AT-AT: AUTOMATIC TOOL OPTIMIZATION FROM AGENT TRACES

Anonymous authors
Paper under double-blind review

ABSTRACT

Efforts to improve LLM-based agents typically focus on modifying the agent itself, yet success also depends on the environment in which an agent acts. Tools mediate this interaction: a tool's description shapes how an agent calls it, and its implementation determines the results and errors the agent receives afterwards. Many tools, such as APIs and MCP servers, are shared by many agents and maintained by providers who cannot change those agents. We introduce AT-AT (Automatic Tool Optimization from Agent Traces), an agentic tool optimizer that improves a shared toolset offline from historical agent executions while holding the agents fixed. A lightweight coding agent reads agent trajectories and their evaluation results, and selectively edits tool descriptions and, when permitted, implementations. On τ^2-bench-verified, AT-AT achieves the highest task completion rates in the Airline and Telecom domains at lower cost than strong prompt- and tool-optimizer baselines, and its Telecom gain is statistically significant under paired permutation tests. Optimizing the tools within a coding agent harness also improves held-out performance on TerminalBench-2 and TBLite. On three enterprise MCP servers in production, applying AT-AT to production logs increased held-out task passes on two servers and reduced tool errors on the third. By turning accumulated execution traces into updates to shared tool environments, AT-AT provides a path toward continual improvement of agent systems through environment-side learning.

1 INTRODUCTION

Efforts to improve LLM-based agents typically modify the agent itself: training the underlying model, optimizing its prompts (Agrawal et al., 2026; Kassianik et al., 2026), or engineering the harness that manages its execution (Lee et al., 2026b; Zhang et al., 2026; Weng, 2026). However, an agent's success is a property not only of the agent, but also of its interaction with the external environment. Tools mediate this interaction, as they define the actions available to the agent and the information returned in response. We study a complementary path to better agent performance: improving the tool environment from accumulated agent experience based on historical execution traces, while holding the agent fixed.

The opportunity for such optimization is large and growing. In an enterprise, an agent's tool environment may include Model Context Protocol (MCP) servers spanning many internal systems. More broadly, nearly any software API can be exposed as an agent-callable tool. Most APIs, however, were designed for human developers or conventional software rather than for the capabilities and limitations of LLM agents. Cloud services can now automatically translate existing APIs into agent-callable tools (Microsoft, 2026; Wadhwa et al., 2025), but callability does not imply usability by LLMs. An API's agent-facing description, schema, outputs, and errors remain consequential design choices that we aim to automatically optimize. These tools are typically maintained by a provider who cannot change the many agents that call them, but any change the provider makes to a tool reaches all of those agents. This is our primary setting.

A tool has two parts. Its *description*, comprising its name, parameter schema, and usage notes, shapes the agent's expectations *before* a call, while its *implementation* determines the results and errors the agent receives *after* it. Because tool results enter the agent's context, actionable errors can enable recovery and relevant outputs can support better subsequent decisions. We call this mechanism *feedback shaping*: improving the information returned by tools without changing the agent or its evaluation objective. For example, a τ^2-bench-verified Telecom tool that reports a line's

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

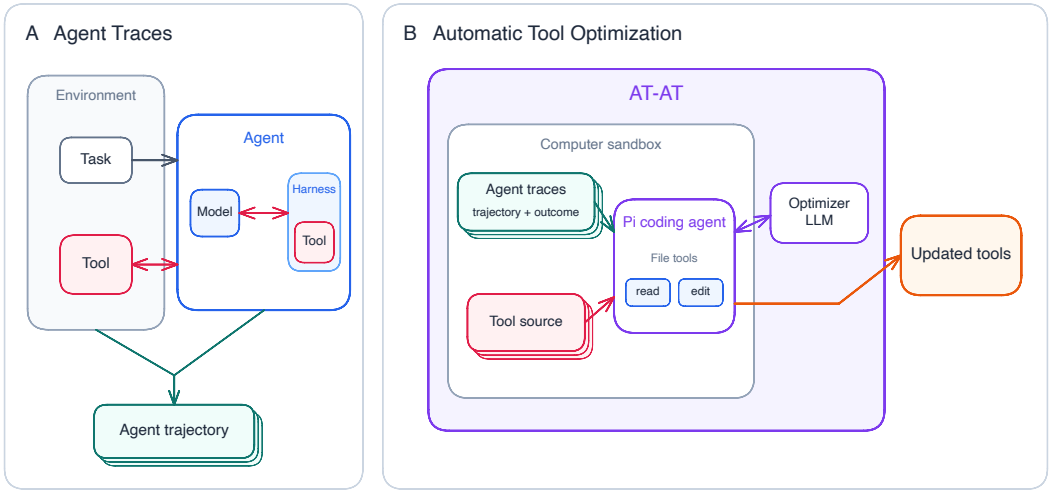


Figure 1: **AT-AT optimizes tools from historical agent traces.** (A) An agent completes a task by calling tools, implemented in its harness or the external environment, producing a trajectory. (B) In a sandbox holding evaluated traces and the current tool source, a Pi coding agent driven by an optimizer LLM selectively edits tool descriptions and implementations.

data usage and plan limit leaves the agent to work out whether the line has run out of data, while an implementation that also returns the remaining data gives the agent this information directly.

Anthropic reported a 40% decrease in task completion time after redesigning the tools available to an agent (Anthropic, 2025b), which suggests that tool design can materially affect agent performance. Existing automatic methods primarily optimize the information presented before a tool is invoked: DRAFT (Qu et al., 2025), Play2Prompt (Fang et al., 2025), EasyTool (Yuan et al., 2025), and ToolObserver (Hallinan et al., 2026) use exploration traces or reflection to refine tool descriptions or usage examples. Harness optimizers such as Meta-Harness (Lee et al., 2026b) can also edit tool code, but they optimize the harness of a single agent rather than a toolset shared by many agents. In this work, we aim to automatically improve both the descriptions and implementations of shared tools from evaluated agent traces.

We introduce AT-AT (Automatic Tool Optimization from Agent Traces), which converts historical agent experience into selective edits to a toolset. A lightweight coding agent searches evaluated agent traces for recurring failure modes, inspects the tool source, and edits tool descriptions and, when permitted, implementations. A coding agent can search many long traces as needed, and it can read the tool's code to decide whether a failure calls for a description or an implementation fix. AT-AT performs a single offline optimization pass using previously collected executions; it does not require new agent rollouts during optimization.

Trace-driven optimization raises two methodological challenges. First, repeated agent executions can produce materially different outcomes even when the task, model, harness, environment, and decoding configuration are held constant. We evaluate the original and optimized toolsets with 30 trials per held-out task and use paired permutation tests to determine whether observed changes exceed stochastic variation. Second, an optimizer can overfit to a small set of trajectories or tasks, which is a particularly consequential risk for shared tools. We expose AT-AT to traces spanning multiple tasks and repeated trials, direct it toward generalizable edits, and evaluate the resulting toolsets on held-out executions.

We evaluate AT-AT in three settings. On τ^2-bench-verified, AT-AT achieves the highest held-out task completion rate in Airline and Telecom at a lower cost per test simulation than prior prompt and tool optimizers, and its gain of 7.25 percentage points in Telecom is statistically significant. On TerminalBench-2 and TBLite, optimizing the tools within the OpenCode harness improves held-out pass@1 and pass@3, demonstrating applicability beyond customer service tools. On three production MCP servers at [Anonymized],[1] description edits learned from historical logs increased held-out

[1]Company identity anonymized only for review; identity will be shared and revealed in any published version.

task passes on two servers and reduced tool errors on the third. Together, these results show that the traces used to diagnose agent failures can also be converted into improvements to the environment itself. Because AT-AT can be rerun as traces accumulate, it provides a mechanism for continual improvement, although we evaluate a single update rather than a continual learning loop.

Our contributions are as follows:

- We formulate tool optimization for fixed agents from previously collected traces, where a tool's description guides the agent before a call and its implementation shapes the feedback the agent receives after the call (feedback shaping).
- We introduce AT-AT, a lightweight coding agent optimizer that translates evaluated historical trajectories into selective edits of tool descriptions and, when permitted, implementations, without new agent executions during optimization.
- We evaluate AT-AT on τ^2-bench-verified, TerminalBench-2, TBLite, and three production MCP servers, using repeated held-out trials and paired permutation tests on τ^2-bench-verified to separate tool improvements from run-to-run noise.

2 Related Work

Prompt and harness optimization. Prompt optimizers search over natural-language or structured prompt programs, using execution outcomes or reflective feedback to guide candidate generation (Schnabel & Neville, 2024; Agrawal et al., 2026). Coding agent optimizers expand this search space: FAPO can modify prompts and multi-step pipeline structure, while Meta-Harness, Self-Harness, and AutoSaddler optimize harness code using execution traces and scores (Kassianik et al., 2026; Lee et al., 2026b; Zhang et al., 2026; Park et al., 2026). Meta-Harness is especially similar in giving an agentic proposer access to an agent's source code, scores, and full traces via a filesystem, but it iteratively searches over the agent harness, which includes the harness's internal tools, rather than a toolset alone (Lee et al., 2026b). When tools are implemented inside the harness, as in our OpenCode experiments, AT-AT performs a restricted form of harness optimization that edits only the tools. Our primary setting instead updates a shared environmental toolset without changing the agents that consume it, since a tool provider typically does not own those agents.

Manual tool design. Anthropic recommends designing and testing tools for LLMs rather than exposing conventional developer-facing APIs and describes an evaluation-driven workflow in which developers inspect agent transcripts and use a coding agent to revise descriptions and implementations, checking generalization on held-out tasks (Anthropic, 2025b;a). AT-AT automates such a workflow.

Documentation and schema optimization. Most prior methods optimize the prompt-visible surface of a tool. EasyTool condenses heterogeneous documentation into standardized instructions, while PA-Tool renames schema components to align with a model's pretrained naming conventions (Yuan et al., 2025; Lee et al., 2026a). DRAFT and Play2Prompt interact with tools to refine documentation or synthesize usage examples without labeled tasks (Qu et al., 2025; Fang et al., 2025). A study of 856 tools from 103 MCP servers found at least one quality defect in 97.1% of tool descriptions, and augmenting the descriptions raised median task success while also increasing execution steps (Hasan et al., 2026). This still leaves executable tool behavior fixed.

Trace-driven tool optimization. Joint context optimization generates training-task rollouts and uses feedback on final answers and tool-call histories to jointly refine agent instructions and tool descriptions, while JTPRO uses rollout-driven reflection to co-optimize global agent instructions and per-tool schemas (Wu et al., 2025; Ghoshal et al., 2026). ToolObserver and DocsChisel repeatedly rerun the agent being optimized and use fixed LLM pipelines to revise only tool documentation from its trajectories. ToolObserver distills patterns through mini-batching and consensus, whereas DocsChisel uses field-level edits with validation and regression checks (Hallinan et al., 2026; Lu et al., 2026). AT-AT instead uses a coding-agent optimizer to update shared tools from previously evaluated trajectories without optimization-time agent reruns, and can make contract-preserving implementation edits that change execution-time results, errors, and recovery cues. ContDa uses self-generated probing traces to continually adapt documentation as tools are added, changed, or deprecated (Wu et al., 2026). Although AT-AT can be rerun as production logs accumulate, the present work evaluates one update to an existing toolset.

Environment optimization and tool synthesis. Other work also improves the environment rather than the agent. Song et al. (2025) derive a taxonomy of agent–environment failures from traces and show that hand-designed environment changes, such as better observability and offloading computation, improve fixed agents; AT-AT instead derives edits automatically from evaluated traces. Tool-synthesis methods expand the action space by creating new tools from API documentation (Ni et al., 2025), composing atomic actions into reusable procedures (Ding et al., 2026), or compiling procedures repeated in execution traces (Kujanpää et al., 2026). AT-AT instead improves tools that are already deployed while preserving the interfaces their callers depend on.

3 Optimization Method

3.1 Problem Formulation

Environmental tool optimization. We consider a population of agents $A \sim \mathcal{A}$. Each agent $A = (H, M)$ consists of a fixed harness H and a fixed large language model M. The harness includes components such as the system prompt, code that manages the agent loop and multiple turns, and internal tools such as file reading and file writing utilities. We also consider a population of tasks $x \sim \mathcal{X}$, which can be drawn from a variety of domains.

Given an agent A, a task x, and an environmental toolset T_E, executing the agent generates a trajectory τ consisting of the LLM outputs, user messages, tool invocations, and tool results that occur during execution. We denote the induced trajectory distribution by $p_A(\tau \mid x, T_E)$. Equivalently, for $A = (H, M)$, $p_A(\tau \mid x, T_E) = p_M(\tau \mid H, x, T_E)$. This distribution captures stochasticity from both the LLM and the environment. The reward $r(\tau, x)$ measures how successfully the trajectory accomplishes the task.

We optimize a shared environmental toolset $T_E = (T_{E_D}, T_{E_C})$, where T_{E_D} denotes the tool descriptions and T_{E_C} denotes the executable tool code. Environmental tools may include external functions, APIs, or MCP servers that are not implemented within the agent harness. Our objective is to select the toolset that maximizes expected performance across a population of agents and tasks:

$$T^*_{E_D}, T^*_{E_C} = \arg\max_{T_{E_D}, T_{E_C}} \mathbb{E}_{A\sim\mathcal{A},\, x\sim\mathcal{X},\, \tau\sim p_A(\cdot|x, T_{E_D}, T_{E_C})} \left[r(\tau, x) \right]. \tag{1}$$

Optimization of tools within a harness. Tools may also be implemented within a harness. In this setting, tool optimization is a restricted form of harness optimization: only the tool descriptions and implementations are editable, while all other harness components remain fixed. Let $T_H = (T_{H_D}, T_{H_C})$ denote the internal tools being optimized, and let H_{-T_H} denote the remaining harness components. The corresponding objective is

$$T^*_{H_D}, T^*_{H_C} = \arg\max_{T_{H_D}, T_{H_C}} \mathbb{E}_{x\sim\mathcal{X},\, \tau\sim p_M(\cdot|H_{-T_H}, x, T_{H_D}, T_{H_C})} \left[r(\tau, x) \right]. \tag{2}$$

We evaluate AT-AT in both tool optimization settings. Our τ^2-bench-verified experiments (Sec. 4.1) and production MCP server experiments (Sec. 4.3) optimize environmental tools as defined in Eq. 1. Our coding agent experiments (Sec. 4.2) instead optimize tools within the OpenCode harness.

3.2 AT-AT Method

Agent traces on a subset of tasks (either the train split on a benchmark, or historical tasks in a production setting) are constructed. Each agent trace consists of the agent trajectory on a certain task (agent outputs, user messages, tool invocations and results) along with the final evaluation result on the task. The agent traces and tools to edit are provided to the agentic tool optimizer as files in a file system. The optimizer is a lightweight, open-source coding agent, the Pi agent (Earendil Inc. and Contributors, 2026). The optimization objective is provided as a system prompt to the Pi agent. An instruction to the optimizer to make generalizable tool edits is also provided in the system prompt. The prompts to AT-AT are provided in Appendix F. The Pi agent makes selective edits to the tool files in the codebase based on what it sees in the agent traces. Once edits are finished, they must pass a validation check. The edited tool files must pass a syntax/compilation/interpreter check, and no files may be added, deleted, or edited outside the editable tool files. If validation fails, the Pi agent's

session is resumed with a prompt stating that validation failed, and the agent may revise its edits. The optimizer is given up to five attempts in total to produce a proposal that passes validation. The implementation of AT-AT is provided in the code in the supplement.

4 EXPERIMENTS & RESULTS

4.1 τ^2-BENCH-VERIFIED

τ^2-bench (Barres et al., 2026) is a popular benchmark for evaluating LLMs on customer service agent tasks. τ^2-bench-verified (Cuadron et al., 2025) is an updated version of τ^2-bench in which the tasks were manually reviewed and corrected. In our setup, the user agent and customer service agent LLMs were GPT-5.1 and GPT-4.1, respectively, which is the τ^2-bench-verified default. A train/test split was conducted using the default train and test task indices provided in the τ^2-bench-verified repository. The evaluators used were the default evaluators provided in τ^2-bench-verified. First, a baseline run of τ^2-bench-verified with the default settings was conducted, with $n = 5$ trials run on each task. The agent trajectories and evaluations were combined to form the baseline agent traces. The same baseline agent traces (train split only) were inputted into all tool optimization methods used, and the tool optimizers were run to generate updated toolsets. The LLM used for all tool optimizers was Claude Opus 4.7. τ^2-bench-verified was rerun using the updated toolsets. Additionally, the average cost of running a task in τ^2-bench-verified was reported, both before and after tool optimizers were applied. This cost was computed using the built-in cost tracker in τ^2-bench-verified. The cost of running the optimizer is also reported and was computed by multiplying the number of tokens used by the cost per token from the LLM APIs.

Paired permutation testing. Agent outcomes vary across repeated executions even when the model, harness, tasks, environment, and decoding configuration are held fixed. A difference between two aggregate benchmark scores therefore does not by itself establish that one toolset reliably outperforms another. To account for this evaluation noise, we evaluate the original and optimized toolsets on the same held-out tasks, with 30 trials per task, and compare their outcomes using a paired permutation test. Pairing outcomes within tasks accounts for differences in task difficulty, while repeated trials capture run-to-run variation in agent execution. Under the null hypothesis that the toolset has no effect, the original and optimized condition labels are exchangeable within each pair; permuting these labels produces the null distribution of the difference in task completion rate. We then test whether the observed improvement is larger than would be expected from run-to-run variation. At small sample sizes, exact computation of the paired permutation test p value is possible, as described in Zmigrod et al. (2022). However, at our larger sample size of $n = 30$, we estimate the p-value using Monte Carlo permutation sampling. Each value we compute is the comparison of the optimized toolset result compared to the original, unoptimized toolset.

Baseline optimizers. We compare to DRAFT (Qu et al., 2025), ToolObserver (Hallinan et al., 2026), and GEPA (Agrawal et al., 2026). DRAFT is an online method requiring interaction with live tools, while our τ^2-bench-verified experiment setup provides pre-collected agent traces. Hence, our implementation contained the offline portions of DRAFT, applying DRAFT's Analyzer and Rewriter stages once to the past agent traces. Similarly, we adapt ToolObserver to a fixed offline setting by applying its minibatch reflection and consensus merging once to pre-collected trajectories, omitting iterative exploration and trajectory recollection. We run GEPA (Agrawal et al., 2026) by treating the toolset as a prompt to be optimized. GEPA was used with 1 exploration/reflection step to provide a fair comparison to the other methods, which also used 1 exploration/reflection step.

Overall results. We report the average reward (pass@1) scores as percentages for each of the three domains: Airline, Retail, and Telecom, both for the baseline (unoptimized) toolset and optimized toolsets produced by AT-AT, our ablations of AT-AT, and the prior baseline optimizers. The results of these experiments can be viewed in Tab. 1. AT-AT has the highest reward score on the Airline and Telecom domains, and its improvement in Telecom is the only improvement that was statistically significant. In both domains, AT-AT also costs less per test simulation than ToolObserver, the next-best prior method. DRAFT resulted in a statistically significant decrease in performance on Telecom. We contextualize the improvements AT-AT was able to achieve by comparing them to submissions on the official τ^2-bench leaderboard (Sierra Research, 2026). The 7-point improvement AT-AT achieved

on Airline is comparable to the improvement from Claude-3.7-Sonnet (released Feb. 24, 2025; 64%) to Claude Sonnet 4.5 (released Sep. 29, 2025; 72%). The 7-point improvement AT-AT achieved on Telecom is comparable to the improvement from Claude-3.7-Sonnet (released Feb. 24, 2025; 49%) to GPT-5.2 (released Dec. 11, 2025; 57%). However, the tool edits AT-AT makes can be applied to a variety of agents and potentially improve performance of many agents (discussed in Sec. 4.1.1).

Costs. Two types of costs are reported: the cost to run a task in τ^2-bench-verified, and the cost to run the optimizer. Optimization increased the cost to run the benchmark by a statistically significant amount in many cases. We attribute this partially to the increased tool file length, leading to a higher number of tokens that the executor agent needs to process. In addition, variance in costs may be due to prompt cache hit rates. Optimizer costs were similar across optimizers, except for ToolObserver, which incurred substantially higher optimization costs. We attribute this to its minibatch-analysis and consensus-merge structure, which required many more LLM calls than the other optimizers.

4.1.1 ABLATIONS

Effect of feedback shaping: implementation edits vs description-only edits. To test our hypothesis that tools' feedback provided to an agent can meaningfully shape its trajectory and improve task success, we ran a version of AT-AT that was constrained to editing tool descriptions only and could not edit tool code. This ablation also omitted agent traces from the user prompt, though they were available in the filesystem. Results are shown in the "Edit descriptions only" rows of Tab. 1. The descriptions-only method performed substantially worse than AT-AT in all domains but Retail (but in the Retail domain, all optimized tools performed worse than unoptimized tools, with no difference in Retail being statistically significant). A further qualitative analysis comparing tool edits from this ablation is in App. B.

Importance of past agent traces. To test if historical agent traces are a useful learning signal for AT-AT, we ran AT-AT without any historical agent traces available to it. Results are shown in the "Without past traces" rows of Tab. 1. The method without past agent traces performed substantially worse than AT-AT in all domains but Retail (but in the Retail domain, all optimized tools performed worse than unoptimized tools). This evidence supports that past agent traces are a useful learning signal for a tool optimizer. In addition to enabling a tool optimizer to learn about tool failure patterns, historical agent traces may also enable an optimizer to learn about a new domain from data that is automatically gathered, rather than being curated or labeled by humans.

Importance of coding agent harness. To investigate the importance of the coding agent setup to the tool optimizer, we ran an experiment with a single-shot LLM-based tool optimizer. Since the LLM-based optimizer did not have the ability to search through a file system itself, the agent traces were only provided in their truncated form in the optimizer user prompt. The optimizer LLM was instructed to output a new tools.py file. Results are shown in the "Without coding agent harness" rows of Tab. 1. The LLM optimizer performed substantially worse than AT-AT in Airline and Telecom.

Generalization across agents. To investigate our hypothesis that edits to tools made by AT-AT can be useful across a variety of agents and tasks, we generated new baseline transcripts using a variety of customer service agent LLMs: GPT-4.1, GPT-4.1-mini, Inkling, Claude Sonnet 5, and GLM-5.2. AT-AT was run using the train split of these trials. The same 5 LLMs were used when running the test split before and after optimization. This ablation also omitted agent traces from the user prompt, though they were available in the file system. Results are shown in Tab. 2. AT-AT is able to improve reward in Airline and Retail, though the gain is not statistically significant. This provides evidence for the hypothesis that AT-AT can improve an environment's overall quality and help a variety of agents.

4.2 TERMINALBENCH-2 & TBLITE

To assess whether AT-AT can generalize to a different domain than the customer service agent tasks assessed in τ^2-bench-verified, we ran AT-AT on TerminalBench-2 and TBLite. TerminalBench-2 (Merrill et al., 2026) is a popular computer use and agentic coding benchmark with long-horizon, multi-turn tasks. TBLite (OpenThoughts-Agent team, Snorkel AI, and Bespoke Labs, 2026) is a lightweight alternative to TerminalBench-2, designed to track results on TerminalBench-2 closely while enabling a faster iteration speed and runtime. The toolset optimized was that within OpenCode

Table 1: **AT-AT achieves the highest test reward in Airline and Telecom.** Held-out results over $n = 30$ trials per task. Test cost is agent plus user-simulator cost per simulation; optimizer cost is the one-time cost of producing the toolset. p: two-sided task-paired permutation test against the domain baseline; $*$ marks $p < 0.05$. Bold: best value per domain (optimizer cost excludes baseline).

	Optimizer	Test reward (%)	Reward p	Test cost (USD/sim.)	Cost p	Optimizer cost (USD)
Airline	Baseline	52.17%	—	**0.0618**	—	—
	AT-AT	**59.17%**	0.2305	0.0675*	0.002155	2.7800
	Without past traces	50.33%	0.3575	0.1205*	0.000002	**0.2330**
	Edit descriptions only	51.33%	0.7609	0.1176*	0.000002	0.3839
	Without coding agent harness	52.50%	0.9446	0.0825*	0.000002	2.2481
	DRAFT	53.50%	0.5988	0.0822*	0.000002	2.4367
	ToolObserver	55.83%	0.1650	0.0870*	0.000002	16.2171
	GEPA	51.50%	0.7539	0.0748*	0.000008	0.8042
Retail	Baseline	**77.00%**	—	**0.0707**	—	—
	AT-AT	74.67%	0.2313	0.0724	0.070629	1.9300
	Without past traces	75.42%	0.3129	0.1383*	0.000010	**0.2174**
	Edit descriptions only	74.08%	0.0992	0.1453*	0.000010	0.5613
	Without coding agent harness	74.92%	0.1979	0.0802*	0.000010	3.8496
	DRAFT	75.50%	0.5296	0.0871*	0.000010	3.9785
	ToolObserver	75.92%	0.6196	0.0926*	0.000010	42.1400
	GEPA	76.08%	0.5731	0.0798*	0.000010	0.8765
Telecom	Baseline	29.50%	—	0.0996	—	—
	AT-AT	**36.75%***	0.0033	**0.0986**	0.508165	10.6900
	Without past traces	30.42%	0.6281	0.2850*	0.000010	**0.2888**
	Edit descriptions only	31.17%	0.4547	0.2857*	0.000010	1.1342
	Without coding agent harness	30.50%	0.6172	0.1146*	0.000010	4.4205
	DRAFT	25.58%*	0.0026	0.1259*	0.000010	4.5630
	ToolObserver	31.67%	0.4686	0.1276*	0.000010	49.1224
	GEPA	29.25%	0.9102	0.1132*	0.000010	0.8846

(Anomaly, 2026), an open-source coding agent harness. OpenCode contains tools with implementations and descriptions written in TypeScript code, and these tools were the target of the tool optimizer. OpenCode was run on TerminalBench-2 and TBLite, and the resulting agent traces were split into a train and test set. The train set was then provided to AT-AT to optimize the tools within the OpenCode repository. In these experiments, past traces were omitted from the optimizer user prompt, although they were available in the file system. Finally, the tool-optimized version of OpenCode was rerun on each benchmark to calculate final performance on the train and test splits. The LLM used by AT-AT was Claude Opus 4.7, while the LLM used by the OpenCode agent was Claude Opus 4.5. We report pass@k as computed by the benchmarks' native evaluators, before and after tool optimization. Results on TerminalBench-2 and TBLite can be viewed in Tab. 3. On TerminalBench-2, AT-AT improves held-out pass@1, pass@3, and pass@10, while on TBLite, AT-AT improves held-out pass@1 and pass@3, while held-out pass@10 is unchanged.

4.3 Production MCP Servers

MCP and task selection. We applied AT-AT to production MCP servers at [Anonymized]. Three production MCP servers at the company were chosen for optimization due to (1) having a high usage rate, (2) having a sufficient number of unique single-user prompts with actionable failures, and (3) the ability within the company infrastructure to expose and modify the tool descriptions in a sandbox environment. The MCP servers used were the following: *Core Developer Tools* (Dev Tools), which provides broad engineering retrieval across internal documentation, collaboration platforms, and enterprise knowledge sources; *Observability Collaboration Toolbox* (Collab Toolbox), which searches, summarizes, and exports channel and thread content from an enterprise collaboration platform; and *Enterprise Engineering Context and Best Practices* (Eng. Context), which provides

Table 2: **AT-AT's tool edits improve held-out reward in 2 of 3 τ^2-bench-verified domains when averaged across five customer service agent LLMs.** Results are averaged across models within each task, then across tasks, with 30 trials per task. Columns and $*$ are as in Tab. 1.

Domain	Optimizer	Test reward (%)	Reward p	Test cost (USD/sim.)	Cost p	Optimizer cost (USD)
Airline	Baseline	58.00%	—	**0.1135**	—	—
	AT-AT	**59.67%**	0.3584	0.1239*	0.000557	2.4029
Retail	Baseline	73.83%	—	**0.1221**	—	—
	AT-AT	**76.42%**	0.1662	0.1315*	0.000010	2.3330
Telecom	Baseline	**41.42%**	—	**0.2457**	—	—
	AT-AT	40.67%	0.7320	0.2602*	0.000100	1.8308

Table 3: **AT-AT improves held-out pass@k on TerminalBench-2 and TBLite.** Pass@k from each benchmark's native evaluator before (Baseline) and after (Optimized) optimizing OpenCode's tools, on the train split used for optimization and the held-out test split.

	Train				Test			
	TerminalBench-2		TBLite		TerminalBench-2		TBLite	
k	Baseline	Optimized	Baseline	Optimized	Baseline	Optimized	Baseline	Optimized
1	45.78%	48.44%	72.80%	72.80%	49.32%	**51.36%**	75.60%	**77.20%**
3	58.43%	58.69%	81.38%	81.95%	59.43%	**61.34%**	88.65%	**89.63%**
10	68.89%	68.89%	84.00%	86.00%	68.18%	**70.45%**	94.00%	94.00%

Table 4: **AT-AT improves task pass rate on two production MCP servers and reduces tool errors on the third.** Held-out test results with the original (Orig.) and AT-AT-optimized tool descriptions; n is the number of scored test tasks. Mean reward and content score are shown as percentages. Session cost covers the agent and user simulator; optimizer cost is one-time.

	Dev Tools (n=10)		Collab Toolbox (n=5)		Eng. Context (n=11)	
Metric	Orig.	AT-AT	Orig.	AT-AT	Orig.	AT-AT
Task pass rate (%)	60.0	**80.0**	0.0	0.0	54.5	**72.7**
Mean reward (%)	78.1	**90.0**	22.3	**35.1**	77.3	**86.4**
Content score (%)	79.8	80.3	37.7	36.7	64.7	**69.0**
Tool calls	**132**	143	121	**86**	114	**76**
Tool errors	11	11	73	**37**	0	0
Session cost	**$0.837**	$0.926	$0.442	**$0.276**	$0.288	**$0.235**
Optimizer cost	–	$7.874	–	$6.935	–	$7.684

enterprise-specific engineering guidance from indexed documentation and code. Tasks were defined as an initial agent prompt provided by a user, and tasks were randomly selected from the prior two weeks of production traffic that each of these MCP servers received. A summary of representative tasks is contained in Tab. 12.

Experiment procedure. The list of tasks was randomly partitioned into train/test splits of 10/10, 5/5, and 11/11 tasks for Core Developer Tools, Observability Collaboration Toolbox, and Enterprise Engineering Context and Best Practices, respectively. Across both splits and in the baseline and optimized runs, system prompt, four-user-turn limit, agent model, user simulator, and all other environment settings were held fixed. The agent executing the task used the open-weight DeepSeek V4 Flash model, and the user simulator used Claude Sonnet 4.6. For each task, a baseline run with the original MCP servers was executed, with the task being the initial prompt provided to the executor agent. The baseline run was evaluated as described below. Traces of agent trajectories and evaluation results from the train split of tasks were constructed and provided to AT-AT, which used Claude Opus 4.7 as its optimizer LLM. Past traces were omitted from the optimizer user prompt, although they were available in the file system. After tool optimization, evaluation was repeated on the train and test splits using the same surrounding agent setup, but with the optimized tool descriptions.

Evaluation. Three evaluation metrics were computed for each task. Firstly, a binary pass/fail value for a task was computed via an LLM judge. Secondly, a reward score was computed for each task.

$$r(\tau, x) = \begin{cases} 1, & \text{if the task-success criteria were met,} \\ \frac{1}{2}\frac{n_{\text{accepted}}}{\max(1, n_{\text{eligible}})}, & \text{otherwise,} \end{cases}$$

Here, n_{eligible} is the number of tool calls remaining after harness-blocked and environmental calls are excluded, and n_{accepted} is the subset of eligible calls whose result was not an error. This framing provided some partial credit to traces that had fewer tool call failures, while capping unsuccessful traces at a reward of 0.5. Thirdly, an LLM judge-based content score was reported to measure answer quality separately from tool execution. An LLM converted each task's success criteria into a fixed rubric: a checklist of 3–7 independently checkable clauses. For K clauses, the content score was

$$s_{\text{content}} = \frac{1}{K}\sum_{j=1}^{K} q_j, \qquad q_j \in \{1, 0.5, 0\}$$

for clause verdicts met, partially met, and not met, respectively. All evaluation generation and judging used GPT-5.6 Luna. In addition, the total number of tool calls, the number of failed tool calls, the mean cost of agent sessions in each domain, and the cost to run AT-AT were reported. Session cost includes the agent and user simulator but excludes evaluation costs, while optimizer cost is a one-time cost that includes all optimization attempts. Task pairs with a session error either before or after optimization were excluded from efficacy, tool-use, and session-cost metrics. One task in the Enterprise Engineering Context and Best Practices server had a session error during evaluation and was omitted.

Results. Results are shown in Tab. 4. On Dev Tools, AT-AT improved task completion and mean reward, while tool calls and execution cost increased and tool errors were unchanged. Gains were concentrated in the documentation-focused task set, while the mixed retrieval set did not improve, indicating that the effect varied by task composition (Tab. 14, App. D). On Collab Toolbox, AT-AT reduced tool calls, tool errors, and execution cost. Mean task reward increased because a larger fraction of eligible tool calls succeeded, but neither arm completed a held-out task and mean content score declined slightly. On Eng. Context, AT-AT improved task completion and content score while reducing tool calls and execution cost. Thus, the three servers exhibit different tradeoffs: task completion gains with higher cost for Dev Tools, tool reliability and efficiency gains without task completion for Collab Toolbox, and simultaneous held-out efficacy and efficiency gains for Eng. Context. App. D gives paraphrased examples of the resulting tool-description changes and one task's evaluation specification.

5 Conclusion

AT-AT optimizes agent tools as a way to improve agents' performance, holding agents fixed and leveraging logged agent traces. This led to performance improvement in multiple benchmarks, notably including a production MCP server at a large enterprise. Because AT-AT learns from logs that deployed systems already produce, it can be rerun as traces accumulate, storing experience in the environment rather than in the agent.

The appendices explore mechanisms that may be at play in greater depth. Taking a closer look at selected transcripts can give a qualitative understanding of feedback shaping, as in App. B. Additionally, we discuss evaluation stochasticity and how to handle it in App. C.

Our results suggest several possible avenues for future work. First, a more explicit model of feedback shaping would be of interest: how changing what a tool returns steers an agent's trajectory. Second, attributing failures to tools, agents, or tasks would bound the headroom for any environment-side intervention and let an optimizer focus on where tools are at fault. Third, a shared tool serves a heterogeneous and evolving population of agents, so tool optimization is fundamentally a problem of generalization across agents and of co-adaptation with agent- and harness-side optimization (App. E), whose interaction is not yet understood. Finally, treating the environment as a site of continual learning calls for guarantees against regression and drift as edits accumulate, and raises the possibility of using tool design to steer agents away from reward hacking and unsafe actions.

AI USE STATEMENT

In this work, we used generative AI tools for the following tasks:

- Required disclosure tasks
  - Implement methods: GenAI assistance was used to write code needed to implement the various tool optimizer methods, as well as the surrounding infrastructure to run experiments.
  - Support qualitative and thematic data analysis: LLMs were used to analyze and summarize large amounts of data to identify relevant examples and common themes within sections with qualitative analysis.
- Recommended disclosure tasks
  - Assist with polishing writing in paper: Some parts of the paper were first written by hand and then run through an LLM to aid in polishing the writing's clarity.
  - Draft parts of the research paper: While most of the research paper was initially written by hand and then in some cases run through an LLM for polishing, in some cases, an LLM was used to generate the first draft of small sections of the writing, which was then reviewed and edited as needed. An LLM was also used to help draft the qualitative analyses in the appendices.
  - Literature review: LLM tools like Deep Research were used to identify related prior works and summarize their findings.

We did not use generative AI tools for the following tasks:

- Required disclosure tasks
  - Generate synthetic datasets
  - Help develop theoretical models or conceptual frameworks
  - Formulate mathematical claims
  - Propose or refine hypotheses
  - Design or provide feedback on research methodology or experiments
- Recommended disclosure tasks
  - Suggest a structure for a research paper
  - Propose a title or keywords for a research paper

The rest of the required/recommended disclosure tasks were not relevant to our work.

We have reviewed all AI-assisted work. All text in the paper that was polished by AI or generated by AI was carefully reviewed and edited for correctness, clarity, and style. All literature sources identified with AI were reviewed manually to assess relevance to the current work. In addition, LLM-generated code was verified and tested for correctness. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

No potential issues regarding adherence to the ICLR Code of Ethics were identified.

REPRODUCIBILITY STATEMENT

The source code for the methods described in the paper, the baseline methods, the experiment driver, and the statistical analysis is included in the repository uploaded in the supplement. The benchmarks τ^2-bench-verified, TerminalBench-2, and TBLite are publicly available. Many repeated trials, ablations, and statistical analyses were conducted to assess the robustness of the results. The data needed to run the method on the production MCP servers is kept private, but the experiment procedure is described and a description of the MCP servers, tasks, evaluations, and optimizer edits is provided.

REFERENCES

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *ICLR*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Anomaly. OpenCode: The open source AI coding agent, 2026. URL https://github.com/anomalyco/opencode. GitHub repository. Accessed August 26, 2026.

Anthropic. Writing effective tools for agents—with agents, September 2025a. URL https://www.anthropic.com/engineering/writing-tools-for-agents. Engineering at Anthropic, September 11, 2025.

Anthropic. Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet, January 2025b. URL https://www.anthropic.com/engineering/swe-bench-sonnet. Engineering at Anthropic, January 6, 2025.

Anthropic. Quantifying infrastructure noise in agentic coding evals, February 2026. URL https://www.anthropic.com/engineering/infrastructure-noise. Engineering at Anthropic, February 5, 2026.

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.

Alejandro Cuadron, Pengfei Yu, Yang Liu, and Arpit Gupta. SABER: Small actions, big errors – safeguarding mutating steps in LLM agents, 2025. URL https://arxiv.org/abs/2512.07850.

Haipeng Ding, Yuexiang Xie, Zhewei Wei, Yaliang Li, and Bolin Ding. From atomic actions to standard operating procedures: Iterative tool optimization for self-evolving LLM agents, 2026. URL https://arxiv.org/abs/2607.07321.

Earendil Inc. and Contributors. Pi agent harness, 2026. URL https://github.com/earendil-works/pi. GitHub repository. Accessed August 27, 2026.

Wei Fang, Yang Zhang, Kaizhi Qian, James R. Glass, and Yada Zhu. PLAY2PROMPT: Zero-shot tool instruction optimization for LLM agents via tool play. In *ACL Findings*, pp. 26274–26290, 2025. doi: 10.18653/v1/2025.findings-acl.1347. URL https://aclanthology.org/2025.findings-acl.1347/.

Sandip Ghoshal, Anshul Mittal, Jyotika Singh, Miguel Ballesteros, Weiyi Sun, Fang Tu, Shailender Singh, Yassine Benajiba, Fahad Shah, Sujeeth Bharadwaj, Sujith Ravi, and Dan Roth. JTPRO: A joint tool–prompt reflective optimization framework for language agents. In *ACL Findings*, pp. 40573–40595, 2026. doi: 10.18653/v1/2026.findings-acl.2017. URL https://aclanthology.org/2026.findings-acl.2017/.

Skyler Hallinan, Thejas Venkatesh, Xiang Ren, Sai Praneeth Karimireddy, Ashwin Paranjape, Yuhao Zhang, and Jack Hessel. OpaqueToolsBench: Learning nuances of tool behavior through interaction. In *CAIS*, 2026. doi: 10.1145/3786335.3813182. URL https://doi.org/10.1145/3786335.3813182.

Mohammed Mehedi Hasan, Hao Li, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E. Hassan. Model Context Protocol (MCP) tool descriptions are smelly! towards improving AI agent efficiency with augmented MCP tool descriptions, 2026. URL https://arxiv.org/abs/2602.14878.

Paul Kassianik, Baturay Saglam, Huaibo Zhao, Blaine Nelson, Supriti Vijay, Aman Priyanshu, and Amin Karbasi. FAPO: Fully automated prompt optimization of multi-step LLM pipelines, 2026. URL https://arxiv.org/abs/2606.19605.

Kalle Kujanpää, Ning Liu, Shahnawaz Alam, Yeshwanth Reddy Sura, Tianyu Yang, Kristina Klinkner, and Shervin Malmasi. Tool-making and self-evolving LLM agents in low-latency systems, 2026. URL https://arxiv.org/abs/2607.08010. To appear in the EMNLP 2026 Industry Track.

Jonggeun Lee, Woojung Song, Jongwook Han, Haesung Pyun, and Yohan Jo. Don't adapt small language models for tools; adapt tool schemas to the models. In *ACL*, pp. 20695–20719, 2026a. doi: 10.18653/v1/2026.acl-long.948. URL https://aclanthology.org/2026.acl-long.948/.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-Harness: End-to-end optimization of model harnesses. In *COLM*, 2026b. URL https://arxiv.org/abs/2603.28052.

You Lu, Kun Zhang, Bihuan Chen, and Xin Peng. DocsChisel: Adaptive tool documentation optimization framework for LLM agents, 2026. URL https://arxiv.org/abs/2608.10037.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *ICLR*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

Microsoft. Expose REST API in API Management as an MCP server, April 2026. URL https://learn.microsoft.com/en-us/azure/api-management/export-rest-mcp-server. Microsoft Learn. Last updated April 30, 2026. Accessed August 28, 2026.

Zairah Mustahsan, Abel Lim, Megna Anand, Saahil Jain, and Bryan McCann. Stochasticity in agentic evaluations: Quantifying inconsistency with intraclass correlation, 2025. URL https://arxiv.org/abs/2512.06710.

Xinyi Ni, Qiuyang Wang, Yukun Zhang, and Pengyu Hong. ToolFactory: Automating tool generation by leveraging LLM to understand REST API documentations, 2025. URL https://arxiv.org/abs/2501.16945.

OpenThoughts-Agent team, Snorkel AI, and Bespoke Labs. OpenThoughts-TBLite: A high-signal benchmark for iterating on terminal agents, February 2026. URL https://www.openthoughts.ai/blog/openthoughts-tblite. OpenThoughts blog, February 18, 2026.

Sungho Park, Wonjoong Kim, Rongyuan Tan, Jue Zhang, Wook-Shin Han, Pengfei Gao, Chanyoung Park, Yongqiang Yao, Rao Fu, Elsie Nallipogu, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. AutoSaddler: Automatic harness optimization with durable updates from agent execution traces, 2026. URL https://arxiv.org/abs/2608.23041.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. From exploration to mastery: Enabling LLMs to master tools via self-driven interactions. In *ICLR*, 2025. URL https://openreview.net/forum?id=QKBu1BOAwd.

Tobias Schnabel and Jennifer Neville. Symbolic prompt program search: A structure-aware approach to efficient compile-time prompt optimization. In *EMNLP Findings*, pp. 670–686, 2024. doi: 10.18653/v1/2024.findings-emnlp.37. URL https://aclanthology.org/2024.findings-emnlp.37/.

Sierra Research. τ^2-bench Leaderboard. https://taubench.com/leaderboard/?benchmark=core, 2026. Accessed: 2026-09-25.

Kevin Song, Anand Jayarajan, Yaoyao Ding, Qidong Su, Zhanda Zhu, Sihang Liu, and Gennady Pekhimenko. Aegis: Taxonomy and optimizations for overcoming agent-environment failures in LLM agents, 2025. URL https://arxiv.org/abs/2508.19504.

Sparsh Wadhwa, Dhawalkumar Patel, and Heeki Park. Streamline AI agent tool interactions: Connect API Gateway to AgentCore Gateway with MCP, December 2025. URL https://aws.amazon.com/blogs/machine-learning/streamline-ai-agent-tool-interactions-connect-api-gateway-to-agentcore-gateway-with-mcp/. AWS Artificial Intelligence Blog, December 8, 2025. Accessed August 28, 2026.

Lilian Weng. Harness engineering for self-improvement, July 2026. URL https://lilianweng.github.io/posts/2026-07-04-harness/. Lil'Log, July 4, 2026.

Bin Wu, Edgar Meij, and Emine Yilmaz. A joint optimization framework for enhancing efficiency of tool utilization in LLM agents. In *ACL Findings*, pp. 22361–22373, 2025. doi: 10.18653/v1/2025.findings-acl.1149. URL https://aclanthology.org/2025.findings-acl.1149/.

Bin Wu, Edgar Meij, and Emine Yilmaz. Beyond static toolsets: Self-evolving LLM tool agents via continual documentation adaptation. In *ACL Findings*, pp. 21519–21539, 2026. doi: 10.18653/v1/2026.findings-acl.1082. URL https://aclanthology.org/2026.findings-acl.1082/.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. EASYTOOL: Enhancing LLM-based agents with concise tool instruction. In *NAACL*, pp. 951–972, 2025. URL https://aclanthology.org/2025.naacl-long.44/.

Hangfan Zhang, Shao Zhang, Kangcong Li, Chen Zhang, Yang Chen, Yiqun Zhang, Lei Bai, and Shuyue Hu. Self-Harness: Harnesses that improve themselves, 2026. URL https://arxiv.org/abs/2606.09498.

Ran Zmigrod, Tim Vieira, and Ryan Cotterell. Exact paired-permutation testing for structured test statistics. In *NAACL*, pp. 4894–4902, 2022. doi: 10.18653/v1/2022.naacl-main.360. URL https://aclanthology.org/2022.naacl-main.360/.

A Detailed τ^2-bench-verified Results

A.1 Additional Ablations

Specificity of method to optimizer LLM used. To assess AT-AT's efficacy with different optimizer LLMs, AT-AT, and ToolObserver for comparison, were run with GPT-5.6 Sol and the open-weight model Kimi K3 with the same τ^2-bench-verified experiment setup. This ablation also omitted agent traces from the user prompt, though they were available in the file system. Results are shown in Tab. 5. AT-AT was able to achieve a statistically significant improvement in Telecom using GPT-5.6 Sol and Kimi K3, although regressions were seen when AT-AT was run with these two models on Airline, though the regressions were not statistically significant. This experiment supports that gains from AT-AT are possible with other models besides Claude Opus 4.7 and that the method is not overfit to a particular optimizer LLM.

Importance of generalization prompt and full traces in file system. To test the contribution of these two additions, we ran the Pi optimizer with only the trace digest in its first user message, without the generalization instruction and without the full per-task traces in its file system. Results are shown in Tab. 6. This variant underperforms AT-AT and does not achieve its gains on Airline and Telecom. Because both components were removed together, this ablation does not separate their individual contributions.

A.2 Detailed τ^2-bench-verified results

Tab. 7 reports the reward and recurring-cost deltas, 95% task-cluster bootstrap confidence intervals, and raw paired-permutation p-values corresponding to the main-paper summary in Tab. 1.

Table 5: **AT-AT's improvements are robust to the choice of optimizer LLM in τ^2-bench-verified's Telecom, where both GPT-5.6 Sol and Kimi K3 produce statistically significant gains.** The customer service agent and user simulator are GPT-4.1 and GPT-5.1, respectively; only the optimizer LLM varies. Results are computed over $n = 30$ trials per held-out test task. Test cost is mean agent plus user-simulator inference cost in USD per simulation, and optimizer cost is the one-time cost of producing the candidate toolset. $*$ marks a statistically significant change from the domain baseline at $p < 0.05$ under a two-sided task-paired permutation test. Best reward (highest), test cost (lowest), and optimizer cost (lowest non-baseline value) in each domain are bold.

Domain	Optimizer	Optimizer LLM	Test reward (%)	Reward p	Test cost (USD/sim.)	Cost p	Optimizer cost (USD)
Airline	Baseline	—	52.17%	—	**0.0618**	—	—
	AT-AT	Opus 4.7	**59.17%**	0.2305	0.0675*	0.002155	2.7800
		GPT-5.6 Sol	47.33%	0.3743	0.1194*	0.000002	**1.2866**
		Kimi K3	54.33%	0.3135	0.1246*	0.000002	4.7419
	ToolObserver	Opus 4.7	55.83%	0.1650	0.0870*	0.000002	16.2171
		GPT-5.6 Sol	52.83%	0.7461	0.1498*	0.000002	10.1014
		Kimi K3	53.33%	0.7576	0.1619*	0.000002	8.5355
Retail	Baseline	—	77.00%	—	**0.0707**	—	—
	AT-AT	Opus 4.7	74.67%	0.2313	0.0724	0.070629	1.9300
		GPT-5.6 Sol	74.75%	0.2487	0.1444*	0.000010	**0.8445**
		Kimi K3	75.92%	0.5848	0.1395*	0.000010	1.4375
	ToolObserver	Opus 4.7	75.92%	0.6196	0.0926*	0.000010	42.1400
		GPT-5.6 Sol	75.83%	0.6272	0.1490*	0.000010	27.5875
		Kimi K3	**77.50%**	0.8646	0.1774*	0.000010	23.4634
Telecom	Baseline	—	29.50%	—	0.0996	—	—
	AT-AT	Opus 4.7	36.75%*	0.0033	**0.0986**	0.508165	10.6900
		GPT-5.6 Sol	**47.58%***	0.0000	0.2633*	0.000010	**1.1033**
		Kimi K3	38.42%*	0.0056	0.2854*	0.000010	1.4343
	ToolObserver	Opus 4.7	31.67%	0.4686	0.1276*	0.000010	49.1224
		GPT-5.6 Sol	40.50%*	0.0001	0.3083*	0.000010	32.8944
		Kimi K3	37.67%*	0.0182	0.3219*	0.000010	25.2612

Table 6: **Pi optimizer without the generalization prompt or full file system traces underperforms AT-AT's reward gains in Airline and Telecom.** Held-out results over $n = 30$ trials per task; baseline and AT-AT rows are repeated from Tab. 1 for comparison. Test cost is agent plus user-simulator cost per simulation; optimizer cost is the one-time cost of producing the toolset. p: two-sided task-paired permutation test against the domain baseline; $*$ marks $p < 0.05$. Bold: best test reward and test cost per domain.

Domain	Optimizer	Test reward (%)	Reward p	Test cost (USD/sim.)	Cost p	Optimizer cost (USD)
Airline	Baseline	52.17%	—	0.0618	—	—
	AT-AT	**59.17%**	0.2305	0.0675*	0.002155	2.7800
	Without generalization prompt or full traces	51.33%	0.8099	**0.0605**	0.363811	2.7800
Retail	Baseline	**77.00%**	—	**0.0707**	—	—
	AT-AT	74.67%	0.2313	0.0724	0.070629	1.9300
	Without generalization prompt or full traces	75.75%	0.3186	0.0710	0.683283	1.9300
Telecom	Baseline	29.50%	—	0.0996	—	—
	AT-AT	**36.75%***	0.0033	**0.0986**	0.508165	10.6900
	Without generalization prompt or full traces	26.92%	0.1373	0.0993	0.750562	10.6900

Table 7: **Detailed τ^2-bench-verified reward and cost results.** Test reward and recurring inference cost are evaluated over 30 trials per held-out task. Each domain's baseline is reported as a separate row. Deltas are optimized minus baseline and are shown with 95% task-cluster bootstrap confidence intervals. Reported p-values are from two-sided task-paired permutation tests; $*$ marks $p < 0.05$. Recurring cost is mean agent plus user-simulator cost in USD per test simulation. Optimizer cost is the one-time cost of producing the candidate toolset.

Domain	Optimizer	Test reward (%)	Reward delta [95% CI]	Reward p	Test cost (USD/sim.)	Cost delta [95% CI]	Cost p	Optimizer cost
Airline	Baseline	52.17%	—	—	**0.0618**	—	—	—
	AT-AT	**59.17%**	+0.0700 [-0.0117, +0.1750]	0.2305	0.0675*	+0.0057 [+0.0025, +0.0095]	0.002155	2.7800
	Without past traces	50.33%	-0.0183 [-0.0533, +0.0133]	0.3575	0.1205*	+0.0587 [+0.0497, +0.0709]	0.000002	**0.2330**
	Edit descriptions only	51.33%	-0.0083 [-0.0483, +0.0350]	0.7609	0.1176*	+0.0558 [+0.0471, +0.0655]	0.000002	0.3839
	Without coding agent harness	52.50%	+0.0033 [-0.0417, +0.0500]	0.9446	0.0825*	+0.0207 [+0.0161, +0.0258]	0.000002	2.2481
	DRAFT	53.50%	+0.0133 [-0.0250, +0.0567]	0.5988	0.0822*	+0.0204 [+0.0150, +0.0269]	0.000002	2.4367
	ToolObserver	55.83%	+0.0367 [-0.0067, +0.0850]	0.1650	0.0870*	+0.0252 [+0.0162, +0.0360]	0.000002	16.2171
	GEPA	51.50%	-0.0067 [-0.0317, +0.0233]	0.7539	0.0748*	+0.0130 [+0.0079, +0.0202]	0.000008	0.8042
Retail	Baseline	**77.00%**	—	—	**0.0707**	—	—	—
	AT-AT	74.67%	-0.0233 [-0.0600, +0.0092]	0.2313	0.0724	+0.0017 [-0.0001, +0.0034]	0.070629	1.9300
	Without past traces	75.42%	-0.0158 [-0.0433, +0.0133]	0.3129	0.1383*	+0.0675 [+0.0610, +0.0746]	0.000010	**0.2174**
	Edit descriptions only	74.08%	-0.0292 [-0.0617, +0.0033]	0.0992	0.1453*	+0.0745 [+0.0668, +0.0832]	0.000010	0.5613
	Without coding agent harness	74.92%	-0.0208 [-0.0517, +0.0067]	0.1979	0.0802*	+0.0094 [+0.0074, +0.0115]	0.000010	3.8496
	DRAFT	75.50%	-0.0150 [-0.0567, +0.0292]	0.5296	0.0871*	+0.0163 [+0.0132, +0.0195]	0.000010	3.9785
	ToolObserver	75.92%	-0.0108 [-0.0500, +0.0275]	0.6196	0.0926*	+0.0219 [+0.0179, +0.0259]	0.000010	42.1400
	GEPA	76.08%	-0.0092 [-0.0375, +0.0183]	0.5731	0.0798*	+0.0091 [+0.0072, +0.0111]	0.000010	0.8765
Telecom	Baseline	29.50%	—	—	0.0996	—	—	—
	AT-AT	**36.75%***	+0.0725 [+0.0292, +0.1242]	0.0033	**0.0986**	-0.0010 [-0.0040, +0.0017]	0.508165	10.6900
	Without past traces	30.42%	+0.0092 [-0.0225, +0.0425]	0.6281	0.2850*	+0.1854 [+0.1772, +0.1932]	0.000010	**0.2888**
	Edit descriptions only	31.17%	+0.0167 [-0.0217, +0.0592]	0.4547	0.2857*	+0.1861 [+0.1755, +0.1959]	0.000010	1.1342
	Without coding agent harness	30.50%	+0.0100 [-0.0250, +0.0467]	0.6172	0.1146*	+0.0150 [+0.0125, +0.0175]	0.000010	4.4205
	DRAFT	25.58%*	-0.0392 [-0.0642, -0.0167]	0.0026	0.1259*	+0.0263 [+0.0238, +0.0287]	0.000010	4.5630
	ToolObserver	31.67%	+0.0217 [-0.0283, +0.0792]	0.4686	0.1276*	+0.0280 [+0.0249, +0.0311]	0.000010	49.1224
	GEPA	29.25%	-0.0025 [-0.0308, +0.0267]	0.9102	0.1132*	+0.0136 [+0.0117, +0.0154]	0.000010	0.8846

B FEEDBACK SHAPING: A QUALITATIVE ANALYSIS ON IMPLEMENTATION VS. DESCRIPTION-ONLY OPTIMIZER EDITS

This section contains a qualitative analysis on the ablation reported in Tab. 1, comparing the main AT-AT condition, which may edit the full tool file, with a variant of AT-AT restricted to editing existing tool docstrings. We combine a source-level audit of the tool edits each condition produced with matched trajectory case studies. Each condition generated one independent, stochastic candidate per domain, so this comparison is informative but is not a controlled estimate of the isolated effect of code-edit permission. We treat the case studies as mechanistic evidence rather than a formal mediation analysis.

B.1 TOOL EDITS

Tab. 8 summarizes the tools edited by each condition. The labels *code*, *description*, and *code + description* indicate which part of each agent-exposed tool changed; changes to private helpers are attributed to every tool whose behavior they affect. The descriptions-only candidates changed 2, 5, and 4 tool docstrings in Airline, Retail, and Telecom, respectively. A structural validator confirmed that their non-docstring bytes and executable abstract syntax trees were identical to the canonical tools. Full-scope AT-AT, by contrast, changed executable behavior in every domain. These interventions affect more than the prose shown to the agent: they change which tools are available, whether calls succeed, the environment state produced by write operations, and the observations returned in subsequent tool messages. Full-scope AT-AT can therefore alter the policy induced by the same agent LLM through a feedback channel unavailable to description rewriting.

B.2 FEEDBACK-MEDIATED TRAJECTORY EVIDENCE

We inspected full transcripts from the held-out evaluations of both conditions and matched examples by domain, task, trial, and simulation seed. All highlighted pairs use the same trial and seed. Because model sampling remained stochastic, they are matched replications rather than deterministic counterfactual replays.

Computed state prevents an unsupported refund. In Airline task 48, the user claimed that reservation `3RK2T9` had been booked 10 hours earlier and demanded a full refund (Tab. 9). The derived time fields prevented the full-scope agent from accepting a user claim that contradicted the database.

Direct entity resolution removes an ambiguous search. In Telecom task `[mobile_data_issue]airplane_mode_on|data_saver_mode_on|user_abroad_roaming_disabled_on[PERSONA:None]`, the full-scope agent could resolve a phone number directly to a line (Tab. 10, abbreviated to the identification window). Both trajectories passed, but the executable edit removed a clarification turn and two exploratory line lookups. This is a trajectory effect without an outcome difference.

Structured diagnostic feedback redirects a failing trajectory. In Telecom task `[mobile_data_issue]bad_vpn|data_mode_off|data_usage_exceeded|user_abroad_roaming_disabled_off[PERSONA:None]`, the customer had exhausted data on line `L1002` (Tab. 11, abbreviated to the diagnostic window). The mechanism is joint: the added lookup selected the correct entity, while the derived status converted raw quantities into an explicit diagnostic signal.

B.3 INTERPRETATION

The evidence supports a capability-bottleneck account of the domain pattern in Tab. 1. Description editing can change how an agent interprets an existing interface, but it cannot add a missing lookup, repair a mutation, or cause a tool to expose latent state. Airline and Telecom contained failures of precisely this form: refund eligibility depended on comparing timestamps, and telecom diagnosis depended on resolving the correct line and interpreting usage state. In these domains, executable edits produced decision-ready observations that changed the agent's next utterance or action. Retail was already high-performing and showed almost no difference between conditions, consistent with fewer failures being limited by the tool implementation itself.

Table 8: **Tools edited by full-scope AT-AT and by the descriptions-only ablation.** Labels in parentheses indicate which part of each agent-exposed tool changed.

Domain		Full AT-AT	Descriptions only
Airline	Tools edited	`book_reservation` (description); `cancel_reservation` (description); `get_flight_status` (description); `get_reservation_details` (code + description); `get_user_details` (description); `search_direct_flight` (code + description); `search_onestop_flight` (code); `send_certificate` (description); `update_reservation_baggages` (description); `update_reservation_flights` (description); `update_reservation_passengers` (description)	`update_reservation_baggages` (description); `update_reservation_flights` (description)
	Qualitative changes	Added timestamp parsing and reservation-time helpers. Reservation lookup now returns current time, booking age, 24-hour eligibility, flight-date bounds, and past/upcoming status. Direct and one-stop search results now carry their flight dates. Description edits added or clarified policy and usage guidance across booking, cancellation, lookup, search, certificate, baggage, flight-change, and passenger-change operations.	Clarified that baggage totals are absolute, documented paid-bag charges and certificate restrictions, and added flight-change guidance covering basic-economy restrictions, downgrade refunds, baggage recomputation, and passenger changes. Runtime behavior and feedback remained canonical.
Retail	Tools edited	`exchange_delivered_order_items` (description); `get_item_details` (code); `modify_pending_order_items` (code + description)	`cancel_pending_order` (description); `exchange_delivered_order_items` (description); `find_user_id_by_name_zip` (description); `modify_pending_order_items` (description); `return_delivered_order_items` (description)
	Qualitative changes	Corrected item lookup for object-backed products, affecting `get_item_details`. Corrected pending-order modification to apply the requested replacement variant's price and options. Clarified that exchanges and pending-order modifications only swap variants of the same product and cannot add or remove items.	Added explicit confirmation and irreversibility guidance, distinguished whole-order cancellation from item modification, required batching one-time return or exchange actions, clarified same-product variant constraints, and clarified preferred customer lookup. Runtime behavior and feedback remained canonical.
Telecom	Tools edited	`get_line_by_phone` (new: code + description); `get_customer_by_phone` (code + description); `get_data_usage` (code + description); `enable_roaming` (code + description); `disable_roaming` (description); `refuel_data` (code + description)	`get_details_by_id` (description); `send_payment_request` (description); `enable_roaming` (description); `disable_roaming` (description)
	Qualitative changes	Added formatting-insensitive phone normalization and a direct phone-to-line/customer lookup. Customer lookup now tolerates formatted numbers. Roaming responses now expose current and prior state; usage responses add remaining data and an exceeded flag; refueling responses add the resulting allowance and exceeded state.	Expanded object-status guidance in generic lookup, advised re-fetching bills after payment requests, and distinguished carrier-side roaming from the device's Data Roaming setting. Runtime behavior and feedback remained canonical.

Table 9: **Airline task 48: computed state prevents an unsupported refund.**

Trajectory step	Full AT-AT	Descriptions only
Tool feedback	`created_at=2024-05-02`, `current_datetime=2024-05-15`, `hours_since_booking=320.95`, `within_24h_of_booking=False`	`created_at=2024-05-02`; no current time, elapsed time, or eligibility flag
Agent response	"Booked more than 24 hours ago (about 13 days ago, not 10 hours ago)"; refuses refund	"Booking made about 10 hours ago (within 24 hours)"; offers and executes cancellation
Outcome	Correctly preserves reservation; reward 1	Incorrectly cancels and refunds; reward 0

The transcript analysis also cautions against equating trajectory improvement with fewer turns or universal success. Some richer feedback shortened a trajectory without changing reward, and full-scope AT-AT did not improve Retail over baseline. The result is therefore conditional: executable optimization is most useful when the existing interface withholds, ambiguously represents, or incorrectly computes task-relevant state.

Table 10: **Telecom roaming task: direct entity resolution removes an ambiguous search.**

Trajectory step	Full AT-AT	Descriptions only
Tool feedback	`get_line_by_phone` directly returns `customer_id=C1001`, `line_id=L1002`, `roaming_enabled=False`	Customer lookup returns `line_ids=[L1001,L1002,L1003]` without identifying the relevant line
Agent action	Immediately calls `enable_roaming(C1001, L1002)`	Asks the user to identify the line, then queries `L1001` and `L1002` separately before enabling roaming
Outcome	Reward 1	Reward 1

Table 11: **Telecom data-exceeded task: structured diagnostic feedback redirects a failing trajectory.**

Trajectory step	Full AT-AT	Descriptions only
Tool feedback	Correct line `L1002`: `15.1/15.0 GB`, `data_remaining_gb=-0.1`, `is_data_exceeded=True`	Wrong line `L1001`: `3.2/5.0 GB`; no derived remaining-data or exceeded fields
Agent response	Attributes outage to exhausted data and offers refueling	Concludes the cap was not reached and investigates account restrictions
Subsequent feedback	After a 2 GB refuel: `is_data_exceeded=False`; agent reports restored access	Never refuels the affected line
Outcome	Reward 1	Reward 0

Several limitations constrain the causal claim. There was one independently sampled candidate per condition and domain; the experiment did not take one full-scope candidate and mechanically remove only its executable edits. The between-condition contrasts were not subjected to a direct paired permutation test, and the trajectory examples were selected post hoc to illustrate observed mechanisms, so they should not be read as an estimate of mechanism frequency. The strongest supported conclusion is thus that executable edits *can* materially improve agent success by changing the action and observation interface, and that this mechanism occurred in Airline and Telecom trajectories; the present experiment does not establish that code-edit permission alone caused the complete aggregate gap.

C Additional Analysis

Effectiveness of coding agents as text optimizers. Our results here, as well as in prior work, suggest that using a coding agent as an optimizer is helpful: coding agents' ability to reason over multiple files and make selective edits in a multi-turn fashion may be useful in text optimization settings. Here, AT-AT outperformed the ablation method without the coding agent harness, as well as the prior tool and prompt optimizers that do not use a coding agent harness, DRAFT, ToolObserver, and GEPA (Qu et al., 2025; Hallinan et al., 2026; Agrawal et al., 2026). This echoes the outperformance of FAPO (Kassianik et al., 2026), a recent prompt optimization method using a Claude Code coding agent as an optimizer, compared to GEPA (Agrawal et al., 2026), the previous state-of-the-art prompt optimizer. Meta-Harness (Lee et al., 2026b) also uses the Claude Code coding agent as a harness optimizer. LLM capabilities have advanced steadily, and coding agents utilize the ability of LLMs to create and execute upon long-horizon plans. Given the promise of coding agents, we encourage future text optimization work to experiment with coding agents as part of their methodology. The Pi agent (Earendil Inc. and Contributors, 2026) that we use in AT-AT, and similar other coding agents that are open-source, lightweight, and extensible, may be good candidates for further experiments and worthwhile to compare to closed coding agents like Claude Code.

Evaluation stochasticity. Agent optimization involves two distinct sources of stochasticity: evaluation noise and optimizer noise. Evaluation noise arises because repeated agent executions can produce different trajectories and outcomes even when the model, harness, tasks, environment, and decoding configuration are held constant (Mustahsan et al., 2025). Differences in evaluation infrastructure configuration or infrastructure crashes can also lead to task failures and contribute to evaluation noise (Anthropic, 2026). Consequently, rankings based on single benchmark point estimates, particularly when the reported differences are small relative to run-to-run variation, should not be interpreted as reliable evidence that one system outperforms another. Our paired permutation statistical testing protocol accounts for evaluation noise and provides statistical evidence for AT-AT's improvements on τ^2-bench-verified. However, it does not address optimizer noise: repeated runs of an LLM-based

optimizer may produce different tool updates and, consequently, different downstream effects. Because we evaluate one toolset from each optimizer, our reported results are conditional on those particular optimization runs. Future work should repeat the complete optimization-and-evaluation pipeline to characterize both sources of variation, reporting uncertainty across optimizer runs as well as repeated downstream evaluations. Given the imperfect nature of benchmarks and evaluations, the ultimate test of an ML method is if it works in real-world scenarios; our production MCP server experiments demonstrate that AT-AT may provide value in a deployment setting, although replication across deployments and time remains necessary to establish robustness.

Failure to achieve gains on Retail. No optimizer significantly changed Retail reward (Tab. 1); all optimized toolsets scored slightly below the 77.00% baseline (74.08–76.08%; all $p \geq 0.099$). We attribute this pattern to two factors. First, Retail offers limited optimization headroom. The baseline reward is already quite high at 77%, and an LLM audit of 24 failed baseline trajectories per domain found that most Retail failures were agent reasoning errors that occurred even though the tools had returned the necessary information. Failures classified by an LLM as addressable via tool implementations accounted for an estimated 3.1% of Retail simulations, compared with 8.9% in Telecom, where AT-AT achieved its only significant gain. Second, the small declines are within evaluation noise. GEPA's selected toolset is byte-identical to the unoptimized tools, yet its evaluation scored 0.92 points below the baseline (76.08%), and the edited toolsets differ from this control by only -2.00 to -0.17 points (all $p \geq 0.20$).

D Enterprise MCP Edits: Further Details

D.1 Representative tasks

Table 12 gives a high-level description of representative tasks for each of the three enterprise MCP servers evaluated in Table 4.

Table 12: **Representative tasks for each enterprise MCP server.**

MCP server	Representative tasks
Core Developer Tools	• Interpret an application performance metric • Investigate VPN-related host reachability • Diagnose repeated software installation failures
Observability Collaboration Toolbox	• Classify date-scoped support threads • Evaluate automated support-answer quality and recommend fixes • Compare activity across support channels
Enterprise Engineering Context and Best Practices	• Assess whether dependency changes caused canary latency • Diagnose a linked CI/CD failure • Reconcile data-compaction settings with an underlying messaging topic

D.2 Description-update examples

Tab. 13 summarizes representative AT-AT updates from the enterprise MCP evaluation. The examples are paraphrased, with organization-specific tool names and infrastructure details omitted. The accepted candidates changed natural-language descriptions only; tool names, parameter contracts, and backend implementations were unchanged.

D.3 Core Developer Tools cohort breakdown

The main paper row aggregates two cohorts of Core Developer Tools tasks. Tab. 14 reports their held-out results separately and supports the observation that task completion gains were concentrated in the documentation-focused cohort.

Table 13: **Paraphrased examples of AT-AT description updates for enterprise MCP tools.**

Tool category	Paraphrased guidance added by AT-AT
Documentation and enterprise search	Use page-retrieval tools only with canonical URLs supplied by the user or returned by search, and return to search after a failed retrieval instead of guessing URL variants. Keep conjunctive keyword filters minimal, place additional concepts in the semantic query, and relax filters after empty results. Use namespace-specific retrieval only when the namespace is known.
Collaboration toolbox	Normalize channel references to the identifier format expected by each tool and resolve display names before calling tools that require channel IDs. Distinguish access-denied or unindexed channels from genuine zero-result searches, and treat an unresolved workspace or channel identifier as an identity or access problem rather than an empty result.
Enterprise engineering context retrieval	Clarify that the tool searches indexed documentation, code, and discussion content rather than live executions, incidents, dashboards, metrics, or logs. Avoid near-duplicate retries after no relevant context is found, and use focused calls for separate parts of a multi-part request.

Table 14: **Held-out Core Developer Tools results by cohort.** Held-out test results with the original (Orig.) and AT-AT-optimized tool descriptions for each cohort; n is the number of scored test tasks. Mean reward and content score are shown as percentages. Session cost covers the agent and user simulator.

	Mixed retrieval (n=5)		Documentation-focused (n=5)	
Metric	Orig.	AT-AT	Orig.	AT-AT
Task pass rate (%)	**80.0**	60.0	40.0	**100.0**
Mean reward (%)	**89.6**	80.0	66.6	**100.0**
Content score (%)	**81.0**	74.5	78.5	**86.0**
Tool calls	**80**	83	**52**	60
Tool errors	6	**1**	**5**	10
Session cost	**$0.494**	$0.528	**$0.344**	$0.398

D.4 Example task evaluation

The following example replaces organization-specific metric and service names while preserving the evaluated request and checklist structure.

Task prompt. *What does* `application.ui_delay_by_label` *measure for client-device telemetry, and how is* `ui_label` *attributed?*

Success criteria. Before either the baseline or optimized run was executed, GPT-5.6 Luna generated the following user-visible criteria from the task prompt alone:

1. Clearly define the metric's measured quantity and scope for client-device telemetry.
2. Explain how `ui_label` is assigned or attributed, including the relevant source or attribution rule.
3. Distinguish the metric measurement from the label-attribution process without unsupported speculation.

Fixed content rubric. Luna converted those criteria, before seeing either trajectory, into three independently scored clauses: two content clauses corresponding to the first two criteria and one correction clause corresponding to the third.

LLM-as-a-judge procedure. We used GPT-5.6 Luna in two separate absolute-scoring calls. The task-success judge received the success criteria and one rendered transcript, then returned a binary `met_criteria` decision with a one-sentence reason. Separately, the content-rubric judge received the fixed checklist and the rendered transcript. For every clause it returned `met`, `partial`, or

not_met, a short reason, and an exact transcript quote for positive credit. Code verified that each quote occurred in the transcript and downgraded unsupported verdicts. Neither judge saw the other transcript or a label indicating whether the transcript was from before or after optimization.

E COMPOSITION WITH HARNESS OPTIMIZATION

Eq. 1 treats the harness as a fixed component of each agent. However, the harness can also be optimized. Let $\mathcal{X}_H$ and $\mathcal{X}_E$ denote separate task distributions used for harness and environmental tool optimization, respectively, with corresponding tasks x_H and x_E and trajectories τ_H and τ_E. Let $T_E^0 = (T_{E_D}^0, T_{E_C}^0)$ denote the fixed initial environmental toolset. Harness engineering optimizes H to maximize expected performance over tasks and trajectories (Lee et al., 2026b):

$$H^* = \arg\max_H \mathbb{E}_{x_H \sim \mathcal{X}_H,\ \tau_H \sim p_M(\cdot|H, x_H, T_{E_D}^0, T_{E_C}^0)} \left[r(\tau_H, x_H) \right]. \tag{3}$$

Harness optimization can be composed with environmental tool optimization. A harness optimized on $\mathcal{X}_H$ can be held fixed and used to generate trajectories on $\mathcal{X}_E$ for optimizing the environmental tools. For a population of models $M \sim \mathcal{M}$, this two-stage objective is

$$\begin{aligned} T_{E_D}^*, T_{E_C}^* &= \arg\max_{T_{E_D}, T_{E_C}} \mathbb{E}_{M \sim \mathcal{M},\ x_E \sim \mathcal{X}_E,\ \tau_E \sim p_M(\cdot|H^*, x_E, T_{E_D}, T_{E_C})} \left[r(\tau_E, x_E) \right] \\ \text{subject to} \qquad H^* &= \arg\max_H \mathbb{E}_{M \sim \mathcal{M},\ x_H \sim \mathcal{X}_H,\ \tau_H \sim p_M(\cdot|H, x_H, T_{E_D}^0, T_{E_C}^0)} \left[r(\tau_H, x_H) \right]. \end{aligned} \tag{4}$$

F AT-AT PROMPTS

This appendix lists the prompts given to the Pi agent optimizer in AT-AT. Prompt text is reproduced verbatim, except that line breaks within paragraphs are reflowed.

System prompt. The system prompt is the concatenation, separated by blank lines, of (1) the optimization objective, (2) benchmark-specific editing rules, and (3) the generalization instruction.

Optimization objective

```
You are given an agent's tools (source you may edit) and transcripts of
the agent attempting tasks with them, each labelled with its reward.
Edit the tools so the agent succeeds on more tasks.

Read the transcripts for places where the tools -- not the agent's
judgment -- got in the way. Look for: the agent picking the wrong tool
or wrong arguments because the docs don't make a tool's purpose or
usage clear; results it can't act on (unclear, incomplete, or silent)
that force it to guess or retry; and repeated calls or workarounds for
something the tools should let it do in one step. Effective tools make
the right action easy to choose, behave as their docs describe, and
hand back what the agent needs to take the next step. Edit the tools to
remove those obstacles, and read the successful transcripts too, so a
change doesn't break what already works.

Constraints: keep every tool's name and signature unchanged and the
source valid; do not encode task identifiers, specific inputs, or
expected answers into the tools.

Apply edits with the edit tool. The harness validates the result; if it
fails, fix the underlying issue rather than working around the check.
```

The editing rules state how to edit the tools of a given benchmark without breaking them. For τ^2-bench-verified, where all tools are defined in a single `tools.py` file:

Editing rules: τ^2-bench-verified

```
The tools live in a single tools.py. Keep it importable: preserve all
imports, class names, and @is_tool method signatures (names, types,
order).
```

For the τ^2-bench-verified ablation that edits tool descriptions only (Sec. 4.1.1), the tool descriptions are extracted into an editable `descriptions.md` file:

Editing rules: τ^2-bench-verified, descriptions only

```
Edit ONLY the tool descriptions in descriptions.md (each `## <tool>`
heading names a tool). The full tools.py is provided read-only for
context; you cannot change code, only the description text under each
heading.
```

For TerminalBench-2 and TBLite, where the optimized tools are TypeScript modules in the OpenCode harness:

Editing rules: OpenCode

```
Each tool is a .ts module exporting `export const <Name>Tool =
Tool.define("<id>", {...})`, optionally with a companion .txt
description. Preserve every Tool.define id and the exported constant
name; do not add imports or reference runtime-context symbols at module
top level; preserve each execute() return shape { title, output,
metadata }.
```

Generalization instruction

```
The tools are scored on held-out tasks, not the ones in these
transcripts. Prefer edits that address the general defect a transcript
reveals over edits that only fit the specific observed case.
```

User prompt. In the main τ^2-bench-verified experiments (Sec. 4.1), the first user message contains a digest of the agent traces followed by a short instruction. The digest includes every failing task's trace, followed by the traces of the first three passing tasks. Each trace is truncated to 4000 characters and preceded by a header giving its task identifier, outcome, and reward:

User prompt

```
### task <task id> [FAIL] (reward=<reward>)
<agent trace, truncated to 4000 characters>
...
### task <task id> [PASS] (reward=<reward>)
<agent trace, truncated to 4000 characters>

Above are transcripts of an agent attempting tasks with the current
tools, each labelled with its reward. Edit the tools as instructed.
```

The descriptions-only ablation and the cross-LLM generalization experiment (Sec. 4.1), and the TerminalBench-2, TBLite (Sec. 4.2), and MCP (Sec. 4.3) experiments omit the digest from the user prompt. Instead, the full agent traces are written to the optimizer's workspace as read-only files, one per task, together with an index file listing each trace's task identifier, outcome, and reward. The user prompt directs the optimizer to these files:

User prompt without trace digest

```
The complete baseline transcripts are available as read-only workspace
files. Start with BASELINE_TRANSCRIPTS_INDEX.md, then use the read tool
to inspect the relevant per-task files (in pages when they are large).
Ground your edits in both failed and successful runs.
```

If a proposal fails validation, the optimizer's session is resumed with the following user prompt instead:

User prompt on retry

```
Your previous attempt did not pass validation. Re-read the files you
edited, reconsider, and try again; fix the underlying issue rather than
working around the check.
```