AutoPEER: ISSUE-GUIDED PROMPT EVOLUTION FOR LONG-HORIZON AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) increasingly power agents that solve complex tasks through multi-step tool interactions. Existing prompt optimization frameworks learn from execution feedback, but give limited attention to credit assignment in long-horizon tasks. Effective prompt evolution requires identifying which decisions in a trajectory should be improved and updating the prompt accordingly. We introduce AutoPEER, a framework that combines Issue-level credit assignment with progressive behavioral verification. AutoPEER locates decisions that need better guidance and records them as Issues to guide targeted prompt patches. Progressive verification tests whether patches change the intended actions, improve complete-task performance, and generalize across tasks after merging. A multidimensional reward rubric selects prompts based on task success and execution quality. Experiments on AppWorld show that AutoPEER improves both task completion and interaction efficiency across models, achieving the highest completion on each evaluated split. These improvements are observed with and without training reference solutions. Notably, prompts learned with one model improve another without further optimization. Additional evaluation on the telecom domain of τ^2-Bench demonstrates gains in a different tool environment. Ablations show that targeted repair and progressive verification help agents solve more tasks, while multidimensional rewards encourage more efficient execution.

1 INTRODUCTION

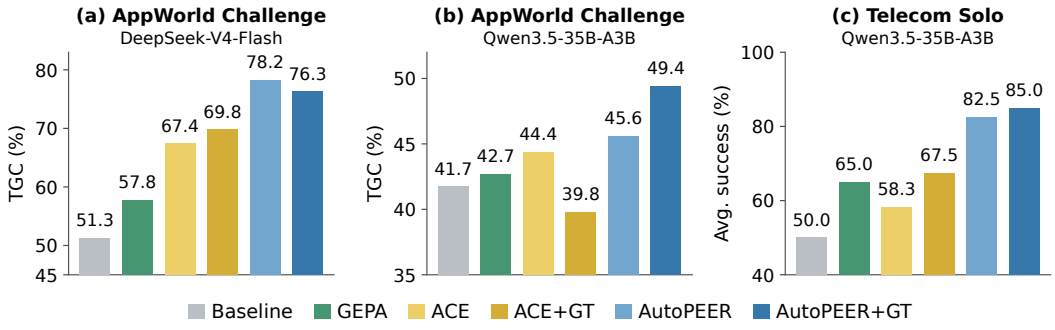


Figure 1: **AutoPEER improves completion across models and environments.** AppWorld Challenge TGC and mean Telecom Solo success over three evaluations. +GT indicates that ground-truth information is available to the optimizer during training.

Large language models (LLMs) increasingly perform long-horizon tasks through interactions with tools and stateful environments (Yao et al., 2023; Yang et al., 2024b; Trivedi et al., 2024). Prompt optimization improves these agents without updating model weights. Recent methods learn from language feedback to revise instructions (Pryzant et al., 2023; Yuksekgonul et al., 2025; Chen et al., 2026), evolve candidate prompts (Agrawal et al., 2026), or accumulate reusable strategies (Zhang et al., 2026). For long-horizon agents, the central question is how to turn a trajectory of decisions and observations into useful changes to the prompt.

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.

In long-horizon agents, prompts guide a sequence of interdependent decisions about planning, tool use, and error recovery. Earlier actions change the environment and thereby shape the agent's subsequent observations and decisions, with effects that can extend to the final task outcome. Improving the prompt therefore requires identifying which decisions most need repair and using trajectory evidence to guide targeted revisions. Credit assignment provides a useful starting point by connecting outcomes to earlier decisions (Arjona-Medina et al., 2019; Setlur et al., 2025). At the same time, revised prompts need to be tested through execution: the changes should improve task performance, and their benefits should extend beyond the few training tasks that motivated them. Diagnosis and verification thus address complementary parts of learning from long trajectories.

We introduce AutoPEER, a framework for issue-guided prompt evolution in long-horizon agents. Each Issue identifies a problem in the trajectory and its supporting evidence, guiding the Patch Generator to propose localized revisions that preserve unrelated instructions. These revisions undergo progressively broader tests. Fast Replay checks whether a patch changes the attributed decision by generating and executing actions from the same recorded history under the original and modified prompts. Full-Episode Validation then tests whether the change improves complete-task performance. Useful patches are merged and evaluated on recent tasks through Rolling, checking for interference and gains beyond the current batch. Throughout this process, a fixed multidimensional reward rubric guides selection by task success and execution quality, distinguishing equally successful prompts by efficiency.

We evaluate AutoPEER on AppWorld (Trivedi et al., 2024) with two models and on τ^2-Bench Telecom Solo (Barres et al., 2026). Our variants lead task and scenario completion on both AppWorld splits for both models, with gains under both feedback settings. Without ground-truth information (GT), DeepSeek Challenge completion rises from 51.32% to 78.18%, and Qwen Telecom Solo mean success rises from 50.0% to 82.5%. The learned prompts show how execution problems become reusable decision rules. For example, Figure 2 addresses redundant queries by directing the agent to reuse information already returned by tools and request details only when needed. By specifying both what to do and when, the revision turns a local observation into guidance for other tasks. Together, the completion, efficiency, and cross-model transfer results support this approach to learning prompts from agent experience.

Our contributions are threefold:

- **Issue-level credit assignment.** We identify problems in long execution trajectories, extract supporting evidence, and locate replay targets. Together, these provide a common basis for generating targeted textual gradients and testing their behavioral effects.
- **Three-stage textual-gradient verification.** The three stages assess a prompt revision's local effect, task-level direction, and generalization. They progressively test whether it changes the targeted behavior, improves complete-task performance, and generalizes across tasks after merging.
- **Prompt evolution for long-horizon agents.** We present AutoPEER, which combines Issue-level credit assignment with three-stage verification and uses multidimensional rewards to guide learning from multi-step tool interactions. Experiments on AppWorld and τ^2-Bench show improved task completion and interaction efficiency. Prompts learned with one agent model also improve another model's task performance.

2 Problem Setting

We study prompt optimization for agents that solve long-horizon tasks through tool interactions. At the start of an episode, the agent receives a task instruction x and an initial observation from the environment. At step t, it selects an action according to

$$a_t \sim \pi_\theta(\cdot \mid P, x, h_t),$$

where P is the agent prompt, θ denotes the model parameters, and h_t contains the observations and actions available so far. Executing an action returns feedback and may change the environment, shaping the agent's subsequent decisions. This interaction continues until the episode ends. The recorded actions, observations, and outcome form an execution trajectory.

Our optimization variable is P, which consists of system instructions and editable tool descriptions. The model parameters, tool interfaces, and execution framework remain fixed. The prompt stays

Baseline

Use detail/show APIs to disambiguate targets before acting, and for time-based tasks use `datetime.now()` or `apis.phone.get_current_date_and_time()` with full-day boundaries for terms like today or yesterday.

Issue

The system prompt does not instruct the agent to reuse nested data from list/detail API responses. When `show_album` and `show_playlist` already return full song objects with `release_date`, the agent redundantly calls `show_song` for each ID, wasting the API budget.

Replay target:	Before Turn 11: regenerate the action with the patched prompt.
Supporting evidence:	Turn 12 repeats the same `show_song` queries for the same song IDs.
Expected effect:	Reuse `release_date` from `show_album` or `show_playlist`; call `show_song` only when that field is missing.

AutoPEER's Optimized Prompt for AppWorld, DeepSeek-V4-Flash

Patch: *System prompt*

```
- Use detail/show APIs to disambiguate targets before acting, and for time-based tasks use datetime.now() or
- apis.phone.get_current_date_and_time() with full-day boundaries for terms like today or yesterday.

+ Use detail/show APIs to disambiguate targets before acting, and reuse nested objects and fields already returned by those
+ detail/show responses instead of calling per-item detail APIs for each ID. For example, if show_album or show_playlist
+ returns song objects with release_date, reuse that data rather than calling show_song for every song_id; only call a
+ per-item detail API for IDs whose needed fields are not already present in a prior response. For time-based tasks use
+ datetime.now() or apis.phone.get_current_date_and_time() with full-day boundaries for terms like today
+ or yesterday.
```

Figure 2: **An Issue links prompt repair to its replay target.** In the DeepSeek AutoPEER-GT run, redundant queries motivate a rule to reuse returned fields. Replay targets Turn 11; Turn 12 supplies supporting evidence. Red (−), green (+), and black mark removed, added, and unchanged text.

unchanged within an episode and is revised across optimization rounds. Updating P therefore changes how the agent approaches a task, including its tool choices, use of returned information, and responses to execution errors.

We formulate prompt learning as maximizing the expected reward of complete episodes:

$$P^\star = \arg\max_P \; \mathbb{E}_{x\sim\mathcal{T}} \left[\mathbb{E}[r(e) \mid x, P]\right], \tag{1}$$

where $\mathcal{T}$ is the task distribution, e is an episode generated by the agent interacting with the environment, and $r(e)$ is assigned by a fixed multidimensional reward rubric. The rubric combines task completion with desired execution behavior, setting the priorities for prompt evolution. Training trajectories provide evidence for prompt revision; reference solutions provide additional guidance when available. Training and evaluation tasks share the same tool interfaces but differ in task instances. The goal is to use training experience to learn a prompt that improves complete-task performance on new task instances.

3 AutoPEER

We introduce AutoPEER to pursue this objective through issue-guided prompt evolution and progressive behavioral verification (Figure 3). Each generation starts from the retained *Champion*, which serves as the *Parent* for candidate comparison. The *Worker* executes tasks to collect trajectories, and the optimizer uses two role prompts. The *Issue Generator* examines complete trajectories to identify problems, supporting evidence, and replay targets. The *Patch Generator* uses these Issues, the current prompt, and the trajectories to propose localized edits. Candidates then undergo progressively broader tests: local action changes, complete-task gains, and improvements on other tasks after merging. A multidimensional reward rubric guides comparison and promotion, rewarding both task completion and desired execution behavior. The retained prompt guides the next generation. Appendix A gives the complete loop.

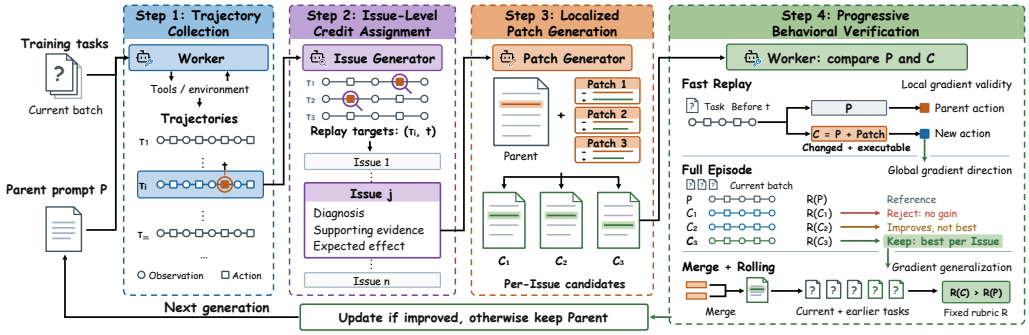


Figure 3: **AutoPEER links prompt repairs to behavioral tests.** Issues guide patches and replay targets. The three checks examine local action changes, complete-task improvement, and reuse on recent tasks after merging. The Worker executes these tests; a Rolling reward gain determines promotion.

3.1 Issue-level credit assignment and prompt patches

Temporal credit assignment traces task outcomes back to the earlier decisions that shaped them (Arjona-Medina et al., 2019; Setlur et al., 2025). AutoPEER applies this idea through an Issue Generator that analyzes complete task trajectories to identify problems that prompt revisions could address. This *Issue-level credit assignment* records the observed problem, supporting trajectory events, and the expected behavioral change in an Issue. The Issue also names one or more replay targets: the trajectories and decisions where the patch will first be tested. The generator selects the earliest decisions supported by the evidence and records observations that would falsify its diagnosis. Related failures can be consolidated into one repair target, while successful trajectories can reveal redundant work. An Issue therefore guides both patch generation and local verification by linking the diagnosed problem to its supporting evidence and replay targets.

The Patch Generator reads the Issue, the current prompt, and the complete task trajectories to decide where and how to revise the instructions. It proposes alternative localized patches that preserve unrelated instructions. Anchored additions, replacements, or deletions specify concrete directions for improving agent behavior. Each patch is linked to the diagnosed problem and its trajectory evidence, making clear what changes and why (Figure 2). Behavioral verification then tests whether the proposed revision works.

3.2 Progressive behavioral verification

Textual-gradient methods use natural-language feedback to guide prompt updates (Pryzant et al., 2023; Yuksekgonul et al., 2025). AutoPEER uses trajectory Issues to guide these updates and progressively verifies whether they change the targeted behavior, improve complete-task performance, and generalize across tasks after merging (Figure 3).

Fast Replay: testing local behavior. A targeted prompt revision should produce an observable change at the decision identified by its Issue. Fast Replay tests this requirement by revisiting the decision under the same task and interaction history. It asks whether the revised prompt changes the agent's next action and whether that action executes successfully without detected invalid calls. Passing this check provides local evidence that the textual gradient has taken effect where the problem was observed.

Full-Episode Validation: testing task-level improvement. Changing one action can affect later decisions and the final outcome. We therefore evaluate the revised prompt on complete tasks to determine whether the change improves performance. Full-Episode Validation compares complete Candidate and Parent episodes on the same batch of m training tasks. Their reward difference tests the global direction of the update. For each Issue, the highest-reward eligible candidate that improves on the Parent proceeds to merging.

Merge and Rolling: testing composition and generalization. Individually useful patches can conflict after merging, so their combined effect needs a new test. Earlier revisions and checks focus on the current training batch. Rolling extends evaluation to earlier training tasks to test whether the improvements carry over to other problems. AutoPEER applies compatible patches to the Parent and evaluates the merged prompt on up to the latest K training batches. Promotion requires higher mean reward than the Parent over this window.

3.3 Rubric-guided prompt evolution

Issues identify problems in execution trajectories, while the reward rubric provides a common standard for evaluating revisions and selecting updates. AutoPEER scores complete episodes based on task completion and execution behavior, using these scores to compare candidates and guide promotion in Full-Episode Validation and Rolling. The scoring criteria specify desired outcomes and behaviors, while their weights express the relative priorities (Gunjal et al., 2026; Qian et al., 2025), together guiding prompt evolution. For a complete episode e, the rubric combines these criteria into a single reward:

$$r(e) = \sum_{j=1}^{d} w_j \, s_j(e). \tag{2}$$

Here, $s_j(e)$ measures criterion j, w_j is its weight, and d is the number of criteria. Positive weights reward desired outcomes; negative weights penalize undesirable behavior. We fix the criteria and weights within each optimization run so candidates are compared against the same objective. Our experiments use this design to encourage efficient task completion, favoring prompts that solve tasks with fewer unnecessary interactions.

The same reward rubric guides candidate selection and prompt promotion. Full-Episode Validation uses reward to compare alternative patches on the current training batch, while Rolling uses it to evaluate their combined effect on recent tasks. Both compare candidates with the Parent on the same tasks. A higher-reward prompt replaces the Parent after Rolling; ties and losses retain the Parent. New trajectory evidence can then guide further revisions in the next generation.

4 Experimental Setup

Benchmarks and metrics. We use two multi-step tool environments with executable evaluation. Training and test tasks share tool interfaces but differ in task instances, testing whether learned instructions transfer to new tasks.

AppWorld agents execute code across applications (Trivedi et al., 2024). We optimize on 90 tasks from 30 scenarios with three variants each, and evaluate on Normal (168 tasks, 56 scenarios) and Challenge (417 tasks, 139 scenarios). Task-goal completion (TGC) is the percentage of tasks solved. Scenario-goal completion (SGC), specific to AppWorld, requires all three task variants in a scenario to succeed, measuring consistency across related tasks.

τ^2-Bench Telecom Solo (Barres et al., 2026) evaluates troubleshooting with customer-service and device-side tools. The agent uses both tool sets from a task ticket, without a simulated user. We use the Verified implementation (74 training tasks, 40 held-out test tasks) and average evaluator success rates over three evaluations of the same test set.

Baselines and feedback. Both benchmarks include the unoptimized Baseline, GEPA's reflective prompt search (Agrawal et al., 2026), and ACE's evolving playbook of reusable strategies (Zhang et al., 2026). With GT, the Issue Generator and Patch Generator can access reference execution information for the current task when diagnosing issues and generating patches. Both feedback settings use execution trajectories and feedback. ACE runs with and without GT. AppWorld also includes Reflection-GT, which rewrites or retains the prompt after each training batch. Final prompts remain fixed during testing, when Workers receive only the task, learned prompt, and allowed environment observations.

Models and optimization. AppWorld Workers are DeepSeek-V4-Flash and Qwen3.5-35B-A3B; Telecom Solo uses Qwen3.5-35B-A3B. GLM-5.2 serves as both generators. Methods share starting agent prompts, runtime, and tool interfaces within each setting. Each generation uses up to three tasks

($m = 3$), extracts at most three Issues, and requests three patches per Issue. Rolling uses the latest three training batches ($K = 3$, up to nine tasks). AutoPEER runs for 30 generations on AppWorld and 25 on Telecom.

For AppWorld, $r(e) = 10S - 0.02T - 0.01A - 0.10I - 5V$, where S indicates success and T, A, I, and V count steps, API calls, invalid actions, and safety violations in episode e. Appendix C provides baseline details and benchmark-specific reward settings.

5 Results and Analysis

We evaluate completion across tasks and Workers, test cross-Worker prompt transfer, and examine how diagnosis, verification, and reward design contribute to the gains.

5.1 Results on AppWorld

Table 1: **AppWorld results with two Workers.** Scores are percentages; superscripts show percentage-point changes from each Worker's Baseline.

Method	GT	Normal		Challenge	
		TGC	SGC	TGC	SGC
DeepSeek-V4-Flash					
Baseline	–	77.98	58.93	51.32	37.41
GEPA	–	79.17 +1.2	64.29 +5.4	57.79 +6.5	41.73 +4.3
ACE	–	80.95 +3.0	75.00 +16.1	67.39 +16.1	48.92 +11.5
ACE	✓	83.93 +6.0	75.00 +16.1	69.78 +18.5	53.24 +15.8
Reflection	✓	85.12 +7.1	75.00 +16.1	70.50 +19.2	53.24 +15.8
AutoPEER (ours)	–	84.52 +6.5	67.86 +8.9	**78.18** +26.9	59.71 +22.3
AutoPEER (ours)	✓	**86.31** +8.3	**76.79** +17.9	76.26 +24.9	**61.87** +24.5
Qwen3.5-35B-A3B					
Baseline	–	52.98	30.36	41.73	22.30
GEPA	–	48.81 -4.2	30.36 +0.0	42.69 +1.0	23.02 +0.7
ACE	–	56.55 +3.6	37.50 +7.1	44.36 +2.6	24.46 +2.2
ACE	✓	58.33 +5.4	39.29 +8.9	39.81 -1.9	18.71 -3.6
Reflection	✓	58.93 +6.0	42.86 +12.5	46.52 +4.8	26.62 +4.3
AutoPEER (ours)	–	**64.29** +11.3	**44.64** +14.3	45.56 +3.8	**30.22** +7.9
AutoPEER (ours)	✓	63.69 +10.7	39.29 +8.9	**49.40** +7.7	**30.22** +7.9

AutoPEER improves task and scenario completion across both Workers and both test splits, achieving the highest task completion rates among the compared methods in both GT and no-GT settings (Table 1). Without GT, AutoPEER still outperforms GEPA and ACE, showing that it can improve prompts through the agent's own execution experience without additional reference execution information. The benefit of GT varies across Workers and splits: it improves Qwen's performance on Challenge, whereas DeepSeek achieves higher Challenge completion without GT. Reference execution information therefore provides additional guidance for optimization, but does not consistently lead to higher completion rates.

AutoPEER makes more effective use of long-horizon evaluation in the DeepSeek comparison. Without GT, it reaches 78.18% Challenge TGC versus GEPA's 57.79% (Table 1), with fewer full-task evaluations (Appendix D.3). GEPA's later checkpoints improve little despite additional budget (Figure 5). Its reflective search repeatedly validates candidates (Agrawal et al., 2026); in long-horizon tasks, each evaluation requires a complete sequence of interactions. AutoPEER first links patches to trajectory Issues and checks their local effects before broader testing. This provides behavioral evidence before committing to full-task comparisons. Section 5.4 examines the resulting validation workload and how later checks distinguish local changes from useful improvements.

AutoPEER also achieves higher completion with less prompt growth than ACE. On Challenge with GT, it reaches 76.26% TGC versus ACE's 69.78% on DeepSeek and 49.40% versus 39.81% on Qwen (Table 1), while adding substantially less text (Table 7). ACE preserves knowledge through incremental playbook updates, avoiding information loss and performance drops from full-context rewriting (Zhang et al., 2026), but experience continues to accumulate even with refinement and deduplication. AutoPEER instead incorporates experience through localized patches that can refine existing rules while preserving unrelated guidance. Its verification stages determine which revisions are retained, testing local effects, complete-task gains, and reuse after merging. This differs from directly integrating Curator updates: preservation is coupled with execution-based selection. Together, the completion and prompt-growth results show that targeted revisions can improve performance without a large expansion of instructions.

5.2 Results on Telecom Solo

To evaluate AutoPEER beyond AppWorld's cross-application coding, we optimize prompts for service troubleshooting in τ^2-Bench Telecom Solo. Here, the agent diagnoses service problems using customer-service and device-side tools. The tools and task goals change, but both settings require decisions based on feedback from earlier actions. AutoPEER raises Qwen's average success from 50.0% to 82.5% without GT and 85.0% with GT (Table 2), outperforming GEPA and both ACE variants. Most of the gain is achieved without reference solutions, suggesting that execution evidence remains useful for prompt revision in this different task setting. GT provides a further, smaller improvement. Together with the AppWorld results, these findings show that AutoPEER provides more effective prompt optimization across different task settings, achieving higher task completion rates than GEPA and ACE.

Table 2: **Telecom Solo results.** AutoPEER leads with and without GT. Scores average three evaluations of the same 40 tasks. Scores and formatting follow Table 1.

Method	GT	Success (%)
Qwen3.5-35B-A3B		
Baseline	–	50.00
GEPA	–	65.00 $^{+15.0}$
ACE	–	58.33 $^{+8.3}$
ACE	✓	67.50 $^{+17.5}$
AutoPEER (ours)	–	82.50 $^{+32.5}$
AutoPEER (ours)	✓	**85.00** $^{+35.0}$

5.3 Do learned prompts transfer across Workers?

AutoPEER's learned prompts improve both Workers over Baseline when transferred without further optimization (Table 3). With Qwen as the evaluation Worker, the prompt evolved with DeepSeek outperforms the one evolved with Qwen. With DeepSeek as the evaluation Worker, the prompt evolved with DeepSeek performs better. We find that the two Workers expose different problems during training, leading the optimizer to focus on different improvements. Qwen encounters more failed training trajectories, and its revisions frequently address incorrect API calls, response fields, and interface distinctions. DeepSeek already completes most training tasks, allowing optimization to focus more on redundant queries and unnecessary exploration. Its retained rules include batching documentation queries, limiting query scope, and reusing returned data. These updates also differ in scope: repairs tied to particular interfaces help when those conditions recur, whereas query organization and data reuse apply across many tasks and can improve several decisions within an episode. Qwen also proposes efficiency improvements, but some do not take effect or survive validation. Transferring the prompt evolved with DeepSeek therefore supplies execution strategies that complement Qwen's own repairs. Their broader applicability helps explain why Qwen benefits from a prompt evolved with another Worker.

Table 3: **Cross-Worker transfer of AutoPEER-GT prompts on Challenge.** Scores and formatting follow Table 1.

Prompt	TGC	SGC
Qwen Worker		
Baseline	41.73	22.30
Same Worker	49.40 $^{+7.7}$	30.22 $^{+7.9}$
From DeepSeek	**55.40** $^{+13.7}$	**35.25** $^{+12.9}$
DeepSeek Worker		
Baseline	51.32	37.41
Same Worker	**76.26** $^{+24.9}$	**61.87** $^{+24.5}$
From Qwen	64.51 $^{+13.2}$	48.20 $^{+10.8}$

5.4 How diagnosis and verification shape prompt updates

We compare four variants with DeepSeek-V4-Flash (Table 4). *w/o Issue* generates patches directly from trajectories; *w/o Fast Replay* retains Issue-guided proposals and sends valid patches directly to full-episode validation. *w/o Merge & Rolling* adopts a single patch after full-episode validation; *w/o Multidimensional Reward* uses success alone for selection and optimizer guidance. All share the agent and benchmark interface and improve on Baseline.

Table 4: **Component ablations with DeepSeek-V4-Flash.** Scores are percentages; superscripts show percentage-point changes from Baseline. Bold marks the best score per metric, including ties.

Method	Normal		Challenge	
	TGC	SGC	TGC	SGC
Baseline	77.98	58.93	51.32	37.41
w/o Issue	85.12 $^{+7.1}$	75.00 $^{+16.1}$	70.98 $^{+19.7}$	54.68 $^{+17.3}$
w/o Fast Replay	**86.31** $^{+8.3}$	**76.79** $^{+17.9}$	**76.26** $^{+24.9}$	56.83 $^{+19.4}$
w/o Merge & Rolling	78.57 $^{+0.6}$	67.86 $^{+8.9}$	69.54 $^{+18.2}$	49.64 $^{+12.2}$
w/o Multidimensional Reward	85.12 $^{+7.1}$	75.00 $^{+16.1}$	66.91 $^{+15.6}$	51.08 $^{+13.7}$
AutoPEER-GT	**86.31** $^{+8.3}$	**76.79** $^{+17.9}$	**76.26** $^{+24.9}$	**61.87** $^{+24.5}$

Removing Issue-level attribution leaves the Patch Generator to propose revisions directly from trajectories. This also removes Fast Replay, which depends on the decision targets identified by Issues. Compared with the full method, this variant lowers AppWorld task completion by 5.28 percentage points on Challenge and 1.19 on Normal (Table 4). These results support the role of Issue-guided repair in the optimization loop: the diagnosis identifies decisions to improve and provides supporting evidence, giving patch generation a clearer direction and local verification a concrete target.

Removing Fast Replay leaves task completion unchanged on both AppWorld splits (Table 4). Retaining it, however, reduces the number of candidates entering full-episode validation from 127 to 112. These results support its intended role: screening patches before costly full-task evaluation reduces validation workload while preserving the final task completion rate.

Full-Episode Validation tests whether local behavioral changes improve complete-task performance. Across the DeepSeek runs, 39 of 93 replay-passing no-GT candidates (41.9%) and 43 of 103 GT candidates (41.7%) fail to improve full-task reward. They change the cited action and pass local execution checks without improving complete episodes. Fast Replay and Full-Episode Validation therefore answer different questions: whether a patch affects the intended decision, and whether that change helps the task.

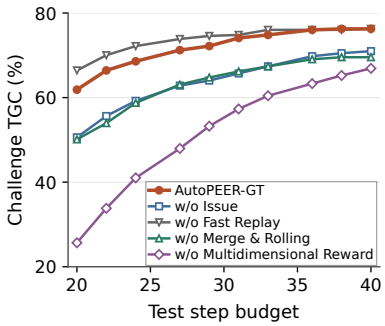


Figure 4: **Completion by test budget on Challenge.** AutoPEER-GT and four ablations with DeepSeek-V4-Flash.

Rolling tests whether the gains from merged patches extend beyond the current training batch. Removing Merge and Rolling lowers TGC by 6.71 points on Challenge and 7.74 on Normal; fully solved Challenge scenarios fall from 86 to 69, reducing SGC from 61.87% to 49.64%. The execution records illustrate the problem these checks address. Login and time repairs each increase the number of completed current-batch tasks from one to three. After merging, however, the number of completed Rolling tasks falls from seven with the Parent to four. The number of completed earlier tasks drops from six to three, so the Parent is retained. Conversely, combining field reuse and cross-source deduplication increases the number of completed Rolling tasks from seven to nine. Of seven GT multi-patch combinations, three are adopted and four rejected. Rolling thus accepts useful combinations while blocking observed regressions beyond the current batch.

5.5 THE REWARD RUBRIC GUIDES EVOLUTION TOWARD EARLIER COMPLETION

The multidimensional reward rubric shapes both task completion and execution efficiency. Figure 4 shows a distinct pattern across the ablations. Removing Issue or Merge and Rolling mainly lowers completion rates. In contrast, the success-only variant falls well behind at smaller budgets and continues to catch up as the budget increases. Its gains occur later, indicating that it requires more interactions to complete tasks, whereas the full method achieves strong performance earlier. An adopted patch illustrates how the rubric encourages this behavior: it combines separate documentation queries into one action. Both Parent and Candidate complete all tasks in the current batch and the Rolling window, but the rubric promotes the more efficient Candidate; success-only scoring would leave them tied. The additional reward dimensions thus distinguish between equally successful candidates and guide prompt evolution toward more efficient execution.

6 RELATED WORK

Automatic prompt optimization. Prompt optimizers generate instructions (Zhou et al., 2023; Yang et al., 2024a; Wang et al., 2024b), evolve prompt populations (Guo et al., 2024; Fernando et al., 2024), and jointly optimize instructions and demonstrations (Khattab et al., 2024; Opsahl-Ong et al., 2024). GEPA combines reflection with Pareto-based selection, preserving candidates with complementary strengths across tasks (Agrawal et al., 2026). AutoPEER organizes candidate revisions by trajectory Issue, focusing the search on alternative repairs to specific problems supported by execution evidence.

Agent experience and context adaptation. Reflection turns feedback into improved outputs and reusable experience (Madaan et al., 2023; Shinn et al., 2023; Zhang et al., 2024b; Zhao et al., 2024). Experience can take the form of workflows, cached plans, manuals, or adaptive memories (Wang et al., 2025b; Zhang et al., 2025b; Chen et al., 2024; Xu et al., 2025; Suzgun et al., 2026). ACE organizes experience into a structured playbook through incremental updates and refinement (Zhang et al., 2026). AutoPEER revises specific instructions and refines their conditions of use. Execution tests select the changes to retain, supporting targeted improvement with limited prompt growth.

Language feedback and credit assignment. ProTeGi uses error feedback to suggest prompt revisions (Pryzant et al., 2023); TextGrad, Trace, and TEP use language feedback to optimize computational components and propagate feedback through compound systems (Yuksekgonul et al., 2025; Cheng et al., 2024; Chen et al., 2026). Credit assignment connects outcomes to intermediate decisions: RUDDER redistributes delayed returns, while Rewarding Progress evaluates steps by their contribution to future success (Arjona-Medina et al., 2019; Setlur et al., 2025). AutoPEER uses attribution to identify decisions that prompt revision could improve. The diagnosis links the evidence motivating a repair to a specific decision where its behavioral effect can be tested.

Reward-guided optimization. Eureka and Text2Reward use LLMs to construct reward functions (Ma et al., 2024b; Xie et al., 2024b), ToolRL studies fine-grained tool-use rewards (Qian et al., 2025), and Optimas aligns local rewards with system objectives (Wu et al., 2026). AutoPEER uses a fixed multidimensional reward rubric to select prompt revisions by task completion and execution behavior.

7 CONCLUSION

We presented AutoPEER, an issue-guided framework for prompt evolution in long-horizon agents, combining credit assignment with progressive behavioral verification. A multidimensional reward rubric guides the evolution toward task success and efficient execution. Experiments on AppWorld and τ^2-Bench Telecom show higher task completion than the compared methods, with or without reference execution trajectories during optimization. Learned prompts also improve another agent model's performance, demonstrating the transferability of the resulting instructions. Together, these findings show that diagnosing trajectory problems and validating targeted revisions provide an effective approach to learning reusable instructions from agent experience while keeping the model weights and tools fixed.

AI USE STATEMENT

In preparing this work, we used generative AI tools to assist with revising and polishing the author-written paper, and identifying and reviewing related literature. The authors reviewed all AI-assisted content and take full responsibility for the content of this paper.

REFERENCES

Eshaan Agarwal, Raghav Magazine, Joykirat Singh, Vivek Dani, Tanuja Ganu, and Akshay Nambi. PromptWizard: Optimizing Prompts via Task-Aware, Feedback-Driven Self-Evolution. In *Findings of ACL*, 2025.

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. In *ICLR*, 2026.

José A. Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. RUDDER: Return decomposition for delayed rewards. In *NeurIPS*, 2019.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment. In *ICML*, 2026. URL `https://icml.cc/virtual/2026/poster/64377`.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. AutoManual: Constructing instruction manuals by LLM agents via interactive environmental learning. In *NeurIPS*, 2024.

Minghui Chen, Wenlong Deng, James Zou, Han Yu, and Xiaoxiao Li. Textual equilibrium propagation for deep compound AI systems. In *ICLR*, 2026.

Ching-An Cheng, Allen Nie, and Adith Swaminathan. Trace is the next AutoDiff: Generative optimization with rich feedback, execution traces, and LLMs. In *NeurIPS*, 2024.

Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. RLPrompt: Optimizing Discrete Text Prompts with Reinforcement Learning. In *EMNLP*, 2022.

Yihong Dong, Kangcheng Luo, Xue Jiang, Zhi Jin, and Ge Li. PACE: Improving Prompt with Actor-Critic Editing for Large Language Model. In *Findings of ACL*, 2024.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *ICML*, 2024.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing. In *ICLR*, 2024.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as Rewards: Reinforcement Learning Beyond Verifiable Domains. In *ICLR*, 2026.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In *ICLR*, 2024.

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 State-of-the-Art Pipelines. In *ICLR*, 2024.

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. *Transactions of the Association for Computational Linguistics*, 2024a.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *ICLR*, 2024b.

Xiaoqian Liu, Ke Wang, Yuchuan Wu, Fei Huang, Yongbin Li, Jianbin Jiao, and Junge Zhang. Agentic reinforcement learning with implicit step rewards. In *ICLR*, 2026.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In *NeurIPS*, 2024a.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, and Anima Anandkumar. Eureka: Human-Level Reward Design via Coding Large Language Models. In *ICLR*, 2024b.

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. In *NeurIPS*, 2023.

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. In *EMNLP*, 2024.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *EMNLP*, 2023.

Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tür, Gokhan Tur, and Heng Ji. ToolRL: Reward is All Tool Learning Needs. In *NeurIPS*, 2025.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. In *ICLR*, 2024.

Amrith Setlur, Chirag Nagpal, Adam Fisch, Xinyang Geng, Jacob Eisenstein, Rishabh Agarwal, Alekh Agarwal, Jonathan Berant, and Aviral Kumar. Rewarding progress: Scaling automated process verifiers for LLM reasoning. In *ICLR*, 2025.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. Dynamic Cheatsheet: Test-Time Learning with Adaptive Memory. In *EACL*, 2026.

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

Xingchen Wan, Ruoxi Sun, Hootan Nakhost, and Sercan Ö. Arık. Teach Better or Show Smarter? On Instructions and Exemplars in Automatic Prompt Optimization. In *NeurIPS*, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024a.

Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric Xing, and Zhiting Hu. PromptAgent: Strategic planning with language models enables expert-level prompt optimization. In *ICLR*, 2024b.

Zora Zhiruo Wang, Apurva Gandhi, Graham Neubig, and Daniel Fried. Inducing Programmatic Skills for Agentic Tasks. In *COLM*, 2025a.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent Workflow Memory. In *ICML*, 2025b.

Cheng-Kuang Wu, Zhi Rui Tam, Chieh-Yen Lin, Yun-Nung Chen, and Hung yi Lee. StreamBench: Towards Benchmarking Continuous Improvement of Language Agents. In *NeurIPS*, 2024.

Shirley Wu, Parth Sarthi, Shiyu Zhao, Aaron Lee, Herumb Shandilya, Adrian Mladenic Grobelnik, Nurendra Choudhary, Eddie Huang, Karthik Subbian, Linjun Zhang, Diyi Yang, James Zou, and Jure Leskovec. Optimas: Optimizing Compound AI Systems with Globally Aligned Local Rewards. In *ICLR*, 2026.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *NeurIPS*, 2024a.

Tianbao Xie, Siheng Zhao, Chen Henry Wu, Yitao Liu, Qian Luo, Victor Zhong, Yanchao Yang, and Tao Yu. Text2Reward: Reward Shaping with Language Models for Reinforcement Learning. In *ICLR*, 2024b.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic Memory for LLM Agents. In *NeurIPS*, 2025.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *ICLR*, 2024a.

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. In *NeurIPS*, 2024b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. In *ICLR*, 2025.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin, and James Zou. Optimizing generative AI by backpropagating language model feedback. *Nature*, 2025.

Peiyan Zhang, Haibo Jin, Leyang Hu, Xinnuo Li, Liying Kang, Man Luo, Yangqiu Song, and Haohan Wang. Revolve: Optimizing AI Systems by Tracking Response Evolution in Textual Optimization. In *ICML*, 2025a.

Qizheng Zhang, Michael Wornow, and Kunle Olukotun. Agentic Plan Caching: Test-Time Memory for Fast and Cost-Efficient LLM Agents. In *NeurIPS*, 2025b.

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. In *ICLR*, 2026.

Tianjun Zhang, Aman Madaan, Luyu Gao, Steven Zheng, Swaroop Mishra, Yiming Yang, Niket Tandon, and Uri Alon. In-context principle learning from mistakes. In *ICML*, 2024a.

Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. Agent-Pro: Learning to Evolve via Policy-Level Reflection and Optimization. In *ACL*, 2024b.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *AAAI*, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *ICML*, 2024a.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *ICLR*, 2024b.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large Language Models Are Human-Level Prompt Engineers. In *ICLR*, 2023.

A Complete Optimization Algorithm

Algorithm 1 gives the complete loop. Each batch contains m training tasks for finding Issues and comparing patches. Rolling includes the current batch and up to $K-1$ preceding batches. We write $R(P; W)$ for the mean rubric reward of episodes produced by prompt P on task window W. All candidates in a generation are compared with the same Parent. The retained Champion generates the next generation's trajectories.

Algorithm 1 AutoPEER: behaviorally verified prompt evolution

Require: Baseline prompt P_0, size-m batches $B_1, \ldots, B_G$, history length K, fixed rubric R

$P \leftarrow P_0$; $\mathcal{H} \leftarrow [\,]$
for $g = 1, \ldots, G$ **do**
  $\tau \leftarrow \mathrm{CollectTrajectories}(P, B_g)$
  $\mathcal{I} \leftarrow \mathrm{IssueGenerator}(P, \tau)$
  $\mathcal{S} \leftarrow \varnothing$
  for each Issue $I \in \mathcal{I}$ **do**
    $\mathcal{C} \leftarrow \mathrm{PatchGenerator}(P, I, \tau)$
    Discard patches outside the editable scope or without applicable edits
    Form $\mathcal{V}$ from Fast Replay passes and candidates requiring full-episode fallback
    Evaluate each $C \in \mathcal{V}$ and P on the same tasks in B_g
    Select the highest-reward eligible C if $R(C; B_g) > R(P; B_g)$
    Add its patch to $\mathcal{S}$, if such a candidate exists
  end for
  Append B_g to $\mathcal{H}$; form W_g from up to the latest K batches
  if $\mathcal{S} \neq \varnothing$ **then**
    $M \leftarrow \mathrm{Merge}(P, \mathcal{S})$
    $P \leftarrow \mathrm{RollingSelect}(P, M, \mathcal{S}, W_g)$ ▷ retain only a verified gain
  end if
  Retain Issue, patch, behavior-test, episode, and selection records
end for
return P

Fast Replay. Fast Replay has two checks: whether the patch changes the model's decision given the same task instruction and interaction history, and whether the resulting action executes successfully. It fixes both before the cited decision, substitutes the Parent or Candidate prompt, and regenerates only the next action. Environment checks replay the recorded prefix in a fresh task environment and execute the generated action. AppWorld re-executes the preceding code blocks; Telecom re-executes tool calls and checks their responses against the recorded history. Earlier decisions are not regenerated. A pair passes if the action changes and executes successfully without detected invalid calls. A patch passes when a majority of repeated pairs pass at at least one cited decision. The Issue's expected effect guides patch generation and remains available for inspection. These environment checks supply external feedback, a source also used for tool-interactive correction (Gou et al., 2024) and language-agent tree search (Zhou et al., 2024a).

Full episodes and Rolling. For each Issue, full-episode comparison selects the highest-reward eligible candidate that improves on the Parent. Merge applies anchored text operations in candidate-ranking order, skipping a patch if its operations cannot be applied to the accumulating prompt. No additional LLM rewrites the merged text. RollingSelect evaluates the merged prompt against the Parent on the same recent-task window and retains it only for a valid reward gain. Where single-patch fallback is enabled, a winner is chosen from current-batch results before Rolling; if the merge fails, this candidate is checked on the same window, unless it is identical to the merge. If neither passes, the Parent remains. Matching Parent results can be reused without another episode. Appendix C lists the run configurations.

B Optimizer Prompts and Repair Representation

The Issue Generator and Patch Generator use distinct role prompts. An Issue diagnoses problems in the trajectory and records supporting evidence, the expected behavioral change, and decisions for replay. The Patch Generator uses this diagnosis and the current prompt to choose where and how to add, replace, or delete text. Execution tests determine whether to retain the resulting agent prompt. Each stage has a separate record.

The Issue Generator receives the current agent prompt and training trajectories; the Patch Generator also receives the Issue and its permitted editing scope: system instructions or named tool descriptions. This scope identifies editable components, not specific passages or text anchors. The templates below condense the task-independent instructions for these roles. Task content, trajectory events, current agent prompt text, and optional training references are inserted with each request. In our setting, a training batch contains three tasks and each Issue receives three alternative patches. Table 5 lists the optimization parameters; Section 4 and Appendix C give the benchmark-specific reward settings.

B.1 Issue Generator

Agentic reinforcement learning can use step-level rewards to assign credit within trajectories (Liu et al., 2026). Our diagnostic output identifies decisions for prompt repair and replay.

Issue Generator — Condensed optimizer template

Analyze the current training trajectories. Identify up to three distinct, evidence–grounded Issues; do not invent Issues to fill the quota.

Diagnose only: do not write patch text or choose edit operations. Identify the earliest supported decision that revising the agent prompt could improve. Explain the observed behavior, its causal mechanism, the relevant instruction component, the expected behavior change, and evidence that could falsify the diagnosis.

A failed task, low reward, or API exception does not by itself establish a prompt defect. Distinguish problems in the instructions from model execution errors, service failures, and environment behavior. Identify independently repairable causes rather than downstream symptoms. Consolidate observations that share one cause and editable target.

For each Issue, identify one to three earliest critical decisions that can be tested by regenerating the action while holding the preceding interaction fixed.

Return structured JSON with the Issue, supporting trajectory events, causal explanation, attribution confidence, editable targets, expected effect, and replay decisions. Do not propose a repair at this stage.

B.2 Patch Generator

Patch Generator — Condensed optimizer template

Propose three materially different patches for the diagnosed Issue. Distinguish candidates by repair mechanism, not by patch size.

Check the diagnosis against its cited events, the current agent prompt, and supplied tool documentation. Address only this Issue and its permitted targets; do not introduce another Issue or broaden the editing scope.

Inspect the current system instructions and tool descriptions for overlap and contradictions. Use anchored additions, replacements, or deletions. Copy anchors and replaced text exactly from the current component. Do not change tool names, schemas, implementations, or the fixed agent protocol.

Revise an existing rule when an addition would duplicate or conflict with it. Add a rule only when it governs independent behavior. Explain how the patch addresses the Issue and which existing behavior it preserves.

Preserve valid alternative strategies. Do not force a reference solution's exact route, omit required verification, impose arbitrary search limits, or declare completion early to reduce cost. Check the complete edited passage for clarity, missing conditions, duplication, and contradictions.

Patch Generator — Condensed optimizer template (continued)

Return JSON containing the edit operations, supporting events, repair mechanism, rationale, expected effect, and regression risks. The rationale proposes an improvement; execution tests determine whether it works.

The Patch Generator selects the exact passages and edit operations within the permitted components. Its candidates address the same diagnosed problem, allowing us to compare different ways of changing the behavior.

B.3 Using training references

Learning reusable guidance from examples has been studied through mistake-derived principles (Zhang et al., 2024a) and instruction–exemplar optimization (Wan et al., 2024; Agarwal et al., 2025). Here, references support diagnosis and revision, while the learned rules must remain usable without the reference.

Training references — Condensed optimizer template

Use the reference solutions for the current training tasks to derive reusable behavior. Do not copy task–specific identifiers, values, or answers into the prompt. Do not use development or test data.

With no-GT, the optimizer uses trajectory evidence and the current prompt. With GT, it also receives the ground-truth information described in Appendix C. Issue attribution identifies which decision to repair; the reward rubric compares candidates using their execution results. Appendix E shows an example of an instruction replacement.

B.4 Reward guidance

The executable rubric scores completed episodes and selects candidates. Its definitions and weights also guide the optimizer's search for useful changes. Trajectory evidence supports the Issue's diagnosis, while execution checks establish the candidate's behavioral effects. The following template summarizes the reward guidance.

Reward guidance — Condensed optimizer template

Prioritize evidence–grounded opportunities to complete tasks correctly. When no clear correctness defect is supported, consider avoidable redundant work or invalid calls.

Use the benchmark–specific scoring dimensions to locate opportunities, not as causal proof. A low score or long trajectory alone is not a prompt defect. Connect the cited evidence, proposed behavior change, and intended improvement; do not invent a numerical reward gain.

Do not reduce cost by skipping task requirements, truncating necessary search, removing required verification, or stopping early. The objective guides candidate selection; do not copy it into the Worker's instructions as permission to trade away task requirements.

C Experimental Settings

C.1 Benchmark selection

AgentBench evaluates agents across diverse interactive environments (Liu et al., 2024b). More specialized benchmarks cover web navigation (Zhou et al., 2024b), desktop workflows (Xie et al., 2024a), and API use (Qin et al., 2024). AgentBoard complements final success with progress-based analysis (Ma et al., 2024a), motivating examination of the execution process as well as task outcomes.

We choose AppWorld and the Telecom Solo setting of τ^2-Bench because, within each benchmark, training and evaluation tasks share the same tool interfaces while presenting different task instances and scenarios. This setting directly matches AutoPEER's goal: learning reusable system instructions and tool-use rules from training trajectories and applying them to unseen tasks. Shared tools make

the learned rules applicable at evaluation time, while changes in tasks and scenarios test their generalization. Both benchmarks provide executable outcome checks to measure task completion.

AppWorld. AppWorld (Trivedi et al., 2024) requires agents to understand APIs, generate code, and perform operations across applications. Decisions within a task depend on one another, and the agent prompt guides tool selection, information gathering, and execution throughout the episode. These tasks therefore test whether AutoPEER can identify problems in long trajectories and turn targeted prompt changes into complete-task improvements. Normal and Challenge provide different difficulty levels. Related task variants within each scenario further test whether learned rules remain useful under different task conditions.

τ^2-Bench Telecom Solo. Telecom Solo (Barres et al., 2026) extends the evaluation to troubleshooting across customer-service and device-side tools, testing AutoPEER beyond interactive coding. The original τ-bench evaluates policy-guided tool use through conversations with simulated users (Yao et al., 2025). In Telecom Solo, the agent receives a task ticket and directly operates both tool sets without a simulated user. The comparison can therefore focus on the agent's tool decisions and prompt optimization while retaining multi-step interaction and changes in environment state.

C.2 Baselines and training feedback

Prompt optimization scope. RLPrompt learns a policy for generating discrete prompts (Deng et al., 2022), PACE uses actor-critic editing (Dong et al., 2024), and Revolve incorporates response changes across optimization steps (Zhang et al., 2025a). These approaches use reward or textual feedback to guide different forms of prompt search. Our comparison uses GEPA for reflective search and ACE for accumulated experience in the same tool environments.

Baseline. The Baseline uses the unoptimized starting prompt and provides a reference for measuring the benefit of automatic optimization. Within each experimental setting, methods use the same Worker, tool interfaces, and execution framework, allowing us to compare different ways of optimizing prompts.

GEPA. GEPA (Agrawal et al., 2026) is a recent reflective prompt optimizer. It uses execution traces and feedback to propose revised prompts, then applies Pareto-based candidate selection to retain strategies that perform well on different tasks. We include GEPA to compare general-purpose reflective prompt search with AutoPEER's approach to long-horizon agents. GEPA can also optimize agents, but its candidate selection requires repeatedly collecting validation trajectories; the authors report that validation accounts for most of its trajectory-generation budget. In long-horizon tasks, collecting each trajectory requires repeated model calls, tool operations, and environment interactions, making broader validation coverage costly. This comparison examines how to propose and test prompt improvements when collecting complete trajectories is expensive and validation budgets are limited.

ACE. ACE (Zhang et al., 2026) directly addresses context adaptation for agents. Its Reflector analyzes execution experience, and its Curator incrementally integrates reusable strategies into an evolving playbook. We include ACE to compare two ways of using execution experience: accumulating and organizing strategies in context, and making targeted revisions to the agent prompt around specific Issues while verifying their behavioral effects. The comparison examines both task performance and growth in the resulting prompt.

Reflection-GT. Reflection-GT is a direct-rewriting baseline. The optimizer reads the current prompt, training trajectories, and reference solutions, then decides whether to rewrite or retain the prompt without structured Issue generation or progressive behavioral verification. This baseline measures what direct prompt revision from training feedback can achieve, providing a comparison with AutoPEER's complete optimization procedure.

Training feedback. Following ACE (Zhang et al., 2026), GT indicates whether ground-truth information is available to the optimizer during training; no-GT methods do not receive this information.

In AppWorld, ground-truth information is the current training task's solution code. AutoPEER-GT, ACE-GT, and Reflection-GT receive the same reference solutions: AutoPEER-GT provides them to Issue diagnosis and patch generation, ACE-GT to the Reflector, and Reflection-GT to the rewrite-or-keep optimizer. ACE's Curator uses the resulting reflections. This setting is shared across both Workers. In τ^2-Bench Telecom Solo, AutoPEER-GT and ACE-GT receive reference tool actions with their arguments and checks on the required environment state. In both GT and no-GT settings, the optimizer uses execution trajectories and environment feedback. During testing, all final prompts remain fixed, and Workers receive only the task, current prompt, and allowed environment observations.

C.3 Implementation settings

Models and adaptation schedule. Workers are DeepSeek-V4-Flash and Qwen3.5-35B-A3B on AppWorld, and Qwen3.5-35B-A3B on Telecom Solo; GLM-5.2 performs optimization. Decoding is non-thinking with temperature zero. System instructions and tool descriptions are editable, while tool schemas, implementations, the evaluator, and the agent protocol remain fixed. On AppWorld, AutoPEER runs for 30 generations with batch size $m = 3$ and a Rolling history of $K = 3$ batches. Candidate validation uses the current batch; Rolling grows from three to six to nine tasks. GEPA uses 30 fixed Dev tasks for selection. Normal contains 168 tasks in 56 scenarios, and Challenge contains 417 tasks in 139 scenarios.

Table 5: **AutoPEER settings on AppWorld and Telecom Solo.** Output-token limits apply to each Optimizer response. Telecom's final training batch contains two tasks.

Parameter	AppWorld	Telecom Solo
Optimizer model	GLM-5.2	GLM-5.2
Temperature	0	0
Thinking	Disabled	Disabled
Maximum output tokens	16,384	8,192
Output format	JSON	JSON
Training tasks per generation	3	3
Maximum Issues per generation	3	3
Candidates per Issue	3	3
Maximum replay decisions per Issue	3	3
Paired replay trials per decision	3	3
Maximum Rolling tasks	9	9
Optimization generations	30	25

Telecom Solo reward. For Telecom Solo, $r(e) = U - 0.01M - 0.002A - 0.05I - 0.25E$, where U is the task reward from the benchmark evaluator; M, A, and I count interaction messages, tool calls, and invalid actions in episode e; and E is 1 if the episode terminates because of repeated errors (`too_many_errors`) and 0 otherwise.

The success-only ablation sets the non-task weights to zero and ranks candidates by completed tasks. It also removes guidance about the other scoring dimensions from the optimizer request, while retaining the verification stages and the fixed agent protocol.

AppWorld training and evaluation tasks. Both Workers use the same 90-task AppWorld Train pool. AutoPEER visits each scenario once, grouping its three variants into one training batch. Candidate verification and Rolling revisit these tasks. ACE visits the pool one task at a time in official file order; Reflection processes three-task batches without replacement. GEPA draws reflection examples from Train and selects candidates on Dev. Normal and Challenge are evaluation sets, with policies fixed throughout testing.

AppWorld baselines. GEPA uses three-example reflection minibatches, instance-level Pareto selection, round-robin module selection, and candidate merging. Each Worker run records 45

optimization iterations and 801 metric calls, including the initial 30-task Dev evaluation. The evaluated prompt has the highest mean Dev reward among candidates from the complete run.

ACE starts from an empty playbook and makes one pass over Train. Both GT and no-GT allow up to five attempts per task and stop after success. GLM-5.2 serves as Reflector and Curator. Reflection-GT processes 30 three-task batches and asks GLM-5.2 to rewrite or retain the prompt, without Issue-specific branches or progressive verification. Neither baseline updates its prompt during testing.

Telecom Solo. The 74 training tasks are processed in 25 AutoPEER generations, using three-task batches and a final two-task discovery batch. The system instructions and 43 tool descriptions are editable; the task ticket, domain policy, and tool interfaces remain fixed. GEPA partitions the training pool into 64 Train and 10 Dev tasks and uses a 600-metric-call budget. ACE makes one pass over all 74 tasks, allowing up to four attempts per task with early stopping on success. Training uses a 50-step limit. Table 2 averages three 40-step evaluations of the same 40 held-out tasks for each fixed final prompt.

Completion metrics. AppWorld completion counts tasks that terminate successfully within 40 steps. TGC is the percentage of completed tasks; AppWorld SGC requires all three variants of a scenario to succeed. Telecom reports task success under its executable evaluator and does not use SGC.

D Additional Experimental Results

D.1 Paired outcomes for component ablations

Table 4 reports completion on both Normal and Challenge. Here we compare which tasks each variant completes.

Ablation variants. *w/o Issue* generates patches directly from trajectories and proceeds to full-episode validation, omitting structured diagnosis and its local replay. *w/o Fast Replay* retains Issue-guided patch generation and sends all valid candidates directly to full-episode validation. *w/o Merge & Rolling* adopts a single patch after full-episode validation. *w/o Multidimensional Reward* uses task success for candidate scoring and optimizer guidance. All variants retain the same agent interface and safety constraints; implementation settings are listed in Appendix C.

The differences are larger on Challenge. Compared with *w/o Issue*, AutoPEER-GT succeeds alone on 51 tasks, while the ablation succeeds alone on 29. The corresponding counts are 61 and 33 for *w/o Merge & Rolling*, and 60 and 21 for *w/o Multidimensional Reward*. On Normal, the counts are 13 and 11, 17 and 4, and 11 and 9, respectively. These counts compare the two methods on the same tasks.

Equal totals, different successful tasks. Full and *w/o Fast Replay* each complete 145/168 Normal tasks and 318/417 Challenge tasks. They share 135 Normal successes and each has 10 exclusive successes; 13 tasks fail under both prompts. On Challenge, 277 successes are shared, each prompt has 41 exclusive successes, and both fail on 58 tasks. Thus, outcomes differ on 20 Normal and 82 Challenge tasks despite equal TGC. At the scenario level, both complete 43 Normal scenarios, sharing 39. On Challenge, Full completes 86 scenarios and the no-Fast-Replay prompt 79: 67 are shared, with 19 and 12 exclusive completions.

Candidate comparisons. The DeepSeek GT run executes 112 full-task candidate comparisons: 103 candidates pass Fast Replay, and nine proceed directly to full-episode validation because local replay is unavailable. Each comparison evaluates one candidate against the Parent on a task batch. The w/o Issue variant executes 239 such comparisons. The shares improving reward are 59.8% and 51.0%, respectively. Candidate totals reflect each run's proposal limits and verification stages.

Table 6 compares the full method with the reported no-Fast-Replay run. Candidate generation is similar in scale, but fewer candidates enter full-episode validation with Fast Replay. Each tested candidate is evaluated on three current-batch tasks.

Table 6: **Full-task validation workload with and without Fast Replay.** Actual counts from the DeepSeek-V4-Flash GT runs. Episodes count current-batch candidate validation, excluding discovery, Parent refresh, and Rolling.

Metric	AutoPEER-GT	w/o Fast Replay
Valid generated candidates	128	127
Candidates entering full-episode validation	112	127
Candidate-validation episodes	336	381

Paired prompt-transfer outcomes. We match the transferred and same-Worker GT policies on all 417 Challenge tasks. On Qwen, both solve 165 tasks, the transferred policy alone solves 66, and the same-Worker policy alone solves 41. Of 139 scenarios, both complete 29, with 20 and 13 exclusive completions, respectively. On DeepSeek, both solve 250 tasks, with 19 transfer-only and 68 same-Worker-only successes. They share 61 completed scenarios, with six and 25 exclusive completions. The task and scenario comparisons therefore show the same asymmetry, while also revealing useful coverage unique to each policy.

D.2 Targeted revisions limit prompt growth

Long-context evaluations show that information use depends on where relevant content appears (Liu et al., 2024a). This motivates examining how instructions are organized and retained, alongside their total length and task performance.

Later Issues can lead to changes in earlier rules. In the DeepSeek runs, replacements account for 9 of 13 adopted edit operations without GT and 14 of 19 with GT. In the GT run, a rule for checking artist names illustrates this process. An earlier update requires a separate lookup; later trajectories show that some search results already contain the needed information. An adopted patch changes the tool description to distinguish these sources and require further checking only when needed. New Issues thus refine the conditions of existing rules. Appendix E presents the no-GT prompt and its adopted revisions.

Table 7: **Prompt growth with ACE and AutoPEER.** AutoPEER adds less text under each Worker and feedback condition. Values are net added characters in system instructions and tool descriptions (final minus baseline). GT indicates whether ground-truth information is available to the optimizer during training. Bold marks the smaller addition within each condition.

Method	GT	DeepSeek	Qwen
ACE	–	+59,271	+103,583
AutoPEER	–	**+4,801**	**+5,692**
ACE	✓	+33,731	+80,037
AutoPEER	✓	**+8,213**	**+10,826**

AutoPEER improves completion with much less added text. ACE organizes experience into a playbook; AutoPEER makes verified edits to the agent prompt. Table 7 compares each final prompt with its own starting prompt. Across both Workers and feedback settings, AutoPEER adds 75.7–94.5% less text than ACE. For DeepSeek with GT, the additions are 8,213 versus 33,731 characters, with Challenge TGC of 76.26% versus 69.78%. The edit history shows how AutoPEER revises existing rules, while these results show that it can improve completion without adding as much text as ACE.

D.3 Training budget and completion

We separate full-task execution from experience analysis and prompt revision. The DeepSeek runs use 801 full-task evaluations for GEPA, compared with 576 for AutoPEER without GT and 612 with GT. For entries without †, the statistics use recorded full task episodes and persisted Optimizer responses. Table 8 reports DeepSeek-V4-Flash Worker usage; Table 9 reports GLM-5.2 Optimizer usage. Cost equivalents aggregate these two categories using the same cache-adjusted prices as Figure 5.

Table 8: **Worker usage during AppWorld optimization.** DeepSeek-V4-Flash executes full task episodes. Tokens (M) counts input and output in millions; USD gives fixed-price cost equivalents. † marks estimated usage.

Method	GT	Tokens (M)	USD
GEPA†	–	136.78	5.296
ACE	–	32.37	1.003
ACE	✓	21.24	0.712
Reflection	✓	15.26	0.591
AutoPEER	–	93.25	3.567
AutoPEER	✓	81.54	3.499

Table 9: **Optimizer usage during AppWorld optimization.** GLM-5.2 analyzes experience and revises prompts. Tokens (M) counts input and output in millions; USD gives fixed-price cost equivalents. † marks estimated usage.

Method	GT	Tokens (M)	USD
GEPA†	–	1.23	1.737
ACE	–	6.13	8.917
ACE	✓	4.50	6.544
Reflection	✓	1.06	1.604
AutoPEER	–	1.70	2.636
AutoPEER	✓	2.06	2.909

M denotes one million input and output tokens. GT indicates whether ground-truth information is available to the optimizer during training. USD converts the reported usage at fixed rates. Worker cache-hit input, cache-miss input, and output cost \$0.007/\$0.22/\$0.66 per million tokens; Optimizer rates are \$0.26/\$1.40/\$4.40. The statistics aggregate successful and unsuccessful recorded task episodes and persisted Optimizer responses. † marks GEPA's estimated usage for its 801-metric-call, 45-iteration run.

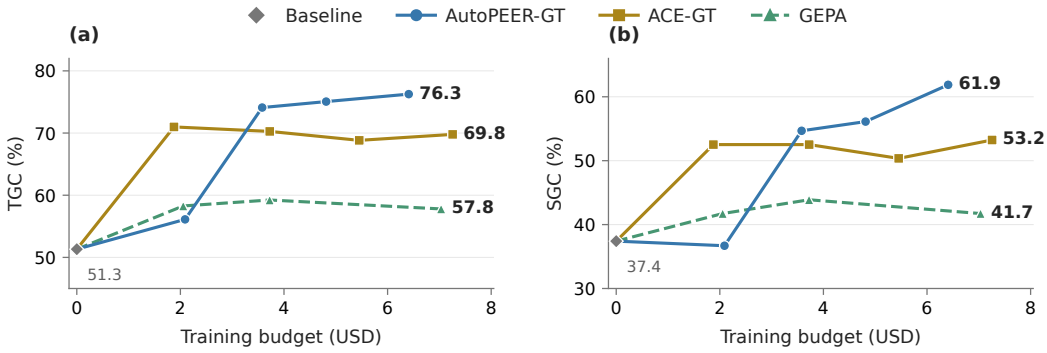


Figure 5: **Training budget and completion on AppWorld Challenge.** DeepSeek-V4-Flash is evaluated on 417 tasks with the 40-step reporting criterion: (a) TGC; (b) SGC. All curves start from the shared baseline. Markers denote evaluated checkpoints, connected without smoothing. The horizontal axis sums the Worker and Optimizer USD equivalents reported in Tables 8 and 9. GEPA checkpoint estimates are proportional to cumulative episode counts.

The split distinguishes two uses of the reported training budget. ACE-GT's Optimizer and Worker amounts are \$6.54 and \$0.71; AutoPEER-GT's are \$2.91 and \$3.50, respectively. The curves also separate early gains from later improvement: ACE-GT reaches high completion early, while AutoPEER-GT improves substantially at the intermediate checkpoints and continues to improve scenario completion.

E Baseline and Evolved Prompts

What does the agent learn without reference execution trajectories? We follow the DeepSeek no-GT AutoPEER prompt used in the main comparison. Nine of its 30 generations produce adopted updates, changing system instructions and three tool descriptions. The boxes below show the baseline prompt components and the revisions retained in the final prompt. Appendix F places them in the full sequence of accepted and rejected updates.

Experience can also be represented as executable skills, as in Voyager (Wang et al., 2024a) and ASI (Wang et al., 2025a). Here, the retained artifacts are system instructions and tool descriptions.

The boxed text reproduces the recorded prompt text. The panels follow Figure 2: dark title bars identify each prompt, with a uniform off-white background for baseline text and adopted revisions. Generation labels appear in title bars, outside the prompt. Where a passage is revised more than once, its introduction and subsequent revision are identified together.

E.1 BASELINE SYSTEM INSTRUCTIONS

The following is the complete baseline editable system prompt for the DeepSeek no-GT run.

Baseline system prompt.

PE–01
Reply with exactly one fenced Python code block per turn and no extra prose. Only the first code block is executed, so make small testable steps, use each execution result as evidence for the next step, reuse persisted Python variables, and stay within the step budget.

PE–02
Before first using any API, inspect its exact schema with `apis.api_docs.show_api_doc(...)`; use `show_api_descriptions` or `search_api_docs` first if needed. Call apps only through `apis` and Python standard library helpers, never OS/system tools or third–party app packages, and pass only documented parameters because extra arguments are rejected.

PE–03
Get personal profile data, addresses, payment cards, and app passwords from `apis.supervisor`; resolve friends, family, and other people through `apis.phone` contacts. Inspect each login API before authenticating, use the documented identifier, store passwords and access tokens in variables, avoid printing secrets, and remember redacted output does not erase stored variables.

PE–04
For any list/search/history API that supports `page_index`, read all pages instead of assuming page 0 is complete; stop only when the response shows exhaustion (for example an empty page). Use detail/show APIs to disambiguate targets before acting, and for time–based tasks use `datetime.now()` or `apis.phone.get_current_date_and_time()` with full–day boundaries for terms like today or yesterday.

PE–05
Prefer read–only discovery before mutation. Do not ask for clarification; if the task leaves a free choice unspecified, choose any valid option from retrieved data. Perform only the minimum changes required, and avoid collateral actions such as broad deletes, clears, account changes, or status changes unless explicitly requested. Treat 'file system' as the `file_system` app, not the OS filesystem.

PE–06
If a call fails or results look wrong, do not guess. Inspect the error/output, re–check the API doc, print the relevant returned objects, correct parameter names/types/constraints, and retry in a bounded way. Never invent IDs, emails, dates, or other values; if authentication may be stale, re–read credentials and log in again.

PE–07
Before finishing, independently verify the final state with fresh read APIs instead of trusting a write call alone. For answer tasks, compute the answer from verified current data and keep it minimal: only the requested entity/value, using numerals for numbers.

PE–08
After verification, call `apis.supervisor.complete_task(...)` with the correct form: pass `answer=` only when the task explicitly asks for an answer, otherwise leave it unset/None. The task is not done until that call succeeds; if you are truly blocked after evidence–based attempts, call `apis.supervisor.complete_task(status='fail')`.

E.2 EVOLVED SYSTEM INSTRUCTIONS, WITH GENERATION LABELS

The response-format and read-only-first paragraphs remain unchanged. The following passages reproduce the remaining system instructions in their final order, with adopted generations in the title bars.

Batched documentation queries (generation 13).

PE–02
Before first using any API, inspect its exact schema with `apis.api_docs.show_api_doc(...)`. Call `show_api_descriptions` or `search_api_docs` first to identify the relevant APIs, then batch all needed `show_api_doc` calls in a single Python code block rather than issuing one call per turn. For example:
```
python
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='login'))
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='show_liked_songs'))
print(apis.api_docs.show_api_doc(app_name='spotify', api_name='review_song'))
```
Call apps only through `apis` and Python standard library helpers, never OS/system tools or third–party app packages, and pass only documented parameters because extra arguments are rejected.

App-specific login identifiers (generation 27).

PE–03
Get personal profile data, addresses, payment cards, and app passwords from `apis.supervisor`; resolve friends, family, and other people through `apis.phone` contacts. Inspect each login API before authenticating, use the documented identifier, store passwords and access tokens in variables, avoid printing secrets, and remember redacted output does not erase stored variables.WARNING: The `username` parameter does NOT always mean email. Different apps require different identifiers – for example, the phone app requires a phone_number, not an email. You MUST read the `username` parameter's `description` field in the login API doc for EACH app before calling login, and retrieve the matching identifier type from `apis.supervisor` profile data. Never assume the identifier type based on another app's login schema.

Filtering during pagination (generation 7).

PE–04
For any list/search/history API that supports `page_index`, read all pages instead of assuming page 0 is complete; stop only when the response shows exhaustion (for example an empty page). When you already know the filter you will apply to the results (e.g., a target date or a specific set of contacts), apply that filter to each page immediately as you fetch it and collect only the matching items. Stop paginating as soon as a full page yields zero matching items, since further pages are unlikely to contain relevant results. This avoids fetching and storing thousands of items before filtering. Use detail/show APIs to disambiguate targets before acting, and for time–based tasks use `datetime.now()` or `apis.phone.get_current_date_and_time()` with full–day boundaries for terms like today or yesterday.

Discovery before detail documentation (generation 23).

PE–05b
Before inspecting schemas for task–specific detail APIs (e.g., `show_album`, `show_playlist`, `show_song`, `play_music`), first use a search or list API (e.g., `search_albums`, `show_playlist_library`, `show_song_library`) to locate the target entity. Only after the target is found should you call `show_api_doc` for the detail/action APIs you will use next. You may still inspect `login` and the first search/list API schema upfront, but do not pre–fetch schemas for detail or action APIs before discovery.

Union and intersection (generation 2).

PE–05a
When a task says to keep items that satisfy condition A "or" condition B, combine the two ID sets with Python union (`|`), not intersection (`&`). When a task says to keep items that satisfy condition A "and" condition B, combine the two ID sets with intersection (`&`). Before writing the set–combination code, state which operator the task wording requires and why.

Handling already-completed actions (generation 24).

PE–06
PE–06a
When a mutation API (e.g., like, follow, add, create) returns a 422 error indicating the resource already exists or the action was already performed (e.g., "You have already liked this song"), treat this as a successful completion of that action. Do not retry the same call. If you are verifying a batch of actions and some items are missing from the result list, check whether those items were already in the desired state before the task started; a 422 'already done' error confirms the item is in the correct state.

If a call fails or results look wrong, do not guess. Inspect the error/output, re–check the API doc, print the relevant returned objects, correct parameter names/types/constraints, and retry in a bounded way. Never invent IDs, emails, dates, or other values; if authentication may be stale, re–read credentials and log in again.

Exact output paths (generation 12).

PE–07
Before finishing, independently verify the final state with fresh read APIs instead of trusting a write call alone. For answer tasks, compute the answer from verified current data and keep it minimal: only the requested entity/value, using numerals for numbers. When a task specifies an exact output file path, before calling any file–writing API, confirm that the file_path parameter matches the task–specified path character–for–character; after writing, verify the returned file_path matches the task–specified path. Do not invent or substitute filenames.

Completion after sending messages (generation 6).

PE–08
After verification, call `apis.supervisor.complete_task(...)` with the correct form: pass `answer=` only when the task explicitly asks for an answer, otherwise leave it unset/None.

∗∗Critical Example∗∗: If a task asks you to "reply" or "send" something (e.g., a text message, email), the act of sending IS the task. Do NOT pass the sent content as `answer=`. For example, if you send a text message with movie titles, `complete_task(answer='Movie A, Movie B')` is WRONG; call `complete_task()` with no answer argument.

The task is not done until that call succeeds; if you are truly blocked after evidence–based attempts, call `apis.supervisor.complete_task(status='fail')`.

Delimited text files (generation 12).

PE–09
When a task requires writing CSV or other delimited text files, do NOT use Python's `csv` module (`import csv`, `csv.writer`, `csv.reader`) – it is blocked by the sandbox and will cause an execution failure. Instead, build the file content as a Python string: create a header line, append one line per data row using `','.join([str(field) for field in row])`, then combine all lines with `'\n'.join(lines)`. Pass the resulting string directly as the `content` parameter to `apis.file_system.create_file(...)`.

E.3 Baseline tool descriptions at the edited locations

Three of the eleven tool descriptions change during this run. Their baseline texts are shown together; the other eight remain unchanged.

Baseline descriptions of the three edited tools.

api_docs:
API documentation explorer: list apps, list an app's APIs, and inspect exact request/response schemas before calling anything.

spotify:
Music streaming and library app for songs, albums, playlists, queue/player controls, likes/follows/reviews, downloads, payment cards, and premium history.

venmo:
Social payments app for friends, balance, payment cards, transactions, requests, comments, notifications, feed, bank transfers, and receipts; unlike Splitwise, it moves money.

E.4 Evolved tool descriptions

API documentation tool (generation 13).

API documentation explorer: list apps, list an app's APIs, and inspect exact request/response schemas before calling anything. When you need schemas for multiple APIs, issue all `show_api_doc` calls in a single Python code block to conserve your API call budget.

Spotify tool (generations 1 and 24).

Music streaming and library app for songs, albums, playlists, queue/player controls, likes/follows/reviews, downloads, payment cards, and premium history. The `like_song` API is idempotent: if the song is already liked, it returns a 422 error with message "You have already liked this song in the past." This means the song is already in the liked state; do not retry. Note that `show_liked_songs` is paginated with a default page_limit of 5; always paginate to fetch all liked songs before concluding a song is missing. Only pass `access_token` to Spotify APIs whose own schema lists it as a parameter (e.g., `follow_artist`, `show_following_artists`); read–only search and list APIs such as `search_artists`, `search_albums`, and `show_genres` do not accept `access_token`.

Venmo tool (generation 7).

Social payments app for friends, balance, payment cards, transactions, requests, comments, notifications, feed, bank transfers, and receipts; unlike Splitwise, it moves money. When using `show_social_feed` with a known date or contact filter, apply the filter to each page as you fetch it and stop paginating once a page yields no matching items, rather than fetching the entire feed first.

F A Complete Prompt-Evolution Run

The complete DeepSeek no-GT history shows how verification selects updates over time. Updates are retained in generations 1, 2, 6, 7, 12, 13, 23, 24, and 27. Rolling rejects the update in fourteen generations, six have no winning candidate, and one produces no Issue. Generations that retain the Parent continue with the same prompt.

How one instruction develops. Generation 13 introduces batched API documentation queries. Generation 23 adds a condition: locate the target entity before inspecting the schemas for detail or action APIs. Generation 30 proposes replacing batching with one-at-a-time, just-in-time schema inspection. This candidate improves reward on the current batch but lowers reward on the Rolling window, so the Parent is retained. The final prompt therefore keeps batching together with the later discovery-first rule. The history shows how new trajectory evidence can refine earlier instructions, while verification determines whether a proposed alternative replaces them.

Reading the history. Each record shows the proposed Issue and whether its update was accepted. An adopted update changes the agent prompt used in later generations. A rejected update stays in the record but does not enter the prompt. Together with the passages in Appendix E, the history shows when new instructions were added, when existing rules changed, and when a proposed change was rejected.

Generation 1 — ADOPTED

Schema-specific authentication parameters. The adopted patch revises the Spotify description to pass access_token only when an API's own schema lists it, distinguishing search endpoints from authenticated operations.

Generation 2 — ADOPTED

Distinguish union from intersection. Issues identify union-versus-intersection errors when retaining liked or downloaded songs and albums. The adopted system rule maps 'or' to set union and 'and' to intersection, requiring the agent to state the intended operator before coding.

Generation 3 — REJECTED BY ROLLING

Reduce schema inspection. The optimizer proposes calling APIs with apparently self-evident parameters without inspecting their full schemas, reserving inspection for unfamiliar or failed calls. Rolling rejects the update.

Generation 4 — REJECTED BY ROLLING

Documentation lookup and song-detail reuse. The optimizer examines repeated schema inspection and individual song-detail reads. The selected patch relaxes inspection for self-descriptive Spotify APIs, but fails Rolling.

Generation 5 — PARENT RETAINED

Recommendation API discovery. The optimizer considers reducing documentation discovery before requesting Spotify recommendations. No candidate replaces the Parent.

Generation 6 — ADOPTED

Complete messaging tasks without an answer. The adopted patch adds a contrastive example to the completion rule: sending or replying to a message is an action task, so the sent content should not also be passed as answer to complete_task.

Generation 7 — ADOPTED

Filter social-feed pages as they arrive. The adopted patch changes the system and Venmo descriptions to apply known date or contact filters while fetching each page. It also instructs the agent to stop pagination after a page has no matching items.

Generation 8 — PARENT RETAINED

Documentation before alarm updates. The optimizer targets documentation calls made before authentication and alarm updates. No candidate replaces the Parent.

Generation 9 — REJECTED BY ROLLING

File parameters and metadata reads. Issues concern guessed file-system parameter names and repeated per-file metadata reads. The selected patch lists file-system parameter names in the tool description, but Rolling rejects it.

Generation 10 — REJECTED BY ROLLING

Time, documentation, and phone login. The optimizer examines host-clock use, repeated schema inspection, and phone login with an email address. The selected system and api_docs edits reduce schema inspection, but fail Rolling.

Generation 11 — REJECTED BY ROLLING

Login identifiers and contact API names. The optimizer combines a per-app login-identifier checklist with an explicit reference to phone.search_contacts instead of show_contacts. Rolling rejects the combined update.

Generation 12 — ADOPTED

CSV construction and exact output paths. The adopted update instructs the agent to construct CSV text with string operations instead of the csv module. It also requires the output file path to match the task exactly, with checks before and after writing.

Generation 13 — ADOPTED

Batch schema inspection. The adopted update changes the system and api_docs descriptions to gather needed schemas in one Python code block after discovering relevant APIs. It retains inspection before first use.

Generation 14 — REJECTED BY ROLLING

Retrieve app-specific login identifiers. The optimizer proposes retrieving passwords and profile identifiers before login, specifying a phone number for phone and an email for several other apps. Rolling rejects this revision.

Generation 15 — REJECTED BY ROLLING

Environment time and file discovery. The optimizer examines host-clock use and guessed file paths. The selected patch strengthens the instruction to obtain time from the simulated environment, but fails Rolling.

Generation 16 — REJECTED BY ROLLING

Cross-turn variables and repeated discovery. Issues concern a cross-turn NameError and repeated documentation lookup. The selected patch proposes treating code blocks as isolated namespaces and re-fetching prior data; Rolling rejects it.

Generation 17 — NO ISSUE; UNCHANGED

No diagnosed Issue. No evidence-grounded Issue is returned, so the prompt remains unchanged.

Generation 18 — REJECTED BY ROLLING

Discovery phases and song-data reuse. The optimizer combines a strict inspect-then-authenticate workflow with Spotify guidance to reuse fields already returned by list APIs. Rolling rejects the combined update.

Generation 19 — PARENT RETAINED

Limit preliminary documentation calls. The optimizer revisits documentation calls made before task actions. No candidate replaces the Parent.

Generation 20 — REJECTED BY ROLLING

Verify login identifiers and API names. The optimizer combines a login-identifier checklist with a requirement to obtain API names from documentation listings before requesting schemas. Rolling rejects the combined system edits.

Generation 21 — REJECTED BY ROLLING

Use an existing valid item. The optimizer proposes stopping search once an owned item meets the task's explicit criteria, alongside guidance against guessed API names. Rolling rejects the combined update.

Generation 22 — PARENT RETAINED

Reuse song details across filtering steps. The optimizer targets repeated show_song calls used first for genre filtering and then for artist IDs. No candidate replaces the Parent.

Generation 23 — ADOPTED

Locate targets before inspecting detail APIs. The optimizer examines premature schema inspection and repeated searches across music collections. The adopted system rule defers detail and action schemas until a search or list API has located the target, while allowing login and initial discovery schemas upfront.

Generation 24 — ADOPTED

Already-completed actions and paginated verification. Issues concern queue selection, repeated likes, and incomplete verification reads. The adopted update treats 'already done' mutation errors as completed actions without retrying, and adds Spotify guidance to paginate liked-song verification.

Generation 25 — REJECTED BY ROLLING

Check temporal filter direction. Issues identify reversed and overly inclusive date filters before song removal. The selected patch requires checking sample items and the temporal boundary before mutation, but fails Rolling.

Generation 26 — PARENT RETAINED

Reuse credentials and documentation. The optimizer targets repeated password retrieval and schema inspection before the main task logic. No candidate replaces the Parent.

Generation 27 — ADOPTED

Read each app's login identifier. Issues concern phone login, a nonexistent contact API, and a cross-turn variable error. The adopted system warning requires reading each login schema's username description and retrieving the matching identifier from supervisor profile data.

Generation 28 — PARENT RETAINED

Private-playlist verification. The optimizer examines missing authentication and a guessed response field when verifying a private playlist. No candidate replaces the Parent.

Generation 29 — REJECTED BY ROLLING

Separate discovery from detail inspection. The optimizer proposes a stricter two-phase version of the generation-23 rule, separating discovery schemas from detail and action schemas in both system and api_docs guidance. Rolling rejects the revision, retaining the earlier rule.

Generation 30 — REJECTED BY ROLLING

Just-in-time schema inspection. The optimizer proposes verifying API names and inspecting only the single API to be called next, replacing the earlier batching instruction. Rolling rejects the final proposed update, leaving the generation-27 prompt unchanged.

G LIMITATIONS AND FUTURE DIRECTIONS

Our experiments use benchmark training and test splits, and AutoPEER has not yet been evaluated on real production tasks. AutoPEER makes localized changes rather than rewriting the entire prompt, and links each proposed change to its Issue, trajectory evidence, and validation results. These records allow developers to inspect what changed, why it was proposed, and what evidence supports its adoption before deciding which updates to deploy. An important next step is to evaluate AutoPEER in production workflows, combining human review with behavioral verification to assess whether these traceable updates support effective and controlled prompt evolution.

Feedback and replay conditions. AutoPEER relies on observable execution traces and feedback that distinguishes useful behavior from errors. Fast Replay also depends on reproducing recorded interactions in a task environment. This is practical in the evaluated benchmarks, but live tools may have irreversible effects or changing external state. Extending verification to these settings will require safe test environments or suitable simulators.

Quality of proposed repairs. The Issue Generator and Patch Generator determine which repairs enter verification. They can miss a consequential decision or propose an overly narrow rule. Behavioral tests can reject an ineffective patch, but cannot recover a useful change that was never proposed. Broader proposal search and targeted human review could help address this limitation.

Validation coverage and transfer. Rolling tests recent tasks rather than the full task distribution. A retained rule can still fail on cases outside this window. Our experiments cover AppWorld and τ^2-Bench Telecom with shared tool interfaces between training and evaluation; cross-Worker transfer also retains the same environment. Larger or adaptively selected validation windows and evaluation with changing tools are natural next steps. Continuous task-and-feedback streams, as studied by StreamBench (Wu et al., 2024), also offer a setting for evaluating ongoing prompt updates.