TRACEPRISM: MACRO-TO-MICRO DIAGNOSIS FOR LONG-HORIZON TARGETED AGENT OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Optimizing long-horizon LLM agents from execution traces requires deciding which failures to analyze, where they originate, and which components to revise. Large trace collections contain recurring failures, while long executions can obscure the steps and component interactions underlying an observed failure. We present TracePrism, a macro-to-micro framework for budget-aware bottleneck localization and agent optimization. At the macro level, TracePrism groups failed trajectories by execution structure and failure information, then selects representative cases under a fixed trajectory budget for detailed diagnosis. At the micro level, it follows data and control dependencies backward from failure manifestations to construct diagnostic slices, localize plausible root-cause steps, and identify editable intervention targets. These diagnoses guide component-specific revisions that are checked against component responsibilities and applied jointly, leaving untargeted specifications unchanged. Across VeruSAGE-Bench, WebArena, and HotpotQA, TracePrism achieves the highest mean task performance among the evaluated methods. On VeruSAGE-Bench, it improves the success rate of an expert-designed agent from 46.2% to 58.8%, a gain of 12.6 percentage points. Scaling experiments on this benchmark further show a favorable trade-off between task performance and optimizer-side LLM cost as the failure corpus grows. The code is available at `https://anonymous.4open.science/r/TracePrism-8CC3`.

1 INTRODUCTION

Large language model (LLM) agents are increasingly capable of solving complex tasks through long-horizon interactions with tools, environments, memories, and other agents (Zaharia et al., 2024; Wang et al., 2024; Xi et al., 2023; Schick et al., 2023). Unlike single-turn prompting, these systems make sequences of interdependent decisions, including planning, tool use, state updates, subtask delegation, and verification. A single execution can therefore generate a long and heterogeneous trajectory containing dozens or even hundreds of steps. While such trajectories provide rich observations of agent behavior and a natural source of supervision for optimization (Pryzant et al., 2023; Shinn et al., 2023; Yang et al., 2024), they also raise a fundamental question: **given massive execution observations, how can we infer where an agent should be optimized?**

This question is particularly challenging when optimization is driven by failed executions. An optimizer may collect failures across diverse queries, environments, and execution paths. Although these failures can appear highly heterogeneous, many may arise from similar execution problems (Cemri et al., 2026; Barke et al., 2026). The same failure pattern may manifest through different tool calls, intermediate states, trajectory lengths, or error messages. Treating each failure independently can repeatedly spend a limited LLM budget analyzing redundant manifestations while overlooking distinct failure modes. The first challenge is therefore to **identify and cover distinct failure patterns** across a large population of failed trajectories.

Pattern discovery alone, however, does not reveal where a failure originated. A long-horizon trajectory may contain successful operations, repeated observations, auxiliary tool calls, and downstream consequences that obscure the observations informative for understanding the failure (Zhang et al., 2025a). In particular, the step at which a failure becomes observable may be substantially downstream from the step that introduced the problematic behavior. This makes **trajectory attribution**

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

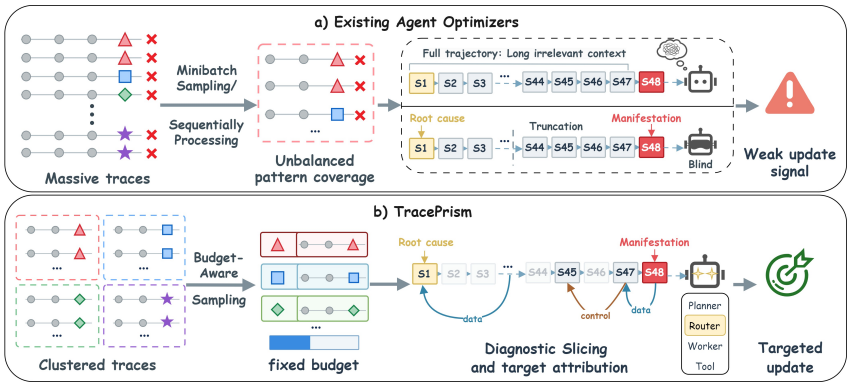

Figure 1: TracePrism narrows failure traces into actionable signals for targeted agent optimization.

challenging: failure-relevant information is latent in a long sequence of observations, and the optimizer must extract **diagnostic trace signals** that trace the observed failure toward informative upstream sources. Rather than aggressively compressing the trajectory, the goal is to isolate the parts of the execution that are most informative for diagnosis and optimization (Liu et al., 2023; Agrawal et al., 2026; Kang et al., 2025).

Finally, the identified failure does not necessarily imply that the entire agent should be optimized. Long-horizon agents often contain multiple interacting components, such as planners, tool selectors, state managers, verifiers, and specialized subagents (Zhang et al., 2025a). Modifying all of them enlarges the optimization search space and may disturb behaviors that are already effective. The diagnostic signals must therefore be connected back to the agent structure, allowing the optimizer to **attribute the failure to implicated components** and focus intervention on them.

Together, these challenges suggest that long-horizon agent optimization requires **progressively narrowing execution observations into actionable intervention targets**. We illustrate this perspective in Fig. 1. We propose **TracePrism**, which operationalizes this progressive narrowing over large-scale agent execution traces. It first identifies and covers distinct failure patterns, then performs trajectory attribution to extract diagnostic trace slices and trace failures toward informative upstream sources. These signals are subsequently used to identify implicated agent components and define a focused intervention scope, after which TracePrism performs targeted repair. By explicitly separating **failure discovery**, **trajectory attribution**, and **intervention targeting**, TracePrism allows the optimizer to allocate its limited computation and LLM budget to the execution patterns and agent components that are most informative for improvement. In this way, TracePrism turns large-scale execution traces from an increasingly costly optimization input into a structured source of actionable signals for agent improvement.

To assess the effectiveness and reliability of this selective design, we conduct evaluations across complementary settings. We first evaluate TracePrism on **VeruSAGE-Bench** (Yang et al., 2025), a domain-specific formal verification benchmark for Rust systems with complex, multi-agent execution trajectories. TracePrism outperforms state-of-the-art baselines, improving the success rate by an absolute **12.6%** (46.2% $\rightarrow$ 58.8%). We further validate the trajectory attribution capability of TracePrism on the **Who&When** benchmark (Zhang et al., 2025b), which provides annotations of responsible agents and decisive steps for agent failures. Finally, we evaluate TracePrism on **HotpotQA** (Yang et al., 2018) and **WebArena** (Zhou et al., 2024) to assess its generality.

2 Related Work

2.1 Automated Prompt Optimization

Automated Prompt Optimization (APO) employs LLMs to iteratively search for superior instructions. Pioneering works formulate this as a natural language search problem based on performance scores (Zhou et al., 2023; Yang et al., 2024). To handle complex workflows, recent frameworks leverage reflexive feedback (Shinn et al., 2023) or modular textual gradients (Pryzant et al., 2023; Khattab et al., 2024) to optimize multi-stage pipelines. Furthermore, evolutionary strategies have

been adopted to enhance exploration through population-based mutation and environmental feedback, such as EvoPrompt (Guo et al., 2025), GEPA (Agrawal et al., 2026), ACE (Zhang et al., 2026), and SCOPE (Pei et al., 2025). However, a critical limitation remains in *context management*. Existing methods typically rely on scalar rewards or truncated summaries to fit context windows. In long-horizon agentic tasks, this compression obscures deep structural dependencies, hindering the diagnosis of complex, multi-step failures.

2.2 Reflexive Agents and Harness Evolution

Distinct from APO, reflexive and self-evolving agents improve agent behavior through feedback-driven revision, ranging from inference-time refinement (Madaan et al., 2023; Sun et al., 2023) and verifier-based reinforcement (Jiang et al., 2025; Zhou et al., 2025) to code- and architecture-level evolution (Novikov et al., 2025; Sharma, 2025; Hu et al., 2025; Zhai et al., 2025). Recent harness-level systems, such as Hermes Agent Self-Evolution (Nous Research, 2026), further use optimizers like GEPA to evolve editable agent components. Meta-Harness searches over harness code through agentic access to historical implementations, execution traces, and evaluation scores (Lee et al., 2026). However, identifying the relevant execution evidence and mapping it to appropriate repair targets remains challenging in long-horizon agent systems. In contrast, TracePrism performs dependency-guided diagnosis before optimization, producing localized repair signals that can transfer to skills and other tunable harness components.

2.3 Agent Diagnostics and Credit Assignment

Credit assignment concerns attributing outcomes to specific decisions (Pignatelli et al., 2024), with reinforcement learning approaches including hindsight credit assignment and reward redistribution (Harutyunyan et al., 2019; Arjona-Medina et al., 2019). For LLM reasoning, min-form credit assignment (Cheng et al., 2025) and generative step-wise critiques (Xie et al., 2025) provide more targeted learning signals. Recent agent diagnostics methods localize failures through trajectory analysis and structured constraint checks (Zhang et al., 2025b; Bi et al., 2026; Barke et al., 2026). Building on these advances, TracePrism integrates failure diagnosis into budget-aware agent optimization through trace selection, dependency-guided slicing, and attribution to editable components.

3 The TracePrism Framework

3.1 Problem Formulation

We consider a long-horizon agent $\mathcal{A}$ composed of optimizable components $\mathcal{C} = \{c_1, \ldots, c_M\}$, such as system prompts, planner strategies, and subagents. Let $\mathcal{F} = \{\tau_1, \ldots, \tau_N\}$ be a corpus of failed execution trajectories. Applying LLM-based diagnosis to every trajectory in $\mathcal{F}$ can be computationally expensive. Our method therefore selects a subset $\mathcal{E} \subseteq \mathcal{F}$ for deeper analysis, subject to a trajectory budget K:

$$|\mathcal{E}| \leq K \ll N. \tag{1}$$

The selected trajectories are used to construct diagnostic trace slices and identify component-level intervention targets. These diagnostic representations guide targeted updates $\Delta_{\mathcal{C}}$ to the agent, yielding $\mathcal{A}' = \mathcal{A} \oplus \Delta_{\mathcal{C}}$. The goal is to improve $\mathcal{J}(\mathcal{A}')$, the expected task success rate on the target distribution, while limiting the number of trajectories requiring deep diagnostic analysis. The budget in Eq. (1) constrains the size of $\mathcal{E}$; it does not constrain the size of the initial failure corpus used for profiling and selection.

3.2 Phase 1: Failure Pattern Discovery

The first phase organizes the failure corpus into structurally constrained failure categories and selects a compact set of trajectories for deeper diagnosis. Selection is guided by execution patterns, available failure information, and corpus-level invocation statistics under the trajectory budget K.

Trace Corpus Profiling. Given the failure corpus $\mathcal{F}$, we use an LLM to generate a Python profiling program from a pilot trace and apply it to the corpus. For each trajectory, the program extracts the

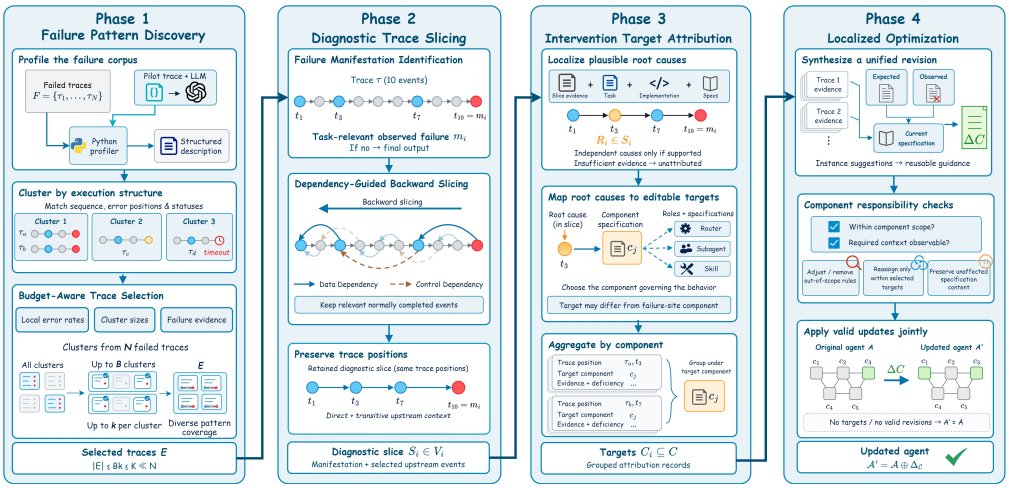

Figure 2: Overview of the **TracePrism** framework. The framework consists of four phases: Failure Pattern Discovery, Diagnostic Trace Slicing, Intervention Target Attribution, and Localized Optimization.

component invocation sequence and state transitions, the user query, component-level invocation outcomes, and task-level failure information available in the trajectory and associated task data. The extracted information is consolidated into textual trace descriptions, which are stored as files accessible to an LLM-based agent for clustering and selection.

Failure Pattern Clustering. The clustering agent reads the trace descriptions and organizes the trajectories into failure clusters $\mathcal{G} = \{G_1, \ldots, G_m\}$. It is instructed to group trajectories only when their component invocation sequences and the locations and statuses of their errors match. For example, an additional component invocation constitutes a different sequence, and a `timeout` is distinguished from a `failed` status; different error messages under the same `failed` status do not, by themselves, require separate structural groups. Within these structural constraints, the agent also considers the semantics of the queries and available task-level failure information when forming clusters. This grouping criterion is intended to keep distinctive execution paths from being absorbed into larger categories before representative selection. The resulting clusters therefore preserve execution-structure differences rather than necessarily merging all trajectories that share an underlying failure mechanism.

Budget-Aware Trace Selection. Let $\mathcal{C}_{\text{call}} \subseteq \mathcal{C}$ denote the components for which invocation outcomes are recorded. For each $c \in \mathcal{C}_{\text{call}}$ invoked at least once in $\mathcal{F}$, we compute its corpus-level, invocation-level local error rate:

$$r_c = \frac{\sum_{\tau \in \mathcal{F}} N_{\text{err}}(c, \tau)}{\sum_{\tau \in \mathcal{F}} N_{\text{call}}(c, \tau)}, \tag{2}$$

where $N_{\text{err}}(c, \tau)$ counts invocations with recorded error states or exceptions, and $N_{\text{call}}(c, \tau)$ counts all recorded invocations of component c in trajectory τ. This rate characterizes explicit invocation errors within the failure corpus rather than task-level failure probability or causal responsibility. The selection agent receives the corpus-level invocation statistics, cluster summaries, and cluster sizes, together with access to member trace descriptions. It is instructed to prioritize clusters associated with components exhibiting elevated local error rates while also considering cluster size and the available failure information. These statistics serve as auxiliary signals, and the agent makes the selection without a fixed numerical ranking function. Given a cluster budget B, it selects $\mathcal{G}_{\text{sel}} \subseteq \mathcal{G}$ such that $|\mathcal{G}_{\text{sel}}| \leq B$. For each selected cluster G, the agent is given the cluster summary and member file paths and instructed to select a subset $\mathcal{E}_G \subseteq G$ that covers distinct failure manifestations while minimizing redundant cases, subject to $|\mathcal{E}_G| \leq \min(k, |G|)$. The trajectories passed to subsequent phases are

$$\mathcal{E} = \bigcup_{G \in \mathcal{G}_{\text{sel}}} \mathcal{E}_G, \qquad |\mathcal{E}| \leq Bk. \tag{3}$$

We use fixed budgets B and k satisfying $Bk \leq K$, thereby respecting the global trajectory budget in Eq. (1).

3.3 PHASE 2: DIAGNOSTIC TRACE SLICING

Long-horizon trajectories interleave failure-relevant behavior with unrelated execution, and the behavior responsible for a failure may precede its observable manifestation by many steps. TracePrism uses dependency-guided backward slicing to construct a compact diagnostic context for subsequent attribution.

Failure Manifestation Identification. For each selected failure instance with trajectory τ_i, an LLM identifies a manifestation point m_i, the event at which the failure under investigation becomes observable. This identification draws on the failure information from Phase 1 and the original trajectory, prioritizing manifestations relevant to the final task failure. When no explicit execution error relevant to the final task failure is identified, m_i is the terminal or final-output event.

Dependency-Guided Backward Slicing. Anchored at m_i, the LLM traces data and control dependencies backward through the execution history. **Data dependencies** link events to earlier outputs they consume, such as plans, retrieved evidence, parameters, and execution results. **Control dependencies** link execution events to decisions governing their invocation, including tool and subagent selection. The LLM follows both direct and transitive dependencies to select upstream context, retaining normally completed events when their outputs or decisions influence the manifestation. Let V_i denote the messages and execution events in τ_i. The diagnostic slice $S_i \subseteq V_i$ consists of m_i and the selected upstream events, represented by their positions in the original trajectory. These positions allow Phase 3 to access the underlying execution evidence for root-cause localization and intervention target attribution. We evaluate the sensitivity of downstream optimization to additions and deletions in the constructed diagnostic slices in Section 4.4.

3.4 PHASE 3: INTERVENTION TARGET ATTRIBUTION

Given the diagnostic context isolated in Phase 2, TracePrism identifies plausible root-cause steps and maps the attributed behavior to actionable component-level intervention targets.

Root Cause Localization. For each diagnostic slice S_i, an LLM examines the inputs, outputs, and decisions at the selected trace positions, informed by the task, the agent's implementation, and the current component specifications. Attribution distinguishes among inappropriate component selection, deficiencies in a component's own execution, and errors inherited from upstream outputs, using component roles and the observed execution context to assess these possibilities. The localized steps $R_i \subseteq S_i$ are those whose decisions or outputs plausibly account for the observed failure, rather than necessarily the earliest erroneous event or the nearest predecessor of the manifestation. Attribution typically identifies one step, allowing multiple steps when the evidence supports independent contributing causes rather than successive propagation of the same error. Additional trajectory context may inform the diagnosis, but candidate root-cause steps remain restricted to S_i; instances with insufficient evidence are left unattributed.

Intervention Target Identification. The localized steps are mapped to an intervention target set $C_i \subseteq \mathcal{C}$ by identifying the components that govern the attributed behavior. When this association is unambiguous, the component identity recorded in the trajectory links the step to the corresponding editable specification. When multiple components, such as a subagent and its skills, jointly influence an event, the LLM examines their roles and specifications to determine which component should be modified. The intervention target may therefore differ from the component at which the failure became observable. Each localized step is associated with its target component and an evidence-grounded explanation of the behavioral deficiency. These records are aggregated by target component to guide the updates in Phase 4. We evaluate failure-step localization and responsible-agent identification on the Who&When benchmark in Section 4.4.

3.5 PHASE 4: LOCALIZED OPTIMIZATION

The final phase converts attributed failures into component-specific updates. TracePrism synthesizes revisions to the selected components, checks that they respect component responsibilities, and applies the resulting updates jointly.

Evidence-Guided Component Revision. For each intervention target for optimization, the optimizer examines its attributed failure instances using their trace locations and attribution rationales.

It contrasts the expected behavior with the observed inputs, outputs, and decisions, records the supporting evidence, and derives concrete component-level improvement suggestions. The LLM jointly considers these instance-level suggestions and the component's current specification to produce a unified update Δ_c. Depending on the target, revisions may clarify instructions or skill definitions, refine decision policies, or adjust configuration settings while preserving unaffected specification content. Suggestions are consolidated into reusable behavioral guidance rather than incorporated as isolated case-specific fixes. Instruction templates is available in the supplementary repository and examples of synthesized updates are provided in Appendix D.2.

Component Responsibility Checks. The optimizer reviews each proposed update against the component's responsibilities and available execution context. For example, a stateless executor should not be assigned behavior that depends on previous failed calls it cannot observe; such cross-call decisions belong to a component with access to the relevant history. Revisions that exceed a component's scope are removed or adjusted, and behavioral guidance may be reassigned only to another selected optimization target. If no intervention targets are identified or no valid revisions remain after these checks, the framework returns the original agent unchanged. Otherwise, the remaining revisions are combined into $\Delta_{\mathcal{C}}$ and applied jointly to obtain $\mathcal{A}' = \mathcal{A} \oplus \Delta_{\mathcal{C}}$, leaving the specifications of components outside the selected target set unchanged.

4 Experiments

4.1 Experimental Setup

Benchmarks and Data. We evaluate TracePrism on three complementary benchmarks for end-to-end agent optimization. **VeruSAGE-Bench** (Yang et al., 2025) serves as our primary testbed for long-horizon formal verification, covering five Rust projects: IronKV, Memory Allocator, Node Replication, NRKernel, and Storage. We use 397 optimization tasks and 106 held-out test tasks. Repeated executions of the unoptimized agent on the optimization tasks yield a fixed corpus of 453 complete failed trajectories; a single task may contribute multiple trajectories. To assess generality beyond formal verification, we use **WebArena** (Zhou et al., 2024) for web interaction across Shopping, CMS, Reddit, and GitLab, with 542 optimization tasks and 139 test tasks. We also use **HotpotQA** (Yang et al., 2018) for retrieval-augmented multi-hop question answering, with a random split of 150 optimization examples and 300 test examples. For all three benchmarks, optimization and test tasks are disjoint, and no separate validation set is used. As a complementary diagnostic evaluation, we use **Who&When** (Zhang et al., 2025b) to assess responsible-agent identification and failure-step localization in Section 4.4.

Base Agents. TracePrism optimizes the textual policies of components within each base-agent workflow. For VeruSAGE-Bench, we use the hierarchical router–executor framework of VeruSAGE (Yang et al., 2025), with 16 optimizable modules and `o4-mini` (OpenAI, 2025) as the backbone. Each task is subject to a limit of 20 repair attempts or 20 minutes. For WebArena, we use the original Chain-of-Thought web-navigation agent with one tunable instruction. For HotpotQA, we use a DSPy-based (Khattab et al., 2024) four-stage reasoning workflow with four optimizable modules. Both use `GPT-4o` (OpenAI, 2024). Detailed architectures and optimizable components are provided in Appendix A.

Baselines. We compare against the unoptimized Base Agent and three categories of methods. *Static strategies* include Naive Few-shot (Brown et al., 2020), which uses correct demonstrations constructed from reference solutions or answers in the optimization set, and Failure-Aware RAG (Lewis et al., 2021), which retrieves failure cases using BM25 (Robertson et al., 2009) and supplies them as context during task execution without learning a new policy. *Automated optimizers* include TextGrad (Yuksekgonul et al., 2024) and GEPA (Agrawal et al., 2026). TextGrad is omitted from the VeruSAGE-Bench main comparison because its full optimization runs were not completed due to high computational cost. *Context management baselines* include Summary-based and Retrieval-based optimization. They construct optimization context through summarization and relevance-based step retrieval, respectively. Both process the full failure corpus without Phase 1 selection, use the same downstream policy-revision procedure as TracePrism, and restrict updates to components where failures manifest rather than applying Phase 3 intervention-target attribution.

Table 1: End-to-end optimization performance on VeruSAGE-Bench, WebArena, and HotpotQA. We report per-project and overall Success Rate (SR) for VeruSAGE-Bench, overall SR for WebArena, and Exact Match (EM) for HotpotQA. All scores are reported as mean ± standard deviation over three seeds. Values in parentheses denote absolute percentage-point gains over the Base Agent. IR, MA, NO, NR, and ST denote IronKV, Memory Allocator, Node Replication, NRKernel, and Storage, respectively. Best results are highlighted in bold. TextGrad is omitted from the VeruSAGE-Bench main comparison because its full optimization runs were not completed due to high computational cost.

Setting	VeruSAGE-Bench					
	IR	MA	NO	NR	ST	**Overall**
Base Agent	44.4 ±4.8	66.7 ±0.0	83.3 ±5.8	28.5 ±7.5	48.7 ±11.8	46.2 ±4.3
Naive Few-shot	47.2 ±8.7	64.8 ±3.2	83.3 ±5.8	33.3 ±7.8	43.6 ±11.8	47.8 ±7.3 **(+1.6)**
Failure-Aware RAG	43.1 ±4.8	63.0 ±3.2	73.3 ±5.8	30.1 ±1.4	46.2 ±7.7	44.7 ±2.0 **(-1.5)**
Summary-based Selection	48.6 ±8.7	64.8 ±3.2	66.7 ±5.8	33.3 ±5.1	35.9 ±8.9	45.6 ±1.4 **(-0.6)**
Retrieval-based Selection	44.4 ±6.4	63.0 ±3.2	76.7 ±15.3	31.7 ±8.4	38.5 ±0.0	45.0 ±3.3 **(-1.2)**
GEPA	50.0 ±4.2	66.7 ±5.6	80.0 ±0.0	31.7 ±4.2	38.5 ±7.7	47.2 ±1.9 **(+1.0)**
TracePrism	**59.7** ±2.4	**83.3** ±5.6	**100.0** ±0.0	**37.4** ±5.1	**59.0** ±4.4	**58.8** ±1.4 **(+12.6)**

Setting	WebArena					HotpotQA
	Shopping	CMS	Reddit	GitLab	**Overall**	
Base Agent	19.6 ±3.0	10.8 ±0.0	4.5 ±4.6	5.7 ±1.4	10.8 ±1.4	22.6 ±12.6
Naive Few-shot	23.9 ±1.4	14.4 ±1.6	9.1 ±0.0	4.0 ±2.8	13.2 ±0.9 **(+2.4)**	50.9 ±7.9
Failure-Aware RAG	20.5 ±2.6	13.5 ±2.7	6.0 ±2.7	6.5 ±1.4	12.2 ±2.2 **(+1.4)**	51.9 ±5.0
Summary-based Selection	25.6 ±0.0	17.1 ±5.6	4.5 ±4.6	9.0 ±3.7	15.1 ±0.0 **(+4.3)**	58.0 ±10.4
Retrieval-based Selection	17.9 ±0.0	12.6 ±4.1	10.6 ±2.6	5.7 ±5.7	11.8 ±2.5 **(+1.0)**	52.7 ±10.1
TextGrad	22.2 ±7.8	**18.0** ±3.1	10.6 ±2.6	4.9 ±4.3	14.1 ±2.8 **(+3.3)**	64.8 ±2.5
GEPA	21.4 ±3.0	16.2 ±4.7	13.6 ±12.1	5.7 ±3.8	14.1 ±2.1 **(+3.3)**	57.7 ±11.3
TracePrism	**29.9** ±1.5	**18.0** ±1.6	**33.4** ±5.3	**12.2** ±4.2	**22.0** ±1.8 **(+11.2)**	**68.2** ±0.76

Optimization Protocol. Within each benchmark, trace-based methods share a fixed initial failure corpus, which is reused across independent runs. TracePrism executes a single pass through Phases 1–4. For all three benchmarks, Phase 1 uses $B = 5$ and $k = 5$, selecting at most five failure clusters and at most five trajectories per cluster, for a maximum of 25 trajectories passed to subsequent diagnosis. Optimizer-side LLM calls across applicable methods use `Claude Sonnet 4.5` with temperature 1.0. We do not enforce equal computational budgets across methods; each method follows its own optimization configuration.

Evaluation and Cost. We report Success Rate (SR) for VeruSAGE-Bench and WebArena, and Exact Match (EM) for HotpotQA. Overall SR is computed over all individual test tasks, rather than by taking an unweighted average across categories. For the main comparisons, we report mean performance over three independent runs. For optimization methods, each run independently produces an updated policy, which is evaluated once on the fixed test set. We report optimizer-side LLM cost in USD, estimated from input and output token usage under the corresponding API prices. This cost excludes initial trajectory collection and final test evaluation.

4.2 Main Results

Performance on the primary benchmark. As shown in Table 1, TracePrism achieves an overall SR of 58.8% on VeruSAGE-Bench, exceeding the Base Agent and the strongest evaluated baseline, Naive Few-shot, by 12.6 and 11.0 percentage points, respectively. It also achieves the highest mean SR on all five projects, with gains of 8.9–16.7 percentage points over the Base Agent. These project-level results show that the overall improvement is accompanied by gains across multiple Rust systems, although the remaining failures, particularly on NRKernel, indicate substantial room for further improvement.

Generality across tasks and agent structures. On WebArena, TracePrism achieves 22.0% overall SR, exceeding the Base Agent and the strongest baseline, Summary-based Selection, by 11.2 and

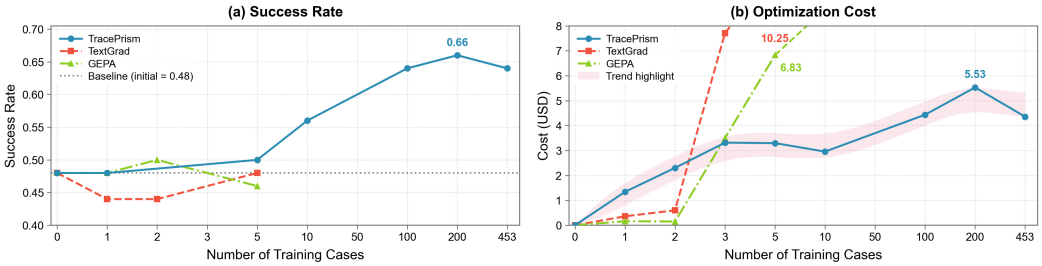

Figure 3: Success rate and optimizer-side LLM cost on VeruSAGE-Bench as the available failure corpus grows from 1 to 453 trajectories. Evaluation uses a fixed 50-task subset of the main test set. Each reported method–scale setting uses one optimization run followed by one evaluation. TracePrism retains a fixed selection budget of $B = 5$ and $k = 5$, while all compared methods receive the same input trajectory subset at each evaluated scale.

6.9 percentage points, respectively. On HotpotQA, it achieves 68.2% EM, surpassing the strongest baseline, TextGrad, by 3.4 percentage points. These results support its applicability beyond the hierarchical VeruSAGE system to a single-instruction web agent and a four-component retrieval-based reasoning workflow.

4.3 Optimization Scalability and Cost Analysis

We evaluate how optimization performance and cost change as the available failure corpus grows on VeruSAGE-Bench. The number of input trajectories ranges from 1 to 453, where each trajectory represents a complete failed execution rather than a distinct task. At each scale, the compared methods receive the same trajectory subset. Evaluation uses a fixed subset of 50 tasks drawn from the test set, with one optimization run and one subsequent evaluation for each method–scale setting. For TracePrism, we keep $B = 5$ and $k = 5$ throughout, so at most $\min(N, 25)$ trajectories are selected for deeper diagnosis from an input corpus of size N. Costs follow the optimizer-side LLM accounting defined in the experimental setup.

Figure 3 shows that TracePrism maintains a favorable balance between success rate and optimizer-side cost as more failure trajectories become available. Full-trace optimization with TextGrad becomes increasingly expensive at the evaluated scales, while TracePrism limits the number of trajectories subjected to deeper diagnosis and subsequent repair. The comparison with GEPA further highlights the importance of considering success rate and optimization cost jointly, rather than judging optimization methods solely by their final task performance.

This scaling behavior is consistent with the distinction between access to a large failure corpus and exhaustive diagnosis of that corpus. A larger input pool makes more failure instances available for pattern discovery and representative selection, while the budget for deeper diagnosis remains fixed. This does not imply constant total optimization cost: Phase 1 still profiles and organizes the input corpus, and its LLM calls are included in the reported cost. The experiment therefore characterizes the empirical cost–performance trade-off of the complete pipeline under increasing data availability; the following phase-wise analysis examines the contributions of its individual diagnostic stages.

4.4 Phase-wise Analysis

The VeruSAGE-Bench analyses below use a fixed pool of 10 complete failed trajectories and the fixed 50-task test subset used in Section 4.3, with one optimization-and-evaluation run per setting.

Phase 1: Failure Selection. We compare the selection strategy of TracePrism with processing all available trajectories and random selection. Table 2 reports 56% SR at an optimizer-side cost of $2.96 for TracePrism, versus 46% at $8.45 without selection and 52% at $2.16 with random selection. Representative selection improves SR over both controls, although random sampling is cheaper.

Phase 2: Diagnostic-Context Sensitivity. We perturb the fixed Phase 2 slices of six selected trajectories before rerunning attribution and optimization. At rates of 10% and 25% of each slice's

Table 2: Selection and joint diagnostic ablations.

Setting	SR (%)	Cost ($)
TracePrism	**56**	**2.96**
Phase 1: Failure selection		
w/o Selection	46	8.45
Random Selection	52	**2.16**
Phases 2–3: Joint ablation		
Current-context Repair	54	2.88
Full-trace Repair	54	5.93

Table 3: Diagnostic-slice perturbations.

Setting	SR (%)
Base Agent	48
TracePrism (unperturbed)	**56**
Remove 10% Steps	54
Remove 25% Steps	50
Add 10% Steps	**56**
Add 25% Steps	**56**

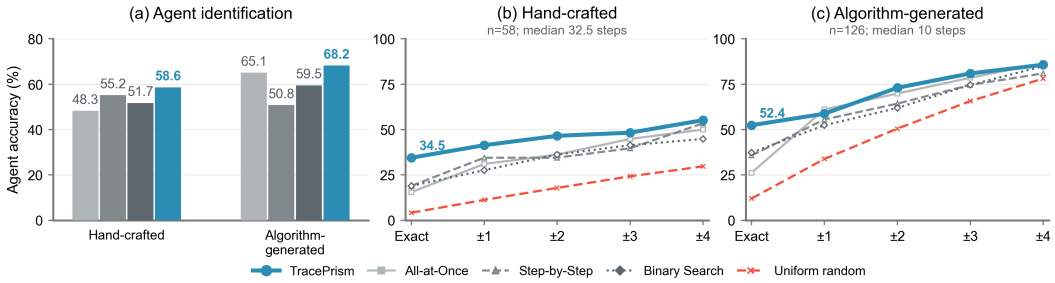

Figure 4: Failure attribution on Who&When. (a) Accuracy of identifying the failing agent. (b,c) Exact and within-$\pm k$-step failure localization. Uniform random selects one step per trajectory.

original step count, we randomly add unselected steps from the same trajectory or remove selected steps, potentially including the manifestation point. Removal yields 54% and 50% SR, respectively, whereas both addition conditions retain 56% (Table 3). This tests downstream context sensitivity rather than the slicing algorithm's robustness to noisy input trajectories.

Phase 3: Attribution Quality. We evaluate Phases 2–3 on all Hand-Crafted and Algorithm-Generated Who&When trajectories (Zhang et al., 2025b), without Phase 1 selection or Phase 4 updates. We compare with All-at-Once, Step-by-Step, and Binary Search; all LLM-based methods use `gpt-5.6-luna`. Task reference answers are provided during inference, while human agent/step annotations are reserved for scoring. Each method performs one complete attribution run per trajectory, returning one agent–step prediction. Agent accuracy measures responsible-agent identification; Exact and tolerance-based localization are scored using the predicted step alone. TracePrism achieves the highest Agent and Exact accuracy in both groups (Figure 4).

Joint Analysis of Phases 2 and 3. Current-context Repair and Full-trace Repair bypass slicing and target attribution, using local or full-trajectory context to update only components where failures manifest. Both reach 54% SR, costing $2.88 and $5.93, respectively (Table 2). The complete pipeline achieves 56% at $2.96, with cost close to Current and approximately half that of Full. These comparisons concern the combined pipeline, not either phase's isolated contribution.

5 Conclusions

We presented TracePrism, a macro-to-micro diagnosis framework for targeted optimization of long-horizon LLM agents. It combines budget-aware failure selection, dependency-guided trace slicing, and intervention-target attribution to guide component-specific revisions. Across VeruSAGE-Bench, WebArena, and HotpotQA, TracePrism achieves the highest mean task performance among the evaluated methods. Scaling experiments on VeruSAGE-Bench show a favorable trade-off between task performance and optimizer-side LLM cost. Attribution results on Who&When and joint ablations provide complementary evidence for the diagnostic pipeline. Together, these findings suggest that effective trace-driven optimization benefits from connecting representative failures to actionable component-level interventions, rather than defaulting to the components where failures manifest. Future work could extend this approach to iterative optimization and evaluate its applicability to a broader range of agent harnesses.

AI USE STATEMENT

We used generative AI tools to suggest edits improving the grammar, clarity, and readability of this manuscript. The authors reviewed and revised the suggestions to preserve the intended technical meaning and take full responsibility for the final content.

REPRODUCIBILITY STATEMENT

Appendix A details the base-agent workflows and optimizable components, while Appendix B provides implementation details. The anonymized supplementary repository contains the implementation, full phase prompts, agent skills, benchmark traces, and setup instructions. These materials support independent inspection and reproduction of the method and experiments.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pp. 8479–8565, 2026.

Anthropic. Claude Agent SDK. `https://github.com/anthropics/claude-agent-sdk-python`, 2025. Python SDK for building agents with Claude Code capabilities. Accessed: 2026-05-22.

Jose A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. *RUDDER: return decomposition for delayed rewards*. Curran Associates Inc., Red Hook, NY, USA, 2019.

Shraddha Barke, Arnav Goyal, Alind Khare, Avaljot Singh, Suman Nath, and Chetan Bansal. AgentRx: Diagnosing AI agent failures from execution trajectories. *arXiv preprint arXiv:2602.02475*, 2026.

Jiayi Bi, Yanjie Gao, Yuanmin Xie, Liqun Li, Tianyin Xu, Fan Yang, and Mao Yang. Diagnosing with insights: Structured analysis of agent failures via behavioral abstractions. *arXiv preprint arXiv:2609.02371*, 2026.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL `https://arxiv.org/abs/2005.14165`.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent LLM systems fail? *Advances in Neural Information Processing Systems*, 38, 2026.

Jie Cheng, Gang Xiong, Ruixi Qiao, Lijun Li, Chao Guo, Junle Wang, Yisheng Lv, and Fei-Yue Wang. Stop summation: Min-form credit assignment is all process reward model needs for reasoning, 2025. URL `https://arxiv.org/abs/2504.15275`.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. EvoPrompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers, 2025. URL `https://arxiv.org/abs/2309.08532`.

Anna Harutyunyan, Will Dabney, Thomas Mesnard, Mohammad Azar, Bilal Piot, Nicolas Heess, Hado van Hasselt, Greg Wayne, Satinder Singh, Doina Precup, and Remi Munos. Hindsight credit assignment, 2019. URL `https://arxiv.org/abs/1912.02503`.

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

Yuhua Jiang, Yuwen Xiong, Yufeng Yuan, Chao Xin, Wenyuan Xu, Yu Yue, Qianchuan Zhao, and Lin Yan. PAG: Multi-turn reinforced LLM self-correction with policy as generative verifier, 2025. URL https://arxiv.org/abs/2506.10406.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A. Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. ACON: Optimizing context compression for long-horizon LLM agents, 2025. URL https://arxiv.org/abs/2510.00615.

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. 2024.

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

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks, 2021. URL https://arxiv.org/abs/2005.11401.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023. URL https://arxiv.org/abs/2307.03172.

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.

Nous Research. Hermes Agent: The agent that grows with you. https://github.com/NousResearch/hermes-agent, 2026. Accessed: 2026-05-25.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI. GPT-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

OpenAI. OpenAI o3 and o4-mini system card, 2025. URL https://openai.com/index/o3-o4-mini-system-card/. Accessed: 2026-07-07.

Zehua Pei, Hui-Ling Zhen, Shixiong Kai, Sinno Jialin Pan, Yunhe Wang, Mingxuan Yuan, and Bei Yu. Scope: Prompt evolution for enhancing agent effectiveness, 2025. URL https://arxiv.org/abs/2512.15374.

Eduardo Pignatelli, Johan Ferret, Matthieu Geist, Thomas Mesnard, Hado van Hasselt, Olivier Pietquin, and Laura Toni. A survey of temporal credit assignment in deep reinforcement learning, 2024. URL https://arxiv.org/abs/2312.01072.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search, 2023. URL https://arxiv.org/abs/2305.03495.

Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: BM25 and beyond. *Foundations and trends® in information retrieval*, 3(4):333–389, 2009.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023. URL https://arxiv.org/abs/2302.04761.

Asankhaya Sharma. OpenEvolve: Open-source implementation of AlphaEvolve. https://github.com/algorithmicsuperintelligence/openevolve, 2025. GitHub repository.

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.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. AdaPlanner: Adaptive planning from feedback with language models, 2023. URL https://arxiv.org/abs/2305.16653.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey, 2023. URL https://arxiv.org/abs/2309.07864.

Guofu Xie, Yunsheng Shi, Hongtao Tian, Ting Yao, and Xiao Zhang. CAPO: Towards enhancing llm reasoning through generative credit assignment, 2025. URL https://arxiv.org/abs/2508.02298.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2024. URL https://arxiv.org/abs/2309.03409.

Chenyuan Yang, Natalie Neamtu, Chris Hawblitzel, Jacob R. Lorch, and Shan Lu. VeruSAGE: A study of agent-based verification for rust systems, 2025. URL https://arxiv.org/abs/2512.18436.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 2369–2380, 2018.

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

Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound ai systems. https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/, 2024.

Yunpeng Zhai, Shuchang Tao, Cheng Chen, Anni Zou, Ziqian Chen, Qingxu Fu, Shinji Mai, Li Yu, Jiaji Deng, Zouying Cao, Zhaoyang Liu, Bolin Ding, and Jingren Zhou. AgentEvolver: Towards efficient self-evolving agent system, 2025. URL https://arxiv.org/abs/2511.10395.

Guibin Zhang, Junhao Wang, Junjie Chen, Wangchunshu Zhou, Kun Wang, and Shuicheng Yan. AgenTracer: Who is inducing failure in the LLM agentic systems? *arXiv preprint arXiv:2509.03312*, 2025a.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *International Conference on Learning Representations*, volume 2026, pp. 86069–86100, 2026.

Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems. In *Proceedings of the 42nd International Conference on Machine Learning*, ICML'25. JMLR.org, 2025b.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

Yifei Zhou, Sergey Levine, Jason Weston, Xian Li, and Sainbayar Sukhbaatar. Self-challenging language model agents, 2025. URL https://arxiv.org/abs/2506.01716.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers, 2023. URL https://arxiv.org/abs/2211.01910.

A AGENT IMPLEMENTATION DETAILS

We describe the base-agent workflows and their editable components. The set of components eligible for optimization is distinct from the intervention targets selected in a particular run: TracePrism updates only the components implicated by Phase 3, leaving the remaining specifications unchanged.

A.1 VERUSAGE AGENT WORKFLOW

For VeruSAGE-Bench, we use the hierarchical router–executor framework of VeruSAGE (Yang et al., 2025)[1], with `o4-mini` as the backbone. The workflow exposes 35 repair actions and contains 16 optimizable modules: a router and 15 specialized subagents. Repair actions and optimizable modules are different units; the number of available actions does not determine the number of editable components.

Verification-driven repair. Each task is processed through an iterative verification–repair loop:

1. The current program is submitted to the Verus verifier.
2. If verification fails, the verifier feedback is parsed to identify the reported errors.
3. The router selects a repair action based on the error information and the current proof context.
4. The selected subagent proposes a patch, which is applied before the next verification attempt.

The loop terminates when verification succeeds or either the 20-repair-attempt or 20-minute limit is reached. A task is counted as successful only if the resulting program passes both Verus verification and the cheat checker.

Specialized components and optimization scope. The subagents cover invariant repair, proof strategies such as case analysis and induction, prover-mode selection, compilation-error repair, and lemma selection. The router and specialized subagents form the candidate component set. Phase 3 determines which of their textual policies to revise from the attributed failures; targets are not fixed in advance or selected solely by invocation frequency. Phase 4 synthesizes component-specific revisions and applies the valid updates jointly.

A.2 WEBARENA AGENT WORKFLOW

For WebArena, we use the original Chain-of-Thought web-navigation agent (Zhou et al., 2024)[2], with `GPT-4o` as the backbone. This is a single-agent workflow with one optimizable instruction prompt. At each interaction step, the agent receives the task instruction, current URL, a textual accessibility tree of the page, and the previous action.

The agent reasons about the current observation and outputs a browser action, such as clicking an element, entering text, scrolling, navigating to a URL, or stopping with an answer. The browser executes the action and returns the next observation. Interaction continues until the agent stops or the configured interaction budget is exhausted.

TracePrism revises the agent's instruction prompt while retaining the underlying browser interface and workflow. Since only one component is editable, this setting evaluates optimization across a different task domain rather than the ability to distinguish among multiple component-level intervention targets.

A.3 HOTPOTQA AGENT WORKFLOW

Our HotpotQA agent is a DSPy-based four-module workflow following the setup used in GEPA (Khattab et al., 2024; Agrawal et al., 2026)[3], with `GPT-4o` as the backbone. The modules execute in the following order:

[1] `https://github.com/microsoft/verus-proof-synthesis`
[2] `https://github.com/web-arena-x/webarena`
[3] `https://github.com/gepa-ai/gepa-artifact`

1. **Summarize1:** Extracts relevant information from the question and first-hop retrieved passages to form an initial context.
2. **Create Query Hop2:** Uses the question and initial summary to formulate a second-hop query for missing evidence.
3. **Summarize2:** Integrates the second-hop passages with the prior context to consolidate evidence across retrieval rounds.
4. **Final Answer:** Produces the answer from the question and accumulated evidence. Evaluation scores the final answer using Exact Match.

All four module instructions are eligible for optimization, but only modules selected through Phase 3 attribution are updated. The retrieval backend and workflow structure are not part of this instruction-optimization scope.

B TracePrism Implementation Details

B.1 Execution and Artifact Handoff

TracePrism is implemented using the Claude Agent SDK (Anthropic, 2025) as a four-phase workflow: *Failure Pattern Discovery*, *Diagnostic Trace Slicing*, *Intervention Target Attribution*, and *Localized Optimization*. The phases exchange structured artifacts stored under `output`, preserving the links between selected trajectories, diagnostic evidence, attributed behavior, and editable specifications.

Trace inspection uses tools such as `get_json_structure`, `search_context_in_file`, and `read_trace_positions`. Events retain their original one-based positions and exact source locators. Events outside a numbered invocation sequence, such as a separately stored final answer, use an exact locator with a `null` position rather than an invented index. Relevant implementation files may clarify dependencies or component responsibilities, but a possible code path is not treated as evidence that it occurred in the recorded execution.

B.2 Phase-wise Implementation

Phase 1: Failure Pattern Discovery. The profiling agent creates or updates `trace_analyzer.py` and processes the failure corpus into per-trajectory `trace_description_xxx.json` files and a corpus-level `trace_analysis_report.md`. Each description preserves the complete invocation sequence, including repeated calls, together with error positions and statuses, state transitions, the query, and available task-level failure information; missing information is explicitly marked as unknown. Clustering uses only these descriptions and the report: Python groups exact structural signatures, and an LLM refines the groups semantically. Each trajectory belongs to exactly one cluster in `failure_clusters.json`. Budget-aware selection then records selected clusters, representative trace paths, effective budgets, and rationales in `representative_traces.json`. The default budgets are $B = 5$, $k = 5$, and $K = 25$, with no requirement to fill the quotas. Targeted reads of original traces are permitted during representative selection when additional evidence is needed.

Phase 2: Diagnostic Trace Slicing. The slicing agent identifies a manifestation point in each selected trajectory: an explicit error relevant to the task failure or, when no such error is identified, the terminal or final-output event. Cases lacking sufficient evidence to establish the failure or locate its manifestation are retained as insufficient-evidence records. Starting from the manifestation, the agent follows upstream data and control dependencies, distinguishing observed links from inferred ones and recording their supporting evidence. State changes guide inspection but do not determine eligibility: rejected calls can provide useful feedback, and normally completed calls can supply essential evidence. Repeated calls are not merged solely because shared state is unchanged. The output, `diagnostic_slicing_chain.json`, preserves original event positions and source locators, potentially branching dependencies, unresolved information, and a `constructed`, `partial`, or `insufficient_evidence` status.

Phase 3: Intervention Target Attribution. The attribution agent examines retained events to distinguish inappropriate selection, deficient execution, and inherited upstream errors, without as-

Table 4: Mean actual repair turns on VeruSAGE-Bench in one evaluation run over the 106 test tasks. Successful and unsuccessful tasks are both included.

Agent	IronKV	Memory Allocator	Node Replication	NRKernel	Storage
VeruSAGE (Base Agent)	12.04	9.17	7.80	14.83	13.00
VeruSAGE (TracePrism-optimized)	8.42	7.39	4.00	8.20	8.31

signing responsibility by default to a router or executor. Candidate root-cause steps remain restricted to the diagnostic slice, and cases with insufficient evidence remain unattributed. For supported step-level findings, the agent inspects the relevant prompts, skills, policies, or configurations to identify responsible editable specifications. `failure_attributions.json` retains every selected trace and distinguishes step localization from target mapping, whose status is `mapped`, `unresolved`, or `no_editable_target`. Verified mappings are aggregated into `intervention_targets.json`, with duplicate component–trace–source-locator references removed. Unattributed cases and unresolved mappings remain documented but are excluded from the supported target set.

Phase 4: Localized Optimization. For each supported target selected for revision, the optimizer examines all attributed instances and records the execution evidence, expected and observed behavior, specification gap, and actionable suggestions under `component_suggestions`. It consolidates these suggestions into a reusable component-specific update and saves the complete revised specification under `revised_components`, preserving unaffected content. Responsibility checks remove changes outside the component's scope; guidance may be reassigned only to another selected target whose responsibilities and available execution context support it. `component_mapping.json` records revised specifications, supporting steps, and unchanged or deferred targets. Validated revisions are applied jointly, while components outside the selected revision scope remain unchanged. If no supported targets or valid revisions remain, the original agent is retained.

B.3 Instruction Templates

The complete instruction templates for all four phases are available in our code repository.[4].

C Supplementary Experimental Results

C.1 Execution Efficiency on VeruSAGE-Bench

We complement SR with the number of outer-loop repair turns used by the base and optimized VeruSAGE agents. A *turn* is one repair attempt, rather than one trajectory event or subagent invocation. For most failure types, invoking a repair subagent counts as one turn. For assertion failures, `assertion_reasoning_pipeline` and its selected downstream subagent jointly count as one turn, although they correspond to two invocation steps in the trace.

For each task, we count the actual turns executed until success or termination. Failed or timed-out tasks are included using their actual counts, which may be below 20; they are not assigned a fixed maximum-turn penalty. Table 4 reports project-level means from one evaluation run on the same 106 test tasks used in the main experiment. These are single-run execution statistics, not averages over the three independent optimization runs reported in the main performance table.

The optimized agent uses fewer turns on average in all five projects, with the largest absolute reduction on NRKernel, from 14.83 to 8.20. Turn counts characterize outer-loop execution effort, not optimizer-side cost. Since termination can occur without success and different turns can involve different numbers of LLM calls, lower turn counts alone do not establish lower latency, lower token cost, or improved per-turn decision quality.

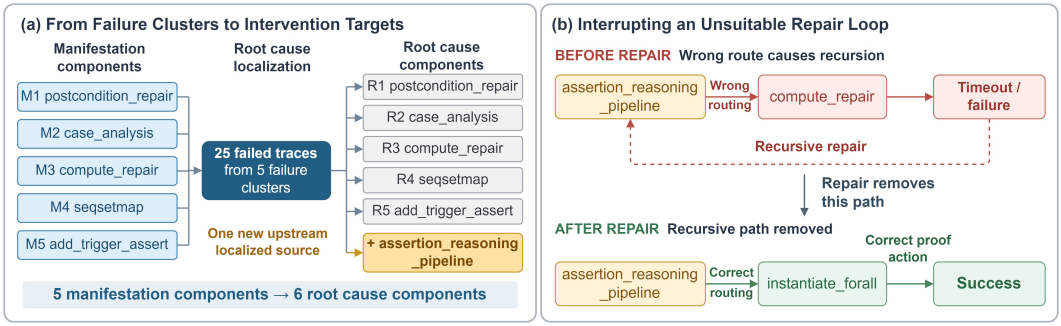

Figure 5: Examples of intervention targeting. (a) Among 25 selected trajectories, attribution redirects 12 to upstream targets; five manifestation components correspond to six intervention targets overall. (b) A repeated `compute_repair` path is traced to a decision in `assertion_reasoning_pipeline`, motivating a controller-level revision.

D Examples of Attribution and Localized Optimization

D.1 Tracing Symptoms to Intervention Targets

Figure 5 illustrates how diagnostic slicing and attribution can redirect optimization from a failure manifestation to an upstream component. The examples are drawn from VeruSAGE-Bench optimization and complement, rather than replace, the annotation-based attribution evaluation on Who&When.

From failure clusters to intervention targets. In a run using 200 complete failed trajectories, Phase 1 selects five failure clusters and five representative trajectories per cluster, yielding 25 trajectories. The failure manifestations in these selected traces involve five distinct components; this module count is a post-selection observation, not the unit used for sampling. Phase 3 assigns upstream targets to 12 of the 25 trajectories, and the aggregated intervention targets cover six components, including `assertion_reasoning_pipeline`. This illustrates the distinction between where a failure becomes observable and which component is selected for revision. The reassignment counts describe the method's diagnostic outputs and should not be interpreted as independently verified attribution accuracy.

Interrupting an unsuitable repair loop. In an IronKV example, repeated invocations of `compute_repair` end in a timeout, making the failure appear local to the executor. Phase 2 traces the relevant execution dependencies back to the routing decision, and Phase 3 attributes the unsuitable repair path to `assertion_reasoning_pipeline`. Revising the controller's routing policy avoids the repeated path and enables successful verification in this example. The intervention therefore changes the component governing the problematic decision rather than only the executor where the symptom appears.

D.2 Examples of Component-Specific Revisions

We summarize examples of instruction revisions synthesized for the `assertion_reasoning_pipeline` component. This controller can access repair history or explicitly supplied attempt records, so the history-dependent rules below are compatible with its available execution context. They would not be appropriate for a stateless executor without those inputs. The examples illustrate how Phase 4 consolidates attributed failures into reusable guidance rather than embedding individual failed instances as case-specific fixes.

Failure-aware stopping and action switching. Track attempts and their outcomes within the current proof context. If an action is repeatedly rejected after two or three attempts, particularly because of syntax errors or failure to address the target assertion, avoid repeating it under the same conditions. Select an alternative available action; if reasonable options are exhausted, report the

[4] `https://anonymous.4open.science/r/TracePrism-8CC3`

limitation and the need for human intervention rather than continuing ineffective retries. For example, repeated syntax failures from `INDUCTION` motivate switching to an applicable alternative such as `USELEMMA` or `CASE_ANALYSIS`. Reporting exhaustion is not itself a successful repair.

Action-selection boundaries and actionable guidance. Guidance should name concrete code anchors and proof obligations rather than provide generic advice. The synthesized examples specialize this requirement to individual actions:

- **`USELEMMA`:** Identify relevant helper lemmas or proof functions in the file and state what each establishes. If no relevant lemma is available, prefer a proof-construction action such as `CASE_ANALYSIS`, `INDUCTION`, or `COMPUTE`.
- **`CASE_ANALYSIS`:** Specify the variable or expression to split on and the obligation to establish in each branch.
- **`INDUCTION`:** State the base case, inductive hypothesis, and its connection to the current goal.
- **`EXTENSIONAL_EQUALITY`:** Identify the collection property requiring element-wise reasoning and the necessary domain or length conditions.
- **`SEQSETMAP`:** Name the two or three most relevant sequence, set, or map lemmas instead of invoking an unnecessarily broad lemma set.

Multi-round proof planning. Use visible results from previously accepted fixes to explain what has been established and how the next action advances the proof. For complex goals, specify a short sequence of intermediate obligations, such as establishing an auxiliary fact, proving the main implication, and connecting it to the target assertion. Action changes should follow this proof progression rather than alternate among unrelated strategies.