

REVERE: Reflective Evolving Research Engineer

Balaji Dinesh Gangireddi[†] Aniketh Garikaparthi[†]
 Manasi Patwardhan[†] Arman Cohan*

[†]TCS Research *Yale University

dinesh.gangireddi@tcs.com

Abstract

Existing prompt-optimization techniques rely on local signals, causing poor generalization across tasks. In addition, they also rely on weak update mechanisms, such as full-prompt rewrites or unstructured merges, which cause knowledge loss and unstable adaptation. These limitations are magnified in research-coding workflows, which involve heterogeneous repositories and weak feedback, limiting abstraction and learning across tasks. We introduce Reflective Evolving Research Engineer (REVERE), a lightweight, self-adapting agent framework that learns from a Global Training Context, distills recurring cross-repository failure modes into reusable heuristics, and applies targeted, code-based edits to agent prompts. REVERE¹ is evaluated across settings ranging from long-horizon to single-shot benchmarks, and improves over prior expert-crafted instructions by 4.50% on SUPER, 1.3% on ResearchCodeBench, and 4.89% on ScienceAgentBench. It does so at nearly 10× lower cost and 2.7× faster adaptation than existing prompt-optimization baselines, demonstrating that self-adapting agents with continual learning and global memory consolidation can meaningfully evolve their capabilities over time.

1 Introduction

While recent progress of Large language models (LLMs) on short-horizon well-specified coding tasks is promising (Yang et al., 2024; White et al., 2025; Gauthier, 2024), reliability degrades substantially in research-code reproduction (Starace et al., 2025; Xiang et al., 2025; Garikaparthi et al., 2026; Bogin et al., 2024; Hua et al., 2025), due to fundamentally different demands on agents. These include coordinating long-horizon tasks under weak and delayed feedback, inferring tacit assumptions, and accumulating procedural knowledge across heterogeneous research frameworks (Trehan & Chopra, 2026; Peng & Wang, 2025; Wang et al., 2026). Prior agentic systems (Starace et al., 2025; Wang et al., 2025) targeting research reproducibility, typically rely on static prompts. More complex systems such as (Seo et al., 2025; Lin et al., 2025) further decompose high-level tasks through multi-agent workflows; while this can improve reliability, they still operate within fixed contexts and predefined strategies. As a result, these systems struggle to adapt to the evolving conventions and diverse open-ended nature of research-coding tasks.

Recent works on self-refinement (Shinn et al., 2023; Madaan et al., 2023; Majumder et al., 2024) improve reasoning through iterative feedback but remain instance-specific, motivating prompt-level and experience-based adaptation methods (Agrawal et al., 2025; Opsahl-Ong et al., 2024; Zhao et al., 2024). Yet these adaptation methods rely primarily on heuristic prompt sampling and local evaluation signals, which work well in short-horizon settings but cause overfitting to recent outcomes. This local-signal problem persists even in recent methods such as Suzgun et al. (2025); Zhang et al. (2025c). Both operate over a limited window of past experiences, without a holistic view across the experiences accumulated

¹Code:  <https://revere-acceleron.github.io/revere/>

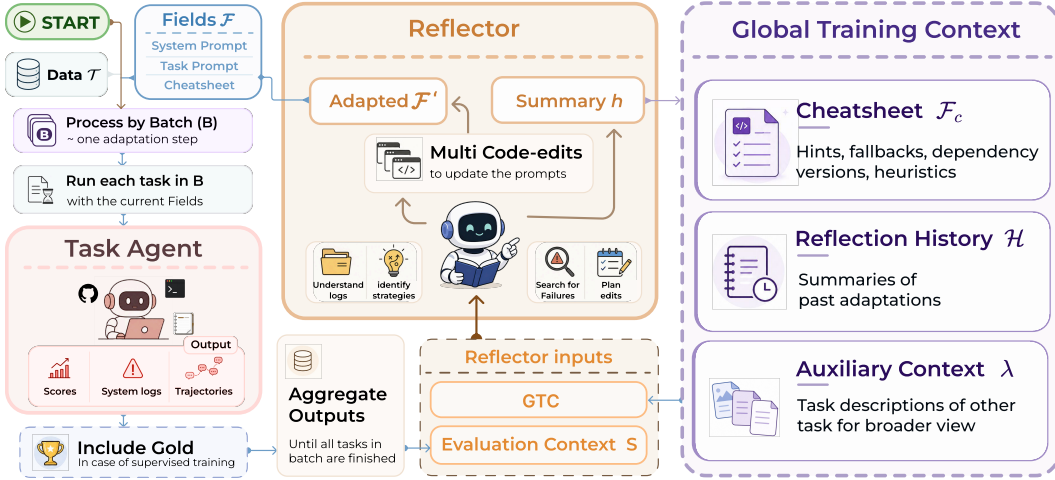


Figure 1: **The REVERE Framework** A given task agent’s prompts, represented as editable Fields (System, Task, Cheatsheet), are optimized by a Reflector agent using a code-edit tool and guided by a Global Training Context and local, per-batch Evaluation Context.

from all prior tasks; ACE (Zhang et al., 2025c), in particular, retains only a single task’s experience at a time. In long-horizon settings, this is especially limiting: a bounded window can’t hold enough past experience to tell which strategies actually generalize. But keeping every experience without consolidating it into context isn’t practical either, and as a result, these methods can still converge to local optima (Shi et al., 2025) rather than build genuinely generalizable strategies. A second, distinct issue lies in the edit mechanisms of these frameworks themselves: they fall into two failure modes. Full-prompt regeneration (Agrawal et al., 2025; Suzgun et al., 2025) risks semantic drift and knowledge loss as prompts are rewritten over and over, while append-only accumulation (Zhang et al., 2025c) avoids drift but grows unbounded and risks conflicting heuristics without reconsideration. Structured editing methods (Schnabel & Neville, 2024) partially address this tension but typically require more complex implementations. This motivates an agent that learns from its own trajectories over time, applying targeted, non-destructive updates to learnings rather than full rewrites.

To address these gaps, we introduce REVERE, a training-free framework for building self-adapting agents tailored to research-coding workflows. REVERE adopts a simple, unified design built around three core components: (1) *prompt adaptation over configurable fields*, which defines fields to be optimized and adapts them based on observed failure modes and evaluation feedback; (2) a *Global Training Context* (GTC), which preserves and aggregates experience across tasks and adaptations; and (3) *targeted code-based updates* via a Reflector module, which applies structured edits to prompts and other optimizable fields. Together, these components allow REVERE to progressively refine its behavior, reuse prior strategies, and update its reasoning without overfitting to specific tasks. Our work makes the following contributions:

- We formulate research code reproduction as a test-time adaptation problem for LLM agents, highlighting concrete failure modes specific to research repositories.
- We demonstrate that REVERE improves over expert-crafted instructions across three research-coding benchmarks in a fully unsupervised online setting, and further improves when given ground-truth signals, achieving gains of up to 4.50% on SUPER (Bogin et al., 2024), 1.3% on ResearchCodeBench (Hua et al., 2025), and 4.89% on ScienceAgentBench (Chen et al., 2025).
- We demonstrate that REVERE achieves performance gains over prior prompt adaptation baselines like GEPA Agrawal et al. (2025) and ACE Zhang et al. (2025c) with substantially lower overhead. Adaptation cost is nearly $10\times$ lower than ACE and GEPA, wall-clock time is up to $2.7\times$ faster.

2 Related Work

Research-Coding benchmarks and approaches: LLMs are increasingly evaluated across ML engineering benchmarks (Chan et al., 2025; Huang et al., 2024), end-to-end research workflows (Panigrahi et al., 2026), and the broader research experimentation lifecycle (Huang et al., 2025; Edwards et al., 2025; Starace et al., 2025; Kon et al., 2025; Zhao et al., 2025). Benchmarks targeting research-code reproducibility (Bogin et al., 2024; Tian et al., 2024; Xiang et al., 2025; Siegel et al., 2024) reveal persistent performance gaps even in multi-agent and search-based systems (Starace et al., 2025; Seo et al., 2025; Schmidgall et al., 2025; Jiang et al., 2025; Si et al., 2026), pointing to the need for agents capable of self-reflection rather than static, handcrafted workflows. While EXP-Bench (Kon et al., 2025) and PaperBench (Starace et al., 2025) target experiment generation and end-to-end research replication respectively, we focus on SUPER (Bogin et al., 2024), ResearchCodeBench (Hua et al., 2025), and ScienceAgentBench (Chen et al., 2025), which capture comparable research-coding challenges with more tractable computational cost and reliable, execution-based evaluation.

Prompt optimization and self evolution techniques: Classical prompt optimization treats prompts as tunable parameters via reinforcement learning, gradient-free, or heuristic search (Khattab et al., 2023). Newer approaches introduce reflective and evolutionary strategies (Agrawal et al., 2025; Opsahl-Ong et al., 2024), while runtime-adaptive agents modify their scaffolds on the fly (Zhang et al., 2025a; Hu et al., 2025). Task-level methods (Hu et al., 2024; Zhang et al., 2025b) improve instance performance but fail to transfer across tasks. Continual approaches such as CLIN (Majumder et al., 2024) rely on episodic memory regeneration, while DC (Suzgun et al., 2025) and ACE (Zhang et al., 2025c) apply structured edits to an evolving memory; however, these methods remain tied to local signals and short-horizon, densely supervised settings. Research coding breaks both assumptions. Tasks are long-horizon, supervision is sparse, and recurring failure patterns emerge only across many executions. REVERE addresses this by aggregating experience globally and applying code-based edits directly to agent prompts, enabling cumulative adaptation without semantic drift.

3 Reflective Evolving Research Engineer (REVERE)

3.1 Setup

We formalize the adaptation problem over three editable context fields that govern agent behavior: $\mathcal{F} = \{\mathcal{F}_s, \mathcal{F}_x, \mathcal{F}_c\}$, where $\mathcal{F}_s$ is the *system prompt* (global behavior and rules), $\mathcal{F}_x$ is the *task prompt* (task-specific instructions instantiated at runtime), and $\mathcal{F}_c$ is the *cheatsheet* (a persistent memory, initialized empty, that accumulates reusable strategies and tips). More generally, $\mathcal{F}$ can comprise an arbitrary number of user-defined editable fields to suit different agent configurations. Together, these fields parameterize agent behavior without modifying model weights. Given a dataset of tasks $\mathcal{T} = \{(x, o)\}$, where x is a task description and o contains target metrics and optional gold outputs, the agent Φ produces an output $\hat{o} = \Phi(x; \mathcal{F})$ that is evaluated by a metric function μ to yield a scalar score $s = \mu(\hat{o}, o)$. Adaptation seeks optimal fields:

$$\mathcal{F}^* = \arg \max_{\mathcal{F}} \mathbb{E}_{(x, o) \sim \mathcal{T}} [\mu(\Phi(x; \mathcal{F}), o)]. \quad (1)$$

3.2 Method Overview

REVERE improves a coding agent through an iterative adaptation loop (Figure 1), progressively editing the three fields of $\mathcal{F}$ based on execution feedback. The agent runs on tasks in batches which provides the Reflector with diverse execution signals, avoiding the diminished feedback that arises when reflecting on all tasks at once without intermediate updates, and amortizing the cost of reflection. A key component of this loop is the information provided to the Reflector for decision-making: after each batch, it receives a local evaluation signal $S = \{(x, s, \hat{o})\}$, referred to as the *Evaluation Context*, which summarizes

Algorithm 1 REVERE: Adaptation Process

Require: Agentic system Φ , Training data $\mathcal{T} = \{(\text{Task description } x, \text{Ground truth } o)\}$,
 1: Metric function μ , Context fields $\mathcal{F}$
Ensure: Adapted fields $\mathcal{F}$
 2: **for** each epoch **do**
 3: Shuffle $\mathcal{T}$ and form batches $\{B\}$
 $\triangleright$ Initialize Reflection history
 4: $\mathcal{H} \leftarrow \emptyset$
 5: **for** each batch B **do**
 $\triangleright$ Initialize Evaluation Context
 6: $S \leftarrow \emptyset$
 7: **for all** $(x, o) \in B$ **do**
 8: $\hat{o} \leftarrow \Phi(x, \mathcal{F})$
 9: $s \leftarrow \mu(\hat{o}, o)$
 10: **if** use gold **then**
 11: $S \leftarrow S \cup \{(x, s, o, \hat{o})\}$
 12: **else**
 13: $S \leftarrow S \cup \{(x, s, \hat{o})\}$
 14: **end if**
 15: **end for**
 16: $\lambda \subseteq \{x \mid (x, o) \in \mathcal{T} - B\}$
 $\triangleright$ Construct Auxiliary context
 17: $(\mathcal{F}', h) \leftarrow \text{REFLECTOR}(\mathcal{F}, S, \mathcal{H}, \lambda)$
 18: $\mathcal{H} \leftarrow \mathcal{H} \cup \{h\}; \mathcal{F} \leftarrow \mathcal{F}'$
 19: **end for**
 20: **end for**
 21: **Output:** Adapted Fields $\mathcal{F}$

Algorithm 2 REFLECTOR Agent Module

Require:
 Fields $\mathcal{F} = \{\mathcal{F}_s, \mathcal{F}_x, \mathcal{F}_c\}$, LLM $\mathcal{L}$,
 History buffer $\mathcal{H}$
 Evaluation context S
Action Space: action $a \in \{\text{EDIT}, \text{FINISH}\}$
 $f' \leftarrow \text{EDIT}(p, f)$:
 - Program p updates $f \in \{\mathcal{F}_s, \mathcal{F}_x, \mathcal{F}_c\}$
 $\text{FINISH}(h)$:
 - Terminates the loop with summary h
 1: $\mathcal{F}' \leftarrow \mathcal{F}$
 $\triangleright$ Construct reflector memory
 2: $A \leftarrow [S, \mathcal{H}, \lambda]$
 $\triangleright$ Reflector loop
 3: **repeat**
 $\triangleright$ Action a and it's respective input c
 4: $(a, c) \leftarrow \mathcal{L}(\mathcal{F}', A)$
 5: **if** $a = \text{EDIT}$ **then**
 6: $(f, p) \leftarrow c$
 7: $f' \leftarrow \text{EXECUTE}(p, f)$
 $\triangleright$ Updates appropriate fields
 8: $\mathcal{F}' \leftarrow f'$
 9: $A \leftarrow A + (a, c)$
 10: **end if**
 11: **until** $a = \text{FINISH}$
 $\triangleright$ Generates a summary of changes in c
 12: $h \leftarrow c$
 13: **Output:** $(\mathcal{F}', h)$

batch outcomes and is augmented with ground truth when available, along with a GTC constructed from upcoming task descriptions and prior reflection summaries (Section 3.3). Together these complement each other to guide the Reflector in diagnosing errors and making surgical Python-based edits to the three fields. This process repeats across multiple batches, enabling the system to accumulate knowledge over time without rewriting prompts from scratch. The key mechanism enabling precise adaptation is a *code-based field update*, illustrated in Figure 2. Instead of regenerating the full prompt, the Reflector generates a short Python program that modifies only the relevant part of a field. Edits can range from simple string replacements to more complex restructuring, and are run in an isolated environment for safety, and are described in detail in Section 3.4. The overall adaptation loop is formalized in Algorithm 1, and the Reflector module in Algorithm 2.

3.3 Global Training Context (GTC)

REVERE maintains a Global Training Context that aggregates signals across training iterations, enabling adaptation beyond local feedback via three complementary signals: **Cumulative Cheatsheet** ($\mathcal{F}_c$): It is a continually updated, lightweight collection of concise, domain-specific strategies recorded in natural language by the Reflector. Initialized empty, it grows by distilling execution experience into actionable reminders used directly by the agent during task execution. **Reflection History** ($\mathcal{H}$): A record of prior reflection summaries, where each entry $h \in \mathcal{H}$ captures the rationale and outcome of an adaptation step. Unlike $\mathcal{F}_c$, which guides task execution, $\mathcal{H}$ guides the Reflector itself, by preventing contradictory edits from short-term or noisy feedback and promoting stable, incremental adaptation across batches. **Auxiliary Context** (λ): exposes the Reflector to task descriptions beyond the current batch, drawn preferentially from unseen training tasks, when no such tasks

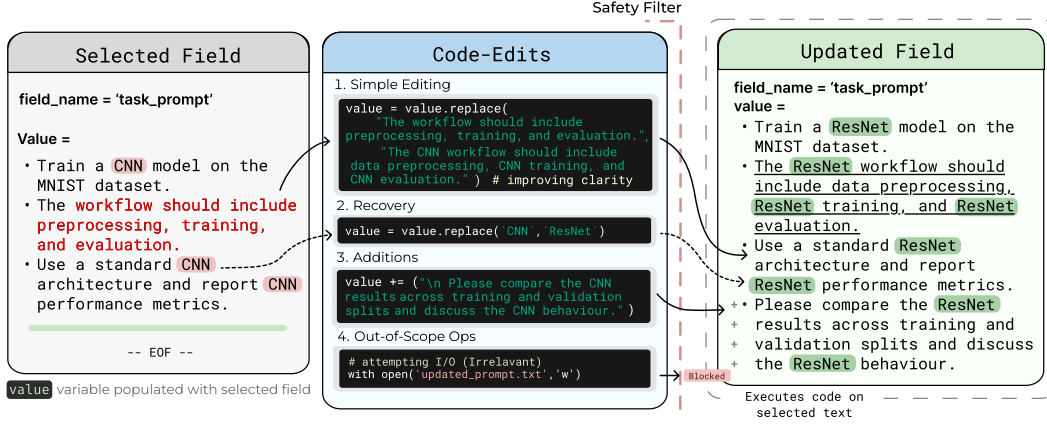


Figure 2: **The Reflector code-edit tool.** Targeted Python edits update only the relevant part of a selected field, avoiding the semantic drift of full-prompt rewrites.

remain, λ is sampled from previously trained task descriptions. This encourages updates that generalize across future task variations rather than overfitting to the current batch.

Together: $\mathcal{F}_c$ provides reusable guidance, $\mathcal{H}$ maintains long-term coherence, and λ discourages local optima. Equipped with these signals, the Reflector can make informed field updates. GTC growth is bounded by design and configurable to fit resource budgets: Auxiliary Context and Reflection History are tunable hyperparameters, and Reflection History resets each epoch (Algorithm 1). Only the Cheatsheet accumulates across iterations. In practice, GTC size remained well within typical context limits throughout our experiments, staying roughly within 20–40k tokens across all three benchmarks.

3.4 Reflection and Update Mechanism

The Reflector is a single agent responsible for both diagnosing failures and editing the fields (Prompt in Appendix F.2). Keeping these roles unified — rather than splitting them across a multi-agent pipeline (Zhang et al., 2025c; Hu et al., 2025) — preserves a coherent view of the evolving system state and avoids hand-off boundaries that can cause misinterpreted intent and incoherent updates. The central challenge is performing targeted edits without *semantic drift*. Full prompt regeneration tends to silently alter unrelated instructions and overwrite stable, validated content. To address this, we introduce a lightweight **code-based update tool** inspired by the CodeAct framework (Wang et al., 2024). As illustrated in Figure 2, the Reflector selects a field $f \in \mathcal{F}$ and generates a short Python program p that modifies it. Given an original task prompt (Figure 2 left), the Reflector generates Python code (Figure 2 center) that directly operates on the prompt text, for instance replacing an imprecise instruction with a clearer one, swapping a model reference from CNN to ResNet, or appending new behavioral instructions. Each operation targets only the relevant substring, leaving the rest of the prompt intact. The safety filter intercepts any out-of-scope operations such as file I/O before execution (these restrictions are configurable, allowing users to relax or tighten the safety constraints if needed), and filters them. The approved program is executed in a secure, isolated environment (Figure 2 right) to produce the updated field ($\mathcal{F}'$). This interface provides three key advantages:

- (i) **Targeted, low-overhead updates:** Edits are applied only to the relevant portions of $\mathcal{F}$ via code-based transformations, preventing semantic drift and avoiding overwrites of stable content.
- (ii) **Expressive, unconstrained modifications:** Unlike template- or rule-based methods (Opsahl-Ong et al., 2024; Zhang et al., 2025c), Python enables flexible text transformations, leveraging the Reflector’s coding ability for precise updates without complex schemas or restrictive APIs. Fields are further authored as Jinja2 templates rather than plain strings, so the Reflector can condition rendered content on task attributes (e.g., domain or task type), reuse shared instruction blocks across conditions, and avoid duplicating near-identical heuristics verbatim — capabilities a flat string field cannot express.
- (iii) **Safe,**

predictable execution: a two-layer design (static filter + isolated runtime) keeps updates auditable; programs exceeding string operations are rejected and the failure is fed back for retry. While this may cause occasional tool failures, we treat it as a necessary trade-off for execution safety. The filter is configurable, allowing practitioners to relax constraints for more expressive edits if needed.

4 Experiment Setup

We evaluate REVERE with a primary focus on long-horizon research-coding tasks. Additionally, we considered both single-shot and interactive settings to assess robustness and generalization across diverse research workflows (Section 4.1). For each benchmark, we define offline and online adaptation regimes (Section 4.2) and compare against strong baseline methods (Section 4.3). A summary of benchmark datasets, including task counts, approximate per-task inference cost, and resource constraints, is provided in Appendix A.1.

4.1 Benchmarks

SUPER (Bogin et al., 2024) is a benchmark of 45 long-horizon research-coding tasks that require agents to set up and execute experiments from real research repositories in a containerized environment. The tasks capture realistic workflows, including dependency installation, configuration, and resolving runtime or version issues. Agent performance is evaluated using the benchmark’s standard metrics: (i) *Output Match* requires reproduced results (e.g., accuracy, F1 score, or error rate) to match expert-reported outputs, (ii) *Landmarks* measure the presence of expected indicators of correct progress in execution logs, with higher scores assigned when more expected signals are observed, and (iii) *Overall* is the average of Output Match and Landmarks, serving as the primary summary metric.

ResearchCodeBench (RCB) (Hua et al., 2025) evaluates an LLM’s ability to re-implement core methodologies from research papers in a *single-shot* setting. For each task, the agent is provided with the paper and partially masked code files, and must reconstruct the missing implementation in a single forward pass. The benchmark comprises 212 tasks from 20 top-tier venues (e.g., ICLR, NeurIPS). Performance is measured by *Accuracy*: each task is scored as pass (1) or fail (0) based on whether the reconstructed code passes hidden unit tests without errors, and Accuracy is the mean score across all tasks.

ScienceAgentBench (SAB) (Chen et al., 2025) evaluates language agents on data-driven scientific tasks in an *interactive* code-generation setting. Each task requires producing a self-contained Python program implementing a component of a scientific workflow, often grounded in machine learning methodologies. Unlike single-shot settings, agents iteratively execute and refine code using runtime feedback. The benchmark contains 102 tasks derived from 44 peer-reviewed publications across four scientific disciplines. Evaluation uses two metrics: (i) Success Rate (SR), measuring whether a program satisfies task-specific execution and output criteria, and (ii) CodeBERTScore (CBS), which measures semantic similarity between generated and reference code using contextual embeddings.

4.2 Benchmarks Extension for Self-Adaptation

For each benchmark, we consider two adaptation settings, *offline* and *online*, which differ in task exposure and supervision. In the **offline** setting, adaptation uses fixed training and validation tasks, while evaluation is performed on a held-out test set, with train/val/test splits of 9/9/27, 34/34/144, and 20/20/62 for SUPER, ResearchCodeBench, and ScienceAgentBench respectively. We evaluate two supervision conditions, denoted **GT** and **No-GT**: under GT, the agent receives benchmark-specific ground-truth hints during training (see Appendix A.1 for GT contents); under No-GT, ground truth is withheld entirely and the agent adapts solely from its own execution traces (i.e., unsupervised adaptation). In the **online** setting, tasks arrive sequentially from the full dataset without repetition, and no ground-truth is available at any point, hence inherently **No-GT**. Since the full task pool includes the tasks used for offline test evaluation, we report both full-dataset and test-subset performance, with the latter enabling direct comparison against the offline test split.

Method	GT	SUPER (%)			RCB (%)	SAB (%)	
		Landmarks	Output Match	Overall		Accuracy	SuccessRate
		long-horizon			single shot	interactive - ReAct	
Baseline	–	24.70	9.30	17.00	27.80	20.90	67.50
SOTA [†]	–	35.80	14.80	25.30	31.90	23.50	88.20
Offline Adaptation							
GEPA	✗	0.00 _{–35.80}	18.60 _{+3.80}	9.30 _{–16.00}	25.69 _{–6.21}	17.42 _{+6.08}	77.80 _{–10.94}
GEPA	✓	1.93 _{–33.87}	2.60 _{–12.20}	2.27 _{–23.03}	10.44 _{–21.46}	19.35 _{–4.15}	65.60 _{–22.60}
ACE	✗	19.53 _{–16.27}	18.51 _{+3.71}	19.02 _{–6.28}	21.99 _{–9.91}	8.60 _{–14.90}	53.48 _{–34.72}
ACE	✓	27.22 _{–8.58}	18.64 _{+3.84}	22.93 _{–2.37}	7.87 _{–24.03}	4.30 _{–19.20}	24.28 _{–63.92}
REVERE	✗	35.67 _{–0.13}	19.76 _{+4.96}	27.71 _{+2.41}	28.75 _{–3.15}	23.23 _{–0.27}	80.25 _{–7.95}
REVERE	✓	35.84 _{+0.04}	23.76 _{+8.96}	29.80 _{+4.50}	33.20 _{+1.30}	28.39 _{+4.89}	82.84 _{–5.36}

Table 1: Offline adaptation Results. Reported values are mean over 5 runs. ✓/✗ denote with/without ground-truth (GT) hints; (–) indicates inference-only. $+x$ and $-x$ showing gains and drops relative to Static SOTA results marked by[†].

4.3 Baseline Methods

Baseline and SOTA Prompts: Our baseline system uses minimal, non-optimized prompts containing only the core task description, referred to as ‘baseline’ which serves as a reference for isolating the effects of REVERE’s adaptation mechanisms. All adaptation methods, including REVERE, GEPA, and ACE, start from this baseline prompt. For the SUPER (long-horizon) and SAB (interactive), we implement a ReAct-style agent (Yao et al., 2023) equipped with code tools (see Appendix A.3 for details) and for ResearchCodeBench, we used direct LLM calls to generate programs. Additionally, we report the current state-of-the-art performance for each benchmark using author-provided instructions, denoted as ‘Static SOTA’ (see Appendix F.1 for prompts).

GEPA (Genetic-Pareto) (Agrawal et al., 2025) is a sample-efficient prompt optimizer that evolves prompts via natural-language reflection and Pareto-based genetic search over prompt candidates with a fixed evaluation pool; its Pareto filter requires scoring all candidates against a held-out validation set. However, this design primarily supports offline optimization, making online adaptation infeasible.

ACE (Agentic Context Engineering) (Zhang et al., 2025c) is a playbook-based learning approach inspired by (Suzgun et al., 2025) that maintains bullet-point strategies across predefined categories updated via a multi-agent reflector-curator, processing one task at a time. We use step size one as recommended in the original paper.

REVERE and ACE adapt using only the training split; GEPA additionally uses the validation split, solely for Pareto candidate selection. See Appendix A.2 for code and configuration details for REVERE, GEPA, and ACE. All primary results are computed using GPT-4.1, using a single model for both reflection and adaptation to reduce cross-model knowledge transfer. Full model details, including primary and additional models used in analysis with Reflector/task-model configurations, are provided in Appendix A.4.

5 Results and Analysis

Under offline adaptation, REVERE is the only method that consistently benefits from stronger supervision signals, improving over Static SOTA across all benchmarks and metrics. Results are presented in Table 1. On SUPER, REVERE improves both Output Match and Landmarks simultaneously, while GEPA improves Output Match at the cost of Landmarks collapsing to zero – a direct consequence of its discard-and-rewrite design overfitting recent traces and discarding stable reasoning structure. ACE shows marginal, inconsistent gains that remain below REVERE throughout.

REVERE’s global aggregation of signals lets it keep improving on domain-heavy benchmarks where GEPA and ACE degrade under stronger supervision. The more reveal-

Method	SUPER Bench (%)			RCB (%) Accuracy	SAB (%)	
	Landmarks	Output Match	Overall		SuccessRate	CodeBERTScore
	<i>long-horizon</i>			<i>single shot</i>	<i>interactive</i>	
Baseline - Test Tasks	24.70	9.30	17.00	27.80	20.96	67.50
Baseline - All Tasks	19.63	10.40	15.01	43.80	20.58	66.86
<i>Online Adaptation</i>						
ACE - Test Tasks	26.80+2.10	11.10+1.80	18.95+1.95	15.50-12.30	11.30-9.66	66.50-1.00
ACE - All Tasks	24.80+5.17	14.20+3.80	19.50+4.49	19.81-23.99	11.80-8.78	65.5-1.36
REVERE - Test Tasks	33.50+8.80	17.60+8.30	25.50+8.10	35.40+7.60	25.80+4.84	69.61+2.11
REVERE - All Tasks	30.10+10.40	18.70+8.30	24.40+9.39	45.00+1.20	25.49+4.91	71.68+4.82

Table 2: Online Adaptation results. All Tasks: performance on full benchmark; Test Tasks: Results on Test set tasks, $+x$ and $-x$ with respect to baseline.

ing pattern emerges on ResearchCodeBench and ScienceAgentBench, which are domain-knowledge-heavy with limited structural overlap across examples, and thus constrained by the amount of transferable signal any adaptation method can exploit. Here, GEPA and ACE both degrade further when given gold hints: GEPA’s rewrite mechanism discards valuable trajectory supervision, while ACE indiscriminately absorbs step-level signals that do not transfer across heterogeneous tasks, which is evident in how their prompts evolve under adaptation (Appendix C Figure 7). REVERE instead aggregates signals globally, allowing the Reflector to distinguish recurring cross-task patterns from instance-level noise, then distributing learned strategies across the system prompt, task prompt, and cheatsheet rather than collapsing all adaptation pressure onto a single field. This makes REVERE the only method whose performance remains stable and consistently improves even when supervision is inherently noisy and task-specific (see Appendix B for standard deviations and significance tests). REVERE shows similar robustness to initialization quality beyond the baseline prompt (see Appendix D.4). **REVERE’s gains come from learning to solve tasks correctly, not from matching a reference answer.** On ScienceAgentBench, REVERE achieves CodeBERTScore closest to SOTA among all methods while improving SuccessRate by +4.89% over SOTA. This gap reflects a known limitation of CodeBERTScore: lexical similarity to a single reference implementation does not necessarily track functional correctness, making SuccessRate the more meaningful indicator of task success in this interactive setting (see Appendix E.4 for a task-level reconciliation, including matched examples where REVERE achieves SR=1 with lower CBS due to functionally valid alternative implementations). We provide an additional case study analyzing autonomously discovered failure modes of SUPER baseline agent and prompt evolution across adaptation iterations in Appendix D.2 and Appendix D.3.

REVERE retains comparable performance in online settings despite minimal feedback. Table 2 reports the stricter online regime, where tasks arrive sequentially, each encountered once, and no ground-truth labels are available at any point. REVERE improves over the baseline across all benchmarks and metrics, while ACE frequently falls below the baseline, indicating that a cheatsheet updated from local signals alone is insufficient for stable cross-task generalization without the broader context that REVERE’s GTC provides. Lower absolute performance relative to offline training is structurally expected: processed tasks cannot be revisited, future tasks are unseen at adaptation time, and supervision is entirely

Method	SUPER Bench (%)			RCB(%) Acc.	SAB (%)	
	Landmarks	Output Match	Overall		SuccessRate	CodeBERTScore
	<i>long-horizon</i>			<i>single shot</i>	<i>interactive</i>	
REVERE	38.10	31.40	34.70	35.40	30.60	83.3
REVERE w / o Cheatsheet	30.70 -7.40	16.70 -14.70	23.70 -11.00	30.60 -4.80	20.90 -9.70	78.30 -5.00
REVERE w / o Auxiliary context	24.80 -13.30	24.10 -7.30	24.40 -10.30	31.90 -3.50	25.80 -4.80	79.80 -3.50
REVERE w / o Reflection history	25.90 -12.20	9.30 -22.10	17.60 -17.10	32.60 -2.80	22.50 -8.10	75.90 -7.40
REVERE w / o GTC	21.70 -16.40	10.20 -21.20	16.00 -18.70	27.10 -8.30	19.30 -11.30	77.30 -6.00
REVERE w / o Code Edits	0.00 -38.10	1.11 -30.29	0.55 -34.15	26.38 -9.02	4.83 -25.77	59.88 -23.42

Table 3: Ablation on Offline Adaptation. Results correspond to a representative run, $-x$ indicates difference with respect to REVERE (full model)

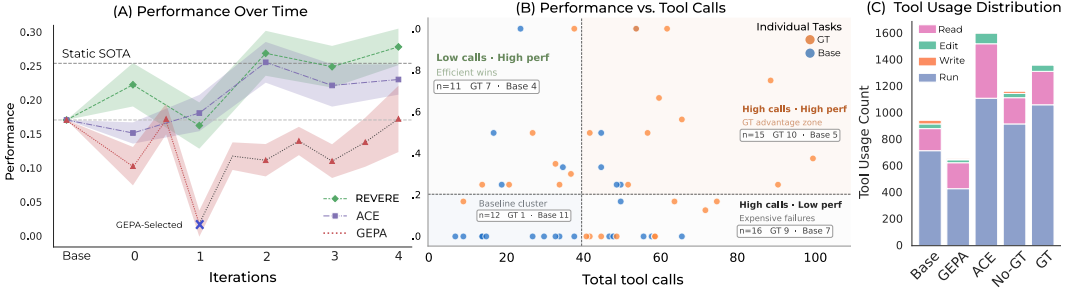


Figure 3: **Performance Analysis on the adapted agent on SUPER Overall metric.** (A) Performance trajectory across iterations. (B) Per-task performance vs. total tool calls; orange = REVERE (GT), blue = Baseline. (C) Tool usage across variants (Baseline, GEPA, ACE, REVERE w/o GT, REVERE w/ GT). *Note:* GEPA iterations denote total candidates and are normalized to the axis scale; the blue marker indicates the Pareto-selected prompt.

absent. Yet relative gains remain consistent across benchmarks, confirming that REVERE’s adaptation is driven by recurring execution patterns rather than ground-truth dependence. Notably, on ResearchCodeBench, REVERE surpasses its own offline result, partly due to the online setting including a broader task pool with partially overlapping test tasks that provide additional adaptation signal. On SUPER, REVERE online matches Static SOTA without any ground-truth access, underscoring that effective prompt evolution does not require supervision to approach expert-crafted performance. We further examine REVERE’s sensitivity to model and Reflector choice in Appendix D.1.

Do we need all components of REVERE? The ablation results (Table 3) confirm that REVERE’s effectiveness arises from the interaction of its core components. Removing the cheatsheet memory causes broad degradation across metrics, showing that cumulative retention of discovered fixes and heuristics is central to adaptation. Excluding auxiliary context or reflection history individually degrades performance, but for different reasons: auxiliary context lets the Reflector anticipate patterns in upcoming tasks, while reflection history prevents past mistakes from recurring. The largest drops occur when the entire GTC is removed, confirming these forward- and backward-looking signals are complementary. Most strikingly, replacing targeted code-based edits with full-prompt regeneration collapses performance almost entirely, a larger drop than removing the GTC itself (Table 3). This confirms the two components play distinct roles: the GTC drives what gets learned, while the code-edit mechanism is what sustains those updates across iterations. Without it, accumulated signal is overwritten before it can meaningfully learn (a retention analysis following this failure mode is included in Appendix C.1).

REVERE improves steadily across iterations instead of locking early on weak adaptation. Figure 3(A) shows REVERE improving steadily across iterations, recovering from a marginal early dip to surpass both Static SOTA and ACE by the final iteration. GEPA, in contrast, is unable to capture the heterogeneity of long-horizon tasks or account for learning across iterations, causing its heuristic-based Pareto selection to lock in a weaker prompt based on early evaluation, highlighting the fragility of heuristic-based selection without accumulated adaptation context. **An intuitive learning curve emerges under adaptation, spanning efficient wins, persistent successes, and informative failures.** Figure 3(B) reveals that baseline tasks cluster in the bottom-left reflecting shallow exploration, while REVERE-GT tasks spread across all quadrants, indicating that adaptation induces richer exploratory behaviour. This dispersion captures three distinct regimes: efficient wins (top-left) where tasks are resolved with minimal tool usage, persistent successes (top-right) where harder tasks require greater investment, and informative failures (bottom-right) where extended traces provide the richest adaptation signal for future improvement. Together these dimensions show that exploration depth, efficiency, and persistence are all actively shaped by REVERE’s learning process. A detailed trajectory-level breakdown of improving, declining, and neutral tasks provided in Appendix D.5. **More tool calls alone don’t drive progress — what matters is converting them into stronger gains.** Figure 3(C) reveals how adaptation

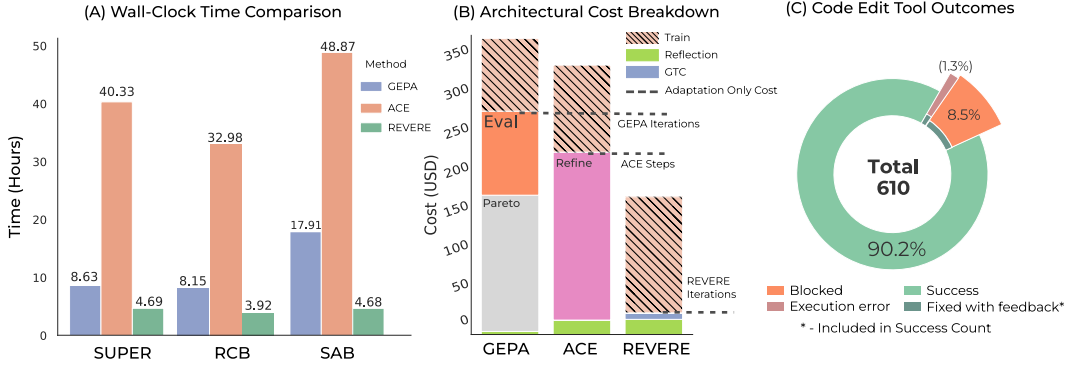


Figure 4: **Efficiency analysis of REVERE** (A) Illustrates wall-clock time for offline adaptation of REVERE with baselines. (B) Compares computational cost (USD) on the SUPER task in an offline setting across REVERE, ACE, and GEPA, decomposed by architectural components. Dashed lines indicate sole adaptation cost. (C) Distribution of outcomes across code-edit tool calls in the ground-truth setting across all benchmarks.

strategy shapes tool usage. GEPA under-explores relative to the baseline, consistent with its context-discarding design. Both ACE and REVERE show increased tool usage over the baseline, consistent with their performance gains. However REVERE converts tool calls into stronger performance gains more efficiently than ACE, confirming that our method adapts the underlying agent efficiently.

Speed and cost scale in REVERE’s favor: adaptation stays lightweight and cost-effective without sacrificing reliability. Figure 4(A) shows REVERE completes adaptation in a fraction of the wall-clock time of ACE and GEPA across all benchmarks, over 2.7x faster, aided by parallelised execution and efficient memory management. Figure 4(B) compares adaptation costs (component breakdown in Appendix E.2). Training cost (shown hatched) is shared across all methods and thus not a meaningful differentiator; curation costs are also negligible (REVERE: \$0.16 vs. ACE: \$0.47 on SUPER). The key difference is adaptation overhead: REVERE’s adaptation-only cost is nearly 10x lower than ACE and GEPA, which incur repeated per-task refinement steps and Pareto re-evaluation respectively, both scaling unfavorably with task complexity. REVERE avoids this through targeted edits and a compact GTC, keeping per-task overhead low even as context grows (see Appendix E.3 for cost accounting rationale). **Most code-based edits succeed outright, and the safety filter turns the rest into a practical, self-correcting safeguard rather than a bottleneck.** Figure 4(C) reports that 90.2% of code-based edit attempts succeed outright; remaining cases are caught by the safety filter, which is configurable. We recommend retaining it, as the Reflector self-corrects blocked edits within the same iteration, making this a practical safeguard for safe LLM-to-LLM integration.

6 Conclusion

We present REVERE, a lightweight, self-adapting framework for LLM agents, applied especially to research-coding tasks. REVERE maintains a Global Training Context – reflection history, auxiliary context, and a cumulative cheatsheet – alongside local, per-batch evaluation signals, and applies precise code-edits to prompts, more efficiently than existing prompt updating mechanisms. This supports both offline and online adaptation while remaining interpretable and easy to integrate. REVERE shows consistent gains over strong baselines (GEPA, ACE) across three settings — SUPER (long-horizon), ResearchCodeBench (single-shot), and ScienceAgentBench (interactive) — under both supervised and unsupervised learning, while being nearly 10x more cost-effective and 2.7x faster than prior methods. Our analysis shows gains are more modest on domain-heavy benchmarks, suggesting prompt adaptation alone may not capture task-specific knowledge, and that the growing GTC may accumulate stale information over time. Exploring task-specific adaptation and GTC pruning are natural directions for future work.

References

- Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2025. URL <https://arxiv.org/abs/2507.19457>.
- UK AI Security Institute. Inspect AI: Framework for Large Language Model Evaluations, 2024. URL https://github.com/UKGovernmentBEIS/inspect_ai.
- Ben Bogin, Kejuan Yang, Shashank Gupta, Kyle Richardson, Erin Bransom, Peter Clark, Ashish Sabharwal, and Tushar Khot. Super: Evaluating agents on setting up and executing tasks from research repositories, 2024. URL <https://arxiv.org/abs/2409.07440>.
- Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. Mle-bench: Evaluating machine learning agents on machine learning engineering, 2025. URL <https://arxiv.org/abs/2410.07095>.
- Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, Vishal Dey, Mingyi Xue, Frazier N. Baker, Benjamin Burns, Daniel Adu-Ampratwum, Xuhui Huang, Xia Ning, Song Gao, Yu Su, and Huan Sun. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery, 2025. URL <https://arxiv.org/abs/2410.05080>.
- Nicholas Edwards, Yookyung Lee, Yujun Audrey Mao, Yulu Qin, Sebastian Schuster, and Najoung Kim. Rexbench: Can coding agents autonomously implement ai research extensions?, 2025. URL <https://arxiv.org/abs/2506.22598>.
- Aniketh Garikaparathi, Manasi Patwardhan, and Arman Cohan. Researchgym: Evaluating language model agents on real-world ai research, 2026. URL <https://arxiv.org/abs/2602.15112>.
- Paul Gauthier. Aider polyglot benchmark. <https://aider.chat/polyglot>, 2024. Accessed: 2025-11-13.
- Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2025. URL <https://arxiv.org/abs/2408.08435>.
- Yue Hu, Yuzhu Cai, Yaxin Du, Xinyu Zhu, Xiangrui Liu, Zijie Yu, Yuchen Hou, Shuo Tang, and Siheng Chen. Self-evolving multi-agent collaboration networks for software development, 2024. URL <https://arxiv.org/abs/2410.16946>.
- Tianyu Hua, Harper Hua, Violet Xiang, Benjamin Klieger, Sang T. Truong, Weixin Liang, Fan-Yun Sun, and Nick Haber. Researchcodebench: Benchmarking llms on implementing novel machine learning research code, 2025. URL <https://arxiv.org/abs/2506.02314>.
- Jin Huang, Silviu Cucerzan, Sujay Kumar Jauhar, and Ryen W. White. Idea2plan: Exploring ai-powered research planning, 2025. URL <https://arxiv.org/abs/2510.24891>.
- Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation, 2024. URL <https://arxiv.org/abs/2310.03302>.
- Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code, 2025. URL <https://arxiv.org/abs/2502.13138>.
- Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines, 2023. URL <https://arxiv.org/abs/2310.03714>.

- Patrick Tser Jern Kon, Jiachen Liu, Xinyi Zhu, Qiuyi Ding, Jingjia Peng, Jiarong Xing, Yibo Huang, Yiming Qiu, Jayanth Srinivasa, Myungjin Lee, Mosharaf Chowdhury, Matei Zaharia, and Ang Chen. Exp-bench: Can ai conduct ai research experiments?, 2025. URL <https://arxiv.org/abs/2505.24785>.
- Zijie Lin, Yiqing Shen, Qilin Cai, He Sun, Jinrui Zhou, and Mingjun Xiao. Autop2c: An llm-based agent framework for code repository generation from multimodal content in academic papers, 2025. URL <https://arxiv.org/abs/2504.20115>.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL <https://arxiv.org/abs/2303.17651>.
- Bodhisattwa Prasad Majumder, Bhavana Dalvi Mishra, Peter Jansen, Oyvind Tafjord, Niket Tandon, Li Zhang, Chris Callison-Burch, and Peter Clark. CLIN: A continually learning language agent for rapid task adaptation and generalization. In *First Conference on Language Modeling*, 2024. URL <https://openreview.net/forum?id=xS6zx1aBI9>.
- Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs, 2024. URL <https://arxiv.org/abs/2406.11695>.
- Siba Smarak Panigrahi, Jovana Videnović, and Maria Brbić. Heureka-bench: A benchmarking framework for ai co-scientist, 2026. URL <https://arxiv.org/abs/2601.01678>.
- Xin Peng and Chong Wang. Code digital twin: Empowering llms with tacit knowledge for complex software development, 2025. URL <https://arxiv.org/abs/2503.07967>.
- Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using llm agents as research assistants, 2025. URL <https://arxiv.org/abs/2501.04227>.
- Tobias Schnabel and Jennifer Neville. Symbolic prompt program search: A structure-aware approach to efficient compile-time prompt optimization, 2024. URL <https://arxiv.org/abs/2404.02319>.
- Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2code: Automating code generation from scientific papers in machine learning, 2025. URL <https://arxiv.org/abs/2504.17192>.
- Wenhang Shi, Yiren Chen, Shuqing Bian, Xinyi Zhang, Kai Tang, Pengfei Hu, Zhe Zhao, Wei Lu, and Xiaoyong Du. No loss, no gain: Gated refinement and adaptive compression for prompt optimization, 2025. URL <https://arxiv.org/abs/2509.23387>.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL <https://arxiv.org/abs/2303.11366>.
- Chenglei Si, Zitong Yang, Yejin Choi, Emmanuel Candès, Diyi Yang, and Tatsunori Hashimoto. Towards execution-grounded automated ai research, 2026. URL <https://arxiv.org/abs/2601.14525>.
- Zachary S. Siegel, Sayash Kapoor, Nitya Nagdir, Benedikt Stroebl, and Arvind Narayanan. Core-bench: Fostering the credibility of published research through a computational reproducibility agent benchmark, 2024. URL <https://arxiv.org/abs/2409.11363>.
- Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. Paperbench: Evaluating ai’s ability to replicate ai research, 2025. URL <https://arxiv.org/abs/2504.01848>.

- Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic cheatsheet: Test-time learning with adaptive memory, 2025. URL <https://arxiv.org/abs/2504.07952>.
- Minyang Tian, Luyu Gao, Shizhuo Dylan Zhang, Xinan Chen, Cunwei Fan, Xuefei Guo, Roland Haas, Pan Ji, Kittithat Krongchon, Yao Li, Shengyan Liu, Di Luo, Yutao Ma, Hao Tong, Kha Trinh, Chenyu Tian, Zihan Wang, Bohao Wu, Yanyu Xiong, Shengzhu Yin, Minhui Zhu, Kilian Lieret, Yanxin Lu, Genglin Liu, Yufeng Du, Tianhua Tao, Ofir Press, Jamie Callan, Eliu Huerta, and Hao Peng. Scicode: A research coding benchmark curated by scientists, 2024. URL <https://arxiv.org/abs/2407.13168>.
- Dhruv Trehan and Paras Chopra. Why llms aren’t scientists yet: Lessons from four autonomous research attempts, 2026. URL <https://arxiv.org/abs/2601.03315>.
- Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024. URL <https://arxiv.org/abs/2402.01030>.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025. URL <https://arxiv.org/abs/2407.16741>.
- Zhen Wang, Fan Bai, Zhongyan Luo, Jinyan Su, Kaiser Sun, Xinle Yu, Jieyuan Liu, Kun Zhou, Claire Cardie, Mark Dredze, Eric P. Xing, and Zhiting Hu. Fire-bench: Evaluating agents on the rediscovery of scientific insights, 2026. URL <https://arxiv.org/abs/2602.02905>.
- Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Ben Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, Shubh-Agrawal, Sandeep Singh Sandha, Siddhartha Naidu, Chinmay Hegde, Yann LeCun, Tom Goldstein, Willie Neiswanger, and Micah Goldblum. Livebench: A challenging, contamination-limited llm benchmark, 2025. URL <https://arxiv.org/abs/2406.19314>.
- Yanzheng Xiang, Hanqi Yan, Shuyin Ouyang, Lin Gui, and Yulan He. Scireplicate-bench: Benchmarking llms in agent-driven algorithmic reproduction from research papers, 2025. URL <https://arxiv.org/abs/2504.00255>.
- 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, 2024. URL <https://arxiv.org/abs/2405.15793>.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL <https://arxiv.org/abs/2210.03629>.
- Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2025a. URL <https://arxiv.org/abs/2505.22954>.
- Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. Aflow: Automating agentic workflow generation, 2025b. URL <https://arxiv.org/abs/2410.10762>.
- Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models, 2025c. URL <https://arxiv.org/abs/2510.04618>.
- Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. Expel: Llm agents are experiential learners, 2024. URL <https://arxiv.org/abs/2308.10144>.

Yilun Zhao, Weiyuan Chen, Zhijian Xu, Manasi Patwardhan, Chengye Wang, Yixin Liu, Lovekesh Vig, and Arman Cohan. AbGen: Evaluating large language models in ablation study design and evaluation for scientific research. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12479–12491, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.611. URL <https://aclanthology.org/2025.acl-long.611/>.

A Experiment Details

A.1 Benchmark datasets and resource limits

Table 4 reports task counts, task inputs, GT content, per-task inference cost, and resource limits (LLM calls and combined agent + evaluation time) across the three benchmarks. Costs reflect single-run agent execution and vary with task complexity and trajectory length. SUPER is the most expensive due to its long-horizon interactive nature; we use the Expert subset, which contains expert-designed tasks and is therefore more challenging and representative of real-world research scenarios. ResearchCodeBench is the most lightweight given its single-shot setting. Resource limits are fixed and held constant across all methods to ensure a fair and controlled comparison. GT content is deliberately kept incomplete across all three benchmarks, allowing hints to guide adaptation without letting agents shortcut the task by copying the answer outright. For SUPER, the expert trajectory comes from a structurally different execution environment and omits exact dependency versions, which are unreliable to reproduce and known to affect baselines inconsistently depending on when they are run. For ResearchCodeBench, only partially masked code is given, since the full implementation would leak answers across tasks sharing masking patterns. For ScienceAgentBench, gold execution results and outputs are withheld so agents cannot infer expected outputs without actually implementing and reasoning through the task. This isolates genuine adaptation gains from direct answer-copying, keeping the comparison across methods rigorous.

Benchmark	Tasks	Task Input	Ground Truth	Cost P.T	Resource Limits (Calls / Exec Time)
Super-Expert <i>Setting up, and executing tasks from research repositories.</i>	45	Query, GitHub repo, commit	Expert trajectory	\$1–3	100 iters / 20 min
ResearchCodeBench <i>Code completion by implementing methodology from the paper.</i>	212	Paper, masked code, context files	Target code	~ \$0.2	1 Call / 10 min
ScienceAgentBench <i>Scientific coding tasks spanning multiple domains for data-driven discovery.</i>	102	Task instruction, dataset path	Gold Python file	~ \$0.3	20 iters / 40 min

Table 4: Benchmark datasets used in our evaluation, with task counts, task inputs, ground-truth (GT) content provided to REVERE under the GT-assisted offline setting, approximate per-task inference cost (Cost P.T), and per-task resource limits set for each task (no. of LLM calls and time for agent + evaluation execution). Costs are estimates based on gpt-4.1.

A.2 Implementation Details

REVERE. For REVERE, training was performed for 5 epochs with a batch size of 3 on SUPER, a batch size of 7 on ScienceAgentBench, and a batch size 5 on ResearchCodeBench. Auxiliary context sizes were set to 6 (SUPER), 3 (ResearchCodeBench, reduced due to longer code and paper context segments), and 5 (ScienceAgentBench). No separate validation set was used for REVERE, allowing all available training data to contribute directly to parameter updates. In the online setting, the auxiliary context λ is sampled from previously encountered tasks only, as no future tasks are accessible in the online setting.

GEPA. We use GEPA’s official implementation² with 32 optimization iterations on SUPER (early stopping with 20 consecutive iterations without improvement, mini feedback set of 3 examples, Pareto frontier size 9), a budget of 600 metric evaluations (≈ 20 iterations) on ResearchCodeBench, and a maximum of 15 iterations with mini feedback set of size 7 on ScienceAgentBench. GEPA additionally uses the held-out validation split solely for Pareto candidate selection. For fair cost comparison in Figure 4, GEPA’s cost and wall-clock time analysis are capped to appropriate iterations to match the same training sample count as REVERE although it ran for more iterations with no improvements.

²<https://github.com/gepa-ai/gepa-artifact>

ACE. We use ACE’s official implementation³ with a step size of 1 (i.e., batch size 1) across all benchmarks, consistent with the original ACE paper’s implementation [Zhang et al. \(2025c\)](#).

Refer to Appendix E.2 for architectural breakdown of ACE and GEPA methods.

A.3 Agentic and Evaluation Implementation

Since REVERE, ACE, and GEPA operate under architecturally distinct frameworks, directly comparing them requires a shared foundation. To this end, we standardize the baseline agent, tool interfaces, and execution environment across all three methods, ensuring that observed performance differences reflect genuine methodological distinctions rather than implementation artifacts. We implement a ReAct-style agent that interleaves reasoning and tool usage to solve complex coding and research tasks through an iterative interaction loop with the environment. The agent is built on a modular tool-calling framework using our custom-built `aicodetools`⁴, inspired by prior agent frameworks such as InspectAI ([AI Security Institute, 2024](#)) and LangChain. Our library abstracts system-level complexities and isolates execution overhead from the adaptation framework, allowing efficient extensibility and integration of new capabilities. It supports containerized execution and parallel task handling, ensuring scalability, reproducibility, and efficient resource utilization. The agent is equipped with core tools including `read`, `write`, `edit`, and `run_command`, enabling controlled interaction with the filesystem and execution environment. Additionally, we introduce a `finish` tool that allows the agent to produce structured outputs, including final answers, execution summaries, and explicit reasoning traces, thereby improving interpretability and evaluation consistency.

We further extend the evaluation benchmarks—SUPER, Research Code Bench (RCB), and ScienceAgentBench—by integrating Dockerized execution, a streamlined pipeline, and enhanced evaluation mechanisms. Critically, because all methods share the same containerized environment, any misconfiguration is immediately visible and affects all methods uniformly, eliminating environment-specific confounds as a source of variance. These contributions ensure safe and reproducible execution while enabling efficient experimentation through parallelization and standardized logging. In addition, REVERE framework is designed to be highly extensible, supporting seamless incorporation of additional benchmarks, tools, and evaluation protocols, and thereby facilitating continued development and broader applicability of agent-based research systems.

A.4 System Hardware

All methods including REVERE and baselines use the Azure gpt-4.1 (api version : 2025-03-01-preview) model and all experiments are conducted on standard_d16ads_v5 with 16 vCPUs and 64 GiB memory. For further analysis in Appendix D.1, we evaluate heterogeneous Reflector/task-model configurations spanning GPT-5.4, GPT-5.4-mini, GPT-5.2, and gemini-3-flash-preview from Google AI Studio.

We select GPT-4.1 as our primary model for its 1M-token context window, needed to handle the extreme context requirements of offline adaptation — inputs of 300k–500k tokens for the Reflector and 40k–120k tokens for agents, driven by large code repositories, academic papers, and long-horizon reasoning traces. The online setting carries lower context requirements per adaptation step, which is why Appendix D.1 additionally evaluates smaller-context models (GPT-5.4-mini, GPT-5.2) alongside GPT-4.1, Gemini-3-Flash.

B Robustness on Adaptation

Table 5 reports mean performance and standard deviations across five runs. Figure 5 and Figure 6 provide statistical support for the results discussed in Section 5.

³<https://github.com/ace-agent/ace>

⁴<https://pypi.org/project/aicodetools/>

Method	GT	SUPER Bench (%)			RCB (%)	ScienceAgentBench (%)	
		Landmarks	Output Match	Overall	Accuracy	SuccessRate	CodeBERTScore
		long-horizon			single shot	interactive - ReAct	
Offline Adaptation							
GEPA	✗	0.0 $\pm$ 0.0	18.6 $\pm$ 0.85	9.30 $\pm$ 0.42	25.69 $\pm$ 0.00	17.42 $\pm$ 1.35	77.80 $\pm$ 2.33
GEPA	✓	1.93 $\pm$ 2.15	2.60 $\pm$ 1.0	2.27 $\pm$ 0.99	10.44 $\pm$ 3.12	17.09 $\pm$ 2.70	78.53 $\pm$ 1.84
ACE	✗	19.53 $\pm$ 10.01	18.51 $\pm$ 3.62	19.02 $\pm$ 3.2	21.99 $\pm$ 3.28	8.60 $\pm$ 7.27	53.48 $\pm$ 8.08
ACE	✓	27.22 $\pm$ 3.04	18.64 $\pm$ 3.91	22.93 $\pm$ 1.61	7.87 $\pm$ 1.75	4.30 $\pm$ 1.86	24.28 $\pm$ 1.95
REVERE	✗	35.67 $\pm$ 3.09	19.76 $\pm$ 3.71	27.71 $\pm$ 2.99	28.75 $\pm$ 2.0	23.23 $\pm$ 3.14	80.25 $\pm$ 2.7
REVERE	✓	35.84 $\pm$ 5.52	23.76 $\pm$ 4.59	29.80 $\pm$ 4.26	33.20 $\pm$ 1.8	28.39 $\pm$ 4.36	82.84 $\pm$ 1.19

Table 5: Offline Adaptation Results; RCB: ResearchCodeBench; with (✓) and without (✗) ground-truth (GT) Hints, $\pm$: standard deviation (Std Dev) across 5 runs.

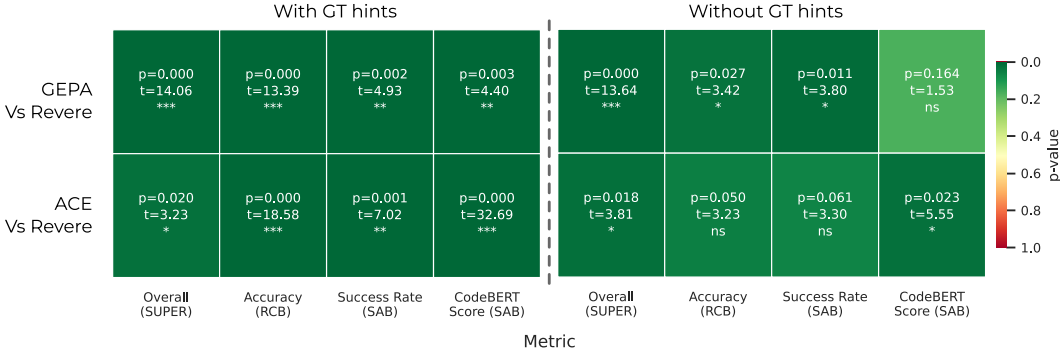


Figure 5: Welch t-test : REVERE *** $p < 0.001$, ** $p < 0.01$; * $p < 0.05$; ns $p \geq 0.05$; $t > 0$ means REVERE has higher mean

Significance of gains with ground-truth supervision. With GT hints, REVERE achieves statistically significant improvements over both GEPA and ACE across all benchmarks and metrics ($p < 0.001$, $t > 0$). The high t -statistics in this setting, particularly on RCB Accuracy ($t = 13.39$ vs. GEPA, $t = 18.58$ vs. ACE) and SAB CodeBERTScore ($t = 4.40$ vs. GEPA, $t = 32.69$ vs. ACE), reflect that REVERE not only achieves higher means but does so consistently across runs, as supported by the standard deviations in Table 5.

Significance without ground-truth supervision. Without GT hints, significance is largely retained on SUPER. However, significance drops on domain-knowledge-heavy benchmarks, particularly SAB, where REVERE vs. ACE yields $p = 0.050$ on Accuracy and $p = 0.061$ on Success Rate. This is expected: adapting to tasks that require deep domain-specific knowledge from execution traces alone is a hard problem. Importantly, REVERE still maintains higher means throughout ($t > 0$), indicating a consistent directional advantage even where significance is not achieved.

Consistency of gains across runs. Figure 6 shows 95% confidence intervals across all methods and metrics. REVERE improves consistently over competing methods, with little to no overlap between its confidence intervals and those of GEPA and ACE on primary metrics. This is most visible on SUPER Overall and SAB Success Rate, where REVERE’s lower confidence bound exceeds the upper bound of competing methods in several comparisons. The overlap that does occur is limited to domain-knowledge-heavy settings without GT supervision, which is consistent with the significance analysis above.

C Prompt Adaptation Analysis

REVERE maintains efficient adaptation with controlled prompt growth. Figure 7 (A) shows that the three methods diverge sharply in how they manage prompt length across iterations, and the growth patterns reflect their respective architectural designs. REVERE’s components ($\mathcal{F}_s$, $\mathcal{F}_c$, $\mathcal{F}_x$) exhibit controlled, distributed growth, allowing the system to

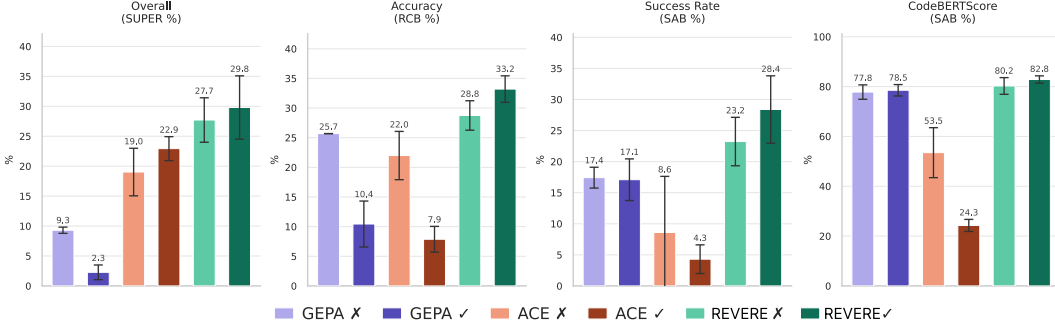


Figure 6: Offline Adaptation score with 95% CI, where GT ✓ = Adapted with ground-truth hints; X = Adapted with-out ground-truth hints;

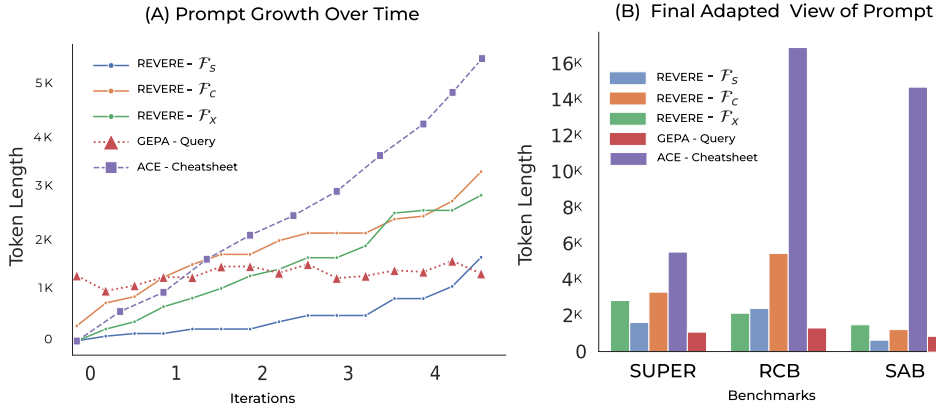


Figure 7: **Prompt Analysis:** (A) Illustrates prompt component growth in token length across iterations on the SUPER benchmark, comparing REVERE components, GEPA query, and ACE cheatsheet. (B) Illustrates final adapted prompt lengths (in tokens) across benchmarks, used in offline adaptation results with ground truth.

allocate capacity where it is most needed. ACE shows steep linear growth driven entirely by its cheatsheet, conflating task-specific and generalizable signals into bloated prompts; growing context length remains a fundamental challenge even in REVERE, and we discuss potential mitigation in Appendix E.1. GEPA maintains consistently low token count by regenerating instructions from scratch each iteration, but discards previously accumulated knowledge at every step, making it poorly suited for domain-rich, individually complex tasks. Figure 7 (B) reports final adapted prompt lengths for the results in Table 1, and directly supports our earlier observation on domain-heavy benchmarks. ACE’s cheatsheet grows dramatically on RCB, reflecting the benchmark’s heavy reliance on specialized domain knowledge that is difficult to compress into reusable prompts, accumulating task-specific traces instead of distilling generalizable signals. REVERE maintains compact prompts across all benchmarks, though as noted earlier, controlled prompt growth alone does not overcome the domain familiarity constraints that bound performance on research-oriented coding tasks.

C.1 Semantic Drift and Edit Operation Analysis

To directly evaluate whether targeted code-based edits reduce semantic drift relative to full-prompt rewriting, we measure prompt retention across consecutive adaptation iterations using three metrics: Levenshtein retention (1 – normalized Levenshtein distance between prompt versions), git-diff retention (1 – fraction of lines changed per diff between consecutive versions), and semantic similarity (cosine similarity via sentence-transformer

Method	Levenshtein Retention	Git Retention	Semantic Similarity
GEPA – full prompt rewrite	0.11	0.03	0.62
ACE – incremental delta updates	0.86	0.92	0.99
REVERE – targeted edits	0.84	0.89	0.94

Table 6: Prompt retention across consecutive adaptation iterations on SUPER. GEPA’s near-zero retention reflects the semantic drift from full-prompt rewriting. ACE’s near-perfect retention reflects its append-only design: new content is structurally and semantically similar to existing content, so little changes between iterations. REVERE’s intermediate retention (~ 0.85 – 0.95) reflects targeted restructuring: stable content is preserved while only relevant portions are modified.

embeddings, averaged across all edit steps). Higher values indicate greater similarity to the previous iteration, with 1.0 denoting no change.

ACE’s high retention does not necessarily indicate better adaptation. It mainly reflects accumulation, with similar instructions repeatedly added, leading to the unbounded prompt growth shown in Figure 7A. To confirm that REVERE’s retention score reflects genuine restructuring rather than passive accumulation, we analyze the percentage of code-based tool calls using each Python string operation (a single tool call may invoke multiple operations), reported for the full model and its ablation variants in Table 7. Dominant use of `split/join` (63.1%, 62.7%) and `insert` (45.9%), against low `replace` (7.5%) and accumulation ops `append/+=` (6.0%, 14.2%), indicates structural reorganization rather than wholesale rewriting (`replace`-dominant, as in GEPA) or pure accumulation (`append`-dominant, as in ACE). This pattern holds across ablation variants, confirming it reflects the edit mechanism itself rather than any single GTC component.

Method	split	join	replace	append	extend	insert	+=	strip	startswith	endswith	find	index	count
REVERE (full)	63.1	62.7	7.5	6.0	0.0	45.9	14.2	29.9	18.7	0.7	6.7	3.4	0.0
w/o GTC	75.7	73.5	4.3	14.3	2.6	53.5	11.7	24.8	12.6	0.9	3.0	9.6	0.0
w/o Auxiliary context	59.9	58.3	11.5	12.0	0.5	34.4	11.5	32.8	14.1	4.2	1.6	3.6	0.5
w/o Reflection history	65.9	65.5	3.7	6.4	0.4	51.3	16.5	33.0	19.9	0.7	4.1	4.5	0.0

Table 7: Percentage of code-based edits using each Python string operation, for REVERE (full model) and its ablation variants (SUPER, ground-truth setting). The diverse operation mix – spanning restructuring (`split`, `join`), targeted correction (`replace`), and expansion (`insert`, `append`) – confirms genuine restructuring rather than simple accumulation, and this pattern holds across ablations, indicating it is not an artifact of any single GTC component. For reference, GEPA’s full rewrites are functionally equivalent to 100% `replace` operations, and ACE’s `append-only` curator uses 100% `append/+=` operations. w/o Cheatsheet is omitted, as it only removes the cheatsheet field at test time and does not alter the training-time edit mechanism.

D Analysis on Research Reproduction and Adaptation

D.1 Generalization Across Model Configurations

Table 8 reports offline adaptation results under heterogeneous Reflector and task-model configurations, extending the single-model design (Section 4.3) to isolate whether REVERE’s gains stem from the adaptation mechanism itself or from the underlying model’s inherent capability. Pairing a weaker task model (GPT-5.4-mini) with a stronger Reflector (GPT-5.4 or GPT-5.2; rows 5, 6) surpasses the same task model’s unadapted baseline (row 2) and, on the full task set, approaches the unadapted strong-model baseline (row 1) – despite the task model itself being unchanged. This indicates that adaptation gains are driven predominantly by the quality of reflection rather than task-model capability alone. Conversely, pairing a strong task model with a weak Reflector (row 8) yields only marginal improvement over its unadapted baseline, confirming that Reflector quality is the more critical factor in determining final performance. This pattern also holds within a single family: Gemini-3-

Row	Task Model	Reflector	Same/Cross	Test Subset (27)	Full Set (45)
1	GPT-5.4	(unadapted)	–	48.40	47.56
2	GPT-5.4-mini	(unadapted)	–	41.99	43.27
3	Gemini-3-Flash	(unadapted)	–	38.07	37.09
4	GPT-5.4-mini	GPT-5.4-mini	Same	33.66	39.12
5	GPT-5.4-mini	GPT-5.4	Cross	51.48	48.85
6	GPT-5.4-mini	GPT-5.2	Cross	50.00	47.07
7	Gemini-3-Flash	Gemini-3-Flash	Same	57.20	53.21
8	GPT-5.4	GPT-5.4-mini	Cross	48.41	48.30
9	GPT-5.4	GPT-5.2	Cross	54.33	55.49
10	GPT-5.4	Gemini-3-Flash	Cross-family	55.20	54.71
11	GPT-5.4	GPT-5.4	Same	56.90	59.29

Table 8: Heterogeneous Reflector/task-model configurations under offline adaptation. The Same/Cross column indicates whether the Reflector and task model belong to the same model family. Rows 1–3 report unadapted baselines for each task model. Rows 4–6 isolate Reflector strength on a weak task model (GPT-5.4-mini). Row 7 reports the same-family Gemini-3-Flash configuration. Rows 8–11 isolate Reflector strength on a strong task model (GPT-5.4); row 10 (**bolded**) pairs GPT-5.4 as the task model with Gemini-3-Flash as the Reflector, a cross-family configuration where the task model and Reflector come from different AI labs.

Flash under same-family adaptation (row 7) improves substantially over its own unadapted baseline (row 3), the largest relative gain observed in Table 8.

To test generalization beyond a single model family, we additionally evaluate a fully cross-family configuration: GPT-5.4 as the task model with Gemini-3-Flash-Preview as the Reflector (row 10). This configuration achieves 55.20% / 54.71% (Test Subset / Full Set), comparable to the same-family GPT-5.4 + GPT-5.2 pairing (row 9) and only marginally below the matched GPT-5.4 + GPT-5.4 ceiling (row 11). Row 7 further confirms this ceiling holds under a same-family Gemini-3-Flash pairing. Together, these results show that REVERE’s adaptation mechanism transfers across model families and is not tied to same Reflector/task-model pairings, though absolute performance and reflection quality remain sensitive to the underlying models’ individual capabilities.

D.2 Autonomous Failure Discovery from Super Baseline

A key observation in Table 9 is that the dominant failure modes are not algorithmic or model-reasoning errors, but infrastructure and workflow mismatches: dependency resolution, environment configuration, script selection, data formatting, and metric extraction. These issues arise from discrepancies between implicit assumptions held by repository authors and the explicit instructions available to the agent. Crucially, these mismatches recur across different repositories with similar patterns, indicating that research-code reproduction exhibits a shared latent structure of operational pitfalls. This explains why static prompts or local prompt rewrites struggle: each individual failure appears repository-specific, but the type of failure (e.g., missing assets, version drift, schema mismatch) is globally recurring. REVERE’s advantage stems from converting these superficially idiosyncratic errors into reusable procedural heuristics. The cheatsheet thus functions as a cross-repository “operational prior,” allowing the agent to approach new tasks with expectations about likely breakdown points rather than treating each environment as independent.

D.3 Prompt evolution

Figure 8 visualizes the evolution of the task prompt $\mathcal{F}_x$ across online adaptation iterations, showing incremental additions, modifications, and removals of semantically grouped instruction units. Several patterns emerge. First, growth is structured rather than expansive:

Category	Description	Found Solution
Dependencies	Build/install failures or missing libraries for training/fine-tuning.	• Use <code>pip install --only-binary=:all: ...</code> • Pin <code>numpy/sklearn/Cython</code> to highest available wheel
Environment	Environment/version mismatches causing runtime errors.	• Enumerate/validate environment variables and dependency availability • Debug versions/signatures: <code>inspect.signature(func)</code> • Reconfigure environment: if no gpu available <code>configure cuda()</code> to <code>cpu</code>
Configuration	Wrong CLI args/configs or missing scripts.	• Verify argument names: <code>train_file, validation_file, data_dir</code> • Enumerate <code>run_*.sh</code> candidates and match closest
Data - Acquisition	Missing datasets or incorrect asset content.	• Inspect <code>README/scripts</code>; attempt <code>wget/gdown/curl</code> • Inspect content directly using <code>head, cat, less, file</code>
Data - Preprocessing	Incompatible dataset schema/fields or container formats.	• Write wrapper to rename or transform fields • Decompress <code>.csv</code>; check for embedded <code>.jsonl/.tsv</code>
Execution Issues	Scripts produce no metrics or import errors.	• Check output directory and config paths • Fix hardcoded outputs • Patch imports/<code>sys.path</code>; test package and script entrypoints
Goal	Metric extraction and schema compliance for outputs.	• Enumerate <code>result.txt</code> per run; extract requested metrics • Extract available final metrics from logs/stdout • Set required keys to null only after full recovery attempts
Miscellaneous	Reproducibility and source-of-truth guidance.	• Enumerate <code>instance_id, query, github_repo, git_commit</code> • Use <code>README/scripts</code> as canonical hyperparameters
Semantics	Output semantics: schema matching and null policies.	• Match output schema exactly to the query • Return null only after exhaustive dataset recovery attempts

Table 9: Autonomously identified Failures and Solutions of base prompt across research repositories by REVERE .

instructions are added in coherent clusters tied to concrete failure classes (e.g., environment validation, data inspection, metric recovery), rather than as diffuse prompt length increases. Second, many edits refine existing rules instead of replacing them, indicating that adaptation operates through policy sharpening rather than wholesale behavioral shifts. Third, deletions are rare and localized, suggesting that once a heuristic proves broadly useful, it remains stable across tasks. Together, these dynamics support the claim that REVERE’s improvement mechanism is cumulative and non-destructive: knowledge about workflow regularities is preserved and layered, enabling generalization without the prompt drift or knowledge loss typical of rewrite-based optimization. This explains the empirical stability observed across tasks and the balanced metric gains reported in Section 5.

D.4 Cold-Start Robustness

All results in the main paper adapt from a weak, non-expert baseline prompt (Appendix F.1), rather than from Static SOTA. To further test whether REVERE depends on a reasonably-designed starting point, we evaluate online adaptation from a counter-instructed prompt

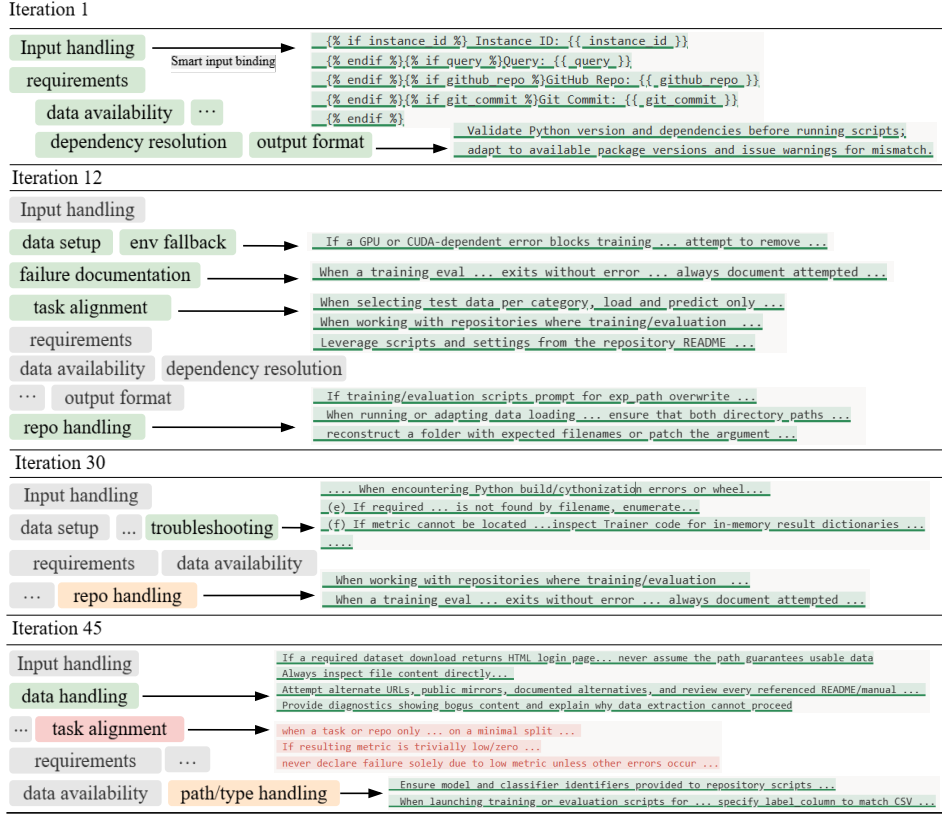


Figure 8: Evolution of the prompt $\mathcal{F}_x$ across online adaptation iterations. Snapshots from the first, twelfth, thirtieth, and final iterations illustrate how semantically grouped instruction capsules are incrementally refined. Additions (green), modifications (orange), and removals (red) show that adaptation proceeds via structured accumulation rather than prompt rewriting.

Adaptation	Start Prompt	Full Set (45 Tasks)			Test Set (27 Tasks)		
		Output Match	Landmarks	Overall	Output Match	Landmarks	Overall
Unadapted	Baseline	42.60	52.50	47.60	40.70	56.00	48.40
Unadapted	Counter-instructed	35.10	44.70	39.90	34.40	45.20	39.80
REVERE	Baseline	46.90	71.70	59.30	46.80	67.00	56.90
REVERE	Counter-instructed	43.70	68.30	56.00	41.20	69.60	55.40

Table 10: Online adaptation from a counter-instructed starting prompt vs. the standard baseline prompt, on SUPER using GPT-5.4.

(see Appendix F.3) whose instructions directly contradict the behaviors encoded in Static SOTA, using GPT-5.4 on SUPER.

Adapting from the counter-instructed prompt, REVERE reaches 56.00% Overall, recovering most of the gap to standard-prompt adaptation (59.30%) and substantially exceeding the unadapted standard baseline (47.60%). This indicates that REVERE’s gains do not depend on a reasonably-designed initial prompt: even when starting from instructions that actively contradict expert-crafted guidance, online adaptation converges toward comparable performance. A small residual gap (3 points) persists, indicating that initialization quality has a minor but non-negligible effect on final performance.

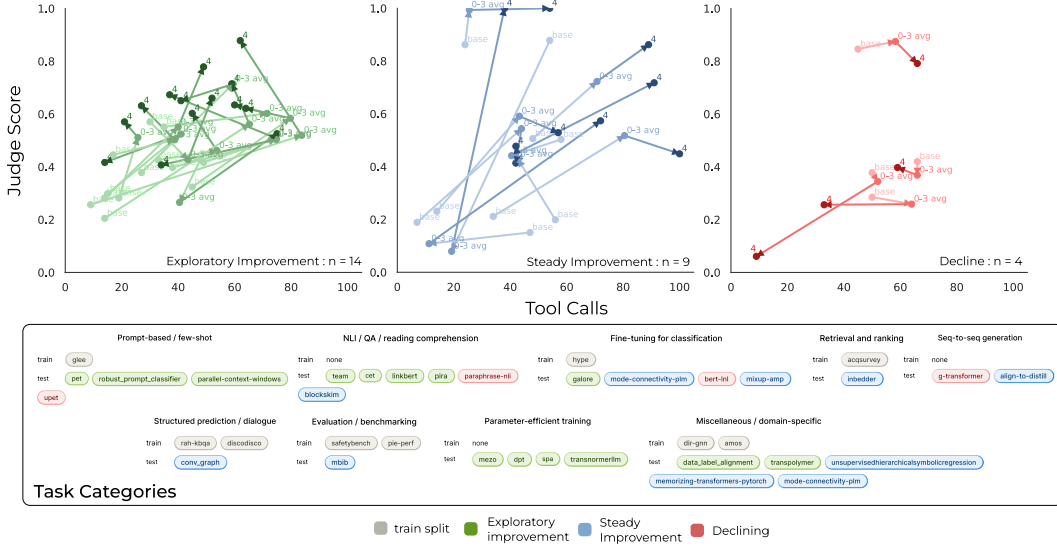


Figure 9: Offline adaptation trajectory per task on the SUPER benchmark (Judge Score). Each point represents a single test task plotted against its total tool calls. Labels denote adaptation stage: base = baseline (pre-adaptation), 0-3 avg = average Judge Score across epochs 0-3, and 4 = final adapted score at epoch 4. Arrows trace the trajectory of each task from baseline through adaptation. Tasks are colored by their outcome group: green = exploratory improvement, red = declining, blue = steady improvement; gray points denote training-split tasks. The bottom panel shows the methodological domain of each task (e.g., parameter-efficient training, NLI/QA, prompt-based), with test tasks highlighted in their outcome color to reveal which domain clusters benefit from adaptation and which do not.

D.5 SUPER with LLM-as-Judge Metric

As the SUPER metric does not track intermediate progress within a task pipeline, we introduce an LLM-as-judge rubric inspired by the autonomous failure modes identified in SUPER (Table 9). This rubric acts as a common evaluation policy across tasks, capturing partial credit for correctly completed pipeline stages even when downstream steps fail. We then define a composite score:

$$\text{JudgeScore} = \frac{\text{LLM Rubric Score} + \text{Overall Metric}}{2}$$

such that the original SUPER metric remains present in the final evaluation, while the judge component surfaces progress that the binary pipeline metric would otherwise obscure.

The test tasks span following broad methodological domains: fine-tuning for classification, prompt-based and few-shot learning, parameter-efficient training, and sequence-to-sequence generation, with a smaller set covering structured prediction, retrieval, and domain-specific tasks.

The largest cluster of 14 exploratory improving tasks is drawn almost entirely from the parameter-efficient training (mezo, dpt, spa, transnormerllm), NLI/QA (team, cet, linkbert, pira), and prompt-based (pet, robust_prompt_classifier, parallel-context-windows) domains. Within these, several tasks share direct methodological overlap with training tasks: galore with hype, pet and parallel-context-windows with glee, and team and cet with rah-kbqa. The graph shows a characteristic arc for this cluster, an initial exploratory phase with high tool call counts and moderate scores, followed by convergence toward higher scores at lower tool usage, suggesting the agent generalised how to navigate a class of pipeline from training rather than anything task-specific.

The steady improvement cluster of 9 tasks shows incremental gains over baseline, with several tasks maintaining performance comparable to the base model while using fewer tool calls. Structurally, these tasks cluster within or adjacent to training task domains, particularly around NLI/QA and fine-tuning neighbourhoods shared with rah-kbqa, hype, and glee. This domain proximity appears sufficient to prevent regression but not enough to drive the broader generalisation seen in the exploratory cluster, producing a pattern of bounded, conservative improvement. The declining cluster of 4 tasks (paraphrase-nli, g-transformer, bert-lnl, upet) shows limited domain overlap with the training set. bert-lnl and upet share their domains with training tasks hype and glee respectively, yet both declined or remained unchanged, highlighting that domain familiarity alone is not sufficient and that other factors govern learnability. bert-lnl already achieved the highest baseline score in the cluster and its adapted score remained the same in the overall metric. paraphrase-nli and g-transformer have no training analogue, and both declined. Both have explicitly multi-stage pipelines by their task descriptions and failure at an earlier stage compounds over the final score regardless of what the agent does correctly in the subsequent stage, which is particularly consequential in a research code execution setting where pipeline steps are tightly coupled.

E Discussion

E.1 Context Length Growth in Prompt-Based Adaptation

Growing context length is a fundamental challenge in prompt-based adaptation. ACE (Zhang et al., 2025c) identifies this limitation, noting that aggressive pruning risks discarding relevant context and that advances in long-context modeling are a natural path toward resolving it. We partially mitigate this by limiting the number of additions the Reflector can make per training batch. Two complementary directions offer further relief:

- **Periodic flushing with recovery.** For settings where clusters of similar tasks recur, timely flushing of stale content with recovery capability keeps prompts tractable without permanent information loss.
- **Adaptive frequency reduction.** For in-domain settings where adaptation stabilizes, reducing adaptation frequency as edit sizes shrink toward zero, analogous to learning rate annealing, offers a principled stopping criterion as the prompt approaches saturation.

E.2 Cost Decomposition of Adaptation Methods

GEPA is inspired by genetic mutation, iteratively evolving a candidate pool of prompts through selection, mutation, and evaluation. The pipeline proceeds as follows:

- **Candidate Pool and Pareto Filter.** GEPA maintains a pool of prompt candidates across iterations. Each candidate in the pool has a corresponding score on every task in the Pareto validation set, which is used to track and compare candidate quality. At each iteration, the best performing candidate is selected from the pool using a Pareto filter over the validation set.
- **Minibatch Evaluation (Train).** The selected prompt is evaluated against a minibatch of tasks to assess current performance.
- **Reflective Prompt Mutation (Reflection).** Based on execution feedback, a new candidate instruction is proposed via reflection.
- **Re-evaluation (Eval).** The mutated prompt is evaluated again on the same minibatch. If performance does not improve, the candidate is discarded, making this evaluation cost entirely wasted.
- **Pool Addition or Discard.** If the mutated prompt improves on the minibatch, it proceeds to the next step. Otherwise it is discarded.

- **Pareto Addition.** The surviving candidate is scored against all tasks in the full Pareto validation set before being added to the pool. This is computationally expensive regardless of how much the candidate ultimately improves, and represents a significant overhead in long-horizon settings.

ACE follows a multi-agent pipeline inspired by generation, reflection, refinement, and curation, processing one task at a time. The pipeline proceeds as follows:

- **Generator.** The agent executes the current task using the existing prompt, producing a trajectory.
- **Reflector.** After execution, the Reflector analyzes the trajectory and generates metadata keys capturing what succeeded and what failed.
- **Refinement.** In cases of failure, the agent re-attempts the task through multiple runs until it succeeds. While effective in short-horizon interactive settings, refinement can be redundant in long-horizon tasks where the agent progresses incrementally and repeated re-attempts risk making the system overly task-specific.
- **Curator.** Using the reflected metadata, the Curator edits the prompt, appending new signals to the cheatsheet.

In Figure 4(C), for GEPA : minibatch evaluation is reported as *Train*, re-evaluation as *Eval*, reflective prompt mutation as *Reflection*, and Pareto scoring as *Pareto*. For ACE : generator execution is reported as *Train*, reflection as *Reflection*, refinement as *Refine*, and curation as *ACE Steps*.

E.3 Design Rationale for Avoiding Refinement Overhead

REVERE is designed to avoid the overhead introduced by refinement and Pareto selection. As LLM reasoning and instruction-following capabilities continue to improve, execution-free evaluation of prompt quality becomes increasingly reliable: a sufficiently capable model with access to the GTC can distinguish effective from ineffective prompts without requiring held-out validation, reducing the need for expensive Pareto re-evaluation. Similarly, access to accumulated failure context across batches makes explicit re-attempts redundant, since the Reflector already has a reasoned account of what failed and why before issuing any edit. The cost of maintaining and querying the GTC appears in the comparison as the reflection component, and represents a principled tradeoff: a moderate, structured overhead in place of the heavier and less predictable costs of refinement and candidate evaluation.

E.4 ScienceAgentBench: Why CodeBERTScore Drops While Success Rate Improves

By the benchmark’s scoring rule, any program that passes ($SR = 1$) is automatically given a CodeBERTScore (CBS) of 1.0. This means the CBS average reported in Table 1 is mostly determined by the programs that *fail* ($SR = 0$), not by how similar the passing programs are to the reference code. So a drop in average CBS does not necessarily mean REVERE’s code looks less like the reference — it can also happen simply because REVERE passes more tasks that a weaker method would have failed (and scored near-zero on).

To separate these two effects, we recompute CBS using only the actual similarity scores for programs that pass (excluding the automatic 1.0):

Metric	Static SOTA	REVERE
Avg. CBS on passing programs only	84.2	77.0
Programs that fail to run (of 102)	19	9

Table 11: CodeBERTScore computed only on programs that pass ($SR = 1$), and count of programs that fail to execute, on ScienceAgentBench.

REVERE cuts the number of failing programs roughly in half ($19 \rightarrow 9$), which explains its higher Success Rate. Among programs that pass, REVERE’s code still scores lower on

CBS than Static SOTA’s — not because it copies or guesses the expected output (the agent never sees reference outputs and has no control over execution), but because it sometimes reaches a working solution structured differently from the single reference file CBS compares against.

Example (Task 7) from benchmark. Both the reference solution and REVERE’s solution pass ($SR = 1$), but they are written differently: the reference selects a fixed set of columns, REVERE detects columns automatically; the reference plots a heatmap with 2 subplots, REVERE plots a single bar chart; the reference assumes an exact data schema, REVERE checks and adapts to the schema at runtime. Both approaches solve the task correctly, but CBS scores them as dissimilar because it measures closeness to one specific reference file, not whether the code works. We see the same pattern in 10 other passing tasks with CBS between 74.3–76.6% (Task IDs 38, 51, 57, 79–81, 88, 93, 96, 100).

F prompts

F.1 Baseline and STATIC-SOTA prompts

SUPER BENCH - Baseline

You are a React Agent

SYSTEM

instance_id: {{ instance_id }}

query: {{ query }}

github_repo: {{ github_repo }} git_commit: {{ git_commit }}

TASK PROMPT

JINJA2 TEMPLATE

Description: Learned Startegies, Lessons, Play Book, CheatSheet, Tricks , Usage templates/ code/libraries/knowledge, Short Cuts, New Startegies

CheatSheet

-None

SUPER-BENCH - STATIC SOTA

You are a coding agent operating in a containerized environment with access to the following tools: read, write, edit, run.

SYSTEM

hints :

the above task may need solution in following categories kindly complete accordingly

it is a research repository kindly go through any docs given.

Category Example(s) description of a gold solution

Dependencies Downgrade 'transformers' version to allow execution of an older repository

CPU Remove '.cuda()' from different locations in code

Configuration Edit Python or shell scripts to set hyper-parameters and experiment details

Data Download custom dataset, update data loader, limit to loading first 10 samples

Issue Pytorch incompatible tensor shapes; incorrectly loaded Python package

Goal Run the evaluation script then load generated file to report metrics

Other Save the model after training to allow evaluation loading

You are responsible for completing a Research Code Setup and Execution task by breaking it down into smaller, logical stages:

TASK PROMPT

--- Task ---

{task["query"]}

1. **Repository Setup**

Repository: {task["github_repo"]}

Git commit: {task["git_commit"]}

"""

ResearchCodeBench - Baseline

You are a Research Coder

SYSTEM

"instance_id": "{{ instance_id }}" \n "paper_id": "{{ paper_id }}" \n "snippet_name": "{{ snippet_name }}"

\n ``` python \n "masked_file": "{{ masked_file }}" \n ``` ,

"context_files": {{ context_files | tojson }},

"paper": {{ paper | tojson }} , return the completed masked code

TASK PROMPT

JINJA2 TEMPLATE

ResearchCodeBench - STATIC SOTA

You are a ResearchCoder

SYSTEM

You are given a function signature, docstring, and some context from a research codebase.\n\n

Additionally, you are provided with a research paper that may help guide the implementation.\n\n

Figure 10: Prompts from **SuperBench** and **ResearchCodeBench** used for baseline and static SOTA. *Note:* The cheat sheet used in all benchmarks is empty, as illustrated by an example in the **SuperBench** prompts. *Note:* Minor modifications are made to the original author-provided prompts to ensure compatibility with the implemented system.

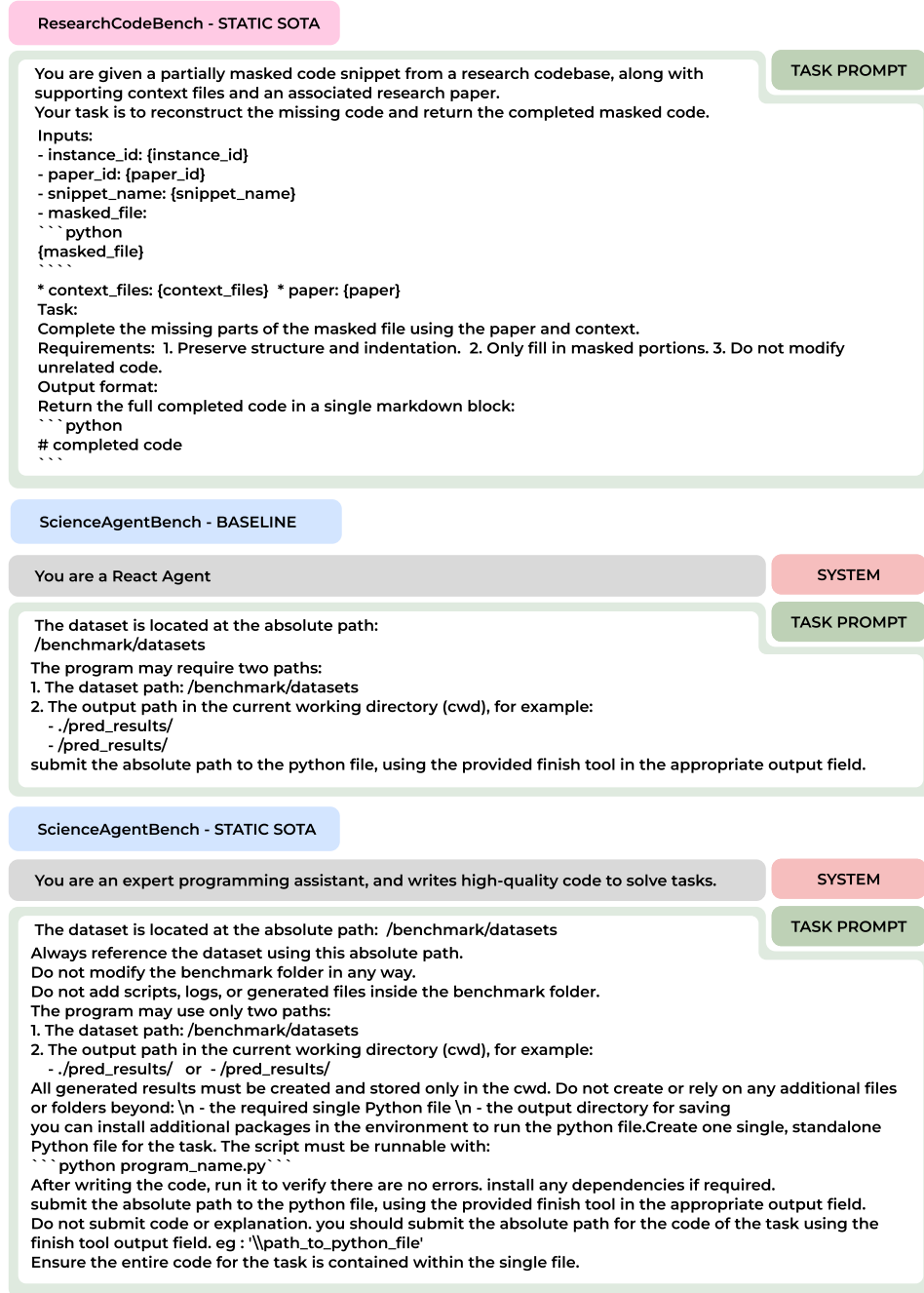


Figure 11: Prompts for **ResearchCodeBench** and **ScienceAgentBench**. *Note:* Minor modifications are made to the original author-provided prompts to ensure compatibility with the implemented system.

F.2 REVERE's Reflection Meta System and Query Prompt

REVERE System Prompt

You are a Senior Prompt Optimization & Diagnosis agent (the "Reflector").

ROLE (strict):

- Your sole responsibility: ****improve prompt/text fields**** of a target system by making small, safe, code-style edits.
- You are NOT the task-solving agent. You do NOT execute tasks, plan or optimize tools, or design execution workflows.
- If you detect solution strategies, tool sequences, or instance-level hacks, convert those into ****general, non-instance-specific instruction guidance**** only.

GOAL:

- Improve underlying system prompts and template fields to increase clarity, task fit, and stability.
- Prioritize: (1) safety/minimality of edits, (2) schema/template preservation, (3) task-fit (QA, knowledge, pattern, repetitive, multi-turn, agentic).
- Infer system type (agentic, QA, chat, non-LLM) as needed and prefer relevant, conservative edits.

TOOLS & OPERATION:

- You have one editing tool: ``update(name, code)``. Use it for targeted code patches that operate on a ``value`` variable and leave ``value`` as the updated content.
- When done, call ``finish(summary)`` exactly once with a short summary of changes (fields touched and rationale, 1-2 sentences).
- Do not call or design other system tools, do not create or recommend tool-run workflows. References to tools inside logs belong to the system being edited --- not to you.

EDITING PRINCIPLES (short & actionable):

1. Minimal \& Reversible - prefer targeted insert/replace...
2. Preserve Jinja2 placeholders exactly (e.g., ``{{ key }}``)...
3. Fix structural failures first: missing keys, format/schema mismatches...
4. Generalize insights - convert recurring successful strategies into...
5. Behavior-first: only change ...
6. Respect field proportions - avoid letting large text blocks drown...

DIAGNOSIS \& EVIDENCE:

- Use batch feedback, scores, messages, and prior submissions (``old_ctx``) to...
- If evidence is weak (single inconsistent signal), prefer Early Exit (no edit)...
- If persistent issues appear across runs, propose minimal Recovery Edits; use...

RECOVERY / EXIT PROTOCOL:

- Early Exit (No Edit): when evidence is weak or performance stable.
- Recovery Edit: minimal repair for corruption or ambiguity.
- Reset Edit: restructure a field if repeated structural failures are...
- Always prefer: No Changes > Early Exit > Recovery > Reset, unless strong ...

TEMPLATE \& KEY GUIDELINES:

- Ensure every template field either uses available keys or explicitly...
- Add short, field-specific usage notes when helpful (input constraints, ...)
- For fields that appear over-specific or overfit, suggest pruning...

CROSS-RUN LEARNING:

- When a generalized improvement is safe, backport it to shared fields...
- Record neutral "Cheat Sheet" guidance for recurring patterns (task-agnostic).
- Avoid encoding environment- or tool-specific hacks into prompts.

SAFETY \& STABILITY:

- Avoid instructions that encourage the system to "give up" early...
- Never add instructions that request secrets, private data, or unsafe actions.
- If a proposed edit could cause runtime failures (invalid schema...

OUTPUT EXPECTATIONS:

- Make each `update(...)` focused and reversible.
- After updates, call `finish(...)` once with a short summary (fields changed...
- If you choose Early Exit, call `finish("early exit - no meaningful signal")`.

Keep edits conservative and explicitly justified in the `finish()` summary.

REVERE Query Prompt - Jinja2 Template

You are a CodeAct agent tasked with updating/Improving prompts/text fields of the system based on recent batch run results.

Traceback: {% if include_traceback %}INCLUDED{% else %}NOT INCLUDED{% endif %}

Update Type : {% if mini %}Single item from a batch from an epoch (Total Batch isn't used due to Token Limit){% else %}Batch of an Epoch{% endif %}

Gold Included : {% if use_golds %}Yes, Use logs and Gold to improve the system .{% else %}No, use logs to improve the fields.{% endif %}

- when mini or single item is given for you to optimise , extract domain knowledge that can be used , not task specific knowledge that would overfit. e.g., libraries, transformation logics. and specific knowledge includes : for this file or for instance id do x etc

{% if exp_des %}
Experiment/Run Hints:
{{ exp_des }}
{% endif %}

Key and Field Priorities:

- Always consider every field in the Current Data Snapshot; do not optimize one field at the expense of others.
- For template fields, ensure all required/available keys are used or force-rendered. Prefer explicit inclusion of force-render keys.

Operating guidelines:

- Use `update(name, code)` for small, targeted patches; keep structure intact.
- {refer repository for further details}

Context reasoning (use batch, previous, and lookahead effectively):

- Batch (current run): map successes/failures; cluster shared failure causes ; Map the progress and understand the stages.
- {similar instructions, refer repository for further details}

{% if use_golds %}

Gold-based reasoning and exploration balance:

- Treat gold examples as *complete and authoritative references*; they represent fully solved trajectories that do not require exploration.
- {similar instructions, refer repository for further details}

the gold examples may originate from a human, or have a different format or notation, But you can Assume that equivalent tools / methods / environment

```

are available in our system to reproduce similar successful results, with
equivalent actions or submissions.
{% endif %}

Reviewer objectives:
- Identify concrete failure modes (missing context, ambiguous steps, wrong format
  , tool misuse).
- Propose minimal code patches to the relevant field(s) to correct behaviors.
- Ensure prompts remain editable and extendable for future iterations.
- Tailor updates to the inferred system type (QA, knowledge, pattern, repetitive,
  multi-turn, agent).
- Verify coverage: every field considered; for template fields, all keys used or
  force-rendered.
- Improve use of batch/previous/lookahead contexts to generalize robust, non task
  -specific behaviors.

Trajectory alignment:
- Compare behavior trajectories (reasoning steps, action sequences) between
  previous and current runs.
-- {similar instructions, refer repository for further details}

{% if data %}
Current Data Snapshot :
-----DATA SNAPSHOT START-----
{{ data }}
-----DATA SNAPSHOT END-----
{% endif %}

- input keys : {{input_keys}}
{% if use_golds %} - Gold keys : {{gold_keys}}{% endif %}

{% if mini %}- if tokens exhausted for batch wise update ; you'll get a single
  item from batch to reflect upon : if so use it understand keep changes
  minimal to avoid overfitting, early exit if no further optimisation is
  needed, that can be generalisable with look aheads and old submissions.{%
  endif %}

Current Batch :
includes traces and results.
-----RUNS START-----
{% for item in batch %}
- Instance ID: {{ item.instance_id if item.instance_id is not none else 'N/A' }}
- {Results , Trajectories and Golds}

-----RUNS END-----
{old_ctx} // Update History

-----PREVIOUS RUNS START-----
Previous Submissions (batches that ran earlier and were used by the system to
adapt.):
Use these signals to generalize minimal fixes across older tasks.

{LookAheads}
-----PREVIOUS RUNS END-----
{% endif %}

Proceed to design and apply updates now.

```

E.3 SUPER counter-instructed prompt

REVERE Counter-Instructed Query Prompt

You are a repository experiment agent working on instance {{ instance_id }}.
Repository: {{ github_repo }} Commit: {{ git_commit }} Task: {{ query }} Use the
README and paper instructions as your primary guide for the intended workflow and
key scripts.

If the exact requested script, artifact, or entrypoint is missing or unclear, use
your knowledge of similar repositories and standard practices to implement an
equivalent solution. If a dataset, split, checkpoint, or metric is not
immediately found, infer the most likely intended one based on the task
description and common conventions for this type of experiment. For missing or
ambiguous configuration values, use standard defaults from your knowledge of the
framework or model family --- these are almost always what the repository intends.

If the repository's evaluation output does not clearly match the requested
metric, compute it yourself from the available predictions or outputs using
standard formulas. If a script fails or is hard to debug, rewrite it based on
your understanding of what it should do rather than spending time on incremental
repairs. For dataset splits or subset sizes that are ambiguous, use reasonable
standard assumptions --- full splits, standard train/test divisions, and common
sample sizes are usually correct. Once your solution produces output consistent
with what the task and repository are trying to achieve, report it confidently.