Argus: Agentic, Reference-Calibrated, Tree-Guided, System-Level Bottleneck Localization

Architecture 2.0 Workshop @ ISCA 2026 – Submission #XXX – Confidential Draft

Vlad-Petru Nitu Harsh Songara Konstantinos Sgouras
Spiros Galanopoulos Konstantinos Kanellopoulos Onur Mutlu
ETH Zürich

Many performance bottlenecks in modern systems reside inside the operating system (OS), a complex codebase spanning memory management, scheduling, and networking subsystems, which makes it hard to localize the specific kernel code path responsible for a slowdown. This problem is only getting worse as microsecond-scale I/O devices, networking, complex memory hierarchies, and hardware accelerators continue to outpace traditional kernel mechanisms.

Existing profilers either expose raw counters (e.g., perf), stop at the subsystem boundary (e.g., Intel VTune), or impose overheads that distort the very microsecond-scale timings they aim to measure (e.g., ftrace). As a result, OS bottleneck analysis remains a tedious, error-prone process of manually inspecting kernel source code.

We introduce **Argus**, a system that leverages LLM-based agents to author instrumentation code and reason over its outputs in a tight feedback loop. Argus *grounds* the agent via two complementary mechanisms: (i) *reference calibration* against a baseline profile collected on an idle system, and (ii) *tree-guided kernel traversal* over a static map of kernel code paths. Unlike prior profilers, Argus traces into the specific kernel code path that creates the bottleneck, turning a vague symptom (e.g., *"the workload is slow"*) into a precise diagnosis: page-fault contention on the Transparent Huge Page (THP) allocation path.

We demonstrate Argus's flexibility and effectiveness on two case studies. *First*, under a co-running THP aggressor against a diverse victim pool, Argus produces per-victim diagnoses. *Second*, five page-fault perturbers each exercise a different combination of kernel code paths; Argus resolves them into four code-path signatures, recovering the kernel's page-fault taxonomy without prior knowledge of which paths each perturber exercises.

Argus achieves $16\times$ more accurate diagnoses than the best-performing baseline we evaluate, and its $30.9\,\mathrm{s}$ median time-to-diagnosis is on par with three LLM-prior baselines.

1 Introduction

Performance bottlenecks impose substantial costs on modern computing infrastructure, since they translate into significant resource waste, energy overhead, and degraded user experience at datacenter scale [1]. In latency-sensitive environments, microsecond-scale jitter alone can violate Service Level Objectives (SLOs), forcing operators to overprovision hardware capacity by 2–10× [2].

Diagnosing such bottlenecks is hard because the OS is a complex codebase that hosts a diverse set of bottlenecks (e.g., 5% of cycles spent in the scheduling subsystem and 5% in the memory-management subsystem [1]). Modern Linux kernels span dozens of subsystems (e.g., memory management, scheduling), each composed of many fine-grained code paths. A useful bottleneck diagnosis must therefore reach the specific code path, rather than stop at the kernel subsystem level.

Reaching the responsible code path is inherently *a closed-loop process*: the engineer narrows the search from subsystem-level signals down to finer-grained code paths, until a single path explains the slowdown. Unfortunately, conventional profiling workflows remain manual: an engineer picks a profiler, deploys it, obtains raw output, and reasons about the results to hypothesize where the bottleneck resides. This is an expert-only, profiler-dependent process, with no single tool that localizes the specific kernel code path responsible for the bottleneck.

What existing profilers miss. Production-grade profilers each cover only part of the stack. Timer samplers [3] reveal *where* CPU time is spent but not *why*. Hardware-event samplers [3], [4], [5], [6] miss OS contention (e.g., lock waits, page faults). Profilers built on static tracepoints [3], [7], [8] are limited to pre-instrumented kernel hooks (e.g., `sched_switch`) and incur high overhead. Dynamic-instrumentation toolkits [9], [10] expose arbitrary probes but require expert scripting. Continuous-sampling profilers [11], [12], [13] run fleet-wide at low overhead, yet attribute OS contention only at kernel subsystem granularity. Lastly, GPU profilers [14], [15] characterise device-side activity in detail but treat the host kernel as a black box. Among production-grade profilers, only programmable kernel tracing via eBPF [16], [17], [18] can attach low-overhead probes to arbitrary kernel code paths. eBPF is backed by a verifier [19] that statically checks each probe for correctness (e.g., termination, memory safety) before loading it into the kernel, enabling safe programmable kernel tracing. Once loaded, eBPF probes perform low-overhead in-kernel aggregation on arbitrary events. We quantify this overhead in §3.

However, two important gaps remain. *First*, writing eBPF probes still requires expert knowledge of kernel internals, limitations of the eBPF verifier, and per-kernel-version symbol drift (e.g., inconsistent function naming across kernel versions [20]).

This *authoring gap* makes the benefits of eBPF inaccessible to less experienced eBPF developers. *Second*, even when a probe passes the verifier checks, eBPF outputs a stream of raw counters that need manual reasoning and mapping to a root cause. This *interpretation gap* complicates bottleneck identification, even for experts.

LLM-based agents create a new opportunity. Recent agents [21], [22] can *author* instrumentation code and *reason* over its outputs in a tight feedback loop, creating an opportunity to close both the authoring and interpretation gaps. However, naive use of agents for eBPF-based profiling faces three limitations. *First*, an agent needs a reference value to determine whether an observed measurement signals a bottleneck. *Second*, such reference values are noisy and vary across hardware configurations. *Third*, without a predefined map of the kernel, the agent hallucinates code paths, inventing function names or misplacing them under the wrong subsystem. These limitations motivate grounding the agent with both measured reference metrics and a structured view of kernel code paths.

Our goal in this work is to develop a profiling methodology that retains the strengths of eBPF (i.e., programmable, low-overhead, and runtime-attachable instrumentation) while closing the authoring and interpretation gaps without expert intervention. To this end, we present **Argus**, an agentic eBPF-based profiler that *grounds* an LLM agent via two complementary mechanisms: *reference calibration* against an idle-system profile, and *tree-guided kernel traversal* over a static tree of kernel code paths. Argus localizes the bottleneck in three deterministic steps (e.g., minor anonymous-page faults).

To isolate the contribution of each component, we define three baselines. (i) *B-LLM-Prior*: the agent receives no reference profile and descends based purely on its LLM prior. This is the weakest baseline we evaluate, and isolates the contribution of LLM intuition alone. (ii) *B-LLM-Prior-Tree*: *B-LLM-Prior* augmented with the static Linux-kernel code-path tree in the agent's context, isolating the contribution of tree-guided traversal in reducing hallucinations. (iii) *B-LLM-Prior-Tree-Probes*: *B-LLM-Prior-Tree* augmented with agent-authored eBPF probes at every tree level, which guide the agent's descent via measured counters rather than LLM intuition, but still without reference calibration. This baseline isolates the contribution of reference calibration over measurement alone.

We empirically motivate Argus's design through an ablation across the three baselines defined above.

We construct a (victim, perturber) matrix that crosses five representative workloads with four perturbations (e.g., THP disable, page-cache drop), and run each baseline three times per cell. We define a **hallucination** as a leaf-level commit that does not match the cell's actual bottleneck. Hallucinations are either **wrong attributions** (real kernel functions the workload does not exercise) or **fabrications** (function names absent from the kernel, e.g., `alloc_cold`). Fig. 1 reports the number of hallucinated diagnoses per (victim, perturber) cell for each baseline. We make four key observations. *First*, B-LLM-Prior hallucinates even on simple microbenchmarks because its prior is based on the perturbation name alone, not on the victim's

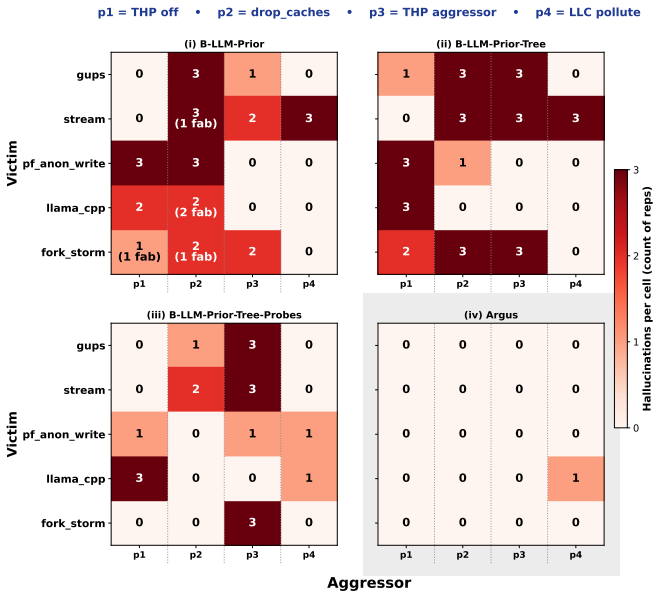

Fig. 1: Per-cell hallucinated diagnoses across (victim, perturber) pairs, for Argus and three LLM-prior baselines.

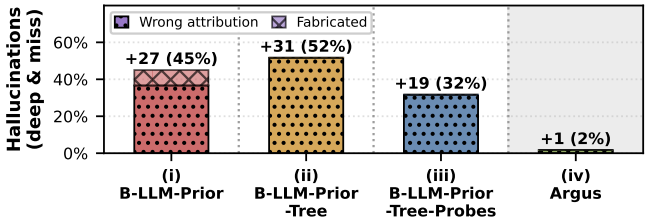

Fig. 2: Average hallucination rate per baseline across all (victim, perturber) pairs. Argus hallucinates 16–26× less than the LLM-prior baselines.

actual access pattern. *Second*, B-LLM-Prior sometimes *fabricates* kernel function names (e.g., `alloc_cold`, `page_alloc`). *Third*, without a reference, B-LLM-Prior has no notion of *small deviation* and cannot recognize null cells (i.e., no bottleneck), sometimes committing deep with little confidence. In contrast, Argus correctly abstains when the perturbation has no measurable effect. *Fourth*, providing the kernel code-path tree (B-LLM-Prior-Tree, B-LLM-Prior-Tree-Probes) removes fabricated hallucinations entirely, since the tree restricts the agent's generation to existing kernel functions. However, B-LLM-Prior-Tree still descends deep without enough evidence, sometimes hallucinating more often than B-LLM-Prior. B-LLM-Prior-Tree-Probes sits between Argus and the prior-only baselines, but still over-commits without a reference. We conclude that adding tree descent without a reference profile is insufficient, motivating the **synergistic interplay of reference calibration and tree guidance**.

Fig. 2 aggregates these results across all (victim, perturber) pairs. We make two key observations. *First*, Argus hallucinates 2% of the time, compared to 45% for B-LLM-Prior and 52% for B-LLM-Prior-Tree (23× and 26× more, respectively). B-LLM-Prior-Tree-Probes reduces hallucinations to 32%, sitting

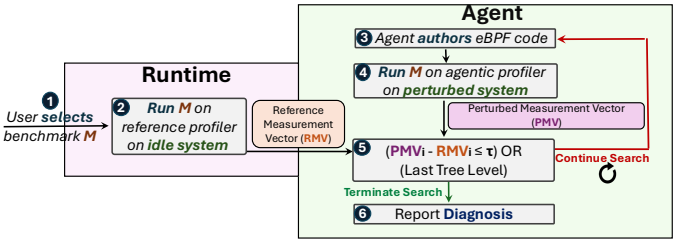

Fig. 3: Overview of Argus's Architecture.

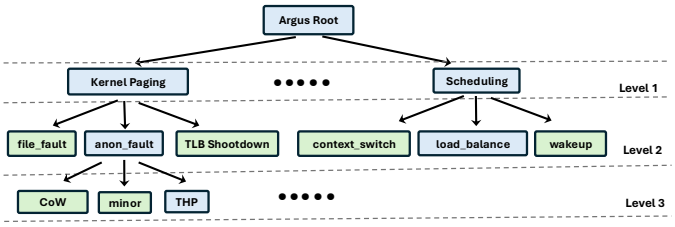

Fig. 4: Argus's static kernel-subsystem tree.

between Argus and the prior-only baselines. Compared to this best-performing baseline, Argus is 16× more accurate. *Second*, while B-LLM-Prior terminates exploration when uncertain, B-LLM-Prior-Tree follows the tree deeper into the codebase even without enough evidence, incorrectly guessing which kernel code path contains the bottleneck. We conclude that the LLM is prone to hallucination when it has no access to the ground truth.

Contributions. We make the following contributions: (i) we propose Argus, the first agentic, autonomous OS-level profiler that grounds an LLM agent through reference calibration and tree-guided traversal; (ii) we present an ablation across three baselines, *isolating* reference calibration as the missing ingredient (16× fewer hallucinations than the best-performing baseline); and (iii) we demonstrate Argus on two case studies: per-victim diagnoses under a co-running THP aggressor, and disambiguating contention within the paging module of the memory management subsystem.

2 Argus: Key Idea, Overview, and Mechanism

We propose Argus, the first agentic profiler that autonomously diagnoses system-level performance bottlenecks down to the specific OS code path that creates them. The **key idea** of Argus is to *ground* the LLM agent via two complementary mechanisms: (i) *reference calibration*, which compares the agent's measurements against a baseline profile of the workload collected on an idle reference system, and (ii) *tree-guided kernel traversal*, which constrains the agent's exploration to a static, hierarchical tree of kernel code paths built offline from kernel source. By doing so, Argus falsifies bottleneck hypotheses under perturbation while avoiding hallucinated, non-existent code paths, enabling autonomous bottleneck localization at code-path granularity without expert intervention.

Fig. 3 illustrates Argus's high-level components. The reference and perturbed measurement vectors (RMV/PMV) each consist of a Feature Vector at level 1 (IFV/OFV) and Code Path Vectors at deeper levels (ICPV-LL/CPV-LL). Argus's pipeline operates in six steps. ❶ The user selects a workload M to profile. ❷ The reference profiler runs M on an idle system, yielding the RMV. ❸ The agent authors a single *multiplexed* eBPF probe (i.e., one program that hooks every branch of the tree at the current level L). ❹ The agentic profiler runs M on the perturbed system with the loaded probe, yielding the PMV. ❺ Argus compares the PMV against the RMV to decide whether to terminate the search or descend one level deeper into the flagged subtree, in which case it returns to step ❸. ❻ Upon termination, Argus reports the final diagnosis: either the flagged code path or *no bottleneck found*.

Hierarchical subsystem tree. Fig 4 illustrates the kernel-subsystems static tree. Level 1 enumerates the top-level kernel subsystems (e.g., paging, scheduling). Level 2 breaks each subsystem into its sub-code paths (e.g., paging → {`anon_fault`, `file_fault`, TLB shootdown}). Level 3 splits each path into fine-grained variants (e.g., `anon_fault` → {`minor`, `cow`, THP collapse}). As a result, Argus localizes a slowdown in at most three deterministic steps, issuing one multiplexed probe per tree level, whereas a flat search over the same code paths would require N probes, where N is the total number of leaves in the tree.

Reference profiling ❷. Argus produces the RMV by profiling M on an idle reference system, which serves as the baseline against which the agent later detects deviations under perturbation. At level 1, Argus runs M under `perf` [3] to collect coarse-grained performance counters (e.g., number of LLC misses), which populate the IFV. At levels 2 and 3, Argus runs one multiplexed eBPF probe [23] per tree level to collect per-code-path and per-call latency counters, which populate ICPV-L2 and ICPV-L3, respectively.

Agent profiling closed loop ❸-❹. At each level L, the agent performs three steps to produce the PMV. (*i*) *Generate*: the agent generates eBPF source code that hooks the level-L code paths in the active subtree (i.e., the subtree rooted at the path flagged at level $L-1$, or the entire tree at $L=1$). At $L=2$ and $L=3$, the agent multiplexes all hooks into a single eBPF program that shares counters through one eBPF map [23], such that one program covers the entire subtree. (*ii*) *Verify and load*: the agent submits the generated program to the eBPF verifier. If the verifier accepts the program, Argus loads it into the kernel. Otherwise, the agent re-prompts itself with the verifier's diagnostic message in a ReAct [24] loop, and repeats step (*i*) until the program is accepted. (*iii*) *Measure*: Argus runs M on the perturbed system (i.e., the system with the loaded eBPF probe), and collects the resulting counters into the OFV at $L=1$ or CPV-LL at $L=2$ and $L=3$.

Bottleneck detection and diagnosis reporting ❺-❻. For each dimension i, Argus computes a z-score $\Delta_i = (\mathrm{PMV}_i - \mathrm{RMV}_i)$ and flags dimensions with $\Delta_i > \tau$ (the *confidence condition*; we set $\tau = 2$ empirically). If no dimension is flagged, Argus reports *no bottleneck found*. If at least one dimension is flagged and L is a leaf, Argus reports the flagged path as the bottleneck (e.g., `kernel_paging` → `anon_fault` → `minor`). If at least one dimension is flagged but L is not a leaf, Argus descends one level into the flagged child subtree and returns to step ❸.

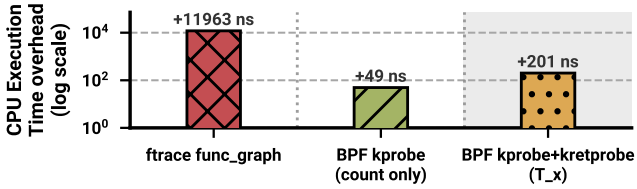

Fig. 5: CPU execution time overhead (log scale) of three kernel-tracing options over an uninstrumented baseline: `ftrace`'s `func_graph`, an eBPF `kprobe` that only counts events, and an eBPF `kprobe+kretprobe` that captures per-call latency T_x (the configuration Argus uses).

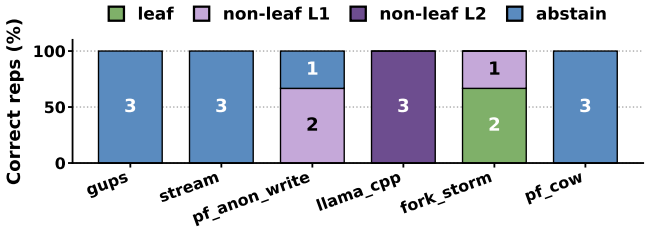

Fig. 6: Aggressor mode with M = *p10_thp_aggressor*. Argus localizes the aggressor's exposure per victim.

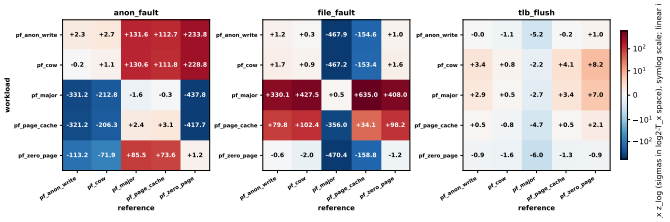

Fig. 7: Page-fault disambiguation: L2 multiplexed T_x signatures of five page-fault perturbers (rows) against five *idle* reference profiles (columns), across three kernel code paths: `anon_fault`, `file_fault`, and `tlb_flush`.

3 Evaluation

Overhead analysis. Fig. 5 quantifies the CPU execution time overhead per kernel function call on a fault-heavy microbenchmark (`pf_anon_write` × `p1`, which sustains 2.6 M `do_anonymous_page` calls over 3 s). We make three key observations. *First*, `ftrace`'s [7] `func_graph` tracer emits a full entry+exit event into the kernel ring buffer for every kernel function the workload executes, incurring 11.9 μs per call. *Second*, a count-only eBPF `kprobe` reduces this overhead by 243×, to 49 ns per call, but loses per-call latency information. *Third*, an eBPF `kprobe+kretprobe` pair that captures per-call entry/exit latency T_x in-kernel incurs 201 ns per call, at only 4× more than the count-only probe, but 60× less than `ftrace`. We conclude that the `kprobe+kretprobe` configuration strikes the best balance between low overhead and diagnostic signal: at a fraction of `ftrace`'s cost, T_x provides the agent with the per-call latency it needs to attribute slowdowns to specific kernel code paths. Argus therefore adopts this configuration.

Time-to-Diagnosis. We evaluate the median Time-to-Diagnosis (TtD) of Argus and the three baselines defined in §1, on the same (victim, perturber) matrix used in the hallucination study (Fig. 1): victims {`gups`, `stream`, `pf_anon_write`, `llama_cpp`, `fork_storm`} × perturbers {`p1`, `p2`, `p3`, `p4`}, with 3 repetitions per cell. We make two key observations. *First*, Argus incurs no measurable TtD overhead over the LLM-prior baselines: its 30.9 s median is on par with all three (B-LLM-Prior: 33.4 s, B-LLM-Prior-Tree: 26.3 s, B-LLM-Prior-Tree-Probes: 28.7 s), despite the additional reference-profiling and z-score-comparison steps that Argus performs. *Second*, B-LLM-Prior is 17% slower than the average of the other three configurations, because the agent thrashes between subsystem hypotheses without a tree to guide its descent. We conclude that reference calibration is *cheap* in terms of TtD, while delivering the hallucination reduction reported in §1.

4 Case Studies

We demonstrate Argus on two case studies: (1) per-victim diagnosis under a THP aggressor, and (2) disambiguating which code paths different page-fault types stress within the paging module of the memory management subsystem.

Case Study 1: THP aggressor. Fig. 6 reports aggressor-mode results with $M =$ `p10_thp_aggressor`, which fragments physical memory and then forces compaction by demanding huge pages on the fragmented region. Argus produces five distinct diagnoses across the victim pool. *First*, **gups** and **stream** (3/3 abstain) pre-allocate in userspace, so the aggressor never reaches their hot path. Thus, Argus concludes that there is not bottleneck. *Second*, **pf_anon_write** (3/3 correct) commits to `kernel_paging` at L1 in 2/3 reps. *Third*, **llama_cpp** (3/3 correct) exposes a non-obvious kernel-paging signal: its 4.6 GB GGUF model is `mmap`'d file-backed, so the aggressor's memory pressure evicts page-cache pages (`file_fault`) while KV-cache growth drives anonymous faults (`anon_fault`); Argus commits to `kernel_paging` at L2. *Fourth*, **fork_storm** (2/3 leaf-level) acquires `mmap_lock` in every `fork()`; Argus commits to `kernel_scheduler.load_balance` at leaf level. *Fifth*, **pf_cow** (3/3 abstain) exposes a tree limitation: its THP-tax spreads across `mmap_lock`, slab, and RCU paths that the current tree does not enumerate. We conclude that aggressor mode distinguishes victims whose kernel-traceable paths are exercised from those whose are not, surfacing failures as tree limitations rather than methodological errors.

Case Study 2: Page-fault disambiguation. Fig. 7 reports L2 T_x z-scores of five page-fault perturbers against five idle references, across tree kernel code paths: `anon_fault`, `file_fault`, and `tlb_flush`. Three clusters emerge. *First*, the `anon_fault` cluster (`pf_anon_write`, `pf_cow`, `pf_zero_page`) lights up against file-backed references, while symmetric cells are deeply depressed. *Second*, `pf_major` dominates `pf_page_cache` in `file_fault` by ∼5×, separating "file-backed fault" from "file-backed fault *with disk I/O*." *Third*, the `tlb_flush` cluster (`pf_cow`, `pf_major`) is smaller in magnitude, since TLB invalidations are *secondary* events on COW and swap-in remapping. We conclude that the five perturbers resolve into four signature classes: (1) *anon-only* (`pf_anon_write`, `pf_zero_page`); (2) *anon+TLB* (`pf_cow`); (3) *file-backed-light*

(pf_page_cache); and (4) *file-backed-with-IO* (pf_major). Argus's L2 multiplex recovers the kernel's page-fault taxonomy, without prior knowledge about the perturbers' behavior.

References

[1] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, USA: ACM, 2015, pp. 158–169.

[2] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, p. 74–80, Feb. 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408794

[3] Linux Kernel, "perf: Linux profiling with performance counters," https://perfwiki.github.io/main/, 2024, accessed: 2026-04-24.

[4] Intel Corporation, "Intel vtune profiler," https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html, 2024, accessed: 2026-04-24.

[5] Advanced Micro Devices, "Amd uprof profiler," https://www.amd.com/en/developer/uprof.html, 2024, accessed: 2026-04-24.

[6] A. Yasin, "A Top-Down method for performance analysis and counters architecture," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Monterey, CA, USA: IEEE, 2014, pp. 35–44.

[7] S. Rostedt, "ftrace: Function tracer for the linux kernel," https://www.kernel.org/doc/html/latest/trace/ftrace.html, 2009, accessed: 2026-04-24.

[8] M. Desnoyers and M. R. Dagenais, "The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux," in *Ottawa Linux Symposium (OLS)*. Ottawa, Canada: Linux Symposium, 2006, pp. 209–224.

[9] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX Annual Technical Conference (ATC)*. Boston, MA, USA: USENIX Association, 2004, pp. 15–28.

[10] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and B. Chen, "Locating system problems using dynamic instrumentation," in *Ottawa Linux Symposium (OLS)*. Ottawa, Canada: Linux Symposium, 2005, pp. 49–64.

[11] Grafana Labs, "Grafana pyroscope: Continuous profiling," https://grafana.com/oss/pyroscope/, 2024, accessed: 2026-04-24.

[12] Polar Signals, "Parca: Continuous profiling for analysis of cpu and memory usage," https://www.parca.dev/, 2024, accessed: 2026-04-24.

[13] Datadog, "Datadog continuous profiler," https://docs.datadoghq.com/profiler/, 2024, accessed: 2026-04-24.

[14] NVIDIA Corporation, "NVIDIA Nsight Systems," https://developer.nvidia.com/nsight-systems, 2024, accessed: 2026-04-25.

[15] Advanced Micro Devices, "AMD Omniperf: GPU kernel performance profiler," https://github.com/ROCm/omniperf, 2024, accessed: 2026-04-25.

[16] IOVisor Project, "BCC: BPF compiler collection," https://github.com/iovisor/bcc, 2024, accessed: 2026-04-24.

[17] ——, "bpftrace: High-level tracing language for Linux," https://github.com/bpftrace/bpftrace, 2024, accessed: 2026-04-24.

[18] eunomia-bpf Project, "eunomia-bpf: A dynamic loading library and compile toolchain for eBPF," https://github.com/eunomia-bpf/eunomia-bpf, 2024, accessed: 2026-04-24.

[19] Linux Kernel Contributors, "eBPF verifier — Linux kernel documentation," https://docs.kernel.org/bpf/verifier.html, 2024, accessed: 2026-05-07.

[20] M. Wilcox, "mm/page_alloc: combine __alloc_pages and __alloc_pages_nodemask," Linux kernel commit 84172f4bb752, Apr. 2021, https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=84172f4bb752424415756351a40f8da5714e1554.

[21] Anomaly, "OpenCode: The open source AI coding agent," https://opencode.ai, 2026, source repository: https://github.com/anomalyco/opencode. Accessed: 2026-05-07.

[22] A. Novikov, N. Vu, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog, "AlphaEvolve: A coding agent for scientific and algorithmic discovery," Google DeepMind White Paper. https://arxiv.org/abs/2506.13131, 2025, arXiv:2506.13131.

[23] ebpf.io Authors, "eBPF maps," https://docs.ebpf.io/linux/concepts/maps/, 2024, accessed: 2026-05-07.

[24] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," in *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023, arXiv:2210.03629.