Autonomous Agent Learning in Production

LithosAI, Inc.
www.lithosai.com

Abstract

Agent applications are increasingly deployed in production, but keeping them accurate and cost-effective as model providers, prompts, and traffic distributions drift remains a manual, ad-hoc process. Existing automated optimizers either target a single component (prompt-level methods such as GEPA [5]) or assume a clean, deterministic evaluator decoupled from real traffic (AlphaEvolve [6]), neither of which captures how agent applications actually fail in the wild. We present ALP, a fully autonomous optimizer that closes the loop between production signals and agent improvements. The system takes a data-centric view: each session constructs its own dataset from recent production requests for the target agent, anchors evaluation in a project-defined LLM-judge rubric, and then runs an agent-guided tree search over candidate edits to the agent's source. The orchestrator reflects on per-case judge outcomes to propose new candidates, branches from the current Pareto frontier, and prunes strictly dominated subtrees. On TerminalBench [8] and SWE-Bench Verified [7], the optimizer lifts pass-rate up to 16 points and cuts cost 2.3 to 2.5x over the baseline within per-session budgets of a few hundred dollars.

1 Introduction

As agent applications have moved from simple demos into production, a new class of operational problem has emerged: keeping a deployed agent both accurate and cost-effective as the conditions around it shift. Model providers release new families on a monthly cadence, prompt templates accumulate edge cases, tool APIs change underneath, and the distribution of real user inputs wanders. Engineering teams currently respond by hand-editing prompts, swapping models, and rerunning ad-hoc evaluations, an iteration loop that does not scale across many agents and many revisions.

Existing automated approaches address neighboring problems. Programmatic prompt optimizers (DSPy [3], Self-Refine [2], Reflexion [4], GEPA [5]) rewrite single prompts but cannot restructure the surrounding agent, orchestrate models, or introduce new tools. AlphaEvolve [6] evolves whole code artifacts but assumes a deterministic scalar evaluator, suited to algorithmic discovery rather than the judge-driven feedback production agents produce.

We present ALP, an autonomous optimizer for agent applications in production. The design rests on two ideas: a *data-centric* formulation that anchors each session in a dataset sampled from recent production requests for the target agent, paired with the project's LLM-judge rubric, with both fixed for the session so candidates are directly comparable; and an *agent-guided tree search with reflection*, where an LLM-driven orchestrator maintains a tree of candidate versions, branches from the current (pass-rate, cost) Pareto frontier, prunes dominated subtrees, and reflects on per-case judge outcomes to propose the next round. The output of a session is a small set of frontier versions a project owner can choose from.

AI Agents for Discovery in the Wild (AID-Wild), Workshop at ACM CAIS 2026.

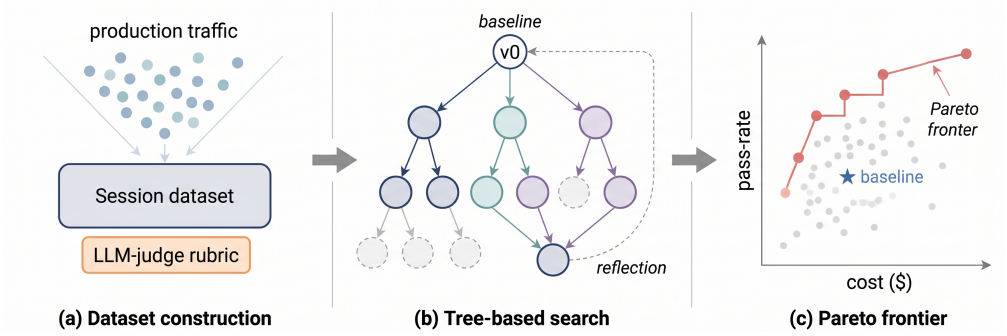

Figure 1: ALP workflow. **(a)** A session dataset is sampled from recent production traffic and paired with the project's LLM-judge rubric. **(b)** The optimizer grows a tree of agent versions, branching from the current Pareto frontier, pruning dominated subtrees (dashed), and reflecting on per-case judge outcomes to propose the next round. **(c)** Candidates are plotted in (pass-rate, cost) space; the frontier (red) gives the user a small set of versions to choose from, with the baseline ($\star$) for reference.

2 Problem Setting

An agent application in production is a composition of many tunable pieces: an orchestrator program that drives control flow, one or more prompts, a set of tools the agent can call, a memory or context-construction policy, a scaffold (single-step, multi-step, or multi-agent), and a choice of model behind each LLM call. Each of these can be edited, and each interacts with the others. A prompt that works well under one model fails under another; an added tool changes which prompts pay off; a memory policy that helps a debugging session can degrade an open-ended one. Optimization in this setting is necessarily over the whole composition, not any single component.

Improvement is also not a scalar. A change that lifts pass-rate by raising token spend is not unambiguously better, and the cheapest viable variant is often what a project ends up shipping. We treat the optimization target as the (pass-rate, cost) Pareto frontier, with frontier movement, rather than gain on a single dimension, as the criterion of progress.

Operating in production adds three constraints that purely synthetic benchmarks do not impose. First, the only valid input distribution is the one users actually send, so candidate evaluation must be anchored to real traffic rather than a fixed academic benchmark. Second, in most application domains there is no deterministic verifier; an LLM-judge under a project-defined rubric stands in for ground truth, and the resulting reward is noisy and subjective. Third, candidate exploration must be safe: speculative agent versions cannot be allowed to influence the production response, so evaluation runs in isolation against the session dataset, with promotion to live traffic gated by a human.

Existing automation matches none of these constraints simultaneously, which motivates the specialized harness described next.

3 Methodology

An ALP session is a closed loop with two components: a session dataset sampled from production that defines the reward signal (Section 3.1), and an agent-guided tree search that proposes, evaluates, and prunes candidate edits to the agent's source (Section 3.2). Figure 1 sketches the loop end-to-end.

3.1 Data-Centric Optimization in Production

A session samples recent production requests for the target agent and freezes them as the session dataset, paired with a short natural-language judge rubric. Both are fixed for the duration of the session, so candidate scores are directly comparable.

The sampling policy is configurable, and choosing it amounts to declaring the goal of the session: targeting recent failures for debugging, broad recent traffic for general improvement, or a prior dataset

for regression checks against a deployed candidate. The downstream search is indifferent to which policy produced the dataset.

Each candidate response is scored by the project's LLM-judge. Pass-rate from the judge and dollar cost from the underlying execution trace together place every candidate at a point in (pass-rate, cost) space, the optimizer's two-axis reward. Candidates are evaluated in isolation from the running production agent; promotion of a chosen winner is left to a human.

3.2 Tree-Based Search and Reflection

The optimizer maintains a tree of candidate agent versions, each node applying a single LLM-generated edit to its parent. The orchestrator selects parents from the current Pareto frontier in (pass-rate, cost) space, expanding distinct cost-quality regions in parallel. Before each round, it reflects on per-case judge outcomes on the parent (which cases failed, what the judge said, how pass-rate and cost shifted) and proposes candidate edits ranging from prompt-level adjustments to broader structural or architectural changes (added tools, alternate scaffolds, model swaps, multi-agent decomposition); worker LLMs materialize each edit. After evaluation, dominated versions are pruned, and pairs of independently-improving edits are tried in combination. The loop terminates when the budget is exhausted or the frontier stalls for several rounds.

Three levers for production agent optimization. ALP is a prototype for a broader position: production agent optimization is its own specialty, distinct from general-purpose code editing. Three levers, in our view, matter most going forward. First, a specialized harness rather than a generic coding agent (Claude Code, Codex, and similar), with judge rubric, production traces, and Pareto-aware branching as first-class operations. Second, application-specific memory and context as optimization knobs alongside prompts and tools (compressed user-preference memory for support agents, codebase indices for coding agents, low-latency summaries for voice agents) rather than fixed components. Third, heterogeneous model routing per LLM call to the cheapest model that suffices, tuned by the same search rather than designed by hand, exploiting the narrowing gap between open-weight and frontier models.

4 Evaluation Results

Setup. We evaluate the system on two coding benchmarks that stand in for the production traffic a deployed agent would see: TerminalBench [8], in which an agent must complete shell-based engineering tasks inside a sandboxed environment, and SWE-Bench Verified [7], in which the agent must patch real GitHub issues against open-source repositories. Each benchmark's task set plays the role of a session dataset, and each task's deterministic verifier (test pass or fail) plays the role of the LLM-judge rubric, with cost measured as the dollar cost of the agent's API calls per task. The deployed baseline in each case is a single frontier coding model wired to the project's default scaffold, matching the configuration a team would ship before turning ALP on. Per-session budgets sit in the low hundreds of dollars and complete within hours of wall-clock.

Headline numbers. On SWE-Bench Verified, the frontier reaches 79.0% pass-rate at $0.24 per task, against a baseline of 75.8% at $0.55, a gain of 3.2 points at 2.3x lower cost. On TerminalBench, the frontier reaches 80.1% pass-rate at $0.53 per task, against a baseline of 64.0% at $1.31, a gain of 16.1 points at 2.5x lower cost. The TerminalBench path through the search makes the contribution of each edit class visible: structural edits to the harness (added file inspection and search tools, an explicit verification step, scaffold changes for vision tasks) lift pass-rate from 64% to 77.5% on the same model, and a subsequent architectural edit, mixing a cheaper open model on the easier slice of the workload with a frontier model on the harder slice, pushes the frontier to 80.1% while reducing cost. On SWE-Bench, the frontier is dominated by architectural edits of the same kind: prompt-only and tool-only edits did not move the frontier, and the winning candidates compose two or more models per workload. Across both benchmarks the loop ran on session budgets in the low hundreds of dollars and terminated when the frontier stopped moving for several consecutive rounds.

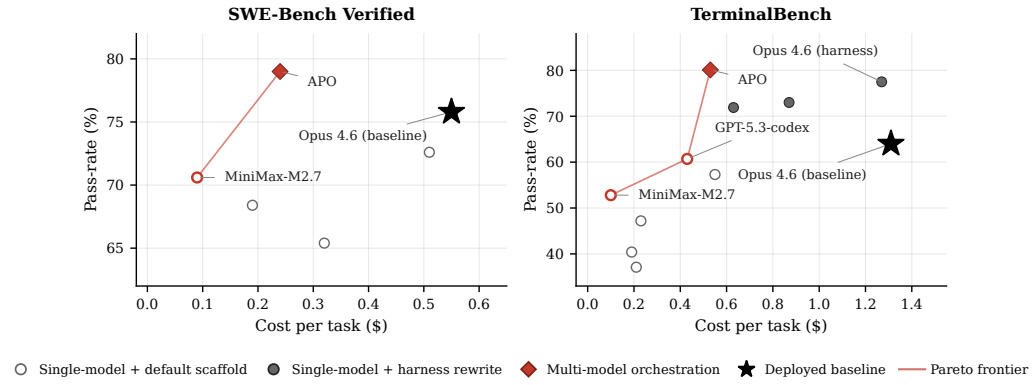

Figure 2: Pareto frontiers produced by ALP on SWE-Bench Verified (left) and TerminalBench (right). Each dot is a candidate version; marker style encodes the edit class (open: single-model, default scaffold; filled: single-model, harness rewrite; red diamond: multi-model orchestration). On both benchmarks the frontier dominates the deployed baseline ($\star$): an ALP candidate is simultaneously more accurate and cheaper.

5 Related Work and Position

ALP draws on three adjacent threads but combines them in a way none admits on its own. GEPA and related prompt-level optimizers (DSPy, Self-Refine, Reflexion) show that natural-language reflection on per-case feedback is a rich signal, and the Pareto-evolutionary frontier of GEPA in particular is a direct ancestor of the search procedure described here; what GEPA does not do is rewrite the surrounding agent, change models, or add tools, so its reach ends at the boundary of the prompt. AlphaEvolve mutates whole code artifacts but assumes a deterministic scalar evaluator, which holds for algorithmic discovery and breaks for production agents judged by an LLM under a project-specific rubric where cost matters as much as accuracy. General-purpose coding agents provide the editing capability but lack the optimization-specific orchestration: judge-aware context, Pareto-aware branching, and dollar-cost as a first-class objective. ALP targets the regime in between: whole-agent edits like AlphaEvolve, an LLM-judge over production traffic like the prompt optimizers, an explicit (pass-rate, cost) Pareto frontier rather than a single scalar, and an orchestrator specialized to agent-application semantics rather than to generic code editing.

Our position. The next round of progress in agent infrastructure will come not from bigger models or smarter prompts but from specialized, data-driven optimization harnesses that close the production feedback loop for whole agent applications, treat memory and routing as first-class optimization knobs, and exploit the narrowing open- and closed-model gap to reduce cost without sacrificing quality.

References

[1] 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.

[2] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, et al. Self-Refine: Iterative Refinement with Self-Feedback. In *NeurIPS*, 2023.

[3] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, et al. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv:2310.03714*, 2023.

[4] Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language Agents with Verbal Reinforcement Learning. In *NeurIPS*, 2023.

[5] Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, et al. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. *arXiv:2507.19457*, 2025.

[6] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, et al. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery. Technical report, Google DeepMind, 2025.

[7] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *ICLR*, 2024.

[8] Mike Merrill, Alex Shaw, Aleksander Boruch-Gruszecki, Vivek Hebbar, Ori Yonay, Anthony Liu, et al. Terminal-Bench: A Benchmark for AI Agents in Terminal Environments. Laude Institute, 2025.