RECREATE: REASONING AND CREATING DOMAIN AGENTS DRIVEN BY EXPERIENCE

Zhezheng Hao[1] **Hong Wang**[2] **Jian Luo**[3] **Jianqing Zhang**[4] **Yuyan Zhou**[2]
Qiang Lin[2] **Can Wang**[1] **Hande Dong**[2†] **Jiawei Chen**[1†]
[1] Zhejiang University [2] Tencent [3] Independent Researcher [4] Shanghai Jiao Tong University

ABSTRACT

Large language model agents are reshaping the industrial landscape. However, most practical agents remain human-designed because tasks differ widely, making them labor-intensive to build. This situation poses a central question: *can we automatically create and adapt domain agents in the wild?* While several recent approaches have sought to automate agent creation, they typically treat agent generation as a black-box procedure and rely solely on final performance metrics to guide the process. Such strategies overlook critical evidence explaining why an agent succeeds or fails, and often require high computational costs. To address these limitations, we propose **ReCreate**, an experience-driven framework for the automatic creation of domain agents. ReCreate systematically leverages agent interaction histories, which provide rich concrete signals on both the causes of success or failure and the avenues for improvement. Specifically, we introduce an *agent-as-optimizer* paradigm that effectively learns from experience via three key components: (i) an experience storage and retrieval mechanism for on-demand inspection; (ii) a reasoning–creating synergy pipeline that maps execution experience into scaffold edits; and (iii) hierarchical updates that abstract instance-level details into reusable domain patterns. In experiments across diverse domains, ReCreate consistently outperforms human-designed agents and existing automated agent generation methods, even when starting from minimal agent scaffolds. [1]

1 INTRODUCTION

As the capabilities of large language models (LLMs) continue to improve (Open AI, 2025; Google, 2025; Anthropic, 2025b; Liu et al., 2025), LLM-based agents have demonstrated striking competence on complex, long-horizon tasks, such as software engineering (Yang et al., 2025; 2024), scientific discovery (Tang et al., 2025; Weng et al., 2024), and web navigation (He et al., 2024; Team et al., 2025). These LLM-based agent systems are typically built on *agent scaffolds* that specify how the model is prompted, how tasks are decomposed and executed, and how tools and environment feedback are integrated (Luo et al., 2025; Xi et al., 2025). The success of these LLM-based agents shows that designing agent scaffolds is a critical step toward unlocking raw LLMs' capabilities and grounding raw LLMs in practical deployments (Wang et al., 2024a; Li et al., 2025c).

Yet in practice, LLM agents still rely on human-designed scaffolds, since different domains call for distinct knowledge and priors (Xia et al., 2024; Ma et al., 2024; Li et al., 2025a;b). Designing scaffolds manually is labor-intensive and hard to scale to numerous open-world scenarios. This tension raises a central question: *can we automatically create domain agents from scratch in real-world environments?* In this work, we refer to this problem as the *domain agent creation*.

A growing line of work studies automated agent generation — replacing human-crafted scaffolds with a meta-agent that iteratively proposes, evaluates, and refines task-agent scaffolds (Hu et al., 2024; Shang et al., 2024; Zhang et al., 2024a; Li et al., 2025d; Wang et al., 2025). The generation in these methods is typically driven by **performance metrics** (such as pass rates or LLM-judged

[†]Corresponding Authors
[1]Code is available at `https://github.com/zz-haooo/ReCreate`.

scores), which are used to select and update candidate agents. While this strategy has yielded promising progress, relying solely on performance metrics presents two limitations: 1) Performance metrics do not provide evidence about *why* and *how* the agent succeeds or fails. Consequently, agent optimization is typically treated as a black-box process, relying on exhaustive trial-and-error to uncover effective directions for scaffold improvement, thereby undermining both efficiency and effectiveness. 2) Obtaining this metric value is often costly. Each candidate scaffold typically requires large-scale evaluation to yield a stable and reliable performance measure. For example, ADAS (Hu et al., 2024) spends about $500 for a single agent generation on ARC dataset (Chollet, 2019) with only 20 task samples.

These limitations stem from treating domain agent creation as a black-box optimization driven purely by performance scores. Motivated by this, we shift towards a white-box optimization paradigm that leverages the agent's **interaction experience** — including execution trajectories, evaluation logs, and environment state — as primary evidence for scaffold refinement. Such experience provides insight into *why* an agent succeeds or fails, offering semantic and concrete evidence for adding rules, updating tools, and revising workflows (Section 3 provides illustrative examples).

To implement this idea, we introduce *ReCreate*, an experience-driven framework for automatically creating domain agents. ReCreate explicitly exploits rich interaction experiences on tasks to guide scaffold updates. However, this process faces three key challenges: (i) The large scale of interaction and environment information is challenging for LLMs to tackle; (ii) Extracting meaningful evidence from complex experiences and converting it into suitable scaffold updates is inherently nontrivial. (iii) Scaffold updates may easily overfit to single-task experiences rather than capture broader domain patterns, thus hindering domain-level generalization. We address these challenges through an *agent-as-optimizer* design comprising three components: 1) an experience storage and retrieval mechanism that enables on-demand evidence inspection within the ReCreate environment; 2) a reasoning–creation synergy pipeline that maps execution evidence into scaffold updates; 3) a hierarchical update mechanism that aggregates instance-level refinements into reusable domain-level patterns. Empirical validations on diverse domains confirm the effectiveness of the ReCreate, which achieves superior and low-cost domain adaptation even when starting from trivial seed scaffolds.

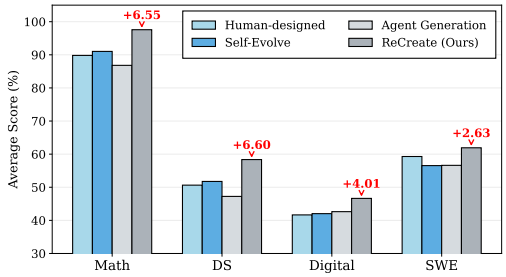


Figure 1: Overall performance across four domains with `gpt-5-mini`. Bars report domain-level average scores of three method families and the proposed ReCreate. Red annotations indicate ReCreate's improvement over the runner-up in each domain.

Overall, the major contributions are:

- *The ReCreate framework:* We highlight the importance of interaction-experience information and propose ReCreate, a framework that automatically creates agent scaffolds by learning from interaction experience rather than relying solely on performance metrics.
- *Agent-as-optimizer design:* Within ReCreate, we introduce an agent-as-optimizer design that efficiently processes large-scale experience logs, infer actionable scaffold modifications from execution evidence, and extract reusable domain-level patterns.
- *Comprehensive evaluation:* We evaluate ReCreate on thirteen benchmarks across four domains, showing consistent performance gains over human-designed agents and existing agent creating methods.

2 PRELIMINARIES

2.1 AGENT SCAFFOLDS

An LLM agent can be viewed as the composition of a base model ϕ and an agent scaffold $\mathcal{A}$ (Suzgun & Kalai, 2024; Xi et al., 2025). Formally, given a base LLM ϕ, an agent scaffold $\mathcal{A}$ denotes the surrounding software layer that makes the base LLM ϕ executable in an environment (Anthropic,

2025a; Meireles et al.). To make agent scaffold editable, we systematically examined recent open-source, general-purpose agent scaffolds and abstracted their common design patterns (Yang et al., 2024; Wang et al., 2024c; Liang et al., 2025). Based on their functions, we decompose $\mathcal{A}$ into the following complementary modular components:

- **Role & Objective**: the system instruction that defines the agent's role, domain priors, high-level goals, and responsibility;
- **Process & Strategy**: the standard operating procedures (SOPs) or workflows that guide step-by-step reasoning, intermediate checks, and termination criteria;
- **Action & Tool**: the action space exposed to the agent, implemented as reusable tools and scripts, including memory tools, search tools, etc;
- **Memory & Retrieval**: the static memory and the mechanism that controls how the agent stores or retrieves memory.

We therefore decompose agent scaffold $\mathcal{A}$ into a tuple $\mathcal{A} = \left(\mathcal{A}^{\text{role}}, \mathcal{A}^{\text{proc}}, \mathcal{A}^{\text{tool}}, \mathcal{A}^{\text{mem}}\right)$, with each component editable. In our implementation, other components, such as the action–observation format and error format, are ignored as they are non-essential to practical performance.

2.2 Domain Agent Creation Problem

Many real-world domains lack expert-crafted agents, offering no ready-to-use workflows, rules, or tools. In this scenario, available resources are limited to a base LLM ϕ, a distribution $\mathcal{D}$ of verifiable tasks, and minimal domain information $\mathcal{I}$ (e.g., interfaces and constraints). Formally, the domain agent creation problem seeks to construct a scaffold $\mathcal{A}$ from the tuple $(\phi, \mathcal{D}, \mathcal{I})$ that transforms ϕ into a reliable agent for $\mathcal{D}$. Unlike prompt tuning or tool learning, the goal is not adaptation to specific queries, but the creation of a plug-in agent that captures domain knowledge and generalizes to unseen tasks.

3 Motivation: Experience Matters

Our core motivation is that interaction experience, which contains the full trajectory, execution results and evaluation results, carries the agent's action path and reasoning process. The information can be leveraged to design stronger agent scaffolds. To illustrate, we present representative patterns in interaction experience that suggest scaffold updates in the form of rules, tools, and workflows.

Experience suggests adding rules. Agents often overlook domain priors unless explicitly guided. For example, in DA-Code (Huang et al., 2024), experience shows that agents often evaluate models using training accuracy without a validation split. Although the base LLM can conduct proper cross-validation, it will not do so reliably if the scaffold does not emphasize this protocol. This suggests a simple scaffold update: add a rule that any model evaluation should construct a train/validation split and report performance on the validation set, rather than on the training data.

Experience suggests creating tools. Repetitive or error-prone steps can be replaced by dedicated tools. For example, experience shows that agents often need to verify whether the solution is non-empty, executable, well-structured, etc. Creating a tool to run these checks not only simplifies the workflow but also ensures that all these validations are applied.

Experience suggests modifying workflows. Failures often stem from the wrong order of actions rather than the lack of capability. For example, in SWE-bench (Jimenez et al., 2023), experience shows that agents often commit changes before submission, leading to an empty patch in evaluation. A simple update is to add a pre-submission validation step that runs `git diff --cached` before finish. Notably, with only this change, agent improves the pass-rate by over 2%.

These cases are detailed in Appendix J. These examples illustrate that interaction experience reveals agent behavior patterns, which can be leveraged to design better scaffolds. This motivates an experience-driven scaffold optimizer that inspects execution traces and converts such evidence into targeted scaffold updates.

4 METHOD

In this section, we begin with problem formulation and ReCreate framework, and then detail our agent-as-optimizer design for scaffold optimization.

4.1 PROBLEM FORMULATION

We first formalize the domain agent creation problem. Given domain $\mathcal{D}$, we denote each task t_i as

$$t_i \triangleq \big(x_i, \mathrm{Env}_i\big), \tag{1}$$

where x_i denotes the problem context and Env_i denotes an executable environment of the given domain. For example, in software engineering, x may contain an issue description and code snippets, while the Env contains the repository, runtime, and unit tests.

Given an agent scaffold $\mathcal{A}$, a base model ϕ, and task t_i, the task agent produces an interaction trajectory

$$\tau_i \sim P_\phi(\cdot \mid \mathcal{A}, t_i), \tag{2}$$

where each trajectory τ_i is a sequence of reasoning steps, tool use, and observations. After the τ_i is generated, an agent submission can be $\mathrm{Exec}\big(\tau_i, t_i\big)$ obtained (e.g., a patch or generated codes). A task-specific verifier then evaluates this submission and produces a performance metric r_i:

$$r_i = \mathrm{Ver}\big[\mathrm{Exec}(\tau_i, t_i)\big] \in \mathcal{R}, \tag{3}$$

where r_i could be pass/fail signals from unit tests on software engineering tasks, or scores from evaluation scripts on scientific tasks.

Given the tuple $(\phi, \mathcal{D}, \mathcal{I})$, domain agent creation can then be formulated as the following bi-level optimization problem:

$$\begin{aligned} &\max_{\mathcal{A}} \ \mathbb{E}_{t_i \sim \mathcal{D}} \mathrm{Ver}\big[\mathrm{Exec}\big(\tau_i^*(\mathcal{A}, t_i), t_i\big)\big] \\ &\text{s.t. } \tau_i^*(\mathcal{A}, t_i) \in \arg \max_{\tau_i \sim P_\phi(\cdot \mid \mathcal{A}, t_i)} \mathrm{Ver}\big(\mathrm{Exec}(\tau_i, t_i)\big). \end{aligned}$$

The inner-level optimization is generating a trajectory τ to maximize the task performance under current scaffold $\mathcal{A}$. The outer-level optimization is creating an agent scaffold $\mathcal{A}$ to maximize the expected performance in domain $\mathcal{D}$. In practice, the bi-level objective can be approximated via iterative scaffold updates: at iteration k, run $\mathcal{A}_k$ on tasks, obtain feedback, and update it to $\mathcal{A}_{k+1}$, starting from $\mathcal{A}_0$ derived from minimal domain information $\mathcal{I}$.

Existing automated agent generation methods can be abstracted as a metric-based update:

$$\mathcal{A}_{k+1} = \text{Meta-Agent}\big(\mathcal{A}_k, r\big).$$

Here, the entire execution process is compressed into a single metric, which lacks process information. Instead, we propose to update scaffolds from interaction experience:

$$\begin{aligned} \mathcal{A}_{k+1} &= \text{ReCreate-Agent}\big(\mathcal{A}_k, e\big), \\ \text{where } \ e &\triangleq \big(\tau, \ \mathrm{Exec}, \mathrm{Ver}\big). \end{aligned} \tag{4}$$

Here the interaction experience e contains the full trajectory, execution results and evaluation results. In this way, the outer optimization can use full inner-level information for scaffold updates.

4.2 THE RECREATE FRAMEWORK

As illustrated in Figure 2, ReCreate formulates the domain agent creation as a bi-level optimization process, which imitates how human experts iteratively refine software systems. In the inner loop, the agent equipped with scaffold $\mathcal{A}_k$ interacts with the environment to solve tasks. This loop captures the agent's task-solving process, which contains its task decomposition, chain-of-thought reasoning and tool usage patterns. In the outer loop, the ReCreate-Agent acts as a scaffold-optimizer. It inspects the collected experience to attribute why the agent succeeds or fails and generates targeted updates from $\mathcal{A}_k$ to $\mathcal{A}_{k+1}$. Unlike existing methods that treat agent generation as a black-box optimization problem guided solely by performance metrics, ReCreate reframes it as a white-box debugging process driven by rich interaction experience (i.e., agent trajectories, execution logs, and environmental states).

ReCreate bridges the gap between agent's execution behavior and agent scaffold design, enabling the creation of domain agents from minimal seeds. Despite its simplicity, the ReCreate framework parallels the workflow of human experts, yet with superior intelligence. This embodies a core philosophy: *as models cross the critical threshold of reasoning and creativity, the labor-intensive process of agent creation can finally be automated by the agents themselves.*

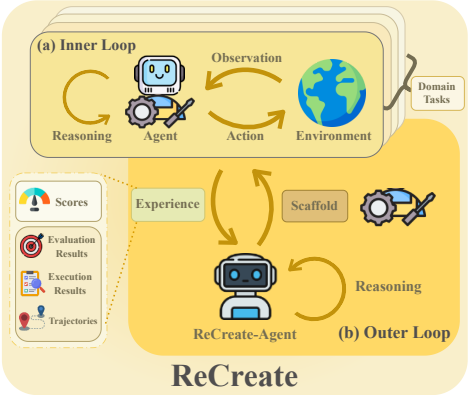


Figure 2: The overview of ReCreate.

4.3 The Agent-as-optimizer Design

While the ReCreate framework leverages interaction experience to improve agent creation, effectively exploiting it is non-trivial for three challenges: **(1)** the full interaction experience is large for LLM to tackle; **(2)** attributing agent experience to actionable agent scaffold updates is complex; **(3)** instance-level fixes often bring overfitting and fail to generalize. Next, we handle these challenges via three components in the Agent-as-optimizer design.

Experience Storage and Retrieval To handle long and noisy experience, we store each task-agent episode as an environment for ReCreate-Agent (call it ReCreate-Environment), which collects the current scaffold, the full interaction trajectories, execution/evaluation results and environment context (e.g., codebase, database and sandbox state). ReCreate-Environment supports on-demand inspection, allowing the ReCreate-Agent to actively retrieve the relevant piece of experience instead of the full experience. Typically, the ReCreate-Agent starts from failure or success information and interacts with ReCreate-Environment to progressively narrow down to the key evidence. To facilitate efficient retrieval, we introduce an *evidence retriever* that indexes critical events (e.g., errors, failing tests, file operations) and links them to their context. This allows ReCreate-Agent to jump from final evaluation information to the relevant context based on its reasoning capability.

Synergizing Reasoning and Creating While the ReCreate-Environment captures comprehensive interaction histories, the raw experiences are often complex to analyze, which brings a gap between experience and agent scaffold creation. To bridge this gap, the ReCreate-Agent acts as an optimizer for the scaffold in the ReCreate-Agent's environment, as illustrated in Figure 3. In the left part, ReCreate-Agent iteratively reasons and inspects the interaction experience to locate key evidence on why the agent succeeds or fails. The on-demand inspection is enabled by our experience storage and retrieval mechanism described above. In the right part, ReCreate-Agent iteratively reasons and creates components to improve scaffold. Specifically, we introduce a *creation router*, including a routing prompt and interfaces for scaffold editing. The *creation router* guides the ReCreate-Agent to decide *which* scaffold component to edit and *how* to edit it based on the retrieved evidence. This design ensures that every scaffold update is grounded in specific evidence in the interaction experience, rather than blind trial-and-error. Based on this pipeline, ReCreate-Agent synergizes Reasoning and Creating for experience-driven agent creation.

Hierarchical Local-to-Domain Updates To address the risk of instance-level overfitting, we propose a hierarchical update mechanism, which couples instance-level update UPD with domain-level update DOMUPD. At the instance level, the agent analyzes individual interaction experience to generate a candidate update $\Delta\mathcal{A}$ accompanied by its corresponding justification κ, which are buffered rather than immediately applied. At the domain level, the ReCreate-Agent synthesizes these instance-level proposals to extract domain patterns. This hierarchical process filters out task-specific noise, ensuring that only generalized updates are integrated into the final domain agent scaffold.

4.4 Comparing with Existing Methods

Comparison to Existing Self-Evolve Methods. Recent self-evolving methods (Xia et al., 2025; Yang et al., 2023; Zhao et al., 2024) also leverage experience to refine pre-existing agents. ReCre-

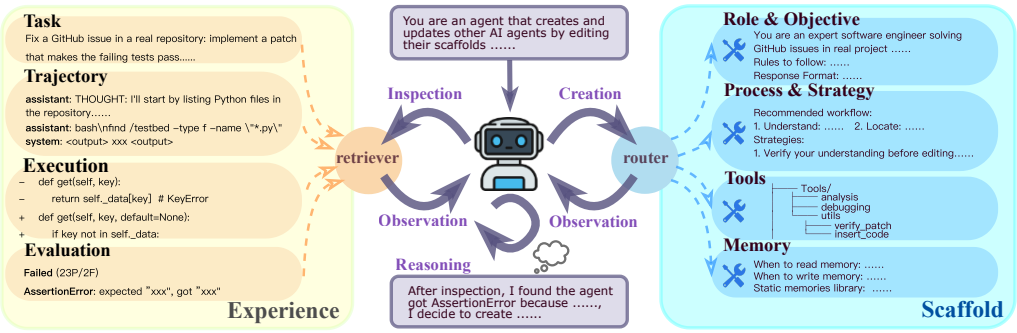


Figure 3: The pipeline of ReCreate framework. ReCreate-Agent iteratively reasons and acts to locate key evidence on why the agent succeeds or fails and reflect how to improve the agent scaffold.

ate differs from self-evolving methods in three aspects: (1) **Scope**: Instead of refining pre-defined scaffolds, ReCreate builds agents from scratch, broadening applicability to scenarios without mature agents. (2) **Objective**: Unlike these methods prioritizing instance-level success, ReCreate aims for domain-level generalization with leveraging hierarchical updates. (3) **Strategy**: Rather than relying on high-level outcomes, ReCreate conducts *fine-grained inspection* of execution traces, extracting concrete meaningful evidence for optimization. Empirically, ReCreate even initialized with a minimal scaffold outperforms these methods with fully-developed scaffolds (cf. Section 5).

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Datasets To validate the effectiveness of ReCreate across diverse real-world scenarios, we conduct experiments on four representative domains widely used for agent evaluation, including Software Engineering (SWE), Data Science (DS), Mathematics (Math), and Digital Assistance (Digital). Specifically, we instantiate these domains by using their most representative subsets: for SWE, we select the two largest repositories, *Django* and *SymPy*, from the SWE-bench-Verified (Jimenez et al., 2023); for DS, we select the three largest subsets from DA-Code (Huang et al., 2024) (*Data Wrangling*, *Machine Learning*, *Statistical Analysis*) and DS-1000 (Lai et al., 2023) (*NumPy*, *Pandas*, *Matplotlib*); for Math, we select the three sub-domain in MATH (Hendrycks et al.) (*Algebra*, *Number Theory*, *Counting&Probability*); for Digital, we select both the *Normal* and *Challenge* subsets of AppWorld (Trivedi et al., 2024). It is worth noting that DA-Code uses a continuous evaluation signal, producing a score in $[0, 1]$ (converted to a percentage in our reporting). In contrast, all other benchmarks considered here use deterministic binary outcomes (0 for failure and 1 for success). Detailed information for datasets are shown in Appendix G.

Implementations We employ `gpt-5-mini` as the backbone for the task agent to ensure inference efficiency and employ `claude-4.5-opus` as the ReCreate-Agent to guarantee high-quality reasoning and scaffold updates. We set the temperature to 0 for the `claude-4.5-opus` and 1.0 for the `gpt-5-mini` (fixed at 1.0 by the API). Following Algorithm 1, we configure the loop with a maximum iteration $N_{\max} = 2$ and a batch size of 4. For data partitioning, we randomly sample a small set of approximately 20 instances as the development set $\mathcal{D}_{\text{dev}}$ for each domain, reserving all remaining data for testing (ranging from 38 to 417). All tasks are executed within Docker sandboxes. Detailed statistics of data splits and prompts for the ReCreate-Agent are provided in Appendix G.

Baselines We compare ReCreate against three related categories of methods to ensure a comprehensive evaluation. The first category is human-designed scaffolds with test-time scaling, including CoT (Wei et al., 2022), Step-Back Abstraction (short as SBA) (Zheng et al., 2023), and Self-Refine(short as Refine) (Madaan et al., 2023). The second category is Self-Evolving methods, where agents autonomously refine themselves for solving tasks. We select representative methods for different evolution targets: LIVE (Xia et al., 2025) for tool evolution, OPRO (Yang et al., 2023) for prompt optimization, and ExpeL (Zhao et al., 2024) for experience accumulation. The third

Domain	Dataset	Human-designed			Self-Evolve			Agent Generation		Ours
		CoT	SBA	Refine	LIVE	OPRO	ExpeL	ADAS	Square	ReCreate
SWE	*Django*	58.29	58.77	56.87	58.77	52.13	59.24	55.92	57.82	**60.19**
	Sympy	61.82	61.82	58.18	58.18	54.55	56.36	58.18	54.55	**63.64**
DS	*DW*	42.81	41.66	42.43	38.55	18.55	47.50	37.98	45.98	**51.94**
	ML	34.32	36.25	35.85	41.68	39.08	40.27	21.53	22.90	**42.88**
	SA	19.33	20.15	20.39	21.83	17.16	22.44	15.66	11.99	**24.50**
	Numpy	62.00	64.50	64.00	68.00	71.00	68.50	61.00	67.00	**77.00**
	Pandas	62.73	61.25	63.10	64.21	63.47	64.94	60.52	63.10	**68.63**
	Matplotlib	78.52	80.74	81.48	82.96	82.96	78.52	76.30	82.96	**85.19**
Math	*Algebra*	81.45	85.48	84.68	85.48	87.10	83.87	80.65	83.87	**92.74**
	NT	91.94	90.32	90.32	93.55	**100.00**	90.32	83.87	93.55	**100.00**
	C&P	94.74	94.74	94.74	92.11	94.74	92.11	84.21	94.74	**100.00**
Digital	*Normal*	48.81	48.81	47.02	47.62	51.79	49.40	50.00	48.81	**52.98**
	Challenge	34.05	36.21	35.01	34.53	34.29	34.53	36.93	34.77	**40.29**
Average		59.29	60.05	59.54	60.57	58.99	60.62	55.60	58.62	**66.15**

Table 1: Pass rate or testing score on various real-world agent benchmarks.

category is Automated Agent Generation, including ADAS (Hu et al., 2024), AgentSquare (Shang et al., 2024) (short as Square). It is noteworthy that the meta agents used in these baselines are also `claude-4.5-opus` for fairness comparison.

5.2 Main Results

Table 1 reports the main results across four domains. First, ReCreate consistently exceeds both human-designed scaffolds and self-evolving baselines across all domains. On average across all benchmarks, ReCreate improves the overall score by more than 5% over the strongest competing method, especially with clear performance gains on DS, Math, and Digital tasks. These notable results are because these baselines rely on human prior knowledge encoded in hand-crafted scaffolds, which can be difficult to acquire and may not generalize well when domain knowledge is scarce. Second, ReCreate delivers substantial gains over Agent Generation methods, improving the overall average by more than 7%. This highlights the effectiveness of leveraging interaction experience, rather than relying solely on a scalar score, for domain agent creation. Besides, Agent Generation methods typically search for or compose agents from a pre-built component pool, while ReCreate updates the scaffold directly from execution experience, without requiring any predefined modules.

5.3 Ablation Study

Observation-level ablation Figure 4 reports ablations over the experience components in ReCreate. Across all five domains, the full ReCreate model has the best performance. Removing any component degrades its performance, which confirms the importance of experience components. Among the variants, removing the full trajectory causes the largest and most consistent performance drop, highlighting that step-by-step traces provide crucial context for diagnosing failures and guiding creation. Removing execution & evaluation feedback also leads to the performance drop, suggesting that outcome signals (e.g., generated files, test results, verifier feedback) are necessary to anchor updates. Removing the environment yields a smaller but consistent decline, indicating the value of runnable execution for faithful inspection and debugging. These results underscore the complementary roles of trajectory, environment and exec/eval feedback for domain agent creation.

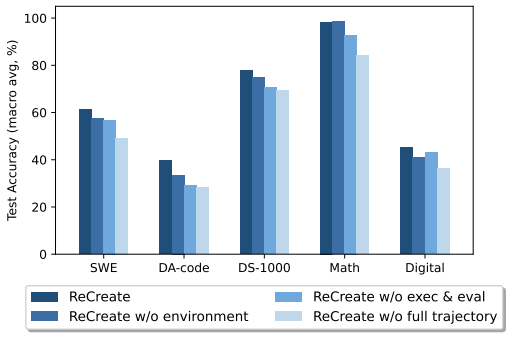


Figure 4: Ablations on experience components.

Reasoning-level ablation. The ReCreate framework requires strong reasoning capability to achieve reliable agent creation. Figure 5 compares ReCreate-Agent with different reasoning capacities and the task agent is fixed as `gpt-5-mini`. The radar values are normalized by ReCreate with `Claude-opus`, indicating each setting's relative performance ratio. We draw two conclusions. First, ReCreate with only initial domain information yields very poor performance in most domains (except Math). This indicates that initial domain information alone is far from sufficient and that effective scaffolds require richer domain knowledge from interaction experience or experts. Second, ReCreate with `Claude-opus` consistently surpasses `Human-designed` scaffolds, whereas ReCreate with `gpt-5-mini` fails to outperform them in most domains. This gap indicates that the stronger reasoning capability substantially improves the ReCreate-Agent's ability to interpret interaction experience and translate it into actionable scaffold updates. Furthermore, it suggests that frontier LLMs are approaching the point of matching or even replacing expert-designed scaffolds in practice.

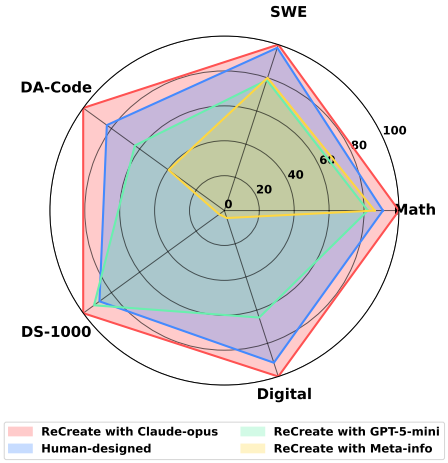
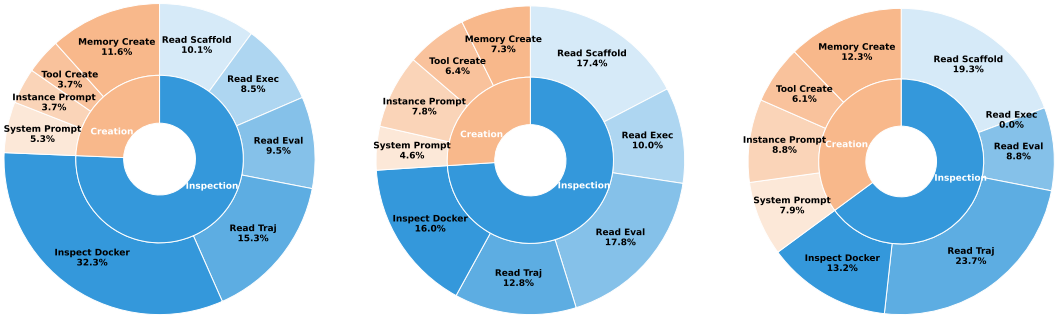


Figure 5: Reasoning-level Ablation.

Action-level ablation. Table 2 ablates two action-level components in ReCreate. Removing either creation router or domain update DOMUPD consistently hurts the performance across SWE, DA-Code, and DS-1000 (DS for short). Without *creation router*, the ReCreate-Agent tends to focus on instance prompt; without DOMUPD, the updates are biased toward instance details. Therefore, the two components are complementary: creation router improves execution reliability of ReCreate-Agent, while DOMUPD improves cross-task generalization.

Method	SWE	DA-Code	DS
ReCreate	60.19	39.77	77.74
w/o creation router	58.00	37.13	75.39
w/o DOMUPD	57.09	37.83	75.96

Table 2: Action-level Ablation.



(a) Django (b) Machine Learning (c) AppWorld

Figure 6: Action distributions of the ReCreate-Agent in various domains.

5.4 STATISTICAL STUDY

To look into the creation process, we count the action distribution of the ReCreate-Agent in three sub-domains, shown in Figure 6. Across all cases, inspection dominates creation (roughly 65%–76% vs. 24%–35%), indicating that ReCreate-Agent typically *locates and verifies evidence* more than proposing scaffold edits. This aligns with our agent-as-optimizer design, where scaffold updates are grounded in execution experience rather than blind trial-and-error.

The dominant evidence sources and update targets vary by domain. On Django, the agent heavily inspects the Docker sandbox for code base, while creation mainly manifests as memory construction

to consolidate debugging findings into reusable rules. On Machine Learning, inspection frequently focuses on scaffolds and evaluation artifacts, and creation is more balanced across prompt adjustments and tool/memory creations. On AppWorld, trajectory inspection is prominent and creation becomes notably more frequent, with more prompt and memory updates. In short, the agent's behavior is highly context-dependent, allocating its reasoning and creation efforts where they yield the highest value for the specific task.

5.5 Analysis of the Update Procedure

In this section, we dive into the update process in ReCreate and study how it affects agent performance on the development set $\mathcal{D}_{\text{dev}}$ and the test set $\mathcal{D}_{\text{test}}$.

Post-update Gain Verification We consider a single-case setting where the development set contains only one instance. Starting from the initial scaffold, we iteratively apply ReCreate updates and re-evaluate the agent on the same case to quantify the post-update gain in pass rate. Figure 7 plots the average pass rate curve for 20 single-case development sets (i.e., 20 distinct cases) in the SWE domain under this setting, with a maximum of $N_{\text{max}} = 4$ update iterations. The left panel shows the average task-level resolved rate, while the right panel reports the fine-grained test-case pass rate. Under this setting, ReCreate steadily improves both the task's resolved rate and its test-case pass rate by iteratively updating the agent scaffold. These results suggest that ReCreate can fix a subset of tasks that initially failed through its experience-driven updates on agent scaffold edits.

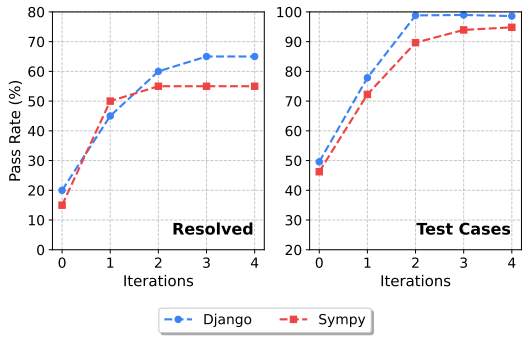


Figure 7: Post-update Gain on SWE tasks.

5.6 Case Study

We present task-solving cases to intuitively showcase how ReCreate evolves the agent scaffold; details are provided in Appendix L.

5.7 Cost

Beyond performance, we also assessed the cost-effectiveness of ReCreate compared to automated agent generation methods. Figure 8 compares the average cost of scaffold optimization across domains between ReCreate and ADAS. ReCreate is more efficient than ADAS, reducing the cost by roughly 36% to 82%. Even though ReCreate employs a strong ReCreate-Agent (e.g., `claude-4.5-opus`) for scaffold updates, it converges with a small development set and fewer iterations thanks to the rich signals from interaction experience. In contrast, ADAS has to repeatedly evaluate each candidate, leading to higher cost.

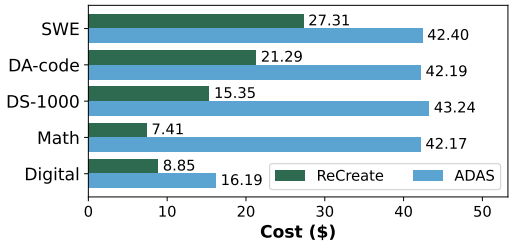


Figure 8: Cost Comparison (USD).

5.8 Generalization on Various Models

Generalization on Various Base Models for ReCreate Agent One may suspect that ReCreate's gains in the main results simply stem from the stronger reasoning capability of the selected ReCreate-Agent. We additionally evaluate ReCreate with Claude Opus 4.5 as both the ReCreate-Agent (Meta-Agent) and the task agent. The results are shown in Figure 9. ReCreate remains consistently better than baseline across all evaluated domains. As can be seen, ReCreate consistently

outperforms the human-designed agent across the four domains, demonstrating that our experience-driven, automatically designed paradigm can effectively replace carefully hand-crafted designs in many settings.

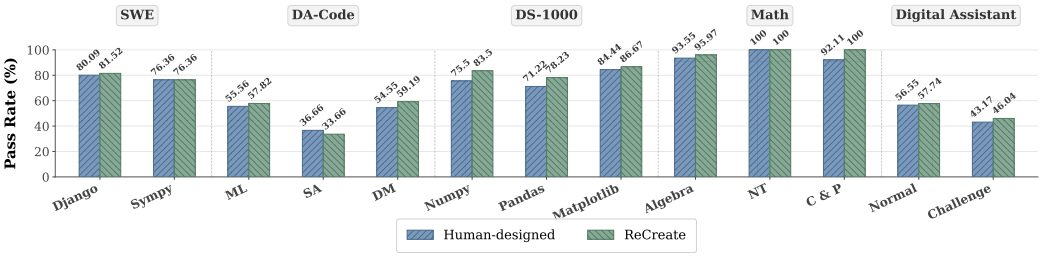


Figure 9: ReCreate on various base models for ReCreate Agent.

Generalization on Various Base Models for Task Agent Beyond the default backbone (GPT-5-mini) in our main experiments, we further evaluate ReCreate by swapping the base LLM from GPT-5-mini to Gemini-2.5-Flash and GPT-4o-mini. As shown in Figure 10, ReCreate consistently outperforms the human-designed scaffold (12/13 tasks on Gemini and 13/13 on GPT-4o-mini), delivering +8.3 and +8.7 absolute pass-rate points on average, with the largest gains on DS-1000 (up to +37.6). These results indicate that ReCreate's experience-driven scaffold edits capture largely model-agnostic role/process/tool/memory patterns rather than backbone-specific prompt tricks.

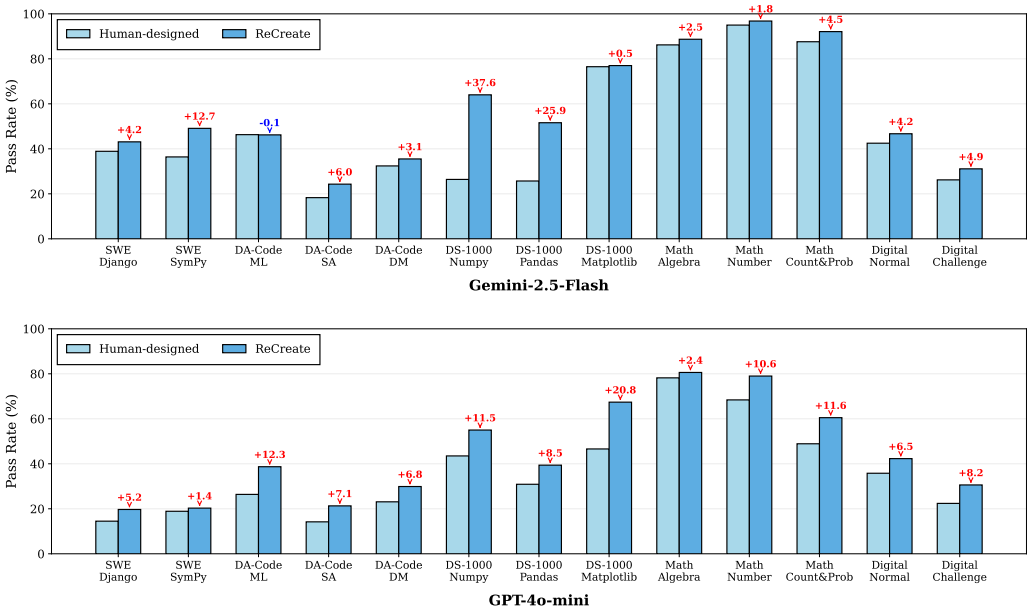


Figure 10: ReCreate on various base models for task agent.

6 Conclusion

We introduced ReCreate, an experience-driven framework for domain agent creation that optimizes agent scaffolds by learning from interaction experience rather than relying solely on performance metrics. Concretely, ReCreate adopts an agent-as-optimizer design with three components, enabling scaffold updates grounded in concrete evidence while improving task generalization. Empirically, ReCreate yields consistent performance gains over baselines across diverse domains even when starting from minimal seed scaffolds. More broadly, we believe our work will inspire future work on moving toward experience-centric agent learning.

REFERENCES

Anthropic. Raising the bar on swe-bench verified with claude 3.5 sonnet. https://www.anthropic.com/engineering/swe-bench-sonnet, 2025a.

Anthropic. Introducing claude opus 4.5. https://www.anthropic.com/news/claude-opus-4-5, 2025b.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*, 2025.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

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

Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.

Google. A new era of intelligence with gemini 3. https://blog.google/products/gemini/gemini-3/, 2025.

Zhenyu Guan, Xiangyu Kong, Fangwei Zhong, and Yizhou Wang. Richelieu: Self-evolving llm-based agents for ai diplomacy. *Advances in Neural Information Processing Systems*, 37:123471–123497, 2024.

Han Lee. Claude agent skills: A first principles deep dive. https://leehanchung.github.io/blogs/2025/10/26/claude-skills-deep-dive/, 2025.

Mohd Ariful Haque, Justin Williams, Sunzida Siddique, Md Hujaifa Islam, Hasmot Ali, Kishor Datta Gupta, and Roy George. Advanced tool learning and selection system (atlass): A closed-loop framework using llm. *arXiv preprint arXiv:2503.10071*, 2025.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

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

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, et al. Da-code: Agent data science code generation benchmark for large language models. *arXiv preprint arXiv:2410.07331*, 2024.

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? *arXiv preprint arXiv:2310.06770*, 2023.

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

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Jia Li, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Huang, Kashif Rasul, Longhui Yu, Albert Q Jiang, Ziju Shen, et al. Numinamath: The largest public dataset in ai4maths with 860k pairs of competition math problems and solutions. *Hugging Face repository*, 13(9):9, 2024.

Shanda Li, Tanya Marwah, Junhong Shen, Weiwei Sun, Andrej Risteski, Yiming Yang, and Ameet Talwalkar. Codepde: An inference framework for llm-driven pde solver generation. *arXiv preprint arXiv:2505.08783*, 2025a.

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. Search-o1: Agentic search-enhanced large reasoning models. *arXiv preprint arXiv:2501.05366*, 2025b.

Xiaoxi Li, Wenxiang Jiao, Jiarui Jin, Guanting Dong, Jiajie Jin, Yinuo Wang, Hao Wang, Yutao Zhu, Ji-Rong Wen, Yuan Lu, et al. Deepagent: A general reasoning agent with scalable toolsets. *arXiv preprint arXiv:2510.21618*, 2025c.

Yu Li, Lehui Li, Zhihao Wu, Qingmin Liao, Jianye Hao, Kun Shao, Fengli Xu, and Yong Li. Agentswift: Efficient llm agent design via value-guided hierarchical search. *arXiv preprint arXiv:2506.06017*, 2025d.

Xinbin Liang, Jinyu Xiang, Zhaoyang Yu, Jiayi Zhang, Sirui Hong, Sheng Fan, and Xiao Tang. Openmanus: An open-source framework for building general ai agents, 2025. URL `https://doi.org/10.5281/zenodo.15186407`.

Xuechen Liang, Yangfan He, Yinghui Xia, Xinyuan Song, Jianhui Wang, Meiling Tao, Li Sun, Xinhang Yuan, Jiayi Su, Keqin Li, et al. Self-evolving agents with reflective and memory-augmented abilities. *arXiv preprint arXiv:2409.00872*, 2024.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*, 2025.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, et al. Large language model agent: A survey on methodology, applications and challenges. *arXiv preprint arXiv:2503.21460*, 2025.

Yubo Ma, Zhibin Gou, Junheng Hao, Ruochen Xu, Shuohang Wang, Liangming Pan, Yujiu Yang, Yixin Cao, Aixin Sun, Hany Awadalla, et al. Sciagent: Tool-augmented language models for scientific reasoning. *arXiv preprint arXiv:2402.11451*, 2024.

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

Mariana Meireles, Rupali Bhati, Niklas Lauffer, and Cameron Allen. The influence of scaffolds on coordination scaling laws in llm agents. In *Workshop on Scaling Environments for Agents*.

Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. Metaicl: Learning to learn in context. In *Proceedings of the 2022 conference of the North American chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2791–2809, 2022.

Open AI. Introducing gpt-5.2. `https://openai.com/index/introducing-gpt-5-2`, 2025.

Zehua Pei, Hui-Ling Zhen, Shixiong Kai, Sinno Jialin Pan, Yunhe Wang, Mingxuan Yuan, and Bei Yu. Scope: Prompt evolution for enhancing agent effectiveness. *arXiv preprint arXiv:2512.15374*, 2025.

Cheng Qian, Chi Han, Yi Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 6922–6939, 2023.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*, 2023.

Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, et al. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution. *arXiv preprint arXiv:2505.20286*, 2025.

Rana Salama, Jason Cai, Michelle Yuan, Anna Currey, Monica Sunkara, Yi Zhang, and Yassine Benajiba. Meminsight: Autonomous memory augmentation for llm agents. *arXiv preprint arXiv:2503.21760*, 2025.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551, 2023.

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

Zhengliang Shi, Yuhan Wang, Lingyong Yan, Pengjie Ren, Shuaiqiang Wang, Dawei Yin, and Zhaochun Ren. Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. *arXiv preprint arXiv:2503.01763*, 2025.

Mirac Suzgun and Adam Tauman Kalai. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

Jiabin Tang, Lianghao Xia, Zhonghang Li, and Chao Huang. Ai-researcher: Autonomous scientific innovation. *arXiv preprint arXiv:2505.18705*, 2025.

Tongyi DeepResearch Team, Baixuan Li, Bo Zhang, Dingchu Zhang, Fei Huang, Guangyu Li, Guoxin Chen, Huifeng Yin, Jialong Wu, Jingren Zhou, et al. Tongyi deepresearch technical report. *arXiv preprint arXiv:2510.24701*, 2025.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. Appworld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901*, 2024.

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

Hongru Wang, Yujia Qin, Yankai Lin, Jeff Z Pan, and Kam-Fai Wong. Empowering large language models: Tool learning for real-world interaction. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 2983–2986, 2024a.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. Toolgen: Unified tool retrieval and calling via generation. *arXiv preprint arXiv:2410.03439*, 2024b.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024c.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *arXiv preprint arXiv:2310.16427*, 2023b.

Yinjie Wang, Ling Yang, Guohao Li, Mengdi Wang, and Bryon Aragam. Scoreflow: Mastering llm agent workflows via score-based preference optimization. *arXiv preprint arXiv:2502.04306*, 2025.

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

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

Yixuan Weng, Minjun Zhu, Guangsheng Bao, Hongbo Zhang, Jindong Wang, Yue Zhang, and Linyi Yang. Cycleresearcher: Improving automated research via automated review. *arXiv preprint arXiv:2411.00816*, 2024.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China Information Sciences*, 68(2):121101, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly? *arXiv preprint arXiv:2511.13646*, 2025.

Shengxiang Xu, Jiayi Zhang, Shimin Di, Yuyu Luo, Liang Yao, Hanmo Liu, Jia Zhu, Fan Liu, and Min-Ling Zhang. Robustflow: Towards robust agentic workflow generation. *arXiv preprint arXiv:2509.21834*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2023.

Jian Yang, Wei Zhang, Shark Liu, Jiajun Wu, Shawn Guo, and Yizhi Li. From code foundation models to agents and applications: A practical guide to code intelligence. *arXiv preprint arXiv:2511.18538*, 2025.

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

Seungyoun Yi, Minsoo Khang, and Sungrae Park. Zera: Zero-init instruction evolving refinement agent–from zero instructions to structured prompts via principle-based optimization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 23334–23348, 2025.

Li Yin and Zhangyang Wang. Llm-autodiff: Auto-differentiate any llm workflow. *arXiv preprint arXiv:2501.16673*, 2025.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R Fung, Hao Peng, and Heng Ji. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*, 2023.

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

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. *arXiv preprint arXiv:2410.10762*, 2024a.

Peiyan Zhang, Haibo Jin, Leyang Hu, Xinnuo Li, Liying Kang, Man Luo, Yangqiu Song, and Haohan Wang. Revolve: Optimizing ai systems by tracking response evolution in textual optimization. *arXiv preprint arXiv:2412.03092*, 2024b.

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.

Boyuan Zheng, Michael Y Fatemi, Xiaolong Jin, Zora Zhiruo Wang, Apurva Gandhi, Yueqi Song, Yu Gu, Jayanth Srinivasa, Gaowen Liu, Graham Neubig, et al. Skillweaver: Web agents can self-improve by discovering and honing skills. *arXiv preprint arXiv:2504.07079*, 2025.

Huaixiu Steven Zheng, Swaroop Mishra, Xinyun Chen, Heng-Tze Cheng, Ed H Chi, Quoc V Le, and Denny Zhou. Take a step back: Evoking reasoning via abstraction in large language models. *arXiv preprint arXiv:2310.06117*, 2023.

Yuanhang Zheng, Peng Li, Wei Liu, Yang Liu, Jian Luan, and Bin Wang. Toolrerank: Adaptive and hierarchy-aware reranking for tool retrieval. *arXiv preprint arXiv:2403.06551*, 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.

CONTENTS

A Acknowledgements **17**

B Usage of LLMs **17**

C Related Work **17**

C.1 Automated Agentic Generation 17

C.2 Self-Evolve Methods 17

D Discussions **18**

E The Pipeline of ReCreate **19**

F Implementations of ReCreate **19**

G Detailed Information for Datasets **20**

H Limitations of Human-designed Scaffolds **20**

I Temperature Sensitivity **21**

J Cases in Motivation **21**

K Prompts of ReCreate-Agent **21**

L Case Study **22**

A ACKNOWLEDGEMENTS

This work was supported by Tencent CodeBuddy (https://www.codebuddy.ai/). We thank Yi Liu and the Tencent CodeBuddy team for their generous support and valuable assistance throughout this project.

B USAGE OF LLMS

Throughout the preparation of this manuscript, Large Language Models (LLMs) were utilized as a writing and editing tool. Specifically, we employed LLMs to improve the clarity and readability of the text, refine sentence structures, and correct grammatical errors. All final content, including the core scientific claims, experimental design, and conclusions, was conceived and written by us, and we take full responsibility for the final version of this paper.

C RELATED WORK

C.1 AUTOMATED AGENTIC GENERATION

Automated agent generation methods can be roughly divided into two lines: they either search agent from a predefined module pools or train a scaffold generator. ADAS (Hu et al., 2024) and AFlow (Zhang et al., 2024a) treat an agent as a program or workflow and use a meta-agent or MCTS-style search to iteratively propose, execute, and retain high-scoring designs in a hand-crafted search space. AgentSquare (Shang et al., 2024) abstracts agents into four interchangeable modules (planning, reasoning, tool use, memory), while AgentSwift (Li et al., 2025d) further enlarges the space by jointly searching workflow structure and functional components under a value-guided, uncertainty-aware hierarchical search. These search-based methods operate over increasingly rich design spaces but still rely on coarse scalar scores, without explicitly reasoning over the interaction experience when updating scaffold.

Another line of approaches learn an LLM policy that generates scaffolds. ScoreFlow (Wang et al., 2025) trains a workflow generator with a score-based preference objective, turning workflow optimization into learning from pairwise preferences induced by evaluation scores. RobustFlow (Xu et al., 2025) extends this view to robustness, optimizing generators so that workflows remain consistent across perturbed but semantically equivalent instructions. FlowReasoner (Gao et al., 2025) instead optimizes a query-level meta-agent with external execution feedback, using distillation plus reinforcement learning to improve the multi-agent systems it designs for each query.

ReCreate differs from both lines by taking full interaction experience (trajectories, logs, execution artifacts, verifier outputs) as input to a ReCreate-Agent that proposes targeted scaffold edits and enables experience-grounded agent optimization.

C.2 SELF-EVOLVE METHODS

Automated Tool Learning A prominent line of self-evolving agents enhances what an agent can do by autonomously expanding and maintaining its tool set. In embodied scenarios, long-horizon settings, Voyager (Wang et al., 2023a) continually explores and accumulates reusable skills, forming a growing library of executable behaviors. For more general-purpose tool creation, works such as Alita (Qiu et al., 2025), Live-SWE-Agent (Xia et al., 2025), ATLASS (Haque et al., 2025), CREATOR (Qian et al., 2023), SkillWeaver (Zheng et al., 2025), and CRAFT (Yuan et al., 2023) generate new functions or APIs from interaction experience and execution feedback, and reuse them across tasks. Beyond tool creation, an additional challenge is tool selection under a large inventory: methods such as ToolGen (Wang et al., 2024b), ToolRet (Shi et al., 2025), and ToolRerank (Zheng et al., 2024) retrieve, rerank, and invoke appropriate tools more reliably. Tool learning is also studied at the level of tool-use competence, e.g., Toolformer (Schick et al., 2023) and ToolLLM (Qin et al., 2023), which train or distill tool-use behaviors to improve tool calling accuracy and robustness.

Automated Context Learning Another core direction evolves what an agent sees in its context window, most commonly through prompt and instruction optimization. Early representative ap-

proaches treat prompt search as a discrete optimization problem: APE (Zhou et al., 2022) and MetaICL (Min et al., 2022) generate candidate prompts and select among them based on validation performance. More agentic variants explicitly plan over the prompt space, such as PromptAgent (Wang et al., 2023b), while population-based evolution is exemplified by PromptBreeder (Fernando et al., 2023). To stabilize and accelerate iterative prompt refinement, OPRO (Yang et al., 2023) and REVOLVE (Zhang et al., 2024b) use model-generated critiques and edits as optimization steps; similarly, ZERA (Yi et al., 2025) performs training-free evaluation–refinement with principle-based critiques and jointly refines system and user prompts (and task descriptions). For agentic settings with long and dynamic traces, SCOPE (Pei et al., 2025) treats prompt evolution as an online optimization problem and updates prompts from execution traces. Beyond single prompts, pipeline-level context learning is captured by DSPy (Khattab et al., 2023), and gradient-style textual optimization is explored in TextGrad (Yuksekgonul et al., 2024) and LLM-AutoDiff (Yin & Wang, 2025). Overall, these methods optimize the in-context specification (instructions, exemplars, and intermediate prompts) to steer agent behavior, and are largely orthogonal to expanding the toolset or updating long-term memory.

Automated Memory Evolving Memory evolving methods update what an agent retains and retrieves across episodes by deciding what to store, revise, and discard. One line focuses on structured long-term memory maintenance: SAGE (Liang et al., 2024) uses a forgetting-curve-inspired retention heuristic, while Mem0 (Chhikara et al., 2025) and MemInsight (Salama et al., 2025) use explicit update operations and semantic organization to support retrieval. Another line treats memory as an experience library by summarizing interaction history into reusable guidance: Expel (Zhao et al., 2024) distills trajectories into actionable rules, and Agent Workflow Memory (Wang et al., 2024d) stores workflow fragments that can be replayed for similar tasks. Memory evolution is also explored in strategic multi-agent settings, where self-play accumulates negotiation knowledge over time, e.g., Richelieu (Guan et al., 2024). Overall, these approaches treat memory as a persistent object that is continually updated and consulted to guide future decisions.

D Discussions

Why Agent-as-optimizer? While the concept of *LLM-as-optimizer* is widely recognized (Yuksekgonul et al., 2024), *Agent-as-optimizer* remains an emerging frontier. We identify *domain agent creation* as a quintessential scenario to exemplify this distinction. Fundamentally, *Agent-as-optimizer* represents a paradigm shift from *Optimization by Prompting* to *Optimization by Doing*. The former follows a linear `Reasoning` $\rightarrow$ `Text` process, passively generating prompts based on static context. Crucially, this approach remains labor-intensive, as it requires humans to manually curate and feed specific optimization targets into the model's context. In contrast, ReCreate establishes an Optimization by Doing loop: `Inspect` $\rightarrow$ `Reason` $\rightarrow$ `Optimize`. Here, ReCreate-Agent acts as an autonomous engineer: it actively retrieves specific trajectories, execution diffs or evaluation results to diagnose failure modes. This shifts the paradigm from reading static history to navigating full experience, enabling the precise localization of bug roots hidden in massive logs.

Comparison to Self-Evolve Recent self-evolving methods (Xia et al., 2025; Yang et al., 2023; Zhao et al., 2024) also leverage experience to refine pre-existing agents. ReCreate differs from these self-evolving methods in three aspects: (1) *Scope:* Instead of iteratively polishing a pre-defined scaffold, ReCreate can *create* an agent from scratch, which makes it applicable even in domains where no mature agent or hand-crafted workflow is available. (2) *Objective:* While prior work mainly optimizes for instance-level success (i.e., improving performance on the specific tasks encountered), ReCreate targets *domain-level generalization* by abstracting reusable improvements through hierarchical updates, thereby reducing overfitting to individual instances. (3) *Strategy:* Rather than relying primarily on coarse outcome signals (e.g., pass/fail or scalar rewards), ReCreate performs *fine-grained inspection* of execution traces and environment feedback, and turns concrete evidence into grounded scaffold edits. Empirically, even when initialized with a minimal seed scaffold, ReCreate outperforms these methods that start from fully-developed scaffolds (cf. Section 5).

Limitations The limitations of this work are twofold:

- First, ReCreate focuses on optimizing agent scaffolds at the textual and code levels, such as prompts, reasoning strategies, and tool scripts. It does not extend to infrastructure adaptations, such as customizing execution environments or systems, as these require heavy engineering that is distinct from the generalizable logic of agent creation.
- Second, the framework adopts a training-free framework and does not update the base model parameters. This work prioritizes a lightweight, plug-and-play approach to efficiently unlock the capabilities of frozen frontier LLMs without the high overhead of gradient-based optimization. While fine-tuning would internalize the discovered scaffold patterns, the data engineering required to curate high-quality training corpora is computationally expensive, which forms a promising direction.

E THE PIPELINE OF RECREATE

Algorithm 1 ReCreate for Domain Agent Creation

Require: LLM ϕ, dataset $\mathcal{D}$, domain init-info $\mathcal{I}$.
Ensure: final domain scaffold $\mathcal{A}_{\text{final}}$
1: $(\mathcal{D}_{\text{dev}}, \mathcal{D}_{\text{test}}) \leftarrow \textsc{Split}(\mathcal{D})$
2: $\mathcal{A} \leftarrow \textsc{Init}(\mathcal{I})$
3: **for** $n = 1$ to $N_{\max}$ **do**
4: $\quad \mathbb{H} \leftarrow \emptyset$, $\mathcal{B} \leftarrow \textsc{Sample}(\mathcal{D}_{\text{dev}})$
5: $\quad$ **for** each task $t \in \mathcal{B}$ **do**
6: $\quad\quad (\tau, r) \leftarrow \textsc{AgentRun}(\mathcal{A}, \phi, t)$
7: $\quad\quad (\Delta\mathcal{A}, \kappa) \leftarrow \textsc{Upd}\big(\tau, \sigma, \rho, r\big)$ [2]
8: $\quad\quad \mathbb{H} \leftarrow \mathbb{H} \cup \{(t, \Delta\mathcal{A}, \kappa)\}$
9: $\quad$ **end for**
10: $\quad \mathcal{A} \leftarrow \textsc{DomUpd}(\mathbb{H}, \mathcal{A}, \mathcal{I})$
11: **end for**
12: **return** $\mathcal{A}$ $\quad\quad$ ▷ report final metrics on $\mathcal{D}_{\text{test}}$

Algorithm 1 summarizes the complete workflow of ReCreate. First, the domain dataset $\mathcal{D}$ is split into development set $\mathcal{D}_{\text{dev}}$ and test set $\mathcal{D}_{\text{test}}$, where $\mathcal{D}_{\text{dev}}$ is used for agent scaffold creation and $\mathcal{D}_{\text{test}}$ is used for agent scaffold evaluation. The agent scaffold $\mathcal{A}$ is initialized through minimal initial information $\mathcal{I}$, including environment interfaces and necessary procedural. For each task in the sampled batch $\mathcal{B}$, ReCreate-Agent derives a local update proposal $\Delta\mathcal{A}$ from interaction experience and buffers it into $\mathbb{H}$ (Lines 5–8). These buffered local update proposals are aggregated to global update by ReCreate Agent (Line 10). Finally, the created agent scaffold $\mathcal{A}$ is evaluated on $\mathcal{D}_{\text{test}}$.

F IMPLEMENTATIONS OF RECREATE

We implement RECREATE as a parallel evolution pipeline that improves a shared scaffold (prompt + tools + memory) across iterations. For each batch, a task agent runs multiple instances in parallel under the same scaffold inside Docker, and we record the trajectory, submitted artifacts (e.g., patches/files), and the evaluator report. A per-instance meta-agent then inspects these artifacts and produces a concrete update (a scaffold diff, a new tool, or a memory entry), and a synthesis meta-agent merges updates from the whole batch into the next scaffold version while removing instance-specific changes. To support five benchmarks with one codebase, we use a `DomainAdapter` that only specifies how to load data, run the agent, and evaluate, while the evolution logic stays identical across domains. The entire process is logged as versioned folders (`global_v000`, `global_v001`, ...) with diffs and statistics, enabling reproducible comparisons to the baseline scaffold.

Following *Agent Skills* design (Han Lee, 2025), we package each tool as a self-contained directory with a `SKILL.md` (YAML `name`/`description` for discovery) plus executable scripts and optional resources. The agent only preloads lightweight metadata, and lazily reads the full instructions or runs scripts on demand, enabling many tools without saturating the context window. In our system, RECREATE-Agent creates and updates these skill-style tool folders from execution evidence

[2] Here, σ refers to execution results and ρ refers to evaluation results for short.

(trajectories, artifacts, and evaluator logs), so improvements are reusable and traceable to concrete failures or successful patterns.

As for memory, we implement two complementary components: a *memory mechanism* and a *static memory bank*. The memory mechanism specifies when the task agent should write new memories and when it should retrieve existing memories (e.g., after repeated failures or before critical steps), making memory usage a controlled part of the workflow. The static memory bank stores reusable experience distilled by RECREATE-Agent (e.g., common failure modes, repair heuristics, and tool-usage tips), which can be searched and reused across future instances.

G DETAILED INFORMATION FOR DATASETS

Our experiments are conducted on five benchmarks (counts in Table 3); we briefly introduce each benchmark below.

SWE We use SWE-BENCH VERIFIED, where each instance corresponds to a real GitHub issue in a target repository. The agent must produce a code patch that is applied and validated in an isolated environment; success is determined by passing the benchmark's tests after patch application. Django and SymPy are the two repositories in SWE-bench Verified that cover the largest number of tasks. In our experiments, we sample 20 tasks from each repository as the development set.

DA-Code This benchmark targets data-science programming workflows, covering common routines such as data transformation/cleaning, classical ML modeling, and statistical analysis. Tasks emphasize producing executable code under practical workflow constraints. In our experiments, we sample 20 tasks from each subset as the development set.

DS-1000 DS-1000 is a data-science code generation benchmark built from real-world questions, paired with automatic evaluation via executable checks. We report results on three core library subsets that represent array computing (NumPy), tabular manipulation (Pandas), and visualization (Matplotlib). In our experiments, we sample 20 tasks from each repository as the development set.

Math AIME24 and AIME25 (Li et al., 2024) contain problems in AIME exams of the corresponding years and evaluate competitive-math reasoning with short final answers. We additionally use MATH500 (Hendrycks et al., 2021), a 500-problem subset of the MATH dataset, and break it down into topic subsets (Algebra, Number Theory, and Counting & Probability) to study subject-specific behavior. In our experiments, we use AIME24 as the development set and evaluate on the MATH500 topic subsets as test sets.

AppWorld APPWORLD is an interactive agent benchmark with a suite of apps and executable APIs. Tasks require multi-step decision making and tool use in a controlled environment. We follow its provided split into a dev set and two evaluation partitions (Normal and Challenge), where the latter typically poses harder or more adversarial scenarios. In our experiments, we sample 30 instances from the dev split as the development set, and evaluate on the Normal and Challenge splits as test sets.

SWE		**DA-Code**			**DS-1000**		
Django	**Sympy**	**Data Wrangling**	**Machine Learning**	**Statistical Analysis**	**NumPy**	**Pandas**	**Matplotlib**
231	75	100	100	78	220	291	155
Math					**AppWorld**		
AIME24	**AIME25**	**Algebra**	**Number Theory**	**Counting&Probability**	**dev**	**Normal**	**Challenge**
30	30	124	62	38	57	168	417

Table 3: The counts for dataset used in our experiments.

H LIMITATIONS OF HUMAN-DESIGNED SCAFFOLDS

In this section, we argue that human-designed scaffolds are not only labor-intensive to build, but also *cap performance*.

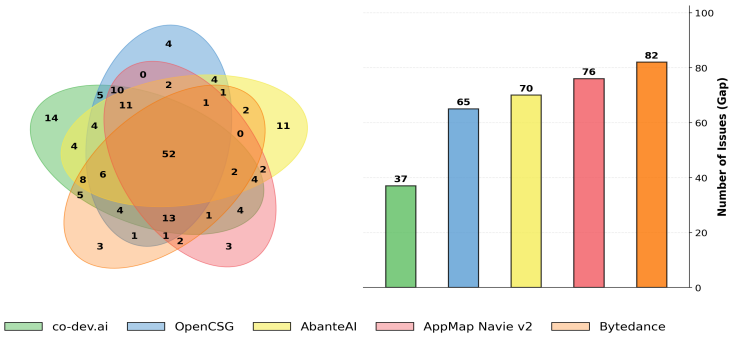


Figure 11: Scaffolds gate what a base model can do

Current agent scaffolds gate what a base model can do. We ask a simple question: *for a fixed base model, how much can the final success depend on the surrounding scaffold?* To isolate the effect of scaffolds, we compare five top-performing open-sourced agents on SWE-bench Lite (300 issues) that all use the same LLM (gpt-4o) but differ in prompts, workflows, and tool setups. Figure 11 (left) shows that their solved issues overlap only partially: the union reaches 184 issues, yet the best single scaffold solves 147. This leaves a *scaffold-fixable headroom* of $184 - 147 = 37$ issues (20% of the union): these issues are solved by the *same* model under some scaffold, but are missed by the strongest human-designed scaffold in this pool. The small intersection (only 52 issues solved by all five) further suggests that scaffolds do not merely guide outputs: they also change the agent's search behavior (what to inspect, which checks to run, how to iterate), effectively routing the model to different solvable regions.

The right panel quantifies this effect by counting, for each scaffold S, how many issues in the union U it fails to solve (i.e., issues solved by at least one other scaffold). Even for the best scaffold, 37 union issues are missed; for other scaffolds, the gap is much larger (65–82 issues). In other words, a substantial portion of what looks like "model limitation" under one scaffold is actually *recoverable* under another scaffold with the same base model. This exposes a key weakness of human-designed scaffolds: they are strong but incomplete samples from a vast design space, and they leave significant performance untapped. These observations motivate us to perform agent scaffold optimization rather than one-off manual engineering. Moreover, as AI continues to surpass human intelligence, it can achieve a higher upper bound in creating agents.

I Temperature Sensitivity

We test the performance of ReCreate with different sampling temperature t of ReCreate Agent (we use `claude-4.5-opus`), as shown in Table 4. It can be observed that ReCreate maintains comparable performance across different sampling temperatures. This suggests that state-of-the-art models have stably approached the capability to create domain agents.

J Cases in Motivation

In this section we present cases that show experience can be important for agent scaffold update, shown in Figure 12, 13, 14.

K Prompts of ReCreate-Agent

The ReCreate-Agent operates as a agent-optimizer. Its system prompt is designed to guide it through the full loop of inspection, diagnosis, and scaffold evolution. Below we present the core components of the prompt (administrative instructions and specific file paths are omitted for brevity), shown in Figure 15.

Domain	Dataset	$t = 0.0$	$t = 0.5$	$t = 1.0$
SWE	*Django*	60.66	61.14	63.51
	Sympy	61.82	58.18	60.00
DS	*DW*	51.94	47.79	50.55
	ML	40.88	51.16	45.78
	SA	23.00	20.33	27.33
	Numpy	77.00	78.00	81.50
	Pandas	67.53	65.68	66.79
	Matplotlib	85.19	78.52	74.07
Math	*Algebra*	95.16	94.35	93.55
	NT	100.00	100.00	100.00
	C&P	100.00	100.00	97.37
Digital	*Normal*	52.98	51.79	55.36
	Challenge	37.41	37.65	38.85
Average		65.66	64.97	65.74

Table 4: ReCreate performance with different temperature.

Motivation 1: Rules (Log: ML/q_39)

Context: Task requires training a classifier on `breast_cancer.csv`.

[STEP 3] Model Training & Verification

>**THOUGHT:** ...missing values are handled. Features are stored in `X` and labels in `y`. I will now fit a Random Forest and check the accuracy to ensure the model is learning correctly before prediction.

>**ACTION:**

```
from sklearn.ensemble import RandomForestClassifier
...
# Initialize model with default parameters
clf = RandomForestClassifier(random_state=42)
clf.fit(X, y)
...
# Evaluate performance
# <--- [FATAL FLAW] Evaluating on training data
train_score = clf.score(X, y)
print(f"Current Accuracy: {train_score:.4f}")
```

>**OBSERVATION:** `Current Accuracy: 0.9420`

>**THOUGHT:** The accuracy is 94.2%, which is excellent. The model is robust. I will proceed to load `test.csv`...

[Motivation] The trace reveals an overfitting hallucination. The agent sees a near-perfect score (0.942) and stops optimizing, unaware that the score is biased without a hold-out set. One naive idea is to inject rule: "CONSTRAINT: You need to use `train_test_split` for internal evaluation."

Figure 12: A case study for adding rules.

The Synthesis prompt of Meta-Agent aggregates per-instance scaffold edits into one unified scaffold, shown in Figure 16. We omit administrative instructions and absolute paths for brevity.

L Case Study

We take the initialization and the final results of ReCreate on Django for demonstration. To ensure a realistic initialization, ReCreate starts with a minimal seed scaffold, shown in Figure 17. Driven by interaction experience, the agent iteratively evolves this seed into a specialized domain system. The final output includes rigorous prompt templates for system constraints, workflows, and memory

Motivation 2: Tools (Log: SA/q_41)

Context: Agent generated `solution.py` and needs to verify it.

[STEP 4] Multi-step Verification

>**THOUGHT:** ...code is written. Before submission, I must ensure the file is not empty, has correct syntax, and is actually executable. I will check these one by one.

>**ACTION:** ...

```
# 1. Check if file exists and is not empty
ls -lh solution.py && cat solution.py | head -n 3
```

>**OBSERVATION:** `-rw-r--r-- 1 root root 402B ...`
`def solve(nums): ...`

>**ACTION:**

```
# 2. Check for syntax errors
python3 -m py_compile solution.py
```

>**OBSERVATION:** (No output, implies syntax OK)

>**ACTION:**

```
# 3. Check execution logic
# <--- [INEFFICIENT] Wasting 3 turns on routine checks
python3 -c "from solution import solve; print(solve([1, 2]))"
```

>**OBSERVATION:** `3`

[Motivation] The trace reveals a tedious verification loop. The agent spends multiple turns performing generic sanity checks (size, syntax, execution). An effective idea is to create a unified tool: `full_check(file_path)` that automates this entire validation pipeline in one call.

Figure 13: A case study for adding tools.

Motivation 3: Workflows (Log: SWE-bench/django-45)

Context: Fixed bug in `django/utils/text.py`. Tests passed.

[STEP 12] Submission

>**THOUGHT:** ...regression tests are passing. I will now commit the changes to the repository and submit the final patch.

>**ACTION:**

```
git add django/utils/text.py
# <--- [FATAL FLAW] Committing clears the Index
git commit -m "Fix regression in slugify"
```

>**OBSERVATION:**

```
[main 8a3c1b] Fix regression in slugify
 1 file changed, 4 insertions(+), 1 deletion(-)
```

>**ACTION:**

```
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
```

>**SYSTEM EVALUATION:** `>> Patch Content:` (empty string)
`>> Result: FAILED (Patch size 0 bytes)`

[Motivation] The trace reveals a "clean working tree" paradox. The agent followed standard Git practice, but the harness requires staged changes (Index) for patch extraction. A necessary update is to enforce a workflow: "CRITICAL: RUN `git diff --cached` before finish."

Figure 14: A case study for enforcing workflows.

interfaces (Figures 18, 19, 20). Additionally, ReCreate crystallizes its experience into actionable memories (Figure 21) and custom tools (Figure 22) to resolve specific domain challenges.

ReCreate-Agent System Prompt

Role Definition You are an agent that creates and evolves other AI agents by editing their scaffolds (prompts, workflows, tools, and memory mechanisms).
Mission Analyze agent execution trajectories, understand success and failure patterns, inspect the agent's environment, and evolve the agent's scaffold and tools so that it performs better on future tasks in the same domain.

Core Philosophy You are discovering **generalizable principles**. Think like a teacher improving a student:

- **Learn from SUCCESS**: Extract winning strategies and encode them as tools.
- **Learn from FAILURE**: Diagnose issues and add safeguards.

The Five Components You Control

1. `system_template`: Agent's identity, core knowledge, principles.
2. `instance_template`: Problem-solving workflow, step-by-step guidance.
3. `memory_template`: Agent's memory read/write strategy.
4. `agent_tools/`: Reusable automation scripts and helper commands.
5. `agent_memory/`: Historical lessons & patterns (static content).

Thinking Framework When analyzing a trajectory, focus on:

- **Patterns**: What behaviors systematically help or hinder progress?
- **Root Cause**: Is this a knowledge gap, strategy gap, or tool gap?
- **Intervention**: What targeted change would steer future trajectories better?
- **Tool Opportunities**: Ask "What repetitive operation could be automated?"

Available Tools (Action Space)

- **Trajectory Analysis**:
  - `read_trajectory.py summary`: Get overview of the run.
  - `read_trajectory.py failures`: List all errors and their context.
  - `read_trajectory.py context --step N`: Inspect specific reasoning steps.
  - .....
- **Environment Inspection**:
  - `inspect_in_docker.py --command "ls -R"`: View the actual file system state.
  - .....
- **Scaffold Editing**:
  - `scaffold_editor.py str_replace`: Modify prompts/templates.
  - `tool_manager.sh create`: Create new Python tools for the agent.
  - `memory_manager.py add`: Inject static knowledge/lessons.
  - .....

Recommanded Workflow

- **Check Submission**: Verify if the patch is empty or valid.
- **Read Trajectory**: Understand what the agent did step by step.
- **Analyze Causes**: Why did the reasoning or tool usage break down?
- **Decide Intervention**: Create a tool (Preferred) or Update Scaffold.
- **Execute & Verify**: Apply changes and confirm they match intent.

Figure 15: Main prompt for ReCreate-Agent.

Synthesis Meta-Agent Prompt (Batch Synthesis)

Role
You are the **Synthesis Meta-Agent**. Review all proposed scaffold modifications from a batch and produce a unified *global* scaffold that generalizes across the domain.
Context
Batch {{ batch_idx }} ran {{ num_instances }} instances from the same base scaffold (global_v{{ batch_idx }}). {{ num_modifications }} instances proposed modifications.

Your Task

1. Review each proposal (summary + diff; open full files when needed).
2. Extract shared patterns and generalizable improvements.
3. Resolve conflicts and synthesize a single unified scaffold.

Where to Inspect Full Proposals

- batch_modifications/<instance_id>/diff.txt
- batch_modifications/<instance_id>/summary.md
- batch_modifications/<instance_id>/scaffold.yaml

Decision Guidelines (Prefer Success)

- **Successful instances:** prioritize reusable tools, stable workflow improvements, and concise rules that clearly contributed to success.
- **Failed instances:** include only low-risk safeguards that address a *general* failure mode; avoid brittle or overly restrictive rules.

Conflict Resolution
Prefer changes supported by multiple instances; otherwise choose the simpler, more general formulation. When uncertain, keep the original rule.

Required Outputs

- Update current/scaffold.yaml with the unified scaffold.
- Write current/synthesis_summary.md (included vs. rejected changes, plus conflict resolutions).
- If useful tools/memories exist, copy/add them into current/.

Completion
Output: echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT

Reminder
Optimize for **generalization across the domain**, not for the specific instances in this batch.

Figure 16: Batch synthesis prompt for aggregating instance-level scaffold edits into a unified global scaffold.

Minimal Seed System Template

You are an expert software engineer solving GitHub issues.

Response Format (CRITICAL)
You MUST respond with EXACTLY this format every turn:

```
THOUGHT: <analysis>
```bash
<ONE command>
```
```

NEVER output multiple code blocks. NEVER skip the THOUGHT section.

Rules

- ONE command per response. Do NOT try to do everything at once.
- No vim/nano. Use `sed -i` or `python3 -c` for edits.
- NEVER use heredoc (`<<EOF`) — it causes truncation errors.
- NEVER read or modify test files.

Figure 17: The Minimal Seed System Template in Django.

Created System Template

```
You are an expert software engineer solving GitHub issues in real open-source projects.
## Response Format (CRITICAL)
You MUST respond with EXACTLY this format every turn:
THOUGHT: <your analysis in a single paragraph>
```bash
<exactly ONE command>
```
NEVER output multiple code blocks. NEVER skip the THOUGHT section.
Rules
• ONE command per response. Work step-by-step, not all at once.
• No interactive tools: NO vim, nano, less, or interactive prompts
• NEVER use heredoc (<<EOF) - it causes truncation errors. Use sed or python -c instead.
• Directory changes don't persist. Use: cd /path && command
• Commands timeout after 100 seconds. Use head/tail/grep for large outputs.
Anti-Cheat Rules (MUST FOLLOW)
• NEVER read, modify, or look at any test files (test_*.py, *_test.py, tests/, testing/)
• NEVER try to discover or reverse-engineer test content
• NEVER modify or delete .git directories
• Focus ONLY on the issue description to understand what needs fixing
• Your fix must be based on understanding the problem, not the tests
File Editing Methods
RECOMMENDED: Use the file_editor tool (handles quoting automatically):
python3 /workspace/agent_tools/utils/file_editor/main.py replace <file> --old "old text"
--new "new text"
python3 /workspace/agent_tools/utils/file_editor/main.py show <file> --start 50 --end 100
python3 /workspace/agent_tools/utils/file_editor/main.py search <file> --pattern "regex"
For simple changes, use sed:
sed -i 's/old_text/new_text/g' filename.py
For complex edits with special characters, use python with raw strings:
python3 -c 'from pathlib import Path; p=Path("file.py"); s=p.read_text();
s=s.replace("old", "new"); p.write_text(s)'
Running Python Scripts
For complex Python scripts (multi-line, classes, if statements), use printf to write to
a file:
printf 'line1\nline2\nprint(result)' > /tmp/test.py && python3 /tmp/test.py
Codebase Location
The repository is at /testbed/. Always work from there.
```

Figure 18: The System Template created from Django experience.

Created Instance Template

```
## GitHub Issue to Solve
{{task}}
## Workflow (Follow Step-by-Step)
1. UNDERSTAND: Read the issue carefully. What is the expected vs actual behavior?
2. LOCATE: Find relevant files using:
 • find /testbed -type f -name "*.py" | grep -E "keyword" | head -20
 • grep -r "function_name" /testbed --include="*.py" -l | head -20
If grep output is truncated, NARROW your search to the relevant subdirectory:
 • grep -r "pattern" /testbed/specific/module/ --include="*.py"
After finding one occurrence, check if the same pattern exists in related files.
3. ANALYZE: Read and understand the code:
 • cat /testbed/path/to/file.py | head -100
 • grep -n "pattern" /testbed/path/to/file.py
4. IMPLEMENT: Make targeted changes using sed or python -c (NO heredoc!)
5. VERIFY: Check your changes:
 • git diff to see what you changed
 • Quick sanity check: python3 -c "import module_you_changed" to verify no syntax errors
 • TEST THE ACTUAL BEHAVIOR: Run a quick test with the specific inputs from the issue
   Example: python3 -c "from module import func; print(func(problematic_input))"
 • Test edge cases: boundary values, empty inputs, unusual but valid inputs
 • For multi-stage operations (ORM queries, serialization, caching): trace the full data
   flow and verify your fix handles ALL stages, not just the first one you found
6. VALIDATE: Before submitting, ask yourself:
 • Does my fix address the root cause described in the issue
 • Did I test the specific scenario mentioned in the issue?
 • Could my change break other functionality?
7. SUBMIT: When confident your fix is complete:
git add -A && git diff --cached && echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT
## Important Reminders
• Work incrementally: one change at a time
• Read code before modifying it
• Verify your changes with git diff before submitting
• NEVER touch test files - your fix will be evaluated against hidden tests
```

Figure 19: The Instance Template created from Django experience.

Created Memory Template

```
## Memory System
You can read and write memories to learn from past experiences.
Read memories (search for relevant lessons):
python3 /workspace/agent_memory/search_memory.py "keyword"
Write memories (save what you learned):
python3 /workspace/agent_memory/write_memory.py --title "Short title" --content "What you
learned" --tags "tag1,tag2"
When to read memories:
• At the start of a task, search for similar issue types
• When encountering an error, search for that error message
• When stuck, search for the technology/library involved
When to write memories:
• After solving a tricky bug, save the key insight
• When you discover a useful pattern or gotcha
• After finding a non-obvious solution approach
```

Figure 20: The Memory Template created from Django experience.

[Generated Memories] agent_memory.yaml

```yaml
memories:
- content: "When using python3 -c for file editing with regex patterns, avoid complex
  escaping. Instead use raw strings (r'pattern') and multi-line format. Better yet, use the
   file_editor tool at /workspace/agent_tools/utils/file_editor/main.py for safe search-and
  -replace operations."
  created: '2026-01-02'
  id: mem_001
  tags: [python, regex, editing, escaping]
  title: Regex escaping in python -c

- content: "For Django issues, always run the specific FAIL_TO_PASS tests mentioned in
  expected_tests.txt before submitting. Use: python tests/runtests.py <module>.<Class>.<
  method>. Don't assume a fix works just because it compiles - the actual test may reveal
  the fix is incomplete."
  created: '2026-01-02'
  id: mem_002
  tags: [django, testing, swe-bench, verification]
  title: Run FAIL_TO_PASS tests before submitting Django fixes

- content: "When replacing entire Python methods/functions, avoid complex regex patterns
  that require escaping. Instead, use str.find() to locate the method start (e.g., 'def
  method_name(') and find the next method at the same indentation level to determine the
  end. This approach is more reliable."
  created: '2026-01-02'
  id: mem_003
  tags: [python, editing, methods, refactoring]
  title: "Method replacement: use find() not regex"
```

Figure 21: A snapshot of the static memory accumulated by ReCreate.

[Created Tool] replace_method.py

```python
#!/usr/bin/env python3
"""
Replace Method Tool - Replace an entire Python method/function by name

Usage:
    python3 replace_method/main.py <file> --method "name" --new-body "def name(): pass"
    python3 replace_method/main.py <file> --method "name" --new-body-file /path/to/impl.py
"""

import argparse, re, sys
from pathlib import Path

def find_method_boundaries(content: str, method_name: str) -> tuple:
    """Find the start and end positions of a method based on indentation."""
    lines = content.split('\n')

    # Pattern to match method/function definition strictly
    method_pattern = re.compile(rf'^(\s*)(def\s+{re.escape(method_name)}\s*\()')

    # ... [Logic to find start_line and base_indent] ...

    # Find the end of the method by checking indentation levels
    end_line = len(lines)
    for i in range(start_line + 1, len(lines)):
        line = lines[i]
        stripped = line.lstrip()

        # Skip empty lines and comments
        if not stripped or stripped.startswith('#'): continue

        current_indent = len(line) - len(stripped)

        # Stop if we find a line at same or lower indentation level
        if current_indent <= base_indent:
            end_line = i
            break

    return start_pos, end_pos, base_indent

def replace_method(filepath: str, method_name: str, new_body: str) -> None:
    """Replace a method/function with new implementation."""
    path = Path(filepath)
    content = path.read_text()

    # 1. Locate the method in the source file
    start_pos, end_pos, indent = find_method_boundaries(content, method_name)

    if start_pos is None:
        print(f"ERROR: Method '{method_name}' not found")
        sys.exit(1)

    # 2. Align the new body to the correct indentation level
    new_lines = new_body.split('\n')
    indented_lines = []
    for line in new_lines:
        if line.strip() and not line.startswith(' ' * indent):
            line = ' ' * indent + line.lstrip()
        indented_lines.append(line)
    new_body_indented = '\n'.join(indented_lines)

    # 3. Perform the string replacement
    new_content = content[:start_pos] + new_body_indented + content[end_pos:]
    path.write_text(new_content)
    print(f"SUCCESS: Replaced method '{method_name}' in {filepath}")

def main():
    # ... [Argparse setup omitted for brevity] ...
    if args.show_only:
        replace_method(args.file, args.method, "", show_only=True)
    elif args.new_body:
        replace_method(args.file, args.method, args.new_body)

if __name__ == "__main__":
    main()
```

Figure 22: A tool created from Django experience.