Published by :
https://www.ijert.org/
An International Peer-Reviewed Journal

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

KiwiDB: An Async-Native Log-Structured Merge Tree Key-Value Store

Chaitanya Paraskar
Department of Computer Engineering
Pune Institute of Computer Technology
Pune, India

Shreyash Dhas
Department of Computer Engineering
Pune Institute of Computer Technology
Pune, India

Shreyash Gajbhiye
Department of Computer Engineering
Pune Institute of Computer Technology
Pune, India

Shreyas Gopnarayan
Department of Computer Engineering
Pune Institute of Computer Technology
Pune, India

Prof. Madhuri Mane
Department of Computer Engineering
Pune Institute of Computer Technology
Pune, India

***Abstract*—Log-Structured Merge Trees (LSM trees) are the dominant write-optimized storage engine architecture, underpinning systems such as LevelDB, RocksDB, and Apache Cassandra. However, existing implementations are written in C/C++ with synchronous APIs, making them incompatible with modern asynchronous application frameworks. This paper presents KiwiDB, an LSM tree key-value store implemented from scratch in Python 3.12+. KiwiDB introduces three design innovations: (1) an async-native API where the event loop never blocks on disk I/O; (2) a parallel flush pipeline with event-chain ordered commits that achieves concurrent SSTable writes while preserving strict FIFO ordering; and (3) subprocess-based compaction that bypasses CPython's Global Interpreter Lock (GIL) for true write-path isolation. The engine employs a hybrid L0-tiering / L1-leveling compaction strategy, a concurrent skip list with lock-free reads, data-derived bloom filter sizing, and a three-tier block cache. Compared with the stock LevelDB design, KiwiDB reduces flush-time write amplification by $10\times$ and provides built-in observability through a FastAPI REST interface and a React web dashboard. The system comprises approximately 6,600 lines of type-checked Python with 211 tests across 25 modules.**

***Index Terms*—LSM tree, key-value store, async I/O, compaction, skip list, bloom filter, write-ahead log, Python, storage engine**

I. Introduction

With the rapid growth of web applications and cloud computing, key-value (KV) stores have become a critical component of modern data infrastructure. A KV store maps a set of keys to associated values and can be considered a distributed hash table. Without having to provide features usually required for a relational database system, KV stores offer higher performance, better scalability, and more availability [15]. They play a central role in data centers supporting Internet-scale services, including BigTable [5] at Google, Cassandra [4] at Facebook, and Dynamo [6] at Amazon.

The B+ tree is a common structure used in traditional databases, because its high fanout reduces the number of I/O operations for a query. However, it is highly inefficient for random insertions and updates. When there are intensive mutations, B+ trees lead to a large number of costly disk seeks and significantly degraded performance. The Log-Structured Merge Tree (LSM tree) [1] addresses this by transforming random writes into sequential writes: incoming data is sorted in an in-memory buffer and flushed to storage as a whole when the buffer is full. Many popular KV stores adopt this design, including BigTable [5], Cassandra [4], HBase [7], and LevelDB [2].

Despite the maturity of these systems, we observe two limitations that motivate this work:

(1) Synchronous APIs. LevelDB and RocksDB [3] expose blocking C++ interfaces. Modern Python applications (FastAPI, aiohttp) run on `asyncio` event loops. Integrating a synchronous storage engine requires thread-pool wrappers that add latency, complexity, and concurrency bugs. To the best of our knowledge, no existing LSM implementation offers a natively asynchronous API.

(2) Opaque internals. Production engines prioritize throughput over observability. Inspecting memtable state, bloom filter hit rates, compaction progress, or SSTable layouts requires external tooling. Internal design decisions are documented sparsely, making these systems difficult to learn from.

To address these limitations, we present KiwiDB, an LSM tree key-value store implemented from scratch in Python 3.12+. The contributions of this work are:

- KiwiDB provides an **async-native API** where all public operations (`put`, `get`, `delete`, `flush`, `close`) are async coroutines. WAL fsync, SSTable flush, and compaction execute off the event loop via `asyncio.to_thread()` and `ProcessPoolExecutor`.
- We design a **parallel flush pipeline** that writes multiple SSTables concurrently while preserving strict FIFO commit ordering via an `asyncio.Event` chain. We prove its crash-safety properties and analyze failure propagation.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

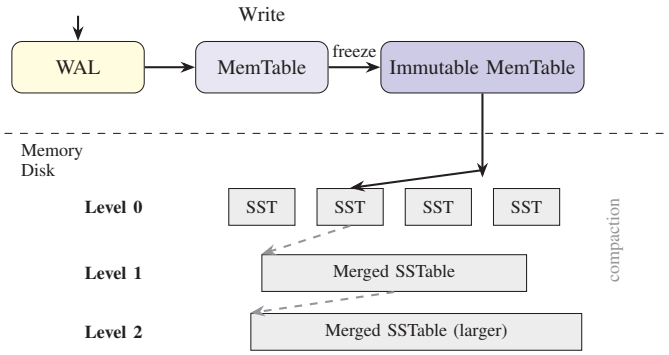

Fig. 1. Architecture of LevelDB. Writes go to the WAL and MemTable. When the MemTable is full, it becomes an Immutable MemTable and is flushed to Level 0. Compaction merges SSTables across levels.

- Compaction merges run in **subprocess workers** via `ProcessPoolExecutor`, bypassing CPython's GIL to achieve true CPU parallelism with zero impact on the write path.
- We adopt a **hybrid L0-tiering / L1-leveling** compaction strategy that reduces write amplification by $10\times$ compared to LevelDB's pure leveling, and implement a **concurrent skip list** with lock-free reads and per-node write locking.
- We provide a **built-in observability layer** with a FastAPI REST interface, a React web dashboard, and live log streaming via WebSocket — making the engine's internal state fully inspectable in real time.

The remainder of this paper is organized as follows. Section 2 provides background on LevelDB's architecture and CPython's GIL. Section 3 describes how we extend the LSM design with our proposed techniques. Section 4 discusses implementation-specific details and Python-specific challenges. We describe related work in Section 5, followed by conclusions in Section 6.

II. Background

In order to present the details of our design, we first review the architecture of LevelDB, the canonical LSM tree implementation. Then we describe CPython's Global Interpreter Lock, which motivates several of our design decisions.

A. LevelDB Architecture

LevelDB [2] is an open-source KV store that originated from Google's BigTable [5]. It is a widely studied implementation of the LSM tree [2], [14]. Figure 1 illustrates its architecture, which consists of two MemTables in main memory and a set of SSTables on disk, along with auxiliary files such as the Manifest and the write-ahead log.

Write path. When the user inserts a KV pair, it is first appended to the write-ahead log (WAL) for durability, then inserted into the MemTable, a sorted in-memory structure. When the MemTable reaches its size limit, it is converted to a read-only Immutable MemTable. A new MemTable is created for incoming writes. A background thread flushes the Immutable MemTable to disk as a Sorted String Table (SSTable). Deletes are a special case: a deletion marker (tombstone) is stored.

Level organization. SSTables are organized into levels. Level 0 is produced directly from MemTable flushes and may contain overlapping key ranges. Levels 1 and above have non-overlapping key ranges within each level. Each level has a size limit that grows exponentially: Level 1 holds up to 10 MB, Level 2 up to 100 MB, and so on.

Compaction. When the total size of Level L exceeds its limit, the compaction thread picks one SSTable from Level L and all overlapping SSTables from Level $L{+}1$, merges them, and produces new Level $L{+}1$ files. This eliminates overwritten values, drops tombstones, and ensures that fresher data resides in lower levels.

Read path. A `get(key)` request searches the MemTable, then the Immutable MemTable, then SSTables from Level 0 upward until a match is found. Since lower levels contain fresher data, the first match is authoritative. A Bloom filter [9] in each SSTable reduces unnecessary I/O by quickly ruling out files that do not contain the requested key.

Limitations. Two aspects of LevelDB's design are relevant to this work. First, there is only **one** Immutable MemTable and **one** background thread for flush and compaction. If the MemTable fills while the Immutable MemTable is still being flushed, writes are blocked. Second, the API is entirely synchronous — every operation blocks the calling thread until completion.

TABLE I
Impact of CPython's GIL on engine components and the concurrency strategy adopted by KiwiDB.

Component	Bound	GIL Impact	KiwiDB Strategy
MemTable insert	CPU	Serialized	Sync (fast, μs)
WAL fsync	I/O	GIL released	`to_thread()`
SSTable read	I/O	GIL released	`mmap` (zero-copy)
SSTable write	I/O	GIL released	Async parallel
Compaction	CPU	**Serialized**	**Subprocess**

B. CPython's Global Interpreter Lock

CPython, the reference Python implementation, uses a Global Interpreter Lock (GIL) that allows only one thread to execute Python bytecode at a time [16]. This has two consequences for a storage engine:

1) **I/O-bound work is unaffected**: Threads release the GIL during system calls (`read`, `write`, `fsync`), so I/O-bound operations like WAL append and SSTable reads achieve true concurrency.
2) **CPU-bound work is serialized**: A compaction merge that iterates millions of records in Python is CPU-bound and cannot benefit from multi-threading. True parallelism requires a separate *process* with its own GIL.

Table I summarizes the impact on each engine component and the concurrency strategy KiwiDB adopts.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

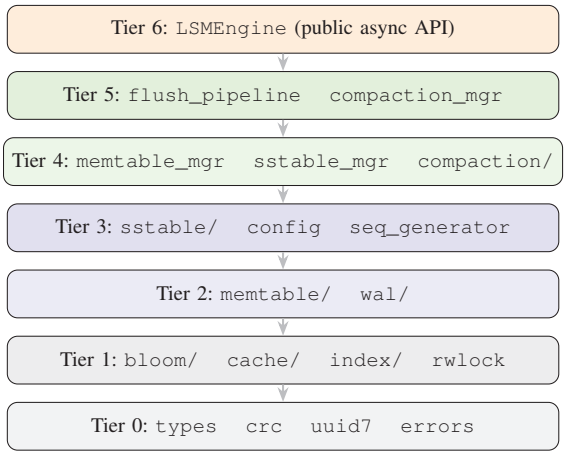

Fig. 2. KiwiDB layered architecture. Each tier depends only on tiers below it. The engine at Tier 6 is a thin coordinator.

TABLE II
KIWIDB SOFTWARE SPECIFICATION.

Property	Value
Language	Python 3.12+
Lines of code	∼6,600 (engine)
Test suite	211 tests across 25 modules
Type checking	mypy strict + basedpyright strict
Linting	ruff + bandit (security)
Build system	uv + setuptools
Web framework	FastAPI + Vite/React
Key dependencies	msgpack, mmh3, structlog, cachetools

III. DESIGN AND IMPLEMENTATION

In this section we first introduce the architectural overview of KiwiDB. Then we describe each subsystem extension over the classical LevelDB design: the async-native write and read paths, the concurrent skip list, the parallel flush pipeline, the hybrid compaction strategy, and the observability layer.

A. Overall Architecture

Figure 2 shows the overall architecture of KiwiDB, organized into a strict layered dependency model. The system consists of seven tiers, from leaf modules (Tier 0: types, CRC, UUID) up to the top-level engine coordinator (Tier 6: `LSMEngine`). No module imports from a tier above it, preventing circular dependencies and enabling each tier to be tested in isolation.

As shown in Figure 2, the software is composed of six subsystems: the Write-Ahead Log (WAL), MemTable with concurrent skip list, SSTable storage with bloom filter and sparse index, flush pipeline, compaction engine, and observability layer. The engine coordinates these subsystems through managers at Tiers 4–5.

Table II lists the software specification of KiwiDB.

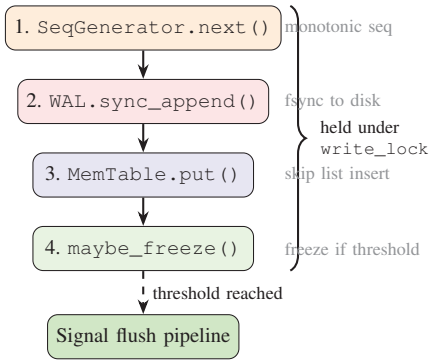

Fig. 3. KiwiDB write path. All four steps execute atomically under `write_lock`. The WAL is fsynced (Step 2) before the memtable is updated (Step 3), ensuring durability before visibility.

TABLE III
MEMTABLE FREEZE THRESHOLDS BY ENVIRONMENT.

Mode	Condition	Default	Config Key
Dev	entry count ≥ limit	10 entries	`max_memtable_entries`
Prod	byte size ≥ limit	64 MB	`max_memtable_size_mb`

B. Async-Native Write Path

All writes in KiwiDB follow the same atomic path under a single lock. The key invariant is **durability before visibility**: the WAL is fsynced before the memtable is updated. Figure 3 illustrates this four-step sequence.

Step 1: Sequence generation. `SeqGenerator.next()` atomically increments a monotonic counter under `threading.Lock`. This sequence number serves as the MVCC version — higher values indicate newer data.

Step 2: WAL durability. The entry is serialized using msgpack and framed as `[4B length][payload][4B CRC32]`. Two operation types are supported: `PUT` (op=1) and `DELETE` (op=2). The file descriptor is fsynced before returning. This is the most expensive step, but it guarantees that even if the process crashes immediately after `put()` returns, the entry is recoverable.

Step 3: MemTable insert. The KV pair is inserted into the active skip list (described in Section III-D).

Step 4: Conditional freeze. If the active memtable exceeds its threshold, it is frozen into an `ImmutableMemTable`, pushed to the immutable queue, and the flush pipeline is signaled. Table III shows the freeze conditions.

Lock ordering. KiwiDB enforces a strict ordering — `write_lock` → `wal_lock` — to prevent deadlock. The write path holds `write_lock` while calling `sync_append()`, which internally acquires `wal_lock`. Reversing this order would deadlock.

Delete vs. Put. A `delete(key)` follows the identical path but writes `OpType.DELETE` with a tombstone sentinel value `b"\x00__tomb__\x00"`. The tombstone propagates through flush and compaction until garbage-collected.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

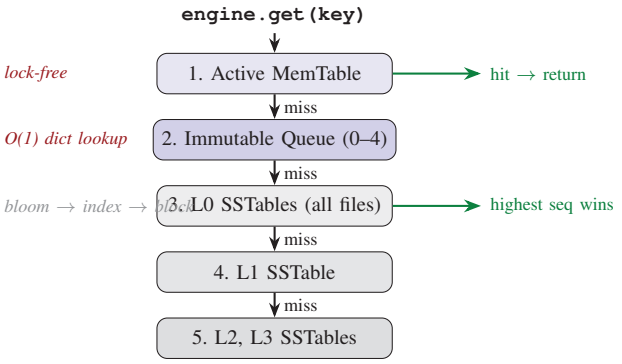

Fig. 4. KiwiDB read path. Data sources are checked newest-first. MemTable lookup is lock-free. Each SSTable uses a bloom → index → block pipeline. The first match with the highest sequence number wins.

C. Read Path

Figure 4 illustrates the read path. Reads scan data sources from newest to oldest, returning the first match with the highest sequence number. Crucially, **no write locks are acquired** — reads are non-blocking.

Step 1: The active memtable is searched via $O(\log n)$ lock-free skip list traversal.

Step 2: The immutable queue (up to 4 frozen snapshots) is scanned newest-first. Each lookup is $O(1)$ via an internal dictionary index.

Step 3: All L0 SSTables are checked because their key ranges overlap. For each file, the lookup pipeline is: *bloom filter* (rule out absent keys) → *sparse index bisect* (find candidate block) → *block scan* (sequential scan within block). The result with the highest sequence number across all L0 files is kept.

Steps 4–5: L1–L3 each have at most one SSTable. The same bloom → index → block pipeline applies, but only one file per level needs checking.

Tombstone resolution: After finding a result from any source, if the value equals the tombstone sentinel, the key has been deleted — the engine returns `None`.

Example. Suppose a user writes `put("x", "v1")` with seq=5, then `put("x", "v2")` with seq=8, then the memtable flushes to L0. A `get("x")` first checks the (now empty) active memtable, then the immutable queue, then L0. In L0, the SSTable contains both records; the one with seq=8 wins. The engine returns `"v2"`.

D. Concurrent Skip List

The memtable is backed by a concurrent skip list with 16 levels and promotion probability $p = 0.5$. Figure 5 shows the node structure and concurrency model.

Each node stores: `key`, `seq` (MVCC version), `timestamp_ms`, `value`, `forward[]` (one pointer per level), a per-node `threading.Lock`, and two boolean flags: `marked` (logical deletion) and `fully_linked` (visibility).

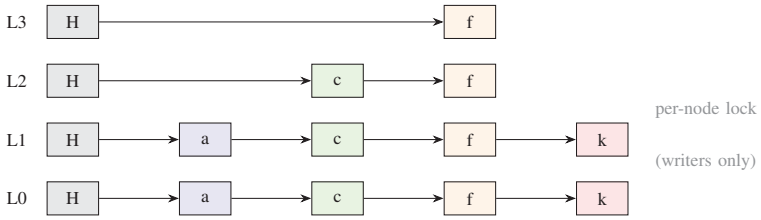

Fig. 5. Concurrent skip list with four keys. Each node has per-node `threading.Lock` for writers. Readers traverse lock-free by checking GIL-atomic `fully_linked` and `marked` boolean flags.

TABLE IV
MEMTABLE CONCURRENCY MODEL COMPARISON.

Engine	Structure	Write	Read
LevelDB	Skip list	Global mutex	Serialized
RocksDB	Skip list	Lock-free (CAS)	Lock-free
KiwiDB	Skip list	Per-node locks	Lock-free (GIL)

Writer protocol. Writers lock only the predecessor nodes at the insertion boundary. Two concurrent inserts with disjoint keys do not contend. The protocol is: (1) top-down traversal to find predecessors and successors, (2) lock predecessors, (3) validate they are still correct, (4) create the node and link at level 0, (5) link at higher levels and set `fully_linked = True`, (6) release locks.

Lock-free readers. Readers perform a top-down traversal checking the `fully_linked` and `marked` flags. Under CPython, single-attribute reads on Python objects are GIL-atomic — no torn reads occur. This makes reader traversal effectively lock-free without CAS instructions.

Table IV compares the concurrency models of the three engines.

E. SSTable Format

Each SSTable is a directory containing four files, as shown in Table V. The presence of `meta.json` is the completeness signal: if it is missing, the SSTable is considered incomplete (e.g., crash during write) and is ignored on recovery.

Record encoding. Each record uses the binary format shown in Figure 6. The CRC32 covers the entire record for integrity verification.

Writer state machine. The SSTable writer follows a strict lifecycle: `OPEN` → `put()`×N → `finish()` → `DONE`. Keys must be in ascending order (enforced). `finish()` is async for L0 flushes (writing bloom and index concurrently via `asyncio.gather`), while `finish_sync()` is synchronous for compaction subprocesses.

Reader with mmap. The reader uses memory-mapped I/O for zero-copy access to `data.bin`. Bloom filter and sparse index are loaded lazily on first `get()` and cached in the shared block cache.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

TABLE V
SSTABLE DIRECTORY LAYOUT.

File	Purpose
`data.bin`	Sorted KV records grouped into ∼4 KB blocks
`index.bin`	Sparse index: first key per block → byte offset
`filter.bin`	Bloom filter bit array + header
`meta.json`	Metadata; written **last** as completeness signal

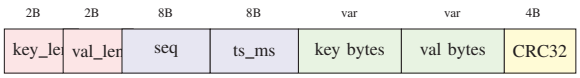

Fig. 6. SSTable record binary format. Key and value lengths are 2-byte big-endian unsigned integers. Sequence and timestamp are 8-byte big-endian signed integers.

TABLE VI
BLOOM FILTER SIZING COMPARISON.

Engine	Size Parameter	FPR Config
LevelDB	Fixed bits/key (default 10)	Fixed at creation
RocksDB	Fixed bits/key (configurable)	Per column family
KiwiDB	**Actual record count**	Per-env: 5% dev, 1% prod

TABLE VII
BLOCK CACHE TIER CONFIGURATION.

Tier	Key	Capacity	Eviction
Data blocks	offset ≥ 0	256 entries	First (lowest priority)
Sparse indexes	offset $= -2$	64 entries	Medium
Bloom filters	offset $= -1$	64 entries	Last (highest priority)

F. Bloom Filter with Data-Derived Sizing

Unlike LevelDB and RocksDB, which use a fixed bits-per-key ratio, KiwiDB sizes each bloom filter using the **actual record count** n at write time:

$$m = \left\lceil \frac{-n \cdot \ln p}{(\ln 2)^2} \right\rceil, \quad k = \left\lceil \frac{m}{n} \cdot \ln 2 \right\rceil \tag{1}$$

where p is the target false positive rate, m is the bit array size, and k is the number of hash functions (MurmurHash3 with distinct seeds). Table VI shows the difference.

Example. If LevelDB uses 10 bits/key for an SSTable with 100 entries, it allocates 1,000 bits — the same ratio it would use for 1M entries. KiwiDB computes the optimal bit count for exactly 100 entries at 1% FPR: $m = \lceil -100 \cdot \ln(0.01)/(\ln 2)^2 \rceil = 959$ bits with $k = 7$ hash functions. Every SSTable gets a right-sized filter.

G. Three-Tier Block Cache

KiwiDB uses a three-tier LRU cache that serves data blocks, sparse indexes, and bloom filters with independent eviction policies. The tiered design reflects access patterns: bloom filters should remain resident (checked on every lookup), indexes are checked on positive bloom results, and data blocks are accessed only when the index indicates a candidate.

Each tier is backed by an independent `cachetools.LRUCache` with `threading.RLock` for thread safety. Cache entries persist across engine restarts: a new `SSTableReader` instance reuses cached bloom filters and indexes from a previous session.

H. Parallel Flush Pipeline

LevelDB uses a single background thread for flushing, which means that if the working MemTable fills while the Immutable MemTable is still being written, user writes are blocked. Wang et al. [14] addressed this by increasing the number of Immutable MemTables and flushing them concurrently to separate SSD channels. KiwiDB adopts a similar insight at the software level — multiple SSTables are written concurrently — but must solve an additional problem: **commit ordering**.

On a multi-channel SSD, each SSTable lands on a separate channel and ordering is implicit. On a standard filesystem, KiwiDB must explicitly ensure that L0 files appear in oldest-first order, because the read path depends on this ordering. We solve this with an *event-chain commit mechanism*.

1) Event-Chain Mechanism: Figure 7 illustrates the mechanism with two workers. Each flush slot carries two `asyncio.Event` objects: `prev_committed` (set by predecessor) and `my_committed` (set by this slot).

Phase 1 (parallel): Up to `flush_max_workers` (default: 2) SSTable writes execute concurrently, bounded by `asyncio.Semaphore`. Each write targets a separate directory.

Phase 2 (ordered): After its write completes, each worker blocks on `prev_committed.wait()` before committing. The commit atomically: (1) registers the new SSTable reader, (2) pops the snapshot from the immutable queue, (3) truncates the WAL. Then it sets `my_committed` to unblock the next slot.

2) Correctness Invariant: The core invariant is: *at every point in time, every durable key must be findable by `get()` — either in the immutable queue or in the SSTable registry, never in neither.*

3) Crash Safety: Table VIII shows that data is recoverable at every crash point.

4) Failure Propagation: If any slot fails during Phase 1: (1) a `batch_abort` event is set, causing downstream slots to skip commits; (2) `my_committed` is set in the `finally` block, preventing chain deadlock; (3) the failed snapshot stays in the queue for retry; (4) the WAL is not truncated.

I. Hybrid Compaction

KiwiDB uses a hybrid strategy: L0 tiering with L1+ leveling. Figure 8 illustrates the level model.

Trigger mechanism. Compaction is flush-triggered, not daemon-driven. After every flush commit, `CompactionManager.check_and_compact()` runs. Table IX shows the thresholds.

K-way merge. The `KWayMergeIterator` merges N sorted inputs via a min-heap keyed on `(key, -seq)`. For

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

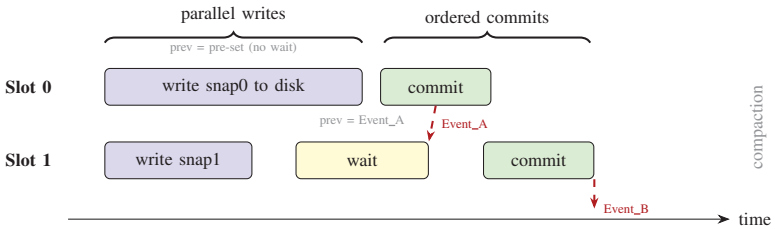

Fig. 7. Parallel flush with event-chain ordering. Writes execute concurrently (bounded by semaphore). Commits are serialized: Slot 1 waits on `Event_A` from Slot 0 before committing. Slot 1 finishes writing 200 ms early but waits — total time is max(write times) + commit overhead.

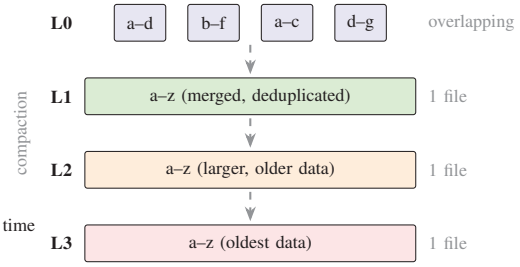

Fig. 8. KiwiDB level model. L0 uses tiering (multiple overlapping files). L1–L3 use leveling (one merged file per level). Compaction merges all L0 files into L1 when file count $\geq$ threshold.

TABLE VIII
CRASH SAFETY ANALYSIS OF THE FLUSH PIPELINE.

Crash Point	Recovery
During write	Snapshot in queue. Incomplete SSTable (no `meta.json`) ignored. WAL replays all.
After register, before pop	SSTable readable; snapshot redundant but safe. WAL deduplicates by seq.
After pop, before WAL truncate	SSTable has data. WAL replayed and deduplicated.
After WAL truncate	Clean state.

TABLE IX
COMPACTION TRIGGER THRESHOLDS.

Transition	Trigger	Default
L0 $\rightarrow$ L1	File count $\geq$ threshold	10 files
L1 $\rightarrow$ L2	Size $\geq 10^2\times$ base	Cascading
L2 $\rightarrow$ L3	Size $\geq 10^3\times$ base	Cascading

```
  "levels": {"1": "file_c", "2": "file_d"}
}
```

Listing 1. Manifest structure.

equal keys, the highest seq (newest) wins. This enables single-pass deduplication. Tombstones with `seq < seq_cutoff` are garbage-collected; those above the cutoff are preserved to shadow values in deeper levels.

Subprocess execution. Each compaction job is a `CompactionTask` — a frozen dataclass with only primitive fields (strings, integers) that can cross the subprocess boundary. The merge runs in a `ProcessPoolExecutor` subprocess with its own GIL. As shown in Table I, this is the only way to achieve true parallelism for CPU-bound work in CPython.

Atomic commit. After the subprocess produces the new SSTable, the commit sequence is: (1) register new reader, (2) write manifest, (3) mark old files for deletion (deferred by ref-count), (4) update in-memory state, (5) evict stale cache. Write locks on source and destination levels prevent reads from seeing inconsistent state.

Write amplification analysis. Table X compares the write amplification of the three strategies.

KiwiDB's L0 tiering eliminates the merge at flush — a pure sequential append. The trade-off is checking up to T bloom filters at L0 per read. With 1% FPR: $10 \times 0.01 + 3 \times 0.01 = 0.13$ expected false positives per lookup, versus LevelDB's $3 \times 0.01 = 0.03$. This $3-4\times$ read cost increase is acceptable given the $10\times$ write improvement.

J. SSTable Manager and Manifest

The `SSTableManager` coordinates all on-disk state via a persistent JSON manifest:

```
{
  "l0_order": ["file_a", "file_b"],
```

The `l0_order` list stores file IDs in newest-first order. Per-level `AsyncRWLock` instances serialize compaction commits while allowing concurrent readers. A reference-counted registry prevents reader closure while in-flight reads are active. The manifest is written atomically via `tempfile` + `os.replace()`.

K. Recovery Model

On startup, KiwiDB reconstructs state in six steps:

1) Load config and open WAL file.
2) Load all SSTables from disk via manifest. Directories without `meta.json` are discarded (incomplete writes).
3) Compute `max_seq` across all loaded SSTables.
4) Replay WAL entries where `seq > max_seq` into a fresh memtable.
5) Restore `SeqGenerator` to `max_seq`.
6) Start flush pipeline and compaction manager.

Entries already flushed to SSTables are skipped. The WAL is the source of truth for unflushed data.

L. Observability Layer

Figure 9 shows the observability architecture.

Structured logging. All engine events are logged via `structlog` with key-value fields (seq, file_id, level, size). Logs go to both a rotating file and a TCP broadcast server.

REST API. FastAPI exposes 20+ endpoints: KV operations (`GET/POST/DELETE /kv/{key}`), memtable inspection, SSTable browsing by level, engine statistics with history, configuration management, and a terminal interface.

Web dashboard. A React SPA (Vite + TypeScript) provides real-time views of memtable fill level, SSTable layout, compaction progress, live log stream, and an in-browser REPL.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

TABLE X
WRITE AMPLIFICATION COMPARISON AT FLUSH TIME. T = SIZE RATIO (DEFAULT 10), L = RECORD SIZE, B = BLOCK SIZE, ϕ = BLOOM FPR.

Strategy	Write amp. (flush)	Read amp. (bloom checks)
LevelDB (leveling)	$O(T \times L/B)$	$L \times \phi = 0.04$
KiwiDB (hybrid)	$O(L/B)$	$T\phi + L\phi = 0.13$
Improvement	**10×**	3.25× worse

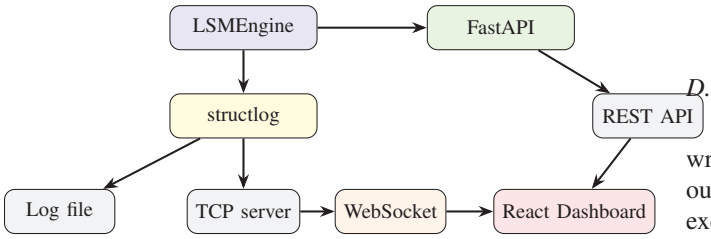

Fig. 9. Observability architecture. The engine emits structured logs to file and a TCP broadcast server. FastAPI exposes a REST API. The React dashboard consumes both REST endpoints and live WebSocket log streams.

Live streaming. The TCP log server relays events to WebSocket clients. Updates arrive without polling.

IV. IMPLEMENTATION-SPECIFIC DETAILS

In this section we discuss techniques specific to implementing an LSM engine in Python 3.12+.

A. Async/Sync Bridge

The engine uses three bridge patterns:

1) **Direct sync:** Microsecond operations (seq generation, memtable insert) execute synchronously inside an `async def` method.
2) **`asyncio.to_thread()`:** Blocking I/O (WAL fsync, mmap reads) is offloaded to the thread pool.
3) **`ProcessPoolExecutor`:** CPU-bound compaction runs in a subprocess.

B. Sync-to-Async Signal Bridging

When a sync write triggers a memtable freeze, the async flush pipeline must wake immediately. KiwiDB bridges this via `loop.call_soon_threadsafe()`: the sync code schedules an `asyncio.Event.set()` on the event loop, waking the pipeline on the next iteration. No polling; sub-millisecond latency.

C. Writer-Preferred AsyncRWLock

KiwiDB implements a custom `AsyncRWLock` with writer preference: once a writer is waiting, new readers block. This prevents writer starvation — critical for compaction commits. The lock tracks three counters: active readers, active writers (0 or 1), and waiting writers.

TABLE XI
KEY RUNTIME-MUTABLE CONFIGURATION PARAMETERS.

Parameter	Dev	Prod	Purpose
`max_memtable_entries`	10	—	Freeze (count)
`max_memtable_size_mb`	—	64	Freeze (bytes)
`l0_compact_threshold`	10	10	L0 file limit
`bloom_fpr`	0.05	0.01	Bloom FPR
`flush_max_workers`	2	2	Parallel flushes
`block_size`	4096	4096	SSTable block (B)
`max_levels`	3	3	Compaction depth

D. Backpressure

When the immutable queue reaches capacity (default: 4), writes block via `threading.Event` with a 60-second timeout. If flush is stuck, a `FreezeBackpressureTimeout` exception is raised rather than allowing unbounded memory growth.

E. Runtime-Mutable Configuration

All parameters are stored in a thread-safe `Config` object backed by a JSON file. Values are read at point of use (not cached), so changes take effect on the next operation. Writes use atomic `os.replace()`. Table XI lists key parameters.

F. Type Safety and Formal Contracts

Two independent type checkers run in strict mode on every CI check: **mypy** (disallows implicit `Any`) and **basedpyright** (additional flow analysis). Formal contracts are defined via ABCs (`StorageEngine`, `Serializable`, `MemTable`), Protocols (structural subtyping), and `.pyi` stub files for the public API.

V. RELATED WORK

LSM tree foundations. The LSM tree was proposed by O'Neil et al. [1] as a write-optimized alternative to B-trees. Luo and Carey [13] formalized the design space across tiering, leveling, size ratio, and merge policy.

Production implementations. LevelDB [2] is the canonical LSM implementation with pure leveling, a single compaction thread, and a global-mutex skip list. RocksDB [3] extends it with concurrent compaction, column families, and configurable strategies. WiredTiger [8] combines B-tree and LSM approaches. All expose synchronous APIs.

LSM on flash storage. Wang et al. [14] proposed LOCS, which extends LevelDB to exploit the channel-level parallelism of open-channel SSDs by increasing the number of Immutable MemTables and flushing them concurrently to separate channels. Their work demonstrates up to 4× throughput improvement with optimized I/O scheduling. KiwiDB shares the insight that concurrent flush improves throughput, but achieves it at the software level with an event-chain commit ordering mechanism on commodity storage.

Write amplification. Dostoevsky [11] introduces lazy leveling as a hybrid between tiering and leveling. KiwiDB's L0-tiering / L1-leveling achieves a similar reduction through a simpler design.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 06 , June - 2026

Bloom filter optimization. Monkey [10] optimizes bloom filter memory across levels. KiwiDB instead sizes each filter from actual data, producing optimal filters without cross-level tuning.

Concurrent data structures. KiwiDB's skip list draws on Herlihy et al. [12], adapted for CPython's GIL: per-node locks for writers, GIL-atomic flag reads for lock-free readers.

VI. Conclusion

We presented KiwiDB, a complete LSM tree key-value store built from scratch in Python 3.12+. By implementing the engine with an async-native API, parallel flush pipeline with event-chain commit ordering, and subprocess compaction for GIL bypass, we demonstrate that a modern storage engine can be built from first principles in a high-level language while incorporating design innovations that address real limitations in LevelDB and RocksDB.

The hybrid L0-tiering / L1-leveling strategy reduces write amplification by $10\times$ at the cost of $3.25\times$ more bloom filter checks — an acceptable trade-off. The concurrent skip list achieves lock-free reads via GIL-atomic flag checks, and the three-tier block cache keeps hot bloom filters and indexes resident.

Limitations. KiwiDB trades feature breadth for depth: no range queries (though the internal merge iterator could support them), no block compression, no multi-key transactions, and $10-100\times$ lower raw throughput than C++ engines.

Future work. Natural extensions include: (1) range query API exposing the merge iterator; (2) LZ4/zstd block compression; (3) WAL group commit for batched fsync; (4) cross-level bloom optimization following Monkey [10]; (5) migration to GIL-free Python (PEP 703) to replace subprocess compaction with true multi-threaded parallelism.

References

[1] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
[2] Google, "LevelDB: A fast and lightweight key/value database library," 2011. [Online]. Available: https://github.com/google/leveldb
[3] Facebook, "RocksDB: A persistent key-value store for fast storage environments," 2013. [Online]. Available: https://rocksdb.org
[4] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.
[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, 2008.
[6] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. ACM SOSP*, 2007, pp. 205–220.
[7] Apache, "HBase," [Online]. Available: https://hbase.apache.org
[8] MongoDB, "WiredTiger storage engine," [Online]. Available: https://source.wiredtiger.com
[9] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[10] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," in *Proc. ACM SIGMOD*, 2017, pp. 79–94.
[11] N. Dayan and S. Idreos, "Dostoevsky: Better space-time trade-offs for LSM-tree based key-value stores via adaptive removal of superfluous merging," in *Proc. ACM SIGMOD*, 2018, pp. 505–520.
[12] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, "A provably correct scalable concurrent skip list," in *Proc. OPODIS*, 2006.
[13] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *The VLDB Journal*, vol. 29, pp. 393–418, 2020.
[14] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. ACM EuroSys*, 2014, pp. 16:1–16:14.
[15] N. Leavitt, "Will NoSQL databases live up to their promise?" *Computer*, vol. 43, no. 2, pp. 12–14, 2010.
[16] Python Software Foundation, "GlobalInterpreterLock," [Online]. Available: https://wiki.python.org/moin/GlobalInterpreterLock