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 sent to 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 to identify opportunities for further improvement, and prunes strictly dominated subtrees. On TerminalBench [8] and SWE-Bench Verified [7], the optimizer lifts pass-rate by up to 16 points and cuts cost 2.3× to 2.5× over the baseline within per-session budgets of a few hundred dollars.

Keywords

Automated agent optimization, LLM-as-judge, Pareto frontier, Production AI systems, Model routing

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, and the distribution of real user inputs shifts. 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, including DSPy [3], Self-Refine [2], Reflexion [4], and the Pareto-evolutionary scaffold of GEPA [5], improve a single prompt or chain by automated rewriting, but cannot restructure the surrounding agent scaffold, orchestrate models, or introduce new tools. AlphaEvolve [6] applies LLM-driven evolutionary search to whole code artifacts, but assumes a deterministic, scalar evaluator suited to algorithmic discovery rather than the noisy, multi-objective, judge-driven feedback that production agents actually produce.

We present ALP, an autonomous optimizer for agent applications in production. The design rests on two ideas. The first is a *data-centric approach*: instead of optimizing against a fixed benchmark, each session constructs its own dataset from recent production requests for the target agent, paired with the project's LLM-judge rubric. Different sessions can target different goals (debugging a known failure mode, lifting overall pass-rate, guarding against regressions) by varying the sampling policy, while the rest of the optimization loop is unchanged. The dataset and rubric are fixed for the duration of the session, so candidates are directly comparable.

The second idea is an *agent-guided tree search with reflection*. The optimizer, itself driven by an LLM, maintains a tree of candidate agent versions in which each node applies a single modification to its parent. After every batch of evaluations, the orchestrator reflects on the per-case judge outcomes, which cases failed, what the judge said about each failure, and how pass-rate and cost moved relative to the parent, and uses that signal to propose the next round of candidates. The search branches from the current Pareto frontier in (pass-rate, cost) space and prunes strictly dominated subtrees, so the output of a session is not a single winner but a concise set of frontier versions a project owner can choose from.

This paper makes the following contributions.

- A data-centric formulation in which each optimization session is anchored to a dataset sampled from production traffic for the target agent.
- An agent-guided, reflection-driven tree search over agent versions, with Pareto-pruned exploration at the granularity of a whole agent application rather than a single prompt or chain.
- An empirical evaluation on TerminalBench [8] and SWE-Bench Verified [7], showing pass-rate gains of up to 16 points and cost reductions of 2.3 to 2.5x over the deployed baseline within per-session budgets of a few hundred dollars.

2 Problem Setting

An production agent application 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 merely a single component.

Improvement is also not a scalar. A change that lifts pass-rate by raising token spend is not necessarily 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 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 approaches do not satisfy all three constraints at once. Prompt-level optimizers can hit a judge rubric and isolate evaluation, but reach only a single component of the agent. Whole-code evolutionary search can mutate any part of the composition, but presumes a deterministic scalar evaluator rather than an LLM-judge over real traffic. General-purpose coding agents can edit the whole agent and respond to instructions, but lack judge-aware reward, dollar cost as a first-class objective, and Pareto-aware branching. Each gives part of what the regime needs; none gives the full set.

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 begins by sampling recent production requests for the target agent and freezing them as the session dataset, paired with a short natural-language judge rubric (e.g., "Did the agent address the user's request? Is the response factually correct?"). Both the dataset and the rubric 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. A *debugging* session preferentially samples cases tagged with recent failures or judge complaints, so the optimizer spends its budget on the slice of traffic the project is actively trying to fix. An *improvement* session samples broadly across recent traffic to surface pass-rate or cost gains across the typical workload. A *regression* session reuses a dataset from a prior run, paired with a deployed candidate, to verify that subsequent edits have not silently degraded other workloads. The downstream search procedure is indifferent to which sampling 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 in which each node is a candidate version of the agent and each edge is a single LLM-generated edit applied to its parent. Restricting nodes to one edit per parent makes the tree interpretable: when a node lifts the frontier, the cause is unambiguous.

Branching. The orchestrator selects a parent from the current Pareto frontier in (pass-rate, cost) space. When the frontier holds several non-dominated versions trading differently along the two axes, any of them can serve as a parent, so distinct cost-quality regions of the frontier are expanded in parallel rather than collapsed onto a single "best" candidate.

Reflection. Before proposing the next round of edits, the orchestrator inspects the per-case judge outcomes for the parent: which cases failed, what the judge's natural-language reason said about each failure, and how pass-rate and cost shifted relative to the grandparent. Conditioned on this evidence, it proposes a few candidate edits, ranging from prompt-level adjustments to broader structural or architectural changes (added tools, alternate scaffolds, model swaps, multi-agent decomposition). Worker LLMs, with smaller context windows, materialize each proposed edit on top of the parent.

Pruning. At the end of each round, versions strictly dominated on both pass-rate and cost are removed from the active set. Pruning is domination-based rather than threshold-based, so the frontier preserves low-cost low-quality variants alongside high-cost high-quality ones; in practice, the cheapest viable variant is often the one a project ends up promoting.

Compounding. When two edits A and B each independently move the frontier, the next round tries the combined edit $A + B$. Compound gains are most reliable when the two edits target different aspects of the agent (a prompt change paired with a structural change, say) and weakest when they target the same aspect.

The loop terminates when the dollar budget is exhausted or the frontier has not moved for several consecutive rounds.

4 Three Levers for Production Agent Optimization

ALP is a prototype supporting a broader position: production agent optimization is its own specialized task, distinct from general-purpose code editing. We highlight three levers we believe will matter most going forward.

4.1 A specialized harness, not a coding agent

Off-the-shelf coding agents (Claude Code, Codex, and similar) are general assistants: they read a codebase, follow instructions, and edit files. Production agent optimization requires more specialization. The reward signal is not test failure or compile success but per-case LLM-judge pass/fail under a project-specific rubric, applied to a multi-step trace with token costs attached. Productive edits are not refactors but targeted changes to prompts, scaffolds, tool choice, memory, and model selection, each requiring different priors.

4.2 Application-specific memory and context

Most production agents today use generic context strategies: full conversation history, naive vector RAG over a knowledge corpus, or static system prompts. These leave both quality and cost on the

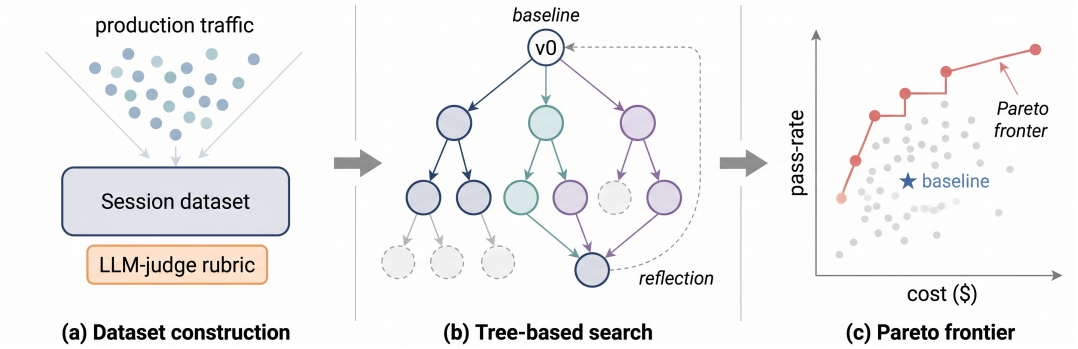

Figure 1: ALP workflow. (a) Dataset construction: a session dataset is sampled from recent production traffic and paired with the project's LLM-judge rubric. (b) Tree-based search: the optimizer grows a tree of agent versions, each node applying one edit to its parent; the search branches from the current Pareto frontier, prunes strictly dominated subtrees (dashed), and reflects on per-case judge outcomes to propose the next round of candidates. (c) Pareto frontier: candidates are plotted in (pass-rate, cost) space; the Pareto frontier (red) gives the user a small set of versions to choose from, with the baseline (⋆) shown for reference.

table. A customer-support agent benefits from compressed user-preference memory; a research agent from cross-session retrieval over prior findings; a coding agent from codebase indices and recent-change context; a voice agent from low-latency context summaries. We argue that memory and context policies should be optimization knobs alongside prompts and tools, mutated by the same search procedure and evaluated against the same judge. Treating memory as a tunable rather than a fixed component is, in our view, one of the more underexplored axes of agent optimization, and one of the most promising for cost reduction.

4.3 Heterogeneous model routing

Open-weight models have closed much of the gap to closed frontier models on many production tasks. The cost-saving move in this regime is not "use the cheapest model everywhere" but "route per call to the cheapest model that suffices". A reasoning-heavy planning step might still call a frontier model; a routine summarization or extraction step inside the same agent can call a much cheaper open-weight model. Designing such routing policies by hand is hard because the routing decision interacts with prompt design and tool choice. An automated optimizer that mutates model selection per call, evaluates the resulting agent against the session judge, and prunes routings that regress quality is a natural fit. The TerminalBench results in Section 5 reflect exactly this dynamic: the frontier-moving edits combine a frontier model on harder cases with a cheaper model on the rest, a routing the search found that a single-model baseline could not satisfy.

5 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 running ALP. Per-session budgets sit in the low hundreds of dollars and complete within hours of wall-clock.

Results. 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.

Caveats. Three points qualify the headline result. First, the benchmarks here use deterministic verifiers, a more forgiving signal than the LLM-judge rubric the framing targets; in production, judges are non-deterministic and the same trace can score differently across runs, so practitioners should expect to stabilize the reward by averaging multiple judge calls per case, majority voting across them, or tightening the rubric until inter-run agreement is high. A run against a real LLM-judge on production traffic is reserved for an extended version of this work. Second, tree search over LLM-generated edits is variance-dominated, and the frontiers reported are from a single search run per benchmark; a different seed could land at slightly different points along the same curve, so the results read as evidence of frontier dominance rather than a

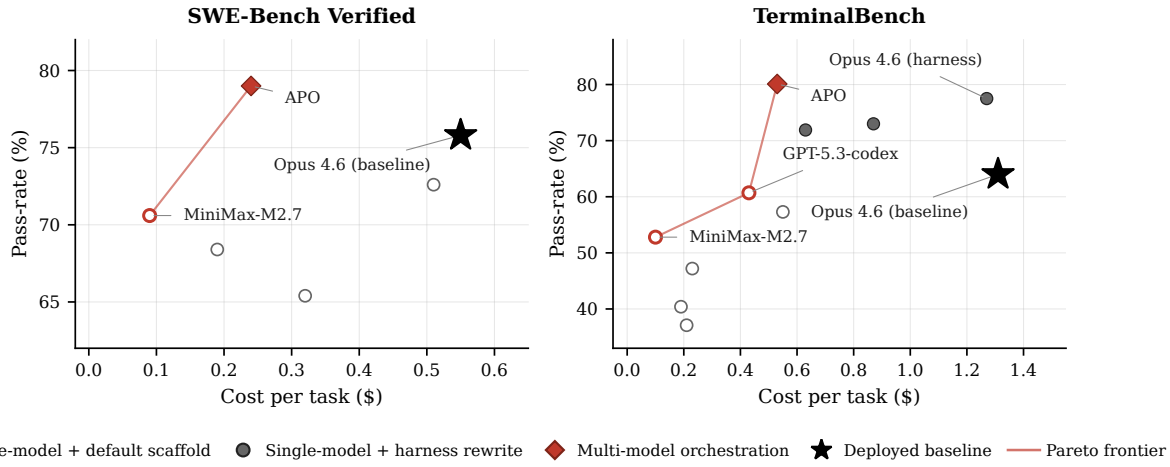

Figure 2: Pareto frontiers produced by ALP on SWE-Bench Verified (left) and TerminalBench (right). Each dot is a candidate agent version explored during the session, with marker style encoding the edit class that produced it: open circle for a single-model candidate on the project's default scaffold, filled circle for a single-model candidate on a harness rewrite, red diamond for a multi-model orchestration candidate. The red curve traces the non-dominated frontier; frontier and baseline points carry leader-line labels. On both benchmarks the frontier dominates the deployed baseline (⋆): a ALP candidate exists that is simultaneously more accurate and cheaper.

precise estimate. Third, the per-task cost reported is the cost of running the evolved agent on each task; the orchestrator's own LLM cost during search is counted toward the per-session budget (low hundreds of dollars), not toward this per-task number. Adopting the harness for a new agent also requires wiring its tools, scaffold, and judge into the loop, so the starting baseline and engineering effort vary across codebases.

6 Related Work and Position

ALP draws on three adjacent lines of work 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 powerful 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, in the other direction, mutates whole code artifacts, but its evaluator is deterministic and scalar, an assumption that holds for algorithmic discovery and breaks for production agents whose quality is judged by an LLM under a project-specific rubric and whose 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 gap between open- and closed-source models 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.