for Finding 1

Table 5 reports the pass rate of each anchor agent (Opus 4.6, Sonnet 4.6, Qwen3-235B) under each evolver on the three benchmarks, alongside the resulting Δ_{update}. These are the per-cell numbers underlying the bars in Figure 3.

B.2 More Details of the Case Study

We elaborate on the case study from Sec. 4.2. We examine the SkillsBench task `flink-query` with the agent backbone fixed at Opus 4.6, comparing its trajectories under three evolver conditions (Fig. 4): no evolver, Qwen3.5-9B as evolver, and Opus 4.6

as evolver. Without an evolver, the agent omits the FINISH-event filter and scores 0.67; with either evolved skill injected at turn 0, the same agent solves the task (score 1.0).

Inspecting the two evolved skills, we find that they encode the same five problem-solving steps:

- Filter SUBMIT events.
- Filter FINISH events.
- Count each SUBMIT separately.
- Emit `(jobId, count)`.
- Apply a 10-minute session window.

The two skills differ only in implementation surface details: Qwen3.5-9B specifies the gap as 10 minutes with manual batch sessionization, while Opus 4.6 specifies 10 minutes with a `KeyedProcessFunction`. Despite these surface differences, both skills yield identical downstream pass rates (1.0) when injected into the same Opus 4.6 agent.

B.3 Additional Results for Finding 2

This subsection extends Observation 2 in Sec. 4.2 to the other two benchmarks, SWE-bench Verified and SkillsBench. We observe the same two patterns: within-agent variation across evolvers remains smaller than between-agent differences in base capability, and even extreme agent-evolver pairings still favor the stronger agent.

Within-agent spread versus between-agent gap. Fig. 8 extends the post-evolution score view of Fig. 5 to SWE and SB. On SWE, the largest within-agent spread across seven evolvers is 5.0 pp, attained by Qwen3-235B. On SB, the largest spread is 9.3 pp, attained by Sonnet 4.6, whose evolved scores range from 22.1% to 31.4%. By comparison, the base-capability gap between Opus 4.6 and Qwen3-235B is 53.5 pp on SWE and 20.9 pp on SB. Thus, the between-agent gap exceeds the within-agent spread by a factor of 11 on SWE and 2.2 on SB. SB is the tightest of the three benchmarks, but the same inequality still holds.

Extreme pairings across benchmarks. Tab. 6 compares the weakest anchor agent W paired with its best-performing evolver against the strongest anchor agent S paired with its worst-performing evolver, separately for each benchmark. Even under this unfavorable comparison for the strong agent, S still outperforms W by 18.6 to 35.2 pp on every benchmark. On SB, the same evolver, Qwen3.5-9B, appears on both sides of the comparison, because it is the best evolver for Qwen3-235B and the worst evolver for Opus 4.6. This reinforces the main conclusion that post-evolution performance is dominated more by the task-solving agent than by evolver identity.

Table 6: **Extreme agent-evolver pairings across benchmarks.** The weakest anchor agent W is paired with its best-performing evolver, while the strongest anchor agent S is paired with its worst-performing evolver. Scores are pass rates (%), and gaps are reported in percentage points (pp).

	SWE	MCP	SB
weak agent W	Q3-235B	Q3-235B	Q3-235B
best evolver for W	Q3-235B	Opus	Q3.5-9B
score(W, best)	40.0	29.3	8.1
strong agent S	Opus	Opus	Opus
worst evolver for S	GPT-OSS	Q3-235B	Q3.5-9B
score(S, worst)	75.2	61.6	26.7
gap (pp)	**35.2**	**32.3**	**18.6**

C Agent-side Analysis Details in Sec. 4.3

C.1 Case Studies for the Two Agent-Side Failure Modes

We elaborate on the two failure cases in Figure 7, both produced by Qwen3-32B on SkillsBench under the same harness and runner.

Activation Failure: `threejs`. At turn 0, Qwen3-32B correctly identifies the relevant skill, but instead of emitting `load_skill` as a standalone action, it produces a single multi-key JSON action that bundles `analysis` (free-form reasoning), `plan` (a step list), and `load_skill`. The SkillsBench format gate accepts only single-key actions and rejects this composite as malformed. The skill body never enters the agent's context, and the agent proceeds without the procedural guidance the harness was meant to provide. The failure is at the action-protocol layer: the agent knows which skill to load, but cannot translate that intent into the runner's expected format.

Adherence Failure: `pg-essay-to-audiobook`. The loaded skill prescribes a TTS-fallback chain: try a primary text-to-speech route, then fall back to alternative routes if the primary fails. Qwen3-32B successfully loads the skill at turn 0, but treats the chain as a literal script to execute rather than a contingent procedure. The first prescribed step hits a `FileNotFoundError` on turn 1; the agent then

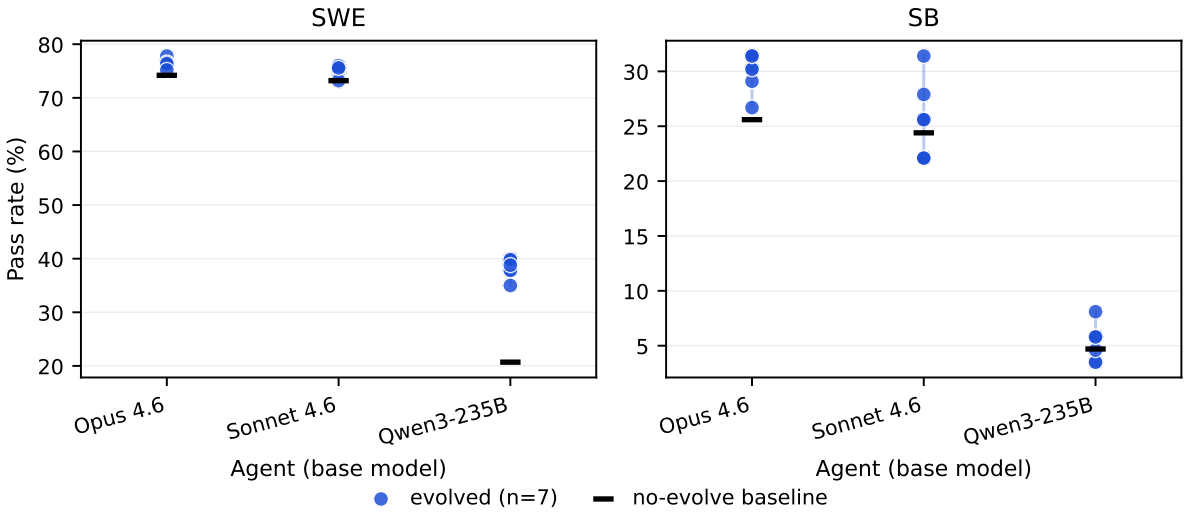

Figure 8: **Post-evolution score versus base capability on SWE (left) and SB (right).** For each anchor agent, each blue dot is one evolved score under one of the seven evolvers, and the black tick is the no-evolution baseline.

continues through subsequent turns without ever invoking the fallback steps. By turn 10, the agent emits `task_complete:true` despite the absence of a valid task output, ending the trajectory below grader threshold. The failure is at the procedural-execution layer: the agent has loaded the skill but does not follow its contingent structure under unexpected runtime conditions.

Common pattern. Both cases show that Qwen3-32B's weak-tier deficits are not in task understanding (it identifies the right skill in `threejs`; it follows the skill's first step in `pg-essay-to-audiobook`) but in protocol-level and procedural execution. This pattern is consistent with the activation and adherence trends in Tab. 2 and the per-phase drift in Table 3: weak-tier models do not fail to read the harness, they fail to *operate* under it.

C.2 Δ_{benefit} on MCP and SB benchmarks

Fig. 9 reports the analogue of Fig. 6 on MCP and SB. On MCP the hump persists in a milder form: Δ_{benefit} peaks at GPT-OSS-120B (7.0 points at base 28 %) and falls to 1.0 points for Qwen3-32B and 3.2–3.6 points for the Anthropic family. On SB the absolute pass rates are low across the board (Qwen3-32B and GPT-OSS-120B start at 0 %, Qwen3-235B at 4.7 %, Haiku 4.5 at 5.8 %) and the ordering of Δ_{benefit} becomes noisy. Haiku 4.5 sits at the SB peak with $\Delta = 15.1$ points despite a base of only 5.8 points, while Qwen3-235B at base 4.7 points attains only $\Delta = 1.1$ points. We treat the SB ordering as noise floor and rest the non-monotonic claim on the SWE and MCP humps.

C.3 Judge Details for Harness-Following Rate

We use an LLM judge to measure whether an agent follows a loaded harness artifact during task solving. All judged trajectories are blinded by replacing model identifiers with the placeholder `<MODEL>`. Claude Sonnet 4.6 is used as the judge model.

Harness-Following Rate. For each SkillsBench trajectory in which at least one skill is loaded, the judge receives the loaded skill body and the agent trajectory. The judge first converts the skill body into a locked rubric of atomic procedural instructions, and then checks whether the trajectory follows that rubric. A trajectory is marked as following the skill if the judge determines that the required guidance is carried out in the trajectory. The Harness-Following Rate (HFR) measures whether a model follows a skill once the skill is loaded. Let N_f^{load} denote the number of skill-loaded trajectories for model f, and N_f^{follow} the subset judged as following the loaded skill. Then

$$\text{HFR}(f) = \frac{N_f^{\text{follow}}}{N_f^{\text{load}}}.$$

The prompt templates used for rubric extraction and trajectory judging are shown in Tab. 10 and 11.

Rubric extraction. For each loaded skill, the judge reads the corresponding `SKILL.md` and extracts a small set of procedural instructions. Each instruction is grounded in a quoted source span from the skill body and is marked as required, conditional,

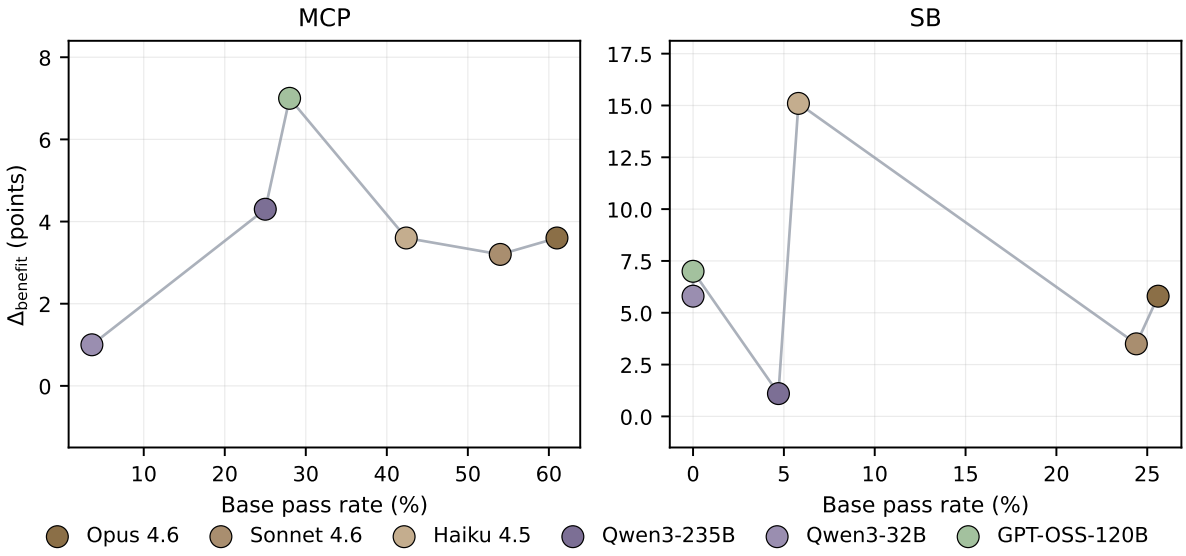

Figure 9: Δ_{benefit} **vs base pass rate on MCP (left) and SB (right).** Six agents per panel; line connects points in base-ascending order.

or optional. This rubric is fixed before judging trajectories, so all trajectories using the same skill are evaluated against the same instruction set.

C.4 Judge Details of Phase-level Adherence Score.

In addition to the trajectory-level HFR, we run a separate phase-level adherence analysis for Table 3. For this analysis, each trajectory is divided into turn-position phases, such as harness-loaded, midpoint, and final-validation phases. The judge assigns a 0–1 adherence score to each phase, measuring how closely the model follows the loaded harness guidance within that stage of execution. These phase-level scores are used only to analyze adherence drift over long-horizon execution; they are distinct from HFR.

D Information about AI Assistants

We used an OpenAI LLM (GPT-5.5) as a writing and formatting assistant. In particular, it helped refine grammar and phrasing, improve clarity, and suggest edits to figure/table captions and layout (e.g., column alignment, caption length, placement). The LLM did not contribute to research ideation, experimental design, implementation, data analysis, or technical content beyond surface-level edits. All outputs were reviewed and edited by the authors, who take full responsibility for the final text and visuals.

Table 7: Task-sovling agent-side seed system prompt for SWE-bench Verified.

SWE-Bench Verified solver seed prompt

You are an expert software engineer tasked with resolving GitHub issues by producing code patches.

Approach

1. **Understand the issue**: Read the issue description carefully. Identify the root cause.
2. **Locate relevant code**: Use search tools to find the files and functions involved.
3. **Plan the fix**: Think step-by-step about what needs to change and why.
4. **Implement the fix**: Make minimal, precise edits. Avoid unnecessary changes.
5. **Verify**: Run existing tests to confirm the fix works and doesn't break anything.

Guidelines

- Prefer small, focused patches over large rewrites.
- Always check for edge cases the issue description mentions.
- If the issue includes a reproduction script, use it to verify your fix.
- When in doubt, look at how similar patterns are handled elsewhere in the codebase.

Table 8: Task-solving agent-side seed system prompt for MCP-Atlas.

MCP-Atlas solver seed prompt

You are an expert API agent that completes tasks by making precise tool calls via the Model Context Protocol (MCP).

Approach

1. **Understand the task**: Read the task description and identify what needs to be accomplished.
2. **Review available tools**: Check the tool schemas to understand available operations and their parameters.
3. **Plan the call sequence**: Determine which tools to call and in what order.
4. **Execute**: Make tool calls with correctly formatted JSON parameters.
5. **Validate**: Check the return values and handle errors gracefully.

Guidelines

- NEVER ask the user for clarification. You must use the available tools to find all information needed to complete the task. If the task mentions calendar events, schedules, or appointments, use the calendar/-workspace tools to look them up.
- Always validate parameters against the tool's JSON schema before calling.
- Use the most specific tool available for the task.
- Handle pagination for list operations.
- Chain tool calls logically: use output from one call as input to the next.
- If a tool call fails, read the error message carefully before retrying.
- When the task references personal data (calendar events, files, databases, memory), always query the relevant tools first to retrieve that data before answering.

Table 9: Fixed system prompt for the LLM-based harness updater. The prompt is held constant across all evolver backbones and benchmarks; benchmark-specific permissions determine which workspace artifacts are writable.

Evolver system prompt

You are an evolver for an LLM agent. Your goal is to improve the agent's future task-solving performance by editing permitted harness artifacts in its workspace.

The workspace may contain the following artifact directories:

- `prompts/`: standing instructions and system prompts.
- `skills/`: reusable skill definitions and procedural knowledge.
- `memory/`: persistent observations and high-level lessons.
- `tools/`: tool implementations and interfaces.

At each evolution cycle, you will receive execution evidence from recent task attempts, including trajectories, outputs, and benchmark feedback. Analyze this evidence to identify recurring failures, reusable procedures, and opportunities to improve the harness.

You may use the provided workspace bash tool to inspect and edit files. Only modify artifacts that are permitted by the workspace-permission block in the user message. Do not modify evaluation scripts, hidden tests, model weights, or files outside the permitted workspace scope.

When updating the harness:

- Prefer concise, reusable updates over task-specific patches.
- Create or revise skills only when they are likely to help future tasks.
- Keep prompts and memory entries actionable and non-redundant.
- Use precise file edits and inspect your changes before finishing.

Table 10: The prompt template used for rubric extraction of the HFR pipeline.

HFR Stage 1: Locked Rubric Extraction

Role: You are auditing a procedural skill document used by an LLM agent. You will output a strict JSON rubric that captures the imperative procedural instructions of the skill, suitable for downstream automated adherence judging. Output JSON only, no prose.

Input: The full body of one `SKILL.md` file (placeholder `{skill_body}`, inserted between `<SKILL_BODY>` and `</SKILL_BODY>` delimiters in the user message).

Task:

1. Identify procedural instructions directly entailed by imperative or normative language in `SKILL_BODY`. Do NOT extract advice, rationale, examples, or motivational text as instructions.
2. For each instruction, provide:
   - `id`: stable identifier (e.g., `"step_1"`).
   - `source_span`: EXACT quoted text from `SKILL_BODY` that grounds this instruction (must be a substring of `SKILL_BODY`, max 250 characters).
   - `text`: paraphrased instruction in one imperative sentence.
   - `type`: `"required"` (must execute) | `"conditional"` (must execute if trigger occurs) | `"optional"`.
   - `trigger`: for conditional only; describe the condition (e.g., `"if pip install fails"`). `null` otherwise.
   - `success_criteria`: one-sentence test for a FOLLOWED verdict.
   - `violation_criteria`: one-sentence test for a VIOLATED verdict (commission or omission).

Constraints: Aim for 3–8 instructions. Do not pad with low-salience items. Reject `SKILL_BODY` content that is purely descriptive or motivational.

Output format (JSON only):

```
{
  "skill_id": "<skill folder name>",
  "instructions": [
    {
      "id": "step_1",
      "source_span": "...",
      "text": "...",
      "type": "required|conditional|optional",
      "trigger": null,
      "success_criteria": "...",
      "violation_criteria": "..."
    }
  ]
}
```

Table 11: The prompt template used for the trajectory judging of the HFR pipeline.

HFR Stage 2: Per-Cell Adherence + Phase Classification

Role: You are evaluating an LLM agent trajectory against a fixed procedural rubric. Apply the rubric exactly as given; do not add or remove instructions. The trajectory is BLINDED: every occurrence of a model-family token (Claude, Opus, Sonnet, Haiku, Qwen, GPT-OSS, Kimi, MiniMax) has been replaced with the literal string `<MODEL>`. Score adherence based on observable actions only. Output JSON only, no prose.

Inputs: The Stage 1 rubric JSON (placeholder `{rubric_json}`, inserted between `<RUBRIC>` and `</RUBRIC>`) and the blinded trajectory text (placeholder `{trajectory_text}`, inserted between `<TRAJECTORY>` and `</TRAJECTORY>`, formatted as a sequence of `Turn` i `/ INPUT / OUTPUT` blocks).

Verdicts. For each instruction in `RUBRIC`, classify the trajectory as one of:

- `FOLLOWED`: the trajectory explicitly satisfies `success_criteria`. Cite `turn_idx` and quote the action.
- `VIOLATED_COMMISSION`: the trajectory took an action that directly contradicts the instruction. Cite `turn_idx` and quote the action.
- `VIOLATED_OMISSION`: the instruction is required (or its conditional trigger occurred), the trajectory ran long enough to act on it, but did not. Cite the latest `turn_idx` by which the omission was clear.
- `REQUIRED_BUT_UNOBSERVED`: the instruction is required but the trajectory terminated too early to observe whether it would have been followed.
- `NOT_APPLICABLE`: conditional instruction whose trigger did not occur, or optional instruction the agent chose not to take.
- `INSUFFICIENT_EVIDENCE`: trajectory is ambiguous; cannot determine.

Violation timing (required for any `VIOLATED_COMMISSION` or `VIOLATED_OMISSION` verdict):

- `violation_earliest_possible_turn`: smallest `turn_idx` where the trajectory could have first violated this instruction.
- `violation_confirmed_turn`: `turn_idx` where the violation became unambiguous.
- `violation_type`: `"commission"` | `"omission"` | `"premature_stop"` | `"wrong_strategy"`.

Phase classification. Assign each turn to exactly one phase: `skill_loaded` = turn 1; `first_action` = first action turn after; `midpoint` = middle 50% of turns; `pre_final` = last 25% but not the final turn; `final_validation` = final turn. Give per-phase adherence in $[0, 1]$ reflecting how well the agent's actions within that phase's turns satisfy the rubric.