

ADIAS: AUTOMATED DESIGN OF INTERACTIVE AGENTIC SYSTEMS

Lekang Jiang*
University of Cambridge
lj408@cam.ac.uk

Bohan Tang*
LIGHTSPEED
bohantang@global.tencent.com

Stephan Goetz
University of Cambridge
smg84@cam.ac.uk

Yiwen Guo†
Independent Researcher
guoyiwen89@gmail.com

ABSTRACT

Automated agent design improves agent harnesses through iterative revision, evaluation, and feedback summarization. Existing methods are largely candidate-centric: cross-round experience is organized around candidate agents, which leaves the repair progress implicit. This causes inefficient repair targeting, slow consolidation of partial progress, and propagation of ineffective interventions across rounds. Therefore, we formulate **issue-centric agent optimization**, in which repair progress is carried forward as an explicit persistent issue state to guide optimization, rather than re-derived from candidate history in each round. We instantiate the formulation in **ADIAS**, a framework for automated full-code agent design with two mechanisms. A persistent issue state maintains stable issue identities, lifecycle status, supporting evidence, and intervention-outcome histories. Issue-guided optimization uses this state to jointly propose repair targets and revision directions for subsequent focused full-code modification. Across five interactive benchmarks, ADIAS outperforms the strongest baseline by 25.2% on average and achieves consistent gains across four backbone models. Controlled ablations further show that removing persistent issue state or replacing issue-centric revision with candidate-centric policies leads to performance drops of up to 40.7%.¹

1 INTRODUCTION

Large language models (LLMs) are increasingly deployed through agentic systems that augment a backbone model with planning, memory, tool use, and other mechanisms (Huang et al., 2024; Wang et al., 2024a). Together, these mechanisms form an agent harness that orchestrates the model’s interaction with the environment (Weng, 2026; Wang et al., 2026a). Since effective harnesses are task-dependent and labor-intensive to design manually, recent work has turned to *automated agent design*, which automatically generates and improves agent systems using execution feedback (Madžar & Mekterović, 2026; Gao et al., 2026; Yue et al., 2026; Ning et al., 2026).

Automated agent design generally proceeds through iterative *revise-evaluate-summarize* rounds. At each round, the optimizer revises an agent design to address issues revealed by prior evaluations, evaluates the resulting candidate to obtain scores and trajectories, and summarizes these outcomes into a cross-round history for subsequent revision (Hu et al., 2025; Zhang et al., 2025b; Lee et al., 2026; Lin et al., 2026a; Chen et al., 2026; Zhang et al., 2026a).

Existing methods can be characterized as *candidate-centric agent optimization*: cross-round experience is organized around candidate agents, and the objective of each round is to propose a higher-quality next candidate agent. For example, in each round, Meta-Harness generates new candidates

*Equal contribution.

†Corresponding author.

¹Code is available at <https://github.com/scyljl1/adias/>.

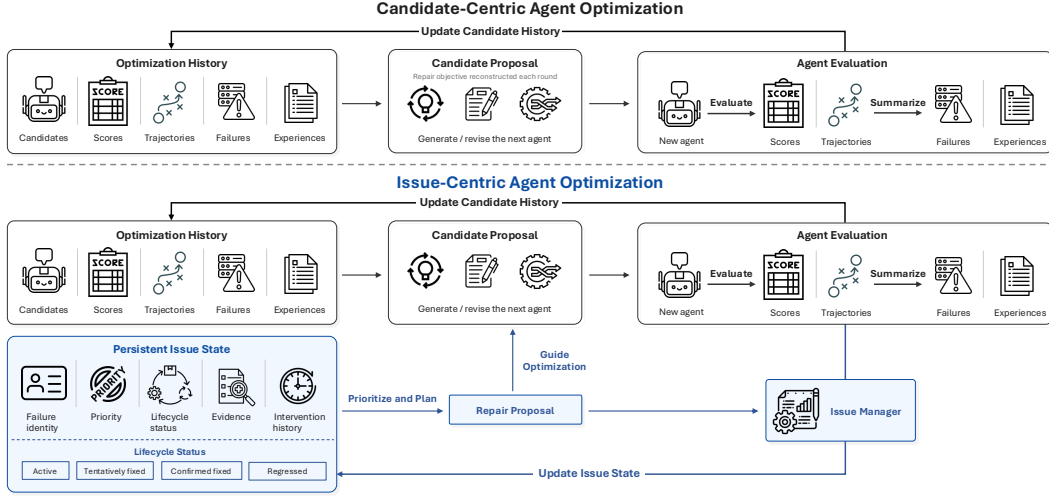


Figure 1: Comparison between candidate-centric agent optimization and issue-centric optimization.

based on the code, scores, and raw execution traces of previous agents (Lee et al., 2026), while HarnessX summarizes execution evidence to propose new revisions to obtain a better candidate agent (Chen et al., 2026). Although these methods may retain rich behavioral evidence and optimization experience, they do not explicitly maintain the lifecycle and repair progress of persistent issues as optimization state.

However, organizing cross-round experience around candidates makes the repair progress implicit, which leads to three limitations. First, *inefficient repair targeting*: evidence about whether an issue remains unresolved, what aspect should be modified, and which interventions have already been attempted is scattered across candidate records. The optimizer must reconstruct this repair context before each revision, often resulting in broad, redundant, or misdirected changes. Second, *fragmented repair-progress consolidation*: because each candidate may affect multiple issues and is evaluated as a whole, the outcome of a particular intervention is difficult to isolate, while complementary progress on the same issue remains distributed across different candidates. Third, *regressive intervention propagation*: because interventions and their outcomes are recorded at the candidate level, beneficial and harmful changes remain entangled, which allows some ineffective interventions to be carried into subsequent rounds.

To address these limitations, we introduce **issue-centric agent optimization**, which organizes cross-round experience around persistent issues rather than candidate agents. Evaluation outcomes and summarized feedback are accumulated into a persistent issue state that explicitly records each issue’s current status, supporting evidence, and intervention history. This state directly guides the next repair target and revision direction, which enables more focused repair decisions, consolidation of partial progress across rounds, and avoidance of previously ineffective interventions. Figure 1 contrasts this paradigm with candidate-centric agent optimization.

Realizing issue-centric agent optimization raises two key challenges. First, *reliable issue-state maintenance* requires transforming heterogeneous execution trajectories and sparse outcomes into stable, evidence-grounded issue identities, while reconciling new observations with historical records and tracking the repair progress of each issue across rounds. The same underlying issue may manifest differently across candidates, while a single revision may simultaneously improve some behaviors and regress others, which complicates both identity association and outcome attribution. Second, *actionable and attributable revision* requires translating a dynamic, multi-issue, and primarily descriptive state into a concrete code modification. The target issue and revision direction must be determined jointly according to the issue’s current evidence and prior intervention history, and the resulting modification must remain sufficiently focused for its subsequent behavioral outcome to be attributed to the intended repair.

To address these challenges, we propose **ADIAS** (Automated Design of Interactive Agentic Systems), an issue-centric agent optimization framework with two mechanisms. First, a *persistent*

issue state associates newly diagnosed failures with stable issue identities and tracks their supporting evidence, lifecycle transitions, and intervention-outcome histories. This structure combines heterogeneous failure modes across rounds while tracking the outcome of the targeted issues separately from improvements or regressions observed on other issues. Second, *issue-guided optimization* jointly selects target issues and revision directions from the current state, and realizes this plan through a focused full-code modification. Joint planning makes the descriptive issue state actionable for code improvement, while focused revision improves the attribution of subsequent behavioral outcomes without restricting the agent design space.

Our main contributions are:

- We introduce **issue-centric agent optimization**, a new paradigm that organizes cross-round experience around persistent issues and uses explicit issue state to guide optimization.
- We instantiate this paradigm in **ADIAS**, a full-code agent-design framework that combines a persistent issue state for maintaining cross-round repair progress with issue-guided optimization for proposing and executing focused revisions.
- We show that ADIAS outperforms the strongest baseline by 25.2% on average across five interactive benchmarks and demonstrates consistent improvements across four backbone models. Controlled ablations further illustrate the importance of issue-centric optimization: removing the persistent issue state or replacing issue-guided revision with candidate-centric policies results in performance drops of up to 40.7%.

2 RELATED WORK

Recent work has increasingly shifted from manually designing agentic systems toward automatically improving them from execution feedback (Madžar & Mekterović, 2026; Gao et al., 2026; Yue et al., 2026; Ning et al., 2026). We review this literature along two dimensions: the expressiveness of the artifact being optimized, and the information preserved to coordinate improvement across rounds.

Automated Agent Design. Prior work automates agent design at increasing levels of expressiveness. Prompt-level methods optimize instructions or textual modules (Zhou et al., 2022; Pryzant et al., 2023; Yang et al., 2024; Agrawal et al., 2026; Khattab et al., 2024). Architecture-level methods search over predefined workflows, module compositions, or structured harness primitives (Li et al., 2024; Zhang et al., 2025c; Zhuge et al., 2024; Shang et al., 2025; Chen et al., 2026). Full-code methods directly generate or modify executable agent implementations, using iterative archives, evolutionary exploration, self-modification, or test-time feedback (Hu et al., 2025; Zhang et al., 2025b; Wang et al., 2025; Lee et al., 2026; Lou et al., 2026; Cai et al., 2026; Nie et al., 2026). ADIAS operates at the full-code level but differs in its search-state control.

Experience and Reusable Knowledge. A line of work improves agents by accumulating reusable experience. Reflection and experiential-learning methods retain lessons or heuristics derived from previous trajectories, while systems such as ACE and SkillOpt maintain editable playbooks or skill documents (Shinn et al., 2023; Zhao et al., 2024; Zhang et al., 2026c; Yang et al., 2026). Other approaches persist and evolve structured knowledge, workflow, tool, or validation artifacts across iterations (Lin et al., 2026b; Huang et al., 2026; Lin et al., 2026a; Wang et al., 2026a). These methods demonstrate the value of carrying information across rounds. However, the persistent object is primarily reusable knowledge or an editable agent artifact. ADIAS instead maintains the evolving state of the failures that motivate those edits, including their status, evidence, and intervention history.

Continual and Regression-Aware Agent Improvement. Recent work shows that iterative agent improvement is not necessarily monotonic. Continual adaptation can degrade previously acquired capabilities across workflow, skill, memory, and model evolution (Yu et al., 2026). GRASP (Moll et al., 2026) applies regression-aware acceptance to edits of a bounded skill library, while SKILL.nb (Hattami et al., 2026) uses lifecycle governance and validation gates to improve the durability of reusable workflows. These approaches primarily govern whether an updated skill or workflow artifact should be retained. ADIAS addresses a different but complementary problem: maintaining continuity of the repair objective itself during full-code agent evolution.

3 METHOD

3.1 TASK DEFINITION AND CANDIDATE-CENTRIC OPTIMIZATION

Task Definition. We study the automated design of an interactive agent for a fixed task environment, evaluator, and backbone model, all of which are isolated from the agent-design workspace. Let $A \in \mathcal{A}$ denote an executable agent implementation in the admissible full-code design space $\mathcal{A}$. Executing A on an instance $x \sim P_q$ from split $q \in \text{train, val, test}$ produces a trajectory $\mathcal{T}(A, x) = (z_1, \dots, z_L)$, where $z_h = (o_h, a_h, o_{h+1}, r_h, d_h)$ records the observation o_h , action a_h , environment response o_{h+1} , task score r_h , and termination indicator d_h at interaction step h . Task feedback is typically sparse: intermediate rewards are often uninformative, while the final task outcome is commonly binary, with 1 indicating successful completion and 0 otherwise. We use $M_q(A)$ to denote the aggregate performance of agent A on split $q \in \{\text{train, val, test}\}$.

Starting from an initial implementation A_0 and an optimization budget of T rounds, the design system produces a new candidate A_t and evaluates its validation performance $M_{\text{val}}(A_t)$ at each round $t = 1, \dots, T$. A diagnostic process then analyzes the training trajectories and performance to produce a diagnostic report D_t , which summarizes observed failures and optimization-relevant experience for subsequent rounds. After T rounds, the system selects the candidate in $A_0, \dots, A_T$ with the highest validation performance and evaluates it on the held-out test set. The objective is to achieve high test performance M_{test} (Wang et al., 2026b).

Candidate-Centric Optimization. Most existing methods maintain an optimization history H_t containing the candidate agents A generated up to round t , together with their trajectories $\mathcal{T}$, performance metrics M , and diagnostic reports D . At each round, the optimizer reinterprets this candidate-organized history to propose the next candidate:

$$A_{t+1} = \arg \max_{A \in \text{Propose}(H_t)} J_{\text{cand}}(A), \quad (1)$$

where $J_{\text{cand}}(A)$ estimates the aggregate quality of a proposed candidate using the information retained in H_t . After A_{t+1} is evaluated and diagnosed, the candidate and its associated trajectories, scores, and diagnostic report D_{t+1} are appended to H_t to form H_{t+1} . Although H_t may retain rich failure information and accumulated cross-round experience, it primarily organizes them around individual candidates. It does not maintain the current repair objective or the progress of each failure as explicit state. Consequently, the optimizer must reconstruct what to repair in every round, and its direction may shift before an ongoing repair is complete.

3.2 ISSUE-CENTRIC OPTIMIZATION

Issue-centric optimization augments the candidate-organized history H_t with a persistent issue state E_t . While H_t records the evolution of agent candidates and their associated evidence, E_t represents the evolving state of the repair process. Each issue persists across candidate generations with a stable identity, priority, lifecycle status, supporting evidence, and an intervention-outcome history. The issue-centric method uses this state to guide optimization in each round:

$$A_{t+1} = \arg \max_{A \in \text{Propose}(H_t, E_t)} J_{\text{issue}}(A; E_t), \quad (2)$$

where $J_{\text{issue}}(A; E_t)$ estimates the expected repair progress of candidate A on the unresolved issues represented in E_t . After A_{t+1} is evaluated and diagnosed, its candidate-level information is incorporated into H_t to form H_{t+1} , following the same update process described in Section 3.1. The resulting behavioral evidence is then updated with the persistent issue state:

$$E_{t+1} = \text{UpdateIssueState}(E_t, H_{t+1}). \quad (3)$$

This update associates newly diagnosed failures with persistent issue identities, records the outcomes of attempted interventions, and advances the repair states across rounds. The key distinction from candidate-centric optimization is therefore not how much historical information is retained, but whether repair progress is represented as an explicit persistent state and used to guide subsequent agent optimization.

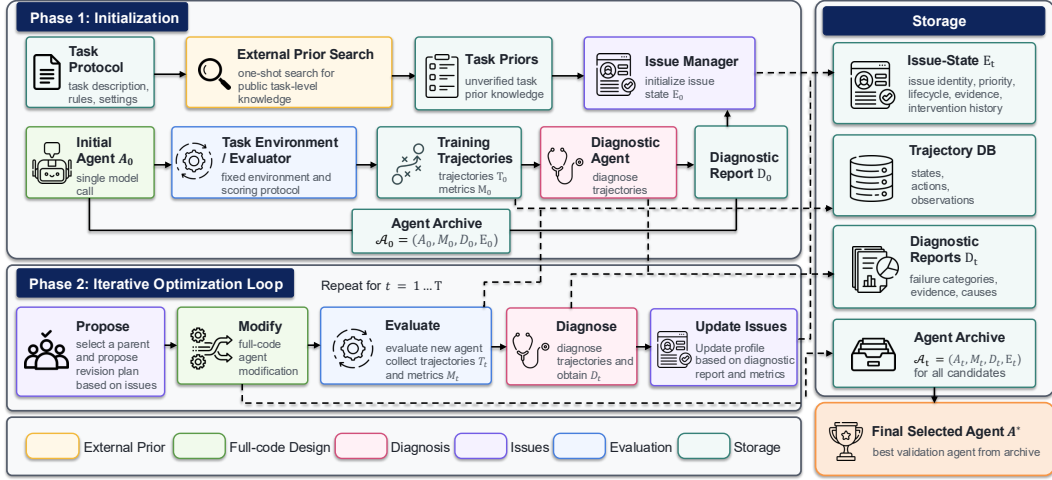


Figure 2: Overview of ADIAS that consists of two stages: initialization and iterative optimization.

3.3 ADIAS

ADIAS is built around two core mechanisms: a *persistent issue state* for organizing cross-round experience, and *issue-guided optimization* for using this state to direct agent revisions. Figure 2 provides an overview workflow of ADIAS, which consists of two stages: initialization and iterative optimization. The initialization stage constructs the initial persistent issue state from task-level priors and behavioral evidence collected from the initial agent. The optimization stage then proceeds as a five-step loop: proposing revisions based on the issue state, modifying agent code, evaluating the new agent, diagnosing trajectories, and updating the issue state.

3.3.1 PERSISTENT ISSUE STATE

The Issue Manager maintains a **persistent issue state**: $E_t = \{e_i^t\}_{i \in \mathcal{I}_{\leq t}}$, $e_i^t = (id_i, q_i^t, s_i^t, \mathcal{B}_i^t, \mathcal{U}_i^t)$, where $\mathcal{I}_{\leq t}$ is the set of issues identified up to round t . For each issue, id_i is a stable identity, q_i^t is the priority, s_i^t is the lifecycle status, $\mathcal{B}_i^t$ is the supporting evidence, and $\mathcal{U}_i^t$ is the intervention-outcome history.

Initialization. ADIAS begins with an initial agent A_0 . To reduce cold-start uncertainty, a one-shot search over publicly available task-level information produces a set of provisional priors P_0 about task requirements, likely failure modes, and potentially useful agent mechanisms. These priors provide initial hypotheses rather than verified issues. The environment then evaluates A_0 and returns interaction trajectories T_0 and performance metrics M_0 . A dedicated diagnostic agent analyzes T_0 to identify evidence-grounded failures, producing an initial diagnostic report $D_0 = (I_0, S_0, C_0)$, where I_0 denotes observed issue categories, S_0 contains supporting evidence, and C_0 describes the inferred causes. The Issue Manager reconciles the diagnostic report D_0 with the task priors P_0 to construct the initial persistent issue state E_0 . External priors are retained, revised, or discarded according to observed agent behavior. Details of prior construction and trajectory diagnosis are provided in Appendix C.1 and Appendix C.2, respectively.

At each round $t \geq 1$, the newly proposed agent A_t is evaluated and diagnosed, producing a diagnostic report D_t . The Issue Manager then updates the persistent issue state from E_{t-1} to E_t .

Issue Association. For each issue identified in D_t , the Issue Manager either associates it with an existing issue in E_{t-1} or creates a new persistent identity. Association is based on the normalized failure category, affected capability, execution context, and supporting evidence. Evidence from repeated observations is merged into the corresponding issue record. This association preserves issue identity across candidate generations, which allows repair progress to accumulate without repeatedly rediscovering the same underlying failure.

Lifecycle Transition. Each issue transitions among ACTIVE, TENTATIVELY-FIXED, CONFIRMED-FIXED, and REGRESSED. New and currently observed issues are marked as ACTIVE, while a previously fixed issue that reappears is marked as REGRESSED. An ACTIVE or REGRESSED issue becomes TENTATIVELY-FIXED when it is no longer observed, and becomes CONFIRMED-FIXED after remaining absent for at least $\alpha_{\min}$ consecutive evaluations ($\alpha_{\min} = 2$ for our experiments). These transitions distinguish temporary disappearance from reliable resolution, while making recurring failures explicit.

Intervention-Outcome History Update. The Issue Manager records the revision applied in round t and its observed outcome in the intervention-outcome history $\mathcal{U}_i^t$ of the targeted issue. The targeted intervention outcome is stored separately from lifecycle changes observed for other issues, which makes progress on the intended repair distinguishable from concurrent improvements or regressions elsewhere. Supported intervention outcomes are further distilled into reusable repair lessons. This issue-specific history supports more reliable outcome attribution and helps subsequent rounds reuse effective interventions while avoiding previously unsuccessful directions.

3.3.2 ISSUE-GUIDED OPTIMIZATION

Issue-Guided Planning. At round t , the Issue Manager uses the current state E_{t-1} to select a small set of target issues and jointly determine a corresponding parent agent A_{p_t} and revision plan R_t . Prioritization favors issues that are severe, repeatedly observed, currently active, or recently regressed. The parent and revision direction are selected according to the targets’ supporting evidence, lifecycle status, and previous intervention outcomes, and are then passed to a code optimizer. This joint planning couples what to repair, where to resume optimization, and how to revise, which prevents these inter-related decisions from being made independently.

Focused Full-Code Revision. Conditioned on A_{p_t} and R_t , the optimizer generates a code patch δ_t and applies it to the parent agent to produce A_t . The optimizer is encouraged to implement a focused revision over a small number of files, while the editable space remains full-code. It may modify any agent-side component, including prompts, observation processing, memory, planning, tool-use policies, control flow, verification, and recovery mechanisms. This focused revision makes intervention attribution easier without restricting the design space available for optimization.

The resulting agent A_t is evaluated to obtain interaction trajectories $\mathcal{T}_t$ and performance metrics M_t , which are subsequently analyzed to produce the diagnostic report D_t . The Issue Manager then uses D_t to update the persistent issue state from E_{t-1} to E_t . This feedback loop converts agent revision into issue-level evidence that informs subsequent optimization rounds.

4 EXPERIMENTS

Benchmarks. We evaluate ADIAS on five interactive agent benchmarks spanning different scenarios, including Tau-Bench (τ -Bench) for tool-use in customer service (Yao et al., 2025), ALF-World for embodied AI planning (Shridhar et al., 2021), TextCraft for compositional crafting games (Prasad et al., 2024), WebShop for web navigation (Yao et al., 2022), and ScienceWorld for scientific experimentation (Wang et al., 2022). These environments require agents to make a sequence of state-dependent decisions rather than produce a single static response. To control computational costs, we construct representative task subsets for the main experiments. We report benchmark details in Appendix D.1.

Baselines. We compare ADIAS with five representative baselines on different editable scopes and experience organization. We use a manually designed ReAct-style agent (Yao et al., 2023) with task-level memory as the fixed, non-automated baseline. SkillOpt (Yang et al., 2026) performs prompt-level agent optimization by maintaining a skill document. Agentic Harness Engineering (AHE) (Lin et al., 2026a) performs architecture-level optimization over pre-defined components of agent harnesses. Meta-Harness (Lee et al., 2026) performs end-to-end optimization over the entire agent harness with filesystem access. DGM-Hyperagents (DGM-H) (Zhang et al., 2026a) integrates the task agent and meta agent into a single editable program and allows the resulting hyperagent to modify both its task-solving logic and the mechanism used to generate future improvements. All methods use the same benchmark wrappers, task splits, action interfaces, and scoring scripts. Detailed descriptions are provided in Appendix D.2.

Table 1: Task performance and efficiency across five interactive benchmarks using DeepSeek-V4-Flash. Best and second-best results are highlighted in green and blue, respectively.

Method	Tau-Bench		ALFWorld		TextCraft		WebShop		ScienceWorld		Avg. Score ↑
	Score ↑	Eff. ↑	Score ↑	Eff. ↑	Score ↑	Eff. ↑	Score ↑	Eff. ↑	Score ↑	Eff. ↑	
Handcrafted	75.0	5.73	70.9	3.60	38.0	1.52	39.1	2.13	38.1	3.05	52.2
SkillOpt	12.5	0.45	0.0	0.00	37.0	1.70	22.3	1.17	30.1	1.89	20.4
Meta-Harness	43.8	2.39	59.7	3.32	24.0	0.95	46.0	3.71	26.5	1.65	40.0
AHE	56.2	3.05	78.4	4.48	35.0	1.39	57.3	5.46	42.6	3.44	53.9
DGM-H	59.4	3.93	69.4	3.42	76.0	5.98	57.5	5.42	50.9	3.98	62.6
ADIAS (Ours)	81.3	6.35	94.0	8.95	91.0	9.01	69.4	5.55	56.3	3.99	78.4

Models. We use DeepSeek-V4-Flash (DeepSeek, 2026) as the default backbone model. To evaluate cross-model robustness, we additionally conduct experiments with GLM-5.2 (Z.ai, 2026), Hy3-Preview (Tencent Hy, 2026), and GPT-5.4 (OpenAI, 2026). For each experimental setting, the same backbone model is used for both the design agent and the task agent. We apply this matched-model protocol to all automated baselines and ADIAS, which ensures that performance differences are attributable to the agent-design method rather than to a stronger optimizer or controller model. For Tau-Bench, the user simulator is held fixed to DeepSeek-V4-Flash across all settings.

Evaluation Protocol. For each method, we select the agent with the highest validation performance and evaluate it on the held-out test set. All methods use the same task splits, environment interfaces, optimization budgets, and benchmark-specific evaluators. We report the native metric of each benchmark (%): success rate for Tau-Bench, ALFWorld and TextCraft, and the normalized native task score for WebShop and ScienceWorld. No test tasks or validation trajectory feedback are exposed during optimization. In addition to task performance, we report interaction efficiency, which measures task performance per environment interaction. Since the semantics of an interaction step differ across environments, efficiency is compared only within the same benchmark. Full metric definitions and evaluation details are provided in Appendix D.3.

Experimental Settings. We run each automated method for a fixed budget of 10 optimization iterations. At each iteration, 15 training episodes are sampled for agent optimization. The same iteration and rollout budgets are applied to all automated methods within each benchmark. Each task-agent episode is limited to 30 environment interaction steps or dialogue turns. We also keep decoding and inference configurations fixed across methods using the same backbone model.

5 RESULTS

5.1 MAIN RESULTS

Strong Task Performance and Efficiency. Table 1 compares ADIAS against fixed handcrafted strategies and recent automated design baselines across five interactive benchmarks. ADIAS achieves the best task performance and interaction efficiency on all five benchmarks, which demonstrates robust generalization across environments with diverse interaction patterns and task objectives. Averaged across benchmarks, ADIAS reaches a score of 78.4 compared with 62.6 for the strongest baseline (DGM-H), a relative improvement of 25.2%. Specifically, scores of ADIAS on Tau-Bench, ALFWorld, TextCraft, WebShop, and ScienceWorld are 81.3, 94.0, 91.0, 69.4, and 56.3, respectively, which consistently surpass all baselines. These gains are not obtained through longer interactions. ADIAS also achieves the highest interaction efficiency on every benchmark, which indicates that its learned strategies translate environment interactions into task progress more effectively. The advantage is particularly pronounced on ALFWorld and TextCraft, where ADIAS reaches efficiencies of 8.95 and 9.01, and substantially exceeds the strongest baseline results of 4.48 (AHE) and 5.98 (DGM-H), respectively.

High Optimization Efficiency and Stability. Figure 3 shows the optimization process on ALF-World. ADIAS identifies high-performing agent designs within the first few iterations and maintains consistently strong validation performance throughout the remaining search. Its cumulative-average score also increases steadily, which indicates that the improvement is not driven by a small number of isolated high-scoring candidates; rather, the optimizer can generate effective designs over time.

Table 2: Cross-model task performance and efficiency on Tau-Bench. Best and second-best results are highlighted in green and blue, respectively.

Method	DeepSeek-V4-Flash		GLM-5.2		Hy3-Preview		GPT-5.4		Avg. Score ↑
	Score ↑	Eff. ↑	Score ↑	Eff. ↑	Score ↑	Eff. ↑	Score ↑	Eff. ↑	
Handcrafted	75.0	5.73	84.4	6.49	15.6	0.86	71.9	5.89	61.73
SkillOpt	12.5	0.45	12.5	0.67	12.5	0.59	15.6	0.74	13.28
Meta-Harness	43.8	2.39	78.1	6.51	68.8	5.38	65.6	5.56	64.08
AHE	56.2	3.05	62.5	4.08	78.1	6.20	75.0	5.81	67.95
DGM-H	59.4	3.93	65.6	4.10	75.0	5.86	71.9	5.33	67.98
ADIAS (Ours)	81.3	6.35	90.6	7.13	84.4	6.54	87.5	6.63	85.95

It is worth noting that SkillOpt modifies only a persistent textual skill document, so its search space cannot introduce additional agent components, such as explicit memory mechanisms, which are essential for effective interaction in ALFWorld.

Cross-Model Robustness. We conduct a cross-model evaluation on Tau-Bench using four heterogeneous language models: DeepSeek-V4-Flash, GLM-5.2, Hy3-Preview, and GPT-5.4. As shown in Table 2, ADIAS consistently achieves the highest task performance across all four backbone models, with scores of 81.3, 90.6, 84.4, and 87.5, respectively. The corresponding interaction efficiencies are also consistently the highest among all compared methods. These results demonstrate that ADIAS provides robust automated design across different backbone models.

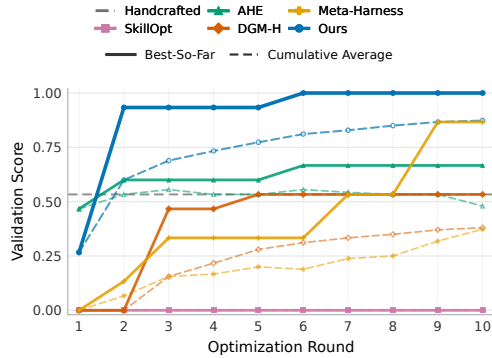


Figure 3: Optimization process on ALFWorld.

5.2 ABLATION STUDY

As shown in Table 3, we separate three roles of the persistent issue state: (1) the evidence used to construct and update it, (2) the representation used to maintain cross-round issues, and (3) whether that state actively controls the optimization process. We show that ADIAS benefits from three coupled properties: informative evidence, a durable issue-level representation, and the use of that representation as an optimization control state.

External priors and round-level diagnosis provide necessary evidence. Removing either source reduces both performance and interaction efficiency across all three benchmarks. Without the external prior, the largest drop occurs on TextCraft, from 91.0 to 61.0. Without round-level diagnosis, ALFWorld decreases from 94.0 to 63.4. These results indicate that the two mechanisms serve important roles: the external prior reduces cold-start uncertainty, while trajectory diagnosis provides the behavioral evidence required to ground and update the issue state.

A raw candidate archive cannot replace persistent issue state. *Archive-Wide Synthesis* removes the durable issue state and follows a Meta-Harness-style mechanism (Lee et al., 2026). The designer receives a raw archive containing candidate agents, evaluation results, trajectory histories, and diagnostic reports, and synthesizes the next optimization step directly from this archive. Its scores decrease to 65.6, 60.4, and 32.0 on Tau-Bench, ALFWorld, and TextCraft, respectively. Although the archive preserves rich historical evidence, identifying an effective revision direction remains difficult in interactive environments with long trajectories and sparse feedback, where useful signals must be recovered from heterogeneous candidate-level records.

Maintaining issue state is insufficient unless it controls optimization. *Best-Candidate Revision* and *Latest-Candidate Continuation* are two of the most commonly used methods for parent selection in the agent optimization process. They retain the same candidate history and persistent issue profiles as ADIAS, but the issue state is treated only as contextual information: the actual parent is selected by validation score or generation order, rather than by the issue-guided revision plan. *Best-Candidate Revision* is the strongest ablation, but remains below ADIAS by 9.4, 18.6, and 17.0 score

Table 3: Ablation study on representative benchmarks using DeepSeek-V4-Flash. Average Score is the unweighted average across the three benchmarks, and Δ denotes its change relative to full ADIAS. Best and second-best results are highlighted in green and blue, respectively.

Setting	Tau-Bench		ALFWorld		TextCraft		Avg. Score $\uparrow$	Δ
	Score $\uparrow$	Eff. $\uparrow$	Score $\uparrow$	Eff. $\uparrow$	Score $\uparrow$	Eff. $\uparrow$		
ADIAS (Full)	81.3	6.35	94.0	8.95	91.0	9.01	88.8	–
<i>Ablating evidence for issue-state construction</i>								
w/o External Prior	75.0	5.64	82.8	5.27	61.0	3.10	72.9	-15.9 (17.9%)
w/o Round-Level Diagnosis	78.1	5.66	63.4	3.41	76.0	5.80	72.5	-16.3 (18.4%)
<i>Replacing persistent issue state with a raw archive</i>								
w/ Archive-Wide Synthesis	65.6	4.43	60.4	3.25	32.0	1.27	52.7	-36.1 (40.7%)
<i>Decoupling issue state from optimization control</i>								
w/ Best-Candidate Revision	71.9	5.10	75.4	4.54	74.0	5.03	73.8	-15.0 (16.9%)
w/ Latest-Candidate Continuation	62.5	4.37	41.0	1.81	60.0	3.53	54.5	-34.3 (38.6%)

points on the three benchmarks. Because each round returns to the aggregate-best candidate, locally useful interventions discovered in different branches remain isolated and are difficult to consolidate into a single improved lineage. *Latest-Candidate Continuation* exhibits the opposite limitation. Because every round continues from the most recent candidate, the next revision is more likely to inherit ineffective or regressive modifications. The parents selected by these two methods are not necessarily the most suitable starting point for the current repair. The persistent issue state must therefore serve as an operational control signal, so that optimization is conditioned on the active issue and the outcomes of previous interventions.

5.3 QUALITATIVE ANALYSIS

We inspect optimization traces to understand how issue state shapes the design process. The behaviors described below recur across all three tested benchmarks in Table 3. We use representative examples to illustrate and support our findings. Figure 4 shows the optimization process on the TextCraft benchmark.

A raw candidate archive provides a weaker control signal than persistent issue state in long-horizon, sparse-feedback environments. We observe a similar anchoring mechanism as shown in Lee et al. (2026): optimization typically returns to a high-scoring candidate or continues from the latest accepted one, and then attempts to summarize a new improvement direction from the surrounding archive. This strategy is less reliable when useful evidence is distributed across long interaction trajectories and sparse outcome signals. For example, in TextCraft (Figure 4), later rounds repeatedly return to previous high-score generations 2 and 6 and explore prompt restructuring, admissible-command matching, and command-selection heuristics, but the underlying quantity missing error is never localized (get quartz vs. get 1 quartz). The optimizer must reconstruct problems from candidate-level records in every round, which makes effective revision directions difficult to identify. Persistent issue state instead maintains an explicit issue–intervention–outcome history, thereby improving revision effectiveness.

Best-Candidate Revision makes partial repairs slower to consolidate across branches. When a revision partially addresses an issue but lowers the aggregate score, its code-level changes are not inherited by the next candidate. The designer has to recover the useful insight from history and attempt a related repair again on a higher-scoring parent. In TextCraft, early branches introduce repairs of command recovery and crafting guidance, but their validation scores remain below generation 2, causing subsequent rounds to repeatedly return to that candidate and abandon parts of the previous implementations. Only after several alternative revisions, generation 7 combines effective revisions and improves validation from 50.0 to 66.7. The same pattern is more pronounced in ALFWorld, where generations 4-10 repeatedly branch from generation 3 while separately targeting object identity, wrong-object recovery, action validation, and planner behavior. These interventions remain available in the history and can eventually be reused, but their useful components must be repeatedly attributed and reimplemented across branches before they improve the aggregate score. Thus, consolidation remains possible, but it is slower and less direct than maintaining issue-specific intervention outcomes.

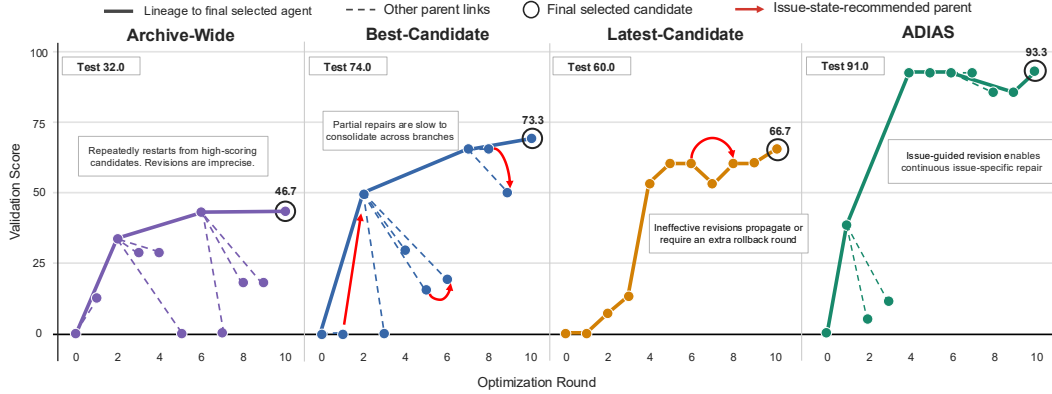


Figure 4: Comparison of optimization processes on the TextCraft benchmark.

Latest-Candidate Continuation makes ineffective revisions more likely to propagate, either accumulating errors or consuming additional rounds to recover. For example, in TextCraft (Figure 4), generation 7 introduces fallback exploration and reduces validation success from 60.0 to 53.3. Although the issue state recommends returning to generation 6, the fixed policy makes generation 8 inherit generation 7. The next round removes the ineffective modification merely to recover the previous validation score, which spends one additional round on rollback rather than further improvement. In contrast, ADIAS would return to generation 6 and resume optimization from a more suitable parent. The same problem is more pronounced in ALFWorld. After generation 8 is identified as unsuccessful, the issue state recommends generation 4 as the next parent, yet generation 9 is still derived from generation 8. In this case, the ineffective revision remains instead of being bypassed, which makes subsequent candidates more likely to inherit its errors.

ADIAS enables continuous issue-specific repair by selecting a more appropriate starting point and revision direction in each round. In TextCraft, ADIAS identifies the recurring command failure and repairs it by normalization. The intervention and its outcome are then written back to the persistent issue state, which allows subsequent rounds to retain the successful repair and continue addressing the remaining grounding and recovery issues. When modifications at generation 9 fail to improve task success rate but contribute to other issues, such as interaction efficiency, ADIAS still treats round 9 as a more suitable parent to accumulate partial revision. A similar process appears in ALFWorld, where optimization progresses from broad planning failures to more specific issues in action correction, planning heuristics, and target-object extraction. Across all benchmarks, ADIAS uses the active issue, previous interventions, and their observed outcomes to determine both where to restart and what to modify, which supports a continuous repair process.

6 CONCLUSION

We introduced issue-centric agent optimization, which reframes automated agent design as a cumulative repair process rather than repeated candidate generation. We instantiated this formulation in ADIAS, a framework for automated full-code agent optimization that combines persistent issue state maintenance and issue-guided optimization. Across five interactive benchmarks, ADIAS outperforms the strongest baseline by 25.2% on average and generalizes consistently across four backbone models. Controlled ablations and qualitative analyses further show that both persistent issue-level organization and its role as an optimization control state are necessary. These results suggest that future agent optimization systems should treat repair progress as an explicit, persistent, and operational state.

Limitations. ADIAS relies on accurate trajectory diagnosis and issue association. Our focus is issue-centric optimization rather than diagnosis quality itself, so we do not separately report the accuracy of failure labels or issue attributions; the diagnostic process is kept fixed in corresponding comparisons. Our evaluation is also limited to text-based interactive benchmarks and leaves multimodal and substantially longer-horizon tasks for future work.

REFERENCES

- Lakshya A Agrawal, Shangyin Tan, Dilara Soyulu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, 2026.
- Qianshu Cai, Yonggang Zhang, Xianzhang Jia, Huajiang Zheng, Wei Xue, Jun Song, Xinmei Tian, and Yike Guo. Moss: Self-evolution through source-level rewriting in autonomous agent systems. *arXiv preprint arXiv:2605.22794*, 2026.
- Tingyang Chen, Shuo Lu, Kang Zhao, Weicheng Meng, Hanlin Teng, Tianhao Li, Chao Li, Xule Liu, Jian Liang, Zhizhong Zhang, et al. Harnessx: A composable, adaptive, and evolvable agent harness foundry. *arXiv preprint arXiv:2606.14249*, 2026.
- DeepSeek. DeepSeek V4 Preview Release. <https://api-docs.deepseek.com/news/news260424>, 2026. Accessed: 2026-07.
- Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Forty-first International Conference on Machine Learning*, 2024.
- Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Qihan Ren, et al. A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026.
- Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 34133–34156, 2024.
- Amine El Hattami, Nicolas Chapados, and Christopher Pal. Skill. nb: Selective formalization and gated execution for durable agent workflows. *arXiv preprint arXiv:2606.08049*, 2026.
- Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations*, volume 2025, pp. 21344–21377, 2025.
- Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*, 2024.
- Yue Huang, Wenjie Wang, Han Bao, Yuchen Ma, Xiaonan Luo, Yi Nian, Haomin Zhuang, Zheyuan Liu, Yue Zhao, and Xiangliang Zhang. Memoharness: Agent harnesses that learn from experience. *arXiv preprint arXiv:2607.14159*, 2026.
- Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, et al. Dspy: compiling declarative language model calls into state-of-the-art pipelines. In *International Conference on Learning Representations*, volume 2024, pp. 54928–54958, 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.
- Yu Li, Lehui Li, Zhihao Wu, Qingmin Liao, Jianye Hao, Kun Shao, and Fengli Xu. Agentswift: Efficient llm agent design via value-guided hierarchical search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 31843–31851, 2026.
- Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821*, 2024.
- Jiahang Lin, Shichun Liu, Chengjun Pan, Lizhi Lin, Shiha Dou, Zhiheng Xi, Xuanjing Huang, Hang Yan, Zhenhua Han, Tao Gui, et al. Agentic harness engineering: Observability-driven automatic evolution of coding-agent harnesses. *arXiv preprint arXiv:2604.25850*, 2026a.

- Minhua Lin, Hanqing Lu, Zhan Shi, Bing He, Rui Mao, Zhiwei Zhang, Zongyu Wu, Xianfeng Tang, Hui Liu, Zhenwei Dai, et al. Position: Agentic evolution is the path to evolving llms. *arXiv preprint arXiv:2602.00359*, 2026b.
- Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.
- Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P Murphy. Autoharness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.
- Maksim Madžar and Igor Mekterović. Automated design of agentic systems: A survey of algorithms for searching, optimizing, and evolving llm agents, workflows, and prompts. *Preprints*, June 2026. doi: 10.20944/preprints202606.0238.v1. URL <https://doi.org/10.20944/preprints202606.0238.v1>.
- Johannes Moll, Jean-Philippe Corbeil, Jiazhen Pan, Martin Hadamitzky, Daniel Rueckert, Lisa Adams, and Keno Bressem. Grasp: Gated regression-aware skill proposer for self-improving llm agents. *arXiv preprint arXiv:2605.29668*, 2026.
- Jun Nie, Yonggang Zhang, Jun Song, Qianshu Cai, Dahai Yu, Yike Guo, Xinmei Tian, and Bo Han. Tthe: Test-time harness evolution. *arXiv preprint arXiv:2607.08124*, 2026.
- Xuying Ning, Katherine Tieu, Dongqi Fu, Tianxin Wei, Zihao Li, Yuanchen Bei, Jiaru Zou, Mengting Ai, Zhining Liu, Ting-Wei Li, et al. Code as agent harness. *arXiv preprint arXiv:2605.18747*, 2026.
- OpenAI. GPT-5.4 Model. <https://developers.openai.com/api/docs/models/gpt-5.4>, 2026. Accessed: 2026-07.
- Archiki Prasad, Alexander Koller, Mareike Hartmann, Peter Clark, Ashish Sabharwal, Mohit Bansal, and Tushar Khot. Adapt: As-needed decomposition and planning with language models. In *Findings of the Association for Computational Linguistics: NAACL 2024*, pp. 4226–4252, 2024.
- Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with “gradient descent” and beam search. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pp. 7957–7968, 2023.
- Jon Saad-Falcon, Adrian Gamarra Lafuente, Shlok Natarajan, Nahum Maru, Hristo Todorov, Etash Guha, E Kelly Buchanan, Mayee Chen, Neel Guha, Christopher Ré, et al. Archon: An architecture search framework for inference-time techniques. *arXiv preprint arXiv:2409.15254*, 2024.
- Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. Agentsquare: Automatic llm agent search in modular design space. In *International Conference on Learning Representations*, volume 2025, pp. 3841–3865, 2025.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.
- Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Cote, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. In *International Conference on Learning Representations*, 2021.
- Tencent Hy. Introducing Hy3. <https://hy.tencent.com/research/hy3>, 2026. Accessed: 2026-07.
- Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a.

- Ruhan Wang, Yucheng Shi, Zongxia Li, Zhongzhi Li, Yue Yu, Junyao Yang, Kishan Panaganti, Haitao Mi, Dongruo Zhou, et al. Harness handbook: Making evolving agent harnesses readable, navigable, and editable. *arXiv preprint arXiv:2607.13285*, 2026a.
- Ruoyao Wang, Peter Jansen, Marc-Alexandre Côté, and Prithviraj Ammanabrolu. Scienceworld: Is your agent smarter than a 5th grader? In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 11279–11298, 2022.
- Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-g\ "odel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. *arXiv preprint arXiv:2510.21614*, 2025.
- Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. In *International Conference on Learning Representations*, volume 2024, pp. 23967–24001, 2024b.
- 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*, 2026b.
- Lilian Weng. Harness engineering for self-improvement. *lilianweng.github.io*, July 2026. URL <https://lilianweng.github.io/posts/2026-07-04-harness/>.
- Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, volume 2024, pp. 12028–12068, 2024.
- Yifan Yang, Ziyang Gong, Weiquan Huang, Qihao Yang, Ziwei Zhou, Zisu Huang, Yan Li, Xuemei Gao, Qi Dai, Bei Liu, et al. Skillopt: Executive strategy for self-evolving agent skills. *arXiv preprint arXiv:2605.23904*, 2026.
- Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.
- 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.
- Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R Narasimhan. τ -bench: A benchmark for tool-agent-user interaction in real-world domains. In *The Thirteenth International Conference on Learning Representations*, 2025.
- Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27890–27913, 2025.
- Ye Yu, Xiaopeng Yuan, Haibo Jin, Heming Liu, Yaoning Yu, and Haohan Wang. Do self-evolving agents forget? capability degradation and preservation in lifelong llm agent adaptation. *arXiv preprint arXiv:2605.09315*, 2026.
- Ling Yue, Kushal Raj Bhandari, Ching-Yun Ko, Dhaval Patel, Shuxin Lin, Nianjun Zhou, Jianxi Gao, Pin-Yu Chen, and Shaowu Pan. From static templates to dynamic runtime graphs: a survey of workflow optimization for llm agents. *arXiv preprint arXiv:2603.22386*, 2026.
- Z.ai. GLM-5.2 Model. <https://docs.bigmodel.cn/cn/guide/models/text/glm-5.2>, 2026. Accessed: 2026-07.
- Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *International Conference on Machine Learning*, pp. 75834–75852. PMLR, 2025a.

- Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. *SuperIntelligence-Robotics-Safety & Alignment*, 2(3), 2025b.
- Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026a.
- Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025c.
- Jiayi Zhang, Yongfeng Gu, Jianhao Ruan, Maojia Song, Yiran Peng, Zhiguang Han, Jinyu Xiang, Zhitao Wang, Caiyin Yang, Yixi Ouyang, et al. Harnessing agentic evolution. *arXiv preprint arXiv:2605.13821*, 2026b.
- Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamannuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, 2026c.
- Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.
- Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *The eleventh international conference on learning representations*, 2022.
- Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Gptswarm: Language agents as optimizable graphs. In *Forty-first International Conference on Machine Learning*, 2024.

A INTERACTIVE AGENT

Interactive environments make automated agent design substantially more difficult in three ways.

First, they introduce a larger and more highly task-dependent design space compared to the single-input-output settings. Successful agents may require different combinations of observation processing, memory, planning, tool use, verification, and recovery mechanisms (Huang et al., 2024; Wang et al., 2024a). Full-code design provides the necessary expressiveness, but must search over a much larger and less structured space of executable implementations (Hu et al., 2025).

Second, interaction feedback is sparse, delayed, and difficult to attribute. Interactive agents may make many dependent decisions over multiple environment states before a final success or failure signal is observed (Liu et al., 2024; Yao et al., 2025). A scalar score reveals *whether* an agent succeeded, but rarely explains *why*: the failure may originate from an early misunderstanding, lost state, incorrect tool use, a policy violation, or ineffective recovery. Simply providing complete trajectories to an LLM is also problematic, as long and heavy traces contain substantial irrelevant context and make decisive failure evidence difficult to isolate. Effective design search therefore requires precise, evidence-grounded failure attribution.

Third, interactive evaluation is expensive, which makes search efficiency critical. Each candidate must be assessed through environment rollouts that may require extensive model calls and interaction steps (Yao et al., 2022; 2025). Existing full-code design and self-improvement methods commonly retain and select evaluated agent variants based primarily on empirical task scores (Hu et al., 2025; Zhang et al., 2025b; 2026a). Without a persistent view of search progress, however, the designer may revisit unsuccessful modifications, overlook the recurrence of previously mitigated failures, and repeatedly spend evaluation budget on already explored directions. Efficient optimization therefore requires tracking what has been tried, what has improved or regressed, and which unresolved bottleneck should guide the next design round.

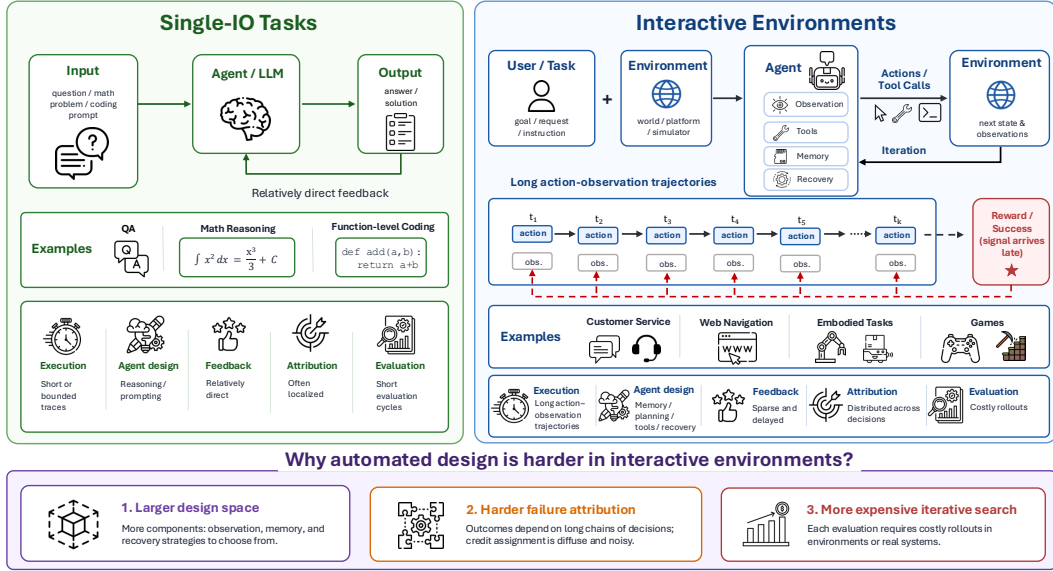


Figure 5: Task comparison between single-IO tasks and long-horizon interactive environments.

B DETAILED RELATED WORK

Recent work has increasingly shifted from manually designing agentic systems toward automatically optimizing them from execution feedback. Automated Design of Agentic Systems (ADAS) formalizes this process as searching over agentic system designs under an evaluation objective (Madžar & Mekterović, 2026). Related surveys study this broader direction through workflow optimization, self-evolving agents, and adaptive agent harnesses (Gao et al., 2026; Yue et al., 2026; Ning et al., 2026). We organize prior work into three levels according to the expressiveness of the optimization target: prompt (text)-level, architecture (harness)-level, and full-code-level design.

Prompt-level Design. Prompt-level methods optimize textual artifacts based on fixed models and an agent execution process. Early automatic prompt optimization methods search directly over natural-language instructions, using LLM-generated candidates (Zhou et al., 2022), scored optimization histories (Yang et al., 2024), textual gradients (Pryzant et al., 2023), evolutionary search (Guo et al., 2024; Fernando et al., 2024), or Monte Carlo tree search (Wang et al., 2024b). More recent methods exploit richer execution feedback: GEPA evolves prompts through trajectory reflection and Pareto-based selection, while DSPy optimizes instructions within modular language-model pipelines (Agrawal et al., 2026; Khattab et al., 2024). ACE and SkillOpt further treat persistent context playbooks or skill documents as editable textual state, accumulating reusable procedures, heuristics, and failure-handling knowledge from experience (Zhang et al., 2026c; Yang et al., 2026).

Architecture-level Design. Architecture-level methods design the agent’s overall structure, while remaining constrained by a predefined graph, module set, or harness abstraction. Existing approaches broadly follow two directions. The first direction directly searches over agent architectures or workflows, including AutoFlow, AFlow, and GPTSwarm (Li et al., 2024; Zhang et al., 2025c; Zhuge et al., 2024). Other methods compose predefined reasoning, planning, tool-use, memory, and inference-time components, as in AgentSquare, Archon, MaAS, and AgentSwift (Shang et al., 2025; Saad-Falcon et al., 2024; Zhang et al., 2025a; Li et al., 2026). More recent systems expose the runtime harness itself as structured, editable artifacts: A-Evolve maintains versioned knowledge, workflow, tool, and validation assets; AHE edits explicit harness components using trajectory observability; and HarnessX evolves compositions of typed harness primitives (Lin et al., 2026b;a; Chen et al., 2026).

Full-code-level Design. At the most expressive level, agents are represented as executable programs and optimized directly in code space. Hu et al. (2025) introduced this direction by using a fixed meta-agent to iteratively propose new task agent implementations and retain successful variants in an archive. DGM combines self-modification with open-ended evolutionary exploration

over a branching archive (Zhang et al., 2025b), while HGM further improves parent selection using estimated descendant potential (Wang et al., 2025). In addition, Meta-Harness searches harness code using selective access to previous code, scores, and traces (Lee et al., 2026); AutoHarness synthesizes executable harnesses from environment feedback (Lou et al., 2026); MOSS performs source-level rewriting from curated failures (Cai et al., 2026); and TTHe evolves harnesses at test time from unlabeled trajectories (Nie et al., 2026). A further line of work makes parts of the improvement mechanism itself editable. Gödel Agent supports self-referential modification of task behavior and improvement logic; DGM-H jointly evolves the task agent and meta-level modification mechanism; and AEvo revises the procedure or context that supports subsequent evolution (Yin et al., 2025; Zhang et al., 2026a;b). These methods extend automated design from optimizing an agent artifact to optimizing how future agent modifications are designed.

C ADIAS DETAILS

Algorithm 1 summarizes the complete procedure.

ADIAS starts from an initial agent A_0 . The initial task agent repository exposes lightweight placeholders for common agent functions, including observation processing, memory, planning, tool policy, verification, and recovery. These placeholders do not define the optimization space: the meta agent may arbitrarily rewrite, remove, merge, or replace them, introduce new modules and control flow, and modify any agent-side executable code. The external searcher is executed once to collect public task-level prior information P based on task protocol $\mathcal{P}$.

$$P = \text{ExternalSearch}(\mathcal{P}). \quad (4)$$

Before any agent modification, ADIAS evaluates A_0 , diagnoses the resulting training trajectories, and incorporates the observed evidence into the first issue state E_0 ,

$$\mathcal{T}_0, M_0 = \text{Evaluate}(A_0), \quad (5)$$

$$D_0 = \text{Diagnose}(\mathcal{P}, \mathcal{T}_0), \quad (6)$$

$$E_0 = \text{InitializeIssueState}(P, D_0, M_0), \quad (7)$$

where $\mathcal{T}_t$ is the set of training trajectories, M_t contains training and aggregate validation metrics, and D_t is the diagnostic report.

Each subsequent round follows the same five-stage cycle: *propose, modify, evaluate, diagnose, and update the issue state*,

$$A_p, R_t \leftarrow \text{Propose}(E_{t-1}), \quad (8)$$

$$A_t \leftarrow \text{Modify}(A_p, R_t), \quad (9)$$

$$\mathcal{T}_t, M_t \leftarrow \text{Evaluate}(A_t), \quad (10)$$

$$D_t \leftarrow \text{Diagnose}(\mathcal{P}, \mathcal{T}_t), \quad (11)$$

$$E_t \leftarrow \text{UpdateIssueState}(E_{t-1}, D_t, M_t), \quad (12)$$

The evaluated candidate and its associated metrics, diagnosis, profile state, and code change are then stored in the agent archive $\mathcal{A}$,

$$\mathcal{A}_t \leftarrow \mathcal{A}_{t-1} \cup \{(A_t, M_t, D_t, E_t)\}. \quad (13)$$

After the design budget is exhausted, the candidate with the highest validation score is selected for held-out test evaluation.

C.1 EXTERNAL PRIOR SEARCH

Full-code agent design exposes an extremely large space of possible implementations. However, the task protocol and public task-level knowledge often provide useful priors about likely interaction requirements and failure modes. ADIAS uses a one-shot external searcher to exploit this information before iterative design begins.

Given the task protocol $\mathcal{P}$, the external searcher retrieves public information such as benchmark descriptions, rule explanations, common failure modes, tool-use strategies, and candidate agent mechanisms. The resulting external prior P is incorporated into the initial issue state as *unverified task*

Algorithm 1 Automated Design of Interactive Agentic Systems**Require:** Task protocol $\mathcal{P}$, initial agent A_0 , optimization round T

```

1:  $P \leftarrow \text{ExternalSearch}(\mathcal{P})$ 
2:  $\mathcal{T}_0, M_0 \leftarrow \text{Evaluate}(A_0)$ 
3:  $D_0 \leftarrow \text{Diagnose}(\mathcal{P}, \mathcal{T}_0)$ 
4:  $E_0 \leftarrow \text{InitializeIssueState}(P, D_0, M_0)$ 
5:  $\mathcal{A}_0 \leftarrow \{(A_0, M_0, D_0, E_0)\}$ 
6: for  $t = 1, \dots, T$  do
7:    $\mathcal{I}_t \leftarrow \text{Prioritize}(E_{t-1})$ 
8:    $A_p, R_t \leftarrow \text{ProposeRepair}(E_{t-1}, \mathcal{I}_t)$ 
9:    $A_t \leftarrow \text{Modify}(A_p, R_t)$ 
10:   $\mathcal{T}_t, M_t \leftarrow \text{Evaluate}(A_t)$ 
11:   $D_t \leftarrow \text{Diagnose}(\mathcal{P}, \mathcal{T}_t)$ 
12:   $E_t \leftarrow \text{UpdateIssueState}(E_{t-1}, D_t, M_t)$ 
13:   $\mathcal{A}_t \leftarrow \mathcal{A}_{t-1} \cup \{(A_t, M_t, D_t, E_t)\}$ 
14: end for
15: return  $A^* = \arg \max_{A_i \in \mathcal{A}_T} V_i$ 

```

prior knowledge. It is explicitly separated from empirical experience and is treated only as a set of design hypotheses. This distinction is important because retrieved information may be incomplete or unsuitable for the current agent. A prior is promoted into accumulated task experience only when supported by trajectory diagnosis, patch outcomes, runtime evidence, or score trends. The external searcher is executed exactly once and is not changed during late design iterations. We restrict retrieval to public task-level information and exclude held-out labels, instance-specific solutions, hidden evaluation data, and hard-coded benchmark outputs. Overall, external search narrows the effective cold-start search space without restricting the expressiveness of the underlying full-code design space.

C.2 EVIDENCE-GROUNDED FAILURE DIAGNOSIS

A scalar evaluation score reveals whether an agent succeeds, but rarely identifies which design deficiency caused a failure. This attribution problem is difficult in interactive tasks because a single episode may contain many state transitions, tool calls, and dependent decisions. Directly placing all trajectories into the context of a designer model also introduces substantial irrelevant information.

ADIAS therefore introduces a dedicated diagnostic agent that analyzes all training trajectories collected in one optimization round. Rather than consuming complete trajectories as a flat prompt, the diagnostic agent interacts with a structured trajectory database through targeted query tools. It can inspect aggregate failure patterns, compare successful and failed episodes, retrieve compact action sequences or local step windows, search for repeated and invalid actions, and examine tool errors, visible final states, etc. The diagnostic agent uses these queries to iteratively examine failures based on trajectory evidence. The output is a diagnostic report

$$D_t = (I_t, S_t, C_t), \quad (14)$$

where I_t denotes observed issue categories, S_t contains supporting evidence, and C_t describes the inferred causes. The objective is not merely to locate a failed action, but to identify recurring behavioral patterns that indicate deficiencies in the current agent design. Diagnosis is deliberately separated from modification, which means the diagnostic agent does not propose suggestions or implement code patches.

C.3 PERSISTENT ISSUE CONTROL

This section describes how ADIAS operationalizes that state for full-code agent design. In our implementation, E_t is maintained as a structured profile containing a global optimization summary and a set of issue records. Each issue record instantiates the fields in Section 3.3.1 with a concise issue description, priority, lifecycle status, evidence references, previously attempted interventions, observed outcomes, and the candidate generations in which the issue was observed or absent. The

Table 4: Summary of the five interactive benchmarks used in our experiments.

Benchmark	Domain	Capability	#Train	#Val	#Test
Tau-Bench	Customer Service	Policy-aware tool use and user interaction	55	15	32
Tau-Bench Retail			500	20	115
ALFWorld	Embodied AI	Long-horizon planning and state tracking	153	15	134
TextCraft	Game	Hierarchical decomposition and dependency planning	60	15	100
WebShop	Web	Information seeking and constraint-aware navigation	84	25	50
ScienceWorld	Scientific Scenes	Grounded procedural and scientific reasoning	66	22	102

candidate archive remains available separately and stores complete programs, trajectories, scores, and parent relations. Thus, E_t does not replace the archive; it provides an issue-indexed control layer over candidate-level history.

After each evaluation, the issue manager compares the diagnostic report with the existing issue records. Observations are associated with an existing issue when they describe the same underlying behavioral failure, affected capability, and execution context, even if their surface descriptions differ across trajectories. Otherwise, a new stable issue identity is created. Supporting evidence stores references to the relevant trajectory segments and diagnostic findings rather than only an aggregate score change. External task priors are handled as provisional hypotheses: they may initialize issue records, but do not become confirmed failures unless supported by observed behavior.

The persistent issue state controls which problem should be addressed and where optimization should continue, but it does not prescribe a fixed implementation. Conditioned on the repair plan, the designer may add, remove, or rewrite any agent-side logic, such as prompts, tools, memory, and task-specific modules. After the revised candidate is evaluated, its diagnostic evidence and measured outcomes are written back to the corresponding issue records, forming an issue–intervention–outcome chain for the next round. This separation preserves the expressiveness of full-code optimization while supporting continuous repair across candidate generations.

D EXPERIMENTAL DETAILS

D.1 BENCHMARK DETAILS

To control computational costs, we construct representative task subsets for the main experiments. We preserve the native task structure and evaluator of each benchmark, and select subsets according to task semantics or trajectory complexity rather than sampling arbitrary individual episodes.

Tau-Bench (τ -Bench). Tau-Bench evaluates agents in tool–agent–user interactions, requiring them to communicate with a simulated user, comply with domain-specific policies, and invoke tools that modify an underlying database state (Yao et al., 2025). We use the retail domain at two evaluation scales.

For the main experiments, we use all qualified return-related tasks, comprising 55 training, 15 validation, and 32 test examples. Return requests form a coherent but nontrivial intent family: agents must identify the relevant order and items, verify policy constraints, collect missing user information, and execute the appropriate tool sequence. We adopt this setting primarily for computational efficiency. Restricting the optimization set to one complete intent family enables controlled comparison under a feasible and uniform budget.

To verify that the conclusions are not specific to return requests, we additionally repeat the comparison on the full Tau-Bench retail distribution, containing 500 training, 20 validation, and 115 test tasks. This expanded setting covers heterogeneous customer-service requests beyond returns and substantially increases the cost and diversity of optimization. The purpose is to show that the return subset provides a cost-efficient proxy for controlled comparison rather than an artificially favorable evaluation setting.

ALFWorld. ALFWorld uses text-based environments to simulate embodied household tasks and evaluates instruction understanding, planning, navigation, and object interaction (Shridhar et al., 2021). For example, the *Clean-and-Place* task family requires an agent to locate a target object,

Table 5: Comparison of baseline design scope and cross-round experience organization.

Method	Design Level	Editable Scope	Cross-Round Experience
Handcrafted	Fixed	None	None
SkillOpt (Yang et al., 2026)	Prompt	Skill document	Accepted/rejected edits and feedback
AHE (Lin et al., 2026a)	Harness	Pre-defined components	Records of edits, predictions, and outcomes
Meta-Harness (Lee et al., 2026)	Full code	Full agent program	Full candidate archive / raw history
DGM-H (Zhang et al., 2026a)	Full code	Full agent program	Full candidate archive / raw history
ADIAS (Ours)	Full code	Full agent program	Full candidate archive & Persistent issue state

identify and navigate to the required locations, clean the object, and place it in a specified place. Successful execution therefore requires long-horizon planning, environment state tracking, and recovery from incorrect intermediate actions. We randomly sampled 153 training samples, 15 validation samples, and used the official test set with 134 examples.

TextCraft. TextCraft is a text-based gaming environment designed to evaluate compositional planning over Minecraft-inspired crafting recipes (Prasad et al., 2024). Given a target item and a set of available crafting operations, the agent must reason recursively about prerequisite materials and execute a valid sequence of gathering and crafting actions. Recipe depth naturally controls task complexity, with deeper recipes requiring longer chains of dependent subgoals. To obtain a complexity-balanced subset, we stratify examples by recipe depth and sample tasks from depths 2, 3, and 4. Our final split contains 60 training and 15 validation examples, where each of the three recipe depths is equally distributed. We use the official test set with 100 samples.

WebShop. WebShop is a simulated e-commerce environment in which an agent must satisfy a natural-language shopping instruction by searching products, navigating result and product pages, inspecting attributes, and selecting an appropriate item for purchase (Yao et al., 2022). The environment evaluates information seeking, constraint tracking, long-horizon web navigation, and comparison among multiple candidate products. To emphasize genuinely interactive tasks, we construct a subset from successful human trajectories and retain instances requiring more than 10 non-redundant interaction steps. This filtering removes short or trivial purchases and focuses evaluation on tasks involving sustained navigation and decision-making. The resulting subset contains 84 training, 25 validation, and 50 test instances.

ScienceWorld. ScienceWorld evaluates scientific reasoning in an interactive text environment grounded in elementary-school science concepts (Wang et al., 2022). Agents must understand a scientific objective and carry out a valid sequence of observations, navigation actions, object manipulations, and experimental procedures. The benchmark tests procedural reasoning, state tracking, and the ability to translate scientific knowledge into executable actions. To reduce experimental cost, we first retain ScienceWorld instances whose successful reference trajectories can be completed within 30 interaction steps, and obtain 22 task types containing more than five eligible instances. For each retained task type, we sample three instances for training, one for validation, and up to five for testing. If fewer than five eligible test instances remain for a task type, we include all available instances without resampling or duplication. The resulting dataset contains 66 training, 22 validation, and 102 test examples.

D.2 BASELINES

Handcrafted. We use a manually designed ReAct-style agent (Yao et al., 2023) with task-level memory as the fixed, non-automated baseline. The agent follows a standard reasoning-and-acting loop and retains relevant interaction history across steps, but its strategy and execution structure are specified by humans and remain unchanged throughout evaluation.

SkillOpt. SkillOpt (Yang et al., 2026) performs text-level agent optimization by treating a persistent natural-language skill document as the editable state of a frozen task agent. A separate optimizer converts scored trajectories into bounded textual edits, and candidate updates are accepted through held-out validation evaluations. The task model and execution harness remain fixed, so the search space is restricted to the agent’s textual strategy representation.

AHE. Agentic Harness Engineering (AHE) (Lin et al., 2026a) performs architecture-level optimization over the agent harness. It exposes multiple editable components, including prompts, tools,

middleware, skills, memory, and other harness modules, and iteratively modifies these components. Compared with SkillOpt, AHE searches over a broader agent architecture and harness. AHE is originally designed for coding tasks, and we make an adaptation to general interactive tasks.

Meta-Harness. Meta-Harness (Lee et al., 2026) performs end-to-end optimization over the entire model harness. It uses an agentic proposer with filesystem access to the source code, evaluation scores, and execution traces of all previous candidates, which allows arbitrary code-level revisions and organizes cross-round experience as a raw candidate archive. Meta-Harness is originally evaluated on online text classification, retrieval-augmented mathematical reasoning, and agentic coding; we adapt it to general interactive tasks.

DGM-H. DGM-Hyperagents (DGM-H) (Zhang et al., 2026a) represents the broadest full-code-level automated design baseline. It integrates the task agent and meta agent into a single editable program and allows the resulting hyperagent to modify both its task-solving logic and the mechanism used to generate future improvements. The method searches over arbitrary program logic, memory, and control flow, which provides a substantially less constrained code-level design space.

D.3 EVALUATION PROTOCOL

We evaluate each method on held-out test tasks after selecting the best agent according to validation performance. All methods use the same task splits, environment interfaces, evaluation budgets, and benchmark-specific evaluators.

Task Performance. We retain the native evaluation signal of each benchmark and define the reported task score as

$$\text{Score} = \begin{cases} \frac{1}{N} \sum_{i=1}^N \mathbb{I}[y_i = 1], & \text{Tau-Bench, ALFWorld, TextCraft,} \\ \frac{1}{N} \sum_{i=1}^N r_i, & \text{WebShop, ScienceWorld,} \end{cases} \quad (15)$$

where N denotes the number of evaluation tasks, $y_i \in \{0, 1\}$ is the binary task-success indicator, and $r_i \in [0, 1]$ is the normalized benchmark-specific task score. For Tau-Bench, ALFWorld, and TextCraft, task performance is measured by the success rate. For WebShop and ScienceWorld, we report the average native task score, which provides graded credit for partial task progress. The native reward of WebShop measures how well the purchased product satisfies the attributes and constraints specified in the shopping instruction (Yao et al., 2022). The score of ScienceWorld captures the degree of progress toward completing the required scientific procedure (Wang et al., 2022). For presentation, all reported scores are multiplied by 100.

Task Interaction Efficiency. We measure how effectively a task agent converts environment interactions into task performance. Let $\bar{L}$ denote the average number of environment interaction steps and S denote the benchmark task score defined above. We define interaction efficiency as

$$\text{Eff.} = \frac{\text{Score}}{\bar{L}}. \quad (16)$$

This performance-per-interaction ratio jointly accounts for task effectiveness and environment interaction cost. We compare efficiency only within the same benchmark, since the semantics and granularity of an environment step differ across domains.

Optimization Efficiency. Beyond the performance of the final agent, we evaluate how rapidly an automated design method discovers stronger agents under a fixed optimization budget. Let V_t denote the validation score of the agent generated at optimization iteration t . We track the best-so-far validation score

$$B_t = \max_{1 \leq j \leq t} V_j, \quad (17)$$

which measures the quality of the agent available after t optimization iterations. A method with a rapidly increasing B_t discovers high-performing designs with fewer search iterations and is therefore more optimization-efficient.

Table 6: Multiple runs of evaluation on Tau-Bench. Pass@2 counts a task as successful if either rollout succeeds, whereas Pass² requires both rollouts to succeed. Best and second-best results are highlighted in green and blue, respectively.

Method	Pass 1	Pass 2	Pass@2	Pass ²	Avg. $\pm$ Std.
Handcrafted	75.0	68.8	87.5	56.3	71.9 $\pm$ 4.4
SkillOpt	12.5	6.3	15.6	3.1	9.4 $\pm$ 4.4
Meta-Harness	43.8	43.8	53.1	34.4	43.8 $\pm$ 0.0
AHE	56.2	37.5	68.8	25.0	46.9 $\pm$ 13.2
DGM-H	59.4	43.8	65.6	37.5	51.6 $\pm$ 11.0
ADIAS (Ours)	81.3	81.3	90.6	71.9	81.3 $\pm$ 0.0
<i>Ablations</i>					
w/o External Prior	75.0	75.0	90.6	59.4	75.0 $\pm$ 0.0
w/o Round-Level Diagnosis	78.1	71.9	87.5	62.5	75.0 $\pm$ 4.4
w/ Archive-Wide Synthesis	65.6	62.6	78.1	50.0	64.1 $\pm$ 2.1
w/ Best-Candidate Revision	71.9	71.9	81.3	62.5	71.9 $\pm$ 0.0
w/ Latest-Candidate Continuation	62.5	62.5	75.0	50.0	62.5 $\pm$ 0.0

Optimization Stability. We further measure whether an optimization procedure consistently generates effective agents rather than relying on isolated high-scoring candidates. At iteration t , we compute the cumulative average validation score over all agents generated so far:

$$\bar{V}_t = \frac{1}{t} \sum_{j=1}^t V_j. \quad (18)$$

While B_t captures the best agent discovered by the search process, $\bar{V}_t$ characterizes the overall quality of its generated agent population. A consistently high or increasing A_t indicates that effective design decisions accumulate throughout optimization, whereas a large gap between B_t and $\bar{V}_t$ suggests that performance depends on occasional favorable candidates. We visualize B_t and $\bar{V}_t$ over optimization iterations to jointly analyze optimization efficiency and stability.

E MORE RESULTS

E.1 EVALUATION ROBUSTNESS

The main experiments report the result of the first evaluation pass for a consistent single-rollout comparison across benchmarks. We conduct a second independent rollout on Tau-Bench to examine whether the reported performance is robust to execution randomness. Both passes use the same agents, task instances, and evaluation protocol.

For task i , let $y_i^{(j)} \in \{0, 1\}$ denote whether the j -th rollout succeeds. We report two complementary repeated-execution metrics. Pass@ k measures whether a task succeeds in at least one of the k rollouts:

$$\text{Pass@}k = \frac{1}{N} \sum_{i=1}^N \left(1 - \prod_{j=1}^k (1 - y_i^{(j)}) \right). \quad (19)$$

In contrast, Pass ^{k} measures consistent success across all k rollouts (Yao et al., 2025):

$$\text{Pass}^k = \frac{1}{N} \sum_{i=1}^N \prod_{j=1}^k y_i^{(j)}. \quad (20)$$

Pass@ k therefore captures whether the agent can solve a task in multiple runs, whereas Pass ^{k} provides a stricter measure of whether it solves the task reliably. We additionally report the arithmetic mean of the two individual pass scores. The same score of two passes merely indicates that the success rates of the two aggregations are the same; it does not imply that the two aggregations resolved exactly the same set of tasks. The consistency of the specific tasks is more accurately reflected by Pass ^{k} .

Table 7: Results on the complete Tau-Bench retail distribution. Pass@2 counts a task as successful if either evaluation rollout succeeds, whereas Pass² requires both rollouts to succeed. Best and second-best results are highlighted in green and blue, respectively.

Method	Pass 1 $\uparrow$	Eff. $\uparrow$	Pass 2 $\uparrow$	Pass@2 $\uparrow$	Pass ² $\uparrow$	Avg. $\pm$ Std. $\uparrow$
Handcrafted	68.7	5.13	67.0	81.7	53.9	67.8 $\pm$ 1.2
SkillOpt	6.1	0.23	5.2	7.8	3.5	5.7 $\pm$ 0.6
Meta-Harness	58.1	4.12	58.1	69.5	46.7	58.1 $\pm$ 0.0
AHE	67.8	4.13	64.3	80.9	51.3	66.1 $\pm$ 2.5
DGM-H	63.5	3.94	59.1	77.4	45.2	61.3 $\pm$ 3.1
ADIAS (Ours)	81.7	6.24	79.1	92.2	68.7	80.4 $\pm$ 1.8

Overall, the repeated-evaluation results preserve the conclusions of the main experiments and show that ADIAS illustrates robust execution performance rather than a favorable outcome from a single rollout. As shown in Table 6, ADIAS obtains identical success rates of 81.3 across the two independent evaluation passes, exceeding the strongest baseline average of 71.9 by 9.4 points. More importantly, ADIAS achieves a Pass² score of 71.9, outperforming the strongest non-ADIAS baseline by 15.6 points. This indicates that its advantage is not driven solely by tasks that succeed under favorable sampling; a substantially larger fraction of tasks are solved consistently across both executions. The ablations further support the importance of the proposed design. Removing either the external prior or round-level diagnosis reduces the two-pass average to 75.0 and lowers Pass² to 59.4 and 62.5, respectively. Replacing the persistent issue-centric state with candidate-based revision policies produces larger reductions in both average performance and consistent success.

E.2 EXPANDED-DISTRIBUTION EVALUATION

The main experiments use the return-intent subset of Tau-Bench to reduce the substantial costs. To verify that the observed gains are not specific to this task family, we further optimize and evaluate all methods on the complete Tau-Bench retail distribution, which contains 500 training, 20 validation, and 115 test tasks spanning a broader range of customer-service requests and interaction patterns. Under this expanded setting, each method is optimized for 15 iterations using 20 training episodes per iteration.

As shown in Table 7, ADIAS remains the strongest method when optimization is scaled to the complete retail distribution. It achieves a Pass-1 success rate of 81.7, outperforming the strongest baseline by 13.0 percentage points. The improvement is not obtained at the expense of interaction efficiency: ADIAS reaches an efficiency of 6.24, compared with 5.13 for the strongest baseline.

The advantage also persists across repeated executions. ADIAS obtains an average success rate of 80.4 $\pm$ 1.8, exceeding the strongest baseline average of 67.8 $\pm$ 1.2 by 12.6 points. Its Pass² score reaches 68.7, outperforming the best baseline by 14.8 points, while its Pass@2 score of 92.2 indicates broad task coverage across the two executions. These results show that ADIAS improves both task performance and execution consistency on the broader retail distribution.

E.3 TRADE-OFF BETWEEN EXPRESSIVENESS AND SEARCHABILITY

The automated agent design baselines exhibit different behaviors as the editable agent-design space expands from text/prompt to harness/architecture components and full code level.

SkillOpt operates at the text level by optimizing a persistent skill document while leaving the agent structure fixed. This restricted design space enables controlled updates, but prevents the optimizer from introducing new modules when the task requires capabilities absent from the initial agent. For example, on ALFWorld, effective long-horizon interaction requires persistent state tracking and memory mechanisms. As SkillOpt cannot alter the agent architecture to introduce such mechanisms, the performance is significantly lower than others.

AHE expands the editable space to the agent harness and can modify multiple components, such as prompts, tools, skills, and memory. This broader scope enables useful improvements across different environments, and AHE achieves non-trivial gains on all five benchmarks. However, its

search remains constrained by a predefined set of harness components and interfaces. As a result, the performance of AHE generally falls to the middle of the performance range.

DGM-H removes this structural restriction by allowing full-code-level self-modification. Its larger design space provides a higher performance ceiling, as illustrated by its strong result on ALFWorld. However, unrestricted code search introduces a substantially more difficult optimization problem. Under a limited evaluation budget, useful agent designs may be extremely sparse in the space of arbitrary program modifications. Thus, DGM-H fails to consistently outperform other baselines.

ADIAS resolves this trade-off between *expressiveness* and *searchability*. Similar to DGM-H, ADIAS retains a full-code design space and can introduce new agent mechanisms when required by the task. Unlike unconstrained code-level evolution, however, ADIAS narrows the effective search space with task priors that guide the designer toward mechanisms likely to be useful for the target environment. In addition, the *diagnostic agent* performs fine-grained failure analysis and precisely attributes observed errors to actionable causes. Finally, a global *issue manager* aggregates recurring patterns across episodes and iterations to prioritize broadly useful modifications and improve optimization efficiency. Overall, ADIAS preserves a high-capacity search space while using task priors to guide exploration, precise diagnosis to localize failures, and global issue state to drive efficient improvement.

E.4 OPTIMIZATION OF THE OPTIMIZATION PROCESS ITSELF

Because ADIAS optimizes the full agent harness, its editable space is not restricted to the task agent. Components involved in diagnosing failures and planning subsequent revisions may also be modified, which allows the optimization process itself to become part of the search space. We observe several such modifications during optimization. The issue manager is revised to normalize the revision plan into a fixed schema containing the target failure mode, selected parent generation, and revision rationale. The code improver is also revised to simplify how the target failure mode is extracted from the revision plan. These edits indicate that ADIAS can identify and revise interfaces within the optimization machinery, rather than modifying only task-execution behavior.