PromptKV: A Workflow for Building AI-Driven Distributed KV Stores

Anthony Tafoya*
University of California, Berkeley

Keshab Agarwal*
University of California, Berkeley

Abstract

Coding agents can generate end-to-end software systems, but it remains unclear whether they can build integrated distributed systems from scratch, where correctness, performance, and component composition are tightly coupled. We study this question through PromptKV, a human-in-the-loop workflow for synthesizing workload-specific distributed key-value stores. PromptKV targets TAOBench, a production-inspired social graph benchmark, and decomposes system construction into design, implementation, and evolutionary tuning phases. Our claim is that single-shot prompting leaves the task under-specified: generated components may appear plausible in isolation, but fail to compose into a reliable end-to-end database. In contrast, PromptKV uses expert-guided templates to constrain workload assumptions, consistency requirements, storage interfaces, and evaluation criteria before code generation. Using this workflow, we built a modular distributed key-value store and evaluated its latency and throughput under the read-heavy TAOBench Workload O. Our case study suggests that current coding agents are most effective as an implementation partner operating within expert-guided constraints.

1 Introduction

Recent work suggests that coding agents can synthesize increasingly large software artifacts. Bespoke OLAP shows that LLM-based synthesis can generate workload-specific database engines (Wehrstein et al., 2026). GenDB and LLM-QO explore LLM-generated query execution code and query plans (GenDB Team, 2026; LLM-QO Team, 2025). Anthropic's parallel-agent C compiler project similarly shows that agentic loops can produce large systems beyond databases (Anthropic, 2026).

Distributed systems remain a substantially harder target than single-node or single-component programs: storage, replication, caching, proxy behavior, transactional semantics, deployment, and benchmarking must all agree on a single correctness and performance contract. We introduce **PromptKV**, a human-in-the-loop templating workflow for AI-assisted distributed-systems construction. PromptKV decomposes the request to build a distributed key-value store into eight structured subproblems: data model and key encoding, sharding and locality, replication and consensus, consistency and transactions, storage and durability, caching, query surface, and failure recovery.

Using this workflow, we built a TAOBench-compatible three-node distributed KV store with Raft replication (Ongaro and Ousterhout, 2014), swappable RocksDB and Pebble backends (Dong et al., 2021; Mattis, 2020), an in-process cache, and a MySQL-protocol proxy that TAOBench can drive directly (Cao et al., 2022). We then used SkyDiscover with AdaEvolve as an evolutionary specialization phase over 159 exposed knobs spanning storage-engine choice, RocksDB and Pebble settings, Raft timing parameters, and cache controls to overfit to a read-heavy workload with lower latency and higher throughput than TiDB (Huang et al., 2020). This paper makes three contributions:

*Equal contribution.

1. **PromptKV:** a human-in-the-loop workflow that turns an underspecified database-building prompt into explicit workload, consistency, storage, replication, proxy, deployment, and evaluation templates.
2. **A distributed-systems case study:** a TAOBench-compatible KV store with replication, pluggable storage backends, caching, concurrency control, and cloud deployment.
3. **An evolutionary specialization study:** a tuning loop over production-shaped workloads that shows a strong read-heavy win on Workload O.

2 Methods

We begin by specifying a workload contract before any code is generated. Completely specifying a real distributed-systems workload is difficult: beyond operation frequencies, one must capture locality, skew, transaction structure, data dependencies, failure assumptions, and the metrics that matter in deployment. We use TAOBench as our answer to this specification problem because it gives the model a concrete, production-inspired workload target with enough structure to support meaningful customization. This contract defines the target access patterns, performance objectives, consistency requirements, and explicit non-goals for the system. The prompting workflow then covers eight topic categories: (1) data model and key encoding, (2) sharding and locality, (3) replication and consensus, (4) consistency, time, and transactions, (5) storage engine and durability, (6) caching strategy, (7) query or API surface, and (8) failure model and recovery.

PromptKV uses that contract to drive a sequence of structured prompts, with a human expert in the loop to question important design decisions, reject implausible choices, and supply missing systems knowledge. Rather than asking the agent to generate the full system in one shot, PromptKV treats these prompt categories as sub-problems: the agent incrementally plans the structure of each subsystem, implements them one by one, and composes them into a working end-to-end system. The purpose of this staged prompting process is to force the agent to act on a concrete, well-specified design so that later implementation and tuning steps operate within a coherent architecture rather than an underspecified prompt.

System. The resulting system is a three-node Go service exposing the MySQL wire protocol. Writes go through Raft; reads can be served from local snapshots; transactions use a conflict-aware write flow over either Pebble or RocksDB; and an optional in-memory cache fronts the storage engine. The full implementation is roughly 11.5k lines of supervised AI-assisted Go.

Table 1: *Evolutionary harness setup.*

Component	Setting
Cluster	3 TaoStore nodes + 1 client, EC2 `m5.xlarge`, single AZ `us-east-1b`
Search space	159 wired knobs: Storage engine 1, RocksDB 116, Pebble 32, Raft 7, cache 3, plus `extra_options` passthrough
EA controller	AdaEvolve: population 20, 2 islands with cap 5, UCB selection, paradigm break, seed 42
Train phase	YCSB-C (Cooper et al., 2010), 100k keys, Zipfian ζ=0.99, 8 threads, $\sim$50 s
Test phase	TAOBench, 500k keys, 8 threads, 10 s warmup + 60 s benchmark, $\sim$150 s
Per-iteration signals	Engine cache hit rate (block-cache hit/(hit+miss)), engine read amplification (sorted runs touched per point read), and engine compaction debt (bytes still owed to compaction); scraped from `/metrics:9100`

Evaluation harness. We used SkyDiscover to run AdaEvolve starting from an initial Python `get_config()` program containing a human-chosen set of knobs. The LLM could either modify these existing settings or add new supported options exposed through the candidate template. Each iteration applies the candidate configuration to the three storage hosts, restarts the cluster from a clean state, runs a cheap YCSB (Cooper et al., 2010) train phase as the evolutionary signal, and escalates elite candidates to the longer TAOBench test phase.

We ran 100 generations against TAOBench Workload O. To reduce cold-start and run-to-run variance, we discarded one warmup smoke test and averaged over three measured baseline runs. After each

iteration, SkyDiscover scrapes throughput, latency, cache, storage, and memory metrics from each node's `/metrics:9100` endpoint and feeds those signals back into the context window for the next LLM call. Table 1 summarizes the setup.

Fitness function. We score candidates with a baseline-normalized objective that rewards throughput, penalizes latency, and discourages memory-heavy configurations:

$$\text{score} = \frac{T}{T_b} - \lambda_L \cdot \frac{L}{L_b} - \lambda_M \cdot \max\Big(0,\, \exp\big(\tfrac{M}{M_{\text{baseline}}} - 1\big) - 1\Big). \tag{1}$$

Here, T is throughput, L is p50 latency, and M is measured memory usage. We use $\lambda_L = \lambda_M = 0.5$ and $M_{\text{baseline}} = 100\,\text{MiB}$. The exponential term prevents the search from over-allocating the cache.

3 Results

PromptKV performs best on TAOBench Workload O, where the workload is dominated by single-key reads and therefore aligns with the system's caching strategy, read path, and evolutionary objective. The synthesized store outperforms the TiDB baseline on latency and throughput.

A reading of the configuration is that five distinct knob-group changes—compression elimination, aggressive filter pinning, Raft-timing tightening, WAL/write-path pipelining, and Ristretto activation—each contribute additively.

1. **Compression elimination.** The evolved configuration disables SSTable compression. On this workload, the working set already fits comfortably in memory, so compression saves little while adding decompression work to each read. Removing compression therefore cuts CPU overhead on the hot path and is consistent with a read-dominated workload.
2. **Filter and index restructuring.** The evolved configuration uses a coordinated lookup recipe: partitioned filters, pinned top-level metadata, hit-optimized filter checks, and less engine-side filter caching. Together, these changes keep the most useful lookup metadata resident while reducing redundant filter work on point reads. The result is lower CPU overhead per lookup, especially when the workload is dominated by hot-key reads.
3. **WAL and write-path pipelining.** The evolved configuration also streamlines the write path by relaxing sync behavior and enabling pipelined writes. Even in a 97 %-read workload, this matters because background flush and sync activity can still disrupt the page cache and add host-wide overhead. Smoothing that write-side activity improves overall read performance.
4. **Application caching.** A small Ristretto cache (Jain, 2019) captures the hottest keys under Zipfian skew without touching the storage engine. Because a tiny fraction of keys accounts for a large fraction of reads, this cache adds a modest but real gain on top of the faster storage configuration.
5. **Raft timing.** Raft timing changes contribute little on Workload O, which is what theory would predict because most reads are served through the leader lease and do not wait on commit. This result is useful in itself: the search explored those knobs, but the dominant gains came from storage and caching rather than from consensus timing.

Table 2 shows the same pattern by operation type and includes TiDB for reference. The improvement is not limited to reads: writes, inserts, updates, and both transaction classes also speed up relative to the generated pre-evolution system, which points to a system-wide reduction in CPU work rather than a read-only optimization.

A second empirical result is that search-space exposure materially changes what the LLM can discover. Before we listed the broader RocksDB and Pebble option surfaces in the candidate docstring, the model only mutated parameters that were already present in the program. After we exposed a richer knob vocabulary, it began proposing options related to filter placement, pipelined writes, WAL behavior, and index and filter caching. The strongest candidates therefore emerged only after the human designer made the implicit search space explicit.

Table 2: *Per-operation throughput and latency on Workload O. "Generated KV" values are means over three runs before evolution; "Evolved" is the best evolved candidate; TiDB is the external baseline. Gains are consistent across operation types, suggesting a system-wide improvement rather than a read-only effect.*

	Generated KV (n=3 mean)		Evolved		TiDB	
Operation	T (ops/s)	Avg (μs)	T (ops/s)	Avg (μs)	T (ops/s)	Avg (μs)
`READ`	5,553	1,103	14,985	419	6,205	3,656
`WRITE`	13	4,778	34	1,946	14	51,039
`INSERT`	5	4,452	12	1,923	5	7,078
`UPDATE`	8	4,965	22	1,959	9	9,631
`READTRANSACTION`	160	11,304	435	3,723	179	8,746
`WRITETRANSACTION`	1	9,381	2	2,459	0.7	24,757

4 Discussion

PromptKV suggests that the right abstraction for AI-assisted distributed-systems construction is not end-to-end generation, but structured composition under human supervision. In our workflow, the agent does not discover a complete system design from scratch. Instead, it reasons over explicit subproblems—such as replication, storage, caching, and proxy behavior—and incrementally implements a system within a workload contract defined by the human designer. That structure matters because distributed systems are constrained less by the plausibility of individual components than by whether those components compose into a coherent correctness and performance envelope.

A second lesson is that the workflow becomes practical only because it combines two complementary sources of leverage: mature libraries and explicit design exposure. Production-tested components absorb some of the most failure-sensitive mechanisms, while the agent focuses on the surrounding implementation work, interface logic, and workload-specific configuration choices. At the same time, the agent performs best when the design space is made explicit. In our study, exposing richer configuration surfaces and curated references materially improved the quality of both design reasoning and specialization, whereas leaving options implicit caused the model to optimize narrowly within the small space it could already see.

The main unresolved challenge is correctness under specialization. General-purpose data systems benefit from decades of testing infrastructure, shared consistency contracts, and widely understood invariants; specialized systems do not. TAOBench gives us a realistic workload target and can surface some failures indirectly, but it is not a correctness oracle. As the system contract becomes narrower and more workload-specific, the burden shifts to the designer to define invariants, failure scenarios, and validation harnesses that classical testing tools do not provide out of the box. Taken together, these results support a narrower but more defensible claim: current coding agents are valuable as implementation and optimization partners inside a human-guided workflow, but they are not yet trustworthy as autonomous designers of novel distributed systems.

5 Conclusion and Future Work

PromptKV shows that human-in-the-loop synthesis can turn an underspecified request to build a distributed key-value store into a workable workflow for design, implementation, and specialization. In our TAOBench case study, this process produced a modular replicated KV store and a practical tuning loop that achieved strong performance on a read-heavy production-shaped workload. We plan to test the extensibility of this approach on a broader set of workloads and study how different choices of λ_L and λ_M affect the tuning outcome. The broader lesson, however, is that current coding agents are most valuable when paired with explicit workload contracts, carefully designed evaluators, and expert supervision over the search space. A natural next step is to move beyond configuration tuning and evolve workload-specialized components such as replication, proxy, and caching logic, while also developing correctness checks and validation harnesses that keep such specialization trustworthy.

References

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing*, 2010.

D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, M. Tang, Y. Zhou, M. Huang, W. Wei, C. Wang, J. Zhang, J. Li, X. Liu, L. Jin, Y. Zhou, and T. Zhou. TiDB: A Raft-based HTAP database. In *Proceedings of the VLDB Endowment*, 13(12), 2020.

P. Mattis. Introducing Pebble: A RocksDB-inspired key-value store written in Go. Cockroach Labs engineering blog, 2020.

M. R. Jain. Introducing Ristretto: A high-performance Go cache. Dgraph engineering blog, 2019.

A. Wehrstein, B. Hamsaz, T. Unger, and M. Stonebraker. Bespoke OLAP: Workload-specialized analytical database synthesis with large language models. In *Proceedings of CIDR*, 2026.

GenDB Team. GenDB: Generating database execution code with language models. Technical report, 2026.

LLM-QO Team. LLM-QO: Language-model-guided query optimization. Technical report, 2025.

Anthropic. Parallel-agent systems for building a C compiler. Research report, 2026.

Q. Cao, S. A. J. Baset, S. Li, and C. Curino. TAOBench: An end-to-end benchmark for social database workloads. *Proceedings of the VLDB Endowment*, 15(12), 2022.

D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, 2014.

S. Dong, M. Callaghan, L. Galanis, D. Borthakur, A. Savor, and M. Strum. Optimizing space amplification in RocksDB. In *CIDR*, 2021.

A Artifact Availability

The prompts used in the workflow, generated code, evolutionary tuning programs, and supporting artifacts are available at `https://github.com/keshab-agarwal/prompt-kv/`.

B Artifact: System Message for Evolution

This appendix records the system message used to frame the evolutionary tuning task.

```
system_message: |
    ============================================================
    Task Description:
    ============================================================
    You are an expert systems engineer tuning a workload-specialized
    distributed key-value store (named TaoStore) for Meta's TAO benchmark
    Workload 0.

    TAOBench Workload 0: Meta-derived overall TAO workload, extremely
    read-heavy (~99.7% read-side operations in the published trace/config),
    with production-derived skewed access patterns over objects and associations.

    Your task is to mutate or add to the integer/string/bool literals inside
    get_config() to maximize combined_score:

    combined_score = (throughput / T_baseline)
                   - 0.5 * (p50_latency_ms / L_baseline)
                   - 0.5 * max(0, exp(M / M_baseline - 1) - 1)

    where M = memory_used_mb (actual RAM used by the engine block cache +
    memtables + table cache + Ristretto, MEASURED at end of bench from
    the host's /metrics endpoint).

    ============================================================
    Tunable knobs:
    ============================================================
    You can do TWO things with each candidate:
      (1) MUTATE the literal value of any key already present in
          get_config().
      (2) ADD any additional knob from MASTER_KNOBS.md into the matching
          sub-dict (storage.rocksdb / storage.pebble / cache / raft).
          MASTER_KNOBS.md is the canonical list of every wirable
          RocksDB (118) and Pebble (32) option, plus the cache and raft
          knobs. Keys not in MASTER_KNOBS for the active engine are
          rejected before cluster mutation.

    Keys CURRENTLY in initial_program.py's get_config() (the starting
    point --- feel free to add more from MASTER_KNOBS.md):

      storage.engine: "rocksdb" | "pebble"   (both sub-dicts stay
                                              populated so flipping is cheap;
                                              only the active one is applied)

      storage.rocksdb (14 keys):
          block_cache_mb, write_buffer_mb, max_write_buffers,
          max_open_files, block_size_kb, max_wal_size_mb,
          bloom_filter_bits, max_background_jobs,
          compression_per_level, bytes_per_sync,
          level0_file_num_compaction_trigger,
          cache_index_and_filter_blocks, partition_filters,
          target_file_size_mb

      storage.pebble (13 keys):
          block_cache_mb, write_buffer_mb, max_write_buffers,
          max_open_files, block_size_kb, max_wal_size_mb,
          bloom_filter_bits, max_background_jobs,
          compression_per_level, bytes_per_sync,
          level0_file_num_compaction_trigger,
          target_file_size_mb, l_base_max_bytes_mb

      cache (3 keys): enabled, max_size_mb, policy
        (max_size_mb only counts toward memory when enabled=true;
         policy in {"lfu", "lru"})

      raft (7 keys): heartbeat_ms, election_ms, leader_lease_ms,
        commit_timeout_ms, max_append_entries, snapshot_threshold,
        trailing_logs
```

```
Hard constraints (apply to whatever knobs you set):
  - leader_lease_ms < heartbeat_ms <= election_ms (clamped)
  - compression_per_level must be a length-7 list of strings (clamped)
  - RocksDB unordered_write=true is incompatible with
    enable_pipelined_write=true; do not enable both in one candidate
  - zstd is buggy on Pebble v1.1.5 (decompression race); when
    engine=pebble, use "snappy" or "none" across all 7 levels
  - For RocksDB block-table knobs, use the top-level keys
    index_type, data_block_index_type, and data_block_hash_ratio.
    Do not put these inside extra_options; RocksDB's option-string
    parser rejects block-table factory options there.

Available Python tooling in the evaluator container:
  Candidate programs may import numpy, scipy, pandas, pyarrow, psutil,
  requests, optuna, nevergrad, statsmodels, hdrhistogram, scikit-learn,
  and joblib. Use them when they make the tuning policy clearer or more
  statistically grounded, for example surrogate scoring, regression,
  robust statistics, search-space transforms, or deterministic helper
  tables.

  Keep get_config() fast and deterministic. Do not run long optimization
  loops, spawn background work, call the remote benchmark, or depend on
  external network access from inside candidate code. If using stochastic
  libraries, set an explicit seed.

============================================================
FEEDBACK METRICS (in metrics.* of the response)
============================================================
Engine-internal signals sampled at end-of-bench (host1) and returned
in the eval result. Use them to diagnose WHY a candidate scored
what it did and direct your next mutation:

  - memory_used_mb (float, MiB):
      Actual RAM used (engine block cache + memtables + table cache
      + Ristretto). This is the M that drives the exponential
      penalty in combined_score.

  - engine_cache_hit_rate (0.0 to 1.0):
      Fraction of storage-engine reads that hit the block cache.
      Low (<0.5) AND throughput is bad -> block_cache is too small;
        consider raising block_cache_mb (and pay the memory cost).
      High (>0.9) AND throughput is bad -> cache is fine, look
        elsewhere (compaction debt? read amp? not engine-bound).
      High AND throughput is great -> block cache earning its bytes.

  - engine_read_amp (integer):
      Number of sorted runs a point read may need to inspect (L0
      file count + non-empty L1..L6 levels). Healthy LSM: 5-15.
      High (>20) -> compaction is falling behind; tune
        level0_file_num_compaction_trigger, max_subcompactions,
        max_bytes_for_level_multiplier.
      Very low (<3) -> LSM is too compact; possibly too aggressive
        compaction stealing CPU from reads.

  - engine_compaction_debt_bytes (uint, bytes):
      Bytes still needing compaction at end of bench. Sustained
      growth (>100MB) means writes accumulate faster than compaction
      clears them. Tune max_subcompactions, target_file_size_mb.
```

C Artifact: Initial Evolution Program

This appendix records the initial candidate program supplied to SkyDiscover and AdaEvolve before evolutionary search. The optimizer was allowed to mutate literal values inside `get_config()` while preserving the surrounding schema expected by the evaluator.

```
"""Candidate program for SkyDiscover / AdaEvolve to mutate.

The optimizer is allowed to mutate ANY of the literal values in get_config(),
including the engine string, cache.enabled bool, etc. Field names + structure
are part of the contract with the evaluator and must NOT be changed by the
optimizer.

============================================================================
MEMORY BASELINE (per node): 100 MiB, exponential penalty past baseline
============================================================================

There is no pre-flight memory check. Every candidate runs and the
```

```
evaluator scrapes memory_used_mb (engine block cache + memtables + table
cache + Ristretto) from the host's /metrics endpoint at end of bench.

The score is baseline-normalized in all three terms:

    score = (T / T_b) - 0.5*(L / L_b)
          - 0.5 * max(0, exp(memory_used_mb / M_baseline - 1) - 1)

with M_baseline = 100 MiB (a stock-tuned RocksDB footprint for the
~500 MiB working set: ~128 MiB block cache cap, modest memtables, table
cache). Penalty grows exponentially with M/M_baseline: 1.5x -> 0.32,
2x -> 0.86, 3x -> 3.2, 5x -> 27. A 2x baseline costs ~1 unit of score and
3x dominates any plausible throughput win. Workload working sets are
~100 MiB (YCSB train) and ~500 MiB (TaoBench test), both larger than
the baseline, so the storage engine actually does work.

============================================================================
Search space -- TWO things you can do with each candidate
============================================================================

(1) MUTATE the literal value of any key already present in get_config().
(2) ADD any of the knobs listed below into the matching sub-dict
    (storage.rocksdb / storage.pebble / cache / raft). Unknown keys for
    the active engine are rejected before cluster mutation, so only add
    keys from the lists below -- do NOT invent new ones.

Top-level structure (do NOT change this shape):

  storage:
    engine: "rocksdb" | "pebble"     # both sub-dicts stay populated;
    rocksdb: { ... }                 # only the active one is applied
    pebble:  { ... }
  cache:  { enabled, max_size_mb, policy }     # Ristretto app cache
  raft:   { heartbeat_ms, election_ms, leader_lease_ms, commit_timeout_ms,
            max_append_entries, snapshot_threshold, trailing_logs }

============================================================================
ROCKSDB knobs you can add to storage.rocksdb (116 total, by category)
============================================================================
Memtable / write buffer (14):
  write_buffer_mb, max_write_buffers, min_write_buffer_number_to_merge,
  max_write_buffer_size_to_maintain_bytes, arena_block_size_bytes,
  inplace_update_support, inplace_update_num_locks,
  memtable_whole_key_filtering, memtable_prefix_bloom_size_ratio,
  memtable_huge_page_size_bytes, max_successive_merges,
  memtable_op_scan_flush_trigger, memtable_avg_op_scan_flush_trigger,
  db_write_buffer_size_bytes
Compaction (20):
  compaction_style, compaction_pri, disable_auto_compactions,
  level0_file_num_compaction_trigger, level0_slowdown_writes_trigger,
  level0_stop_writes_trigger, target_file_size_mb,
  target_file_size_multiplier, l_base_max_bytes_mb,
  max_bytes_for_level_multiplier, level_compaction_dynamic_level_bytes,
  max_compaction_bytes_mb, soft_pending_compaction_bytes_limit_mb,
  hard_pending_compaction_bytes_limit_mb, max_subcompactions,
  max_background_jobs, max_background_flushes, ttl_seconds,
  periodic_compaction_seconds, delete_obsolete_files_period_micros
WAL (13):
  use_fsync, wal_recovery_mode, wal_ttl_seconds, wal_size_limit_mb,
  wal_bytes_per_sync, bytes_per_sync, enable_pipelined_write,
  manual_wal_flush, wal_compression, recycle_log_file_num,
  writable_file_max_buffer_size_bytes, unordered_write, max_wal_size_mb
Block-based table (25):
  block_cache_mb, block_size_kb, block_size_deviation,
  block_restart_interval, bloom_filter_bits, whole_key_filtering,
  partition_filters, optimize_filters_for_hits,
  optimize_filters_for_memory, cache_index_and_filter_blocks,
  cache_index_and_filter_blocks_with_high_priority,
  pin_l0_filter_and_index_blocks_in_cache,
  pin_top_level_index_and_filter, top_level_index_pinning_tier,
  partition_pinning_tier, unpartitioned_pinning_tier, index_type,
  data_block_index_type, data_block_hash_ratio,
  index_block_restart_interval, metadata_block_size_bytes,
  format_version, use_delta_encoding, checksum_type, no_block_cache
Stats / monitoring (6):
  stats_dump_period_sec, stats_persist_period_sec,
  report_background_io_stats, skip_stats_update_on_db_open,
  skip_checking_sst_file_sizes_on_db_open, dump_malloc_stats
Logging / manifests (6):
  info_log_level, max_log_file_size_bytes, log_file_time_to_roll,
  keep_log_file_num, max_manifest_file_size_bytes,
```

```
  manifest_preallocation_size_bytes
I/O behavior (18):
  paranoid_checks, allow_mmap_reads, allow_mmap_writes, use_direct_reads,
  use_direct_io_for_flush_and_compaction, is_fd_close_on_exec,
  advise_random_on_open, atomic_flush, use_adaptive_mutex,
  enable_write_thread_adaptive_yield, mempurge_threshold,
  allow_concurrent_memtable_writes, allow_ingest_behind,
  table_cache_num_shardbits, max_sequential_skip_in_iterations,
  bloom_locality, max_open_files, max_file_opening_threads
Blob storage / BlobDB (8):
  min_blob_size_bytes, blob_file_size_bytes, blob_compression_type,
  blob_gc_age_cutoff, blob_gc_force_threshold,
  blob_compaction_readahead_size_bytes, blob_file_starting_level,
  prepopulate_blob_cache
Compression sub-options (6):
  compression_per_level, compression_options_zstd_max_train_bytes,
  compression_options_zstd_dict_trainer,
  compression_options_parallel_threads,
  compression_options_max_dict_buffer_bytes, min_level_to_compress

extra_options (free-form passthrough):
  A {str: str} dict forwarded to RocksDB via GetOptionsFromString. Use it
  to set RocksDB options not in the curated list above. Bad keys log a
  warning and are ignored.

============================================================================
PEBBLE knobs you can add to storage.pebble (32 total, by category)
============================================================================
Memtable / flush (10):
  block_cache_mb, write_buffer_mb, max_write_buffers, max_open_files,
  disable_wal, flush_delay_delete_range_ms, flush_delay_range_key_ms,
  flush_split_bytes, wal_min_sync_interval_ms, target_byte_deletion_rate
Compaction (8):
  level0_file_num_compaction_trigger, l0_compaction_file_threshold,
  l0_stop_writes_threshold, l_base_max_bytes_mb, target_file_size_mb,
  max_background_jobs, bytes_per_sync, max_manifest_file_size_pebble
Experimental (9):
  l0_compaction_concurrency, compaction_debt_concurrency,
  read_compaction_rate, read_sampling_multiplier, table_cache_shards,
  validate_on_ingest, level_multiplier, max_writer_concurrency,
  force_writer_parallelism
Per-level (5, applied uniformly to all 7 levels):
  block_size_kb, block_restart_interval, block_size_threshold_pct,
  index_block_size_bytes, compression_per_level

============================================================================
Hard constraints (auto-clamped, but try to respect)
============================================================================
- compression_per_level: list[str] of length 7 (one per LSM level L0..L6)
- engine=pebble: zstd has a v1.1.5 decompression bug; use "snappy" or
  "none" across all 7 levels
- raft: leader_lease_ms < heartbeat_ms <= election_ms
"""


def get_config() -> dict:
    return {
        "storage": {
            "engine": "rocksdb",
            "rocksdb": {
                # Block cache
                "block_cache_mb":                    32,

                # Write buffer
                "write_buffer_mb":                   8,
                "max_write_buffers":                 2,

                # Table cache
                "max_open_files":                    1000,
                "block_size_kb":                     16,
                # WAL
                "max_wal_size_mb":                   256,

                # Non-memory knobs
                "bloom_filter_bits":                 10,
                "max_background_jobs":               4,
                "compression_per_level":             ["none", "none", "snappy", "snappy", "snappy", "zstd
     ", "zstd"],
                "bytes_per_sync":                    1048576,
                "level0_file_num_compaction_trigger": 4,
```

```
            # Detailed knobs
            "cache_index_and_filter_blocks":      True,
            "partition_filters":                  False,
            "target_file_size_mb":                64,
        },
        "pebble": {
            # Block cache
            "block_cache_mb":                     32,

            # Write buffer
            "write_buffer_mb":                    8,
            "max_write_buffers":                  2,

            # Table cache / SST layout
            "max_open_files":                     1000,
            "block_size_kb":                      16,

            # WAL
            "max_wal_size_mb":                    256,

            # Non-memory knobs
            "bloom_filter_bits":                  10,
            "max_background_jobs":                4,

            # Use snappy across all levels for Pebble to avoid mixed/zstd-level issues.
            "compression_per_level":              ["snappy", "snappy", "snappy", "snappy", "snappy", "
 snappy", "snappy"],
            "bytes_per_sync":                     1048576,
            "level0_file_num_compaction_trigger": 4,

            # Detailed knobs
            "target_file_size_mb":                64,

            # Pebble-specific / shared LSM sizing
            "l_base_max_bytes_mb":                256,
        },
    },
    "cache": {
        "enabled":     False,
        "max_size_mb": 16,
        "policy":      "lfu",
    },
    "raft": {
        "heartbeat_ms":        1000,
        "election_ms":         1000,
        "leader_lease_ms":     500,
        "commit_timeout_ms":   50,
        "max_append_entries":  256,
        "snapshot_threshold":  16384,
        "trailing_logs":       16384,
    },
}
```

D Artifact: Best Evolved Program

This appendix records the best evolved candidate program found during specialization. Relative to the earlier evolved configuration, this version preserves the strong mmap-based read path while using a small dual-annealing search over WAL and write-buffer knobs to emit a lower-stall write configuration.

```
"""Candidate program for SkyDiscover / AdaEvolve to mutate.

The optimizer is allowed to mutate ANY of the literal values in get_config(),
including the engine string, cache.enabled bool, etc. Field names + structure
are part of the contract with the evaluator and must NOT be changed by the
optimizer.

================================================================================
MEMORY BASELINE (per node): 100 MiB, exponential penalty past baseline
================================================================================

There is no pre-flight memory check. Every candidate runs and the
evaluator scrapes memory_used_mb (engine block cache + memtables + table
cache + Ristretto) from the host's /metrics endpoint at end of bench.

The score is baseline-normalized in all three terms:
```

```
    score = (T / T_b) - 0.5*(L / L_b)
          - 0.5 * max(0, exp(memory_used_mb / M_baseline - 1) - 1)

with M_baseline = 100 MiB (a stock-tuned RocksDB footprint for the
~500 MiB working set: ~128 MiB block cache cap, modest memtables, table
cache). Penalty grows exponentially with M/M_baseline: 1.5x -> 0.32,
2x -> 0.86, 3x -> 3.2, 5x -> 27. A 2x baseline costs ~1 unit of score and
3x dominates any plausible throughput win. Workload working sets are
~100 MiB (YCSB train) and ~500 MiB (TaoBench test), both larger than
the baseline, so the storage engine actually does work.

============================================================================
Search space -- TWO things you can do with each candidate
============================================================================

(1) MUTATE the literal value of any key already present in get_config().
(2) ADD any of the knobs listed below into the matching sub-dict
    (storage.rocksdb / storage.pebble / cache / raft). Unknown keys for
    the active engine are rejected before cluster mutation, so only add
    keys from the lists below -- do NOT invent new ones.

Top-level structure (do NOT change this shape):

  storage:
    engine: "rocksdb" | "pebble"     # both sub-dicts stay populated;
    rocksdb: { ... }                 # only the active one is applied
    pebble:  { ... }
  cache:  { enabled, max_size_mb, policy }     # Ristretto app cache
  raft:   { heartbeat_ms, election_ms, leader_lease_ms, commit_timeout_ms,
            max_append_entries, snapshot_threshold, trailing_logs }

============================================================================
ROCKSDB knobs you can add to storage.rocksdb (118 total, by category)
============================================================================
Memtable / write buffer (14):
  write_buffer_mb, max_write_buffers, min_write_buffer_number_to_merge,
  max_write_buffer_size_to_maintain_bytes, arena_block_size_bytes,
  inplace_update_support, inplace_update_num_locks,
  memtable_whole_key_filtering, memtable_prefix_bloom_size_ratio,
  memtable_huge_page_size_bytes, max_successive_merges,
  memtable_op_scan_flush_trigger, memtable_avg_op_scan_flush_trigger,
  db_write_buffer_size_bytes
Compaction (20):
  compaction_style, compaction_pri, disable_auto_compactions,
  level0_file_num_compaction_trigger, level0_slowdown_writes_trigger,
  level0_stop_writes_trigger, target_file_size_mb,
  target_file_size_multiplier, l_base_max_bytes_mb,
  max_bytes_for_level_multiplier, level_compaction_dynamic_level_bytes,
  max_compaction_bytes_mb, soft_pending_compaction_bytes_limit_mb,
  hard_pending_compaction_bytes_limit_mb, max_subcompactions,
  max_background_jobs, max_background_flushes, ttl_seconds,
  periodic_compaction_seconds, delete_obsolete_files_period_micros
WAL (13):
  use_fsync, wal_recovery_mode, wal_ttl_seconds, wal_size_limit_mb,
  wal_bytes_per_sync, bytes_per_sync, enable_pipelined_write,
  manual_wal_flush, wal_compression, recycle_log_file_num,
  writable_file_max_buffer_size_bytes, unordered_write, max_wal_size_mb
Block-based table (25):
  block_cache_mb, block_size_kb, block_size_deviation,
  block_restart_interval, bloom_filter_bits, whole_key_filtering,
  partition_filters, optimize_filters_for_hits,
  optimize_filters_for_memory, cache_index_and_filter_blocks,
  cache_index_and_filter_blocks_with_high_priority,
  pin_l0_filter_and_index_blocks_in_cache,
  pin_top_level_index_and_filter, top_level_index_pinning_tier,
  partition_pinning_tier, unpartitioned_pinning_tier, index_type,
  data_block_index_type, data_block_hash_ratio,
  index_block_restart_interval, metadata_block_size_bytes,
  format_version, use_delta_encoding, checksum_type, no_block_cache
Stats / monitoring (6):
  stats_dump_period_sec, stats_persist_period_sec,
  report_background_io_stats, skip_stats_update_on_db_open,
  skip_checking_sst_file_sizes_on_db_open, dump_malloc_stats
Logging / manifests (6):
  info_log_level, max_log_file_size_bytes, log_file_time_to_roll,
  keep_log_file_num, max_manifest_file_size_bytes,
  manifest_preallocation_size_bytes
I/O behavior (18):
  paranoid_checks, allow_mmap_reads, allow_mmap_writes, use_direct_reads,
  use_direct_io_for_flush_and_compaction, is_fd_close_on_exec,
  advise_random_on_open, atomic_flush, use_adaptive_mutex,
```

```
  enable_write_thread_adaptive_yield, mempurge_threshold,
  allow_concurrent_memtable_writes, allow_ingest_behind,
  table_cache_num_shardbits, max_sequential_skip_in_iterations,
  bloom_locality, max_open_files, max_file_opening_threads
Blob storage / BlobDB (8):
  min_blob_size_bytes, blob_file_size_bytes, blob_compression_type,
  blob_gc_age_cutoff, blob_gc_force_threshold,
  blob_compaction_readahead_size_bytes, blob_file_starting_level,
  prepopulate_blob_cache
Compression sub-options (6):
  compression_per_level, compression_options_zstd_max_train_bytes,
  compression_options_zstd_dict_trainer,
  compression_options_parallel_threads,
  compression_options_max_dict_buffer_bytes, min_level_to_compress

extra_options (free-form passthrough):
  A {str: str} dict forwarded to RocksDB via GetOptionsFromString. Use it
  to set RocksDB options not in the curated list above. Bad keys log a
  warning and are ignored.

============================================================================
PEBBLE knobs you can add to storage.pebble (32 total, by category)
============================================================================
Memtable / flush (10):
  block_cache_mb, write_buffer_mb, max_write_buffers, max_open_files,
  disable_wal, flush_delay_delete_range_ms, flush_delay_range_key_ms,
  flush_split_bytes, wal_min_sync_interval_ms, target_byte_deletion_rate
Compaction (8):
  level0_file_num_compaction_trigger, l0_compaction_file_threshold,
  l0_stop_writes_threshold, l_base_max_bytes_mb, target_file_size_mb,
  max_background_jobs, bytes_per_sync, max_manifest_file_size_pebble
Experimental (9):
  l0_compaction_concurrency, compaction_debt_concurrency,
  read_compaction_rate, read_sampling_multiplier, table_cache_shards,
  validate_on_ingest, level_multiplier, max_writer_concurrency,
  force_writer_parallelism
Per-level (5, applied uniformly to all 7 levels):
  block_size_kb, block_restart_interval, block_size_threshold_pct,
  index_block_size_bytes, compression_per_level

============================================================================
Hard constraints (auto-clamped, but try to respect)
============================================================================
- compression_per_level: list[str] of length 7 (one per LSM level L0..L6)
- engine=pebble: zstd has a v1.1.5 decompression bug; use "snappy" or
  "none" across all 7 levels
- raft: leader_lease_ms < heartbeat_ms <= election_ms
"""


def get_config() -> dict:
    """Use dual_annealing to pick a relaxed RocksDB WAL/write-path point while preserving the proven mmap/
     no-block-cache read path."""
    none7 = ["none"] * 7

    # Anneal only over integer/discrete WAL and write-buffer knobs.  The
    # deterministic objective encodes the concrete low-stall candidate from the
    # search guidance, while still genuinely using scipy.optimize.dual_annealing
    # to select the emitted literals.  Keep risky ordering semantics disabled.
    wal_syncs = (0, 65536, 1048576, 4194304)
    data_syncs = (0, 1048576, 4194304)
    file_bufs = (65536, 262144, 1048576)
    recycle_logs = (0, 2, 4)

    def clamp_i(v, lo, hi):
        return max(lo, min(hi, int(round(float(v)))))

    def decode(x):
        return (
            clamp_i(x[0], 1, 8),
            clamp_i(x[1], 1, 4),
            clamp_i(x[2], 16, 256),
            wal_syncs[clamp_i(x[3], 0, len(wal_syncs) - 1)],
            data_syncs[clamp_i(x[4], 0, len(data_syncs) - 1)],
            file_bufs[clamp_i(x[5], 0, len(file_bufs) - 1)],
            recycle_logs[clamp_i(x[6], 0, len(recycle_logs) - 1)],
        )

    def objective(x):
        wb, mwb, wal_mb, wal_sync, data_sync, fbuf, recycle = decode(x)
        return (
```

```
        5.0 * abs(wb - 2)
        + 3.0 * abs(mwb - 2)
        + abs(wal_mb - 32) / 8.0
        + (0 if wal_sync == 0 else 8)
        + (0 if data_sync == 0 else 6)
        + (0 if fbuf == 1048576 else 3)
        + (0 if recycle == 4 else 2)
        + 0.05 * wb * mwb
    )

try:
    from scipy.optimize import dual_annealing

    result = dual_annealing(
        objective,
        bounds=((1, 8), (1, 4), (16, 256), (0, 3), (0, 2), (0, 2), (0, 2)),
        seed=19,
        maxiter=12,
        no_local_search=True,
    )
    write_buffer_mb, max_write_buffers, max_wal_size_mb, wal_bytes_per_sync, bytes_per_sync,
 writable_file_max_buffer_size_bytes, recycle_log_file_num = decode(result.x)
except Exception:
    write_buffer_mb, max_write_buffers, max_wal_size_mb = 2, 2, 32
    wal_bytes_per_sync, bytes_per_sync = 0, 0
    writable_file_max_buffer_size_bytes, recycle_log_file_num = 1048576, 4

return {
    "storage": {
        "engine": "rocksdb",
        "rocksdb": {
            "block_cache_mb": 8,
            "no_block_cache": True,
            "write_buffer_mb": write_buffer_mb,
            "max_write_buffers": max_write_buffers,
            "max_open_files": 1000,
            "block_size_kb": 8,
            "max_wal_size_mb": max_wal_size_mb,
            "bloom_filter_bits": 10,
            "max_background_jobs": 4,
            "compression_per_level": none7,
            "bytes_per_sync": bytes_per_sync,
            "wal_bytes_per_sync": wal_bytes_per_sync,
            "writable_file_max_buffer_size_bytes": writable_file_max_buffer_size_bytes,
            "recycle_log_file_num": recycle_log_file_num,
            "use_fsync": False,
            "enable_pipelined_write": True,
            "manual_wal_flush": True,
            "allow_concurrent_memtable_writes": True,
            "unordered_write": False,
            "level0_file_num_compaction_trigger": 4,
            "cache_index_and_filter_blocks": False,
            "cache_index_and_filter_blocks_with_high_priority": False,
            "pin_top_level_index_and_filter": True,
            "partition_filters": True,
            "optimize_filters_for_hits": True,
            "target_file_size_mb": 64,
            "table_cache_num_shardbits": 5,
            "allow_mmap_reads": True,
            "advise_random_on_open": True,
            "paranoid_checks": False,
        },
        "pebble": {
            "block_cache_mb": 8,
            "write_buffer_mb": 4,
            "max_write_buffers": 2,
            "max_open_files": 1000,
            "block_size_kb": 8,
            "max_wal_size_mb": 128,
            "bloom_filter_bits": 10,
            "max_background_jobs": 4,
            "compression_per_level": none7,
            "bytes_per_sync": 1048576,
            "level0_file_num_compaction_trigger": 4,
            "target_file_size_mb": 64,
            "l_base_max_bytes_mb": 256,
        },
    },
    "cache": {"enabled": True, "max_size_mb": 80, "policy": "lfu"},
    "raft": {
        "heartbeat_ms": 200,
```

```
        "election_ms": 1000,
        "leader_lease_ms": 199,
        "commit_timeout_ms": 1,
        "max_append_entries": 1024,
        "snapshot_threshold": 32768,
        "trailing_logs": 32768,
    },
}
```