DIVIDE AND EVOLVE: ORCHESTRATING AGENT HARNESS EVOLUTION WITH DIAGNOSTIC MULTI-AGENT SYSTEM

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-driven agents increasingly rely on well-designed harnesses to execute complex long-horizon tasks, yet harness design flaws can recur across tasks and require costly human intervention to resolve. Harness evolution addresses this by automatically searching for and implementing harness modifications based on execution traces and task feedback, but effective evolution demands sound multi-stage decision coordination and principled control of modification boundaries and search directions, challenges that existing approaches fail to address simultaneously. To this end, we propose **DiVE**, a diagnosis-driven multi-agent harness evolution framework that integrates constraint-based boundary control and experience-guided search. **DiVE** assigns dedicated roles to harness analysis, repair planning, and patch implementation, propagating decision rationale through explicit intermediate outputs to support error localization, while incorporating program constraints, historical experience, and task feedback to jointly govern modification scope and search direction across iterations. Extensive experiments on τ^2-Bench and Terminal-Bench demonstrate that **DiVE** consistently improves harness performance across diverse task environments, and that small LLMs trained on complete evolution trajectories can also produce effective harness improvements, suggesting that structured evolution trajectories provide useful supervision for learning evolution capabilities.

1 INTRODUCTION

Large language model (LLM)-driven agents (Anthropic, 2025; OpenAI, 2025) are increasingly undertaking complex long-horizon tasks(Jimenez et al., 2024; Deng et al., 2025; Merrill et al., 2026), whose reliable execution depends heavily on the Harness (Lopopolo, 2026; Rajasekaran, 2026), a framework that manages context, tool invocation, and runtime state. As the execution layer connecting models to their environments, harness design flaws may recur across tasks, turning local issues into persistent execution bottlenecks. However, addressing these defects currently relies on human experts to continually diagnose and maintain harnesses, incurring escalating costs as models and environments evolve (Lin et al., 2026; Guo et al., 2026). To reduce this burden, enabling agents to refine their harnesses through recursive self-improvement (RSI) (Chen et al., 2026b) based on execution outcomes offers a promising approach to scaling agentic systems. However, automating this process raises a central question: *when harness modification is automated, how can execution failures guide effective improvements without accumulating new problems across iterations?*

Harness evolution offers a direct approach to this goal by using execution traces and task feedback to automatically search for and implement harness modifications. Existing studies explore automated harness search and self-modification (Lee et al., 2026; Zhang et al., 2026a), improve reliability through observability, modification constraints, and candidate screening (Lin et al., 2026; Zhang et al., 2026c; Luo et al., 2026), and use historical experience to guide subsequent optimization (Zhang et al., 2026d; Du et al., 2026; Guo et al., 2026). These advances enable automated RSI while highlighting the importance of sound evolution decisions: incorrect failure attribution can lead to ineffective modification strategies, implementations that deviate from their plans can disrupt existing behavior, and the resulting changes may become the starting point for subsequent iterations. An effective harness evolution system must therefore support inspection of modification rationales,

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

constrain the scope of changes, and use feedback to refine subsequent decisions. Organizing this process is a central challenge in achieving reliable, sustained harness evolution.

Effectively orchestrating such an evolutionary process requires addressing two interrelated challenges. **(i) Coordination of multi-stage decisions.** From execution feedback to effective code modification, the process involves interdependent stages of diagnosis, planning, and implementation. Deviations at any stage may propagate to the final output, yet the task score alone cannot reveal whether failures originate in diagnosis, planning, or implementation, making it necessary to delineate the responsibilities and intermediate outputs of each stage so that evolutionary errors can be localized. **(ii) Control of modification boundaries and search direction.** Sustained evolution requires both delimiting the modifiable scope and selecting exploration directions in a principled manner. Without coordination among program constraints, historical experience, and task feedback, the evolutionary process risks exhausting the evaluation budget on redundant attempts or achieving local gains at the cost of regression in other capabilities. However, existing approaches fail to address both challenges simultaneously, limiting the effectiveness and reliability of harness evolution.

To address the above challenges, we propose **DiVE** (**Di**vide and evol**VE**), a diagnosis-driven multi-agent harness evolution framework that integrates constraint-based boundary control and experience-guided search.**DiVE** assigns dedicated roles to harness analysis, repair planning, and patch implementation, enabling each stage to focus on stage-relevant information and propagate decision rationale through explicit diagnoses, plans, and code changes, making the reasoning chain inspectable and supporting error attribution across stages. For coordinated control of modification boundaries and search directions, the framework incorporates program constraints, historical experience, and task feedback into the iterative process, where constraints delimit the modifiable scope, experience guides diagnosis and strategy selection, and task feedback evaluates modification effectiveness and informs subsequent decisions. Together, these two mechanisms organize a complete evolutionary process from execution evidence to candidate modifications and effect verification. Extensive experiments on τ^2**-Bench** and **Terminal-Bench 2.1** show that **DiVE** improves average score and best score on τ^2-Bench by +0.51% and **+1.24%** (absolute), and Pass@1 and Pass Rate on Terminal-Bench by +6.5% and **+11.6%** (absolute). Furthermore, small LLMs trained on evolution trajectories can also produce effective harness improvements, suggesting that structured trajectories provide useful supervision for evolution capability learning.

The main contributions of this paper are summarized as follows:

- **A diagnosis-driven multi-agent evolution framework.** Evolution responsibilities are divided into failure diagnosis, repair planning, and code implementation, where explicit intermediate outputs connect execution evidence with modification decisions, supporting rationale inspection and error localization.
- **An evolution control mechanism integrating constraints, experience, and feedback.** Program constraints delimit the modifiable scope, historical experience guides subsequent strategy selection, and current task feedback verifies actual effectiveness, jointly governing the modification boundaries, search directions, and iterative basis of the evolutionary process.
- **Empirical evaluation across diverse task environments.** The proposed method is evaluated on τ^2-Bench and Terminal-Bench, with ablation studies analyzing the roles of the collaboration and control mechanisms, and further exploration of the value of complete evolution trajectories for training small models to acquire evolution capabilities.

2 Related Work

Harness Engineering. Harness engineering extends LLM reasoning into controllable and verifiable task execution by coordinating external context, tools, feedback, and runtime state. Early advances in chain-of-thought prompting (Wei et al., 2022) and ReAct (Yao et al., 2023), and Toolformer (Schick et al., 2023) laid the groundwork for reasoning and tool interaction. Building on these foundations, coding harnesses (Yang et al., 2024; Mini-SWE-agent, 2026) have explored both interface design and execution infrastructure. Broader platforms, including Claude Code (Anthropic, 2025), Codex (Bolin, 2026), and others (Anomaly, 2026; Mullen & Salva, 2025; Qwen Team, 2025; Wang et al., 2025), support repository-level workflows through closed-loop tool execution, environmental feedback, sandboxing, context management, and extensible interfaces. Beyond in-

dividual tasks, OpenClaw (Steinberger, 2026) and Hermes Agent (Nous Research, 2026) enable cross-session execution through multi-channel routing, persistent memory, and scheduling. Complementing these capabilities, DeepSeek Harness (DeepSeek-AI, 2026) integrates models, tools, and scheduling through composable plugins, with event logs supporting state recovery and execution tracing. Despite these advances, harness adaptation and maintenance remain largely manual, limiting cost-efficient development for large-scale, dynamic tasks.

Harness Evolution. To alleviate manual harness engineering costs, harness evolution automatically optimizes agent harnesses along three directions. The first is harness search. Meta-Harness searches harness code (Lee et al., 2026), while subsequent studies improve search reliability and evaluation through edit constraints, generalization requirements, and candidate screening (Lin et al., 2026; Luo et al., 2026; Zhou et al., 2026; Wang et al., 2026). However, these searches often navigate the modification space blindly without explicit failure attribution. The second is experience utilization. These methods organize execution experience into policies, memories, or state graphs (Zhang et al., 2026d; Du et al., 2026) and refine search through experience calibration, distillation, simulation feedback, or mechanism optimization (Guo et al., 2026; Chen et al., 2026a; Zhang et al., 2026b). While historical experience guides macro-level strategy, it struggles to localize micro-level code errors. The third is training models for evolution. Harness-R1 trains a dedicated harness engineer through reinforcement learning with task-success feedback (Shao et al., 2026). However, these end-to-end designs entangle failure diagnosis, strategy planning, and code implementation, increasing the difficulty of error attribution and process supervision. In contrast, **DiVE** separates these roles, constrains edits through programmatic checks and task feedback, and uses complete evolution trajectories to teach smaller LLMs, supporting controllable and learnable harness evolution.

3 DiVE

3.1 Problem Formulation and Overview

Let M denote a fixed task-execution model and H_t the agent harness at evolution step t. Executing M under H_t on the evolution task set $\mathcal{D}_{\text{evo}}$ produces trajectories $\mathcal{T}_t = \{\tau_i\}_{i=1}^{N}$, each recording model responses, tool use, environment observations, and task feedback. Harness evolution seeks to improve task performance by modifying H_t while preserving model parameters and evaluation logic. A direct mapping $(H_t, \mathcal{T}_t) \rightarrow H_{t+1}$ leaves the intermediate decisions implicit, making it difficult to examine how observed failures motivate particular code changes. Effective evolution requires identifying harness-level problems, selecting appropriate repair strategies, and implementing executable modifications.

As illustrated in Figure 1, **DiVE** makes these decisions explicit through three specialized roles: the *Harness Analyst*, *Repair Planner*, and *Patch Developer*. Given the current harness, its structural summary, execution evidence, modification constraints, and historical experience $\mathcal{B}_t$, these roles organize candidate generation as

$$\mathcal{T}_t \rightarrow W_t \rightarrow P_t \rightarrow \Delta H_t \rightarrow \widetilde{H}_t, \qquad \widetilde{H}_t = H_t \oplus \Delta H_t, \tag{1}$$

where W_t contains consolidated diagnoses and supporting evidence, P_t specifies the repair plan, ΔH_t is the proposed patch, and $\oplus$ denotes patch application. This decomposition lets each role focus on its stage-specific objectives, while explicit intermediate outputs support rationale inspection and error localization across diagnosis, planning, and implementation.

3.2 Diagnosis-Driven Multi-Agent Collaboration

Moving from execution feedback to effective code changes involves evidence analysis, strategy selection, and implementation. These stages are interdependent but differ in information requirements and evaluation criteria, and interleaving them within a single process allows insufficiently supported diagnoses to directly shape implementation, making it difficult to attribute ineffective modifications to problem identification, strategy selection, or coding errors. **DiVE** therefore separates evolution into three specialized stages connected through explicit diagnoses W_t, repair plans P_t, and code changes ΔH_t, with each stage handled by a dedicated agent: the *Harness Analyst*, the *Repair Planner*, and the *Patch Developer*.

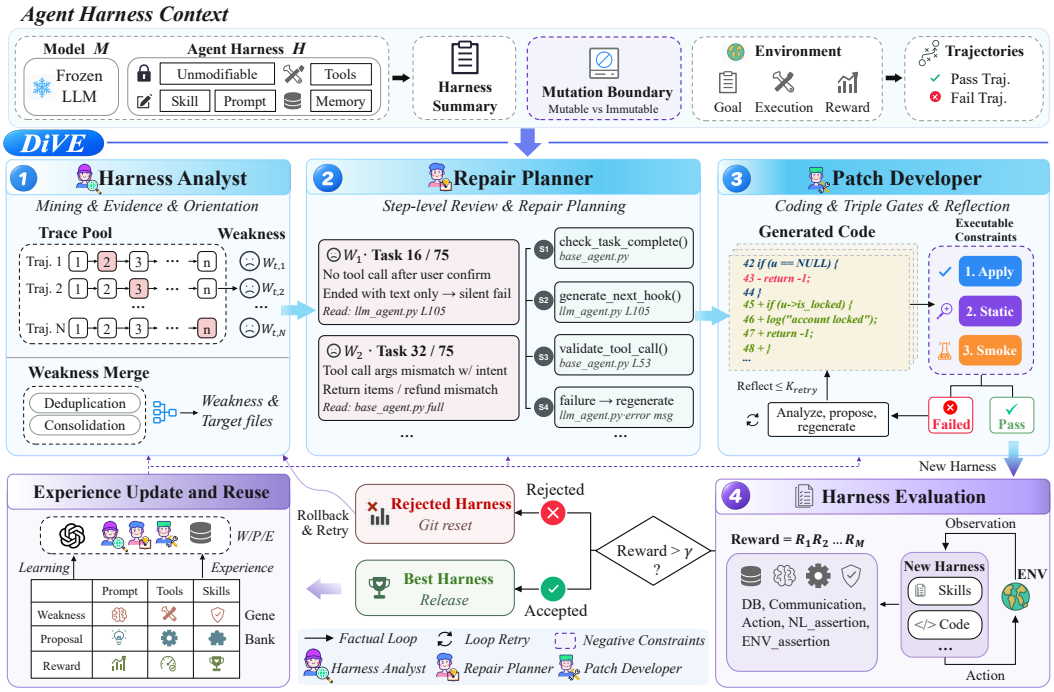

Figure 1: Overview of **DiVE**. The *Harness Analyst*, *Repair Planner*, and *Patch Developer* collaborate to diagnose failures, plan repairs, and implement modifications. Programmatic constraints bound changes and accumulated experience informs subsequent evolution.

Harness Analyst. An isolated failure may reflect an incidental error rather than a recurring harness defect. The Harness Analyst therefore examines successful and failed trajectories alongside the harness summary $S(H_t)$, modification constraints $\mathcal{C}$, and relevant historical experience $\mathcal{B}_t^A$. It first extracts candidate issues from individual trajectories, then consolidates repeated findings:

$$\widehat{W}_{t,i} = \mathcal{A}\big(\tau_i, S(H_t), \mathcal{C}, \mathcal{B}_t^A\big)\,,$$
$$W_t = \mathrm{Merge}\left(\bigcup_{i=1}^{N} \widehat{W}_{t,i}\right). \tag{2}$$

Each consolidated diagnosis is represented as $w_j = (d_j, \mathcal{E}_j, \mathcal{F}_j)$, where d_j describes the issue, $\mathcal{E}_j$ contains supporting trajectory evidence, and $\mathcal{F}_j$ identifies relevant files or components. Successful traces provide behavioral references, while failed traces expose potential defects. Evidence consolidation reduces reliance on individual anomalies and avoids redundant planning. The resulting diagnoses remain hypotheses to be assessed through subsequent modification and evaluation.

Repair Planner. A diagnosis does not uniquely determine a repair, since different strategies may affect different components and behaviors. The Repair Planner separates strategy selection from coding by inspecting relevant harness code and producing an explicit plan:

$$P_t = \mathcal{P}\big(W_t, H_t, \mathcal{C}, \mathcal{B}_t^P\big) = (p_1, \ldots, p_L). \tag{3}$$

Each planned operation is represented as $p_\ell = (\mathcal{I}_\ell, f_\ell, a_\ell, b_\ell)$, where $\mathcal{I}_\ell$ identifies the diagnoses it addresses, f_ℓ specifies the target component, a_ℓ describes the modification, and b_ℓ states the intended behavior. The sequence records the implementation order where dependencies exist. These explicit links allow the strategy and scope to be examined before coding and provide a reference for detecting omitted or unrelated changes.

Patch Developer. An appropriate repair strategy can still fail through incorrect implementation. The Patch Developer therefore focuses on implementing P_t within the permitted scope:

$$\Delta H_t^{(0)} = \mathcal{E}\big(P_t, H_t, \mathcal{C}, \mathcal{B}_t^E\big)\,, \qquad \widetilde{H}_t^{(0)} = H_t \oplus \Delta H_t^{(0)}. \tag{4}$$

Each candidate undergoes patch application, static checks, and smoke tests. If a check fails, its feedback $F_t^{(k)}$ guides a local revision:

$$\Delta H_t^{(k+1)} = \mathcal{E}\Big(P_t, H_t, \Delta H_t^{(k)}, F_t^{(k)}, \mathcal{C}, \mathcal{B}_t^E\Big), \quad 0 \le k < K_{\text{retry}}. \tag{5}$$

Here, $k = 0$ denotes the initial patch, and each revised patch is defined relative to H_t. A passing candidate proceeds to task evaluation, while exhaustion of the retry budget ends the local revision process. This separation supports correction of implementation errors while preserving the original diagnosis and plan for rationale inspection.

Together, these roles establish explicit connections among evidence, strategy, and implementation. When a candidate fails to deliver the expected gains, the intermediate outputs support inspection of whether the diagnosis was sufficiently grounded, the plan addressed the diagnosis, and the code correctly implemented the plan. This organization supports error tracing and targeted correction while making evolution decisions easier to examine.

3.3 Evolution under Constraints and Experience Guidance

Multi-agent factorization makes the formation of modification decisions explicit. However, candidate patches may still exceed the permitted scope, repeat ineffective attempts, or introduce regressions. To control evolution, **DiVE** uses *negative constraints* to delimit feasible modifications and *positive guidance* to prioritize search directions using successful and failed experiences. Task evaluation then determines candidate acceptance, and experience updates inform subsequent iterations.

Negative Constraints. Generated modifications may violate predefined boundaries or fail to execute because of implementation errors. **DiVE** therefore constrains both modification scope and program executability, exposing detectable problems before full task evaluation.

- *Mutation Boundary.* Harness modifications may involve prompts, tool interfaces, skills, and execution logic, while task definitions and evaluation mechanisms must remain unchanged. **DiVE** explicitly distinguishes editable and immutable components and provides this boundary information to all evolution roles. Candidate changes must satisfy:

$$\text{Changed}(\Delta H_t) \subseteq \mathcal{F}_{\text{edit}}, \qquad \widetilde{H}_t\big|_{\mathcal{F}_{\text{fixed}}} = H_t\big|_{\mathcal{F}_{\text{fixed}}},$$

  where $\mathcal{F}_{\text{edit}}$ and $\mathcal{F}_{\text{fixed}}$ denote the editable and immutable files or components, respectively. Boundary information guides planning, while programmatic checks verify actual changes. This enforces scope restrictions throughout planning and implementation and restricts attempts to obtain apparent gains by altering evaluation logic.
- *Executable Constraints.* Candidates within the permitted scope must also pass patch application, static checks, and smoke tests. These checks assess whether the patch applies, satisfies static validity requirements, and supports basic execution:

$$G(\widetilde{H}_t) = g_{\text{apply}}(\widetilde{H}_t) \wedge g_{\text{static}}(\widetilde{H}_t) \wedge g_{\text{smoke}}(\widetilde{H}_t).$$

  When a check fails, specific errors are returned to the Patch Developer for local revision, with at most K_{retry} retries. Candidates that fail these checks do not proceed to full task evaluation, reducing expenditure on non-executable modifications. Passing the checks establishes basic validity, while performance gains require task evaluation.

Positive Guidance. Programmatic constraints define permissible changes but do not determine which directions merit exploration. **DiVE** therefore uses successful and failed experiences as complementary evidence for diagnosis, planning, and implementation:

- *Successful Experience* includes effective behaviors observed in successful trajectories and harness modifications validated through improved task performance. The former identifies execution patterns worth preserving, helping avoid disruption of existing behavior. The latter connects modification strategies to observed outcomes and provides references for subsequent planning.
- *Failure Experience* records unsuccessful attempts and their feedback, including patch errors, runtime exceptions, task failures, and reward regressions. Implementation errors guide local code revision, while task failures and regressions motivate reassessment of diagnoses and strategies. Retaining these records allows rejected candidates to inform subsequent search, reducing repeated exploration without new supporting evidence.

Harness Evaluation. After passing programmatic checks, a candidate harness $\widetilde{H}_t$ is evaluated with the fixed task model on $\mathcal{D}_{\text{evo}}$. Its task outcomes are aggregated into the evaluation score $R(\widetilde{H}_t)$. Evaluation includes tasks beyond those that motivated the modification, allowing execution traces to expose unresolved issues and potential regressions. The candidate is retained only if $R(\widetilde{H}_t) > \gamma$. Otherwise, the workspace is restored to the previous version. Task-level outcomes and traces are retained to guide subsequent diagnosis and planning.

Experience Update and Reuse. To support continued guidance from previous exploration, **DiVE** links each round's diagnoses, plans, code changes, and validation feedback in an experience record:

$$e_t = \left(W_t, P_t, \Delta H_t, F_t, R(\widetilde{H}_t), a_t\right),$$

where F_t denotes checking and evaluation feedback, and a_t indicates candidate acceptance. Each record preserves the modification rationale, implementation, and observed outcome. It is incorporated into the historical experience $\mathcal{B}_{t+1}$, allowing subsequent decisions to reference the basis and effects of earlier strategies.

In subsequent evolution rounds, each role receives relevant experience: prior issues and evidence inform diagnosis, strategy outcomes guide planning, and implementation feedback supports patch development. Every candidate still undergoes programmatic checks and task evaluation, with new feedback updating the experience for further evolution.

3.4 Evolution Capability Learning

While larger LLMs can optimize harnesses within the evolution framework, smaller models may still struggle with system-level diagnosis and repair planning, generating syntactically correct patches while failing to address the underlying defects. Given their practical value in resource-constrained settings, we apply supervised fine-tuning on structured evolution trajectories, converting each stage into a learnable objective to enhance their harness evolution capabilities.

Concretely, the decomposed structure of **DiVE** explicitly preserves the inputs and outputs of each stage. We filter high-quality evolution trajectories based on validation outcomes and construct stage-level supervision data as follows:

$$\mathcal{D}_{\text{SFT}} = \bigcup_{t \in \mathcal{I}} \left\{ (x_t^A, W_t),\ (x_t^P, P_t),\ (x_t^E, \Delta H_t) \right\},$$

where $\mathcal{I}$ denotes the index set of selected records. Each input contains the role instruction, relevant harness context, and stage-available evidence, with supervision targets corresponding to the diagnostic output, repair plan, and code modification respectively. This organization preserves the connections among execution evidence, modification rationale, and code implementation, providing stage-level supervision for smaller models. Fine-tuning is performed by minimizing:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\frac{1}{|\mathcal{D}_{\text{SFT}}|} \sum_{(x,y) \in \mathcal{D}_{\text{SFT}}} \sum_{j=1}^{|y|} \log p_\theta(y_j \mid x, y_{<j}).$$

The fine-tuned model operates within the **DiVE** framework, retaining its program constraints, feedback-based correction, and candidate selection mechanisms. Structured evolution trajectories thus serve both as a record of the evolution process and as a unified interface for training evolution capabilities. Data construction and training details are provided in the appendix A.

4 Experiments

4.1 Experimental Setup

Tasks and Datasets. We evaluate harness evolution on two benchmarks spanning agent and code tasks, **τ^2-bench** (Barrès et al., 2025) and **Terminal-Bench 2.1** (Merrill et al., 2026). For τ^2-bench, we consider three domains: Airline (20 tasks), Retail (40 tasks), and Telecom (40 tasks), covering heterogeneous tool-use and multi-turn interaction scenarios. For Terminal-Bench 2.1, we evaluate

Table 1: Main results on τ^2-bench and Terminal-Bench 2.1. For τ^2-bench, the Base Harness is the benchmark's native harness, while for Terminal-Bench 2.1, NexAU$_0$ is used as the Base Harness. For evolution-based methods, green and red arrows denote absolute improvements and degradations over the corresponding Base Harness. Bold values indicate the best result.

A. τ^2-bench *Evaluation across Airline, Retail, and Telecom domains*

Method	Airline (20)		Retail (40)		Telecom (40)		Full	
	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
DeepSeek Harness	80.00	86.25	85.16	88.12	93.59	94.38	87.50	90.25
Codex	82.19	85.00	82.81	86.25	**100.00**	**100.00**	89.56	91.50
Claude Code	76.88	80.00	87.50	89.40	99.08	**100.00**	90.01	91.76
Base Harness	83.75	86.30	86.88	87.50	99.85	**100.00**	91.44	92.26
AHE	81.90↓1.85	**88.70**↑2.40	87.33↑0.45	88.70↑1.20	99.10↓0.75	**100.00**	90.95↓0.49	93.22↑0.96
DiVE (*Ours*)	**85.03**↑1.28	**88.70**↑2.40	**87.95**↑1.07	**90.00**↑2.50	99.40↓0.45	99.40↓0.60	**91.95**↑0.51	**93.50**↑1.24

B. Terminal-Bench 2.1 *Evaluation across Difficulty Levels*

Method	Easy		Med.		Hard		Full	
	Pass@1	Pass%	Pass@1	Pass%	Pass@1	Pass%	Pass@1	Pass%
DeepSeek Harness	**75.0**	**75.0**	48.8	58.1	22.7	36.4	42.0	52.2
Codex	**75.0**	**75.0**	50.0	58.1	20.5	22.7	42.0	47.8
Claude Code	**75.0**	**75.0**	50.0	51.2	25.0	36.4	43.5	47.8
Base Harness	**75.0**	**75.0**	46.5	51.2	20.5	36.4	39.9	47.8
AHE	**75.0**	**75.0**	50.0↑3.5	60.5↑9.3	**29.5**↑9.0	40.9↑4.5	44.9↑5.0	55.1↑7.3
DiVE (*Ours*)	**75.0**	**75.0**	**52.3**↑5.8	**65.1**↑13.9	**29.5**↑9.0	**45.5**↑9.1	**46.4**↑6.5	**59.4**↑11.6

on the full benchmark and further report results across Easy, Medium, and Hard difficulty levels. For τ^2-bench, we use the benchmark-provided native harness as the base harness, while for Terminal-Bench 2.1 we use NexAU$_0$ as the initial harness. All harness evolution is conducted exclusively on a separate evolution split, disjoint from the final test set. The held-out test tasks are never exposed during evolution and are used only for final evaluation, ensuring that reported gains reflect generalizable harness improvements rather than test-set adaptation.

Evolution Setting. We explicitly decouple the model used for *harness evolution* from the model executed inside the target agent. Unless otherwise specified, we use **GLM-5.2**[1] as the evolution backbone responsible for failure diagnosis, evolution planning, and harness modification, while the task-solving agent is consistently instantiated with **DeepSeek-V4-Flash**[2].

Metrics. For τ^2-bench, we report average and maximum task success scores per domain and an aggregate **Full** score over Airline, Retail, and Telecom weighted by task count (20/40/40). For Terminal-Bench 2.1, we report **Pass@1** and **Pass%** across difficulty levels and the full benchmark. For evolution-based methods, we additionally report the absolute performance change relative to the corresponding Base Harness. Further evaluation details are provided in the Appendix B.1.

Baselines. We compare **DiVE** with both human-written harnesses and automatic harness evolution methods. Human-written baselines include the benchmark Base Harness, **DeepSeek Harness**, **Codex**, and **Claude Code**, where results are available. For automatic evolution, we compare with **AHE** (Lin et al., 2026), a representative observability-driven method. To ensure a controlled comparison, evolution-based methods start from the same Base Harness and use the same task-solving model when applicable. The Base Harness is the native implementation for τ^2-bench and NexAU$_0$ for Terminal-Bench 2.1. Additional setup details are provided in the Appendix B.2.

[1] https://huggingface.co/zai-org/GLM-5.2

[2] https://deepseek.com/en/news/v4-preview/

Table 2: End-to-end versus three-stage harness evolution on τ^2-Bench. Full scores are weighted by domain task counts (20/40/40). Δ denotes the absolute gain of DiVE in percentage points.

	Airline		Retail		Telecom		Full	
Method	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
End-to-End	77.18	81.20	79.70	82.50	**99.40**	**100.00**	87.08	89.24
DiVE	**85.03**	**88.70**	**87.95**	**90.00**	**99.40**	99.40	**91.95**	**93.50**
Δ	+7.85	+7.50	+8.25	+7.50	0.00	−0.60	+4.87	+4.26

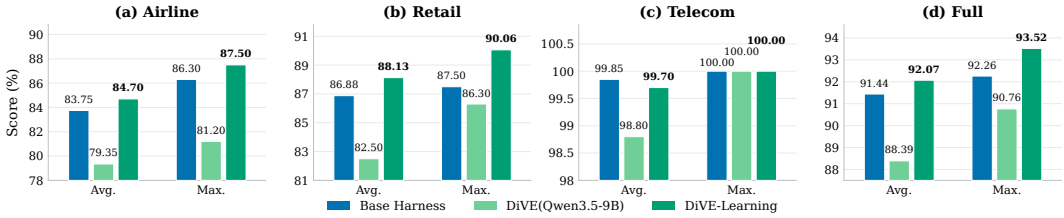

Figure 2: Learning harness evolution with Qwen3.5-9B. Average and maximum scores are compared for the Base Harness and harnesses evolved by Qwen3.5-9B before and after supervised fine-tuning. Full denotes the task-weighted aggregate across the three domains.

4.2 Overall Performance

Table 1 compares **DiVE** with strong manually designed harnesses and an automatic harness-evolution baseline on both τ^2-bench and Terminal-Bench 2.1. Overall, **DiVE** achieves the strongest aggregate performance across the two benchmarks. On τ^2-bench, **DiVE** improves the Base Harness from 91.44 to 91.95 in Full Avg. and from 92.26 to 93.50 in Full Max., corresponding to gains of $+0.51$ and $+1.24$, respectively. The improvements are particularly clear in Airline and Retail, where **DiVE** increases the average score by $+1.28$ and $+1.07$, and the maximum score by $+2.40$ and $+2.50$. In comparison, AHE slightly decreases the Full Avg. score from 91.44 to 90.95, despite improving the Full Max. score to 93.22. These results indicate that **DiVE** provides not only stronger peak performance, but also more consistent improvements across repeated evaluations.

4.3 Learning to Evolve with a Small Language Model

To investigate whether harness evolution is a learnable capability, we train Qwen3.5-9B on high-quality evolution trajectories collected by **DiVE** and deploy it for harness evolution within the **DiVE** framework (see Appendix A for details). Before fine-tuning, Qwen3.5-9B performs poorly at harness evolution. After training, its evolved harnesses improve Full maximum performance from 92.26% to 93.50%, approaching the evolution performance of GLM-5.2, with maximum-score gains of 2.80 and 1.93 percentage points on Airline and Retail, respectively (Figure 2). These results demonstrate that small LLMs can acquire effective evolution behaviors through training, validating the capability-learning design of **DiVE**: structured evolution trajectories provide explicit supervision that links failure diagnosis, improvement planning, and code modification.

4.4 Ablation Study

To assess the value of explicit stage decomposition in **DiVE**, we compare its diagnosis, planning, and implementation pipeline with end-to-end harness evolution on τ^2-Bench. **DiVE** improves Full Avg. and Full Max. by 4.87% and 4.26%, respectively, with average gains of 7.85% on Airline and 8.25% on Retail, while Telecom remains unchanged on average. These results support the effectiveness of staged evolution, suggesting that explicit diagnoses and repair plans help ground code modifications in execution evidence. By making intermediate decisions explicit, this decomposition also provides a basis for checking whether implementation follows the repair plan and addresses the failures.

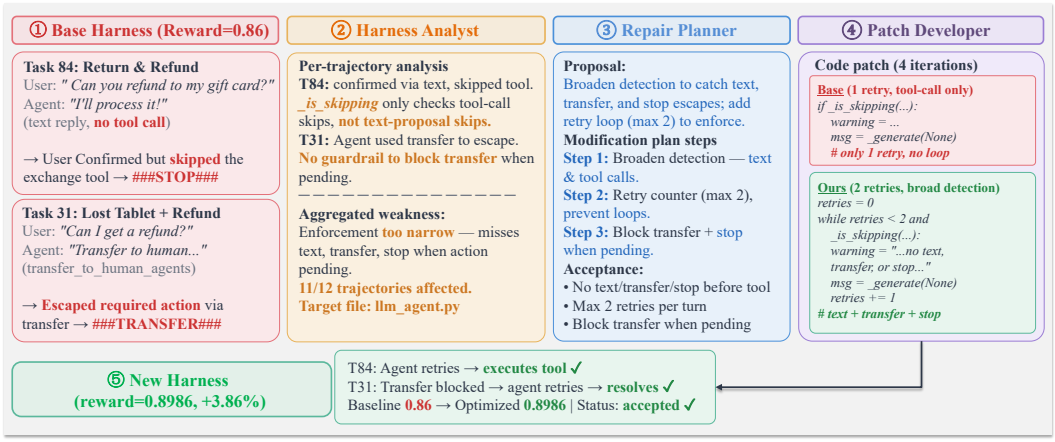

Figure 4: Case study of diagnosis-driven harness repair. **DiVE** identifies skipped required actions, broadens detection, and introduces bounded retries to prevent premature transfers and termination.

4.5 Analysis

Transferability of DiVE Evolved Harness. We evaluate **DiVE** with different execution backbones, including DeepSeek-V4-Flash and Kimi-K2.6. As shown in Figure 3, **DiVE** consistently improves harness performance across both models, demonstrating that evolved harnesses capture transferable system-level improvements rather than backbone-specific optimization. Notably, gains are more pronounced for weaker backbones: Kimi-K2.6 achieves +10.05% in Full score, while DeepSeek-V4-Flash shows stable improvement despite its stronger baseline. These results suggest that better harness design can partially compensate for weaker execution capabilities.



Figure 3: Transferability of the evolved **DiVE** across different execution LLMs.

Case Study. To understand why **DiVE** improves task execution, we trace how recurring failures guide harness refinement. As shown in Figure 4, diagnostic analysis of 12 trajectories reveals that 11 exhibit incomplete action enforcement, where pending tool calls are bypassed via text responses, transfers, or premature termination. Guided by this diagnosis, four successive patch iterations broaden skip detection, block premature exits, and introduce up to two retries per action. The refined harness resolves the identified failures and raises the reward from 0.86 to 0.8986, a relative improvement of 3.86%. This case illustrates how aggregated failure evidence enables targeted control-flow repairs, improving execution reliability by enforcing required actions and supporting recovery from transient errors.

5 Conclusion

We presented **DiVE**, a diagnosis-driven framework for harness evolution that combines specialized multi-agent collaboration with programmatic constraints, historical experience, and task feedback. Explicit intermediate outputs make modification rationales traceable, while constraints and feedback guide iterative refinement. Experiments on τ^2-Bench and Terminal-Bench demonstrate its effectiveness, and trajectory-based fine-tuning further shows that smaller models can learn useful evolution behaviors. These results support structured, controlled harness evolution as a practical approach to improving agent execution.

AI USE STATEMENT

Generative AI assisted with manuscript organization, language editing, and figure and table formatting to improve the clarity of author-written content. The research ideas and methodological design originated from the authors, who take full responsibility for the experimental results, claims, and references.

REFERENCES

Anomaly. OpenCode: Providers. Official documentation, 2026. URL https://opencode.ai/docs/providers/. Documentation accessed 2026-09-25.

Anthropic. Claude 3.7 Sonnet and Claude Code. Official announcement, feb 2025. URL https://www.anthropic.com/news/claude-3-7-sonnet. Published 2025-02-24; accessed 2026-09-25.

Victor Barrès, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025. doi: 10.48550/arXiv.2506.07982. URL https://arxiv.org/abs/2506.07982.

Michael Bolin. Unrolling the Codex agent loop. OpenAI Engineering, jan 2026. URL https://openai.com/index/unrolling-the-codex-agent-loop/. Published 2026-01-23; accessed 2026-09-25.

Kaiyuan Chen, Shuangyu Xie, Letian Fu, Justin Yu, William Pacini, Sandeep Bajamahal, Hudson Kim, Jaimyn Drake, Daehwa Kim, Haoru Xue, Jonathan Francis, Christian Juette, Peter Schaldenbrand, Muhammet Yunus Seker, Ruwan Wickramarachchi, Uksang Yoo, Guanzhi Wang, Adithyavairavan Murali, Balakumar Sundaralingam, S. Shankar Sastry, Spencer Huang, Yuke Zhu, Linxi "Jim" Fan, and Ken Goldberg. Gap: A graph-as-policy multi-agent self-learning harness for variational automation tasks. *CoRR*, abs/2607.05369, 2026a. doi: 10.48550/ARXIV.2607.05369. URL https://doi.org/10.48550/arXiv.2607.05369.

Mingguang Chen, Licheng Wang, and Bo Qu. Recursive self-improvement in ai: From bounded self-refinement to autonomous research loops, 2026b. URL https://arxiv.org/abs/2607.07663.

DeepSeek-AI. DeepSeek Harness: Everything is a plugin. Official project website, 2026. URL https://deepseek.com/harness/en/. Developer preview; accessed 2026-09-25.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Yuetian Du, Yucheng Wang, He Xu, Jiexu Xu, Shanwen Tan, Bing Zhao, Boyu Yang, Zhijie Xu, Ming Kong, Hu Wei, Jie Liu, and Qiang Zhu. Living-harness is an interactive-agent evolver. *arXiv preprint arXiv:2607.26598*, 2026.

Hanghui Guo, Weijie Shi, Zhangze Chen, Shengxiang Xu, Yishu Wang, Yimei Zhang, Wangze Ni, Jia Zhu, and Shimin Di. DREvo: Distilling recalibrated historical experience for harness self-evolution. *arXiv preprint arXiv:2607.26722*, 2026.

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

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

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

Ryan Lopopolo. Harness engineering: Leveraging Codex in an agent-first world. OpenAI, February 2026. URL https://openai.com/zh-Hans-CN/index/harness-engineering/. Published February 11, 2026. Accessed September 25, 2026.

Xiaotian Luo, Dizhan Xue, Fengxingyu Wang, Chuanrui Hu, and Yafeng Deng. HarnessBank: Semantic gene-bank search with gated verification for agent-harness self-evolution. *arXiv preprint arXiv:2607.13683*, 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868. Section 3.1 and Appendix F describe Terminus 2.

Mini-SWE-agent. mini-SWE-agent: The minimal ai software engineering agent. Official software repository, 2026. URL https://github.com/SWE-agent/mini-swe-agent. Repository accessed 2026-09-25.

Taylor Mullen and Ryan J. Salva. Gemini CLI: Your open-source AI agent. Google Blog, jun 2025. URL https://blog.google/innovation-and-ai/technology/developers-tools/introducing-gemini-cli-open-source-ai-agent/. Published 2025-06-25; accessed 2026-09-25.

Nous Research. Hermes Agent. Official software repository, 2026. URL https://github.com/NousResearch/hermes-agent. Accessed 2026-09-25.

OpenAI. Codex cli. https://github.com/openai/codex, 2025. Official GitHub repository; accessed 2026-09-24.

Qwen Team. Qwen3-Coder: Agentic coding in the world. Official technical blog, jul 2025. URL https://qwenlm.github.io/blog/qwen3-coder/. Includes the initial Qwen Code announcement, 2025-07-22; accessed 2026-09-25.

Prithvi Rajasekaran. Harness design for long-running application development. Anthropic, March 2026. URL https://www.anthropic.com/engineering/harness-design-long-running-apps. Published March 24, 2026. Accessed September 25, 2026.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.

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

Peter Steinberger. Introducing OpenClaw. OpenClaw Blog, jan 2026. URL https://openclaw.ai/blog/introducing-openclaw. Published 2026-01-29; documents the project's origin in November 2025; accessed 2026-09-25.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Daniel Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/a4b6ad6b48850c0c331d1259fc66a69c-Abstract-Conference.html.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents. *arXiv preprint arXiv:2607.12227*, 2026.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R. Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

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

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-harness: Harnesses that improve themselves. *arXiv preprint arXiv:2606.09498*, 2026a.

Jiayi Zhang, Yongfeng Gu, Jianhao Ruan, Maojia Song, Yiran Peng, Zhiguang Han, Jinyu Xiang, Zhitao Wang, Caiyin Yang, Yixi Ouyang, Bang Liu, Chenglin Wu, and Yuyu Luo. Harnessing agentic evolution. *arXiv preprint arXiv:2605.13821*, 2026b.

Luan Zhang, Ruochen Zhou, Dandan Song, Zhengyu Chen, Yuhang Tian, Jun Yang, Huipeng Ma, Chenhao Li, Guangyuan Feng, Xudong Li, Yizhou Jin, and Yan Xu. Harnesscompass: Guiding automatic harness evolution toward generalizable and effective agent harnesses. *arXiv preprint arXiv:2608.01918*, 2026c.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations (ICLR)*, 2026d.

Leijun Zhou, Zhihao Liu, Xiang Qu, Chenxu Liu, Yifei Liu, Yanke Yu, Jingzhe Xu, Xuejun Wu, Buyue Qian, Xi Chen, Yaowei Zheng, and Junhao Hu. GDPevo: Evaluating agent self-evolution on real business tasks. *arXiv preprint arXiv:2608.03764*, 2026.

A Learning to Evolve with Small LLMs

To train a small LLMs with harness evolution capability, we adopt a unified data construction and training pipeline on τ^2-Bench and Terminal-Bench, comprising execution trace collection, mixed sample construction, evolution trajectory generation, reward-based filtering, and supervised fine-tuning. The two benchmarks use their respective task environments and evaluators, while all other processing steps remain identical.

(1) Execution Trace Collection. The initial harness is executed on the training tasks of both benchmarks, using the native harness for τ^2-Bench and NexAU_0 for Terminal-Bench. Collected traces include agent outputs, tool calls, and environment feedback, covering domain tool interactions and terminal operations respectively. Task outcomes and rewards are then obtained using each benchmark's evaluator and associated with the corresponding traces. Data from each batch is aggregated by benchmark and domain, yielding a trajectory pool containing both successful and failed cases.

(2) Mixed Trajectory Sample Construction. Successful and failed cases are sampled from the trajectory pool; each evolution input typically consists of 2–3 failed trajectories and 1–2 successful ones, together with the corresponding harness content. Samples for τ^2-Bench are constructed per domain, while Terminal-Bench applies the same composition strategy within its task set. Failed cases expose execution bottlenecks, while successful ones serve as behavioral references to identify problems requiring modification and effective strategies worth retaining. Combining multiple trajectories provides richer diagnostic evidence and reduces reliance on any single execution outcome.

(3) Multi-Agent Evolution Trajectory Generation. The constructed samples are fed into **DiVE**, which sequentially performs failure diagnosis, improvement planning, and code implementation. The diagnosis stage localizes problems by jointly analyzing traces and harness content; the planning stage formulates concrete modification proposals; and the implementation stage applies the modifications and performs validation. Both benchmarks record the inputs, outputs, and validation feedback at each stage, producing complete evolution trajectories from execution evidence to

candidate harness modifications, which serve as supervised examples of diagnosis, planning, and implementation for subsequent training.

(4) Reward-Based Data Filtering. Samples are filtered according to the validation reward produced during the evolution process, retaining only records whose reward falls within $[0.75, 1.0]$. Both benchmarks apply identical filtering criteria, with validation feedback generated by their respective task environments and evaluation pipelines. This step ties data selection to actual evolution outcomes, discarding samples that fail to meet the prescribed reward threshold and thereby controlling the quality of the resulting supervised data.

(5) Training Data Integration and Supervised Fine-Tuning. Filtered evolution records from both benchmarks are aggregated and formatted into the input–output format supported by LLaMA-Factory[3]. Inputs contain the execution evidence, harness content, and available intermediate results required at each stage, while outputs correspond to the target stage's diagnosis, plan, or modification. Qwen3.5-9B is then fine-tuned with supervised learning to acquire the ability to translate execution feedback from diverse task environments into actionable harness improvements. After training, the model is integrated into **DiVE** and used to drive evolution on both benchmarks; the resulting harnesses are evaluated under a fixed task-execution model to assess whether the learned evolution behavior yields tangible performance gains.

B Experimental Details

B.1 Evaluation Details

τ^2-Bench. On τ^2-Bench, evaluation is conducted on the test split of each domain, with agent, user simulator, and environment interactions coordinated through the standard half-duplex orchestrator. Both the task execution model and the user simulator use DeepSeek-V4-Flash, with temperature fixed at 0. Each task is evaluated 16 trails with a maximum concurrency of 64, a maximum of 200 simulation steps per trajectory, and a consecutive tool error limit of 10. Evaluation components requiring LLM-based judgment are configured with GLM-5.2, with temperature set to 0 for both natural language assertions and environment interfaces. Evaluation uses the default random seed 300, disables LLM caching, and saves execution traces and task outcomes for subsequent analysis.

Terminal-Bench 2.1. The 89 tasks in Terminal-Bench 2.1 are partitioned into 20 evolution tasks and 69 held-out test tasks using random seed 42. The evolution split supports harness optimization, while the test split is reserved for final evaluation. Evaluations run through Harbor with the NexAU adapter in Docker environments. The task execution model is DeepSeek-V4-Flash, with temperature 0.7, top-p 0.95, and a maximum output length of 32,000 tokens per model response. Each task is evaluated over two trials. Let $r_{i,j} \in \{0, 1\}$ denote the verifier outcome of attempt j on task i, where N is the number of tasks and $K = 2$. Following the repeated-rollout evaluation in AHE (Lin et al., 2026), Pass@1 estimates single-attempt success by averaging outcomes across all tasks and attempts:

$$\text{Pass@1} = \frac{1}{NK} \sum_{i=1}^{N} \sum_{j=1}^{K} r_{i,j}. \tag{6}$$

Pass Rate measures the fraction of tasks solved in at least one of the two attempts:

$$\text{Pass Rate} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1} \left[\sum_{j=1}^{K} r_{i,j} > 0 \right]. \tag{7}$$

Both metrics are reported as percentages.

booktabs tabularx array seqsplit

[3] https://github.com/hiyouga/LlamaFactory

Table 3: Terminal-Bench task partition, comprising 20 evolution tasks and 69 held-out test tasks with no overlap.

Evolution Split (20 tasks)
bn-fit-modify build-cython-ext caffe-cifar-10
configure-git-webserver constraints-scheduling dna-assembly
financial-document-processor gcode-to-text git-leak-recovery
make-mips-interpreter merge-diff-arc-agi-task model-extraction-relu-logits
polyglot-c-py polyglot-rust-c qemu-alpine-ssh
query-optimize reshard-c4-data sqlite-db-truncate
train-fasttext write-compressor
Test Split (69 tasks)
adaptive-rejection-sampler break-filter-js-from-html build-pmars
build-pov-ray cancel-async-tasks chess-best-move
circuit-fibsqrt cobol-modernization code-from-image
compile-compcert count-dataset-tokens crack-7z-hash
custom-memory-heap-crash db-wal-recovery distribution-search
dna-insert extract-elf extract-moves-from-video
feal-differential-cryptanalysis feal-linear-cryptanalysis
filter-js-from-html
fix-code-vulnerability fix-git fix-ocaml-gc
git-multibranch gpt2-codegolf headless-terminal
hf-model-inference install-windows-3.11 kv-store-grpc
large-scale-text-editing largest-eigenval llm-inference-batching-scheduler
log-summary-date-ranges mailman make-doom-for-mips
mcmc-sampling-stan modernize-scientific-stack mteb-leaderboard
mteb-retrieve multi-source-data-merger nginx-request-logging
openssl-selfsigned-cert overfull-hbox password-recovery
path-tracing path-tracing-reverse portfolio-optimization
protein-assembly prove-plus-comm pypi-server
pytorch-model-cli pytorch-model-recovery qemu-startup
raman-fitting regex-chess regex-log
rstan-to-pystan sam-cell-seg sanitize-git-repo
schemelike-metacircular-eval sparql-university sqlite-with-gcov
torch-pipeline-parallelism torch-tensor-parallelism tune-mjcf
video-processing vulnerable-secret winning-avg-corewars

B.2 Baseline Settings

Manual Harness. We compare **DiVE** with three mainstream agent harnesses: Codex, DeepSeek Harness (DSH), and Claude Code. For τ^2**-Bench**, all harnesses are implemented as subclasses of τ^2's native `HalfDuplexAgent`, integrating with its standard orchestrator loop. For Codex and Claude Code, the domain tools are exposed as dynamic tool definitions; during execution, each tool call pauses the current turn, yields control to the τ^2 orchestrator for execution and logging, and resumes once the result is returned. DSH, whose SDK provides no pausable tool surface, instead runs each turn to completion via a dedicated MCP server process that executes tools against a live environment, buffers the tool-call decisions, and replays them one-by-one to the orchestrator for trajectory recording. For **Terminal-Bench**, Claude Code, Codex, and DSH are integrated into Harbor via dedicated adapters. Each adapter subclasses Harbor's `BaseAgent` or one of its derivatives, and is responsible solely for run configuration, task dispatch, process lifecycle management, and execution trace collection. During task execution, the model inference and tool interaction loops are handled entirely by each framework's own native runtime.

Base Harness. The two benchmarks adopt τ^2's native harness and NexAU_0 as their respective base frameworks. For τ^2-Bench, the native harness employs a turn-based tool-calling structure built on `LLMAgent`, taking domain rules, tool definitions, and conversation history as model inputs to generate either user-facing responses or tool call requests. A standard orchestrator coordinates the agent, user simulator, and domain environment, executes tool calls, and returns results, forming a

Table 4: Hyperparameters of the AHE.

Block	Hyperparameter	Value
Agent Debugger	Backing model	GLM-5.2
	Concurrent tasks	16
	Per-task timeout	600 s
	Retry attempts	3
	Retry backoff	2 s
Evolve Agent	Backing model	GLM-5.2
	Temperature	0.7
	Max context	200,000 tokens
	Max generation per turn	32,000 tokens
	Max model turns	500
	Skill packages	3
	Context compaction threshold	0.75
Explore Agent	Backing model	GLM-5.2
	Wall-clock budget	60 min
	Web sources	10
	Code sources	1
Outer loop / dispatcher	Iterations T	10
	Rollouts per task k	2
	Concurrent rollouts	96
	Sandbox	Local Docker container
	Sandbox lifetime	3,600 s

closed interaction loop that continuously updates the dialogue state. For Terminal-Bench, NexAU_0 is a minimal coding agent built on the NexAU framework, driven by a system prompt and interaction history, and interacts with the environment solely through a single Bash tool. This base configuration includes no additional skills, middleware, sub-agents, or long-term memory, providing a lean and extensible starting point for subsequent harness evolution.

AHE. Agentic Harness Engineering (AHE) is an automated harness evolution method driven by observable execution feedback, which iteratively refines the agent harness while keeping the underlying model fixed. It operates across three levels of observability: components, experience, and decisions. System prompts, tools, middleware, and other components are organized as independently editable files; the Agent Debugger distills execution traces into per-task diagnostic reports and an overall summary; and the Evolve Agent modifies the harness based on this evidence. Each modification is annotated with its intended effect and validated through subsequent task evaluations, forming an iterative loop of execution, analysis, modification, and verification. Detailed experimental parameters are provided in Table 4.