Fidelio: Agentic AI Framework for Synthesizing Microarchitecture-Faithful Datacenter Benchmarks

Alan Andrade[†], Petar Acimovic[‡], Wei Su[*], Suyash Mahar[*],
Apostolos Kokolis[*], Abhishek Dhanotia[*], Jovan Stojkovic[†]
[†]The University of Texas at Austin [*]Meta [‡]Georgia Institute of Technology

***Abstract*—Datacenter workloads drive modern processor design, yet representative benchmarks remain scarce. Production services cannot be shared due to confidentiality, and manually engineered proxies are expensive to build, difficult to maintain, and quickly drift out of date. On the other hand, prior automated approaches, e.g., statistical cloning, target a single hardware configuration, offering no guarantee that the resulting benchmark will respond to microarchitectural design changes in the same way as the original workload.**

We present Fidelio, an agentic AI framework that automatically synthesizes portable benchmark clones from lightweight hardware telemetry. Fidelio orchestrates a closed-loop, multi-agent workflow in two phases. First, agents iteratively generate, measure, and refine candidate programs on real hardware to match a tiered, weighted set of performance counters. Next, agents sweep key design knobs (e.g., prefetcher, branch predictor, replacement policy) through architectural simulations for these candidates and repair any divergence in performance-sensitivity trends relative to the target workload. Microarchitecture-aware diagnostic playbooks guide each transformation, ensuring that fixes for one objective do not regress another.

We evaluate Fidelio on open-source datacenter benchmarks from the DCPerf suite and production workloads at a hyperscaler. Across both sets, Fidelio produces clones that faithfully reproduce hardware-counter profiles and preserve sensitivity to microarchitectural design changes. This enables credible, privacy-preserving design-space exploration for realistic datacenter workloads.

***Index Terms*—datacenter services, benchmarking, agentic AI.**

I. Introduction

Motivation. Datacenter computing has become the primary driver of server design. Cloud and hyperscale operators account for roughly half of total server revenue [6], [63] and they continuously refresh their fleets to serve search [7], social media [56], recommendations [25], storage [65], analytics [21], and increasingly AI-driven services [60]. The performance, efficiency, and cost of datacenter workloads directly shape processor roadmaps and motivate many of the industry's largest capital investments [11].

However, the workloads that dominate datacenters differ fundamentally from the benchmarks that have historically guided CPU design. For instance, prior characterization studies from hyperscalers such as Google [8], [33] and Meta [56], [61] show that many datacenter applications are disproportionately *frontend-bound*. These workloads have large instruction footprints, high instruction-cache pressure, frequent branch mispredictions, and pipeline stalls that arise from complex control flow, large software stacks, and rapidly shifting code paths. This is in contrast to traditional benchmarks, such as SPEC CPU2017 [26], that typically spend the majority of their time retiring instructions. Hence, architectural conclusions drawn from conventional benchmarks can be misleading.

This gap creates an important problem for hardware vendors and the research community. Vendors design processors for datacenters [3], [4], [30], but they often lack access to the workloads that matter most, because datacenter operators cannot share production services due to confidentiality, proprietary business logic, data sensitivity, and security concerns. Academic researchers face the same barrier [35], [36], [57]–[59], limiting reproducibility and slowing progress on architecture ideas that target the true bottlenecks of production. Representative, openly available benchmarks are thus essential. They enable credible design-space exploration, allow fair evaluation across platforms, and help ensure that next-generation CPUs are optimized for realistic workload behavior.

Prior Approaches. However, building such benchmarks is difficult, and researchers have explored two main directions.

The first direction is *statistical workload cloning* [14], [20], [45], [52], [53]. The state-of-the-art system, Ditto [45], builds multi-tier application skeletons with generated assembly templates. However, its assembly-centric approach yields rigid, ISA-specific artifacts, relies on intrusive profiling (e.g., Valgrind) that distorts runtime behavior, and targets high-level service structure rather than microarchitectural properties such as i-cache or branch pressure.

The second direction is *manually engineered proxy benchmarks* [18], [19], [61]. For instance, DCPerf [61] provides open-source benchmarks intended to reflect Meta's production services. However, creating and maintaining such proxies demands engineering effort, domain expertise, and constant re-tuning as software evolves. Techniques to boost metrics can also be artificial, e.g., DCPerf's I-cache boosting [49] abruptly context-switches into a synthetic instruction injection loop instead of naturally increasing pressure in service phases.

Importantly, prior approaches typically optimize for a single hardware configuration. A benchmark that matches an application's IPC and cache miss rates on one machine provides no guarantee that it will respond similarly when the prefetcher or cache hierarchy is changed. This is precisely what benchmarks are *for*: evaluating the impact of microarchitectural design changes. Two programs with identical counter profiles on a baseline configuration can respond in completely different ways to a new prefetcher if the underlying memory access patterns differ. Without *sensitivity fidelity*, i.e., ensuring that

the benchmark's performance responds to design knobs in the same way as the original workload, cloned benchmarks risk producing misleading design conclusions.

Our Work. We argue that *agentic AI* can close the datacenter benchmark gap by automatically synthesizing benchmarks that match a workload's performance counters on a given machine and, more importantly, its sensitivity to microarchitectural design changes. Benchmark cloning is a natural fit for agentic AI since the search space is too large and hardware-dependent for static analysis or one-shot generation. Moreover, hardware counters and simulation traces provide a measurable reward signal after every candidate edit. This enables an autonomous loop of measuring, diagnosing, transforming, and re-measuring. Agents can re-run the loop whenever the target service changes, without additional manual effort.

However, existing agentic AI systems [5], [13], [37], [47], [55], [69] are not designed for this task. State-of-the-art coding agents such as SWE-agent [68] and multi-agent frameworks such as MetaGPT [27] optimize for functional correctness [31]. They pass test suites or satisfy natural-language specifications, but they are not designed for multi-objective, hardware-dependent targets that can conflict (e.g., increasing branch MPKI while preserving instruction-cache pressure and IPC). They also lack *microarchitectural causal grounding*. Furthermore, existing code-generation pipelines overfit to a single platform configuration, producing artifacts whose fidelity degrades as design knobs change.

We propose *Fidelio*, a microarchitecture-aware, multi-agent cloning workflow. Fidelio closes the loop around two complementary signals: hardware-counter telemetry collected on real machines and performance-sensitivity trends extracted from architectural simulations. Each iteration 1) diagnoses which microarchitectural gaps dominate, 2) applies a targeted code transformation informed by that diagnosis, and 3) re-evaluates the candidate, continuously narrowing the distance to the target workload's properties across counters and their response to design-knob changes.

Fidelio converges to the target workload in two phases. First, the *Counter Matching* phase uses lightweight hardware-counter telemetry to match a tiered, weighted set of performance events, iterating quickly on real hardware to bring the candidate into the correct performance domain. Then, the *Sensitivity Matching* phase validates and repairs robustness by sweeping the candidate through an architectural simulator across design knobs (e.g., prefetching, branch prediction, cache sizes) to match the resulting performance sensitivity trends of the original workload.

Counter Matching provides a cheap, measurement-driven path to a strong baseline candidate, avoiding an expensive search in simulation. On the other hand, Sensitivity Matching ensures the clone is useful for its intended purpose: design-space exploration. The result is a system that can take a running datacenter service and, using non-intrusive counters and live-collected traces, automatically produce a portable, open-source benchmark clone that faithfully mirrors the original workload's response to architectural what-if questions.

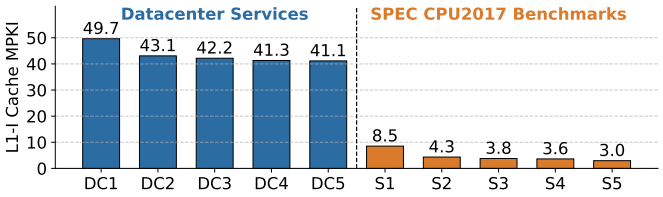

Fig. 1. L1 I-cache MPKI for the five most I-cache-intensive services at Meta and SPEC CPU2017 benchmarks on the same hardware generation.

Results. We evaluate Fidelio on open-source benchmarks from the DCPerf suite [61] and on production workloads from Meta. Across both sets, synthesized benchmarks match the target across microarchitectural axes while also preserving sensitivity to prefetcher, branch predictor, and cache design changes. The production workloads demonstrate that Fidelio can clone proprietary services using non-intrusive telemetry, without access to source code or binaries, producing portable, open-source proxies suitable for external design-space exploration.

In summary, this paper makes the following contributions:

- We present Fidelio, an *agentic AI* system for privacy-preserving, production-faithful benchmark cloning.
- We introduce a two-phase convergence methodology for the agentic workflow: *Counter Matching* on real hardware followed by *Sensitivity Matching* via simulation sweeps.
- We design microarchitecture-aware diagnostic playbooks that map performance gaps to targeted code transformations.
- We evaluate Fidelio on open-source DCPerf benchmarks and production workloads from a hyperscaler.

II. The Datacenter Benchmark Gap

A datacenter benchmark must capture two properties: the microarchitectural profile of production workloads and, importantly, the way that profile shifts when hardware design knobs turn. We show that neither traditional benchmarks (§II-A) nor carefully engineered proxies (§II-B) satisfy both requirements.

A. Datacenter Workloads vs. SPEC CPU2017

To illustrate the gap between datacenter and traditional workloads, we profile production services from Meta alongside SPEC CPU2017 [26] benchmarks on the same server platform.

Instruction-cache pressure. Figure 1 compares the L1 instruction-cache miss rate, measured in misses per kilo-instruction (MPKI), for the top-5 services at Meta (ranked by L1-I MPKI) and the top-5 SPEC CPU2017 benchmarks (ranked by L1-I MPKI). The datacenter services exhibit an order of magnitude higher L1-I MPKI values. This is due to their large instruction footprints: datacenter services are built from deep software stacks with extensive library dependencies, frequent indirect calls through virtual dispatch, and code paths that shift with incoming request patterns. SPEC benchmarks, by contrast, execute compact, self-contained loops whose instruction working sets fit comfortably in the L1-I cache.

Branch misprediction pressure. Figure 2 shows the branch MPKI for the top-5 services at Meta (ranked by branch MPKI) and the top-5 SPEC CPU2017 benchmarks (ranked by branch MPKI). Datacenter services consistently suffer higher branch

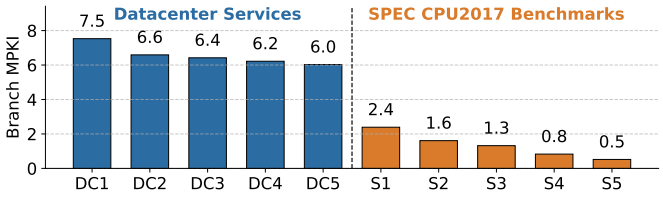

Fig. 2. Branch MPKI for the five most branch-intensive services at Meta and SPEC CPU2017 benchmarks on the same hardware generation.

misprediction rates. They traverse complex, data-dependent control flow across RPC handlers, (de)serialization layers, and dynamically dispatched request processing pipelines. These patterns are inherently difficult for branch predictors to learn, as the taken/not-taken outcomes depend on request inputs and inter-service communication patterns that vary at runtime. SPEC benchmarks tend to have more regular, loop-dominated control flow with branch patterns that modern predictors handle well after a short warm-up period.

B. Proxy Benchmarks Lack Sensitivity Fidelity

Given the mismatch with SPEC, recent efforts such as DCPerf [61] have produced proxy benchmarks designed to replicate datacenter workload behavior. These proxies are carefully engineered to match the performance-counter profiles of production services on a reference hardware platform. However, we find that this *static* fidelity does not translate when the microarchitecture changes. A proxy matching a production workload's counters on one configuration can diverge significantly when hardware knobs are varied.

We demonstrate this fragility using two experiments on our server, isolating different microarchitectural dimensions. We compare how a production service and its corresponding DCPerf proxy respond to the same hardware change.

Varying LLC capacity. We use Intel Resource Director Technology (RDT) [28] to partition the last-level cache (LLC), effectively varying the LLC capacity available to workloads. Figure 3 shows the normalized performance of a production service and its proxy relative to the full 16-way LLC allocation. Both workloads start at the same performance on the full cache (by construction, the proxy was tuned on this configuration), but their sensitivity curves scale differently as capacity is reduced. The production service degrades by roughly 29.3% when restricted to a single LLC way, exhibiting a steep, nearly linear decline as capacity shrinks. The proxy, by contrast, loses only about 4.4% under the same restriction, with a much flatter curve across all configurations. This multi-fold difference in sensitivity magnitude means the proxy dramatically underestimates how much the production service depends on LLC capacity. An architect using the proxy would conclude that halving the LLC has minimal performance impact, when in reality the production workload suffers a significant degradation.

We observed the same divergence when using LLC code-data prioritization [30] (via RDT), where the proxy responded differently from the production service when the LLC was configured to prioritize instruction lines over data lines.

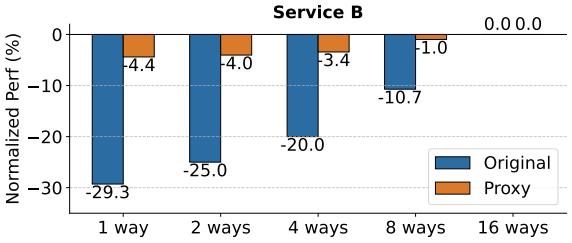

Fig. 3. Performance of a datacenter *Service B* and its proxy benchmark with different LLC sizes.

Enabling and disabling hardware prefetchers. We measure the performance impact of selectively disabling hardware prefetchers. Figure 4 shows normalized performance relative to a baseline with all prefetchers enabled, for three configurations: disabling all prefetchers, disabling only L1 prefetchers, and disabling only L2 prefetchers. This decomposition reveals which level of the prefetcher hierarchy each workload depends on most. Both the production service and its proxy degrade when prefetchers are disabled, but by substantially different amounts and with different patterns across the three configurations. The production service's memory access patterns (pointer-chasing through large, irregular data structures and instruction-fetch sequences across a wide code footprint), interact with each prefetcher level in ways that the proxy's more regular, synthetically tuned access patterns do not replicate. For example, the proxy may underestimate the impact of disabling L1 prefetchers while overestimating the impact of disabling L2 prefetchers, or vice versa. An architect using the proxy to evaluate prefetcher design alternatives would observe a qualitatively different performance across prefetcher levels than the production workload exhibits.

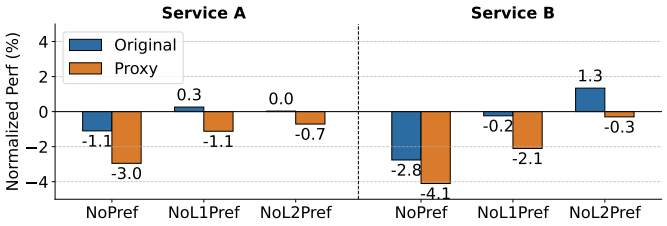

Fig. 4. Performance of datacenter *Services A and B* and their proxy benchmarks under three configurations: disabling all prefetchers, disabling only L1 prefetchers, and disabling only L2 prefetchers, normalized to enabling all prefetchers.

Takeaway. These experiments reveal a fundamental limitation of current proxy benchmarks: *matching performance counters on a single hardware configuration does not guarantee that the benchmark will respond to microarchitectural changes in the same way as the target workload*. Two programs can have identical IPC, cache miss rates, and branch MPKI on a baseline machine, yet diverge sharply when the cache hierarchy, prefetcher, or allocation policy changes. The reason is that the underlying code structure, memory access patterns, and control flow that *produce* those counters are structurally different. This motivates the need for *sensitivity-aware* benchmark cloning.

III. Fidelio Design and Implementation

Guided by the insights from our characterization in §II, we design *Fidelio*, a multi-agent system that turns non-intrusive telemetry into portable, open-source microbenchmarks.

The main insight is that a single code generation step cannot produce a program with the right counter profile *and* the right response to architectural change. Instead, Fidelio treats the problem as a guided search: each candidate program is compiled, executed, profiled, and the resulting performance gap drives the next code edit. Hardware counters and simulation traces supply a rich, quantitative signal at every step, turning an intractable generation task into a convergent optimization loop. Fidelio relies on LLM-based agents to navigate this loop as the search space is *semantic*: e.g., closing a gap in branch MPKI without regressing I-cache pressure requires understanding which code transformations affect which architectural mechanisms. This causal reasoning cannot be performed by rule-based or gradient-based optimizers.

Fidelio uses the performance profile of a production service without ever accessing the application's source code or binary. It emits a self-contained microbenchmark that reproduces both the workload's counter signature on a reference machine and, importantly, the way that signature responds to hardware design exploration, e.g., with new prefetchers or branch predictors. Since the system reasons solely from aggregate statistics, the resulting artifacts carry no proprietary information and can be released openly.

Fidelio addresses the benchmark design challenges through a closed-loop, multi-agent workflow organized into two phases. Figure 5 depicts the workflow of both phases. In the first phase, *Counter Matching*, Fidelio rapidly converges hardware counters on real hardware using lightweight telemetry and fast iteration cycles. In the second phase, *Sensitivity Matching*, Fidelio aligns the benchmark's performance-sensitivity trends across several categories of microarchitectural design knobs through simulation sweeps.

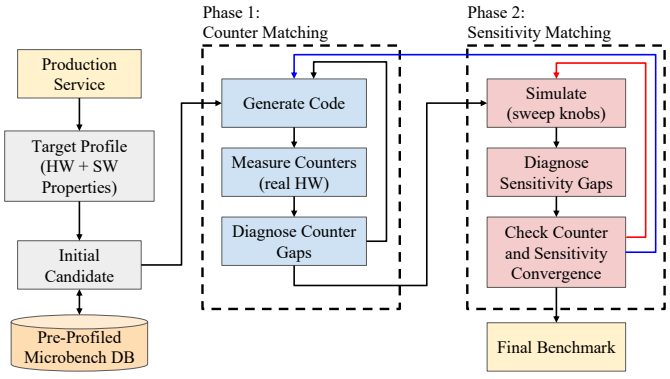

Fig. 5. Two-phase benchmark generation in Fidelio.

Scope. Fidelio targets *CPU microarchitecture* exploration. Its fidelity model spans frontend, backend, and memory-system behavior, and the metric tiers are programmable; a backend- or memory-bound phase of an application is handled by the same loop as the frontend-bound services (§IV-A1). Operating-system, kernel, and I/O behavior are *outside* Fidelio's current scope: it clones the user-space microarchitectural core of a service, not its system-call, networking, or storage surface. We view these as complementary rather than competing concerns. Fidelio regenerates the microarchitectural core on demand, while full-system suites supply the OS and I/O surface.

TABLE I
Metric tiers and weights. The optimization loop addresses lower-numbered tiers first.

Tier	Metric	w_i	Rationale
T0	IPC	1.0	Primary fidelity signal
T1	L1 I-cache MPKI	0.8	Dominant DC bottleneck
T1	Branch predictor MPKI	0.7	Frontend control-flow
T2	L2 I-cache MPKI	0.6	Deeper I-fetch pressure
T2	BTB MPKI	0.5	Indirect branch complexity
T2	L1 D-cache MPKI	0.5	Memory pattern fidelity
T3	LLC MPKI	0.3	Last-level cache pressure
T3	I-TLB / D-TLB MPKI	0.3	Footprint size signal
T3	Top-down L1 breakdown	0.3	Pipeline balance

A. Fidelity Model

Before synthesizing benchmarks, Fidelio must define precisely what "faithful" means. Not all hardware counters matter equally: an IPC mismatch renders a benchmark fundamentally unrepresentative, while a TLB discrepancy is secondary. Treating all counters with equal urgency risks over-fitting low-priority metrics at the expense of more important signals. Thus, Fidelio introduces a *tiered fidelity model* that reflects the causal structure of the microarchitecture.

Given a target profile T from the original workload and a candidate benchmark whose measured profile is M, Fidelio defines per-metric fidelity as $f_i = 1 - |M_i - T_i|/T_i$ and aggregates using a weighted composite $F = \sum_i w_i \cdot f_i / \sum_i w_i$. The weights w_i encode a priority hierarchy (Table I). IPC is the primary fidelity signal—if IPC diverges, the benchmark is fundamentally unrepresentative regardless of how well it matches other counters. Tier 1 captures the dominant datacenter bottleneck [33], [61]: frontend stalls driven by instruction-cache misses and branch mispredictions. Tiers 2 and 3 capture progressively finer-grained effects—BTB pressure, data-cache behavior, TLB footprint, and overall pipeline balance—that matter for completeness but should never take priority over the frontend metrics that dominate datacenter performance.

Counter fidelity converges when $F \geq F_{\min}$ *and* every T0/T1 metric individually satisfies $f_i \geq f_{\min}$, where $F_{\min}$ and $f_{\min}$ are configurable thresholds (we use $F_{\min} = 0.85$, $f_{\min} = 0.90$ for T0 and $f_{\min} = 0.80$ for T1). Without the per-tier floor, a benchmark could achieve high composite F by perfectly matching TLB and D-cache behavior while exhibiting a 30% IPC error—a trade-off that the weighted average would allow, rendering the benchmark useless.

While Table I reflects the frontend-dominated profile typical of datacenter services, the hierarchy is programmable. For compute-bound workloads, an architect can reconfigure the tiers to prioritize backend metrics, without changing the optimization loop.

TABLE II
REPRESENTATIVE OPTIMIZATION CATEGORIES AND DESIGN-KNOB VALUES SWEPT DURING SENSITIVITY MATCHING. IN EACH SWEEP, THE FOCUS CATEGORY VARIES WHILE ALL OTHERS REMAIN AT BASELINE.

Category	Swept values
Prefetcher (PF)	none, next-line, stride*, IP-Str, IPCP, Berti, Bingo
Branch predictor (BP)	bimodal, gshare, TAGE*, TAGE-SC-L, perceptron
L1-I cache size (L1I)	16 KB, 32 KB*, 48 KB, 64 KB
L1-D cache size (L1D)	16 KB, 32 KB*, 48 KB, 64 KB
L2 cache size (L2)	256 KB, 512 KB*, 1 MB, 2 MB
Replacement (REPL)	LRU*, SRRIP, DRRIP, SHiP, Hawkeye

*Baseline value for that category. These categories and values are representative rather than exhaustive; Sensitivity Matching can incorporate additional design knobs and values.

Counter fidelity, however, is only half the story. We saw that two programs with identical IPC, I-cache MPKI, and branch MPKI on a baseline configuration can respond in completely different ways to microarchitectural feature changes, e.g., hardware prefetching, because the underlying access patterns that *produce* those miss rates differ in structure. This observation motivates *sensitivity fidelity*: the requirement that the benchmark responds to microarchitectural design-knob changes in the same way as the original workload.

Fidelio defines six optimization categories (Table II), each representing a microarchitectural parameter that architects routinely sweep during design-space exploration. For each category C with values $\{v_1, \ldots, v_n\}$ and a designated baseline v_b, the sensitivity profile captures the vector of relative IPC changes:

$$\mathbf{S}_C = \left[\frac{\text{IPC}(v_i) - \text{IPC}(v_b)}{\text{IPC}(v_b)} \right]_{i=1}^{n}. \tag{1}$$

Fidelio quantifies sensitivity fidelity per category using two complementary lenses. Cosine similarity captures whether the sensitivity *curves* have the same shape, independent of absolute IPC:

$$SF_C = \frac{\mathbf{S}_C^{\text{orig}} \cdot \mathbf{S}_C^{\text{bench}}}{\|\mathbf{S}_C^{\text{orig}}\| \, \|\mathbf{S}_C^{\text{bench}}\|}. \tag{2}$$

Kendall's τ captures whether the sensitivity *rankings* agree—which design-knob value is the best, which is the second-best, and so on:

$$RO_C = \tau\Big(\text{rank}(\mathbf{S}_C^{\text{orig}}),\ \text{rank}(\mathbf{S}_C^{\text{bench}})\Big). \tag{3}$$

Both metrics are necessary. Cosine similarity alone could declare convergence when two profiles share the same slope but swap which prefetcher wins, which is a failure for design-space exploration. Rank correlation alone could miss dramatic magnitude differences as long as the ordering holds. Sensitivity fidelity converges when $SF_C \geq SF_{\min}$ and $RO_C \geq RO_{\min}$ for every category C, where both thresholds are configurable (we use $SF_{\min} = 0.90$ and $RO_{\min} = 0.85$ in our experiments).

B. Counter Matching

The first phase closes the gap between the candidate's hardware-counter profile and the target's profile on a baseline configuration. Each iteration requires *compilation*, *a measurement pass* that collects hardware counters and software properties via sampling-based profiling [38], and *a diagnostic step*. The entire cycle completes in a few minutes, making it practical to execute dozens of iterations and converge tightly before investing in expensive simulation sweeps.

The tier ordering reflects what matters most for datacenter workloads: IPC is the most important fidelity signal, followed by the frontend metrics (I-cache and branch MPKI) that our characterization identified as the dominant bottleneck. This ordering also has a practical benefit: some tiers are causally entangled. IPC, for instance, is affected by changes to nearly every other metric, so fixing a T1 I-cache gap will inevitably shift IPC as well. By always addressing the highest-priority failing tier first, Fidelio avoids wasting iterations on lower-tier fixes that would be disrupted by a subsequent higher-tier correction.

Within each iteration, Fidelio selects between two modes based on gap magnitude relative to a configurable *escalation threshold* θ_e (we use $\theta_e = 20\%$ in our experiments). When the focus metric's gap falls below θ_e, the system applies *parameter tuning*—adjusting loop counts, array sizes, trip counts, and pointer-chase depth—changes that are low-risk and unlikely to regress other metrics. When the gap exceeds θ_e, or when two consecutive parameter-tuning iterations fail to make progress (a stall), Fidelio escalates to a *structural edit*: adding or removing functions, reshaping control-flow topology, altering branch patterns, or modifying the instruction mix. Since structural edits can ripple across tiers, Fidelio re-validates *all* metrics after every such edit, not just the focus metric. This escalation mechanism balances between convergence speed and stability, i.e., the system always tries the gentlest fix first and resorts to more invasive surgery only when necessary.

IPC deserves special treatment because it emerges from the interaction of all microarchitectural components rather than from any single code property. Fidelio never prescribes "increase IPC" directly. Instead, it decomposes the IPC gap through the Top-Down Level-1 breakdown and delegates to the appropriate *playbook*—a structured mapping from a specific performance gap (e.g., "I-cache MPKI is below target") to a prioritized sequence of code transformations, their expected microarchitectural effects, and constraints on what must not change. If the gap is frontend-bound, the system invokes I-cache or branch-predictor playbooks; if backend-bound, D-cache or LLC playbooks; if driven by bad speculation, branch-entropy adjustments. This decomposition transforms a vague macro-level signal ("IPC is too high") into an actionable directive ("the frontend is too efficient, thus, inflate I-cache pressure").

C. Sensitivity Matching

Counter convergence only guarantees that the benchmark looks like the original workload on *one* configuration. Sensitivity Matching ensures that the benchmark *behaves* like the original across configurations, e.g., that it responds to prefetcher

or branch predictor changes with the same performance trends. This transforms the benchmark from a static counter clone into a *behavioral proxy* that architects can trust for design-space exploration.

The phase opens with a simulation sweep: the system runs the counter-converged candidate across categories (Table II), sweeping each category independently while holding other knobs at baseline. Fidelio computes SF_C (Eq. 2) and RO_C (Eq. 3) for each category, marks categories that meet both thresholds as converged, and selects the category with the worst SF_C as the focus for repair.

The agents then perform the most challenging reasoning in the system: bridging the semantic gap between a numerical sensitivity mismatch and a code-level root cause. Different microarchitectural mechanisms favor different access patterns: for prefetchers, Berti [50] excels at irregular but temporally correlated streams, IPCP [51] captures complex stride patterns, and next-line prefetchers reward spatial locality. Hence, the system must reason about *which* mechanism benefits from *which* code structure. To ground this reasoning, Fidelio augments the prompt with distilled knowledge about each mechanism drawn from academic papers, architecture manuals, and vendor white papers, enabling it to map a sensitivity gap to a specific structural diagnosis (e.g., "the access stream has too much temporal correlation that Berti exploits"). This kind of second-order reasoning—not "does the benchmark have the right miss rate?" but "does it have the right *kind* of misses, the kind that respond to each hardware mechanism in the right way?"—distinguishes Fidelio from prior approaches.

The fix plan that emerges from this diagnosis targets the specific code region responsible for the sensitivity gap. Prefetcher and cache-size fixes modify memory access patterns and data-structure layouts. Branch-predictor fixes reshape branch topology and correlation depth. I-cache-size fixes adjust code footprint distribution. Every fix plan carries explicit constraints that protect already-converged categories and counter fidelity, i.e., Fidelio will try not to prescribe a prefetcher fix that alters branch structure if branch-predictor sensitivity has already converged.

After applying the fix, Fidelio performs a Counter Matching re-check: a quick measurement pass that confirms counter fidelity has not regressed beyond a configurable tolerance (we use $\Delta F > 0.03$). If regression exceeds this bound, the system rolls back the fix and re-invokes the reasoning agents with tighter constraints. This safeguard is essential because Sensitivity Matching edits target the structures, such as access patterns, branch topologies, footprint distributions, that determine counter values. A prefetcher-sensitivity fix that introduces pointer-chasing to reduce regularity will inevitably perturb D-cache MPKI. The re-check catches cases where this perturbation exceeds acceptable bounds. When the re-check passes, Fidelio re-sweeps only the focus category to verify convergence, avoiding the cost of a full six-category sweep.

The interplay between the two phases gives Fidelio its power. Counter Matching cheaply establishes the right ballpark on real hardware, so Sensitivity Matching starts from a candidate that already matches the target's counter profile rather than from a random program. Sensitivity Matching then refines the *internal structure* of the code—the patterns and topologies that determine how the benchmark interacts with microarchitectural mechanisms—while the Counter Matching re-check ensures these refinements do not break the counter match. The resulting benchmark captures both the static snapshot (counters on one configuration) and the dynamic response (performance trends as configurations change).

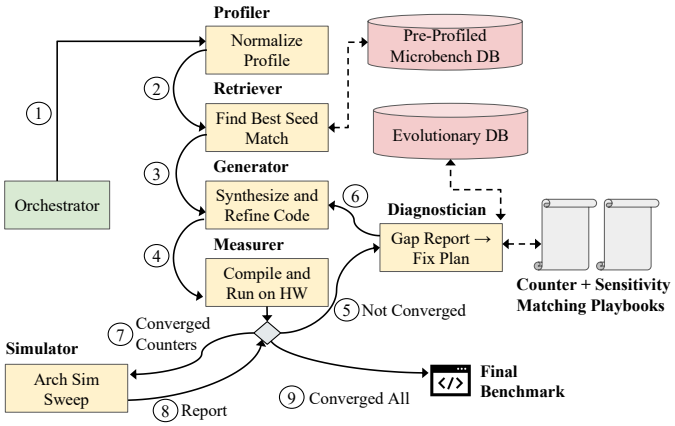

Fig. 6. Agentic workflow architecture of Fidelio.

D. Agent Architecture

Figure 6 shows Fidelio's high-level architecture. Current models struggle to maintain context and consistency when using a single monolithic LLM to simultaneously reason about counter gaps, generate code, and evaluate results. Thus, Fidelio assigns each subtask to a focused agent with a narrow interface and clear success criteria. Then, these six specialized agents, each responsible for a distinct competency, are all coordinated by an Orchestrator.

Profiler. The pipeline begins with the Profiler, which ingests preprocessed `perf` output and transforms it into a normalized target profile. The Profiler computes derived metrics—MPKI from raw miss and instruction counts, the Top-Down Level-1 breakdown from pipeline slots—and flags anomalies such as IPC values that exceed the theoretical dispatch width of the target microarchitecture. Thus, every downstream agent will operate on a consistent, validated representation of the target workload.

Retriever. Generating a microarchitecturally faithful benchmark from scratch is difficult. We observed that even frontier LLMs, when asked to produce code with a specific IPC or cache miss rate, generate programs whose counter profiles bear little resemblance to the target. Fidelio sidesteps this cold-start problem by starting from existing microbenchmarks that already exhibit *similar* behavior. To bootstrap the search, Fidelio builds a knowledge database comprising thousands of microbenchmarks that stress different parts of the processor pipeline: frontend-heavy loops, memory-bound pointer chases, branch-intensive control flow, and mixed workloads. Each entry in this database stores source code, compilation flags,

TABLE III
SENSITIVITY MATCHING DIAGNOSTIC PLAYBOOKS (REPRESENTATIVE ENTRIES). EACH ENTRY MAPS A SENSITIVITY MISMATCH SYMPTOM TO ITS MICROARCHITECTURAL ROOT CAUSE AND A TARGETED CODE TRANSFORMATION.

Category	Symptom	Root Cause	Fix Action
Prefetcher	Benchmark benefits *more* from advanced PF than original	Access patterns too regular; stride PF captures most benefit, advanced PFs find even more	Add pointer-chasing indirection, randomize index computation, mix strides
	Benchmark benefits *less* from advanced PF than original	Access patterns already irregular or working set fits in cache	Increase data working set beyond L1-D, introduce regular strided regions that reward prefetching
Branch predictor	Benchmark benefits *more* from TAGE-SC-L over TAGE than original	Too many long-history-correlated branches that the statistical corrector exploits	Shorten branch correlation depth, use more locally predictable branches
	Benchmark is *less* sensitive to predictor table size than original	Too few unique branch PCs; all entries stay warm	Increase number of distinct branch sites across more functions
Cache size	L1-I sensitivity too steep	Code footprint sits at a cliff near the swept sizes	Redistribute hot functions across pages; adjust function count to shift the knee
	L1-D sensitivity too flat	Data working set much smaller or larger than swept range	Tune array sizes so the working set falls between the smallest and largest swept sizes
Replacement	DRRIP/SHiP helps the benchmark more than original	Scan-like streaming accesses that thrash LRU	Reduce streaming fraction, add more temporal reuse
	DRRIP/SHiP helps the benchmark less than original	Lacks a mixed reuse-distance pattern that makes RRIP effective	Add bimodal reuse distances (frequently-reused + rarely-reused lines)

a full hardware-counter profile, software properties (basic-block size distribution, read-after-write dependency distance, instruction-type mix, unique I-cache lines, loop depth, call depth), and, when available, sensitivity profiles from prior simulation sweeps. The Retriever queries this database to find seeds whose profiles most closely match the target.

All fields project into a joint embedding vector that supports approximate nearest-neighbor retrieval. Each feature dimension in the embedding is scaled by $\sqrt{w_i}$, where w_i is the tier weight from Table I, and the resulting vector is L2-normalized to unit length. This construction ensures that dot products between two embeddings approximate the weighted cosine similarity defined by the tier weights, so that high-priority metrics such as IPC and I-cache MPKI dominate the distance computation. As a result, a seed that closely matches IPC and I-cache MPKI automatically ranks above one that matches only TLB behavior.

When sensitivity profiles exist for the target, the embedding additionally incorporates sensitivity-vector dimensions, biasing seed selection toward microbenchmarks whose design-knob responses already resemble the original workload's. In practice, starting from a well-chosen seed reduces Counter Matching from hundreds of iterations to tens, because the system needs to close a 10–20% gap rather than synthesize microarchitectural behavior from nothing.

Generator. The Generator is the system's code-synthesis engine, implemented as an LLM-based agent that operates in two distinct modes. In *Init* mode, the Generator receives a seed template alongside the target profile and produces an initial candidate by reshaping loop trip counts, array sizes, branch patterns, function count, basic-block sizes, and instruction mix. In *Refine* mode, the Generator receives the current candidate together with a structured fix plan from the Diagnostician and applies *only* the prescribed transformations, preserving code regions for metrics already within tolerance. This modal separation matters: Init mode explores broadly to establish a reasonable starting point, while Refine mode makes surgical, constrained edits that converge without oscillation.

To improve the quality of generated code, the Generator is augmented with *an evolutionary database* of successful edits accumulated across prior optimizations. Each entry records the performance gap that triggered the edit, the code transformation that was applied, and the resulting counter improvement. At generation time, the Generator retrieves the most relevant entries via Retrieval Augmented Generation (RAG) and includes them as in-context examples, so that the model can draw on proven transformations for similar tasks. All emitted code must compile and should execute deterministically. The Orchestrator enforces these constraints through a compilation gate before any measurement.

Measurer. The Measurer compiles the candidate with specified flags, runs it on a given machine to collect counters, and returns a structured profile in the same schema as the target. This agent is invoked on every Counter Matching iteration and during counter re-checks after Sensitivity Matching edits. Thus, its speed is critical to the overall convergence time.

Simulator. The Simulator is invoked during Sensitivity Matching. For each optimization category in Table II, it runs the compiled benchmark through an architectural simulator under every design-knob value, sweeping one category at a time while holding all others at baseline. All sweeps run in parallel. The Simulator returns per-category sensitivity profiles in the same schema as the target, allowing Fidelio to compute cosine similarity and rank correlation.

Diagnostician. This is the core agent that performs diagnostic reasoning while consulting microarchitecture-aware *playbooks*. Given a gap report and machine's specifications, it emits a fix plan specifying the focus metric or category, the

gap direction and magnitude, a prioritized sequence of code transformations, and explicit constraints protecting already-converged metrics. This separation transforms what would otherwise be an open-ended "make IPC closer to the target" instruction into a precise directive like "add N unique functions with M basic blocks each to increase I-cache MPKI by ≈P points, without altering branch structure or array sizes."

The Diagnostician draws on two families of playbooks. *Counter Matching* playbooks address counter mismatches on the baseline configuration. When I-cache MPKI falls below the target, the playbook traces the root cause to insufficient code footprint or excessive temporal reuse and prescribes adding unique functions, reducing loop trip counts, and inserting non-trivial basic blocks to inflate the footprint. When I-cache MPKI exceeds the target, the playbook prescribes the reverse: consolidating functions, inlining hot paths, and concentrating execution on fewer code paths. Branch-predictor playbooks follow an analogous pattern, manipulating branch entropy and prediction-history depth: replacing predictable counted loops with data-dependent branches to increase mispredictions, or converting data-dependent branches to conditional moves to decrease them. D-cache playbooks adjust array sizes relative to L1-D capacity and modify access stride and indirection depth. For IPC gaps, the Diagnostician decomposes the problem through the Top-Down breakdown and delegates to the appropriate component playbook—a frontend-bound IPC gap becomes an I-cache or branch problem, a backend-bound gap becomes a D-cache or TLB problem.

Sensitivity Matching playbooks are the center of Fidelio's reasoning, because they address *second-order* effects, i.e., reasons that determine how the benchmark responds to hardware changes. Some examples are shown in Table III. Consider the prefetcher example. When the benchmark benefits too much from an advanced prefetcher, the root cause is not simply "too many cache misses"—counter fidelity may be perfect—but rather that the *pattern* of misses is too regular, giving advanced prefetchers more predictable streams to learn than the original workload provides. The fix does not reduce the miss rate (which would break counter convergence); instead, it reshapes the miss *pattern* by introducing pointer-chasing indirection that defeats all prefetchers equally, pulling the advanced-prefetcher benefit down to match the target.

Every fix plan follows the same structure: a focus metric or category, the gap direction and magnitude, a prioritized sequence of transformations with expected impact, and an explicit constraint set identifying code regions and parameters that must remain unchanged. The Generator applies only the prescribed transformations and may not alter constrained regions. This bounded-blast-radius approach prevents oscillation between competing objectives.

Prompting style. Fidelio does not provide the LLM with curated input/output examples of the cloning task. Instead, each prompt supplies concrete domain knowledge: per-metric playbook recipes with specific techniques and explicit constraints on what must not change. On each iteration, the prompt also includes the previous candidate's source code, its measured hardware counters, and successful edits retrieved from the evolutionary database, so the model sees what was tried, what it produced, and which transformations worked for similar gaps in prior runs.

E. Privacy

Generated benchmarks are safe to release as open-source artifacts, despite being derived from proprietary workloads. Fidelio never observes the original application's source code. It reasons entirely from statistical profiles: counter values, distributions, and sensitivity vectors. All synthesized code is structurally novel; the system does not decompile, disassemble, or transplant production code. Data arrays contain synthetic patterns such as permutation-based pointer chasing, ensuring no production data leaks into the output.

F. End-to-End Example

We illustrate the full pipeline on a frontend-bound datacenter workload with low IPC, high L1-I and branch MPKI, and a prefetcher sensitivity curve showing substantial IPC gains from advanced prefetchers relative to the stride baseline.

During Counter Matching, the Retriever selects a seed whose IPC is close to the target but whose I-cache and branch MPKI fall short. The Diagnostician addresses tiers in priority order, beginning with T0. Because the I-cache gap exceeds θ_e, the first iteration applies a structural edit: it adds unique functions and reduces the main loop's trip count. The remaining gap falls below θ_e, so the next iteration applies parameter tuning, i.e., it adds basic blocks per function, pushing F above 0.93 with all T0/T1 metrics within tolerance. Counter Matching converges in three iterations.

These iterations illustrate how the playbooks navigate coupled metrics. Since all MPKI values share the instruction count as their denominator, naively adding executable instructions to raise I-cache MPKI dilutes branch MPKI. The I-cache playbook avoids this by inserting non-executed padding bytes between code blocks, which inflates the address-space footprint without changing the instruction count. Similarly, when the Top-Down breakdown attributes an IPC gap to the backend, the selected backend playbook injects high-latency serialized divisions, each consuming tens of cycles but counting as a single retired instruction. This is a Top-Down-selected playbook action, not a direct "lower IPC" command. When the breakdown still indicates excess backend headroom without disturbing cache or branch behavior, the playbook introduces contended atomic operations across threads, creating coherence stalls that burn cycles independently of the code's control-flow and memory-access structure.

Sensitivity Matching then sweeps the counter-converged candidate across all six design-space exploration categories. Branch-predictor and L1-I sensitivity converge immediately, but the benchmark benefits too much from Berti, indicating overly regular access patterns. The Diagnostician converts a fraction of array accesses to pointer-chasing through a shuffled indirection array; a counter re-check confirms that F remains above the threshold, and a re-sweep shows prefetcher

Algorithm 1 Structure of the clone emitted for the end-to-end example, with the playbook that introduced each construct.

Static code layout (I-cache and BTB playbooks)
generate $F = 8192$ functions $f_0 \dots f_{F-1}$
for all f_i **do**
    emit ∼20 executed instructions ▷ hold instruction count down
    emit `jmp` past 512 dead bytes ▷ spread exec. code across cache lines
    emit 12 indirect call sites into $f_{\sigma(i)}$ ▷ BTB, branch MPKI

Data layout (prefetcher and L1-D playbooks)
$A \leftarrow$ array of N nodes, one cache line each ▷ N set at the L1-D knee
$\sigma \leftarrow$ shuffle of $[0, N)$ with a fixed seed ▷ defeat stride prefetch
for all $k \in [0, N)$ **do**
    $A[\sigma(k)].next \leftarrow \&A[\sigma(k{+}1 \bmod N)]$

Execution body (IPC and Top-Down playbooks)
spawn $T = 4$ worker threads
barrier
for all thread t **in parallel do**
    $p \leftarrow \&A[t \cdot N/T]$; $acc \leftarrow 0$
    for $j \leftarrow 1$ **to** M **do**
        for $c \leftarrow 1$ **to** 12 **do** ▷ indirect call chain
            $acc \leftarrow f_{\mathrm{next}(c)}(acc)$
        for $s \leftarrow 1$ **to** D **do** ▷ data walk, dependent misses
            $p \leftarrow p \rightarrow next$; $acc \leftarrow acc \oplus p$
        for $d \leftarrow 1$ **to** 27 **do** ▷ serialized 64-bit divisions
            $acc \leftarrow acc / v_d$; `lfence`
        atomic $C \leftarrow C + 1$ on a shared line ▷ coherence stalls
barrier; consume acc so no work is optimized away

convergence. A second fix shrinks a hot array to soften an overly steep L1-D sensitivity curve. Finally, all categories meet their thresholds, and Fidelio emits a self-contained C program of roughly 200 lines.

The input profile. For this example the target vector is IPC 0.90, L1-I MPKI 27.4, branch MPKI 5.8, BTB MPKI 3.7, L1-D MPKI 17.4 and LLC MPKI 2.8, with a Top-Down Level-1 split of 36/27/28/15% across frontend, backend, retiring and bad speculation. Alongside these counters, the Profiler supplies LBR-derived software properties: a median basic block of 8 instructions, a taken rate of 0.63, a jump-distance distribution with a P90 of 64 KB, and a load/store/branch mix of 0.34/0.28/0.15. The sensitivity phase adds one relative-IPC curve per category, for instance the prefetcher curve that shows a 3.4% gain from Berti over the stride baseline. That vector is the whole specification. No source, binary, or raw execution trace of the service itself is used, which is what makes the profile safe to publish in place of the workload it describes.

The resulting program. Algorithm 1 gives the structure of the emitted clone. It has three parts, each contributed by a different playbook. The static layout is built first during Counter Matching: thousands of small functions, each executing only a handful of instructions but padded with dead bytes that the control flow jumps over, so that the code footprint grows without the instruction count growing with it. Chained indirect calls thread through these functions to supply BTB and branch pressure. The data layout is built second, during Sensitivity Matching, when the prefetcher fix converts regular array traversal into a shuffled pointer chain whose node count is tuned to the L1-D knee. The execution body then combines the two. Worker threads synchronize on a barrier, and each iteration of the main loop walks the indirect-call chain, performs a dependent data walk over the shuffled chain, issues a chain of serialized divisions, and updates a contended atomic on a shared cache line. The trip counts M and D and the division count are the parameter-tuning knobs the Diagnostician adjusts once the structure is in place, while the function count, padding size and chain length are the structural ones.

A backend-bound target reshapes this skeleton. Fidelio would keep the same loop and barrier structure, but the padding and indirect-call blocks would shrink to almost nothing while the data walk would grow into two chains sized to different levels of the hierarchy. The same playbook produces both, driven only by the difference in the target vectors.

G. Implementation

Fidelio comprises ∼5K lines of Python code and is organized around a central Orchestrator that drives the two-phase optimization loop. The Orchestrator invokes all agents, manages candidate generation, and enforces compilation and measurement gates.

The Profiler extracts a unified representation of program behavior from hardware performance counters and execution traces using the `perf` tool and `Intel's LBR` [38]. It derives 27 features spanning PMU counters, Top-Down Level-1 metrics, and code structure properties such as basic-block distributions, instruction mix, and control-flow characteristics. These features are canonicalized into a fixed schema to enable consistent reasoning across agents.

The Retriever indexes this feature space using a ScaNN [62] approximate nearest-neighbor structure over a corpus of 2,264 pre-profiled microbenchmarks [48]. It returns similar programs to guide optimization via tier-weighted embeddings (§III-D). This pre-profiled corpus spans diverse microarchitectural regimes—compute-heavy kernels, large memory-copy sequences, branch-intensive control flow—so the Retriever can place the starting candidate close to the target's performance region, narrowing the search space. Without it, agents must match counters from scratch, and even frontier LLMs fail to converge within tens of iterations under such a setup.

The Generator and Diagnostician are Gemini 2.5 Pro [23] LLM agents. The Generator uses the model for code generation and for handling repair and compilation issues. The Diagnostician maps performance signals to fix plans via constrained prompting, while the Generator applies these plans to produce modified C/C++ code. Candidate programs are iteratively refined through feedback from the measurement loop and validated via compilation.

The Measurer executes candidates and collects performance statistics (median of five runs), feeding results back into the optimization loop. For Sensitivity Matching, Fidelio exposes a REST interface to an external simulator (SST [54]), which provides target sensitivity profiles. Fidelio iteratively diagnoses mismatches, applies fixes, and returns updated candidates for re-simulation.

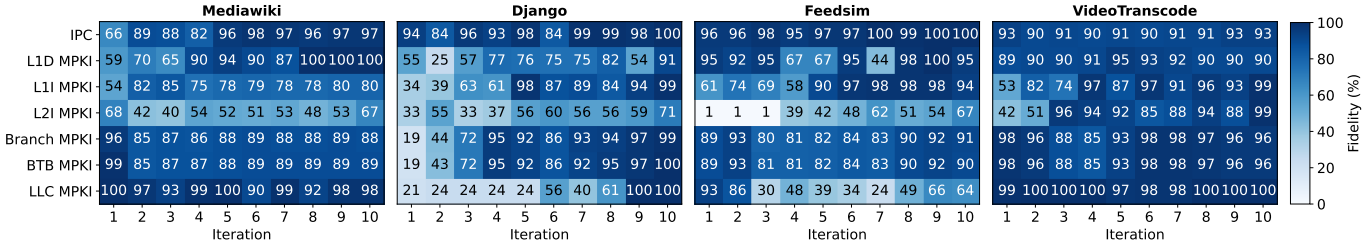

Fig. 7. Counter matching accuracy for DCPerf workloads (Mediawiki, Django, FeedSim, VideoTranscode) on Intel Emerald Rapids as heat maps over Fidelio iterations. Each cell reports per-metric fidelity f_i; darker shading means higher accuracy.

H. Limitations and Future Work

Fidelio optimizes for hardware-counter fidelity and sensitivity, but does not preserve the software structure of the original workload. By matching counters and software-side distributions (e.g., basic-block sizes, instruction mix), the synthesized benchmarks capture microarchitectural behavior without reflecting the libraries, frameworks, or software patterns that produce it. For example, a production service may spend significant time in compression libraries, memory allocators, or RPC layers whose calling conventions and code layout contribute to the observed counter profile. The current benchmarks reproduce the *effect* of these components but not their *presence*. In future work, we plan to address this gap by including library-level awareness into the cloning pipeline.

IV. Evaluation

Hardware platform. We run all experiments on an Intel Emerald Rapids (EMR) server [29] with a 28-core Intel Xeon Gold 5512U processor at 2.1 GHz and 128 GB of memory (8×16 GB 5600 MT/s). To minimize noise, we disable turbo boost and report the median of five runs for all counter measurements. To evaluate cross-platform generalization (§IV-F), we additionally run clones on an AMD Turin (TRN) server [2] with 126 cores at 3 GHz without any re-tuning.

Profiling and simulation. We collect hardware counters via `perf` and extract software properties (basic-block size distributions, branch behavior, instruction mix) from `perf record` with Intel Last Branch Record (LBR) sampling. For Sensitivity Matching, we use the SST simulator [54] with DRAMSim3 [44]. We sweep the six optimization categories from Table II, running each design-knob configuration independently while holding all others at baseline.

Workloads. We evaluate Fidelio on two sets of target workloads. The first set consists of four DCPerf [61] benchmarks: Mediawiki, Django, Feedsim, and VideoTranscode. Table IV summarizes their microarchitectural profiles. The second set consists of four production services from our hyperscaler, each corresponding to one of the DCPerf proxies, allowing direct comparison between Fidelio's clones and the manually engineered DCPerf proxies against their respective ground-truth production workloads.

Knowledge database. We build the microbenchmark knowledge database by profiling 2,264 benchmarks from wdlBench [48] on both platforms. Each benchmark is profiled to collect the full counter and software-property schema described in §III-D, and the resulting profiles are indexed for approximate nearest-neighbor retrieval.

TABLE IV
Microarchitectural profiles of DCPerf applications

Metric	Mediawiki	Django	Feedsim	VideoTr.
IPC	0.90	0.95	1.51	1.65
L1-D / L1-I MPKI	17.4/27.4	11.8/28.3	13.5/5.2	26.7/12.6
LLC MPKI	2.8	2.1	4.2	2.5
Branch / BTB MPKI	5.8/3.7	3.8/3.7	5.4/1.5	1.6/0.1
D-TLB / I-TLB MPKI	17.8/0.8	19.1/1.7	35.9/0.9	17.4/0.2
L2 TLB MPKI	1.9	1.2	5.7	0.1
CPU utilization (%)	95.4	88.7	76.7	97.2
Memory BW (GB/s)	35.9	34.3	42.7	79.2
FE/BE/Ret/BadSpec (%)	34/26/27/13	40/17/33/10	31/14/35/20	20/17/54/9

Thresholds. We use $F_{\min} = 0.85$ and $f_{\min} = 0.90/0.80$ for Counter Matching (composite fidelity and T0/T1 metric floors), $SF_{\min} = 0.90$ and $RO_{\min} = 0.85$ for Sensitivity Matching (cosine similarity and rank correlation per category), and an escalation threshold of $\theta_e = 20\%$ for switching from parameter tuning to structural edits. Counter re-check tolerance after Sensitivity Matching fixes is $\Delta F > 0.03$.

A. Counter Matching Accuracy

We first evaluate how accurately Fidelio's clones reproduce hardware-counter profiles on the baseline configuration. For each workload, we report per-metric fidelity f_i (§III-A) for seven key counters: IPC, L1 D-cache MPKI, L1 I-cache MPKI, L2 I-cache MPKI, branch MPKI, branch target buffer (BTB) MPKI, and LLC MPKI.

DCPerf workloads. Figure 7 shows the counter matching accuracy for the four DCPerf workloads on the Intel EMR server, presented as heat maps over successive Fidelio iterations. Each cell encodes the per-metric fidelity between the clone and the target, with darker shading indicating higher accuracy. Across all workloads, the highest-priority metrics, i.e., IPC, I-cache MPKI, and branch MPKI (T0 and T1), reach high fidelity, typically 80–100%, by the final iteration. The tiered priority is visible in the convergence dynamics: IPC stabilizes early and remains high, while the frontend metrics follow closely in subsequent iterations. Convergence is not monotonic. Since metrics are coupled through shared microarchitectural resources, intermediate iterations sometimes regress one metric while improving another. For example, Django's LLC MPKI drops sharply mid-search before recovering to 100% in later iterations, as the agents repair the regression through the closed-loop re-check mechanism de-

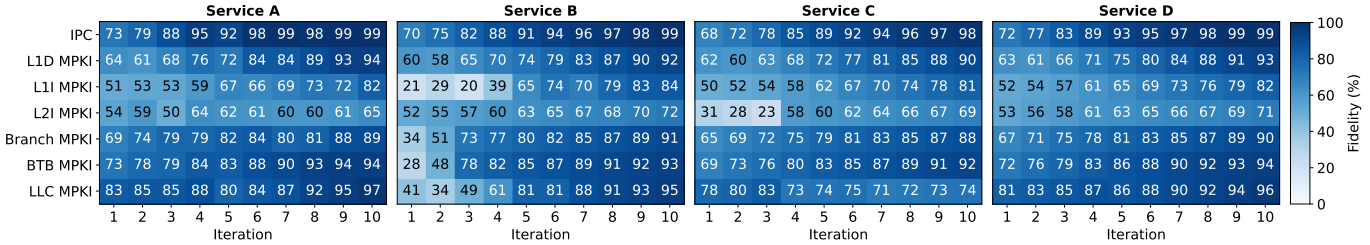

Fig. 8. Counter matching accuracy for four datacenter services at our hyperscaler, shown as heat maps over Fidelio iterations.

scribed in §III-F. Lower-tier metrics such as L2 I-cache MPKI prove harder to match, plateauing between 67–99% depending on the workload. The spread largely reflects seed quality: VideoTranscode, a compute-heavy workload well represented by similar kernels in the knowledge database, starts from a strong seed and achieves the highest accuracy across all tiers, while the frontend-dominated workloads require more aggressive structural edits that make it harder to simultaneously satisfy lower-tier constraints. This trade-off is consistent with the system's design: the tiered fidelity model deliberately deprioritizes T2/T3 metrics in favor of the frontend signals that dominate datacenter performance.

1) Backend-Bound Workloads: The workloads above are mostly frontend-bound. To show Fidelio is not specialized to this regime, we clone a backend-bound workload: PageRank from the GAP Benchmark Suite (GAPBS) [12], a graph kernel with irregular, pointer-chasing traversal. The application has low IPC and it is dominated by memory-bound stalls. Since the default tiers prioritize now-secondary frontend signals, we exploit the programmable hierarchy: IPC stays at T0 and the memory-boundness metrics (Memory Bound, L1 data cache, and LLC) are promoted above the frontend metrics, leaving the loop otherwise unchanged. With this tiering, Fidelio reproduces the behavior that defines PageRank, reaching IPC 99%, Memory Bound 99%, L1 data cache 82%, and LLC 93% fidelity, all meeting the convergence thresholds. The same closed loop, retargeted only through the tier weights, is thus capable of cloning a backend-bound graph workload faithfully.

Production workloads. Figure 8 presents the same analysis for four datacenter services at our hyperscaler. IPC converges strongly across all services (98–99%), confirming that Fidelio's telemetry-driven approach generalizes to production workloads where no source code or binary is available. The remaining T0/T1 metrics follow a similar pattern to DCPerf, though L1 I-cache MPKI settles slightly lower (81–84%) than for the open-source benchmarks, reflecting the deeper software stacks and larger instruction footprints of production services that are inherently harder to replicate. Convergence is more gradual than for DCPerf, with fewer sharp regressions, as the agents take more conservative steps when navigating the more complex counter landscape. The same T2/T3 trade-off observed in DCPerf holds here: L2 I-cache MPKI plateaus between 65–72%, and LLC MPKI varies across services, with Service C reaching only 74% as its irregular memory hierarchy behavior proves difficult to reproduce without regressing higher-tier metrics.

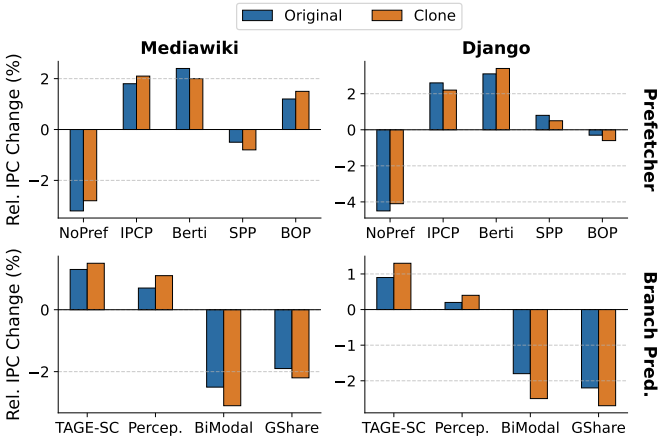

Fig. 9. Sensitivity matching for DCPerf workloads under prefetcher and branch predictor sweeps.

B. Sensitivity Matching Accuracy

We now evaluate how well the clones reproduce the target's sensitivity to microarchitectural design changes.

DCPerf workloads. We collect execution traces of DCPerf and run them through our SST simulator while varying the prefetcher and branch predictor configurations (Table II). We experimented with all six optimization categories. We present the prefetcher and branch predictor sweeps in Figure 9 and summarize the remaining cache-size and replacement-policy categories below. Figure 9 shows the sensitivity profiles (relative IPC change from baseline) for the original DCPerf workloads and their Fidelio clones across prefetcher and branch predictor configurations. We show results for Mediawiki and Django; other workloads behave similarly. The clones track the originals closely, achieving cosine similarity (SF_C) of 0.99–1.00 for branch predictor and 0.98–0.99 for prefetcher. The clones also preserve the *ranking* of design-knob values: for every workload, the best and worst designs with the clone are the same as with the original. This means an architect using the clone to compare prefetchers or branch predictors would reach the same conclusions as if running the original workload.

We also swept the cache-size and replacement-policy categories: L1-D and L1-I capacity over 16/32/48/64 KB, L2 over 256 KB–2 MB per core, and replacement across LRU, SRRIP, DRRIP, SHiP, and Hawkeye. Across all four, the clones reproduce the original's design-knob ranking exactly—the best and worst configurations always match—and achieve cosine similarity (SF_C) of 0.91 for L1-D, 0.94 for L1-I, 0.89 for L2, and 0.83 for cache replacement.

Production workloads. Porting the production services to

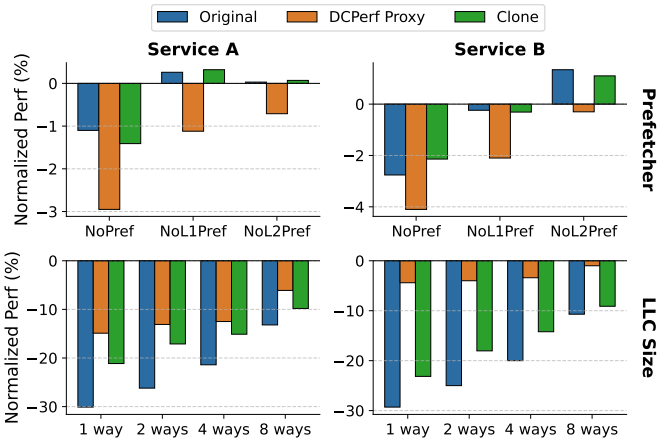

Fig. 10. Sensitivity matching for production workloads on real hardware under prefetcher and LLC-size variations.

a simulator is impractical, so we evaluate sensitivity on real hardware using two knobs: enabling/disabling hardware prefetchers and varying LLC capacity via Intel RDT [28] (the same methodology used in §II-B). Figure 10 compares the sensitivity of each production service, its corresponding DCPerf proxy, and the Fidelio clone. The Fidelio clones closely track the production services' sensitivity, while the DCPerf proxies diverge substantially. For prefetcher sensitivity, the clones achieve cosine similarity above 0.99 relative to the original, while the proxies drop to ∼0.80—and in some cases get the direction wrong entirely (e.g., Service B gains IPC when L2 prefetchers are disabled, but its proxy predicts a loss). The gap is even starker for LLC sensitivity: both production services degrade by ∼29–30% when restricted to a single LLC way, and the clones reproduce 70–79% of this degradation, while the proxies capture only 15–50%.

Guarding against overfitting. Agentic search risks overfitting to the knobs it tunes. We run a leave-out test: we remove the prefetcher and replacement-policy categories from the generation loop, so the agents never sweep or optimize for them. Then, we evaluate the resulting clone *blindly* on those two held-out categories. The metric here is sensitivity fidelity SF_C (Eq. 2), i.e., the cosine similarity between the clone's and the target's curves of relative IPC change across the swept values of a knob. It is a different quantity from the per-metric counter fidelity f_i reported in §IV-A, and the two are not on a comparable scale, since f_i scores a single scalar while SF_C scores the shape of a five-point response curve. On the held-out categories, the clone reaches SF_C of 0.93 for prefetcher and 0.81 for replacement, against 0.98 and 0.83 when those categories are tuned. In both cases it reproduces the design-knob ranking exactly (Kendall's $\tau = 1.0$): an architect using the held-out clone still selects the same best and worst prefetcher and the same best and worst replacement policy. Likewise, Counter fidelity is unaffected as seen with the held-out clone's IPC fidelity unchanged at 97%, so the movement in SF_C does not correspond to any regression in the counter metrics. This shows that the clone can reproduce sensitivity it was never optimized for. Optimizing for the retained categories

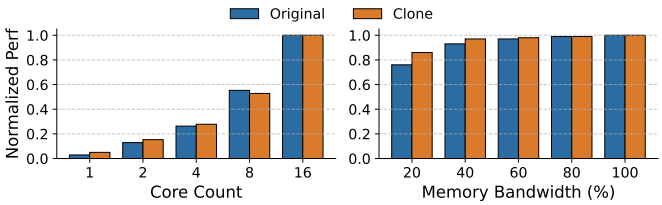

Fig. 11. Sensitivity to core count and memory bandwidth.

(e.g., cache size) already shapes the clone's code and data structures into a form amenable to exploration along the held-out knobs, so their sensitivity follows.

1) Sensitivity to Core Count: Core count is a consequential knob a vendor sets per generation, but it is not among the six categories in Table II. We use it to show that Fidelio's category set is *extensible*: adding a knob requires only a sweep definition and a playbook entry, leaving the closed loop unchanged. We add core count as a new category and sweep Mediawiki on real hardware by pinning it to $\{1, 2, 4, 8, 16\}$ cores via `cgroups`, taking the relative throughput change versus the 4-core baseline (Eq. 1) as the scaling curve. The playbook tunes thread-level scalability through shared-versus-private working set, contended atomics, and synchronization frequency. As shown in the left part of Figure 11, the Fidelio clone tracks Mediawiki's scaling curve with cosine similarity (SF_C) of 0.94 and an identical ranking of core counts (Kendall's $\tau = 1.0$), reproducing the near-linear scaling.

2) Sensitivity to Memory Bandwidth: Memory bandwidth is a first-order concern when services are bin-packed and compete for the memory subsystem, and like core count it is not in Table II. Adding it again requires only a sweep definition and a playbook entry. We sweep Mediawiki's available bandwidth on real hardware using Intel Memory Bandwidth Allocation (MBA). We throttle cores to $\{20, 40, 60, 80, 100\}\%$ of peak bandwidth and we take the relative IPC change versus the unthrottled baseline as the sensitivity profile. The playbook reuses Fidelio's working-set and streaming controls. As shown in the right part of Figure 11, the Fidelio clone tracks Mediawiki's bandwidth-throttling curve with cosine similarity of 0.89, reproducing the degradation as bandwidth is constrained.

C. Generation Latency

We measure the wall-clock time of generating each benchmark clone using Fidelio, computed from current API pricing for the Gemini model [22]. Across the four DCPerf workloads, Fidelio converges in 6–11 hours and 140–383 iterations. The variation reflects differences in prompt complexity: Video-Transcode converges fastest (140 iterations, 6 h) but its RAG-augmented prompts inject large C++ source fragments, while Django requires the most iterations (383, 11 h) but uses smaller prompts. The four production services cost modestly more, converging in 8–13 hours and 220–450 iterations, as their deeper software stacks and larger instruction footprints demand more refinement iterations and larger RAG contexts. Fidelio's few hours of wall-clock time contrast with the weeks or months of engineering effort required to manually develop

and tune proxy benchmarks such as DCPerf. This makes it practical to re-clone benchmarks on major service updates.

D. Sensitivity to Model Capability

All experiments so far use Gemini 2.5 Pro, the most capable but also the slowest, most expensive model available. We repeat the full cloning pipeline for Mediawiki using alternative models: Gemini 2.5 Flash and Gemini 2.5 Flash Light and Claude Opus 4.8. We also evaluate two open-source models: Qwen3-30B-A3B and Llama-3.3-70B, as well as the current frontier Gemini 3.1 Pro.

All configurations have a 5-hour budget. Claude Opus iterates under its own general-purpose agentic harness (Claude Code). It has shell access to the compiler and to `perf` and the same 5-hour budget, but without Fidelio's diagnostic playbooks, tiered fidelity model, retrieval-augmented seeding, and escalation policy. Figure 12 compares counter matching accuracy across models. Gemini 2.5 Pro achieves 92% mean fidelity across metrics. Flash Light (80%) outperforms Flash (72%) despite being a smaller model, because its lower per-call latency allows more iterations within the fixed time budget. This shows the value of Fidelio's iterative loop, where additional refinement cycles can compensate for weaker per-step reasoning. The gap relative to Pro is most pronounced on metrics that require nuanced code transformations: Flash drops to 58% on branch MPKI and 49% on BTB MPKI, where matching branch behavior requires careful control-flow reasoning that stronger models handle better. For cost-sensitive settings, Flash Light offers a good trade-off, achieving near-Pro accuracy on most metrics. Across families, Claude Opus 4.8 with the full Fidelio loop is the strongest non-Gemini configuration at 89% mean fidelity. On the other hand, the smaller open-source models are weaker (78% mean fidelity for Qwen3 and 64% for Llama-3.3) and badly miss individual frontend metrics. Importantly, we observe that Claude Opus 4.8 *without* Fidelio reaches only 59% mean fidelity, confirming that structured, microarchitecture-aware refinement, not raw model capability, is what produces accurate clones.

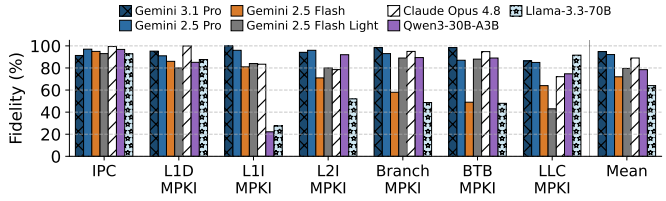

Fig. 12. Counter matching accuracy across different LLMs.

Upgrading to the latest frontier (as of August 2026), Gemini 3.1 Pro raises mean fidelity on Mediawiki from 92% to 94.8% (+2.8) points, and the other workloads behave similarly. By comparison, running Claude Opus 4.8 inside Fidelio's full loop rather than in a standalone setup alone increases mean fidelity from 59% to 89% (+30) points. The gains land where weaker models fail, i.e., on the structural, control-flow-dependent metrics: L1-I MPKI reaches 99.9%, L2-I 94.2%, and branch and BTB MPKI both 98.4%, against the 58% and 49% that Flash manages on branch and BTB. The residual error instead shifts to IPC (91.3%) and LLC MPKI (86.3%). This is consistent with the rest of our results: these are the two *emergent* metrics, which no single code transformation controls directly and which respond to more search rather than to better per-step reasoning. The comparison also puts the contribution of the harness in perspective.

Model capability is improving quickly, but on this task the structure wrapped around the model still dominates it. We expect Fidelio to keep compounding with new frontier models.

E. Ablation Study

To isolate the contribution of each component, we run: *(i)* a plain LLM (Gemini 2.5 Pro), where the model generates benchmark code in a single pass; *(ii)* an LLM augmented with the structured prompting from our diagnostic playbooks; and *(iii)* the full Fidelio pipeline.

Figure 13 shows the results for the Mediawiki application. The plain LLM baseline performs poorly: the generated code frequently fails to compile or crashes at runtime, and even for the runs that succeed, fidelity on the critical metrics is low. For instance, IPC, I-cache MPKI, and branch MPKI, all stay below 20%. Some metrics such as L1-D MPKI land near the target by coincidence, but the metrics that require deliberate code structure are far off. This confirms that one-shot code generation, even with a frontier LLM, cannot reliably produce programs with specific microarchitectural behavior. The reason is that the interaction between code structure and hardware is too complex for open-loop synthesis. Augmenting the LLM with structured playbook prompting yields a substantial improvement, raising IPC fidelity from 10% to 73% and I-cache MPKI from 10% to 68%, as the microarchitecture-aware guidance helps the model select appropriate code transformations. Without closed-loop measurement and iterative refinement, the model cannot detect and correct regressions across metrics, and the final fidelity plateaus below the convergence thresholds. Fidelio combines structured reasoning with iterative measurement, retrieval-augmented seeding, and escalation, achieving the highest accuracy across all metrics.

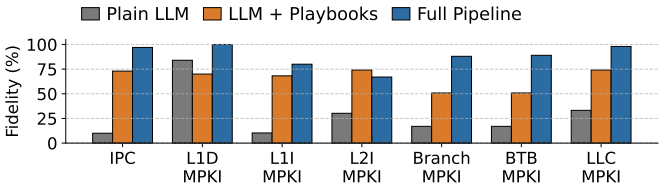

Fig. 13. Ablation study comparing plain LLM generation, LLM with structured prompting, and full Fidelio.

F. Cross-Platform Generalization

To test portability, we take the Mediawiki clone generated on Intel EMR and run it on AMD TRN without any re-tuning. Figure 14 shows that fidelity remains high on both platforms. On Intel, the clone achieves 97% IPC fidelity, 80% L1I MPKI, 88% branch MPKI, and 98% LLC MPKI. On AMD, despite the different cache hierarchy, branch predictor, and frontend design, fidelity stays within 5–10 points of the Intel results.

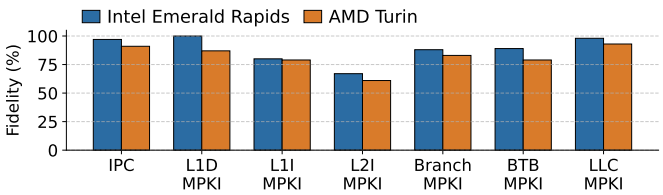

Fig. 14. Cross-platform generalization: the Mediawiki clone generated on Intel EMR, evaluated on AMD TRN.

This portability arises because Fidelio's clones are written in portable C/C++ and target microarchitectural properties rather than vendor-specific mechanisms.

G. Comparison with Ditto

We compare Fidelio against Ditto [45], the state-of-the-art statistical cloning system. We run Ditto on the same four DCPerf workloads and measure counter fidelity using the same metrics. Ditto achieves a median per-metric accuracy of 54% on Mediawiki (its best case) and 39% on VideoTranscode (its worst case), with Django and Feedsim falling in between. On Mediawiki, Ditto reaches 82% on L1 I-cache MPKI but only 36% on L2 I-cache and 58% on branch MPKI. By contrast, Fidelio achieves a median accuracy of 89% on Mediawiki and 99% on VideoTranscode. The gap is largest on the metrics that depend on structural code properties—branch prediction, instruction fetch, and deep cache hierarchy behavior—which Ditto's aggregate statistical profiles cannot capture.

Beyond accuracy, Ditto's profiling requires intrusive instrumentation (Valgrind, Intel SDE, SystemTap) that drastically distorts runtime behavior: in Feedsim an average request grows from 0.68 s to 28.6 s under Intel SDE and 87.0 s under Valgrind (a 128× slowdown), which triggers networking timeouts and makes live production profiling infeasible. Ditto also requires substantial manual per-workload setup of parameters and configurations, whereas Fidelio runs fully automatically at under 5% profiling overhead.

V. Related Work

Datacenter benchmark suites. Conventional datacenter benchmark suites such as CloudSuite [18], TailBench [34], DeathStarBench [19] and DCPerf [61] remain the primary vehicles for processor evaluation. These suites are manually developed and tuned over months or years of engineering effort to approximate datacenter workload behavior. While invaluable, they suffer from the proxy fragility problem (§II): once the target service's software stack evolves, the static proxy benchmark drifts, and no automated mechanism exists to re-align it. Fleetbench [1] pursues fleet representativeness at a different granularity. It assembles microbenchmarks for the primitives that dominate datacenter cycles, e.g., compression or (de)serialization, and drives them with input distributions harvested from a production fleet. Our work complements these suites. Fidelio can regenerate clones on demand, keeping the benchmark current with the workload it represents.

Synthetic workload generation and cloning. Prior work has explored automated construction of representative workloads through statistical modeling and synthetic trace generation. Microarchitectural cloning techniques [14], [32] distill workload characteristics into compact kernels by matching instruction mix, branch behavior, and memory access patterns using hand-crafted code templates. Others synthesize miniaturized multicore proxies from parallel-pattern characterization [17] or, when a similar open-source program exists, generate a matching *dataset* so that the program reproduces the target's microarchitectural profile [41]. Some approaches use genetic programming [9] or constraint-driven synthesis to produce stress tests targeting specific hardware bottlenecks. These methods typically optimize for a fixed set of counters on a single configuration and do not ensure that the generated code responds to microarchitectural design changes in the same way as the target. Fidelio explicitly optimizes for sensitivity matching through its two-phase convergence loop.

LLMs and agentic AI for code and systems. LLMs have been applied to code generation [15], [24], [43], program repair [67], compiler optimization [16], database tuning [40], debugging [42], and hardware description generation [64], among other use cases. Multi-agent coding frameworks such as SWE-agent [68] and MetaGPT [27] coordinate LLM agents around software engineering tasks, optimizing for functional correctness via test suites [31], while ECO-LLM [47] refactors existing code by matching historical performance anti-patterns. These efforts target either functional correctness or single-objective performance, and operate in open-loop or narrowly scoped feedback loops. Fidelio differs by targeting multi-dimensional, hardware-dependent fidelity within a closed-loop pipeline that combines measurement, retrieval-augmented generation, tiered optimization, and escalation between parameter tuning and structural transformations.

Agentic kernel generation and optimization. A growing body of work uses LLM agents to generate and optimize compute kernels under measured performance feedback. Systems such as KernelEvolve [46], the AI CUDA Engineer [39], GEAK [66], and Kevin [10] iteratively edit kernels, using runtime or profiler signals as a reward to converge toward faster implementations. Fidelio shares this closed-loop, feedback-driven structure but pursues a different objective: rather than maximizing the performance of a single kernel, it synthesizes benchmarks that *match* a target's microarchitectural counter profile and its sensitivity to design-knob changes.

VI. Conclusion

This paper presented Fidelio, an agentic AI framework that automatically synthesizes datacenter benchmark clones from lightweight hardware telemetry, without access to source code or binaries. Fidelio converges in two phases: *Counter Matching* rapidly aligns a tiered, weighted set of performance counters on real hardware, while *Sensitivity Matching* validates and repairs the clone's response to microarchitectural design knobs. Our evaluation on DCPerf benchmarks and production workloads showed high-fidelity cloning across hardware structures, with clones that faithfully preserve sensitivity to microarchitectural design changes.

REFERENCES

[1] A. Abel, Y. Li, R. O'Grady, C. Kennelly, and D. Gove, "A Profiling-Based Benchmark Suite for Warehouse-Scale Computers," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '24)*, 2024.

[2] AMD, "5th Generation AMD EPYC™ Server CPUs," https://www.amd.com/en/products/processors/server/epyc/9005-series.html, 2026.

[3] ——, "Data Center Solutions from AMD," https://www.amd.com/en/solutions/data-center.html, 2026.

[4] ARM, "Maximize Cloud Computing Performance with Unmatched Efficiency," https://www.arm.com/markets/computing-infrastructure/cloud-computing, 2026.

[5] Z. Asgar, M. Nguyen, and S. Katti, "Efficient and Scalable Agentic AI with Heterogeneous Systems," 2025. [Online]. Available: https://arxiv.org/abs/2507.19635

[6] M. Ashare, "Cloud, Enterprise Spend on Data Center Gear Spiked in 2024," https://www.ciodive.com/news/cloud-enterprise-data-center-infrastructure-spend-generative-ai/737049/, 2024.

[7] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory Hierarchy for Web Search," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '18)*, 2018.

[8] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "AsmDB: understanding and mitigating front-end stalls in warehouse-scale computers," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19, 2019, p. 462–473.

[9] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu, and M. D. Wong, "BOOM-Explorer: RISC-V BOOM Microarchitecture Design Space Exploration Framework," in *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD'21)*, 2021, pp. 1–9.

[10] C. Baronio, P. Marsella, B. Pan *et al.*, "Kevin: Multi-Turn Reinforcement Learning for Writing CUDA Kernels," arXiv preprint arXiv:2507.11948, 2025, https://arxiv.org/abs/2507.11948.

[11] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*, 2013. [Online]. Available: http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[12] S. Beamer, K. Asanović, and D. Patterson, "The GAP Benchmark Suite," 2017. [Online]. Available: https://arxiv.org/abs/1508.03619

[13] P. Belcak, G. Heinrich, S. Diao, Y. Fu, X. Dong, S. Muralidharan, Y. C. Lin, and P. Molchanov, "Small Language Models are the Future of Agentic AI," 2025. [Online]. Available: https://arxiv.org/abs/2506.02153

[14] R. H. Bell and L. K. John, "Improved automatic testcase synthesis for performance model validation," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05, 2005.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[16] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, and H. Leather, "Large Language Models for Compiler Optimization," 2023. [Online]. Available: https://arxiv.org/abs/2309.07062

[17] E. Deniz, A. Sen, B. Kahne, and J. Holt, "MINIME: Pattern-aware multicore benchmark synthesizer," *IEEE Transactions on Computers*, vol. 64, no. 8, 2015.

[18] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, 2012.

[19] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[20] K. Ganesan and L. K. John, "Automatic Generation of Miniaturized Synthetic Proxies for Target Applications to Efficiently Design Multicore Processors," *IEEE Trans. Comput.*, p. 833–846, Apr. 2014.

[21] A. Gonzalez, A. Kolli, S. Khan, S. Liu, V. Dadu, S. Karandikar, J. Chang, K. Asanovic, and P. Ranganathan, "Profiling Hyperscale Big Data Processing," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[22] Google, "Gemini Developer API pricing," https://ai.google.dev/gemini-api/docs/pricing, 2026.

[23] ——, "Gemini Models," https://ai.google.dev/gemini-api/docs/models, 2026.

[24] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence," 2024. [Online]. Available: https://arxiv.org/abs/2401.14196

[25] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H.-H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang, "The Architectural Implications of Facebook's DNN-Based Personalized Recommendation," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'20)*, 2020.

[26] R. Hebbar S R and A. Milenković, "SPEC CPU2017: Performance, Event, and Energy Characterization on the Core i7-8700K," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19)*, 2019.

[27] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber, "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework," 2024. [Online]. Available: https://arxiv.org/abs/2308.00352

[28] Intel, "Intel RDT Software Package," https://github.com/intel/intel-cmt-cat/tree/master, 2024.

[29] ——, "Intel® Xeon® Gold 5512U Processor," https://www.intel.com/content/www/us/en/products/sku/237565/intel-xeon-gold-5512u-processor-52-5m-cache-2-10-ghz/specifications.html, 2025.

[30] ——, "Intel® Data Center Solutions," https://www.intel.com/content/www/us/en/data-center/overview.html, 2026.

[31] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," *ACM Trans. Softw. Eng. Methodol.*, Jan. 2026.

[32] A. Joshi, L. Eeckhout, R. H. Bell, and L. John, "Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks," in *Proceedings of the IEEE International Symposium on Workload Characterization*, 2006, pp. 105–115.

[33] 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*, ser. ISCA '15, 2015.

[34] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'16)*, 2016, pp. 1–10.

[35] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-SPY: Context-Driven Conditional Instruction Prefetching with Coalescing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '20)*, 2020.

[36] T. A. Khan, D. Zhang, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA '21)*, 2021.

[37] J. Kim, B. Shin, J. Chung, and M. Rhu, "The Cost of Dynamic Reasoning: Demystifying AI Agents and Test-Time Scaling from an AI Infrastructure Perspective," 2026. [Online]. Available: https://arxiv.org/abs/2506.04301

[38] A. Kleen, "An introduction to last branch records," https://lwn.net/Articles/680985/, 2016.

[39] R. T. Lange, Q. Sun, A. Prasad, M. Faldor, Y. Tang, and D. Ha, "Towards Robust Agentic CUDA Kernel Benchmarking, Verification, and Optimization," 2025. [Online]. Available: https://arxiv.org/abs/2509.14279

[40] J. Lao, Y. Wang, Y. Li, J. Wang, Y. Zhang, Z. Cheng, W. Chen, M. Tang, and J. Wang, "GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization," *Proc. VLDB Endow.*, vol. 17, no. 8, p. 1939–1952, Apr. 2024. [Online]. Available: https://doi.org/10.14778/3659437.3659449

[41] H. R. Lee and D. Sanchez, "Datamime: Generating Representative Benchmarks by Automatically Synthesizing Datasets," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[42] K. H. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, "ChatDBG: Augmenting Debugging with Large Language Models," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, p. 1892–1913, Jun. 2025. [Online]. Available: http://dx.doi.org/10.1145/3729355

[43] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "StarCoder: may the source be with you!" 2023. [Online]. Available: https://arxiv.org/abs/2305.06161

[44] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 106–109, 2020.

[45] M. Liang, Y. Gan, Y. Li, C. Torres, A. Dhanotia, M. Ketkar, and C. Delimitrou, "Ditto: End-to-End Application Cloning for Networked Cloud Services," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023, 2023.

[46] G. Liao, H. Qin, Y. Wang, A. Golden, M. Kuchnik, Y. Yetim, J. J. Ang, C. Fu, Y. He, S. Hsia, Z. Jiang, D. Li, U. Pashkevich, V. Puvvada, F. Shi, M. Steiner, R. Xiao, N. Yan, X. Yu, Z. Fang, R. Levenstein, K. Ho, H. Zhu, A. Hammond, R. Li, A. Mathews, K. Gondkar, A. Zainul-Abedin, K. Singh, H. Yu, W. Chi, B. Huang, S. Zhang, N. Weller, Z. Marine, W. Cook, C.-J. Wu, and G. Liu, "KernelEvolve: Scaling Agentic Kernel Coding for Heterogeneous AI Accelerators at Meta," 2026. [Online]. Available: https://arxiv.org/abs/2512.23236

[47] H. Lin, M. Maas, M. Roquemore, A. Hasanzadeh, F. Lewis, Y. Simonson, T.-W. Yang, A. Yazdanbakhsh, D. Altinbüken, F. Papa, M. N. Edmonds, A. Patil, D. Schwarz, S. Chandra, C. Kennelly, M. Hashemi, and P. Ranganathan, "ECO: An LLM-Driven Efficient Code Optimizer for Warehouse Scale Computers," 2025. [Online]. Available: https://arxiv.org/abs/2503.15669

[48] Meta, "WDLBench," https://github.com/facebookresearch/DCPerf/tree/main/packages/wdl_bench, 2025.

[49] Meta, "ICache Buster," https://github.com/facebookresearch/DCPerf/blob/main/packages/feedsim/third_party/src/workloads/search/gen_icache_buster.py, 2026.

[50] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals-Yúfera, and A. Ros, "Berti: an Accurate Local-Delta Data Prefetcher," in *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22)*, 2022.

[51] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: instruction pointer classifier-based spatial hardware prefetching," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA '20, 2020, p. 118–131.

[52] R. Panda and L. K. John, "Proxy benchmarks for emerging big-data workloads," in *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[53] G. S. Ravi, R. Bertran, P. Bose, and M. Lipasti, "MicroGrad: A Centralized Framework for Workload Cloning and Stress Testing," 2020. [Online]. Available: https://arxiv.org/abs/2009.04622

[54] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, 2011.

[55] Y. Shavit, S. Agarwal, M. Brundage, S. O'Keefe, R. Campbell, T. Lee, P. Mishkin, T. Eloundou, A. Hickey, K. Slama, L. Ahmad, P. McMillan, A. Beutel, A. Passos, and D. G. Robinson, "Practices for Governing Agentic AI Systems," https://api.semanticscholar.org/CorpusID:266312974.

[56] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[57] J. Stojkovic, A. Farrell, Z. Gong, C. J. Hughes, and J. Torrellas, "AccelFlow: Orchestrating an On-Package Ensemble of Fine-Grained Accelerators for Microservices," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA '26)*, 2026.

[58] J. Stojkovic, C. Liu, M. Shahbaz, and J. Torrellas, "μManycore: A Cloud-Native CPU for Tail at Scale," in *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23)*, 2023.

[59] ——, "HardHarvest: Hardware-Supported Core Harvesting for Microservices," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture (ISCA'25)*, 2025.

[60] J. Stojkovic, C. Zhang, I. Goiri, E. Choukse, H. Qiu, R. Fonseca, J. Torrellas, and R. Bianchini, "TAPAS: Thermal- and Power-Aware Scheduling for LLM Inference in Cloud Platforms," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2025, p. 1266–1281.

[61] W. Su, A. Dhanotia, C. Torres, J. Gandhi, N. Gholkar, S. Kanaujia, M. Naumov, K. Subramanian, V. Andrei, Y. Yuan, and C. Tang, "DCPerf: An Open-Source, Battle-Tested Performance Benchmark Suite for Datacenter Workloads," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25, 2025.

[62] P. Sun, "Announcing ScaNN: Efficient Vector Similarity Search," https://research.google/blog/announcing-scann-efficient-vector-similarity-search/, 2020.

[63] Synergy Research Group, "Hyperscale Market Tracker," https://www.srgresearch.com/research/hyperscale-market-tracker, 2025.

[64] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "VeriGen: A Large Language Model for Verilog Code Generation," 2023. [Online]. Available: https://arxiv.org/abs/2308.00708

[65] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, 2017, p. 1041–1052.

[66] J. Wang, V. Joshi, S. Majumder, X. Chao, B. Ding, Z. Liu, P. P. Brahma, D. Li, Z. Liu, and E. Barsoum, "Geak: Introducing Triton Kernel AI Agent & Evaluation Benchmarks," 2025. [Online]. Available: https://arxiv.org/abs/2507.23194

[67] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-Trained Language Models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1482–1494. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00129

[68] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering," 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[69] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," 2023. [Online]. Available: https://arxiv.org/abs/2210.03629