SELF-ADAPTING AGENTS FOR AUTOMATING RESEARCH CODING WORKFLOWS

Balaji Dinesh Gangireddi
TCS Research, India
dinesh.gangireddi@tcs.com

Aniketh Garikaparthi
TCS Research, India
aniketh.garikaparthi@tcs.com

Manasi Patwardhan
TCS Research, India
manasi.patwardhan@tcs.com

Arman Cohan
Yale University, New Haven, CT, USA
arman.cohan@yale.edu

ABSTRACT

Existing prompt-optimization techniques use only local signals to update behavior, neglect broader, recurring patterns across tasks, leading to poor generalization; they often rely on full-prompt rewrites or unstructured merges, causing knowledge loss. These limitations are magnified in research-coding workflows, characterized by heterogeneous repositories, underspecified environments, and weak feedback; where reproducing results from public codebases is an established evaluation regime. We introduce Self-Adapting Research Engineer (SARE), a framework that learns from Global Training Context, cross-repository execution trajectories recognizes recurring failure modes, distills them into reusable heuristics, and performs targeted edits over configurable fields: the system prompt, a task-prompt template, and a cumulative cheatsheet preserving validated instructions while incrementally adding strategies. SARE, via this reflective prompt-optimization framework, improves performance over prior state-of-the-art human performance by 23.6% on SUPER, 3.5% on ResearchCodeBench and 7.1% on ScienceAgentBench across respective metrics, surpassing prior prompt-optimization technique.

1 INTRODUCTION

Large language models have advanced rapidly, and their evaluation has increasingly centered on coding Jimenez et al. (2024); White et al. (2025); Gauthier (2024) and reasoning performance Wang et al. (2024c); of America (2024); Chollet et al. (2026) as primary measures of model development. Within this broader emphasis on coding ability, a particularly practical and consequential evaluation setting is research-code reproduction (Starace et al., 2025; Bogin et al., 2024; Hua et al., 2025): assessing whether an agent can successfully run public research codebases, configure environments, acquire and preprocess data, execute full workflows, and reproduce reported results. While recent progress on short-horizon Yang et al. (2018); Rein et al. (2023); Hendrycks et al. (2021), well-specified coding tasksJimenez et al. (2024); Muennighoff et al. (2024) is promising Jiang et al. (2025); Wang et al. (2025); Yang et al. (2024), reliability degrades substantially in research-code reproduction Starace et al. (2025); Xiang et al. (2025), which places fundamentally different demands on a system Seo et al. (2025a); Lu et al. (2024). It requires agents to operate in realistic, open ended research environments, where correct solutions may admit multiple valid implementations. There is a need to coordinate long-horizon tasks under weak and delayed feedback, infer tacit undocumented repository conventions and assumptions, extract metrics from heterogeneous outputs, cope with evolving workflow conventions, and accumulate procedural knowledge across heterogeneous research ecosystems Trehan & Chopra (2026); Peng & Wang (2025); Siegel et al. (2024); Bogin et al. (2024); Hua et al. (2025).

Prior agentic systems such as paperbench Starace et al. (2025) or OpenHands Wang et al. (2025), which work for research reproducibility task, typically rely on static prompts or local, per-task adaptation mechanisms, which limits their ability to improve performance beyond narrowly specified tasks. Adapting agents to the evolving conventions of research repositories often requires learning

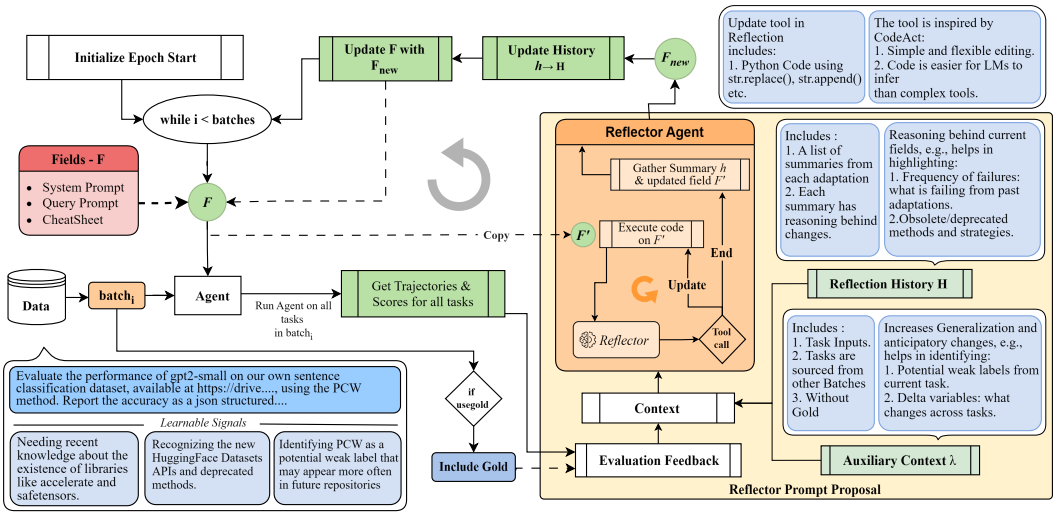

Figure 1: **The SARE Framework** employing an iterative optimization loop where a Reflector Agent dynamically adapts three editable prompt fields (System Prompt, Task Prompt, and Cheat-sheet) using our code-based edit mechanism, global training context and evaluation feedback

from feedback accumulated over multiple runs and failures. While fine-tuning Jung et al. (2026) or extended reasoning Guo et al. (2025) could, in principle be applicable; these approaches are not robust enough for research-code reproduction due to rapidly evolving conventions and procedural knowledge across repositories and the frequent absence of reliable ground-truth reasoning trajectories or reference executions that can serve as the supervision.

Recent work has explored prompt-level adaptation as a more immediate and lightweight alternative, Self-refinement approaches such as Shinn et al. (2023); Madaan et al. (2023); Zhang et al. (2025b). improve reasoning through iterative critique of model trajectories, but their improvements are typically reused only within the current problem instance rather than carried forward. Prompt-level adaptation methods, including Agrawal et al. (2025); Opsahl-Ong et al. (2024), extend optimization beyond single trajectories but still rely primarily on local evaluation signals, often derived from heuristic prompt sampling. This works well in short-horizon settings but becomes unreliable in long-horizon tasks, where intermediate decisions influence downstream outcomes in ways that may not be reflected in immediate or sampled feedback. As a result, crucial learning signals can be missed, and adaptation may overfit recent outcomes rather than generalizable patterns. Systems such as Suzgun et al. (2025); Zhang et al. (2025c) move towards accumulating reusable strategies and knowledge rather than relying purely on sampling or selection, and show improved performance in longer-horizon settings. However, they still depend on local evaluation signals (Shi et al., 2025) and operate over bounded contextual assemblies rather than an explicit persistent global memory shared across executions, which limits long-term knowledge retention.

Moreover, most prompt-adaptation frameworks update behavior through full prompt regeneration, increasing the risk of semantic drift and knowledge loss as prompts grow. Zhang et al. (2025c) instead emphasizes targeted edits, but its update mechanism introduces additional inference overhead due to complex metadata tracking and curator-style integration, making updates harder to scale efficiently. What is needed instead is an agent capable of learning from its own execution trajectories over time by identifying recurring failure modes, distilling them into reusable heuristics, and maintaining them within a persistent global context. Such an agent should apply targeted, non-destructive updates to prompts, plans, and tool-use strategies across tasks without gradient-based retraining. This form of test-time self-adaptation is essential for keeping pace with the scale and variability of real-world research codebases.

To address these gaps, we introduce SARE (Self-Adapting Research Engineer), a framework for building self-adapting agents tailored to research-coding workflows. SARE 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*, which preserves and aggregates experience across tasks and adaptations; and (3) *targeted code-based updates* via a Reflector module, which applies structured based edits to prompts and other optimizable fields. Together, these components allow SARE to progressively refine its behavior, reuse prior strategies, and update its reasoning without overfitting to specific task. 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 propose SARE a self-adapting research-coding agent that unifies prompt adaptation over configurable fields, a global context, and targeted code-based updates, enabling it to learn from execution trajectories across repositories without gradient-based training.
- We show that SARE improves overall performance on SUPER Bogin et al. (2024), a benchmark for setting up and executing tasks from research repositories, by 23.6%, on ResearchCodeBench Hua et al. (2025), a benchmark for translating machine learning research contributions into code, by 3.5% and ScienceCodeBench Chen et al. (2025) a benchmark on data-driven scientific discovery tasks, by 7.1% , over strong prompt optimization baselines.

Beyond these gains, our findings suggest two practical takeaways: (i) lightweight test-time self-adaptation can narrow the performance gap toward reliable research coding without retraining and with minimal overhead, and (ii) by revealing where agents succeed or fail on real research repositories, SARE offers a more grounded view of how LLM-based systems might assist in everyday scientific workflows.

2 Problem Definition

We formalize the adaptation of a research-coding agent as the problem of optimization over structured context fields that guide the agent's behavior.

System and Fields. Let Φ denote a research-coding agent that interacts with an environment through sequences of observations and actions. The agent is governed by a set of editable context fields. $\mathcal{F} = \{\mathcal{F}_s, \mathcal{F}_x, \mathcal{F}_c\}$, where The field $\mathcal{F}_s$ is the system prompt, which specifies the agent's overall behavior, tone, and high-level rules. The field $\mathcal{F}_x$ is the task prompt, which provides task-specific instructions and is instantiated with task-dependent inputs at runtime. The field $\mathcal{F}_c$ is the cheat-sheet, a persistent memory that stores useful strategies, common fixes, and domain-specific tips learned from previous tasks. These fields collectively parameterize how the agent behaves and act as high-level policy parameters for the system. By isolating these three editable components, we form a structured interface for adaptation and systematically improving performance.

Given a task input x, the agent induces a distribution over execution trajectories $\hat{\tau} = (o_1, a_1, \ldots, o_T, a_T)$,, where o_t are observations and a_t are actions , the trajectory induces an output $\hat{m}$ (e.g., generated code, a report, or task output), representing the agent's final result and implicitly determined by the trajectory. The behavior of the agent is modeled as a trajectory distribution conditioned on the fields:

$$P_\Phi(\hat{\tau} \mid x, \mathcal{F}) = \prod_{t=1}^{T} P_\Phi(a_t \mid o_{\leq t}, a_{<t}, x, \mathcal{F}),$$

where the execution trajectory $\hat{\tau}$ is jointly governed by the fixed agent Φ and the editable fields $\mathcal{F}$. **System-Level View** : Since the trajectory and final artifact are fully determined by the system under context $\mathcal{F}$, we equivalently denote the overall behavior at a system level as: $\hat{\tau}, \hat{m} = \Phi(x; \mathcal{F})$,
Tasks and Feedback. We consider tasks drawn from a distribution $(x, m, \tau) \sim \mathcal{T}$, where x is task inputs , m denotes evaluator metadata such as unit tests, target outputs, or grading rubrics and τ is the gold trajectory or logs optionally provided by the expert. Solving a task with configuration $\mathcal{F}$ yields an output whose quality is evaluated by a reward function r, defined as a combination of the metric function μ, which produces a scalar performance score, and optional natural-language feedback derived from reflection on trajectory and outputs.

Optimization Problem. Adaptation is performed by modifying the context fields rather than model weights. We seek optimal fields $\mathcal{F}^*$ that maximize expected performance over the task distribution. The learning problem is therefore :

$$\mathcal{F}^* = \arg\max_{\mathcal{F}}; \mathbb{E}_{(x,m,\tau)\sim\mathcal{T}}\big[r(\Phi(x;\mathcal{F}),[\tau;m])\big],$$

This formulation views learning as direct optimization of the structured context that guides agent trajectories, while the underlying model parameters remain fixed.

3 SELF-ADAPTING RESEARCH ENGINEER (SARE)

We introduce SARE a framework for adapting LLM agents to flexibly work with available starter code repositories and documentation using our novel prompt optimization technique. The overall architecture and optimization loop are illustrated in Figure 1 and the algorithm is detailed 1.

Our objective is to improve the performance on the target metrics m. SARE achieves this by executing tasks with the configured fields $\mathcal{F}$ and adapting them based on observed failures and evaluation feedback. This directly corresponds to the first core component of SARE prompt adaptation over configurable fields. The remaining two components govern how information across tasks is accumulated and how updates are applied: (1) Global Training Context (Section 3.1), and (2) Reflection and Update Mechanism (Section 3.2).

Algorithm 1 SARE: Adaptation Process

Input:
Agentic system Φ, Editable context fields $\mathcal{F}$
Training data $\mathcal{D} = \{(x, m, \tau)\}$, x: task description, m: target metrics, τ: gold trajectory
epochs N, gold flag δ_{gold}, Auxiliary task context λ of size l, Metric function μ
for epoch $= 1$ **to** N **do**
  Shuffle $\mathcal{D}$ and form batches $\{B\}$
  $\mathcal{H} \leftarrow \emptyset$ ▷ Initialize Reflection history
  for each batch B **do**
    $S \leftarrow \emptyset$ ▷ Initialize Eval Step Context
    for all $(x, m, \tau) \in B$ **do**
      $(\hat{\tau}, \hat{m}) \leftarrow \Phi(x, \mathcal{F})$
      $s \leftarrow \mu(\hat{m}, m)$
      if $\delta_{\text{gold}} = 1$ **then**
        $S \leftarrow S \cup \{(x, \tau, s, \hat{\tau})\}$
      else
        $S \leftarrow S \cup \{(x, \hat{\tau}, s)\}$
      end if
    end for
    $\lambda \leftarrow \{x \in \mathcal{D} - B\}$; $|\lambda| = l$
    $(\mathcal{F}', h) \leftarrow \text{REFLECTOR}(\mathcal{F}, S, \mathcal{H}, \lambda)$
    $\mathcal{H} \leftarrow \mathcal{H} \cup \{h\}$; $\mathcal{F} \leftarrow \mathcal{F}'$
  end for
end for
Output: Adapted agent Φ

Algorithm 2 Reflector Agent Module

Input:
Fields $\mathcal{F} = \{\mathcal{F}_s, \mathcal{F}_x, \mathcal{F}_c\}$
History buffer $\mathcal{H}$, Evaluation step context S, Auxiliary task context λ
LLM $\mathcal{L}$
Action Space:
$f' \leftarrow \text{EDIT}(f, p)$: edits $f \in \{\mathcal{F}_s, \mathcal{F}_t, \mathcal{F}_c\}$ using python program p
FINISH(h): terminates the loop by with summary h
$\mathcal{F}' \leftarrow \mathcal{F}$
$A \leftarrow [\,]$ ▷ Stores actions
repeat
  $(a, c) \leftarrow \mathcal{L}(S, \mathcal{H}, \lambda, \mathcal{F}', A)$ ▷ action a $\in$ {EDIT,FINISH} and c is input to a
  if $a =$ EDIT **then**
    $(f, p) \leftarrow c$
    $f' \leftarrow Execute(p, f)$
    $\mathcal{F}' \leftarrow f'$ ▷ Updating chosen field
    $A \leftarrow A + (a, c)$
  end if
until $a =$ FINISH
$h \leftarrow c$
Output: $(\mathcal{F}', h)$

Using a dataset ($\mathcal{D}$) consisting of task descriptions and their respective task inputs, we create batches and process each batch through the reflective loop for learning correct strategies. For each task (x) in a training batch b, the agentic system (ϕ) generates a task output $\hat{m}$ and a trajectory ($\hat{\tau}$) consisting of actions and observations. The predicted output is evaluated against the target metrics m using the metric function μ , yielding a task-level score s. The task description, predicted trajectory, target metrics, and resulting score together form the Evaluation Step Context (S), which captures all batch-local information used for subsequent reflection and adaptation. Depending on whether gold supervision $\delta_{gold}$is allowed, the corresponding gold trajectory (τ) may be included or omitted in S.

In parallel, additional context is constructed from the Auxiliary Context λ (refer to section 3.1), the editable prompts $\mathcal{F}$, and a reflection history $\mathcal{H}$ containing summaries of prior updates. The Reflector receives $(\mathcal{F}, S, \mathcal{H}, \lambda)$ as input and is tasked with reflecting, critiquing, and reasoning to iteratively adapt the fields $\mathcal{F}$. As detailed in Algorithm 2, the Reflector operates as an agent that repeatedly selects any of the target field f such that $f \in \{\mathcal{F}_s, \mathcal{F}_t, \mathcal{F}_c\}$ and emits executable program p to edit the textual content of f it. This process continues until the Reflector determines that no further edits are required. The module then returns the updated fields $\mathcal{F}'$ along with a concise reflection summary h, which is appended to the reflection history $\mathcal{H}$. Across epochs, the cheat-sheet $\mathcal{F}_c$ progressively accumulates reusable strategies and domain-specific knowledge, while the task prompt $\mathcal{F}_t$ is refined to encode improved task interpretation and execution policies. In parallel, the system prompt $\mathcal{F}_s$ is gradually updated to reflect more effective high-level behaviors and constraints, enabling sustained performance improvements over time. SARE improves performance through batch reflection and through the use of global training context and targeted updates to the agent's fields.

3.1 Global Training Context

SARE maintains a Global Training Context that aggregates information across tasks and training iterations, enabling adaptation beyond local feedback via three complementary signals:

1. **Auxiliary Context (λ)**: This consists of a small set of task descriptions and inputs of size l, drawn preferentially from unseen training tasks. When no such tasks remain, λ is sampled from randomly shuffled past training tasks. By exposing the Reflector to tasks beyond the current batch, this context encourages anticipatory adaptation and generalization, rather than purely reactive updates driven by immediate feedback.

2. **Cumulative CheatSheet ($\mathcal{F}_c$)**: The CheatSheet is a continually updated collection of concise, domain-specific strategies. These strategies are recorded in natural language by the reflector and used directly during task execution, which keeps the memory lightweight. Initialized empty, it grows over time as reusable insights from past failures are distilled into short heuristics and actionable reminders rather than full trajectories or detailed rationales.

3. **Reflection History ($\mathcal{H}$)**: The Reflection History is a record of prior reflection summaries, where each entry $h \in \mathcal{H}$ captures the rationale and outcome of a completed adaptation step. Unlike the CheatSheet, which supports task execution, $\mathcal{H}$ supports the Reflector by enabling it to reason over earlier updates. This helps prevent contradictory adaptation, where successive edits conflict with prior changes due to short-term or noisy feedback. By preserving high-level justifications for past updates, the Reflection History promotes stable, incremental adaptation across batches and reduces the risk that an erroneous or hallucinated reflection introduces inconsistencies into subsequent edits.

Together, these components provide a comprehensive learning strategy for the Reflector: Auxiliary Context prevents batch overfitting, the CheatSheet provides reusable hints, and Reflection History supports long-term coherence. Equipped with these signals, the Reflector can make informed concrete field updates. We now describe this update mechanism.

3.2 Reflection and Update Mechanism

A central component of SARE is the Reflector (Algorithm 2), a unified agent responsible for both reflection and field updates (Prompt in Appendix E). This design ensures that each adaptation remains consistent with prior edits and maintains a coherent view of the evolving system state. In contrast, multi-agent approaches that separate reflection from editing Hu et al. (2025); Zhang et al. (2025c) introduce hand-off boundaries, where an editing agent may misinterpret reflective intent or lack sufficient context, leading to incoherent or contradictory modifications. However, enabling a single agent to reason over multiple trajectories while performing precise field-level updates requires a controlled and low-overhead editing mechanism. To address this, we introduce a lightweight update tool inspired by CodeAct framework Wang et al. (2024a), which executes Python code generated by the Reflector to directly transform selected textual fields of $\mathcal{F}$. This interface supports structured, localized edits while preserving the Reflector's existing adaptation dynamics. This design provides two key advantages:

1 . Targeted, low-overhead updates: Edits are applied only to the relevant portions of $\mathcal{F}$via code-based transformations, allowing the Reflector to add, replace, or remove specific segments without

regenerating entire prompts. This localized editing reduces *semantic drift*, where full regeneration unintentionally alters unrelated instructions, and prevents overwriting stable, validated content.

2. Expressive, unconstrained modifications: Unlike template-based or rule-driven update schemes Opsahl-Ong et al. (2024); Zhang et al. (2025c) , code-based edits support arbitrary transformation logic over textual fields . By leveraging the Reflector's code-generation capability, the system enables precise updates without requiring complex tool schemas or restrictive editing APIs.

Together with the Global Training Context, this editing interface enables SARE to refine its prompts and cheatsheet iteratively while maintaining robustness across diverse research-coding tasks.

4 Experiment Setup

4.1 Benchmarks

We evaluate SARE on 3 challenging research-coding benchmarks: SUPER Bogin et al. (2024) and ResearchCodeBench Hua et al. (2025), ScienceAgentBench Chen et al. (2025), which comprise of tasks involving real-world research repositories, and characterized by limited documentation , constrained computational budgets and weak feedback signals. The task assessment is performed with sparse, delayed evaluations as opposed to deterministic test cases or intermediate ground truths.

SUPER (Bogin et al., 2024) consists of 45 research-coding tasks that require agents to interactively set up, configure, and execute experiments from real research repositories. This *long-horizon* setting is is executed by a ReAct-style coding agent in a containerized environment. These tasks reflect realistic research workflows in which agents must reproduce reported results by initializing repositories, installing dependencies, resolving version conflicts, configuring experimental settings, and handling runtime issues. We use the *Expert* subset of SUPER, which contains expert-designed tasks and is therefore more challenging and representative of real-world scenarios. Agent performance is evaluated using the benchmark's standard metrics. *Output Match* requires reproduced results (e.g., accuracy, F1 score, or error rate) to match expert-reported outputs, while *Landmarks* measure the presence of expected indicators of correct progress in execution logs , with higher scores assigned when more expected signals are observed.

ResearchCodeBench (Hua et al., 2025) evaluates an LLM's ability to re-implement core methodologies proposed in research papers under a *single-shot* code generation setting. For each task, the model is provided with the paper and partially masked code files and is required to reconstruct the missing implementation in a single forward pass. Performance is measured by unit test pass rate, where the reconstructed code is executed against hidden tests and considered correct if it runs without errors. The benchmark comprises 212 tasks from 20 top-tier venues (e.g., ICLR, NeurIPS).

ScienceAgentBench (Chen et al., 2025) Evaluates language agents on data-driven scientific discovery tasks in an *interactive* code-generation setting. Each task requires the agent to produce a self-contained Python program that implements a core component of a scientific workflow, with a strong emphasis on machine learning–based methodologies. Unlike single-shot generation, the setting is interactive: the agent can iteratively execute generated code, observe runtime feedback, debug errors, and revise implementations until it reaches a satisfactory solution. The benchmark contains 102 tasks derived from 44 peer-reviewed publications across four scientific disciplines. All tasks standardize the target output format to an executable Python file.

Extension of Benchmarks for Self-adaptation we consider both *offline* and *online* adaptation settings. In the *offline* setting, the agent is allowed to adapt using a fixed set of training and validation tasks, while evaluation is performed on a held-out test set that remains unseen during adaptation. In the *online* setting, tasks are processed sequentially at test time, and the agent may accumulate updates across previously seen test instances. This setting mirrors real-world research workflows, where new tasks arrive over time and supervision is limited or unavailable. For each benchmark, we adopted a three-way data split: train, validation, and test across all experiments. Train and validation sets are used for adaptation; the test split is reserved exclusively for evaluation. We adopt 9/9/27, 34/34/144 and 20/20/62 split for training, validation, and testing, for SUPER, ResearchCodeBench and ScienceAgentBech, respectively. Unless otherwise specified, no task-level supervision from the test set is used in the offline setting.

Method	GT	SUPER Bench (%)				RCB (%)	ScienceAgentBench (%)	
		Submission	Landmarks	Output Match	Overall	Accuracy	SuccessRate	CodeBleu Score
		long-horizon				*single shot*	*interactive - ReAct*	
Baseline	–	55.6	24.7	9.3	29.9	27.8	20.5	67.5
Static sota	–	37.03	35.8	14.8	29.2	31.9	23.5	88.2
Offline Adaptation								
GEPA	✗	96.3 +59.27	0.0 -35.8	20.1 +5.3	38.8 +9.59	25.69 -6.21	24.5 +1.0	74.8 -13.4
GEPA	✓	37.0 -0.03	5.2 -30.6	1.9 -12.9	14.7 -14.51	16.10 -15.8	19.35 -4.15	65.6 -22.6
SARE	✗	63.0 +25.97	35.3 -0.5	17.96 +3.16	38.8 +9.54	30.55 -1.35	25.4 +1.9	78.8 -9.4
SARE	✓	88.9 +51.87	38.1 +2.3	31.4 +16.6	52.8 +23.59	35.41 +3.51	30.6 +7.1	83.3 -4.9

Table 1: Offline Adaptation Results; RCB: ResearchCodeBench; with (✓) and without (✗) availability of ground-truth (GT) to the Reflector, (-) indicate only Inference based results.

4.2 Baseline Methods

Baseline and SOTA Prompts: Our baseline system uses minimal, unoptimized prompts, referred to as 'baseline' which serves as a reference for isolating the effects of SARE's adaptation mechanisms. For the SUPER, we implement a ReAct agent Yao et al. (2023) equipped with code tools . Additionally, we report current state-of-the-art performance using the official prompts designed for the benchmark, as provided by the authors, denoted as 'static SOTA'. All results are evaluated using GPT-4.1 to ensure consistency and comparability across evaluations.

GEPA (Genetic-Pareto) (Agrawal et al., 2025) is a sample-efficient prompt optimizer that evolves prompts using natural-language reflection and Pareto-based genetic search. It analyzes execution traces (reasoning steps, tool actions, outputs), diagnoses failures, and generates candidate prompt updates. A Pareto frontier maintains a diverse set of high-performing prompts, improving robustness and avoiding local minima. We use GEPA's official research repository [1] for all implementations. Refer Appendix A.2 for detailed configuration and Appendix A for implementation details of SARE.

5 Results and Analysis

Offline adaptation results are presented in Table 1, showing results on both setting: with and without leveraging ground truth hints (ground truth expert logs) as feedback. SARE improves agent performance (over the baseline) across all metrics for all the Benchmarks. In the hint-free SUPER setting, SARE improves Submission and Output Match while maintaining Landmarks at a level comparable to the Static SOTA results, producing a higher overall score despite limited supervision, whereas for GEPA the submission metric spikes, but its Landmarks collapse to zero. This imbalance reflects GEPA's tendency toward overfitting recent execution traces, which results in brittle prompt rewrites that harm intermediate reasoning stages. SARE avoids this pitfall because its updates remain constrained by the Global Training Context. Counterintuitively, GEPA degrades further when provided with gold hints. Gold trajectories contain rich, task-specific reasoning signals that are not easily generalizable across instances. GEPA's discard-and-rewrite mechanism suppresses this information during prompt updates, resulting in the systematic loss of valuable supervision. In contrast, SARE explicitly preserves task- and domain-specific insights in a cumulative cheatsheet, aided by a global training context that helps in guiding what information should be retained, emphasized, or omitted in future prompts. This difference explains why SARE fully exploits gold supervision, while GEPA's performance deteriorates despite access to stronger feedback.

SARE responds positively to ground truth hint trajectories and improves performance on all benchmarks on all metrics as compared to the baseline as well as Static SOTA results (except for ScienceAgentBench the Code BLEU Score it the results are at par with the Static SOTA resutls). The absolute gains on ResearchCodeBench and ScienceAgentBench are smaller as compared to the SUPER. This is as expected as the tasks of these benchmarks are domain-knowledge-heavy and are single-turn with limited structural redundancy across tasks. Because performance is tightly constrained by pretrained model familiarity with research conventions and terminology, adaptation has

[1] GEPA : https://github.com/gepa-ai/gepa-artifact

Method	SUPER Bench (%)				RCB (%)	ScienceAgentBench (%)	
	Submission	Landmarks	Output Match	Overall	Accuracy	SuccessRate	CodeBlueScore
	long-horizon				*single shot*	*interactive - ReAct*	
Baseline	55.6	24.7	9.3	29.84	27.8	20.96	67.50
Baseline - all tasks	71.1	19.63	10.4	33.72	43.80	20.58	66.86
Online Adaptation							
SARE	77.8 +22.2	33.5 +8.8	17.6 +8.3	42.94 +13.10	35.40 +7.6	25.80 +4.84	69.61 +2.11
SARE - all tasks	75.6 +4.5	30.1 +10.4	18.7 +8.3	41.44 +7.72	45.00 +1.2	25.49 +4.91	71.68 +4.82

Table 2: Online Adaptation Results RCB: ResearchCodeBench, obtained without the use of ground-truth labels. We report performance on the subsets used for adaptation (27 tasks for SUPER and 144 tasks for ResearchCodeBench) and on the Complete task suites (45 and 212 tasks, respectively).

Method	SUPER Bench (%)			RCB(%)	ScienceAgentBench (%)	
	Landmarks	Output Match	Overall	Acc.	SuccessRate	CodeBlueScore
	long-horizon			*single shot*	*interactive - ReAct*	
SARE	31.4	38.1	34.7	35.4	30.6	83.3
SARE w / o Cheatsheet	16.7 -14.73	30.7 -7.42	23.7 -11.07	30.6 -4.86	20.9 -9.7	78.3 -5.0
SARE w / o Auxiliary context	24.1 -7.33	24.8 -13.29	24.4 -10.30	31.9 -3.47	25.8 -4.8	79.8 -3.5
SARE w / o Reflection history	9.3 -22.14	25.9 -12.17	17.6 -17.15	32.6 -2.78	22.5 -8.1	75.9 -7.4
SARE w / o Global Training Context	10.2 -21.22	21.7 -16.37	16.0 -18.79	27.1 -8.33	19.3 -11.3	77.3 -6.0

Table 3: Ablation on Offline Adaptation

less signal to exploit. Still, SARE consistently outperforms the baseline, indicating that adaptation provides benefits beyond pretraining even under strong knowledge constraints.

Table 2 shows that the same trends hold in the stricter online setting, where no ground-truth labels are available and each task is encountered only once. SARE improves for all benchmarks for all metrics, both on the adaptation subsets and across the full task suites. Although absolute performance is lower than in offline training, this gap is structurally expected. Online learning is incremental cannot revisit prior data, so accumulated loss from earlier tasks is preserved, lacks multi-epoch refinement and the strongest supervision signals. Despite this, SARE maintains consistent relative gains, indicating that its adaptation mechanism does not depend on hindsight access to labels but instead leverages recurring execution patterns observed during interaction.

The ablation results (Table 3) confirm that SARE'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 is central to adaptation. Excluding auxiliary context reduces overall performance, indicating that local task signals alone are insufficient without structured contextual guidance. The largest drops occur when Reflection History and Global Training Context are removed, highlighting that temporal continuity and global constraints stabilize learning by preventing contradictory or myopic updates. These results demonstrate that SARE's design is effective by construction: non-destructive updates, memory accumulation, and global context jointly enable steady performance growth. We provide a case study on SUPER illustrating prompt evolution and observed failure modes is provided in the appendix D.

6 Conclusion

We presented SARE a lightweight, unsupervised adaptation framework for LLM agents tackling multi-step, code-intensive research reproducibility tasks. SARE augments standard agent setups with a Global Training Context that aggregates reflection History, auxiliary context, and a cumulative cheatsheet, together with our reflection and update mechanism that issues precise, code-level edits to optimizable fields. This design supports both offline and more realistic online adaptation, mitigating context myopia from purely local updates while keeping the adaptation loop interpretable and easy to integrate into existing agents. SARE demonstrates consistent gains on three challenging research-coding benchmark settings: SUPER-Bench (long-horizon) , ResearchCodeBench (single-shot) and SciennceAgentBench (interactive). Future work includes applying SARE to other tool-rich domains beyond research coding, tightening credit assignment for long-horizon adaptations, and exploring curricula where agents repeatedly revisit and refine their evolving training context.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, 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.

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.

Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arc-agi-2: A new challenge for frontier ai reasoning systems, 2026. URL https://arxiv.org/abs/2505.11831.

Paul Gauthier. Aider polyglot benchmark. https://aider.chat/polyglot, 2024. Accessed: 2025-11-13.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Hanwei Xu, Honghui Ding, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jingchang Chen, Jingyang Yuan, Jinhao Tu, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaichao You, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingxu Zhou, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645 (8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Rushil Gupta, Jason Hartford, and Bang Liu. LLMs for experiment design in scientific domains: Are we there yet? In *ICML 2025 Generative AI and Biology (GenBio) Workshop*, 2025. URL https://openreview.net/forum?id=dIEeOwrmOe.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021. URL https://arxiv.org/abs/2009.03300.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2025. URL https://arxiv.org/abs/2408.08435.

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.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

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.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Yeonsung Jung, Trilok Padhi, Sina Shaham, Dipika Khullar, Joonhyun Jeong, Ninareh Mehrabi, and Eunho Yang. Co-evolving agents: Learning from failures as hard negatives, 2026. URL https://arxiv.org/abs/2511.22254.

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.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery, 2024. URL https://arxiv.org/abs/2408.06292.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

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, Harshit Surana, Dhruv Agarwal, Bhavana Dalvi Mishra, Abhijeetsingh Meena, Aryan Prakhar, Tirth Vora, Tushar Khot, Ashish Sabharwal, and Peter Clark. Discoverybench: Towards data-driven discovery with large language models, 2024. URL https://arxiv.org/abs/2407.01725.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models, 2024. URL https://arxiv.org/abs/2308.07124.

Mathematical Association of America. American invitational mathematics examination (aime). https://www.maa.org/math-competitions, 2024. Accessed: 2025.

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.

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.

Alberto Sánchez Pérez, Alaa Boukhary, Paolo Papotti, Luis Castejón Lozano, and Adam Elwood. An llm-based approach for insight generation in data analysis, 2025. URL https://arxiv.org/abs/2503.11664.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. Gpqa: A graduate-level google-proof q&a benchmark, 2023. URL https://arxiv.org/abs/2311.12022.

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.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2code: Automating code generation from scientific papers in machine learning, 2025a. URL https://arxiv.org/abs/2504.17192.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2code: Automating code generation from scientific papers in machine learning, 2025b. 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.

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.

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, 2024a. URL https://arxiv.org/abs/2402.01030.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024b. 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.

Zirui Wang, Mengzhou Xia, Luxi He, Howard Chen, Yitao Liu, Richard Zhu, Kaiqu Liang, Xindi Wu, Haotian Liu, Sadhika Malladi, Alexis Chevalier, Sanjeev Arora, and Danqi Chen. Charxiv: Charting gaps in realistic chart understanding in multimodal llms, 2024c. URL https://arxiv.org/abs/2406.18521.

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, Siddartha 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.

Hjalmar Wijk, Tao Roa Lin, Joel Becker, Sami Jawhar, Neev Parikh, Thomas Broadley, Lawrence Chan, Michael Chen, Joshua M Clymer, Jai Dhyani, Elena Ericheva, Katharyn Garcia, Brian Goodrich, Nikola Jurkovic, Megan Kinniment, Aron Lajko, Seraphina Nix, Lucas Jun Koba Sato, William Saunders, Maksym Taran, Ben West, and Elizabeth Barnes. RE-bench: Evaluating frontier AI r&d capabilities of language model agents against human experts. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=3rB0bVU6z6.

Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. Live-swe-agent: Can software engineering agents self-evolve on the fly?, 2025. URL https://arxiv.org/abs/2511.13646.

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.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering, 2018. URL https://arxiv.org/abs/1809.09600.

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.

Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. Survey on evaluation of llm-based agents, 2025. URL https://arxiv.org/abs/2503.16416.

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.

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/.

Category	Description	Found Solution
Dependencies	Build/install failures or missing libraries for training/fine-tuning.	• Use `pip install --only-binary=:all: ...` • Pin `numpy`/`sklearn`/`Cython` to highest available wheel
Environment	Environment/version mismatches causing runtime errors.	• Enumerate/validate environment variables and dependency availability • Debug versions/signatures: `inspect.signature(func)` • Reconfigure environment: if no gpu available configure `cuda()` to `cpu`
Configuration	Wrong CLI args/configs or missing scripts.	• Verify argument names: `train_file`, `validation_file`, `data_dir` • Enumerate `run_*.sh` candidates and match closest
Data – Acquisition	Missing datasets or incorrect asset content.	• Inspect README/scripts; attempt `wget`/`gdown`/`curl` • Inspect content directly using `head`, `cat`, `less`, `file`
Data – Preprocessing	Incompatible dataset schema/fields or container formats.	• Write wrapper to rename or transform fields • Decompress `.csv`; check for embedded `.jsonl`/`.tsv`
Execution Issues	Scripts produce no metrics or import errors.	• Check output directory and config paths • Fix hardcoded outputs • Patch imports/`sys.path`; test package and script entrypoints
Goal	Metric extraction and schema compliance for outputs.	• Enumerate `result.txt` 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 `instance_id`, query, `github_repo`, `git_commit` • Use README/scripts 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 4: Failures modes autonomously identified across research repositories by SARE.

A Experiment Details

A.1 Data

Benchmark	Cost Per Task	Task Count
Super-Expert	\$1–3	45 tasks
Setting up, and executing tasks from research repositories.		
ResearchCodeBench	$\sim$ \$1	212 tasks
Code completion by implementing methodology from the paper.		
ScienceAgentBench	$\sim$ \$1	102 tasks
Scientific coding tasks spanning multiple domains for data-driven discovery.		

Table 5: Benchmark datasets used in our evaluation, with task counts and approximate per-task cost.

A.2 Hyperparameters

For the SUPER benchmark, GEPA was configured using the official implementation with 32 optimization iterations. Early stopping was applied when no improved prompt was discovered for 20

consecutive iterations. a mini feedback evaluation set of three examples was used to estimate prompt quality, and the Pareto frontier size of 9. For SAR, we trained the system for five epochs with a batch size of three. A look-ahead window of six future examples was used to stabilize adaptation, and no separate validation set was employed—allowing all available training data to contribute directly to the update process.

For ResearchCodeBench, GEPA was allocated a budget of 600 metric evaluations, corresponding to approximately 20 optimization iterations. The same batch size settings used for SUPER were maintained for consistency. For SAR, the overall setup mirrored the configuration used for SUPER, with the exception of a reduced look-ahead window. This adjustment was necessary due to context-length limitations arising from the longer code/paper/context files in a single call segments present in ResearchCodeBench tasks.

A.3 HARDWARE

All methods including SAR and baselines : uses the Azure `gpt-4.1-2025-03-01-preview` model and all experiments are conducted in `standard_d16ads_v5 ( 16 vcpus, 64 GiB memory)`

A.4 CODE AND DATA AVAILABILITY

The codebase, experiment logs, prompts, and data necessary to reproduce the results are available at `https://anonymous.4open.science/r/sare-artifact-BB2F`.

B RELATED WORK

LLMs are increasingly used across the research experimentation lifecycle, including data analysis, data-driven discovery, experiment planning and generation, research reproduction, and implementation of novel ideas Pérez et al. (2025); Majumder et al. (2024); Zhao et al. (2025); Gupta et al. (2025); Seo et al. (2025b); Wijk et al. (2025). Many of these systems rely on LLM-powered agents Yehudai et al. (2025); Luo et al. (2025), ranging from ReAct/CodeAct-style frameworks that interleave reasoning and tool use AI Security Institute (2024); Wang et al. (2024b), to multi-agent research systems Lu et al. (2024); Schmidgall et al. (2025); Huang et al. (2024), and search-based approaches that optimize code and reasoning trajectories via tree or evolutionary search Zhang et al. (2025a); Jiang et al. (2025). In parallel, a complementary line of work improves LLM behavior through context engineering rather than weight updates, optimizing prompts, instructions, tool specifications, and memory. Classical prompt optimization treats prompts as tunable parameters using RL, gradient-free, or heuristic search methods Khattab et al. (2023), while newer approaches such as GEPA use reflective models and evolutionary strategies to refine prompts for compound LM programs Agrawal et al. (2025). Runtime-adaptive agents further modify their scaffolds and tooling on the fly Xia et al. (2025); Hu et al. (2025), and test-time context adaptation methods such as Dynamic Cheatsheet and ACE maintain persistent, evolving playbooks via generation and reflection Suzgun et al. (2025); Zhang et al. (2025c). However, these methods are typically evaluated in densely supervised, shorter-horizon settings, whereas research coding workflows are long-horizon, weakly supervised, and require context updates tightly grounded in repository structure, environments, and execution traces rather than only high-level natural language feedback.

C LIMITATIONS AND FUTURE WORK

While SARE demonstrates that structured, test-time self-adaptation can substantially improve agent reliability in research-coding workflows, we identified few limitations:

SARE relies on reflective calls and a growing Global Training Context; this introduces inference overhead and can run into context-window limits for long repositories and papers. Our evaluation spans two research-coding benchmarks and a focused set of baselines; broader domains and stronger agentic baselines would further validate generality.

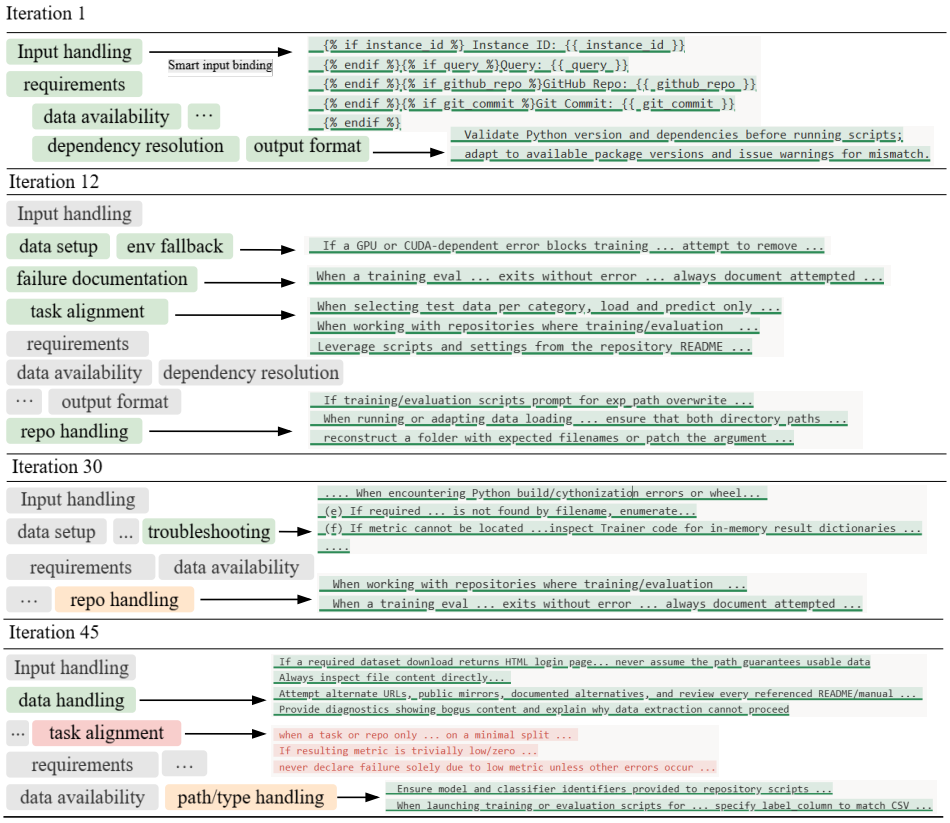

Figure 2: Evolution of the prompt $\mathcal{F}_t$ across online adaptation iterations. Snapshots from the first, twelfth, thirtieth, and final iterations illustrate how semantically grouped instruction capsules are incrementally refined. Additions (green), modifications (orange), and removals (red) show that adaptation proceeds via structured accumulation rather than prompt rewriting.

Future directions include: (i) distilling self-adaptation trajectories into parameter updates via lightweight finetuning or RL on the collected feedback to test whether model-level learning complements context edits; (ii) better credit assignment and compression/pruning of the Global Training Context to bound cost; and (iii) mechanisms for unlearning or de-emphasizing stale or harmful heuristics as distributions shift.

D SUPER BENCHMARK CASE STUDY

To better understand how SARE improves performance on long-horizon research-repository tasks, we analyze the recurring failure modes it encounters on the SUPER benchmark and the strategies it autonomously learns to address them. Rather than isolated bugs, these failures cluster into systematic categories that reflect the practical reality of research-code reproduction. Table 4 summarizes these categories and the corresponding remedies that SARE accumulates in its evolving cheatsheet and prompts.

D.1 FAILURE MODES

A key observation in Table 4 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. SARE'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.2 Prompt evolution

Figure 2 visualizes the evolution of the task prompt F_t 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: 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 SARE'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.

E SAR's Reflection Meta System and Query Prompt

SAR 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, 12 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.
```

SAR 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 a epoch
    (Total Batch isn't used due to Token Limit){\% else \%}Batch of
    an Epoch{\% endif \%}
Gold Inlcuded : {\% if use_golds \%}Yes, Use logs and Gold to
    imporve the system.{\% else \%}No, use logs to improve the
    feilds.{\% 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. example libraries, tranformation logics. and
     specific knowledge include : 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 undertand 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 exhasted 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.
```