HydraKV: Adversarial AI Discovery of a Larger-than-Memory Key-Value Store that Outperforms FASTER on Skewed YCSB-A

Koushik Sen
EECS Department, UC Berkeley
ksen@berkeley.edu

Abstract

We report on HydraKV, a larger-than-memory key-value store for skewed point-operation workloads. The setting is fully specified: YCSB-A [2] (50% reads, 50% blind upserts, Zipfian $\theta = 0.95$) over 250M keys with 100-byte values ($\sim$25 GB of data) under a hard 8 GiB memory budget, on a 64-vCPU machine with eight local NVMe SSDs in RAID0. On this configuration Microsoft's FASTER [1], a state-of-the-art larger-than-memory store built for this kind of point-operation workload, sustains 0.93 Mops/s with the same harness. HydraKV sustains 5.50 Mops/s on the same workload, a 5.9$\times$ speedup (runs 5.50 / 5.50 / 5.52), about 89% of the miss-bandwidth bound implied by the measured device ceiling and cache-hit rate, and 3.87–5.67 Mops/s across four adversarial workload variants (sparse SplitMix64 [14] key spaces, clustered and drifting hot sets motivated by published workload studies [4, 5], and a mixed holdout). HydraKV is $\sim$4,470 lines of dependency-free C++17: an append-only O_DIRECT log with CRC-32C-sealed, LSN-ordered slots, a fingerprint hash index, an admission-controlled write-back cache [8, 7], a raw `io_uring` [13] read-miss path, scan-based crash recovery, fail-soft I/O error handling, background log compaction with hole punching and position reuse, disk-backed multi-slot records for oversized values, and a non-blocking, crash-durable delete path (an intent log makes an acknowledged delete survive a process kill; tombstoning drains through a background reaper). None of these mechanisms is individually new; the contribution is their composition under a hard memory budget, and the evidence behind it.

Correctness and robustness rest on an end-to-end test suite run across six build and sanitizer configurations, real fault injection (disk-full, SIGKILL, external truncation), a sweep of every operation mix, value-size regime, and failure path the harness can generate, and three independent audits: the second re-ran the engine on the reference hardware and verified the scored result as genuine and unhacked (5.51 Mops/s median, in budget), and the third demonstrated that our remediation of the second had itself introduced a delete crash-resurrection defect, since repaired and re-verified at an unchanged score. We describe the engine's design, the evaluation and its scope limits, and the measured throughput ceiling that placed mid-task goals of 10 Mops/s (and a later 7 Mops/s stretch) out of reach for this design on this machine. HydraKV was designed, implemented, and tested almost entirely by an AI agent framework [27] through a sequence of natural-language tasks; the development record is reproduced in the paper.

Preprint.

1 Introduction

Point-operation key-value stores sit under much of today's internet infrastructure: session stores, feature stores, ML embedding services, social-graph caches [6, 4, 5]. The economically interesting regime is *larger-than-memory with a hard RAM budget*: the data set (here $\sim$25 GB) is a few times bigger than the memory the operator is willing to give the store (here 8 GiB), the SSD is fast but not free, and the access pattern is heavily skewed (Zipfian $\theta = 0.95$, so a small hot set dominates). Update-heavy skew is the setting YCSB workload A models [2] and the setting FASTER [1] was engineered for: FASTER's hybrid log gives in-place updates for the in-memory hot tail and spills the cold majority to storage, and its paper reports throughput up to 160M ops/s when the working set fits in memory. When the data does not fit (250M keys, 100-byte values, an 8 GiB budget, reads that must return verbatim stored bytes), FASTER on our reference machine sustains 0.93 Mops/s. That number is the baseline this paper's artifact was asked to beat.

HydraKV was produced by KISS Sorcar [27], an open-source agent framework for long-horizon software engineering, driven by seven successive natural-language tasks that are reproduced verbatim in Section 6. Two of the tasks define the performance problem (beat FASTER under the memory budget); the other five demand correctness and robustness under a throughput floor. Because an optimizing agent scored against a single trace will overfit it, a special case of reward hacking [24] that evaluator-guided program search and prompt optimization face structurally [23, 22, 28, 29], development followed an *adversarial discovery* loop: the agent alternated between generating literature-informed workload variants intended to break the current engine and repairing the engine until it met the goal on all of them, finishing with a held-out variant used only for measurement. Correctness was enforced by end-to-end tests under sanitizers, cross-model review, fault injection, and three independent audits; the second and third re-ran the engine on the reference hardware and probed deployment behaviors the benchmark cannot reach, and the third caught a regression that the response to the second had introduced. Section 6 documents the process, the engine's evolution across the seven tasks, and the observations we draw from it. The rest of the paper describes the engine as it stands.

HydraKV is $\sim$4,470 lines of self-contained C++17 (no dependencies beyond pthreads and raw Linux syscalls). Its architecture (Section 4) is a deliberately conservative composition of known ideas. Values live in an append-only 128-byte-slot log on O_DIRECT storage, in the family of log-structured designs [10, 1], each slot sealed with a 56-bit log sequence number (LSN) and a CRC-32C. A cache-line-bucketed fingerprint hash index with lock-free CAS insertion plays a role similar to FASTER's hash table but maps keys to positions in the log; after a restart the index is rebuilt Bitcask/FASTER-style [15, 1] by a CRC-verified, newest-LSN-wins scan of the log. A set-associative write-back RAM cache uses SLRU-style segmented second-chance insertion [8], guarded by a probabilistic doorkeeper bitmap borrowed from TinyLFU [7]. Read misses go through an asynchronous path built on hand-rolled `io_uring` rings [13] (the target machine has no liburing). A background compactor relocates live slots, punches reclaimed regions out of the file, and recycles their position space. I/O errors, including disk-full, are absorbed fail-soft with sticky error counters. The engine contains no constant derived from the workload's key count, key density, skew parameter, or trace files. Its cache-entry geometry does, however, assume the benchmark interface's small-value regime (Section 4).

On the original workload HydraKV sustains 5.50 Mops/s (median of three cold-cache 30-second runs, 5.50 / 5.50 / 5.52, or 5.9$\times$ FASTER's 0.93) while staying under the enforced 8 GiB resident cap, and passes an untimed read-back validation of all 250,000,000 keys. On the four adversarial variants (sparse keys, clustered hot sets, drifting hot sets, and a mixed held-out workload generated with fresh seeds after engine work stopped), the engine revision that closed the discovery loop ("v4c") sustained 5.67, 5.56, 3.97, and 3.87 Mops/s (Section 5). The scored variant matrix was not re-run with later revisions. A roofline-style argument (Section 5.3) shows the original-workload number sits close to the bound implied by the measured device ceiling (718k 4 KiB random-read IOPS) and the 77% cache-hit rate observed across every admission policy tried. The same arithmetic is why a mid-task goal of 10 Mops/s was, in our judgment, out of reach for this class of design on this machine. The engine opens its log O_DIRECT so that its disk traffic cannot circumvent the memory budget by riding the OS page cache.

Contributions.

1. **An artifact.** A ∼4,470-line, dependency-free, larger-than-memory KV store that outperforms FASTER by 5.9× on one demanding, fully specified YCSB-A configuration, and that ships with the full set of deployment subsystems described in Sections 4.1 and 4.2, from crash recovery to the observability surface, with the engine, variant generator, test suite, audits, and discovery ledger released alongside the paper.[1]
2. **A verification protocol.** The apparatus that stands behind the artifact's correctness and robustness claims: end-to-end workload tests across six build configurations (including ASan/UBSan, a TSan subset, and an `io_uring`-disabled fallback), real fault injection (disk-full, SIGKILL, external truncation), adversarial workload variants terminated by a held-out variant, three independent audits (one code-level, two re-running the engine on the reference hardware with their own oracles: one verified the headline result as genuine and unhacked, the next demonstrated with self-asserting repro programs a crash-durability regression our previous remediation had introduced), an all-workload sweep (seven operation mixes including delete-heavy and write-only, oversized and bimodal values, tiny budgets, crash/restart in every mode), and throughput no-regression gates verified against same-day A/B controls.
3. **A measured ceiling.** A quantitative argument for why 10 Mops/s was unreachable for this design on this hardware, and a description of the reward-hacking opportunities (page-cache exploitation, value regeneration, trace precomputation) that the task specification and the audits closed off.

Non-contributions and scope. HydraKV contains no individually novel data structure or replacement policy. It carries the subsystems a deployable store needs, chief among them log compaction and crash recovery, both core features of RocksDB [11] and FASTER (Sections 4.1 and 4.2), and it has been exercised across every workload the harness generates. What remains short of a deployable store is disclosed in Section 7: checkpoint-bounded durability, scan-time recovery, a list of residual risks none of which is reachable by the scored workload, and the absence of any long-duration endurance campaign. All absolute throughput numbers are specific to one machine, one harness, and one engine version each (Table 1). We compare against FASTER only on the original workload, the one configuration where both systems were measured. The claim we defend is narrower than "a better KV store": that on this fully specified configuration, a conservative composition of known mechanisms outperforms the natural baseline while carrying the robustness subsystems a store needs, with released evidence for both halves of that claim.

2 Problem Setting

The benchmark is fixed by a harness the engine may not modify. Each operation is a read or a blind upsert with equal probability (YCSB-A [2], unseeded RNG). Keys are drawn Zipfian with $\theta = 0.95$ from 250,000,000 unique 64-bit keys. Values are exactly 100 bytes, an 8-byte little-endian counter plus random bytes, stored verbatim. A load phase inserts all 250M keys (untimed). Run-phase keys are a subset of loaded keys, so reads never miss at the key level, and run-phase upserts overwrite existing keys only: the keyspace stays exactly 250M keys during the scored window. The scored phase is a fixed 30.0-second window driven by 16 worker threads pinned to one NUMA node, with reads issued asynchronously up to 256 in flight per worker. The rest of the scored environment is pinned by the specification (completion polling every 64 operations, load-key validation off, bimodal values off, fast exit off, no operation-count caps), and any deviation invalidates a run. The score is the median of three runs, with `drop_caches` before each, so every scored run starts from a cold cache. A run that exits nonzero, hangs, is killed by the out-of-memory killer, or exceeds the resident cap scores zero and is not retried.

Correctness is enforced by three separate instruments, with different strengths. A separate untimed validation run reads back all 250,000,000 loaded keys and verifies full values against FNV-1a-32 checksums. During scored runs, an auditor thread performs randomized cross-thread read-after-write checks (its operations are not counted toward throughput, and it runs identically for every engine,

[1] `projects/kv_adversarial/` in `https://github.com/ksenxx/kiss_ai`: `hydra.cc` (engine), `gen_variant.cc` (trace generator), `test_hydra.cc` and `run_all_tests.sh` (test matrix), `HYDRA_PROD_AUDIT.md` (first independent audit), `PROD_READINESS.md` (audit→fix→test map), `AUDIT2.md` and `AUDIT2_FIXES.md` (second independent audit and its remediation), `AUDIT3_FIXES.md` (third independent audit's findings and their remediation), `WORKLOAD_HARDENING.md` (all-workload campaign), `DISCOVERY_LOG.md` (ideas ledger), `benchmark/` (the complete benchmark kit from the reference node), and `refnode_rerun_jul21/`, `refnode_workloads_jul21/`, `refnode_audit3_jul21/` (raw reference-node evidence logs and review reports).

including the FASTER baseline), and the harness verifies the 8-byte counter of every read. The harness documentation is explicit that the random suffix bytes of *run-phase* overwrites are not re-verified during scoring (only their counters are, plus all load-phase values in validation), and equally explicit that exploiting this (e.g., regenerating or compressing values) is forbidden and treated as an integrity failure. The engine's own test suite uses deterministic full-value oracles to cover this blind spot at smaller scales.

The specification also sets a bar beyond the score. It asks for a real, deployable, production-grade store (no crashes, hangs, leaks, or corruption over long runs) and states that a fast number that is not correct and robust does not count. Durability is exempt from the throughput score, since `Checkpoint` may by specification be a no-op for scoring, but is expected of anything presented as prod-grade. The specification further forbids, at run time: network access, reading outside the task directory, storage outside the provided spill directory, offline precomputation from the trace files, and returning wrong bytes. The audits of Section 5.4 cover the engine-runtime items (no trace reads, no key-population constants, spill-directory-only I/O). The development process itself ran over SSH and searched the literature, a scoping discussed in Section 6. Section 7 assesses HydraKV against the prod-grade bar.

The machine is a GCP `n2-standard-64` (2× Cascade Lake Xeon, 64 vCPUs, 251 GB RAM) with eight 375 GB local NVMe SSDs in RAID0 (ext4, 512 KiB chunk), running Ubuntu 22.04.5 on kernel 6.8. Engines are compiled with g++ 11.4 (`-O3 -march=native -DNDEBUG -flto=auto -std=c++17`) and are restricted by the specification to C++17 and pthreads, with no external dependencies. The engine's resident data is capped at 8 GiB (runs that exceed it score zero), and the whole process runs in a 14 GiB cgroup with swap disabled, the headroom covering the harness's own key and checksum arrays. Since ∼25 GB of live data cannot fit in 8 GiB, the engine must spill to SSD and cache a hot subset. The RAID0 device's random-read limits, measured with `fio` (4 KiB O_DIRECT random reads at high queue depth on the RAID device), are 718k IOPS and 2.8 GB/s at ∼350 μs latency. The filesystem's 4096-byte logical block size makes 4 KiB the smallest read the engine issues. FASTER, run on this machine with this harness at the same budget, sustains 0.93 Mops/s. That is the baseline used throughout. We report FASTER as configured by the benchmark's maintainers for this harness and did not tune it ourselves, a caveat that applies symmetrically: neither engine was tuned by the other's authors.

3 State of the Art and Its Limitations

LSM-trees. RocksDB [11] and its LSM [10] relatives are the workhorses of persistent KV deployment. The Facebook study [4] characterizes three of its production use cases in detail. LSMs earn their complexity on write-heavy traffic with range scans. For pure point operations on NVMe, however, the machinery becomes overhead: KVell [3] showed that on modern NVMe SSDs, where random and sequential bandwidth are comparable, LSM and B-tree stores become *CPU*-bound on compaction, sorting, and synchronization, and that an unsorted, shared-nothing design with batched device access recovers 2–5× throughput. WiscKey [17] had already shown the intermediate step: separating values out of the LSM into their own log, keeping only keys sorted, sharply cuts the tree's I/O amplification on SSDs. HydraKV takes value separation to its endpoint for this workload: with no scans to serve, the sorted key structure is dropped too, leaving only the value log and a hash index. Our workload has no scans, so sorted storage buys nothing.

Hash-index log stores. The lineage HydraKV actually belongs to is older and simpler than the LSM: an append-only value log under an in-memory hash index, recovered by scanning the log. Bitcask [15] is the canonical statement of that design, motivated at Riak by the same goals this benchmark scores (point-operation latency and throughput, datasets larger than RAM, crash friendliness, and a code base simple enough to trust); its keydir, merge compaction, and scan-based recovery are the direct ancestors of HydraKV's index, compactor, and `recover_log()`. Bitcask's known cost is that the full-key index must fit in RAM. SILT [16] attacked precisely that cost, combining partial-key (fingerprint) hashing with entropy-coded indexes to reach 0.7 bytes of DRAM per key and ≈1.01 flash reads per lookup. HydraKV's fingerprint index and one-4 KiB-read miss path sit between the two: SILT-style compact fingerprints, but mutable, lock-free, and disk-verified rather than multi-store and mostly-static, because this workload is 50% blind upserts. uDepot [18] carried the hash-over-log design to fast NVMe with a two-level index that grows with the inserted items and a task-based asynchronous I/O runtime, arguing, as KVell did, that the store must be built around the device's queue

depths; HydraKV's hand-rolled `io_uring` rings are the same argument made with a newer kernel interface. None of these systems, however, couples the log to an *admission-controlled* RAM cache, which is where HydraKV differs (Section 4). FASTER [1] is the natural baseline: a cache-optimized concurrent hash index over a hybrid log whose in-memory tail supports in-place updates, with epoch-based synchronization. Its headline numbers are for in-memory working sets. Larger-than-memory reads go through asynchronous disk I/O and an admit-on-miss read-cache region. One structural limitation is documented in its own design: records fetched from disk are admitted unconditionally, so the one-touch tail of a Zipf distribution continuously evicts hot entries. Admission control is absent. The FASTER lineage's own follow-up, F2 [26], confirms the diagnosis from the inside: it names high indexing and compaction overheads and the mismanagement of distinct read-hot and write-hot working sets as FASTER's failure modes on this same class of workload (point operations, working sets much larger than memory, natural skew), and restructures the store around separate hot and cold tiers to fix them. HydraKV, built without consulting it, converged on the same problem statement with a different mechanism: one log, with heat separation done by the admission-controlled cache instead of by tiering the log itself. We compare only against the FASTER version the benchmark ships; F2 was not measured on this harness. Beyond FASTER's documented admission gap we can only offer hypotheses for the measured gap, since we did not instrument FASTER's internals on this harness: candidate costs include index and log-page metadata competing with cached values for the 8 GiB budget at 100-byte records, and per-pending-operation completion overhead on the miss path. We flag these as unverified; only the baseline itself, 0.93 Mops/s, is a measurement.

Caching engines. Admission and scan resistance have been studied most thoroughly in the caching literature: segmented LRU [8], CLOCK-style second-chance recency [9], TinyLFU admission with its doorkeeper filter [7], and the engineering synthesis in CacheLib [6]. The recent turn in that literature is toward radical simplicity on the same problem: S3-FIFO [19] filters one-hit wonders with a small probationary FIFO plus a ghost queue, and SIEVE [20] gets state-of-the-art miss ratios from a single lock-free CLOCK-like hand. HydraKV's memory tier is an independent instance of the same convergence, arrived at by measurement rather than trace studies: a one-bit second-chance doorkeeper-guarded design matched or beat every heavier policy tried during development at this concurrency (Section 6). The Twitter trace study [5] found production skew often stronger and more dynamic than synthetic benchmarks assume, which is what makes naive LRU fragile. But caches may drop data; a store may not. The design question for HydraKV, addressed in Section 4, is how much of the cache literature's admission machinery can be bolted onto a log-structured store without breaking last-write-wins semantics under 16-way concurrency.

4 HydraKV Design

HydraKV is a single C++17 translation unit of ∼4,470 lines implementing the harness's `IKVStore` interface (`Read`, `ReadAsync`/`CompletePending`, `Upsert`, `RMW`, `Delete`, `Checkpoint`). At a high level:

Disk tier: an append-only slot log. Values live in fixed 128-byte slots in a single O_DIRECT file: 8 bytes of key, 4 of size, 4 of `prev` (a back-pointer forming per-key temporal chains that also route fingerprint aliases), up to 101 bytes of data, a 56-bit log sequence number (LSN), and a CRC-32C over the slot's other 124 bytes. Values past the inline data cap occupy multi-slot *x-records* in the same log (Section 4.2). Slots are staged into per-session 1 MiB chunk buffers, *sealed* (LSN allocated from a global counter, CRC computed with the SSE4.2 `crc32` instruction, ≈16 instructions per slot) at staging time, and landed with sequential writes. Landed pages are immutable except for tombstoning by `Delete`. Sessions reserve disjoint 8,192-slot extents, which makes loading embarrassingly parallel but also means slot position is not a global write-order: a newer version of a key can land at a lower position than an older one. The per-slot LSN, not position, is therefore the authoritative version order, a property that recovery, multi-version reads, and compaction's position reuse all depend on. Every write-back of an overwritten key appends a fresh slot; a threshold-triggered background compactor (Section 4.1) reclaims the superseded ones, bounding the log and, by recycling position ranges, removing the 512 GiB ceiling the 32-bit position field would otherwise impose.

Position index: fingerprint hashing with disk-verified reads. An open-addressing table of 8-byte words, each packing a 32-bit key fingerprint and a 32-bit slot position, in 64-byte buckets scanned

with linear probing and populated by lock-free CAS. The table is the largest power of two fitting in a quarter of the memory budget (2 GiB at 8 GiB). Fingerprints alone collide constantly at this scale (millions of expected 32-bit pairs among 250M keys), so no read ever trusts the index: every candidate position is verified against the full key stored in the on-disk slot, and same-signature candidates are chained through `prev` pointers. The event that actually matters is rarer: two keys agreeing in both the 32-bit fingerprint and the table's 25 bucket-selection bits, an effective 57-bit signature that collides somewhere in the key population with probability $\approx$20% per run, so aliased candidates must be disambiguated safely. Because sessions pre-reserve extents, position order is not globally temporal, and highest-position is therefore not a safe recency proxy across alias chains. All paths that resolve a key to a slot (the synchronous miss path, the `io_uring` completion path, and recovery's index rebuild) instead select the newest CRC-valid verified match by *LSN*, which is a total order over slot versions regardless of which session landed them.

Memory tier: segmented write-back cache with a doorkeeper. The remaining budget (5.25 GiB at 8 GiB, after 2 GiB of index, 0.25 GiB of doorkeeper, and 0.5 GiB of slack) forms an 8-way set-associative cache of 120-byte entries, about 47M values. Entries inline values up to 101 bytes; larger values are stored durably on disk as multi-slot *x-records* in the same log (Section 4.2). A capped in-memory overflow map (one eighth of the memory budget by default, environment-tunable) is a fail-soft absorber for failure paths (disk-full, pinned victims, index capacity), charging every entry it holds, inline-sized included, with over-cap upserts rejected before any state changes and surfaced through rejection counters (Section 7 notes a bounded advisory-check overshoot). Map contents are persisted crash-safely through an atomically renamed sidecar file (Section 4.1). Two mechanisms keep the Zipf tail from churning the hot set. First, a *doorkeeper* in the spirit of TinyLFU [7], though simpler than TinyLFU proper: two rotating generations of budget-sized hash bitmaps (2×128 MiB), no frequency sketch. A read miss is admitted only if its key was already marked in a recent generation, so most one-hit wonders are read once and forgotten (hash collisions admit a small fraction early). Second, *segmented insertion* modeled on SLRU [8]: each set has four probation and four protected ways with one-bit second-chance recency (CLOCK [9]) rather than true LRU ordering. New admissions normally evict within probation only. Protected ways are populated by promoting entries re-referenced while recent, with two pragmatic exceptions (empty protected ways may be filled directly, and a forward-progress fallback may evict a clean protected entry when a set is pathologically pinned). One-pass scans therefore churn roughly the probation half of the cache rather than all of it. Blind upserts of uncached keys are absorbed as dirty entries and written back lazily. Upserts of cached keys update in place under a per-set spinlock with a version bump.

Write-back: cleaners and pin-ownership landing. Four background cleaner threads scan probation ways for *cold* dirty entries (hot dirty keys stay cached and coalesce their overwrites), stage them into flush chunks, and land them with sequential appends, after which the index is repointed old$\rightarrow$new position by CAS. Landing is guarded by a *pin-ownership* protocol. A staged entry is pinned in a `FLUSHING` state, its 32-bit version field overwritten with a fresh staging token (unique until 32-bit wrap, an accepted and documented risk horizon). The flush record may touch the index only if the exact pin (key, state, token, old position) is still present under the set lock at landing time. A concurrent upsert or delete invalidates the pin, and the orphaned record lands as dead bytes in the log but never reaches the index. The landing-time re-check matters because a record staged before a delete can land after the delete's tombstone pass; only the pin re-check keeps such a record from ever reaching the index.

Read misses: raw `io_uring`. `ReadAsync` completes cache hits inline. Misses are sharded to eight dedicated I/O threads, each owning a hand-rolled `io_uring` ring (queue depth 128, using direct `io_uring_setup`/`io_uring_enter` syscalls with explicit acquire/release barriers on the shared rings [13], since the machine lacks liburing) driving a state machine that reads the 4 KiB page, verifies the key, walks alias chains if needed, consults the doorkeeper, and completes the harness's read slot in its required publication order (value bytes and status are stored before the release-store of the slot's done flag). If ring setup fails the engine degrades to a 48-thread synchronous `pread` pool. A runtime environment knob (`HYDRA_NO_URING`) forces this path so tests cover it. `Delete` is supported via a 64-way-sharded deleted-key registry whose fast path is a single relaxed atomic load when no delete is in flight. A deletion mark persists until the key is re-upserted, is re-established at recovery, and is made durable in two layers: a 24-byte CRC-protected *delete intent* appended to a small buffered side log before `Delete` returns (about a microsecond, and process-crash durable, since a SIGKILL cannot

revoke bytes already in the kernel page cache), and on-disk tombstoning of every index-reachable slot of the key, which runs on a background *reaper* queue drained by the cleaner threads (Section 4.2), so `Delete` performs no synchronous slot I/O. The queue is bounded at 64K keys; past the bound `Delete` falls back to the synchronous poison, trading latency for bounded memory. `Checkpoint` and shutdown drain the queue and `fdatasync` before retiring intents, extending acknowledged deletes to the power-loss contract writes already had; the intent log exists because draining alone was not enough, a defect the third audit demonstrated (Section 6, Task 7). `Checkpoint` flushes the calling session's partial chunk and all currently-dirty cache entries with 32 parallel workers, persists the overflow sidecar, then `fdatasync`s. It cannot drain other sessions' partially filled chunks, including idle ones, so completeness requires every other session to have been stopped or to have flushed its own partial chunk first; the engine documents this as a precondition of `Checkpoint`. Clean shutdown is stronger: the destructor lands every session's partial chunk and then runs a full `Checkpoint`, so a clean close loses no acknowledged write. `Checkpoint` is the durability boundary: writes acknowledged after the last successful `Checkpoint` or clean shutdown can be lost on a hard crash, the same checkpoint-based contract FASTER offers.

Sizing and workload assumptions. The harness initializes the engine with two size hints (a 2^{27}-entry hash table and a 16 GiB log) that the specification declares non-binding, and the engine ignores them. Index, doorkeeper, and cache are instead all sized from `mem_budget_bytes`. Keys are hashed with a SplitMix64-style finalizer [14], so dense and sparse key spaces behave identically. The engine never reads the trace files. Two classes of fixed constants remain: the slot/entry geometry (128-byte slots, 120-byte entries, 101-byte inline cap) is chosen for the interface's small-value regime (values past the inline cap take the x-record path of Section 4.2, which is correct and tested but not throughput-tuned), and the thread-pool widths (8 rings, 4 cleaners, 1 compactor, 48 fallback readers, 32 checkpoint workers) are hand-tuned to this machine class.

4.1 Production subsystems

Beyond the scored hot path, HydraKV carries the subsystems that let it survive restarts, disk faults, and unbounded runtimes. (Their scope was enumerated by an independent code-level audit, released with the engine as `HYDRA_PROD_AUDIT.md` together with the audit→fix→test map `PROD_READINESS.md`; Section 6 recounts the development record.)

Crash recovery. On a non-empty log, `recover_log()` performs a 1 MiB-buffered sequential scan that skips never-landed (zero-size) and CRC-torn slots and rebuilds the index newest-LSN-wins, Bitcask/FASTER-style [15, 1]; tombstones participate, so deleted keys stay deleted across restart, and tombstone-winning keys are re-marked in the delete registry. Recovery scratch arrays are freed before the cache is allocated, keeping peak RSS inside the budget. Overflow-map contents (Section 4) persist through a sidecar file written at `Checkpoint` (and clean shutdown) to a temporary file, `fdatasync`ed, atomically `rename()`d, and directory-fsynced, so a mid-write crash can never tear the previous snapshot; per-entry LSNs are compared against the slot log's, so a key with both sidecar- and log-resident versions recovers to the true last write. Recovery time is linear in log size (a persisted index snapshot was considered and rejected: it doubles checkpoint I/O for a cost that is seconds per gigabyte and Bitcask-precedented [15]). If the log holds more keys than the current budget's index capacity (e.g., reopening with a smaller budget), the excess keys are dropped with a counted warning rather than absorbed into unbounded RAM (and `recover_ok` reports the failure, Section 4.2).

Fail-soft I/O. The runtime contains no `abort()` paths. A failed chunk write publishes nothing: the chunk stays staged, every staged cache entry stays pinned with its RAM copy authoritative (the "fsyncgate" lesson [21]: never trust a later fsync after a failure), and the flush is retried on the next trigger, with sticky `write_errors_` and `durable_ok_` flags surfaced to the operator. Bounded retry loops (16 landing attempts for a writer, 64 victim searches) fall back to overflow-map absorption so writers never hang on a dead disk. Read errors zero-fill and increment a counter; a short read below the landed high-water mark is classified as a fault (external truncation) while EOF beyond it is the normal pre-reserved-extent case; `io_uring` completion errors abandon one chain, never the process, and mid-page short reads resubmit the remainder instead of zero-filling bytes the device still owes. `EINTR`/`EAGAIN` are retried with backoff everywhere. A failed `fdatasync` makes `durable_ok` sticky-false. An O_DIRECT→buffered open fallback is logged and surfaced. The surviving `abort()`s are

construction-time invariants only (both log fds failing to open, `mmap`/`posix_memalign` failure, and `io_uring_enter` returning a programming-error errno). This behavior is verified by fault injection: an end-to-end test fills a real tiny filesystem to ENOSPC mid-run and checks that the process stays up, cached reads stay correct, the sticky flags trip, and full durability returns after the disk is repaired.

Log compaction and position reuse. A background compactor wakes every 50 ms and triggers when allocated log bytes exceed a configurable multiple (default $2\times$) of live bytes. It scavenges the log in 8,192-slot (1 MiB) regions, skipping regions owned by active session extents. Live slots are relocated through the existing flush protocol end to end: cache-resident slots via the ordinary staged-flush pin, uncached ones (max-LSN verified by a disk walk) by admitting them clean and then staging them, with under-lock revalidation. Every delete/re-upsert interlock of Section 4 therefore applies to relocations with zero new landing states. After the relocations land and an `fdatasync` barrier, a verification pass confirms no index word and no cache entry still references the region; only then is it hole-punched (`FALLOC_FL_PUNCH_HOLE|KEEP_SIZE`) and its position range pushed onto a free-extent list that the allocator recycles. Position reuse is what removes the 32-bit ceiling, and the per-slot LSN is what makes it safe: any stale CRC-valid bytes that remain readable lose every LSN comparison, and stale `prev` pointers into reused space verify-fail on key or lose on LSN. On filesystems without hole punching the punch failure is nonfatal and surfaced; extent reuse alone stops file growth. Compaction relocates tombstones of still-deleted keys instead of dropping them: an abandoned flush can leave a CRC-valid, unlinked orphan of a deleted key in the log, harmless only while a newer tombstone survives, so reclaiming a tombstone's region while the orphan's region persisted would let the orphan win the newest-LSN recovery scan.

Observability. Because the harness freezes the `IKVStore` interface, production counters are exposed through a C-linkage seam (`hydra_get_prod_stats`): `durable_ok`, `recover_ok`, recovered keys, torn slots skipped, write/read errors, oversize rejections and bytes, compactions run, allocated/live/reclaimed log bytes, the buffered fallback, and unsupported hole punching.

Cost on the scored path. None of this runs in a scored window: the harness wipes the spill directory before every run (so recovery sees an empty log and does one `fstat`), and a 30-second window cannot push the log past the compaction threshold. Three additions do run on the scored path: one hardware-CRC seal and one LSN `fetch_add` per staged slot, one CRC check and an LSN comparison per served read miss (tens of nanoseconds against a $\sim$350 μs NVMe read), and a relaxed load of the free-extent counter per chunk allocation. A/B measurements of engine revisions with and without these subsystems score identically within the harness's $\pm$0.02 Mops/s run-to-run noise (Sections 5.1 and 6).

4.2 Deployment subsystems: oversized values, deletes, and shutdown

Three further subsystem groups govern the engine's behavior outside the scored workload's reach: a durable on-disk representation for oversized values, a non-blocking delete path, and the accounting, recovery, and shutdown contracts. (Their scope was set by a second independent audit, an all-workload campaign, and a third independent audit of the delete path, released with the engine as `AUDIT2.md`, `AUDIT2_FIXES.md`, `WORKLOAD_HARDENING.md`, and `AUDIT3_FIXES.md`; the corresponding tasks appear in Section 6.)

Oversized values: x-records. Values larger than the 101-byte inline cap are stored durably as *x-records*: multi-slot records in the main log, a 32-byte header plus data across up to 33 slots, extent-aligned, written through a second buffered file descriptor on the same file, linked into the fingerprint index under the set lock, LSN-ordered against inline versions of the same key, and restored by the ordinary recovery scan, so they need no sidecar. Because buffered x-appends can leave the file's end unaligned, and ext4 rejects O_DIRECT reads that cross such a tail, an atomic x-extent bitmap routes every read that touches an x-page through the buffered descriptor. Inline$\leftrightarrow$x transitions of a key resolve by LSN in both directions. X-record extents are never compacted, a documented leak class (Section 7).

Non-blocking, crash-durable deletes. `Delete`'s entire linearization runs under the key's set lock, the same lock every upsert path publishes under (the third audit's review rounds showed anything weaker admits a no-fault race in which a concurrent upsert resurfaces the key's old checkpointed

value): it marks the registry, allocates the delete's LSN, sweeps the cache and overflow map, appends the delete intent to the buffered side log, and enqueues the key for the reaper, then returns in about a microsecond, on par with an upsert; it never waits on a key pinned in another session's staged chunk. The intent record is what makes an acknowledged delete survive a process kill: tombstoning may not have happened yet when the process dies, so recovery replays intents newest-LSN-wins against everything the log scan reconstructed. `Checkpoint` retires the intent log only under a conservative proof of redundancy (reaper and synchronous-poison quiescence, a successful `fdatasync` with lifetime-zero fault counters, no reinsert races since the stripe scan, no un-landed session chunks, and a cleanly recovered instance); on any other outcome it `fdatasync`s the intents instead, extending them to power-loss durability. Every fallback on the failure ladder (synchronous poison, overflow-sidecar rewrite) is counted, and any counted fault permanently blocks retirement. Consistency is preserved at landing time as well: every chunk landing re-checks the delete registry under the key's set lock and reseals a deleted first-insert slot as an on-disk tombstone. On-disk poisoning of a deleted key's older versions rides the background reaper queue of Section 4, bounded at 64K keys with a synchronous fallback as backpressure, and drained by `Checkpoint` and shutdown. `tombstone_slot` verifies the stored key and requires a live-parsed size before resealing (a stale position must never poison an innocent key, and a hole-punched page reads back zeros, where key 0 would otherwise false-match), refuses whole-page read-modify-writes on x-pages, and counts every refusal. Compaction restages a fresh tombstone for a still-deleted key before reclaiming the region that held the old one, and a pre-punch compaction-epoch bump lets a reader that overlaps a compaction retry once against the post-move index rather than surface a transient NOT_FOUND.

Accounting, recovery, and shutdown contracts. Every overflow-map entry is charged against the cap, inline-sized entries included, and all absorb paths go through the budget-capped rejection logic. Delete-registry marks are charged too (64 bytes each, into the same budget-shared pool, with a loud one-time warning at the cap); marks are load-bearing for correctness and are never silently trimmed. A `Checkpoint` that cannot persist entries parked in another active session's un-landed chunk counts them and prints an actionable warning rather than silently reporting completeness. `recover_ok` reports 0, with an exported drop count, whenever recovery cannot retain every key (e.g., reopening a large log under a smaller budget whose index cannot hold it). The recovery scan reads through the buffered descriptor, never O_DIRECT, so a crash-torn unaligned tail cannot cause whole regions to be skipped. Extent reservation is a single critical section covering the allocation advance, ownership registration, and bitmap mark, making reservation and compaction mutually exclusive by construction. Shutdown's parallel `Checkpoint` degrades to fewer workers if a thread cannot be spawned. One architectural limit is surfaced explicitly: the fingerprint index needs $\approx$8 bytes of budget per distinct key, so a 2 GiB budget cannot index 250M keys; initialization warns with the minimum budget that fits, and over-capacity insertions are rejected and counted.

5 Experimental Evaluation

5.1 Main results

All runs use the unmodified harness, cold caches, the enforced 8 GiB resident cap, and the scored environment (100-byte values, asynchronous reads with pipeline depth 256, integrity auditor on). Every configuration first passes the untimed validation run: all 250,000,000 keys read back with correct checksums, zero missing, zero wrong, zero cross-thread audit failures.

HydraKV's three runs on the original workload were 5.50 / 5.50 / 5.52 Mops/s (median 5.50), with StoreRSS $\approx$7.9 GB on every run and a same-day A/B control of the immediately preceding revision landing at the same median. The number is stable: every re-score of the original workload across the engine's revision history, including an independent audit's re-run on the reference hardware and the workload-hardened revision's 5.51, landed between 5.50 and 5.52 Mops/s, a band explained by the harness's $\pm$0.02 Mops/s run-to-run noise plus day-to-day box drift (Section 6 reports each score with its same-day A/B control), so the recovery, compaction, delete-durability, and oversized-value subsystems cost nothing measurable, as the hot-path budget of Section 4.1 predicts. Table 2 reports the engine's behavior on the operation mixes the benchmark does not score. Beyond those mixes, the same protocol was run with `VALUE_SIZE`=1024/4096, bimodal 20 B/200 B values, synchronous and `io_uring`-disabled read paths, 1 and 32 threads, a 2 GiB budget, tmpfs and buffered-fd fallbacks, and crash/restart in every mode, with zero integrity failures everywhere. The shape of Table 2 follows

Table 1: Throughput on the reference machine: median of three 30 s cold-cache runs, 16 workers, per-run values where recorded. The adversarial matrix (lower block) was scored with the v4c engine revision and frozen when the discovery loop closed; only the original workload was re-scored with later revisions, every one of which landed between 5.50 and 5.52 Mops/s (per-revision scores in Section 6, including an independent audit's re-run at 5.51). FASTER was measured only on the original workload, so no speedup is reported for the variants. The holdout used fresh seeds, was generated after all engine work stopped, and received no tuning. Its first measurement launch was killed externally mid-run after scoring 3.86 and was relaunched without any engine change (runs 3.87 / 3.87 / 3.21). StoreRSS stayed ≤ 8 GiB on every scored run.

System	Workload	Mops/s	vs. FASTER
FASTER	original (Zipf θ=0.95)	0.93	1.0×
HydraKV	**original** (runs 5.50 / 5.50 / 5.52)	**5.50**	**5.9**×
HydraKV, v4c	original	5.48	—
HydraKV, v4c	V1: sparse SplitMix64 keys	5.67	—
HydraKV, v4c	V2: clustered hot set	5.56	—
HydraKV, v4c	V3: drifting hot set	3.97	—
HydraKV, v4c	Holdout: 60/25/15 mix, θ=0.92	3.87	—

Table 2: Throughput on the harness's other operation mixes over the same 250M-key Zipf-0.95 traces, run under the scored protocol (same cgroup, NUMA pinning, cold caches, integrity auditor on) but not part of the scored benchmark, so single runs, and FASTER was not measured on them. The 50:50, 0:100, 5:95, and timeseries rows are the final (delete-durability) engine; the RMW, B, and C rows were measured with a mid-campaign revision and not re-run. Every run exited 0 with zero integrity-audit failures.

Mix (read:write)	Character	Mops/s
50:50 (YCSB-A, scored)	the benchmark workload	5.50
0:100 (auditors' mix)	blind-upsert-only	5.13
5:95 (auditors' mix)	write-heavy	5.19
95:5 (YCSB-B)	read-mostly	2.93
100:0 (YCSB-C)	read-only	2.78
RMW-only	synchronous read-modify-write	0.30
Timeseries	delete-heavy (1.28M deletes, 0 resurrections, 0 lost keys)	0.08

from the design: blind upserts absorb into the write-back cache at RAM speed, so the write-heavy mixes run near the scored number, while read-dominated mixes are pinned near the device's miss bandwidth (the ceiling arithmetic of Section 5.3 with T mostly reads), and the RMW-only and delete-heavy mixes are synchronous-latency-bound by design.

5.2 Adversarial variants and the holdout

The variant generator (released with the engine) produces load/run trace pairs in the harness's format. All generated variants draw keys as SplitMix64 hashes of insertion indices (sparse, uniformly scattered 64-bit keys, unlike the published trace's dense key space) and vary three knobs: skew θ, hot-set geometry, and RNG seeds. The geometries are *scrambled*, *clustered*, *drift*, and *mix*. Scrambled scatters hot keys arbitrarily. Clustered draws hot keys from contiguous *insertion-index* ranges, which after hashing concentrates them in on-disk placement rather than in external key ranges: a deliberate placement-locality stressor, inspired by (but not an implementation of) the key-range locality documented at Facebook [4]. Drift crossfades the hot set between epoch-specific scrambles across the run, an author-designed temporal-shift stressor motivated by the observation that production popularity is dynamic [5]. Mix is a seeded blend of the three. Variants were constrained to stay within the target workload's operation mix, value size, key count, and skew regime, so that a variant could not "win" by being physically incapable of high throughput (a uniform-random variant would prove nothing on this device). These are literature-informed synthetic stressors, not trace replays, and their fidelity to production workloads remains a threat to validity.

V1 (sparse keys) is the variant that rules out dense-key index designs: a flat 4-byte-per-key position array sized to the published trace's dense $[0, N)$ key space cannot exist for sparse 64-bit keys, and the engine revision that assumed it was OOM-killed (Section 6). The fingerprint hash index costs memory and CPU but generalizes to arbitrary keys. V2 and V3 were generated against the fingerprint-index engine. The clustered variant scored slightly higher than the scrambled original (5.56 vs. 5.48). The difference is about 1.5%, and we did not collect the I/O-level measurements needed to attribute it. Drift (V3) is genuinely harder (3.97): a moving hot set halves the effective lifetime of every admission decision, the measured hit rate falls accordingly, and no engine change we tried recovered the difference without hurting the stationary workloads. The holdout (fresh seeds, mixed geometry, $\theta = 0.92$, generated only after all engine work stopped) landed at 3.87 Mops/s, slightly below the drift variant, consistent with its drift component dominating its difficulty. One measurement caveat, disclosed in Table 1: the holdout's first `run.sh` launch was killed by an external signal mid-way (after a 3.86 first run) and the measurement was relaunched. No engine or configuration change occurred between launches, so the no-adaptation property holds, though the measurement did span two launches.

5.3 The throughput ceiling, and the 10 Mops/s goal

The task sequence twice posed throughput goals beyond the reported score: a flat 10 Mops/s, and later a 7.0 Mops/s stretch (Section 6). Neither was met; the reason is a ceiling argument, which we reproduce here with its scope made explicit. Each scored run starts cold and lasts 30 s. At 50% reads, T Mops/s implies $T/2$ million read ops/s, of which a fraction $1 - h$ miss the cache and, in this design, cost one 4 KiB device read each. The device delivers at most 718k such reads/s. Sustaining T=10 therefore needs $h \geq 1 - 0.718/5$, i.e., ≈86%. But across every admission policy tried (doorkeeper generations and thresholds, neighbor co-admission, frequency variants), the hit rate of a cold 30-second window pinned at 76.8–77.1%: at ∼5.5 Mops/s the window simply does not re-reference enough distinct keys often enough for any of them to do better. At h=77% the miss-bandwidth fixed point is $2 \times 0.718/0.23 \approx 6.2$ Mops/s. The engine's 5.50 is about 89% of that bound, with the remainder going to write-back I/O contention and CPU. This is an empirical bound for *this class of design* (one 4 KiB read per miss, admission-based caching, 8 GiB of accounted memory), supported by a policy sweep. It is not an impossibility proof over all designs. A scheme that co-locates hot records on shared pages could in principle beat it, though our attempt at that scored worse (Section 6). Breaking ≈86% within this class would require more effective cache than 8 GiB. The one cheap way to get it, letting the OS page cache inside the 14 GiB cgroup absorb the spill file, was rejected as violating the budget's intent. The engine opens its log O_DIRECT so that its disk traffic cannot ride the page cache.

5.4 Correctness and testing

The engine ships with an end-to-end test suite (a 2,170-line test program plus a matrix runner, 28 tests) that drives the real engine through real files with scaled workloads, with no mocks, covering concurrent read/upsert/delete/checkpoint interleavings, fingerprint-alias key sets crafted by inverting the index hash, oversized-value paths, the `io_uring`-disabled fallback, SIGKILL crash-resurrection oracles, and delete-resurrection oracles (four readers hammering keys while a writer flaps upsert/delete, with per-key issue/completion interval brackets making "value from the wrong phase" detectable). The matrix runs six configurations: optimized, buffered-I/O on tmpfs, no-`io_uring`, ASan+UBSan, TSan (a nine-suite subset, since TSan's slowdown makes the full suite impractical), and a coverage build. All pass, and the standard matrix executes 89.3% of engine lines under gcov (measured on the revision preceding the delete-durability additions; the uncovered remainder is dominated by fail-stop I/O-error paths and sub-microsecond race windows, per manual inspection of the gcov report). Known residuals the suite does not exercise are listed in Section 7. The delete-resurrection oracle is the suite's strongest instrument: it passes only under the full pin-ownership protocol of Section 4. A final audit confirmed the engine performs no trace reads and contains no key-population constants.

Recovery and fault tests. `recover` verifies that a clean restart returns exact bytes, deletes stay deleted, and inline↔oversized transitions resolve by LSN. `crashrecover` forks a child that `Checkpoints` and is SIGKILLed; the parent must recover every checkpointed key exactly, with CRC-torn slots skipped, never served. `enospc` fills a real tiny filesystem to disk-full mid-run: no

crash, sticky error flags, correct cached reads, full durability after repair. `readfault` externally truncates the log: NOT_FOUND plus a counter, never an abort. `recoverdrop` reopens a log under a smaller budget: `recover_ok` must fail and count its drops.

Compaction tests. `compact` verifies that the log's block count shrinks under concurrent readers and that the store survives restart on the hole-punched log. `compactcold` forces the compactor's *uncached*-relocation path and exercises the deepest cross-restart interaction in the engine (flush abandonment, tombstone relocation, and recovery's newest-LSN scan, the invariants of Section 4.1); a 15/15-iteration ASan stress loop validates it, and a third restart generation checks that every deleted key stays deleted and that x-resident values need no sidecar.

Delete, capacity, and value-size tests. `updel2` verifies that deletes of keys pinned in an idle foreign session return promptly and stay correct across the landing and a restart. `delcost` enforces a latency budget (N deletes $\leq 10\times$ N upserts). `delcrash` SIGKILLs forked children that delete checkpointed keys, across two crash generations, and asserts that no acknowledged delete is ever resurrected at recovery, with the reaper paused by a test hook so the outcome provably depends on the intent log rather than on reaper timing, plus oversized, checkpointed-reinsert, and reinsert-racing-`Checkpoint` facets (the last must recover as the reinsert or NOT_FOUND, never the pre-delete value). `memfull` drives bounded memory and counted rejection through a 16 MiB store. `oversizebound` pushes 30K$\times$4 KB values, more data than the whole budget, with zero rejections, since x-records make oversized values disk-resident. `bimodal` runs 300K mixed 20 B/200 B keys, four writers against four readers on both read paths, a tiny budget, forced compaction, full verification, and a restart leg that verifies the clean-shutdown contract. `capwarn` asserts the index-floor warning in both directions on captured stderr.

6 How HydraKV Was Built: a Task Sequence in KISS Sorcar

HydraKV's development is inseparable from its methodology, so we document it as the single case study it is. The process observations below are drawn from this one project and were not obtained by controlled comparison. All work ran in KISS Sorcar [27], whose relevant properties here are: long-horizon task execution with automatic continuation across context windows, unattended operation against a remote benchmark machine over SSH, mixing models from different vendors inside one task via prompts, and persistent per-repository progress notes that later sessions read. The human contribution was seven task-opening prompts and two shorter mid-task steering messages. The agent wrote every line of engine and test code. The benchmark specification demands the problem be solved with no external help, no reference code, and no prior artifacts, and forbids network access at run time. No external implementation or reference code was consulted or reused at any point, and every artifact in the engine's lineage is a revision of the project's own code. The driving prompts, however, ordered extensive internet research, and the agent retrieved published workload-characterization studies that shaped the variant axes of Section 5.2. We read the specification's restrictions as governing the engine and its runtime and the prompts as governing the development process. A stricter reading of "no external help" would count the literature research against it. We reproduce the seven prompts below word for word (typos, model directives, and the author's own server address included) because they constitute the specification the agent worked from. Nothing was cleaned up for publication, though TeX re-wraps lines and collapses runs of spaces. (Three pieces of notation recur. `claude-fable-5` and `gpt-5.6-sol` are the identifiers under which the framework ran its development and review models, "e2e" abbreviates end-to-end, and `./KV_TASK.md` is a local copy of the benchmark's task specification.) One of the two steering messages is quoted in place below. The other survives only as a contemporaneous summary, which we flag when we reach it. Tasks 2–7 explicitly split the roles (`claude-fable-5` for development, `gpt-5.6-sol` in a read-only debugging role, capped in every task but the third at 20% of the task budget), and independent reviews ran at the major milestones described below. Total model-API spend across the first three tasks' sessions, summed from the framework's per-session budget accounting, was under $200; Task 4's reviewer spend was roughly $5, Task 6's three reviewer rounds consumed about 12% of its budget, and Task 7's two rounds about 2%, all under their caps.

Task 1: baseline and first engine. The first task asks only that the agent set up the benchmark repository on the remote machine and run the setup document's steps; those steps in turn require

supplying an engine and scoring it (the setup's pass bar was a validated run with a median above the 0.93 baseline, with a stated target of about $2\times$ that):

```
I have a server where I can login using: ssh ksen@34.70.16.69 .
My working directory on the server is /mnt/ssd/ksen/kv50-benchmark .
After logging in can you clone https://github.com/shubham3-ucb/baselines-kiss-sorcar.git using the
credentials of ksenxx user on github (the local machine has the credentials). Then do the following:
```
cd baselines-kiss-sorcar/task
export KV_SPILL=/mnt/ssd/kv_spill
```
Then run the steps 3 to 6 at https://github.com/shubham3-ucb/baselines-kiss-sorcar/blob/main/SETUP.md .
```

The agent produced a first-principles engine in one session: a flat 4-byte-per-key position array (exploiting the published trace's dense key space), fixed 128-byte slots written sequentially at load, in-place 4 KiB read-modify-writes for write-back, and a version-checked write-back cache. It validated (250,000,000/250,000,000 keys) and scored 1.85 Mops/s, just under $2\times$ FASTER. The dense-key assumption also left it heavily overfitted to the published trace, as the next task showed.

Task 2: adversarial AI discovery. The second task installed the loop, in full:

```
Can you perform adversarial AI Discovery for the task at ./KV_TASK.md so that the goals in the task are
met? Look at the previous task on how to validate the engine on the server. You MUST not stop until the
goals are met. Generate adversarial workload variants to make sure that the engine works fast on the
variant workloads. Do the following iteratively while maintaining a variable iteration_count variable
which starts at 1 and increments by 1 on each iteration:
Generate a variant workload that are realistic like the original workload, but breaks the performance
gain of the engine. Do extensive internet search to understand how to make the variant workload
realistic to real-world workloads and robust to reward hacking or cheating. Then run the engine on the
variant workload. If the goals are not met, use AI discovery to improve the engine on all workloads
until you achieve the goals without cheating using the workloads. Then generate a new workload repeat
the process until the engine can achieve the goals on the new test workload on which AI discovery was
not performed.

Search the internet extensively. Use 'claude-fable-5 model' for all tasks, including software
development. Use 'gpt-5.6-sol' (not codex) for a thorough read-only review and debugging of the other
model's work. Thoroughly check whether the other model has missed any code or wiring or introduced any
bugs. Use at most 20% of task budget in gpt-5.6-sol for reviewing and debugging. Use the model names
literally without hallucinating new model names.
```

The first of the two steering messages arrived mid-loop, raising the goal from $2\times$ FASTER to a flat 10 Mops/s and adding a strict no-hardcoding requirement (no constants derived from the trace's dense key space or key count). Its exact wording was not preserved in the session log, so we summarize its content here. The agent grounded its variant axes in the workload-characterization literature it retrieved (key locality at Facebook [4], skew and temporal dynamics at Twitter [5], YCSB's generator design [2]), wrote the variant generator, and executed the loop of Section 5.2. V1 broke the dense index (OOM). The rebuilt fingerprint-index engine passed V1 at 1.91 Mops/s. The goal raise triggered the profiling-driven redesign (log-append write-back, sharded I/O queues, doorkeeper, then `io_uring` rings and segmented insertion) that reached 5.50. V2/V3 and the holdout closed the loop. Two disciplines were enforced by the prompt and are visible in the artifact: the same binary ran every workload, and each accepted idea appears in the ledger with its measured score, each rejected one with its failure number (the accepted-and-rejected-ideas record below). The 10 Mops/s goal ended as the ceiling argument of Section 5.3.

Task 3: find every bug, keep the speed. The third task, again in full:

```
can you find all redundancies, inconsistencies, obvious bugs, corner case bugs, race conditions,
deadlocks in your final implementation of the engine? Validate them by writing e2e tests with workloads.
Then remove them and make sure that all tests pass. Write e2e tests with workloads to achieve 100%
coverage of the egine code. Check the correctness of the engine by writing adversarial e2e tests with
workloads. Honestly review the code to check if you have cheated based on the workload knowledge. If so,
remove them. You MUST NOT reduce the performance below 5.48 Mops while fixing bugs. Search the internet
extensively. Use claude-fable-5 model for all tasks, including software development. Use gpt-5.6-sol
(not codex) for a thorough review and debugging of the other model's work. Thoroughly check whether the
other model has missed any code or wiring or introduced any bugs.
```

The prompt's hard floor of 5.48 Mops/s is the v4c score on the original workload. Two of its literal demands were not met: coverage stopped at 92.84% of lines and 93.42% of branches (Section 5.4), and "all" bugs is not something any test suite certifies (Section 7 lists the residual risks left open). This

task consumed three sessions and put the most weight on the split between developer and reviewer models: the `gpt-5.6-sol` reviews produced dozens of line-referenced findings whose serious ones shaped the LSN-ordered read paths and the pin-ownership landing protocol of Section 4. When the assembled fixes scored 3.6 Mops/s, the second steering message supplied the decisive hint, in its entirety: *"Did you start with the most performant variant of the engine?"* The agent bisected archived engine revisions, discovered the answer was no (the fixes had been layered onto the rejected relocation branch), re-based all of them onto the fastest revision, and re-ran the full matrix and the scored gate: 5.51 / 5.51 / 5.53, slightly above the floor. The recovery came from two prefetch hints found during the re-port.

Task 4: make it production-ready, keep the speed. The fourth task arrived after an independent code-level audit (`HYDRA_PROD_AUDIT.md`) had compared HydraKV head-to-head against `kv_5050`, a second engine built independently by a different agent lineage for the same benchmark. The audit concluded that HydraKV was faster (5.9× vs. 3.49× FASTER) because it "drops general-prod robustness". The prompt, in full:

```
Can you start with the latest best performant engine code and make it production ready (as pointed out
in https://github.com/shubham3-ucb/baselines-kiss-sorcar/blob/hydra-audit/HYDRA_PROD_AUDIT.md) while
keep the performance at 5.5 Mpos/s or increasing it to 7.0 Mpos/s using adversarial AI discovery . Write
end-to-end 100% coverage tests for the feature first. Then implement the features. Use 'claude-fable-5
model' for all tasks, including software development. Use 'gpt-5.6-sol' (not codex) for a thorough
read-only review and debugging of the other model's work. Thoroughly check whether the other model has
missed any code or wiring or introduced any bugs. Use at most 20% of task budget in gpt-5.6-sol for
reviewing and debugging. Use the model names literally without hallucinating new model names.
```

(The linked audit branch was private to the agent's credentials; it worked from the local copy, now committed beside the engine.) The agent researched recovery and fault-handling precedent before designing (RocksDB's WAL formats, recovery modes, and background-error handling; LevelDB's log resync blocks; Bitcask's checksum-verified recovery scan and merge compaction; FASTER's rebuild-by-scan recovery; the PostgreSQL "fsyncgate" lesson, systematized by Rebello et al. [21], that dirty state must stay authoritative in RAM after a failed fsync; `fallocate`, `io_uring`, and hardware-CRC32C references), wrote seven end-to-end tests first (the recovery, fault, and compaction tests of Section 5.4), confirmed each failed on the shipping engine, and then implemented the subsystems of Section 4.1, logging every design decision and rejected alternative in the same ideas ledger the discovery loop used. As in Task 3, one literal demand went unmet and is reported as such: coverage reached 93.12% of lines, not 100%. The `gpt-5.6-sol` reviews ran twice, strictly read-only. The first produced 18 line-referenced findings, of which 12 were fixed in code and 6 documented as residual risks with rationale (Section 7). The second, a diff review of the final state, returned approve-with-nits and zero code findings. The performance leg mirrored Task 3's gate: the first full pipeline landed at a six-run median of 5.44, and a re-measured baseline showed roughly half the gap was box noise. A literature-driven micro-optimization round closed the rest, one idea at a time against a re-measured 5.50 baseline: cache-line isolation (`alignas(64)`) of ten code-verified falsely-shared hot atomics was accepted (5.50→5.51, non-negative and removing a real hazard); `MADV_HUGEPAGE` on the index, cache, and doorkeeper was implemented, measured at no gain (the advisory promoted only 32 MiB of ∼9 GB on the fragmented box), and reverted; batched per-session LSN allocation was rejected on paper because it provably breaks the per-key LSN monotonicity recovery relies on; counter batching, branch hints, and CRC tuning were rejected by cost analysis as below the ±0.02 Mops/s measurement noise. Every idea whose projected "headroom" required exploiting the page cache or the harness's blind spots was refused as violating the budget's spirit. The task ended at a six-run median of 5.52 (runs 5.50 / 5.51 / 5.53 / 5.51 / 5.53 / 5.53) against the 5.5 floor, with the 7.0 stretch documented as unreachable by the ceiling arithmetic of Section 5.3.

Task 5: answer the second audit. Between tasks, the second independent audit (`AUDIT2.md`) had re-run the engine on the reference hardware with its own self-asserting programs and verified the headline number as genuine (median 5.51 Mops/s, 5.9× FASTER; StoreRSS within the 8 GiB budget on every run; true O_DIRECT end to end; verbatim CRC-verified values; no workload constants; in the audit's words "not reward-hacked, not copied"), while probing deployment behaviors outside the scored window's reach. The fifth task pointed the agent at the findings, in full:

```
Can you address the issues raised by ./projects/kv_adversarial/AUDIT2.md thoroughly and precisely and
make sure that similar defects are not present? Make sure that scores must not go below th current best
```

```
score. Use 'claude-fable-5 model' for all tasks, including software development. Use 'gpt-5.6-sol' (not
codex) for a thorough read-only review and debugging of the other model's work. Thoroughly check whether
the other model has missed any code or wiring or introduced any bugs. Use at most 20% of task budget in
gpt-5.6-sol for reviewing and debugging. Use the model names literally without hallucinating new model
names.
```

Every finding was closed at root cause with a regression test, and "similar defects" was read broadly: the reviewer model's pass over the remediation produced further corrections, all closed, and the audit's code-confirmed-but-narrower findings were closed as well (Section 4.2). "Not below the current best score" was verified with a control: the agent re-ran both the fixed engine (two suites, medians 5.50) and a same-day A/B run of the archived pre-fix engine (median 5.50), attributing the 0.01 delta against July 20 to the box rather than to itself. The fixes and their evidence are released as `AUDIT2_FIXES.md` with raw logs in `refnode_rerun_jul21/`.

Task 6: all workloads, all paths. The sixth task forwarded the auditors' feedback verbatim inside the prompt (the inner quotation is the auditors' own message, including its texting shorthand):

```
Here is the feedback I got on HydraKV. Can you test it end to end for all kinds of workloads taking
different program paths and fix all bugs? I do not want to hear similar complaints in the future. Fix
all possible bugs via thorough testing and make it production ready. Use 'claude-fable-5 model' for all
tasks, including software development. Use 'gpt-5.6-sol' (not codex) for a thorough read-only review and
debugging of the other model's work. Thoroughly check whether the other model has missed any code or
wiring or introduced any bugs. Use at most 20% of task budget in gpt-5.6-sol for reviewing and
debugging. Use the model names literally without hallucinating new model names. Make sure the score does
not fall below 5.5 Mops/s.

"can AI build systems (like the KV store here) where reality (real-world users and deployment) is the
only true test of whether it's actually usable?

For more task - U can use the same setting but just switch to 0:100 workload, and/or 5:95 workload
(read:write, same skew etc, generating YCSB variants is easy). This is what we use for benchmarks."
```

This task ran four sessions and produced the workload campaign of Section 4.2: the sweep matrix, the subsystems it left in the engine, and the documented architectural limits. The reviewer model ran three read-only rounds ($\approx$12% of budget), each yielding real corrections. The task's endgame was procedural: the TSan configuration was looped 15/15 clean, a stale harness binary was caught (the scored runner never rebuilds; the engine's new statistics line proved which binary actually ran) and the scored gate re-run on a fresh build, closing at 5.51 Mops/s against the prompt's 5.5 floor. Evidence logs are released in `refnode_workloads_jul21/` and the campaign is documented in `WORKLOAD_HARDENING.md`.

Task 7: the audit of the fix. The auditors returned with a third audit. It opened by confirming the four earlier findings fixed and the score preserved, then demonstrated, with four self-asserting repro programs, that the remediation of the delete-latency finding had introduced a worse defect: `Delete` now returned before anything about the delete was durable, so killing the process between the return and the next `Checkpoint` resurrected the key's durably checkpointed versions at recovery (5,000 of 5,000 deletes in the minimal repro; 60K of 97K at scale). Two further findings: `Checkpoint` could not persist entries parked in other sessions' chunks, and the delete registry's RAM was uncharged. The seventh prompt, in full:

```
Here is new feedback https://github.com/shubham3-ucb/baselines-kiss-sorcar/tree/hydra-audit/July_21. Can
you thoroughly test if there are any more regression bugs introduced. Use 'claude-fable-5 model' for all
tasks, including software development. Use 'gpt-5.6-sol' (not codex) for a thorough read-only review and
debugging of the other model's work. Thoroughly check whether the other model has missed any code or
wiring or introduced any bugs. Use at most 20% of task budget in gpt-5.6-sol for reviewing and
debugging. Use the model names literally without hallucinating new model names.
```

The repair is the delete-intent log of Section 4.2, designed around the observation that a buffered `write(2)` survives SIGKILL at about a microsecond of cost, restoring crash durability without re-importing the synchronous latency the reaper had been built to remove. The reviewer model ran two read-only rounds. The first returned fourteen findings against the initial implementation (intent-log truncation ordering, unbounded replay reads, LSN reuse on an intent-only restart, among others). The second found the task's deepest bug, a no-fault linearizability race: an upsert publishing between `Delete`'s registry mark and its cache sweep could resurface the key's old checkpointed value at runtime, with no crash involved. The delete's entire linearization moved under the key's set lock in response, and `Checkpoint`'s intent-retirement condition was rebuilt as the conservative

proof of Section 4.2. Final state, verified on the reference node: all four repro programs clean (zero resurrections; `Delete` at 1.0 μs $= 1\times$ an upsert, versus the 244 μs the latency finding had originally measured), 28/28 tests locally and at scale 4 on the node, a TSan loop over the new test ten-for-ten clean, and a scored median of 5.50 Mops/s (runs 5.50 / 5.50 / 5.52) with a same-day pre-fix control at the same median, so the durability repair is score-neutral. The findings, the design, and its documented residuals are released as `AUDIT3_FIXES.md` with both review reports in `refnode_audit3_jul21/`. The defect had been introduced by the remediation of the previous audit and survived that task's own reviews, sanitizers, and test matrix; it was caught because the auditors wrote repro programs against the modified path itself.

Accepted and rejected ideas. The discovery loop's bookkeeping (an append-only ideas ledger the agent maintained to avoid retrying failures) records both directions of the search. On the accepted side: the first engine reached 1.85 Mops/s, and the remaining $3\times$ accumulated across the loop's redesigns (log-append write-back replacing in-place 4 KiB read-modify-writes, doorkeeper admission, segmented insertion, and the `io_uring` miss path), each adopted after measurement on the scored configuration. The measurements were taken on an evolving code base, so these per-mechanism scores are an engineering log, not a controlled ablation, and each rejected idea below is scored on the original workload against its own then-current base, not against the others. Strict-probation segmentation scored 0.78 Mops/s: forward progress stalls when probation fills with pinned entries. Two-way probation scored 0.95. Page-neighbor co-admission on read misses scored 1.98, because admitting a missed page's other residents pollutes probation faster than it prefetches. Always admitting load-phase-clean keys scored 4.17. Cleaners that flush any dirty entry rather than only cold ones scored 0.47: hot-key reflush storms saturate write bandwidth. A heat-clustered relocation scheme that groups hot keys onto shared pages scored 4.03–4.34. The extra write traffic exceeded the locality benefit at this value size. The last entry matters beyond its number: the hardening task's initial 3.6 Mops/s "regression" turned out to be correctness fixes layered onto this already-rejected relocation branch rather than onto the fastest revision, and only bisection against archived scored revisions exposed it. That episode is the strongest argument we have for keeping an immutable record of what was tried, on which base, at what score.

Observations. Five observations from this one case study stood out. (1) *Adversarial self-evaluation was cheap and decisive.* Generating a distribution shift and re-scoring cost minutes and immediately exposed the dense-index overfit. The held-out variant then bounded how much the repairs themselves had overfit the loop. (2) *Cross-model review surfaced different bugs than same-model review.* The reviewer model, prompted read-only with no authorship stake, repeatedly flagged concurrency hazards the developer model had rationalized. The hardest concurrency hazards were ultimately pinned down by tests the reviewer's findings inspired, not by either model's reading of the code. (3) *Agents need immutable experimental provenance.* Both failure modes hit during development (retrying known-bad ideas, and hardening the wrong engine version) were provenance failures, repaired by artifacts (the ideas ledger, archived scored revisions) that cost almost nothing to maintain. (4) *Tests-first, coverage-guided, at feature scope.* Task 4's discipline of writing each end-to-end test before its feature and proving it failed made the audit items falsifiable, and the one test added purely because coverage showed an unexercised path (`compactcold`) guards the one class of defect no runtime oracle can see: state that goes wrong only one restart after the racy interleaving. (5) *Internal rigor did not substitute for external adversaries.* Tasks 3 and 4 ran sanitizers, coverage, cross-model review, and fault injection, and still left deployment defects standing. Those defects were found only when an audit with no authorship stake attacked the built engine on hardware with its own oracles, and when a workload sweep exercised operation mixes (deletes under concurrency, oversized values, write-only traffic) that no amount of code-path coverage on the scored workload would reach. The audit also published its retractions alongside its findings. The pattern then repeated at a level up: the third audit's headline defect was created by the remediation of the second's, had survived that remediation's own reviews and matrix, and fell only to external repro programs.

7 Limitations

We list what still separates HydraKV from the production systems it is compared against, together with the residual risks left open when work stopped.

Durability and recovery model. Durability for writes is checkpoint-based, as in FASTER's fold-over model: writes acknowledged after the last successful `Checkpoint` or clean shutdown are lost on SIGKILL or power loss. Acknowledged deletes are asymmetrically stronger, because the symmetric contract turned out not to hold and was repaired after the third audit (Section 6, Task 7): the delete-intent log makes an acknowledged delete survive a process kill from the moment `Delete` returns, and the next `Checkpoint`'s `fdatasync` extends it to power loss. The write-side window is real but, measured, small: the second audit SIGKILLed a store that had never checkpointed and lost only $\sim$3K of 195K acknowledged writes, because the cleaners land most within milliseconds. Recovery is a full log scan, linear in log size (seconds per gigabyte on this device); a persisted index snapshot that would make restart O(index) was considered and rejected as doubling checkpoint I/O. Reopening a large log under a much smaller memory budget drops the keys that no longer fit the index; the drop count is exported and `recover_ok` reports failure. `Checkpoint` drains only the calling session's parked partial chunk (a stated precondition matching the harness's stop-then-checkpoint pattern), and the overflow sidecar snapshot is not a point-in-time cut with respect to concurrent writers. HydraKV also remains an embedded, single-node engine: no replication, no network service, no multi-file storage layout.

The specification's prod-grade bar. The task specification defines the goal as a real, deployable store, states that a fast number that is not correct, durable, and robust does not count, and expects durability of anything presented as prod-grade even while permitting `Checkpoint` to be a scoring no-op. Against that bar: every deployment subsystem of Sections 4.1 and 4.2 exists and is gated by an end-to-end test that fails without it, including real fault injection (disk-full, SIGKILL, external truncation), and the engine has been exercised on every operation mix the harness generates, not only the benchmark-shaped one. What the evidence still does not cover is time and scale: the stress suites run for seconds to minutes at reduced scale, no always-on endurance campaign was run, no production trace was replayed, and a leak-free ASan exit is not evidence about multi-day behavior. We therefore claim all three audits' findings are closed, not that the store is proven deployable.

Known residual risks (documented with rationale in `PROD_READINESS.md`, `WORKLOAD_HARDENING.md`, and `AUDIT3_FIXES.md`; none reachable by the scored workload, several reachable through the public interface by workloads the harness does not generate). A superseded slot can be the fingerprint-alias routing bridge to a different key in its hash group ($\approx 2^{-49}$ per key pair); compaction's verification pass checks index words and cache references, not incoming `prev` edges, so reclaiming such a bridge could orphan the alias at runtime (recovery is immune, since it scans raw slots). The overflow-absorption cap is advisory under concurrency (overshoot bounded by one value per in-flight thread) and additive to the cache sizing's slack. The fingerprint index needs $\approx$8 bytes of budget per distinct key and its words are never garbage-collected, so a store generation supports about index-capacity distinct keys and then rejects further inserts with a counted error (init warns, with the minimum budget that fits, when a size hint exceeds the capacity). X-record extents are never compacted, a documented leak class alongside poisoned slots. Read-*admission* freshness re-checks still use position order, which under position reuse is weaker than the LSN order used for serving (the served value is always the LSN winner). A read that overlaps a compaction retries once against the post-move index; a read that sampled a candidate before relocation can still transiently miss rather than retry twice. The deleted-key registry grows with live deletes and is re-populated at recovery, so delete-heavy logs cost registry RAM (charged, 64 bytes per mark); tombstone relocation is one-for-one, with no per-key coalescing; and under sustained delete storms the bounded reaper queue degrades `Delete` to its old synchronous cost, keeping memory bounded at the price of latency. Delete durability degrades fail-soft, never fail-stop: after any counted tombstoning, sync, or replay fault the intent log is retained indefinitely (re-synced at every `Checkpoint`) rather than retired, an acknowledged delete whose every durable fallback also failed on a dead disk is process-crash-safe only, and recovering from a crash inside a massive uncheckpointed delete storm queues intents past the reaper's 64K runtime bound ($\sim$72 bytes per distinct key, loudly warned). Finally, the 32-bit staging-token wrap horizon, one RMW-vs-Upsert first-touch interleaving, and formal (not observed) object-lifetime concerns for atomics in `mmap`ed memory remain open.

Evaluation scope. The adversarial matrix in Table 1 was scored with the v4c engine revision. Only the original workload was re-scored with the later revisions (each score appears in Section 6; all lie between 5.50 and 5.52 Mops/s), so the variant numbers carry the v4c engine's identity. The later revisions pass all functional tests on variant-style workloads, but we did not re-run the scored matrix. The operation-mix sweep of Table 2 widens the workload axis but consists of single runs, mixes

rows from two engine revisions as its caption states, and has no FASTER counterpart. Recovery and compaction costs appear in no scored number: the harness wipes the spill directory before every scored run and a 30-second window never crosses the compaction threshold, so those subsystems are exercised only functionally, at reduced scale, by the end-to-end suite. FASTER was measured only on the original workload. No per-variant comparison exists. We report medians with per-run values where recorded, but no dispersion statistics beyond that, and no CPU/I/O utilization breakdowns for the headline runs. The negative-result scores of Section 6 are single measurements on evolving bases. Absolute Mops/s are specific to one machine and one 30-second cold-start window. The multiplier over FASTER is a same-box, same-harness, same-workload comparison, and we make no claim about other hardware, other value sizes, longer windows, or production traces. Finally, the variants are synthetic stressors from one generator family. Surviving them is evidence against the specific overfits they probe, not evidence of production readiness.

8 Related Work

Key-value stores. FASTER [1] is the direct baseline and the closest design: hash index over a log with a memory-resident mutable region. HydraKV differs in making the RAM tier an admission-controlled cache of individual values rather than a log prefix, and in verifying every index hit against the disk-resident key. The log-plus-hash-index skeleton itself descends from Bitcask [15], whose scan-based recovery and merge compaction HydraKV reimplements with per-slot LSNs and CRCs standing in for Bitcask's per-record timestamps and checksums. SILT [16] is the closest precedent for the index discipline: partial-key fingerprints in RAM, full keys verified on flash, about one device read per lookup; HydraKV trades SILT's entropy-coded compactness for a mutable, lock-free table because half of every scored second is blind upserts. WiscKey [17] separates values from the LSM's sorted keys to cut I/O amplification; HydraKV drops the sorted structure entirely, which is only tenable because the workload has no scans. KVell [3] argued for unsorted, shared-nothing NVMe designs. HydraKV follows its spirit (no sorting, batched I/O) while keeping a shared cache because the skewed workload rewards it. uDepot [18] matches raw NVM device performance with an async task runtime and a resizable two-level hash index, but deliberately caches nothing, serving every read from the device; it targets devices fast enough to make that viable, where this benchmark's RAID0 is not. RocksDB [11] and the LSM lineage [10] target a broader operation mix (scans, transactions, compaction-based space reclamation) that this workload does not exercise. Redis [12] serves the in-memory regime and does not natively address a larger-than-memory budget. On the robustness side, the fail-soft write path (staged chunks and pinned cache entries remain authoritative in RAM after a failed write or fsync, with sticky error flags) is HydraKV's answer to the failure semantics documented by Rebello et al. [21], whose study of fsync failures found that none of five widely used stores handled them safely, losing or corrupting data, because file systems mark pages clean even when the flush failed.

Caching. The admission-and-segmentation toolkit HydraKV borrows is from the caching literature: segmented LRU [8], CLOCK-style recency [9], the TinyLFU/W-TinyLFU admission line with its doorkeeper filter [7], and the engineering synthesis in CacheLib [6]. Recent eviction work argues that simple mechanisms win at production concurrency: S3-FIFO [19] filters one-hit wonders with a small probationary FIFO and a ghost queue, and SIEVE [20] achieves state-of-the-art web-cache miss ratios with a single CLOCK-like hand and no locking on hits. Flashield [25] is the closest hybrid precedent: a DRAM-plus-SSD key-value cache that uses DRAM as an admission filter deciding which objects have earned flash residence, with a compact in-memory index over the flash-resident objects. HydraKV points the filter the other way (the SSD log is the store of record and admission protects the scarce DRAM tier), but both designs rest on the same observation, that in a two-tier system the interesting policy question is what may cross the tier boundary, not what to evict. HydraKV's memory tier reached the same conclusion under a different constraint set (a write-back tier that may not drop dirty data): probation-only insertion with one-bit second chance survived measurement, while every heavier policy tried (strict segmentation, frequency admission variants, co-admission) measured worse or failed to improve (Section 6). The workload studies that motivated the adversarial variant axes are the Facebook RocksDB characterization [4] (key-space locality, and YCSB's lack of it) and the Twitter cache analysis [5] (write-heavy skew, temporal dynamics). YCSB itself [2] defines the workload family and Zipfian setting used throughout.

AI-driven code discovery. Machine-discovered systems code now has direct precedents: AlphaDev [23] found faster sorting routines by reinforcement learning over assembly, verified against test evaluators and merged into a production standard library, and FunSearch [22] paired an LLM with a systematic evaluator in an evolutionary loop to surpass best-known results in combinatorics and bin packing. AlphaEvolve [28] scaled the same evaluator-guided evolution to whole codebases, and GEPA [29] applies it to prompts with explicit train/validation/test splits to control overfitting to the evaluation set. All of these optimize against a fixed evaluator, the condition under which proxy objectives invite gaming [24]; FunSearch and AlphaDev counter with evaluators that verify candidate outputs exactly, GEPA with data splits. The adversarial workload loop used here is a domain-specific instance of the same concern where no exhaustive verifier exists: the evaluator is made a moving, realism-constrained target, regenerated adversarially after each repair and terminated by a held-out workload, closer in spirit to adversarial evaluation practice in ML robustness than to static benchmarking. Two further differences separate this case study from the program-search systems: the artifact is three orders of magnitude larger than a priority function (a concurrent storage engine with crash recovery), and the loop's correctness half (sanitizers, coverage, cross-model review, tests-first fault injection) carried as much weight as its performance half. The agent framework, its layered architecture, and its multi-model task execution are described in the KISS Sorcar paper [27]. This paper is a case study of that framework doing sustained systems work.

9 Conclusion

HydraKV is a $\sim$4,470-line key-value store engine that outperforms FASTER by 5.9$\times$ on a demanding larger-than-memory YCSB-A configuration and carries the deployment subsystems of Sections 4.1 and 4.2 at no measurable throughput cost, and it ships with evidence (adversarial variants, a held-out workload, sanitizer-clean end-to-end tests, fault-injection tests written before their features, an all-workload sweep, same-day A/B controls, and a disclosed list of residual risks) about how far those claims generalize and where they stop. The second audit verified the performance as genuine and unhacked. The third audit found that a correctness repair had introduced the next defect, which no internal instrument caught and which fell to an external attack on the fix itself. Where a goal exceeded the measured ceiling of the design class (10 Mops/s once, 7 Mops/s a second time), Section 5.3 gives the supporting arithmetic. The development process that produced the engine, and the experimental hygiene that kept it honest, are documented in Section 6; we suspect that recipe transfers to most performance engineering of this kind, and establishing that beyond a single case study is future work.

Acknowledgments

This research is supported in part by gifts from Accenture, Amazon, AMD, Anyscale, Broadcom, Google, IBM, Intel, Intesa Sanpaolo, Lambda, Lightspeed, Mibura, NVIDIA, Samsung SDS, SAP, by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research through the X-STACK: Programming Environments for Scientific Computing program (DE-SC0021982), and the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR00112590134. We would like to thank Shubham Agarwal and Ion Stoica for providing the task descriptions, and subsequent feedback and bug reports.

References

[1] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2018.

[2] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010. DOI 10.1145/1807128.1807152.

[3] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel. KVell: the Design and Implementation of a Fast Persistent Key-Value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 447–461, 2019. DOI 10.1145/3341301.3359628.

[4] Z. Cao, S. Dong, S. Vemuri, and D. H. C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *USENIX Conference on File and Storage Technologies (FAST)*, 2020.

[5] J. Yang, Y. Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[6] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger. The CacheLib Caching Engine: Design and Experiences at Scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[7] G. Einziger, R. Friedman, and B. Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage*, 13(4), 2017. arXiv:1512.00727.

[8] R. Karedla, J. S. Love, and B. G. Wherry. Caching Strategies to Improve Disk System Performance. *IEEE Computer*, 27(3):38–46, 1994.

[9] F. J. Corbató. A Paging Experiment with the Multics System. In *In Honor of Philip M. Morse* (H. Feshbach and K. U. Ingard, eds.), MIT Press, 1969.

[10] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33:351–385, 1996. DOI 10.1007/s002360050048.

[11] Meta Platforms, Inc. RocksDB: A Persistent Key-Value Store for Fast Storage Environments. `https://rocksdb.org`, accessed 2026.

[12] Redis Ltd. Redis: An In-Memory Data Store. `https://redis.io`, accessed 2026.

[13] J. Axboe. Efficient IO with io_uring, 2019; and io_uring(7), Linux Programmer's Manual, `https://man7.org/linux/man-pages/man7/io_uring.7.html`, accessed 2026.

[14] G. L. Steele Jr., D. Lea, and C. H. Flood. Fast Splittable Pseudorandom Number Generators. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 453–472, 2014. DOI 10.1145/2660193.2660195.

[15] J. Sheehy and D. Smith. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data. Basho Technologies white paper, 2010. `https://riak.com/assets/bitcask-intro.pdf`.

[16] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–13, 2011. DOI 10.1145/2043556.2043558.

[17] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. In *USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[18] K. Kourtis, N. Ioannou, and I. Koltsidas. Reaping the Performance of Fast NVM Storage with uDepot. In *USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[19] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak. FIFO Queues Are All You Need for Cache Eviction. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 130–149, 2023. DOI 10.1145/3600006.3613147.

[20] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. V. Rashmi. SIEVE is Simpler than LRU: An Efficient Turn-Key Eviction Algorithm for Web Caches. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2024.

[21] A. Rebello, Y. Patel, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Can Applications Recover from fsync Failures? In *USENIX Annual Technical Conference (ATC)*, 2020.

[22] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, P. Kohli, and A. Fawzi. Mathematical Discoveries from Program Search with Large Language Models. *Nature*, 625:468–475, 2024. DOI 10.1038/s41586-023-06924-6.

[23] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru, E. Leurent, S. Iqbal, J.-B. Lespiau, A. Ahern, T. Köppe, K. Millikin, S. Gaffney, S. Elster, J. Broshear, C. Gamble, K. Milan, R. Tung, M. Hwang, A. T. Cemgil, M. Barekatain, Y. Li, A. Mandhane, T. Hubert,

J. Schrittwieser, D. Hassabis, P. Kohli, M. Riedmiller, O. Vinyals, and D. Silver. Faster Sorting Algorithms Discovered Using Deep Reinforcement Learning. *Nature*, 618(7964):257–263, 2023. DOI 10.1038/s41586-023-06004-9.

[24] J. Skalse, N. H. R. Howe, D. Krasheninnikov, and D. Krueger. Defining and Characterizing Reward Hacking. arXiv:2209.13085, 2022.

[25] A. Eisenman, A. Cidon, E. Pergament, O. Haimovich, R. Stutsman, M. Alizadeh, and S. Katti. Flashield: A Hybrid Key-Value Cache that Controls Flash Write Amplification. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 65–78, 2019.

[26] K. Kanellis, B. Chandramouli, T. Hart, and S. Venkataraman. From FASTER to F2: Evolving Concurrent Key-Value Store Designs for Large Skewed Workloads. *Proceedings of the VLDB Endowment*, 18(12):4910–4923, 2025. DOI 10.14778/3750601.3750615.

[27] K. Sen. KISS Sorcar: A Stupidly-Simple General-Purpose and Software Engineering AI Assistant. `https://github.com/ksenxx/kiss_ai` (paper at `papers/kisssorcar/`), 2026.

[28] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog. AlphaEvolve: A Coding Agent for Scientific and Algorithmic Discovery. arXiv:2506.13131, 2025.

[29] L. A. Agrawal, S. Tan, D. Soylu, N. Ziems, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang, C. Potts, K. Sen, A. G. Dimakis, I. Stoica, D. Klein, M. Zaharia, and O. Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2507.19457.