Adapting the Interface, Not the Model: Runtime Harness Adaptation for Deterministic LLM Agents

Anonymous ACL submission

Abstract

LLM agents are shaped not only by their language models, but also by the runtime harness that mediates observation, tool use, action execution, feedback interpretation, and trajectory control. While existing agent adaptation methods mainly update model parameters, many failures in deterministic, rule-governed domains stem from mismatches at the model–environment interface. We propose LIFE-HARNESS, a lifecycle-aware runtime harness that improves frozen LLM agents without changing model weights or evaluation environments. LIFE-HARNESS evolves from training trajectories by converting recurring interaction failures into reusable interventions across environment contracts, procedural skills, action realization, and trajectory regulation, and remains fixed for evaluation on unseen tasks. On seven deterministic environments from τ-bench, τ^2-bench, and AgentBench, LIFE-HARNESS improves 116 out of 126 model–environment settings across 18 model backbones, with an average relative improvement of 88.5%. Harnesses evolved only from Qwen3-4B-Instruct trajectories transfer to 17 other models, showing that LIFE-HARNESS captures reusable environment-side structure rather than model-specific behavior. These results position runtime interface adaptation as a complementary alternative to model-centric agent training. Code is available at Anonymous GitHub.

1 Introduction

An LLM agent is not just an LLM. As shown in Figure 2 (a), it is a model embedded in a stateful interaction loop: the environment exposes observations, the runtime system specifies available tools and actions, the model emits an action or tool call, an executor applies it to the environment, and the resulting feedback updates the next decision (Wang et al., 2024; Anthropic, 2026). The resulting behavior is therefore determined not only

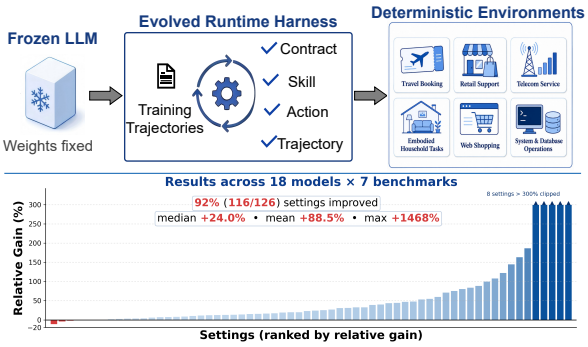

Figure 1: Adapting the runtime harness, not the model. LIFE-HARNESS keeps LLM weights fixed and evolves reusable interface interventions from training trajectories, yielding broad and substantial gains across agentic tasks, benchmarks, and model backbones.

by the model, but also by the runtime harness that mediates how the model observes the environment, understands tools, realizes actions, and interprets feedback. This system-level view is increasingly important across agent applications, including software engineering assistants (Yang et al., 2024b; OpenAI, 2026; OpenCode, 2026), web navigation (Zhou et al., 2024), database manipulation (Lei et al., 2025), operating-system control (Xie et al., 2024; Liu et al., 2024), tool-using business workflow agents (Yao et al., 2024; Barres et al., 2025), and embodied interaction (Shridhar et al., 2020).

Despite this, agent adaptation is still commonly framed as model adaptation, including model scaling (Dubey et al., 2024; Team, 2024), fine-tuning (Chen et al., 2023; Prabhakar et al., 2026), reinforcement learning (Guo et al., 2025; Qi et al., 2025), and distillation (Kang et al., 2026; Agarwal et al., 2024b). These approaches are powerful, but they implicitly absorb domain-specific behavior into model parameters. In deterministic, rule-governed domains, much of the relevant structure instead lives outside the model: tool schemas, API contracts, admissible actions, feedback rules, stopping conditions, and recovery

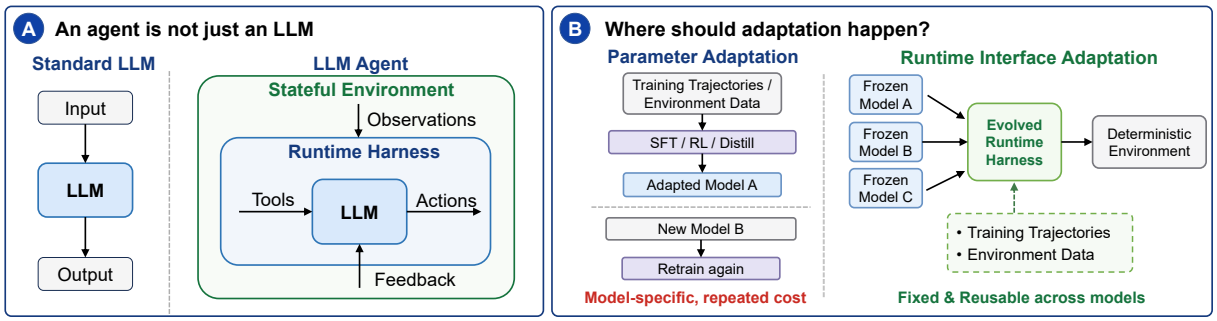

Figure 2: (a) An agent is not just an LLM: its behavior is shaped by the runtime harness that mediates observations, tools, actions, and feedback. (b) We adapt this runtime interface, rather than model parameters, yielding a fixed and reusable harness for frozen agents across deterministic environments.

strategies. This distinction is reflected in the gap between static capability and interactive performance. For example, Qwen3.5-4B achieves 74.0% on HMMT Feb (HarvardMIT Mathematics Tournament, 2026), a competition-level mathematical reasoning benchmark, but only 43.1% on ALFWorld (Shridhar et al., 2020), a deterministic embodied interaction benchmark. Such failures often stem not from a lack of latent reasoning ability, but from mismatches at the model–environment boundary: poorly structured observations, misunderstood tool contracts, non-executable actions, feedback that fails to trigger recovery, or degenerate trajectory dynamics. This suggests an alternative path for agent improvement: rather than embedding all environment-specific constraints into model weights, one can expose stable environment-side structure through the runtime interface where the model observes, acts, and recovers.

Recent work has begun to explore this direction by optimizing the runtime harness around frozen LLM agents, including reasoning-time compute controllers (Zheng et al., 2026), online workspace adaptation in interactive games (Sarafian et al., 2026; Karten et al., 2026), harness-flag optimization (Sengupta and Wang, 2026), and automated harness-code search (Lee et al., 2026; Lin et al., 2026). These studies establish harness optimization as an important alternative to model training, but they largely treat the harness as a policy, mutable state, configuration space, or code artifact to be optimized as a whole. We instead focus on deterministic agent domains, where the harness can be viewed as a stable runtime interface between a frozen model and a rule-governed environment. In this setting, recurring failures can be localized to specific stages of the interaction lifecycle, allowing failure-specific interface interventions to be evolved from training trajectories and then fixed during evaluation on unseen tasks. This raises our central question: *Can training trajectories reveal stable failure structures that can be converted into a structured runtime interface to improve LLM agents across unseen tasks and diverse model architectures?*

We answer this question with LIFE-HARNESS, a lifecycle-aware runtime harness for deterministic LLM agents. Rather than updating model parameters or searching over unconstrained harness code (Lee et al., 2026), LIFE-HARNESS adapts the runtime layer that mediates how a frozen model observes the environment, uses tools, realizes actions, interprets feedback, and recovers from degenerate trajectories. To model this runtime interface, LIFE-HARNESS organizes adaptation into four lifecycle layers. The **Environment Contract Layer** calibrates tool descriptions and interface constraints before interaction. The **Procedural Skill Layer** distills reusable procedures from past trajectories and retrieves them for the current task and state. The **Action Realization Layer** validates and canonicalizes model-generated actions before execution, preventing deterministic failures. The **Trajectory Regulation Layer** monitors post-execution dynamics, detects degenerate patterns such as repetition, stagnation, invalid retries, or budget exhaustion, and triggers recovery when needed.

LIFE-HARNESS is then evolved from training trajectories by diagnosing recurring interaction failures and converting them into reusable, auditable runtime interventions, while the resulting harness remains fixed for evaluation on unseen tasks.

We evaluate LIFE-HARNESS on seven deterministic agent environments from τ-bench (Yao et al., 2024), τ^2-bench (Barres et al., 2025), and AgentBench (Liu et al., 2024), spanning household inter-

action, web shopping, OS control, database tasks, and policy-guided business workflows. Across 18 instruction-tuned, reasoning, and agent-specialized backbones, LIFE-HARNESS improves 116 of 126 model–environment settings, yielding an average relative gain of **88.5%**. The harnesses are evolved only from Qwen3-4B-Instruct (Team, 2025) trajectories and then reused across the other 17 backbones, suggesting that LIFE-HARNESS captures reusable environment-side structure rather than model-specific behavior. Finally, LIFE-HARNESS is complementary to model training: it enables Qwen2.5-32B-Instruct (Team, 2024) to outperform its tool-use-trained derivative, xLAM-2-32b-fc-r (Prabhakar et al., 2026), while further improving xLAM itself.

Our contributions are threefold: (1) We formulate *harness-based runtime interface adaptation* for deterministic LLM agents, framing agent improvement as evolving the reusable interface between a frozen model and a rule-governed environment. (2) We introduce **LIFE-HARNESS**, a lifecycle-aware framework that converts recurring trajectory failures into targeted interventions across environment contracts, procedural skills, action realization, and trajectory regulation. (3) We demonstrate broad **cross-model gains** across seven environments and 18 backbones, achieving an average relative improvement of **88%** without updating model weights or modifying evaluation protocols. Together, these results suggest that many practical agent failures can be addressed by evolving the **runtime interface** rather than modifying model parameters.

2 Related Work

Harness Optimization. A very recent line of work has begun to optimize the scaffold around frozen LLM systems. AutoTTS (Zheng et al., 2026) searches reasoning-time controllers for allocating width/depth computation in mathematical reasoning. Workspace Optimization (Sarafian et al., 2026) and Continual Harness (Karten et al., 2026) study online adaptation in interactive game-like environments, where agents revise workspace state, prompts, skills, memories, or executable artifacts from their own episode history. HARBOR (Sengupta and Wang, 2026) treats harness tuning as Bayesian optimization over pre-existing feature flags, while Meta-Harness (Lee et al., 2026) searches over complete harness programs using prior candidate code, scores, and execution traces. More recently, AHE (Lin et al., 2026) performs observability-driven evolution of coding-agent harnesses by exposing editable components as files, distilling trajectory evidence, and verifying edits through prediction manifests.

LIFE-HARNESS shares the premise that frozen agents can be improved outside model weights, but targets a different scope and abstraction. Where Meta-Harness and AHE focus on automated harness engineering for coding agents, LIFE-HARNESS studies deterministic, rule-governed agent domains beyond coding, including household interaction, web shopping, database tasks, and policy-guided workflows. Rather than treating the harness as a free-form code artifact to be searched or continuously edited, LIFE-HARNESS treats it as a structured runtime interface whose adaptation is organized by the agent interaction lifecycle. Recurring training-trajectory failures are mapped to fixed interventions for environment contracts, procedural skills, action realization, and trajectory regulation; these interventions are then evaluated on unseen tasks and reused across model backbones.

Prompt Adaptation Methods. Prompt optimization adapts frozen LLM systems by rewriting instructions, demonstrations, or prompt templates instead of model weights (Agarwal et al., 2024a; Fernando et al., 2023). Representative methods include automatic prompt optimization, LLM-as-optimizer approaches such as OPRO (Yang et al., 2024a), textual-gradient methods such as ProTeGi (Pryzant et al., 2023) and TextGrad (Yuksekgonul et al., 2024), and reflective optimizers such as GEPA (Agrawal et al., 2025). These methods are complementary to LIFE-HARNESS: they optimize model-facing text, whereas LIFE-HARNESS adapts the broader runtime interface, including prompt-facing contracts, action validation, feedback-driven recovery, and trajectory regulation.

Model-side Adaptation for LLM Agents. Most agent adaptation work improves the model itself through instruction tuning (Team, 2024), tool-use fine-tuning (Prabhakar et al., 2026), reinforcement learning (Guo et al., 2025; Xu et al., 2026; Qian et al., 2026), distillation (Lu and Lab, 2025), and related post-training methods (Dubey et al., 2024; Team, 2025; DeepSeek-AI, 2026; Xiao et al., 2026). While such methods can substantially improve agent behavior, their adaptations remain tightly coupled to specific model checkpoints and training distributions. LIFE-HARNESS offers a complementary paradigm: it leaves model weights frozen and

instead adapts the runtime interface through which the model observes and acts.

3 From Parameter Adaptation to Runtime Interface Adaptation

3.1 LLM Agents as Runtime Systems

We formalize an LLM agent as a policy π_θ interacting with a stateful environment, which may include both the environment E and a user U. An episode is defined by a task x, the combined environment E (including U), an environment contract C, and a step budget B. The contract C specifies the intended interaction protocol: available tools, argument and feedback formats, answer formats, and task-specific policies. The episode begins with

$$s_0, o_0 = E.\textsc{Init}(x),$$

where s_0 and o_0 denote the initial state and observation. At step t, the trajectory

$$\tau_t = (C, x, o_0, a_0, o_1, \ldots, a_{t-1}, o_t)$$

contains the contract, task, past actions, and observations. The agent produces an action

$$a_t \sim \pi_\theta(\cdot \mid \tau_t),$$

where a_t can be a tool call, text command, or final-answer submission. Text actions can be treated as pseudo-tools (e.g., ACT, SUBMITANSWER) for a unified notation. The environment processes the action:

$$s_{t+1}, o_{t+1} = E.\textsc{Step}(s_t, a_t),$$

If a_t is unsupported or ineffective, the environment may return an error message; such feedback is still part of the interaction trajectory. The updated trajectory is

$$\tau_{t+1} = (C, x, o_0, a_0, \ldots, a_t, o_{t+1}),$$

and the episode continues until task completion, environment termination, or step budget exhaustion. This runtime perspective emphasizes that performance depends not only on π_θ but also on how outputs are mediated before and after execution, motivating explicit harness adaptation.

3.2 Parameter vs. Runtime Interface Adaptation

Conventional approaches adapt agents by updating model parameters from training trajectories:

$$\theta' \leftarrow \mathcal{A}_{\text{param}}(\theta, \mathcal{T}_{\text{train}}),$$

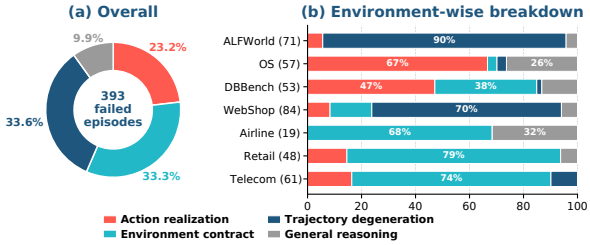

Figure 3: Failure diagnosis on training tasks.

absorbing task-specific structure into the weights. This *parameter adaptation* is inherently **model- and task-specific**, and must be repeated when the base model changes or when deployed in new environments.

We study a complementary approach: keeping θ fixed while adapting the runtime harness:

$$H' \leftarrow \mathcal{A}_{\text{harness}}(H, \mathcal{T}_{\text{train}}), \qquad \theta \text{ fixed}.$$

The adapted harness H' changes how the model interacts with the environment, while leaving model weights and evaluation protocols unchanged. We term this *runtime interface adaptation*, which is **environment-specific** but **model-agnostic**: a harness evolved for one environment generalizes to different model backbones that follow the same interaction protocol, without retraining.

4 Method: LIFE-HARNESS

We propose LIFE-HARNESS, an evolving lifecycle runtime harness for deterministic LLM agents. The harness adapts the model–environment interface rather than model weights. It operates on the interaction loop defined in Section 3.1: the environment contract C, the task description x, the environment state s_t, the model action a_t, and the trajectory τ_t.

4.1 Failure Diagnosis

Before designing the harness, we first diagnose the primary failure modes of baseline agents. We evaluate a frozen Qwen3-4B-Instruct on training tasks across diverse interactive agent environments, and manually inspect failed trajectories. For each failed episode, we assign a primary failure type according to the earliest dominant bottleneck in the agent–environment loop, following the classification rules detailed in Appendix A.1. The resulting taxonomy is summarized in Figure 3. We identify four recurring categories. **Action realization failures** occur when the model's intent is plausible but not expressed in an environment-executable form,

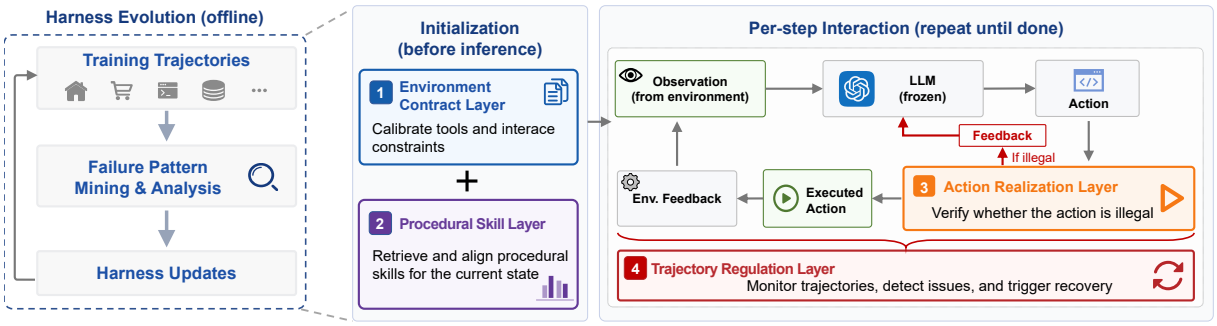

Figure 4: Overview of LIFE-HARNESS. The harness adapts the model-environment interface through four lifecycle layers spanning before interaction, task conditioning, before environment execution, and after execution.

such as free-form actions or missing arguments. **Environment contract mismatches** occur when an action is syntactically executable but violates the intended tool usage or calling protocol. **Trajectory degeneration** occurs when individual actions are valid, but the episode falls into repetition, stagnation, or ineffective recovery. The remaining **general reasoning failures** arise from incorrect inference, computation, or decision-making despite largely following the protocol.

As shown in Figure 3, deterministic agent failures are heterogeneous: all four categories occur in practice, and the dominant mode varies substantially across environments. This motivates a harness with multiple intervention points, spanning environment-contract clarification, procedural retrieval, pre-execution action validation, and post-execution trajectory regulation.

4.2 Overview

Guided by the failure diagnosis above, LIFE-HARNESS comprises four layers integrated across different stages of the agent lifecycle. Figure 4 provides an overview of LIFE-HARNESS.

❶ **Environment Contract Layer** operates before interaction. It makes stable environment constraints explicit, including tool-use rules, policy constraints, and common pitfalls that agents frequently encounter in the target environment.

❷ **Procedural Skill Layer** operates at the task-conditioning stage. It maintains a skill library distilled from training trajectories and retrieves relevant skills based on the user's task description. This layer provides non-parametric guidance for general decision-making.

❸ **Action Realization Layer** operates after the model outputs an action and before the environment executes it. It verifies whether the action is executable under the environment contract, canonicalizes unambiguous interface-level errors, and blocks actions that would deterministically fail. This layer ensures that the model's intended operation is reliably mapped to a valid tool call or environment action.

❹ **Trajectory Regulation Layer** operates after environment feedback is returned. It monitors the updated trajectory for non-progressing patterns such as repetition, stagnation, or budget exhaustion, and triggers recovery when needed. This layer specifically targets trajectory degeneration.

Together, these layers adapt the runtime interface through which the model interacts with the environment. The model weights remain fixed, and the evaluation environment is unchanged.

4.3 Detailed Design of LIFE-HARNESS

4.3.1 Environment Contract Layer

This layer makes stable environment constraints explicit before interaction by adapting the model-visible contract C. Formally, it produces an enhanced contract $C' = C \oplus \Delta_C$, where Δ_C contains concise updates derived from environment policies, API behavior, and recurring failures in training trajectories. The enhanced contract C' is shown to the model in place of C, enabling the agent to better utilize the given tools. In practice, Δ_C may specify how tools should be called, which actions are admissible under the environment protocol, and which environment-specific pitfalls should be avoided.

4.3.2 Procedural Skill Layer

This layer provides non-parametric guidance from training trajectories. A skill is a compact and reusable strategy that captures the essence of how to accomplish specific subtasks.

Let $\mathcal{S}$ be the skill memory constructed from training trajectories. For a task description x, the har-

ness retrieves relevant skills:

$$\mathcal{K}_x = \mathrm{TopK}_{k \in \mathcal{S}} \, \mathrm{score}(x, k),$$

where score is implemented with BM25 in our experiments. The retrieved skills $\mathcal{K}_x$ are inserted into the initial system prompt to guide the model on how to solve specific common problems.

4.3.3 Action Realization Layer

This layer operates after the model outputs an action and before the environment executes it. Given the model action a_t, the current trajectory τ_t, and the current state s_t, the layer either submits this action a_t to the environment or returns a model-visible block message m_t:

$$z_t = \textsc{RealizeAction}(a_t, \tau_t, s_t)$$
$$z_t \in \{\mathrm{EXEC}(a_t), \textsc{Block}(m_t)\}.$$

It uses deterministic environment evidence, such as tool schemas, admissible action sets, argument constraints, and task policies, to enforce the prevention of erroneous tool calls at the execution level.

4.3.4 Trajectory Regulation Layer

The Trajectory Regulation Layer monitors the interaction after environment execution. Many agent failures are self-reinforcing: the agent repeats the same invalid command, loops between equivalent states, or exhausts the budget without making progress. Such failures are often detectable from trajectory-level patterns rather than deep semantic understanding. Given the executed action, returned observation, remaining budget, and environment evidence, the layer computes:

$$r_t = \textsc{RegulateTrajectory}(\tau_t, a_t, o_{t+1}, b_t)$$

where $b_t = B - t - 1$ is the remaining budget. The output r_t may be empty, a soft recovery message, a warning regarding repeated failures, or a stronger corrective directive when the trajectory exhibits clear signs of degradation.

The four layers act at complementary stages of the agent lifecycle: contract calibration before interaction, skill retrieval during task conditioning, action realization before execution, and trajectory regulation after execution, adapting the runtime interface between model and environment without changing model weights, the environment, or the evaluation protocol. The full LIFE-HARNESS algorithm is shown in Algorithm 1.

Algorithm 1: LIFE-HARNESS loop

```
Input: task x, environment E, contract C, budget B
C' ← EnvContract(C), x' ← SkillLayer(x);
s_0, o_0 ← E.Init(x'), τ_0 ← (C', x', o_0);
for t = 0, ..., B − 1 do
    a_t ← LLM(τ_t);
    z_t ← RealizeAction(a_t, τ_t, C', s_t);
    (s_{t+1}, o_{t+1}) ←
      ExecuteOrBlock(E, s_t, z_t);
    τ_{t+1} ← τ_t ⊕ (z_t, o_{t+1});
    τ_{t+1} ←
      RegulateTrajectory(τ_{t+1}, C', s_{t+1});
    if IsEnd(s_{t+1}, o_{t+1}) then
        break
    end
end
```

4.4 Trajectory-Driven Harness Evolution

LIFE-HARNESS is evolved from training trajectories with the assistance of a coding agent, Codex (OpenAI, 2026). We repeatedly execute a frozen model on the training tasks to collect complete interaction traces. The coding agent then reads these traces together with the harness design criteria and proposes updates to the corresponding layers. The objectives are twofold: (1) to extend harness coverage for recurring failure patterns, and (2) to detect regression cases where interventions may over-trigger or compromise otherwise correct behavior. The prompts used for harness evolution and the final harness generated by each task are provided in the Appendix A.2.

5 Experiments

5.1 Experimental Setup

Benchmarks. We evaluate LIFE-HARNESS on three benchmark suites: τ-bench (Yao et al., 2024), τ^2-bench (Barres et al., 2025), and AgentBench (Liu et al., 2024), covering seven task scenarios: Airline, Retail, Telecom, ALFWorld (Shridhar et al., 2020), WebShop (Yao et al., 2022), OS, and DBBench. These benchmarks share the properties central to our setting: **stable environments and deterministic tasks**, making the runtime harness a high-leverage target for adaptation.

Models. We use Qwen3-4B-Instruct (Team, 2025) as the source model for harness evolution. Specifically, we run it on training tasks, collect trajectories, and use a coding agent, Codex (OpenAI, 2026), to inspect traces and iteratively update the harness. **In the harness evolve process, the test set is always hidden to ensure generalization.** The final evolved harness is then frozen and reused for

Suite	Benchmark	Metric	w/o	w/ LIFE-HARNESS	Rel. Gain	Improved
AgentBench	ALFWorld	Pass@1	41.1%	**75.7%**	+84%	17/18
	WebShop		31.4%	**44.0%**	+40%	18/18
	OS		34.7%	**41.2%**	+19%	18/18
	DBBench		48.4%	**64.6%**	+34%	18/18
τ-bench	Airline	Pass@1	49.7%	**62.6%**	+26%	16/18
		Pass^3	34.7%	**52.2%**	+50%	17/18
	Retail	Pass@1	56.2%	**61.8%**	+10%	14/18
		Pass^3	37.9%	**45.3%**	+19%	15/18
τ^2-bench	Telecom	Pass@1	55.3%	**69.0%**	+25%	17/18
		Pass^3	41.5%	**52.6%**	+27%	18/18

Table 1: Main results averaged over 18 model backbones.

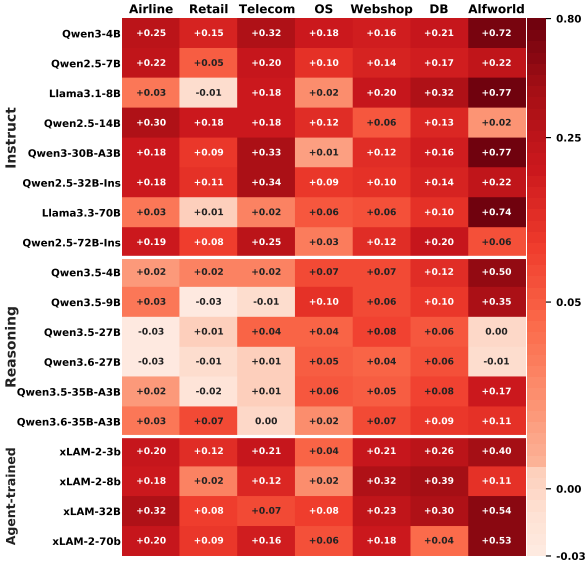

Figure 5: Absolute performance improvement across 18 model backbones and 7 benchmarks.

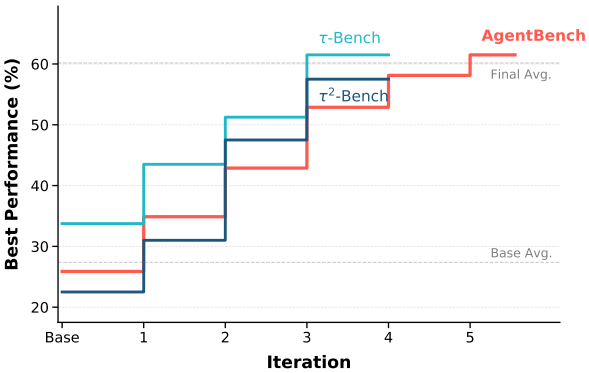

Figure 6: Training set performance improves steadily as the number of evolutionary iterations increases.

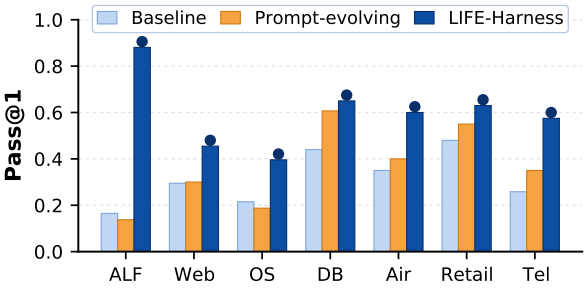

Figure 7: Comparison with prompt evolving method.

Setting	τ-bench		τ^2-bench	AgentBench			
	Airline	Retail	Telecom	ALFWorld	WebShop	OS	DBBench
LIFE-HARNESS	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
w/o Contract	-8.3%	-17.5%	-16.0%	-1.0%	-4.4%	-14.1%	-16.9%
w/o Skill	-8.3%	-15.9%	-17.4%	-1.0%	-2.2%	-14.1%	-3.1%
w/o Action	-61.7%	-15.9%	-10.1%	-1.0%	-6.6%	-59.6%	-4.6%
w/o Trajectory	-3.3%	-16.7%	-36.2%	-86.5%	-26.4%	-14.1%	-4.6%

Table 2: Leave-one-layer-out ablation on Qwen3-4B-Instruct. Values report the relative **accuracy drop** compared with the full LIFE-HARNESS. "Contract", "Skill" "Action", and "Trajectory" denote the Environment Contract, Procedural Skill, Action Realization and Trajectory Regulation layers, respectively.

evaluating 17 additional open-source model backbones. Our model set covers Qwen-family models (Team, 2024, 2025; Qwen Team, 2026), Llama-family models (Dubey et al., 2024), and xLAM-family models (Prabhakar et al., 2026), including instruction-tuned models, reasoning models, and models post-trained for agentic benchmarks.

Evaluation Parameters. All evaluations use a sampling temperature of 0.0. For τ-bench and τ^2-bench, we use DeepSeek-V4-Flash (DeepSeek-AI, 2026) as the user LLM and evaluate each task three times, reporting both single-run success and Pass^3, where Pass^3 requires all three runs to succeed. For AgentBench, each task is evaluated once following the official implementation. More detailed per-task configurations are provided in Appendix B.

5.2 Main Results

Performance Gains of LIFE-HARNESS. Table 1 reports the performance improvements brought by LIFE-HARNESS across all benchmarks, averaged over 18 models, while Figure 5 provides a model-by-benchmark breakdown over the 18 models and 7 benchmarks. We make two observations. 1) LIFE-HARNESS consistently improves performance on all benchmarks, achieving relative gains of $10 \sim 84\%$. With the evolved runtime harness, smaller models can become competitive with substantially larger baselines. 2) Although LIFE-HARNESS is evolved only from Qwen3-4B-Instruct trajectories, it generalizes broadly across other backbones: 92% of all model–benchmark settings improve, covering instruct models, reasoning models, and models trained specifically for agentic tasks. These results suggest that LIFE-HARNESS captures reusable environment-side structure rather than model-specific behavior. Complete experimental results are provided in Table 8.

Evolution Dynamics. Figure 6 shows the training-

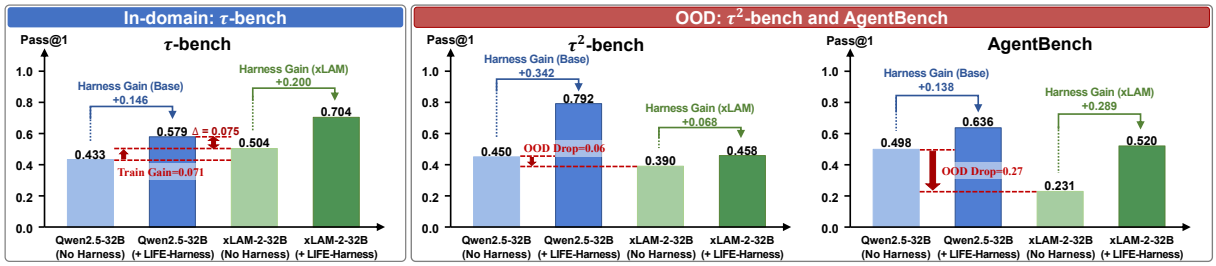

Figure 8: Comparison between specialized tool-use training and runtime harnessing. Harnessing can outperform tool-use training without updating model weights, remains useful after training, and mitigates the limited OOD transfer of specialized training.

set performance of Qwen3-4B-Instruct as LIFE-HARNESS is iteratively evolved on each task. Performance improves steadily over evolution rounds and eventually saturates, suggesting that iterative harness evolution is both practical and efficient. The rapid convergence also reflects the benefit of the four-layer design, where updates are localized to identifiable failure modes instead of rewriting the harness as an unconstrained whole (Lee et al., 2026; Lin et al., 2026).

Comparison with Prompt Evolving. Figure 7 compares LIFE-HARNESS with prompt-only evolving (Agrawal et al., 2025; Yang et al., 2024a), which iteratively optimizes only the input prompt. While prompt-only evolving provides modest gains, LIFE-HARNESS achieves substantially higher Pass@1 performance, adding an average relative improvement of 120%. This gap highlights a key property of agentic tasks: performance depends not only on the initial prompt, but also on how the runtime mediates tools, actions, feedback, and multi-step trajectories.

5.3 Ablation Study

All Lifecycle Layers Matter. Table 2 shows the leave-one-layer-out ablation results of LIFE-HARNESS. The results demonstrate that all four layers in LIFE-HARNESS are indispensable: removing any layer leads to substantial performance drops on some datasets. Moreover, different tasks benefit from different layers, reflecting the distinct characteristics of their task environments.

Does Model Training Remove the Need for Harnessing? Figure 8 compares specialized tool-use training with LIFE-HARNESS. We use xLAM-2-32B as a representative trained model, which is initialized from Qwen2.5-32B-Instruct and further trained on tool-use scenarios related to τ-bench. The results show three patterns. 1) Harnessing can outperform specialized training without weight updates: Qwen2.5-32B equipped with LIFE-HARNESS surpasses xLAM-2-32B by 7.5 percentage points on the in-domain τ-bench setting. 2) Harnessing remains beneficial after training: applying LIFE-HARNESS to xLAM improves all evaluated benchmark groups by $6.8 \sim 28.9$ percentage points. 3) Specialized tool-use training does not necessarily transfer to out-of-domain (OOD) agent environments: on τ^2-bench and AgentBench, xLAM underperforms its base Qwen2.5 model, suggesting reduced OOD generalization. We observe the same trend for xLAM-2-8B, trained from Llama-3.1-8B-Instruct.

These results suggest that training and harnessing adapt different parts of an agent system: training tunes model parameters to a source distribution, whereas LIFE-HARNESS adapts the runtime interface to a target deterministic environment. Thus, harnessing targets reusable environment-side structure and remains model-agnostic and complementary to model training.

6 Conclusion

We present LIFE-HARNESS, a lifecycle-aware runtime harness for deterministic LLM agents. Rather than updating model parameters, LIFE-HARNESS evolves reusable interface interventions from training trajectories, spanning environment contracts, procedural skills, action realization, and trajectory regulation. Across seven environments and 18 model backbones, LIFE-HARNESS achieves broad performance gains while keeping both model weights and evaluation environments fixed. These results suggest that many agent failures can be mitigated by adapting the runtime interface between frozen LLMs and rule-governed environments, providing a complementary alternative to model-centric agent training.

Limitations

This work focuses on deterministic, rule-governed agent environments where the tool interface, feedback rules, and evaluation criteria are relatively stable. This setting is common in database manipulation, web shopping, and policy-guided business workflows, and it enables failures to be reproduced and converted into reusable harness interventions. Related prior work on runtime harnessing has also primarily focused on coding or text-based agents, where the interface and task structure are well-defined (Lee et al., 2026; Lin et al., 2026). However, extending the same idea to fully open-ended agent tasks remains challenging. In open-domain settings (Ye et al., 2026), each task may involve different goals, tools, external resources, and success criteria, making it harder to define a stable runtime interface or evolve a harness that generalizes across arbitrary tasks. We consider harness construction in such open-ended environments to be an important avenue for future research.

References

Eshaan Agarwal, Joykirat Singh, Vivek Dani, Raghav Magazine, Tanuja Ganu, and Akshay Nambi. 2024a. Promptwizard: Task-aware prompt optimization framework. *arXiv preprint arXiv:2405.18369*.

Rishabh Agarwal, Nino Vieillard, Yongchao Zhou, Piotr Stanczyk, Sabela Ramos Garea, Matthieu Geist, and Olivier Bachem. 2024b. On-policy distillation of language models: Learning from self-generated mistakes. In *International Conference on Learning Representations*, volume 2024, pages 21246–21263.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, and 1 others. 2025. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*.

Anthropic. 2026. Claude code. https://claude.com/product/claude-code. Accessed: 2026-05-19.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-Bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. 2023. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*.

DeepSeek-AI. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv e-prints*, pages arXiv–2407.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

HarvardMIT Mathematics Tournament. 2026. Hmmt problem sets. https://www.hmmt.edu/. Accessed: 2026-05-19.

Minki Kang, Jongwon Jeong, Seanie Lee, Jaewoong Cho, and Sung Ju Hwang. 2026. Distilling llm agent into small models with retrieval and code tools. *Advances in Neural Information Processing Systems*, 38:106501–106538.

Seth Karten, Joel Zhang, Tersoo Upaa Jr, Ruirong Feng, Wenzhe Li, Chengshuai Shi, Chi Jin, and Kiran Vodrahalli. 2026. Continual harness: Online adaptation for self-improving foundation agents. *arXiv preprint arXiv:2605.09998*.

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*.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. 2025. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. In *International Conference on Learning Representations*, volume 2025, pages 28691–28735.

Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shihan Dou, Xuanjing Huang, Hang Yan, Zhenhua Han, and Tao Gui. 2026. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, and 1 others. 2024. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pages 52989–53046.

Kevin Lu and Thinking Machines Lab. 2025. On-policy distillation. *Thinking Machines Lab: Connectionism*. Https://thinkingmachines.ai/blog/on-policy-distillation.

OpenAI. 2026. Codex cli. https://developers.openai.com/codex/cli. Accessed: 2026-05-19.

OpenCode. 2026. Opencode: The open source ai coding agent. https://opencode.ai/. Accessed: 2026-05-19.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Manoj Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, and 1 others. 2026. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *Advances in Neural Information Processing Systems*, 38.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. 2023. Automatic prompt optimization with gradient descent and beam search. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 7957–7968.

Zehan Qi, Xiao Liu, Iat Long Iong, Hanyu Lai, Xueqiao Sun, Jiadai Sun, Xinyue Yang, Yu Yang, Shuntian Yao, Wei Xu, and 1 others. 2025. Webrl: Training llm web agents via self-evolving online curriculum reinforcement learning. In *International Conference on Learning Representations*, volume 2025, pages 79791–79821.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tur, Gokhan Tur, and Heng Ji. 2026. Toolrl: Reward is all tool learning needs. *Advances in Neural Information Processing Systems*, 38:105523–105553.

Qwen Team. 2026. Qwen3.5: Towards native multimodal agents.

Elad Sarafian, Gal Kaplun, Ron Banner, Daniel Soudry, and Boris Ginsburg. 2026. Workspace optimization: How to train your agent. *arXiv preprint arXiv:2605.09650*.

Biswa Sengupta and Jinhua Wang. 2026. Harbor: Automated harness optimization. *arXiv preprint arXiv:2604.20938*.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2020. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*.

Bangjun Xiao, Bingquan Xia, Bo Yang, Bofei Gao, Bowen Shen, Chen Zhang, Chenhong He, Chiheng Lou, Fuli Luo, Gang Wang, and 1 others. 2026. Mimo-v2-flash technical report. *arXiv preprint arXiv:2601.02780*.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, and 1 others. 2024. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094.

Tianshi Xu, Yuteng Chen, and Meng Li. 2026. Cleaner: Self-purified trajectories boost agentic reinforcement learning. *arXiv preprint arXiv:2601.15141*.

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, Howard Chen, John Yang, and Karthik Narasimhan. 2022. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*.

Bowen Ye, Rang Li, Qibin Yang, Yuanxin Liu, Linli Yao, Hanglong Lv, Zhihui Xie, Chenxin An, Lei Li, Lingpeng Kong, and 1 others. 2026. Claw-eval: Toward trustworthy evaluation of autonomous agents. *arXiv preprint arXiv:2604.06132*.

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*.

Tong Zheng, Haolin Liu, Chengsong Huang, Huiwen Bao, Sheng Zhang, Rui Liu, Runpeng Dai, Ruibo Chen, Chenxi Liu, Tianyi Xiong, and 1 others. 2026. Llms improving llms: Agentic discovery for test-time scaling. *arXiv preprint arXiv:2605.08083*.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pages 15585–15606.

A Detailed Harness

A.1 Failure Annotation Protocol

We annotate failed trajectories with the help of a coding agent, Codex (OpenAI, 2026). For each failed episode, Codex reads the complete interaction trace and assigns one primary failure category according to the rule-based protocol below. The classification is performed trajectory by trajectory, rather than from aggregate statistics. We use a priority-based protocol: the annotator first checks for action realization failures, then environment contract mismatches, then trajectory degeneration, and finally assigns the episode to the residual category. This priority order prevents later symptoms from hiding earlier interface failures. For example, if an agent writes a tool call in plain text, the environment never executes it, and the episode eventually exhausts the step budget, we annotate it as an action realization failure rather than trajectory degeneration.

Action realization failures. This category corresponds to failures targeted by the *Action Realization Layer*. A failed episode is assigned to this category if any of the following conditions hold:

- The assistant does not produce executable `tool_calls`, but instead writes in `content` an action or answer that should have been submitted through a tool, such as `take_action({...})`, `answer_action({...})`, a natural-language command, SQL text, or the final answer text.
- The assistant produces a tool call, but the call cannot be executed because of an invalid function name, JSON parsing failure, missing required arguments, or incorrect argument types.
- In DBBench, the SQL query is not executable due to interface, dialect, or formatting violations, such as unquoted column names containing spaces, malformed table names, or invalid concatenation syntax.

The key criterion is that the model's intent may be reasonable, but it is not submitted in a form executable by the environment.

Environment contract mismatches. This category corresponds to failures targeted by the *Environment Contract Layer*. It is considered only when the action interface is executable. A failed episode is assigned to this category if any of the following conditions hold:

- The agent uses the wrong tool for a critical step, such as calling a finish tool when an answer-submission tool is required.
- The tool choice or calling order violates the task protocol, such as submitting prematurely, skipping a required intermediate tool, or using a search tool to encode constraints that should not be searched directly.
- The argument has an incorrect semantic format, although it satisfies the JSON or schema-level interface. For example, in DBBench, the final answer may be required to be an exact value or list, but the agent submits a natural-language explanation, or multi-column results are concatenated into free-form text instead of the required format.

The key criterion is that the function call is structurally present and executable, but the agent misunderstands the tool's purpose, boundary, calling protocol, or argument semantics.

Trajectory degeneration. This category corresponds to failures targeted by the *Trajectory Regulation Layer*. It is considered only when neither action realization nor environment contract mismatch is the primary cause. A failed episode is assigned to this category if any of the following conditions hold:

- The episode ends due to `task limit reached` or an equivalent budget-exhaustion signal, and the trajectory contains a clear repetition pattern, such as repeatedly issuing the same action, oscillating between two states, repeatedly calling `look`, `inventory`, or `examine`, repeatedly searching or clicking, or repeatedly returning to the same page.
- The agent commits to an incorrect strategy early and reinforces it throughout the trajectory, such as searching once and purchasing without comparing candidates or required attributes, or repeatedly visiting already explored locations after failing to find the target.
- The environment repeatedly returns no-progress feedback, such as `Nothing happens` or an unchanged page/state over many steps.

The key criterion is that individual actions are usually executable, but the long-horizon interaction fails to use environment feedback to revise the strategy.

Residual reasoning failures. A failed episode is assigned to this category when none of the above categories apply. This category includes cases where the tool-use protocol is largely followed, but the agent makes an incorrect reasoning, computation, SQL, retrieval, or value-selection decision. It also includes cases where the agent retrieves the wrong object or answer for reasons not attributable to tool-protocol misunderstanding or trajectory loops, or where the correct task path is attempted but the final value, condition, or filtering logic is wrong.

A.2 Harness Evolution Prompt

We use a coding agent Codex (OpenAI, 2026) to evolve LIFE-HARNESS from training trajectories. The agent is given: (1) the current harness implementation, (2) a directory containing training trajectories and summary metrics from the previous iteration, and (3) a design guide describing the four lifecycle layers. The prompt used for harness evolution is shown below.

Prompt Template for Trajectory-Driven Harness Evolution

System Instruction.
You are a coding agent responsible for improving a runtime harness for a deterministic LLM-agent environment. Your goal is to improve task performance by adapting the runtime interface between the frozen model and the environment, without changing the model weights, the benchmark tasks, or the environment evaluation logic.

Inputs. You are given:

- the current harness implementation: `{HARNESS_DIR}`;
- a trajectory directory from the previous iteration, including the summary metrics: `{TRAJECTORY_DIR}`;
- the harness design guide: `{DESIGN_GUIDE}`. (The following content may either be included in this file or provided directly to the model.)

Harness Design Principles. The harness has four lifecycle layers:

1. **Environment Contract Layer**: clarify stable tool, action, policy, and answer-format constraints before interaction.
2. **Procedural Skill Layer**: retrieve compact procedural skills distilled from training trajectories and align them with the current task state.
3. **Action Realization Layer**: validate model-generated actions before execution, canonicalize unambiguous interface-level errors, and block actions that would deterministically fail.
4. **Trajectory Regulation Layer**: monitor post-execution trajectories, detect repeated failures or non-progressing behavior, and trigger recovery when needed.

Use these layers to address runtime-interface failures, not to solve tasks with hidden oracle information. The harness may expose stable environment-side structure, but it must not use test labels, modify benchmark tasks, alter environment transitions, or change evaluation criteria.

Analysis Requirements. Inspect the previous iteration's trajectories and identify recurring failure patterns. For each pattern, determine the earliest lifecycle point where it can be reliably detected or prevented:

- before interaction, via contract clarification;
- during task conditioning, via procedural skill retrieval;
- before environment execution, via action validation or canonicalization;
- after execution, via trajectory monitoring and recovery.

Focus on failures that are mechanically identifiable from deterministic environment signals, such as invalid action formats, wrong tool conventions, missing required fields, repeated no-op actions, loops, premature submissions, budget exhaustion, or recurring procedural mistakes.

Update Requirements. Propose and implement targeted updates to the appropriate harness layer. Each update should satisfy the following constraints:

- it should be triggered by precise environment or trajectory evidence;
- it should be as local and minimal as possible;
- it should not override model reasoning when the correct action is ambiguous;
- it should preserve the original environment and evaluation protocol;
- it should be robust to unseen tasks from the same environment.

Regression Check. After implementing updates, inspect cases where the harness may over-trigger, block a valid action, inject misleading guidance, or reduce performance on previously successful trajectories. Revise the harness if any layer introduces negative side effects.

Output. Return:

1. a concise summary of the dominant failure patterns found;
2. the harness layer responsible for each proposed update;
3. the implemented code changes;
4. a short explanation of why each update is safe under the deterministic environment contract;
5. any remaining failure modes that should be monitored in the next iteration.

A.3 Final Evolved Harness Inventory

This appendix summarizes the concrete harness components used in the seven evaluated scenarios. The inventory follows the four lifecycle layers described in the main method section. Detailed implementations of these components are provided in our submitted codebase: Anonymous GitHub. The τ-bench and τ^2-bench harness components are listed in Table 5, and the AgentBench components are listed in Table 6 and Table 7.

B Additional Experiments

Dataset statistics and splits. Table 3 reports the data statistics used in our experiments. For each environment, we distinguish between the raw training pool, the subset used for harness evolution, and the held-out test/evaluation set. The "Train Used" column denotes the number of training examples sampled for each harness-evolution run, rather than the full raw training pool executed in every run. This design keeps harness evolution efficient while ensuring that evaluation is performed on held-out tasks that are not used during harness construction. The "Used Total" column reports the total number of examples actually used in our experiments, and "Raw Pool Total" reports the full available pool size for each environment.

Detailed Evaluation Configuration Table 4 summarizes the detailed evaluation configuration. For the Procedural Skill Layer, we use only the top-1 retrieved skill across all experiments, preventing irrelevant skills from contaminating the model context.

Main Results. Table 8 reports the full results of LIFE-HARNESS across 18 model backbones and 7 benchmarks, showing consistent gains across diverse models and environments.

C AI Assistant Usage

We used AI assistants for language polishing and improving the clarity of the manuscript. For the experimental implementation, the iterative development of the runtime harness was assisted by the coding agent Codex. All experimental design, analysis, and final results were reviewed and verified by the authors.

Task	Raw Train Pool	Train Used for Harness Evolution	Test / Eval	Raw Pool Total
ALFWorld	3,150	100	109	3,259
DBBench	4,803	100	300	5,103
OS Interaction	1,000	100	144	1,144
WebShop	9,000	100	200	9,200
Airline	30	30	20	50
Retail	74	74	40	114
Telecom	74	74	40	114

Table 3: Dataset statistics for harness evolution and evaluation.

Table 4: Evaluation sampling and interaction-budget settings.

Benchmark	Temperature	Max tokens per step	Max step
ALFWorld	0.0	4096	50
DBBench	0.0	4096	15
OS	0.0	4096	8
WebShop	0.0	4096	20
Airline	0.0	2048	200
Retail	0.0	2048	200
Telecom	0.0	2048	200

Domain	Layer	Implemented harness content
Airline	Environment Contract Layer	search tools remind that flight search takes only origin, destination, and date; search results must be compared by requested cabin price, total travel time, and seat availability; cancellation tool lists the four valid cancellation conditions and transfers flying/landed flights to humans; update tool requires complete round-trip itineraries and reservation-wide cabin changes; booking tool states payment-method caps, five-passenger cap, bookable-flight status, insurance, and baggage allowance rules; certificate tool states eligibility, amount cap, and compensation timing; transfer tool lists the cases that are actually out of scope.
	Procedural Skill Layer	communicate exact charge/refund after every cancel, update, or booking write; when multiple reservations exist, inspect candidate reservation details before acting; use at most one certificate per reservation and split only when separate bookings still satisfy the user request; submit all outbound and return legs in one round-trip update; check booking time, airline cancellation, business cabin, and insurance before cancellation; calculate total card charge or budget impact before writing; scan all reservations for duplicate, same-day, or schedule-mixup requests.
	Action Realization Layer	`book_reservation` checks duplicate payment IDs, at most one certificate, one credit card, and three gift cards; `book_reservation` checks max five passengers, connected itinerary, available flight status, enough seats, correct baggage fee, and exact payment total; `cancel_reservation` checks flown-flight guard and the four cancellation conditions; `update_reservation_flights` blocks basic-economy flight changes, origin/destination/trip-type changes, missing return legs, post-departure cabin changes, certificates as payment, and insufficient gift-card capacity; `update_reservation_baggages` blocks baggage decreases and wrong nonfree-bag count; `update_reservation_passengers` blocks passenger-count changes; `send_certificate` checks user eligibility, real delayed/cancelled flight status, prior change/cancel for delay compensation, and amount multiple/cap.
	Trajectory Regulation Layer	`get_user_details` appends one-line summaries for every reservation when a user has several; `get_reservation_details` appends total paid by payment method, membership, insurance, cabin, passenger count, free-bag allowance, and current round-trip legs; `cancel_reservation` appends exact refund amount and destination payment instrument; `update_reservation_flights` appends exact extra charge or refund; `book_reservation` appends total charged by payment instrument.
Retail	Environment Contract Layer	`cancel_pending_order` description says the tool cancels the whole pending order and accepts only `no longer needed` or `ordered by mistake`; `modify_pending_order_items` description says item modification is once per pending order and replacement IDs must be same-product, different, and available; `modify_pending_order_payment` description requires a saved payment ID and enough gift-card balance for the full positive total; address tools distinguish profile address from order shipping address; exchange/return descriptions say one return or exchange per delivered order and exact item IDs from order details; return description clarifies `available=false` does not block returning an owned item; transfer description lists only out-of-scope cases.
	Procedural Skill Layer	count every variant with `available=true` for availability-count tasks; copy hidden target addresses from profile or previous orders before writing; for pending orders, do not use full-order cancellation when the user rejects that fallback; for bulk requests, inspect all candidate orders and group writes by order; exchange all requested items from the same delivered order in one call; compare numeric option values such as storage, zoom, size, or piece count; keep theme-scoped requests limited to matching items; use item modification, not a new order, for pending same-product replacements; after clear user confirmation, call the write tool instead of summarizing again.
	Action Realization Layer	`cancel_pending_order` checks order status and reason; `modify_pending_order_items` checks status, once-only modification, non-empty `item_ids`, non-empty `new_item_ids`, equal list lengths, old item IDs in the order, duplicate IDs against order quantity, replacement IDs in the catalog, same-product replacement, and not replacing with the same item; `modify_pending_order_payment` checks new method differs and gift card covers the full total; `exchange_delivered_order_items` checks delivered status, one exchange/return only, valid old/new IDs, duplicate old IDs, same-product replacement, and not same item; `return_delivered_order_items` checks delivered status, valid return items, duplicate return IDs, and refund to original method or user's gift card.
	Trajectory Regulation Layer	`get_order_details` appends status-specific routing, in-progress return/exchange or already-modified warnings, tracking numbers, and order total; `get_product_details` appends available variant count, cheapest/most-expensive available variants, and numeric option min/max summaries; `get_item_details` explains that out-of-stock only blocks new purchases, not returns/exchanges of owned items; exchange writes append price difference; return writes append refund amount and remaining-item total when relevant; cancel writes append refund amount; gift-card write paths can append updated gift-card balance; completion annotators mention other eligible orders only as conditional follow-up, not as permission to act.
Telecom	Environment Contract Layer	`send_payment_request` description says use it only for overdue bills and only one awaiting-payment bill at a time; `resume_line` description says all overdue bills must be paid and expired contracts require transfer; `refuel_data` description says line must be active, max 2 GB per call, and price/consent must be confirmed; `suspend_line` description says line must be active and the user must accept the monthly fee; `enable_roaming` description separates account-side enablement from the user's device-side Data Roaming toggle; transfer description requires an actual tool call for expired contracts, SIM PIN/in-store support, or issues outside the tool set.
	Procedural Skill Layer	data issues retrieve a quota-check skill that calls `get_data_usage` before chasing APN/VPN causes; abroad issues retrieve roaming skills that call `enable_roaming` when needed and always ask the user to toggle Data Roaming; service-loss issues retrieve the overdue-bill workflow: get bills, send payment request, wait for payment, resume line, reboot; MMS skills require WiFi Calling off, mobile data/APN/SIM/app permissions, and quota check before transfer; multiple-cause skills force the agent to continue after one fix if the symptom persists; line-selection skill uses the line matching the caller phone number; SIM PIN skill separates human transfer for PIN lock from fixable billing work; hard-persona skill switches to one short instruction per turn.
	Action Realization Layer	`get_customer_by_phone` checks NXX-NXX-XXXX phone format; write tools check `customer_id` format/existence and line ownership; `send_payment_request` checks bill ownership, overdue status, no expired-contract restoration trap, and no other awaiting-payment bill; `resume_line` checks line is suspended, all overdue bills are paid, and contract is not expired; `refuel_data` checks max 2 GB and active line status; `suspend_line` checks active line status; `enable_roaming` blocks when roaming is already enabled and redirects to device toggle; `disable_roaming` blocks when roaming is already disabled.
	Trajectory Regulation Layer	`get_customer_by_phone` appends the `line_id` matching the caller's phone when the customer has multiple lines; `get_data_usage` appends quota-exhausted warning and refuel suggestion; `get_details_by_id` appends roaming-off, quota-exhausted, suspended-line, or expired-contract notes; `get_bills_for_customer` appends overdue bill IDs and the send-payment-request to resume-line workflow; `send_payment_request` appends the required customer payment, verification, resume, and reboot steps; `enable_roaming` appends whether the account was newly enabled or already enabled and reminds the device-side Data Roaming toggle.

Table 5: τ-bench and τ^2-bench harness inventory under the four lifecycle layers.

Domain	Layer	Implemented harness content
ALFWorld	Environment Contract Layer	task parsing extracts task type, target object, destination receptacle, required transform, object count, and subgoal chain; task-order hints spell out the step order for pick/place, clean, heat, cool, look-at, and pick-two tasks inside `take_action`; action-tool schema patching appends the hint so the model sees admissible-action and task-order constraints on every call.
	Procedural Skill Layer	task-type skill retrieval filters the ALFWorld skill library and BM25-ranks matches; injectable contract skills tell the agent to pick up the object before cleaning/heating/cooling and to use `examine X with desklamp`, not plain `examine X`; non-injected library entries document pickup-then-deliver, two-object staging, unseen-location search, appliance-as-midpoint, "nothing happens" recovery, second-object traps, and loop breaking.
	Action Realization Layer	`pre_validate_action` compares the raw action with the current admissible list; `_gate_action` canonicalizes close string matches only when the verb is compatible; invalid non-navigation actions increment a counter and are blocked after the threshold; empty actions trigger explicit feedback; forced next actions from later monitors are executed only if still admissible and are discarded when the model chooses a task-critical action itself.
	Trajectory Regulation Layer	`WorldModel` updates inventory, current location, visited/unvisited locations, observed objects, target location, placed locations, destination locations, and lamp location after every observation; subgoal advancement moves through FIND, TAKE, CLEAN/HEAT/COOL, GOTO_DEST, PUT, lamp, and EXAMINE states; post-step monitoring catches empty turns, examine-without-lamp loops, oscillation, repeated open/close, dead-end look/inventory loops, put/take "nothing happens", and generic stalls; step guidance proposes a concrete admissible next action; budget checking warns when search is late and forces PUT only when the held item and admissible put action make completion deterministic.
WebShop	Environment Contract Layer	task parsing extracts item keywords, required color or color alternative, size, material, max price, quantity, style, measurement specs, and product category; page-type detection identifies home, search-result, and product pages from observations and clickables; page-state tracking records selected attributes, ASIN, current price, search queries, back-to-search loops, and page stalls; tool-description patching tells the model to search with concise keywords, click only visible options, select attributes before buying, and keep budget words out of search text.
	Procedural Skill Layer	WebShop skill retrieval first filters skills by product category, then BM25-ranks against the instruction; skills tell the agent that about 5% over budget can be acceptable, compound colors need exact-or-closest matching, petite/tall/plus sizes are combined options, measurements can appear under size or other attributes, food flavor and pack count are clickable variants, home items use dimension options, code prefixes such as `01# black` should be matched by visible text, and unavailable exact values should fall back to the closest option.
	Action Realization Layer	`pre_validate_action` accepts only WebShop search and click action syntax; search queries are cleaned to remove price/budget suffixes; `_fuzzy_match_click` maps near-miss clicks to current clickables; unknown tools and invisible click targets are blocked; repeated non-navigation clicks are blocked after the threshold; `_buy_now_precheck` blocks `click[buy now]` while visible required color, size, spec, or unselected defensive attribute groups remain; forced `click[buy now]` is cancelled if the precheck still fails.
	Trajectory Regulation Layer	post-step monitoring checks the first product-page title against the requested item category; detects repeated back-to-search and duplicate-query loops; warns once per ASIN when price exceeds budget beyond tolerance; detects product-page stalls and asks for an immediate buy-or-back decision; step guidance proposes initial searches, ranks visible search results, and builds an attribute checklist from requirements and clickables; budget checking warns near the end and forces `click[buy now]` only when the button is visible.

Table 6: AgentBench harness inventory under the four lifecycle layers (ALFWorld and WebShop).

Domain	Layer	Implemented harness content
OS Interaction	Environment Contract Layer	task parsing classifies the request as count-files, count-lines, count-matches, count-unique, largest, smallest, list, read-content, system-info, sum-size, average, mutate, or other; it extracts answer shape, target path, extension, recursive/non-recursive scope, case-sensitivity, time filter, and size filter; tool-description patching tells the model to call bash through the tool interface, inspect paths before broad commands, use `find`/`grep`/ `awk`/`sort`/`wc` with correct counting semantics, avoid dangerous commands, and submit only the requested answer.
	Procedural Skill Layer	OS skill retrieval tokenizes the task with stopword removal, filters by parsed task type, applies a BM25 score threshold, and force-inserts high-confidence skills such as case-insensitive matching; the skill library gives concrete command patterns for file counts by extension/time/size, recursive search, excluding directories, grep line counts, total line counts, unique fields/words, largest or smallest files, size totals, disk/memory/process questions, output truncation, bracketed/date grep, safe averages, hidden files, field extraction before `sort -u`, non-recursive counts, IP/status parsing, max-number frequency, atime versus mtime, LOC, and date-entry counts.
	Action Realization Layer	tool-call rescue converts malformed model text into bash or `answer_action` calls from JSON, keyword, positional, bare-command, and XML-like formats; `pre_validate_action` rejects dangerous shell patterns; repeated identical bash commands are blocked after the threshold; text-only loops trigger a forced answer when a plausible candidate exists or finish when none exists; `normalize_answer` strips units/prose for integer or size answers while preserving the required string/path answers; `note_answer_submitted` marks the task complete for later monitors.
	Trajectory Regulation Layer	bash-state tracking records bash history, raw output, truncation, errors, empty-output streaks, numeric candidates, string/path candidates, and implausible values; post-step monitoring handles truncated output, command-not-found, path/glob/permission errors, grep/find/xargs warnings, empty output, repeated bash loops, text-only loops, `grep -c` per-file counts that must be summed, post-`ls` false-zero cases, and budget force/warn; step guidance lints missing recursion, wrong case handling, file-versus-line counting, bad `find -path`, missing `tr` input, date extraction, and human-readable size mistakes; it also aggregates extension component counts, formats `wc` filename answers, verifies early numeric answers, and accepts correct zeroes, promotes clean string/path candidates, and gives first-turn templates.
DBBench	Environment Contract Layer	task parsing classifies the SQL request as SELECT, INSERT, UPDATE, DELETE, counting, ranking, MAX/MIN/-SUM/AVG/COUNT aggregation, comparison, or other; answer-shape detection decides scalar integer/float/string, single-column multi-row, multi-column multi-row, or mutation-hash answer; schema-map building mirrors DBBench identifier sanitization, 64-character truncation, and duplicate suffixing; schema-card injection supplies the exact sanitized table and column names; tool-description and system-prompt patching state MySQL syntax, mutation, schema-card, and commit-format requirements.
	Procedural Skill Layer	DB skill retrieval filters skills by parsed SQL task type, BM25-ranks them, and applies heuristic boosts for grouped answers, unique entities, and dataset-average updates, next/previous row lookup, currency-text comparison, and insert requests; skills give concrete reminders for backticking identifiers, MySQL `CONCAT`, casting text numerics, DESCRIBE/SHOW recovery, LIKE/OR filters, SELECT result shape, COUNT, ranking with ORDER BY/LIMIT, returning the requested column not the rank, SUM/AVG denominators, INSERT value order, and all columns, UPDATE/DELETE WHERE clauses, preview-then-mutate, multi-row commit formatting, bare numeric commits, NULL-as-zero, mutation-before-commit, date passthrough, GROUP BY/HAVING, BETWEEN, subqueries, apostrophe escaping, mutation verification, and the exact inserted text formats.
	Action Realization Layer	tool-call rescue extracts `execute_sql` or `commit_final_answer` from XML, JSON, keyword syntax, and truncated partial answers; automatic backtick repair wraps known sanitized table/column names while leaving string literals untouched; dialect repair converts common SQLite-style patterns to MySQL, where safe; dangerous SQL is blocked; commit gates prevent mutation tasks from committing before INSERT/UPDATE/DELETE succeeds, prevent empty non-mutation commits, and block scalar commits after clear multi-row results; answer-list normalization formats scalar and multi-row answers; NULL aggregate candidates become `0` when the evaluator expects zero; repeated SQL with a plausible candidate can force `commit_final_answer`.
	Trajectory Regulation Layer	SQL-state tracking records raw and normalized SQL history, result text, error kind, empty streak, candidate answer, answer shape, implausibility, mutation-attempt flag, row/column count, and text-only streak; post-step monitoring handles text-only tool loops, MySQL syntax errors, unknown columns, and unknown tables, mutation tasks that only ran SELECT, NULL aggregates, repeated empty mutation previews, repeated empty query results, identical SQL loops, and budget force/warn; step guidance promotes plausible candidates, explains the tuple-string format for multi-column rows, lints contains-without-LIKE, case-insensitive without LOWER, and ranking without ORDER BY, supplies first-turn SQL templates, and warns when numeric or scalar candidates look implausible.

Table 7: AgentBench harness inventory under the four lifecycle layers (OS Interaction and DBBench).

Model	AgentBench				τ-bench Airline			τ-bench Retail			τ^2-bench Telecom		
	ALFWorld	WebShop	OS	DBBench	pass@1	pass@3	pass^3	pass@1	pass@3	pass^3	pass@1	pass@3	pass^3
Qwen3-4B-Ins	0.1651	0.2950	0.2150	0.4400	0.3500	0.6500	0.0500	0.4800	0.7000	0.2500	0.2583	0.4750	0.0750
w/ LIFE-HARNESS	**0.8807**	**0.4550**	**0.3960**	**0.6500**	**0.6000**	**0.7000**	**0.5000**	**0.6300**	**0.8000**	**0.4500**	**0.5750**	**0.8750**	**0.3000**
Qwen3.5-4B	0.4312	0.3450	0.4236	0.5933	0.8500	0.9500	0.7000	0.7917	0.9250	0.6500	0.9750	1.0000	0.9250
w/ LIFE-HARNESS	**0.9266**	**0.4150**	**0.4931**	**0.7133**	**0.8667**	**0.9500**	**0.7500**	**0.8167**	0.8750	**0.7000**	**1.0000**	**1.0000**	**1.0000**
Qwen3.5-9B	0.5688	0.3750	0.4028	0.6067	0.8500	0.9000	0.7500	0.8333	0.9250	0.6750	0.9750	1.0000	0.9250
w/ LIFE-HARNESS	**0.9174**	**0.4400**	**0.5069**	**0.7033**	**0.8833**	**0.9000**	**0.8500**	0.8000	0.9000	**0.7000**	0.9667	**1.0000**	**0.9250**
Qwen2.5-7B-Ins	0.1284	0.3000	0.2500	0.5133	0.2000	0.4000	0.1000	0.2333	0.4250	0.0750	0.3500	0.6000	0.1500
w/ LIFE-HARNESS	**0.3486**	**0.4450**	**0.3542**	**0.6833**	**0.4167**	**0.5500**	**0.3500**	**0.2833**	**0.5500**	**0.1000**	**0.5500**	**0.7750**	**0.2750**
Qwen2.5-14B-Ins	0.4036	0.3950	0.2917	0.5300	0.1833	0.5000	0.0000	0.4417	0.6750	0.2000	0.4000	0.5750	0.2500
w/ LIFE-HARNESS	**0.4220**	**0.4500**	**0.4097**	**0.6600**	**0.4833**	**0.6000**	**0.3500**	**0.6167**	**0.8750**	**0.3750**	**0.5833**	**0.8500**	**0.2750**
Llama-3.1-8B-Ins	0.0550	0.2250	0.3125	0.1733	0.3000	0.4000	0.2500	0.0750	0.1250	0.0250	0.2417	0.3250	0.1500
w/ LIFE-HARNESS	**0.8257**	**0.4250**	**0.3333**	**0.4967**	**0.3333**	**0.4000**	**0.3000**	0.0667	**0.1250**	0.0000	**0.4250**	**0.5750**	**0.3500**
xLAM-2-3B	0.0275	0.1450	0.0694	0.2800	0.2000	0.4000	0.0500	0.4000	0.6750	0.2000	0.2583	0.4500	0.1000
w/ LIFE-HARNESS	**0.4312**	**0.3550**	**0.1111**	**0.5400**	**0.4000**	**0.4500**	**0.3000**	**0.5167**	0.6500	**0.3500**	**0.4667**	**0.8000**	**0.1500**
Llama-xLAM-2-8B	0.0092	0.1150	0.1389	0.2433	0.3667	0.6500	0.1500	0.6083	0.7750	0.4250	0.3167	0.5750	0.1000
w/ LIFE-HARNESS	**0.1193**	**0.4350**	**0.1597**	**0.6300**	**0.5500**	**0.6500**	**0.3500**	**0.6333**	**0.8750**	0.3750	**0.4417**	**0.8000**	**0.1750**
Qwen3-30B-A3B-Ins	0.1101	0.3500	0.4444	0.5767	0.4167	0.6000	0.2000	0.4583	0.6750	0.2250	0.3750	0.5750	0.1750
w/ LIFE-HARNESS	**0.8807**	**0.4650**	**0.4583**	**0.7333**	**0.6000**	**0.7000**	**0.5000**	**0.5500**	**0.7250**	**0.3250**	**0.7083**	**0.8750**	**0.5250**
Qwen3.5-27B	0.8899	0.3850	0.5417	0.6833	0.8833	0.9000	0.8500	0.8333	0.9500	0.6750	0.9417	1.0000	0.8750
w/ LIFE-HARNESS	**0.8899**	**0.4600**	**0.5833**	**0.7400**	0.8500	**0.9000**	0.8000	**0.8417**	0.9250	**0.7500**	**0.9833**	**1.0000**	**0.9500**
Qwen3.5-35B-A3B	0.7431	0.3800	0.5139	0.6133	0.8167	1.0000	0.7000	0.8417	1.0000	0.7000	0.9500	1.0000	0.8750
w/ LIFE-HARNESS	**0.9174**	**0.4300**	**0.5694**	**0.6933**	**0.8333**	0.9500	**0.7500**	0.8250	0.9000	**0.7250**	**0.9583**	**1.0000**	**0.9000**
Qwen3.6-35B-A3B	0.7890	0.3800	0.5208	0.6233	0.8500	0.9000	0.7500	0.8000	0.9000	0.6750	0.9917	1.0000	0.9750
w/ LIFE-HARNESS	**0.8991**	**0.4500**	**0.5417**	**0.7133**	**0.8833**	**1.0000**	**0.7500**	**0.8667**	**0.9750**	**0.7250**	**0.9917**	**1.0000**	**0.9750**
Qwen3.6-27B	0.9083	0.3850	0.5208	0.6833	0.8333	0.9500	0.7000	0.8417	0.9500	0.7250	0.9667	1.0000	0.9000
w/ LIFE-HARNESS	0.8991	**0.4300**	**0.5694**	**0.7433**	0.8000	**0.9500**	**0.7000**	0.8333	0.9250	0.6750	**0.9750**	**1.0000**	**0.9500**
Qwen2.5-32B-Ins	0.7064	0.3650	0.3681	0.5533	0.3333	0.4500	0.2000	0.5333	0.9250	0.2000	0.4500	0.5500	0.3000
w/ LIFE-HARNESS	**0.9266**	**0.4650**	**0.4583**	**0.6933**	**0.5167**	**0.6500**	**0.4000**	**0.6417**	0.8750	**0.3500**	**0.7917**	**0.9250**	**0.6500**
Qwen2.5-72B-Ins	0.7890	0.3600	0.4444	0.5167	0.4167	0.6500	0.2500	0.6167	0.8750	0.3000	0.4667	0.7000	0.1750
w/ LIFE-HARNESS	**0.8532**	**0.4800**	**0.4722**	**0.7133**	**0.6034**	**0.7000**	**0.4500**	**0.7000**	0.8500	**0.4750**	**0.7167**	**1.0000**	**0.4000**
Llama-3.3-70B-Ins	0.1835	0.3800	0.3403	0.5833	0.2667	0.3000	0.2500	0.0667	0.1000	0.0500	0.3083	0.3250	0.2750
w/ LIFE-HARNESS	**0.9266**	**0.4400**	**0.3958**	**0.6867**	**0.3000**	**0.3500**	**0.2500**	**0.0750**	0.0750	**0.0750**	**0.3333**	**0.3750**	**0.3000**
xLAM-2-32B	0.3761	0.2300	0.2639	0.0533	0.3833	0.6000	0.1500	0.6250	0.8750	0.3750	0.3917	0.6500	0.1250
w/ LIFE-HARNESS	**0.9174**	**0.4600**	**0.3472**	**0.3567**	**0.7000**	**0.8500**	**0.5000**	**0.7083**	**0.8750**	**0.5000**	**0.4583**	**0.7750**	**0.1250**
Llama-xLAM-2-70B	0.1101	0.2400	0.1875	0.4433	0.4500	0.6500	0.1500	0.6333	0.8000	0.4000	0.3333	0.5250	0.1250
w/ LIFE-HARNESS	**0.6422**	**0.4200**	**0.2500**	**0.4833**	**0.6500**	**0.7500**	**0.5500**	**0.7250**	**0.9250**	**0.5000**	**0.4917**	**0.7750**	**0.2500**

Table 8: Full main results across all evaluated model backbones and benchmarks. For AgentBench environments, we report pass@1 scores. For τ-bench and τ^2-bench environments, we report Pass@1, Pass@3, and Pass^3. Each model is evaluated with and without LIFE-HARNESS. Bold values indicate that LIFE-HARNESS improves or matches the corresponding no-harness result.