# Grounding Agent-Driven Code Optimization in Production Telemetry

Piotr Bejda*
Datadog
New York, USA
piotr.bejda@datadoghq.com

Junaid Ahmed*
Datadog
New York, USA
junaid.ahmed@datadoghq.com

## ABSTRACT

Closed-loop LLM optimizers are starting to match or exceed hand-tuned heuristics for systems code [2, 4, 7], fitting within a broader effort to redesign data systems around agentic workloads [5]. The weak point is the benchmark: an agent minimizing a synthetic benchmark tunes for whatever distribution the benchmark exercises, whether or not that matches production. We describe DODO (Datadog Observability-Driven Optimizer), a system that closes this gap by grounding the benchmark in live production telemetry. A benchmark agent reads two signals—a CPU profile from Datadog Continuous Profiler and samples of real production calls from Datadog Live Debugger—and iterates a Go micro-benchmark until its execution shape matches production at ≥98% similarity. A simple optimization agent then uses the benchmark as a scoring function for code changes. We deploy the system against eight hot paths in a Datadog metrics-intake service and report speedups between 4% and 82%. Three of these optimizations have been validated in production, together cutting over 8% of the service's total CPU cost.

## KEYWORDS

LLM agents, code optimization, production telemetry, benchmarking, continuous profiling, Go

**ACM Reference Format:**
Piotr Bejda and Junaid Ahmed. 2026. Grounding Agent-Driven Code Optimization in Production Telemetry. In *Proceedings of Supporting Our AI Overlords Workshop at the ACM Conference on AI and Agentic Systems (SAO @ CAIS 2026).* ACM, New York, NY, USA, 4 pages.

## 1 INTRODUCTION

Using LLMs to optimize code paths that have been tuned by hand over years of production service is appealing, and recent work suggests it is increasingly practical [2, 4, 7]. The hard problem is the benchmark and the underlying deep production context: offline optimization can only generalize to production if the benchmark is representative of how the code runs in production. Writing such a benchmark by hand is difficult—and letting the agent write one from synthetic assumptions defeats the purpose. We propose to address this by grounding the agent in production telemetry.

We dynamically capture real invocations of the target function from live traffic. Probes attached to the running service sample invocations across all production instances, capturing for each the function's inputs, outputs, and execution state. Sampling across many instances yields a diverse set of real-world examples. We want variety in the kinds of inputs that impact CPU cost differently; statistical significance is not required, since the agent later fits the weight of each kind to match the production profile.

We also collect a CPU profile of the target function from production. Running across all production instances, it captures how CPU time is spent inside the target function during execution—how much in the function itself, and how much in each subroutine it calls, all the way down.

These two signals combine to define what a faithful benchmark looks like. The data captured by the debugger seeds a set of realistic test cases—concrete examples of how the function is invoked in production. The profile defines the execution shape the benchmark must reproduce when it runs those test cases: the relative time spent in each branch of the call tree. A benchmark agent draws on both—the test cases to write the benchmark's setup, and the profile as both target and feedback signal, scoring each candidate benchmark's own profile against production's. Grounding the benchmark in production telemetry along both axes—as the source of test cases and as the target execution shape—is the main contribution of the paper.

Once the benchmark exists, the optimization step is comparatively straightforward. Prior work has explored sophisticated synthesis and evolutionary loops for systems-code optimization [4, 7]; we found that a simple LLM agent—read the code (including realistic benchmark data), propose a change, run the tests, run the benchmark, repeat—produces strong results without elaborate scaffolding. This likely reflects how much progress recent LLMs have made at organizing their own work.

We implement this approach in DODO, targeting Go services, but the design is not language-specific.

## 2 RELATED WORK

DODO builds on a small but rapidly growing line of work on closed-loop LLM-driven optimization for systems code. AlphaEvolve [7] orchestrates an evolutionary pipeline of LLMs that iteratively edits algorithm implementations under feedback from automated evaluators; Cheng et al. [2] generalize the pattern as AI-driven research for systems and argue that systems problems are uniquely well-suited to it because they admit reliable, inexpensive verifiers. Closer to our setting, GenDB [4] uses an LLM agent to synthesize query execution code per incoming query, and DBPlanBench [3] exposes a query engine's physical plan for LLM-proposed edits scored by an evolutionary loop. Outside data systems, ComPilot [6] establishes the same generate-and-verify pattern with a compiler in the loop, and KernelBench [8] articulates the dense-feedback-signal framing—structured profiling output rather than scalar reward—that we also adopt in §3.1. Eudoxia [9] is the closest structural

*These authors contributed equally.

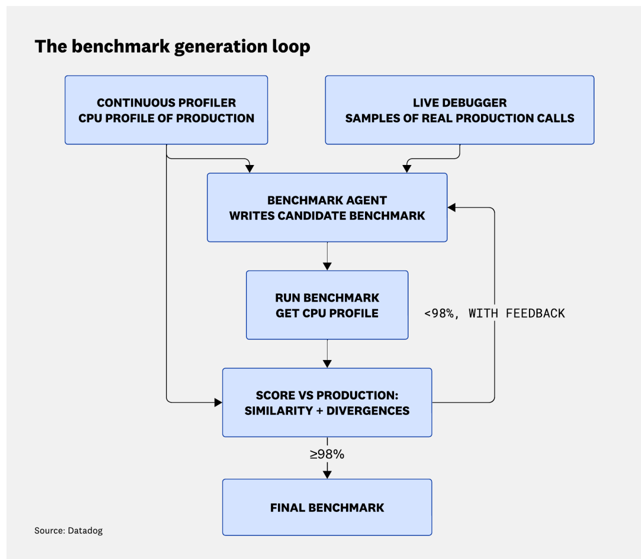


**Figure 1: The benchmark generation loop. The agent iterates until the benchmark's CPU profile matches production at ≥98% similarity.**

analog to our two-loop design: an LLM proposes a scheduling policy, a deterministic simulator verifies it on standardized traces, and divergences steer the next generation.

What unifies these systems and motivates ours is that the verifier is either synthetic (a benchmark suite, a competitive-programming testbed) or trace-replayed (a simulator over recorded inputs); DODO instead grounds the verifier in two streams of live production telemetry. Liu et al. [5] make a complementary argument that data systems themselves must adapt to agentic workloads.

Two narrower threads inform our design. On the feedback signal, differential flame graphs [1] established the idea of comparing two CPU profiles structurally to localize regressions, which we adapt as the per-path divergence list returned by `evaluate_benchmark`. On the agent–tool interface, SWE-agent [10] argues that agent-computer interface design is load-bearing for LM agent performance, which we corroborate in §3.1.1.

## 3 ARCHITECTURE

DODO consists of two loops: benchmark generation and code optimization.

### 3.1 Loop 1—production-grounded benchmark generation

The benchmark agent runs an iterative loop, sketched in Figure 1: read the two production signals, write a candidate benchmark, run it, score its CPU profile against production's, and use the divergences as feedback for the next iteration. The loop terminates when the similarity threshold is met or the turn budget (the maximum number of agent iterations allowed) is exhausted.

The benchmark agent is given a target function, a benchmark file path, and two streams of production signal:

- **A pruned CPU call tree rooted at the target function.** We fetch an aggregated flamegraph from the Datadog profiling API for the target service, filtered by the CPU architecture, walk to the target function, and prune subtrees contributing less than 1%. This tree is included in the system prompt and used as ground truth by the evaluation tool.
- **Real invocations from live traffic.** The agent places Live Debugger probes to capture production invocations of the target, including arguments and receiver state.

Capturing the receiver state is as important as capturing the arguments. Many hot-path targets carry elaborate internal configurations—rule sets, pre-populated caches, lookup tables—whose construction from setup code is time-consuming and brittle to reproduce. Direct state capture lets the benchmark agent assemble semantically valid test cases in the first place, and inherit realistic call characteristics as a side effect.

Profile similarity is only meaningful if both sides are measured on the same hardware. Two normalizations help: architecture-specific function names are aliased to a canonical form, and internal frames within Go runtime functions are collapsed to their top frame—enough to identify the operation without comparing implementation depths that differ by architecture. Raw CPU costs, however, cannot be normalized away: an `amd64` hash instruction and its `arm64` equivalent have different latencies, and no amount of input tuning by the agent can close that gap. We sidestep the problem by fetching the production profile with a CPU-architecture filter and running the benchmark on matching hardware. Removing this hardware-parity requirement—normalizing away per-instruction cost differences, or measuring at simulated parity—remains an open problem.

Given these inputs, the agent writes a single Go benchmark function. The `evaluate_benchmark` tool compiles it, runs it three times with CPU profiling, parses the resulting profile, prunes it the same way as the production tree, and computes a similarity score. The score is a convex combination of (a) L2 similarity on the self-plus-children weight vector at each node and (b) a weighted recursive similarity on shared children. The tool returns the score plus the top divergences—labeled *missing*, *extra*, *over*, or *under*—as absolute percentages of total profile time, so the agent can see exactly which call paths its benchmark over- or under-exercises.

*3.1.1 Shaping agent behavior.* A few engineering choices around prompts and tools matter as much as the production signals themselves. Tools should be scoped to a complete unit of work, exposed as single high-level operations rather than their constituent steps—capturing production calls, for example, involves placing a probe, waiting, retrieving the data, and tearing it down, but the agent calls a single `capture_production_data` tool that handles all four. An earlier version that surfaced each step separately consumed 5–10 agent turns on bookkeeping. Context the agent will need but does not choose belongs in the prompt rather than in the tool surface—information known up front (the target function's calling convention, the production deployment we instrument, the target's location in the call tree) is injected into the system prompt rather than fetched on demand, eliminating a class of "agent goes looking for things" failure modes. Prompts describe affordances rather than procedures: telling the agent what it can do, not what to do when.

The agent constructs an optimal hill-climbing path by exploring the surface through progressive planning, improving in each step. More prescriptive prompts caused agents to treat the procedure as load-bearing and get stuck when a step did not apply. Similarly, we do not impose explicit planning scaffolding—no separate planning phase, scratchpad, or task-list decomposition. Modern LLMs handle implicit planning well enough that adding such scaffolding tends to constrain rather than help.

*3.1.2 Benchmark similarity score.* Both profile trees—production's and the candidate benchmark's—are pre-processed identically: pruned to drop branches contributing less than 1% of the target function's total CPU, with architecture-specific function names aliased and Go runtime internals collapsed to their top frame.

At each tree node, we partition CPU time across the node's direct children plus a self bucket, forming a weight vector with one entry per name. The L2 distance between the production and benchmark vectors—normalized to $[0, 1]$ by dividing through $\sqrt{2}$, the maximum distance between two probability distributions—gives a local similarity. For child names that appear in both trees, we recurse and average those subtree similarities, weighted by $\max(\mathit{prod_share}, \mathit{bench_share})$ so high-impact children dominate. The score at a node is:

$$0.6 \times \text{L2_similarity} + 0.4 \times \text{recursive_child_similarity}.$$

The root's score is what the agent sees, alongside a sorted list of *divergences*: each call path where the absolute share of total CPU differs by ≥1% between production and benchmark, labeled *missing* (production-only), *extra* (benchmark-only), *over*, or *under*.

## 3.2 Loop 2—optimization against the generated benchmark

The optimization agent receives the frozen benchmark from Loop 1 and standard code-reading and editing tools, plus `run_tests` and `run_benchmark`. Before the loop starts, the tests are run three times to fingerprint pre-existing flakes, and the benchmark is run once to establish a baseline `ns/op` and CPU profile; both are surfaced in the system prompt.

Each `run_benchmark` call re-runs the same command as the baseline, compares `ns/op` against the baseline, and snapshots the current code change as a numbered patch. The snapshot with the lowest `ns/op` so far is retained as the best observed state, so a regressing final edit does not overwrite earlier gains.

## 3.3 Keeping the two loops decoupled

*Disjoint write access.* The benchmark agent only writes the benchmark file; the optimization agent only writes service code and reads the benchmark. The benchmark is fixed before the optimizer starts, so it cannot drift under optimization.

*Feedback is dense, not scalar.* `evaluate_benchmark` returns the similarity score and the top divergences as a sorted list of `(path, prod%, bench%, gap)` tuples. In practice, this is what lets the agent close gaps in one or two iterations: "callee X is 12% in production and 2% here—I need more inputs that trigger the X branch."

**Table 1: Per-target speedup and the optimization the agent found. "Sim." is the benchmark's CPU profile similarity to production (see §3.1). "Prod" marks optimizations validated in production.**

| Target | CPU% | Sim. | Speedup | Prod |
|---|---|---|---|---|
| `intern` | 9.8% | 97% | 40% | ✓ |
| `MergeTags` | 11.5% | 98% | 4% | |
| `NormalizeTags` | 7.5% | 97% | 22% | |
| `HandleFromSortedTags` | 3.7% | 98% | 15% | |
| `ComputeTagsHash` | 3.2% | 89% | 27% | |
| `FilterPayloads` | 2.7% | 98% | 75% | ✓ |
| `writeTagsetsMut` | 2.0% | 94% | 76% | ✓ |
| `filterTags` | 1.1% | 98% | 82% | |

## 3.4 Implementation

We implement DODO on top of two Datadog platform capabilities. Datadog Continuous Profiler provides always-on, low-overhead CPU profiling across all production instances, queryable via API with per-architecture filters—this supplies the production call tree. Datadog Live Debugger dynamically attaches probes to running production code without redeployment, capturing function arguments and receiver state across instances—this supplies the real invocations. Both are generally available Datadog products; the methodology itself requires only a continuous profiler and a dynamic instrumentation system with equivalent capabilities.

# 4 RESULTS

We evaluated DODO on a mature Datadog metrics-intake service written in Go. The service ingests and processes the metric data Datadog receives from customer agents. After years in production at scale and many rounds of performance tuning by experienced engineers, the hot paths in this service are not new code—they had already been examined and optimized by humans before DODO ran against them.

We deployed DODO against eight such hot paths, running on `arm64` (Graviton). The targets range from 1.1% to 11.5% of service CPU. Table 1 summarizes the speedups measured against the generated benchmarks.

Two vignettes illustrate the kind of change the optimizer produced.

*FilterPayloads.* The `(*StringerFilter)` receiver carries an elaborate production configuration of filter rules—a mix of literal, prefix, and regex matchers—that the benchmark agent captured directly from live traffic rather than reconstructing from the service's dense setup code. The baseline profile showed 86.5% of time in `Filters.Matches`, a linear scan through all filter rules. The agent observed that most rules are exact-string matches, introduced an `IndexedFilters` type backed by a map for O(1) literal lookup, and kept a linear pass only for regex and prefix matchers. The new type was threaded through four files across two packages.

*writeTagsetsMut.* The baseline showed 59% of time in `sort.Sort` (`pdqsort` on `uint32` tag IDs). The agent observed that tag IDs are bounded to a small integer range and replaced the sort with a bitset: set a bit for each ID, then iterate via `TrailingZeros64`. O(n) instead

of O(n log n), plus inlined streamvbyte encoding with direct indexed writes.

The role of production grounding is visible in `NormalizeTags`: captured invocations revealed that roughly 25% of tags contain uppercase characters (timestamps containing `T`, version strings containing `RC`). The benchmark agent preserved that ratio; the optimization agent then found a fast ASCII case-fold path whose payoff depends on exactly that distribution. A synthetic benchmark would not have surfaced the opportunity.

## 5 FUTURE WORK

The benchmark generation loop described here is one instance of a broader opportunity: grounding agentic developer workflows in production telemetry across the software lifecycle. Pre-release, the same signals—profiler data and captured invocations—could seed unit and acceptance tests with production-realistic inputs, or validate changes against production execution patterns before merge. Post-release, trace aggregates and metrics anomalies could drive automated hardening of services, while session-level analysis could close the loop between deployment and business outcomes. For LLM applications specifically, production prompts could serve as inputs for local evaluation and experimental validation.

## ACKNOWLEDGMENTS

This work would not have been possible without the endless support from the Debugger, Profiling, and Metrics Intake teams. In particular, Andrew Werner and Andrei Matei have closely worked on making this approach feasible.

## REFERENCES

[1] Cor-Paul Bezemer, Johan Pouwelse, and Brendan Gregg. 2015. Understanding Software Performance Regressions Using Differential Flame Graphs. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 535–539.

[2] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. *arXiv preprint arXiv:2510.06189* (2025). https://arxiv.org/abs/2510.06189

[3] M. H. Erol, X. Hao, F. Bianchi, C. Greco, J. Tagliabue, and J. Zou. 2026. Making Databases Faster with LLM Evolutionary Sampling. *arXiv preprint arXiv:2602.10387* (2026). https://arxiv.org/abs/2602.10387

[4] Jiale Lao and Immanuel Trummer. 2026. GenDB: The Next Generation of Query Processing—Synthesized, Not Engineered. *arXiv preprint arXiv:2603.02081* (2026). https://arxiv.org/abs/2603.02081

[5] Shu Liu, Soujanya Ponnapalli, Shreya Shankar, Sepanta Zeighami, Alan Zhu, Shubham Agarwal, Ruiqi Chen, Samion Suwito, Shuo Yuan, Ion Stoica, Matei Zaharia, Alvin Cheung, Natacha Crooks, Joseph E. Gonzalez, and Aditya G. Parameswaran. 2025. Supporting Our AI Overlords: Redesigning Data Systems to Be Agent-First. *arXiv preprint arXiv:2509.00997* (2025). https://arxiv.org/abs/2509.00997

[6] M. Merouani, I. Kara Bernou, and R. Baghdadi. 2025. Agentic Auto-Scheduling: An Experimental Study of LLM-Guided Loop Optimization. *arXiv preprint arXiv:2511.00592* (2025). https://arxiv.org/abs/2511.00592

[7] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery. *arXiv preprint arXiv:2506.13131* (2025). https://arxiv.org/abs/2506.13131

[8] A. Ouyang, S. Guo, S. Arora, A. L. Zhang, W. Hu, C. Ré, and A. Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels? *arXiv preprint arXiv:2502.10517* (2025). https://arxiv.org/abs/2502.10517

[9] J. Tagliabue. 2025. AI for Distributed Systems Design: Scalable Cloud Optimization Through Repeated LLMs Sampling and Simulators. *arXiv preprint arXiv:2510.18897* (2025). https://arxiv.org/abs/2510.18897

[10] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, Vol. 37.