BtrLog: Low-Latency Logging for Cloud Database Systems

Maximilian Kuschewski
Technische Universität München
maximilian.kuschewski@tum.de

Lam-Duy Nguyen
Technische Universität München
lamduy.nguyen@tum.de

Matthias Jasny
Technische Universität Darmstadt
matthias.jasny@tu-darmstadt.de

Tobias Ziegler*
TigerBeetle
tobias@tigerbeetle.com

Viktor Leis
Technische Universität München
leis@in.tum.de

Muhammad El-Hindi
Technische Universität München
muhammad.el-hindi@tum.de

ABSTRACT

Cloud database systems cannot rely on instance-local disks for write-ahead logging (WAL) durability, forcing WAL onto remote storage. Existing options are unsatisfying: remote block storage like EBS is easy to adopt but adds substantial write latency and cost, while object storage offers excellent durability and low storage cost but is impractical for OLTP due to high latency and per-append cost. Many cloud-native databases, therefore, depend on purpose-built logging backends, which are typically proprietary and tightly coupled to engine-specific replication and recovery protocols, limiting reuse. We present BtrLog, a reusable cloud logging service that combines low-latency durable appends with low-cost archival for the common single-writer architecture. BtrLog replicates log records across a quorum of SSD-backed log nodes in a single network round trip, reducing sensitivity to stragglers in commit latency. To minimize storage cost, log nodes archive records to object storage as large segments, which are written asynchronously and off the latency-critical write path. In our evaluation, BtrLog achieves lower latency than EBS and enables higher end-to-end transaction throughput when integrated into a DBMS.

PVLDB Reference Format:
Maximilian Kuschewski, Lam-Duy Nguyen, Matthias Jasny, Tobias Ziegler, Viktor Leis, and Muhammad El-Hindi. BtrLog: Low-Latency Logging for Cloud Database Systems. PVLDB, 19(10): 2894 - 2907, 2026.
doi:10.14778/3828612.3828640

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/maxi-k/btrlog.

1 INTRODUCTION

Durable cloud logging is remote. On-premise database systems ensure durability through write-ahead logging (WAL) to a local disk or SSD. In the cloud, instance-local storage is ephemeral and can be lost on instance failure or deprovisioning [4]. Cloud database systems must therefore store their log on durable remote storage. An ideal cloud logging backend combines strong durability and availability with low-latency appends, low per-append cost, and low storage cost. Table 1 qualitatively compares logging backends along these dimensions.

Table 1: Comparison of storage options for WAL in the cloud.

		Availability	Latency	Capacity/$	Append/$
Local SSD		−	++	o	++
EBS (io2)	1 AZ	+	o	−	o
S3 Express	1 AZ	+	−	o	−
BookKeeper	1 AZ	+	o	−	o
BtrLog	1 AZ	+	+	+	+
S3	3 AZs	++	−−	+	−
BtrLog	3 AZs	++	o	+	o

Remote block storage: easy, but slow and expensive. A straightforward approach is remote block storage, such as Amazon EBS, which provides a network-attached volume that can be re-attached to a different VM after failure. EBS makes it straightforward to lift on-premise architectures to the cloud and is used in many deployments, including Amazon RDS [15]. However, even high-end EBS variants (e.g., io2) incur substantially higher write latency than instance-local SSDs and are expensive, with costs driven by both provisioned capacity and provisioned IOPS.

Object storage: great archive, poor WAL backend. Cloud object storage, such as S3, provides very high availability and durability, low cost per GB, and high throughput for scans. These properties make it attractive as a log archive, and we use it for this purpose in our design. However, placing a transactional WAL directly onto object storage (even its lower-latency variant, S3 Express) is impractical due to high latency and high per-append cost.

Log systems exist, but are not reusable. The limitations of general-purpose storage services for logging are well known, and many cloud-native database systems therefore rely on purpose-built logging backends. Examples include Microsoft Socrates' XLOG [6], Huawei TaurusDB's PLog [25], and Neon's Safekeeper [62]. AWS's proprietary internal "Journal" service is used by Amazon Aurora DSQL, S3, DynamoDB, Lambda, and Kinesis [20]. However, these systems are typically proprietary and tightly coupled to system-specific replication and recovery protocols, which makes them difficult to reuse outside their original architectures. Because they are not publicly described in sufficient detail, the community still lacks both a reusable design and a systematic understanding of the latency/cost/availability tradeoffs for cloud logging backends. In contrast, the open-source log system Apache BookKeeper [46]

*Work done while at Technische Universität München and Technische Universität Darmstadt.

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828640

was designed for a pre-cloud world and does not exploit modern low-latency networks and object storage for WAL archival.

BtrLog: fast quorum, cheap archive. We argue that logging is a fundamental primitive and should be available as reusable infrastructure. BtrLog (pronounced "better log") is a cloud logging service that combines low-latency appends with low-cost archival. It replicates log records across multiple SSD-backed log nodes; for the common single-writer architecture, each append sends one request per log node and becomes durable once a quorum (e.g., 2 of 3 nodes) has persisted it, limiting the impact of stragglers on commit latency. Once a log segment (e.g., 16 MB) fills, it is asynchronously written to S3, exploiting S3's low storage cost without suffering from its high latency. BtrLog can be deployed within a single data center (Availability Zone or AZ in AWS) for the lowest latency and cost, or across AZs to protect the unarchived log tail against datacenter-wide failures. As Table 1 shows, this design achieves high availability, low latency, and low per-append and storage cost.

Novelty. While specialized cloud logging backends already exist in the industry, they are proprietary and system-specific. The novelty of BtrLog is therefore not merely to provide another logging service, but to make this design space explicit through a reusable service tailored to cloud WAL workloads. We build on established distributed systems ideas, combining them in a way specifically tailored to cloud database logging. To the best of our knowledge, BtrLog is the first cloud logging system that simultaneously

- achieves single network round-trip append latency for the common single-writer architecture,
- provides a reusable service interface instead of coupling logging to a specific engine,
- exploits cloud object storage for low-cost, high-durability archival without placing it on the critical path, and
- is engineered for high-speed datacenter networks and NVMe SSDs, with explicit attention to tail latency and throughput.

Microbenchmarks in realistic AWS setups show that BtrLog reduces append latency by up to 4× compared to the high-end io2 variant of EBS, and end-to-end experiments with LeanStore [52] show how these gains translate into higher transaction throughput.

2 BACKGROUND: WAL IN THE CLOUD

In this section, we briefly summarize WAL semantics and the resulting requirements on a logging backend, then review existing log systems and cloud storage services. The goal is to clarify the tradeoffs that motivate our design choices.

2.1 WAL Semantics and Requirements

Log interface. Write-ahead logging (WAL) in database systems requires only a small interface. During transaction execution, the DBMS appends log records via `append` [61]. Each record is assigned a monotonically increasing *log sequence number* (LSN). At commit, the engine calls `sync` to ensure all records up to the commit LSN are persisted – the ARIES paper [61] refers to this as *forcing* the log. For restart recovery, the engine issues a `scan` operation to retrieve log records starting from a given LSN. During recovery or rollback, it calls `read` to fetch individual records by LSN for *undo* processing.

Key requirements. Because the log is the source of truth for recovery, ARIES-style logging requires durable storage that survives process, system, and device failures [61]. Accordingly, the logging backend must provide *durability* and *high availability*. It must also support *low-latency appends*, since WAL lies on the critical path of transaction processing. Cloud database systems may also operate at extreme transaction rates (e.g., 70 M/s [53]), producing massive log volumes. This makes *cost-efficiency*, both per append and per GB, another important requirement.

Single-writer semantics. A key observation underlying our work is that many cloud and distributed databases achieve durability via per-partition (or per-replica) log streams. Each log stream has an exclusive writer at any given time. In primary/secondary architectures such as Amazon Aurora [76], Microsoft Azure SQL HyperScale [6], Alibaba PolarDB [53], and Google AlloyDB [36], only the writer instance issues updates and generates the corresponding log records. Similarly, in strongly consistent distributed database systems such as Google Spanner [23], each replicated partition (Paxos group) elects a leader. All writes are ordered through that leader, yielding a single total order of writes per group [23]. Even systems that expose multi-region write APIs (e.g., Cosmos DB [60]) implement durability via locally replicated physical partitions. Ordering guarantees are defined by a leader within the associated replication scopes [60]. As we discuss next, exploiting the monotonic, single-writer append order (via sequence numbers) is key to reducing WAL latency.

2.2 Log Service Architectures

Durability and ordering guarantees. General-purpose distributed log systems, and some distributed streaming systems (e.g., Apache Kafka), provide an append-only abstraction and use replication to achieve durability and availability. In contrast to the single-writer WAL streams common in cloud database systems, they are designed for multiple concurrent writers. They therefore include a *sequencer* that orders writes across clients, linearizing appends. The need to coordinate ordering and replication adds substantial latency on the append path, and architectures differ in how these responsibilities are implemented, as Figure 1 visualizes.

Consensus-based systems (Paxos, Raft, VSR). As the left-hand side of Figure 1 shows, leader-based consensus protocols [50, 66, 67] replicate a log across a set of nodes coordinated by a leader. The leader serializes appends and replicates them to a quorum of followers, combining ordering and replication on the write path. Each append requires four network hops (or two network round trips): client → leader and leader → follower. The leader is both a throughput bottleneck and a source of latency variance.

Throughput-optimized systems (Corfu, Scalog). Corfu [12], Scalog [27], and similar systems [11, 45] decouple sequencing from storage. This avoids the leader bottleneck by scaling the sequencer layer across nodes. However, these systems still use *eager ordering* [56]: clients receive an LSN for an append once both ordering and replication are complete (`append(rec)` → `LSN`). As Figure 1 shows, this results in additional network communication across layers. The replication strategy further impacts latency: Corfu's chain replication requires six network hops; Scalog's primary-backup strategy reduces it to four, which is still well above the single-round-trip latency achievable with single-writer WAL semantics.

Lazy ordering (LazyLog). LazyLog [56] reduces append latency by separating ordering from durability. Clients send metadata to

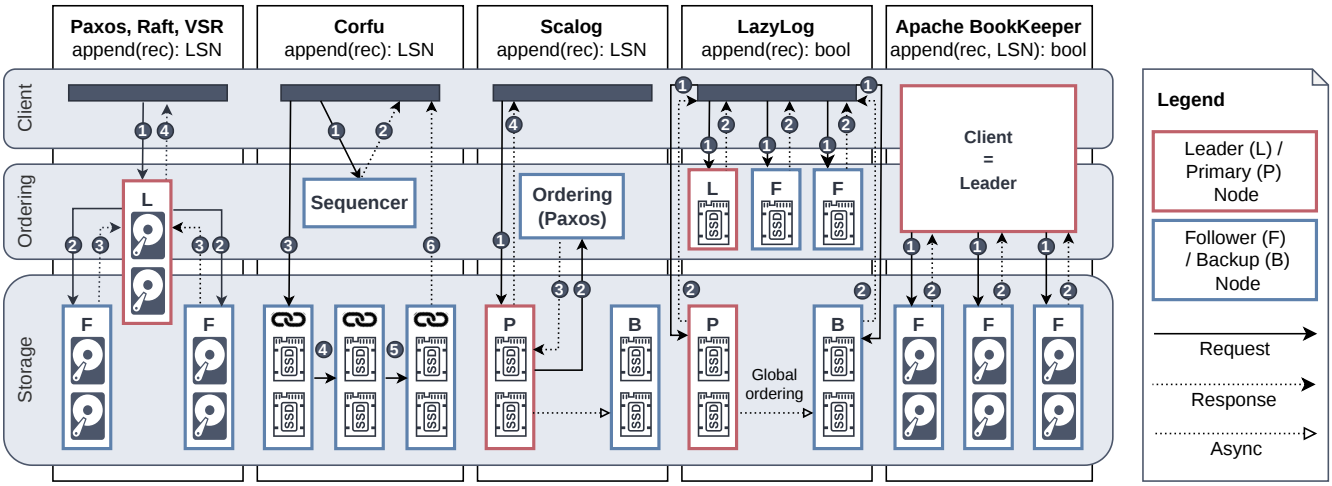


Figure 1: Comparison of existing log systems. Leader-based systems incur latency overheads due to centralized sequencing (Paxos, Corfu, Scalog). LazyLog and BookKeeper achieve optimal single-round-trip latency, which inspires BtrLog's design.

sequencers while also sending data to storage nodes in parallel. Appends are acknowledged without an LSN (`append(rec)` → `bool`) since ordering is resolved lazily during reads. This yields single round-trip appends, but leaving LSNs unspecified at append time is incompatible with ARIES-style WAL, which requires a well-defined log order (and commit LSNs) on the write path.

Single-writer systems (Apache BookKeeper). In contrast to multi-writer log systems, Apache BookKeeper [46] targets single-writer logs, as in primary/replica database architectures. On writer failure (e.g., a crash), safe failover requires additional mechanisms. BookKeeper therefore implements *fencing* and *recovery* to prevent split-brain scenarios during failover. With a single writer, the order of records is implicitly defined by the writer's append sequence. Thus, sequencing shifts to the client, eliminating the need for dedicated ordering nodes and their associated latency overheads. BookKeeper uses client-driven quorum replication; in the common case, an append completes in a single client round trip.

Implications for BtrLog. All discussed systems and architectures can provide the high durability and availability required for WAL in the cloud. However, the previous discussion shows that architectural decisions directly affect the append latency of a logging system. Similarly, cost efficiency is also impacted by architectural decisions such as the integration of cloud object storage. The rest of the paper describes how BtrLog achieves low latency and cost while meeting WAL durability and availability requirements.

3 BTRLOG DESIGN AND DEPLOYMENT

Design space. One possible solution for WAL in the cloud is writing log records directly to cheap and durable object storage. However, object storage writes take tens of milliseconds [18] and are expensive for workloads with many small writes – such as WAL. At the other extreme, one can envision a DRAM-based design similar to RAMCloud [68], where the client sends data to a quorum of nodes ("log nodes") that keep data in DRAM and provide durability through replication. This approach provides low latency but is expensive due to high DRAM cost [72]. Furthermore, it is prone to correlated failures such as data center-wide outages or software bugs. Log nodes can mitigate correlated failures by writing data to local SSDs, which are guaranteed to persist data across power-loss reboots [4]. In an extreme design, clients could persist data on remote SSDs directly using NVMe-over-TCP [65], a protocol supported by the kernel and systems such as simplyblock [71] and DAOS [74]. However, building a correct WAL backend using raw remote SSD semantics requires the client itself to handle log ownership, writer fencing for failover, remote node failures, and load balancing. BtrLog instead distributes this logic between a client library and protocol-aware log nodes, as described in the following.

3.1 System Overview

BtrLog's high-level design. BtrLog overcomes durability and cost tradeoffs by combining a staging layer for low-latency appends with object storage for durable, cost-efficient persistence. BtrLog stages appends on a cluster of log nodes and flushes them asynchronously to object storage in large segments. Log nodes retain only a bounded unflushed *tail* locally: the tail resides in RAM for fast reads and is backed by local NVMe SSDs for recovery after crashes or power loss. By flushing large segments asynchronously, BtrLog amortizes object store PUT cost and eliminates both object store latency and long-term storage capacity management. BtrLog thus consists of four components, visualized in Figure 2:

(1) a *client library* used by the DBMS to append log records,
(2) a cluster of *log nodes* that form the staging layer,
(3) a *cloud object store* for durable, low-cost storage, and
(4) *metadata storage* for log metadata and cluster configuration.

BtrLog requires few metadata operations, which can be implemented using conditional writes (*If-Match* in S3). Using S3 or a compatible service for metadata simplifies deployment and avoids a separate coordination system such as ZooKeeper or etcd.

Log abstraction and API. BtrLog exposes a single-writer, append-only log abstraction. The client library hides the internal storage tiering and provides a WAL-compatible API: `append`, `sync`, `scan`,

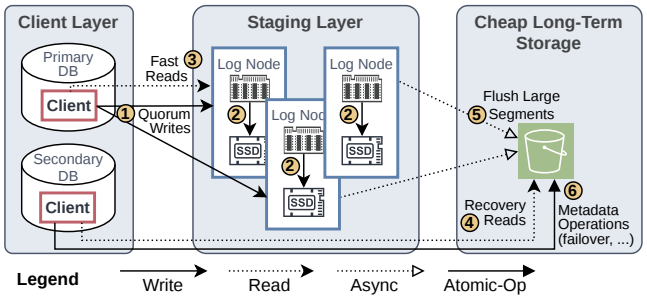


Figure 2: System overview showing the append (write) and read paths across the staging layer and object storage.

and `read`. Failover, metadata updates, and other control operations are handled transparently when the client `opens` a log for writing.

Write path. The append (write) path is BtrLog's hot path, optimized for single network round-trip commits. As Figure 2 illustrates, each client maintains a local LSN counter and replicates appends in parallel to all log nodes (1). Upon receiving an `append`, a node adds the record to an in-memory segment and writes it out-of-place [51] to a local NVMe SSD (2). Only after the SSD write and fsync complete, the node acknowledges the append to the client. The client library returns success to the DBMS after receiving acknowledgments from a quorum of nodes. If multiple appends are in flight, the client library ensures that appends are acknowledged in LSN order to preserve ARIES semantics. This client-driven sequencing eliminates a separate sequencer, enables single round-trip latency, and masks network and SSD latency jitter [38] via quorum replication.

Read path. BtrLog distinguishes between hot and cold reads. The hot, unflushed tail (e.g., for transaction rollback) is read from in-memory segments on log nodes (3). Cold entries are read from flushed segments on object storage (4) (e.g., for point-in-time recovery), leveraging its aggregate read bandwidth while reducing load on log nodes. The client routes requests transparently via `read` and `scan` based on the requested LSN range and the replicated commit watermark, and may cache recently read tail records.

3.2 Guarantees and Usage in Database Systems

Durability and correlated failures. BtrLog replicates each append to a quorum of Q_w log nodes before acknowledging it and thus tolerates up to $Q_w - 1$ node failures without data loss. For example, with 3 log nodes, $Q_w = 2$, and the system tolerates 1 node failure. Log segments are flushed asynchronously to object storage for long-term durability (5). To tolerate correlated failures (e.g., software bugs or power outages), each node persists the hot tail on its local NVMe SSD, similar to prior work on highly durable storage backends [19]. This increases append latency, but strengthens durability: if all log nodes crash because of a software fault, in-memory data is lost, whereas data persisted on local SSDs can be recovered after fixing the fault and restarting the processes.

Single writer, multiple readers. BtrLog enforces a single-writer, multi-reader abstraction, supporting use cases like log shipping. The writer maintains a commit watermark that is replicated across log nodes, allowing readers to distinguish committed from uncommitted data. Enforcing a single writer prevents inconsistent appends during DBMS failover (e.g., when replacing a failed primary).

Failure handling in the cloud. Quorum replication masks transient node failures and network jitter, providing predictable tail latency for appends. The client-side failover protocol (6) prevents split-brain during DBMS failover. BtrLog also handles compound and AZ+1 failure scenarios, as discussed in Section 4.

Database integration. As Figure 2 illustrates, the DBMS integrates the BtrLog client into its WAL subsystem rather than writing log records to a storage device. When the primary starts, or when explicitly requested, the client creates a new log on the log nodes and registers the log in the metadata management layer (c.f., Section 4), which establishes the primary as the log's single writer. It is then free to append log records as described in Section 3.1.

Transaction rollback. The client may cache recently appended records in memory to enable low-latency reads for abort and rollback. In addition, log nodes keep the hot log tail in memory, which accelerates hot reads. The hot log tail is asynchronously flushed to object storage when it fills up or becomes cold.

Database recovery. The BtrLog client supports high-throughput reads of cold log data for database recovery. For example, when a database node must replay a large log prefix to reconstruct database state, the client transparently fetches the hot tail from log nodes and streams older segments from high-bandwidth object storage [30].

Database failover. For planned maintenance or primary failure, a backup node can take over as the writer. When a BtrLog client opens an existing log for writing, it acquires ownership and fences off the previous writer to prevent conflicting appends. Section 4 details the protocol that implements these mechanisms.

Ease of use. Integrating BtrLog into an existing DBMS that relies on file system storage requires minor source code changes due to interface differences (`append` vs. `write`). Note, however, that file-system-based logging necessitates protecting against torn writes and identifying the last successful log record write during recovery. BtrLog prevents both issues due to its record-level atomicity and failover support: its interface is purpose-built for database WAL.

Parallel logging. To avoid contention on a single log, some high-performance DBMSs support parallel logging through multiple (e.g., per-thread) log streams [26, 41, 64, 78]. Such designs already manage transaction order across multiple log streams to ensure correct recovery and can thus simply use multiple BtrLog logs.

3.3 Deployment Options

Configurable availability. BtrLog supports two deployment options to adapt to different latency and durability/availability requirements: single-AZ and multi-AZ. Although we use AWS terminology, the same concepts apply to other cloud providers as well.

Single-AZ deployment. To minimize append latency, BtrLog can be deployed in the same availability zone as the client application. An AZ consists of one or more data centers within an AWS region, which are isolated from failures such as power, network outages, or flooding [2]. Services such as EBS and S3 Express operate within a single AZ. Intra-AZ network communication has latency of about 100 µs and is free of charge. BtrLog log nodes reside in partitioned placement groups [3] to reduce correlated failures like rack-level

power or network faults. If the *entire* AZ becomes unavailable (e.g., due to a major power outage), all log nodes in that AZ and BtrLog become unavailable as well. Because BtrLog writes log records to instance-local SSDs, which persist data across power-loss reboots [4], data is still durable and can be recovered upon restart. Records that were already flushed to object store remain accessible during the outage, since S3 replicates data across multiple AZs.

Multi-AZ deployment. To improve availability, BtrLog also supports deployments that span multiple AZs. A straightforward approach would place three log nodes across three AZs, ensuring that AZ failures affect only one replica. However, an additional independent failure (e.g., an SSD failure in a second AZ) can reduce a three-node deployment below the required 2/3 quorum, making the log unavailable for reads. To tolerate such AZ+1 failures [76], BtrLog uses six nodes (two per AZ). This configuration remains write-available after losing any two nodes or an entire AZ (4/6 append quorum), and remains read-available after losing up to three nodes (3/6 read quorum). Even with six nodes, BtrLog persists records to local SSDs to guard against correlated software failures.

Improving cost-efficiency in multi-AZ deployments. Besides increasing latency, multi-AZ deployments also increase cost due to cross-AZ data transfer. For example, replicating 100 million 1 KB log records from one AZ to another AZ of the same AWS region transfers 100 GB and costs $2. The overhead is amplified by quorum replication: if a client sends each append to all four remote nodes, it pays the transfer charge per remote replica (e.g., $4 \cdot \$2 = \$8$). BtrLog reduces this overhead by 2× via hierarchical replication: the client sends each record to one node per AZ, which forwards it to its peer in the same AZ. This significantly reduces cost but only slightly increases latency because intra-AZ latency (about 100 µs) is significantly lower than cross-AZ latency (about 500 µs).

4 PROTOCOLS AND FAULT TOLERANCE

Our design is simple: quorum-replicated appends to a staging layer, with asynchronous, segment-based persistence to object storage. The challenge is to preserve BtrLog's guarantees under realistic cloud failure modes – transient faults, correlated node/AZ outages, and DBMS failovers – without sacrificing one-round-trip commits. To this end, we employ a leader-based quorum protocol with view changes, similar to Apache BookKeeper, customized for BtrLog's object storage-oriented persistence path. The protocol is grounded in two invariants: (i) log records commit in order, and (ii) no committed entries are lost, even in the presence of failures within our failure model. We formalize these safety properties and an abstract model of the protocol in TLA+ and model-check them (see artifact URL). The remainder of this section gives a high-level overview of the protocol design.

Failure model. BtrLog adopts the standard non-Byzantine crash-stop failure model in asynchronous networks [22]. Accordingly, our protocol tolerates message loss, reordering, duplication, delay, network partitions, log node failures, and client failures. In the following, we discuss client and log node failures separately. Client failures coincide with database failover. Log node failures must not compromise the availability or durability of committed data in the log tail. Because cloud object storage provides high durability and availability (e.g., S3 advertises 99.99% availability and 11 nines of durability [5]), we treat object storage and the metadata store as reliable and focus on failures in the staging and replication layers.

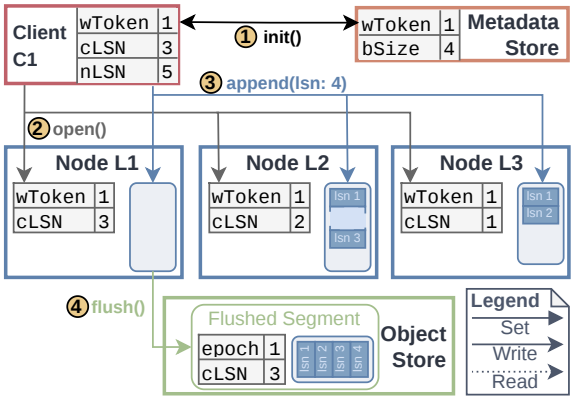


Figure 3: State for a single log without failures. The client acquires a write token (*wtoken*) from the metadata store and installs it on log nodes via open. Appends are replicated, advancing the committed watermark (*cLSN*). Log segments are flushed to object storage in large (*bSize*) segments.

Protocol overview. BtrLog targets the same failure modes as Paxos-like protocols, but shifts responsibilities to reduce latency while preserving fault tolerance. It assigns the leader/proposer role to the single writer (the client) to minimize commit latency, and leverages WAL monotonicity to avoid per-record ordering at log nodes. BtrLog integrates object storage to simplify node recovery and avoid re-replication across log nodes. Cloud object storage also serves as a highly available metadata store with atomic compare-and-swap (CAS) semantics for leader election (client failover). In the following, we discuss how BtrLog uses these concepts to achieve fault tolerance. We assume a single-AZ deployment for simplicity, but our explanations generalize to BtrLog's multi-AZ deployment. We first describe the failure-free operations that enable low-latency appends and consistent reads, then we discuss failure scenarios.

4.1 Failure-Free Operation

Log creation. Before appending, clients have to create a new log by initializing its metadata (such as the segment size (*bSize*)) in the metadata store, as shown in Figure 3 ①. To open a log for writing, the client acquires a write token (*wtoken*) from the metadata store (c.f. Section 4.2) and sends an open request to all log nodes to install that token ②. Before issuing append requests, the client waits for acknowledgments from a quorum of log nodes.

Append operations. The BtrLog client sends append requests to all log nodes and commits after replies from a write quorum ③. In addition to the payload, each append request includes three fields: (1) the record's log sequence number (nLSN), (2) the last committed LSN (cLSN, a commit watermark for readers), and (3) the record's byte offset within the current segment, computed as the prefix sum of payload sizes of preceding records in the segment. The prefix sum allows log nodes to place each record at a deterministic offset within a segment, enabling optimized flushing to object storage.

Idempotent segment flushes. On the failure-free fast path, log nodes flush full segments to object storage asynchronously ④. If

all nodes wrote the same segment, this would incur 3× the PUT cost in a 3-node deployment. To avoid redundant PUTs, BtrLog uses the prefix sum of record sizes to construct byte-by-byte identical segments across all nodes. As a result, each node derives the same deterministic object name for the segment, making the flush idempotent. Nodes create the object using a conditional PUT (with *If-None-Match* in S3), so at most one concurrent uploader succeeds. To optimize costs, a node checks whether the object already exists before issuing a PUT and skips the write if it does. Since flushes are performed asynchronously, BtrLog can use expensive cryptographic hashing to avoid accidental hash collisions. In the rare case of packet loss or writer fencing, hashes may differ; BtrLog then tolerates duplicate flushes to reduce coordination overhead and performs data de-duplication during read operations.

Read guarantees. The read protocol guarantees that all committed data is readable. Internally, BtrLog distinguishes hot reads, served by log nodes, from cold reads, served by object storage. The client handles both types of reads transparently to the DBMS.

Hot data reads. Hot data residing on log nodes is typically read either for transaction rollback (where records are read to undo a transaction), for log shipping, or for read-replication, as in Microsoft Socrates [6], for example. Because the write path tolerates message loss, committed entries may be absent on some log nodes. Similarly, the committed LSN watermark (cLSN) may be stale on nodes that have not yet received the latest appends. Hot data reads are therefore performed as quorum reads, which enable clients to tolerate gaps on individual nodes and observe up-to-date committed LSN watermarks. To optimize bandwidth utilization, readers may optimistically decide to read only from a single node and switch to quorum reads when encountering gaps or detecting a stale cLSN on that node. If the requested LSN range has already been flushed, log nodes redirect the request to object storage.

Cold data reads. BtrLog clients leverage object storage for high-throughput reads of cold data, as required for database recovery. After a crash, the DBMS replays log segments directly from object storage, reducing load on log nodes and allowing them to prioritize low-latency tail operations. To enable clients to build an index via LIST operations and use it to retrieve ranges, log nodes encode the log ID, epoch, and LSN range in addition to the hash of the data in the object key. The index can be incrementally maintained on clients using metadata from log nodes (e.g., the last flushed LSN).

Deferred eviction for low-latency reads. The core protocol guarantees that no committed data is lost, but does not by itself guarantee low read latency for all access patterns. For example, data requested for log shipping or transaction rollback may have already been evicted to object storage. In practice, object store eviction can be controlled without modifying the core protocol using the *LSN window* mechanism, which is described in Section 5.

4.2 Client Failover

Given the client's leader role, client failures require explicit failover handling. The key challenge is to preserve the single-writer invariant and prevent split-brain, in which two clients concurrently believe they hold leadership for the same log. Without fencing, concurrent append operations can create divergent tails across log nodes and conflicting persisted segments in object storage.

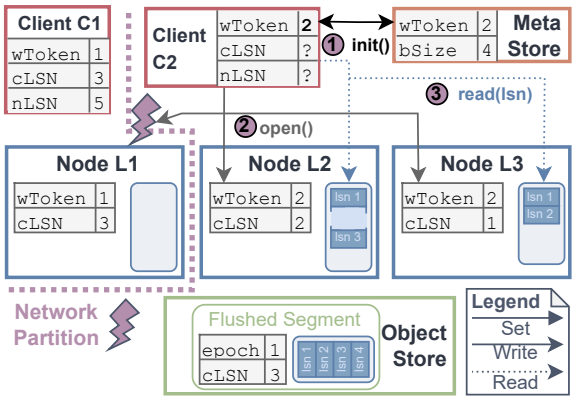


Figure 4: Client failover: The new client acquires a write token (*wtoken*) and installs it on a quorum of log nodes, fencing off the old writer. It repairs the tail by identifying the highest LSN and re-replicating records using the new *wtoken*.

Running example. We use the example depicted in Figure 4 to show how BtrLog prevents split-brain scenarios and enforces single-writer semantics through its failover protocol. In this example, client C1 is isolated in a network partition and retains connectivity only to log node L1. To replace the faulty client, a new client, C2, opens the existing log as the writer.

Writer fencing. Concurrent writes to the same log violate BtrLog's single-writer invariant. We therefore use a monotonically increasing write token (*wtoken*) to implement *writer fencing*. In Figure 4, client C2 atomically increments the log's write token from 1 to 2 ① via an atomic metadata operation. It then installs the token on log nodes by sending open and waiting for acknowledgments from a write quorum ②. A log node accepts requests only for the highest token it has observed; if it has already seen a higher token, it rejects open and returns the higher value. If the client cannot obtain a quorum of acknowledgments, it aborts the takeover and reports failure to the application (which may retry).

Avoiding data loss. Before taking over, a new writer must establish a correct log tail to preserve monotonicity and avoid losing committed data. Figure 4 highlights a subtlety introduced by object storage: Even if client C2 cannot reach log node L1, it may still observe segments that node L1 has already flushed to object storage. Naively consulting object storage during failover would add substantial latency (tens of milliseconds per access), so the protocol avoids putting object storage reads on the critical path.

Finding the log tail. Fortunately, failover does not require using object storage. To locate the tail of the log without losing committed data, the new writer determines the largest contiguous recoverable LSN prefix supported by a read quorum of log nodes. To accelerate this step, log nodes piggyback their last committed LSN watermark when acknowledging the installation of the client's write token. The client then starts reading log records sequentially ③, starting from the highest watermark it observed (cLSN = 2 in the example).

Determining committed data. The retrieved log record's LSN might be below the last committed watermark observed by a node, and only one copy of the record might exist. A stricter setting that requires the writer to observe a quorum of copies might cause data

loss, as illustrated in Figure 4. In the example, C2 observes only one copy of LSN 3 even though the previous writer replicated it to 2/3 nodes and reported success to the application (cLSN = 3 on C1). Such cases occur because quorum replication can introduce gaps, e.g., due to message loss, and the last committed watermark can lag, especially when multiple appends are in flight. To avoid data loss, new writers are conservative when inferring the log tail: they include every consecutive LSN present on at least one node in the quorum. Only LSNs that a quorum of log nodes confirms as absent could not have been committed. In the example, although LSN 4 is durably stored in object storage, C2 can infer it was not committed because a read quorum confirms it absent.

Log repair and appending. Under-replicated records (e.g., LSNs 2 and 3 in Figure 4) are more susceptible to data loss during network partitions. For example, the only accessible copy of LSN 2 is lost when node L3 fails. To improve durability of under-replicated records, the new client re-replicates such records to all nodes using the new write token ④. These repair writes may fail if the new client is also fenced, in which case it stops the failover process and informs the application to handle retries. Writer fencing can happen at any stage, but the protocol guarantees that concurrent repairs will recover the same or a higher log tail. Once fencing and repair are complete, the new client continues appending to the same log.

Reconstructing the client's log state. As described earlier, the failover protocol determines all required metadata for the new writer's state, including the last committed LSN and the next LSN (nLSN). During tail recovery, the writer also retrieves each record's byte offset from log nodes. This information is used to reconstruct the segment prefix sum for idempotent object storage flushes.

4.3 Node Failures

Detecting node failures. Although BtrLog uses a client-driven replication protocol, it does not depend on clients to detect or handle node failures. If failure detection were delegated to clients, node outages could go unnoticed when a client idles or crashes, leaving the tail unreplicated and increasing the risk of data loss. Instead, BtrLog detects node failures via peer-to-peer heartbeats between log nodes. If a node misses heartbeats from a peer for a configurable interval, it marks that peer as failed.

Handling node failures. Traditional protocols, like VSR [66], Raft [67], Corfu [24], or Scalog [27], use re-replication from surviving nodes to recover and rebuild failed nodes. Healthy nodes serve read requests to recovering nodes, increasing load. To avoid this overhead, BtrLog flushes a snapshot of the log segments to object storage rather than re-replicating the data to other nodes. The snapshot flush persists all LSNs up to the point at which the failure was detected. Entries committed *after* detection are still replicated to a quorum by design. Besides reducing load on healthy nodes, this strategy also enables new or recovering nodes to accept new writes immediately after receiving the current write token.

Cost-efficient failure handling. Flushing log segment snapshots to object storage on every node failure might, however, increase costs due to the number of PUT requests. To reduce the flush costs, BtrLog introduces two optimizations: First, log nodes trigger a flush only when they detect $N - Q_w$ node failures. For example, in a cluster with 3 nodes and $Q_w = 2$, a single failure would already trigger a flush. In a cluster of 6 nodes and $Q_w = 4$, snapshot flushes are only triggered after two failures. Second, we deterministically assign a per-log flush leader: the leader flushes immediately, while other nodes first check whether the object already exists before writing. This optimization avoids unnecessary flushes.

Node replacement and reconfiguration. BtrLog does not automatically redeploy failed nodes and assumes that a higher-level monitoring component handles such logic. Currently, faulty nodes can be replaced by removing a node and replacing it with a new node, reusing the same DNS name or IP address. More automated node replacement strategies and cluster resizing through reconfiguration are interesting avenues of future work.

4.4 Unreliable Networks

Impact of network issues. BtrLog assumes unreliable networks where append requests may get reordered or dropped entirely. Network issues can introduce gaps in node-local segments and delay the propagation of the committed watermark from the client to the log nodes. Nodes may also flush incomplete or overlapping segments to object storage, potentially leading to uncommitted or duplicated records stored there. For instance, a node may flush while client failover is in progress. Figure 4 illustrates how this failure case can cause data to appear on object storage even without being replicated and committed by quorum.

Handling uncommitted data. To keep the write path fast, BtrLog postpones handling such cases to the read path. A key invariant for the read path is that data in object storage is only *durable*, not necessarily *committed*. Readers identify committed data through the last committed watermark (cLSN) that the client propagates to log nodes, which in turn include it as object metadata on each flush. For example, when clients retrieve data with the `read` API, nodes return both the requested data and the committed LSN they observed. This mechanism enables readers to distinguish committed data from merely durable data. Similarly, when retrieving data from object storage, clients evaluate the committed watermark in the object's metadata to filter out uncommitted data if required.

Handling overlapping data. Due to quorum replication, log segments may have gaps and overlapping ranges. For instance, some nodes might observe LSN1 and LSN3, while others receive LSN2 and LSN4. When such overlapping log segments are flushed to object storage, e.g., due to a node failure, clients may observe overlapping copies of those segments. The BtrLog client transparently merges log segments while reading from object storage.

Handling duplicate data. A more subtle issue can occur during client failover. As Figure 4 illustrates, pending requests from an old writer (client C1) can fill up log segments and trigger a flush to object storage. When a new writer (client C2) takes over the log, it might append new data with the same LSN, for instance, LSN4 in Figure 4. Once the new log segment is flushed to object storage, readers would observe multiple different copies of LSN4. To resolve such conflicts in object storage, BtrLog utilizes the write token as an epoch. Each flushed object encodes the epoch in its key and includes it in its metadata. Similar to the last committed watermark, BtrLog clients use this information to ignore outdated data from prior epochs while streaming from object storage.

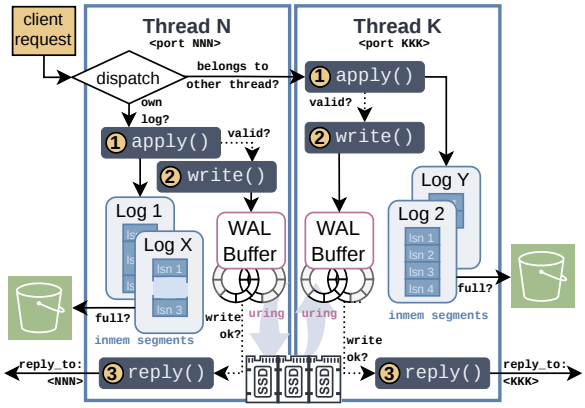


Figure 5: BtrLog log node architecture for managing parallelism across requests, CPU cores, and SSDs.

5 ENGINEERING BTRLOG FOR LOW LATENCY

Implementation overview. Achieving high throughput and low latency requires efficient protocols and an optimized implementation that can effectively utilize modern networks, NVMe SSDs, and multi-core CPUs [43]. Our prototype implements the latency-critical, failure-free append path; failure cases are modeled in TLA+.

Lower latency bound. Microbenchmarks using *sockperf* [59] and *fio* [9] in AWS show that a cluster of *c6id.metal* instances can achieve approximately 40 µs network round-trip and 30 µs SSD write latency without load in a partitioned placement group. Our implementation thus targets 70 µs latency without load. Achieving this requires careful optimization of cache misses, allocations, and system calls: For example, system calls and large allocations can take hundreds of nanoseconds or even microseconds, potentially deteriorating response latency by several percent. Thus, our technology stack needs to allow for fine-grained control over such operations.

Technology stack: Rust, io_uring, UDP. BtrLog is implemented in Rust. Since no tested off-the-shelf asynchronous Rust runtime offered consistently low latency and observability, we implemented our own. Our custom asynchronous runtime is built on *io_uring* [10], which allows batching system calls for common operations such as *send*, *recv*, and *write* [44]. Finally, since the quorum-based BtrLog protocol obviates the need for many of TCP's guarantees, we use UDP by default, avoiding stream and connection overhead.

Multi-Core architecture. BtrLog is designed as a multi-tenant cloud service, so it expects to manage many independent open log streams simultaneously. Conversely, each log requires sequential semantics internally, and the most common *append* operation needs few CPU cycles: As Figure 5 illustrates, each thread (1) checks the message and copies it to the in-memory log segment if valid, (2) asynchronously writes it to the local SSD, ensuring fsync semantics, (3) replies to the client once data is durable, and then asynchronously flushes full segments to S3 if necessary. This model naturally yields an architecture where each log is owned by a single thread that sequentially processes requests for that log.

Per-Thread runtime. I/O dominates the end-to-end latency of requests, so I/O wait times should be utilized to process other requests. One could achieve this by using synchronous I/O requests and letting the OS schedule other threads. However, the resulting thread-to-core oversubscription and excessive OS scheduling can lead to latency spikes. BtrLog thus spawns one thread per CPU core and uses cooperative scheduling within a custom asynchronous Rust runtime to interleave I/O and CPU work on each thread.

Symmetric networking. Many systems use dedicated networking threads to simplify the architecture and guarantee fast packet acceptance. However, this asymmetric approach requires cross-thread synchronization for every packet and is detrimental on small nodes with few CPUs, since one CPU thread is designated to networking only. In BtrLog, every thread accepts network requests on its own port using the *io_uring*-based runtime and processes them directly.

Internal load balancing. The symmetric networking approach has a significant drawback: If clients send requests directly to individual threads on BtrLog nodes, the node cannot balance log segments across threads. Load balancing is important since we expect multi-tenant workloads to serve tenants with both very high and very low append frequency. Having multiple high-frequency logs handled by the same thread would negatively impact peak throughput. To enable load balancing while still allowing threads to respond to clients directly and reduce cross-thread synchronization, our network protocol includes a `reply_to` field that specifies a port number. Clients can use *any* previously used port, including a well-known port for initial requests, and the BtrLog node may internally forward the message to a different thread, handing over the connection. The receiving thread responds to the client directly, setting its own port in the `reply_to` field, which the client uses for future requests – until the log moves to a different thread again. Figure 5 illustrates the resulting architecture.

S3 integration. BtrLog uses AWS's Rust S3 SDK to flush full log segments to S3. As Section 4 describes, BtrLog reduces network utilization and costs when flushing to S3: Log segments flushed to S3 by some node will not be flushed again by another. The Rust S3 SDK integrates with Rust's asynchronous ecosystem but exhibits long-running blocking operations (single-digit milliseconds) that deteriorate latency when run on regular log node worker threads. The implementation thus executes S3 operations on dedicated threads.

Concurrent appends. The BtrLog protocol permits clients to issue a bounded number of appends concurrently. We implement this via an LSN window W respected by both the client library and the log nodes: a client does not submit LSN X until a write quorum has acknowledged LSN $X - W$. Conversely, log nodes delay flushing segments overlapping with the LSN window to S3, ensuring that the corresponding records remain accessible in memory. Beyond allowing pipelining, the LSN window also serves as a retention mechanism for tail-following readers such as database secondaries, as discussed in Section 4.1. A larger window accommodates slower readers but entails higher memory usage per active log stream.

6 EVALUATION

Outline. After detailing the experimental setup, this section evaluates BtrLog across a number of dimensions, including append latency (Sec. 6.1), single-AZ and multi-AZ deployment (Sec. 6.3), impact of node failure and quorum (Sec. 6.2), cost and availability (Sec. 6.5), and end-to-end OLTP performance (Sec. 6.4).

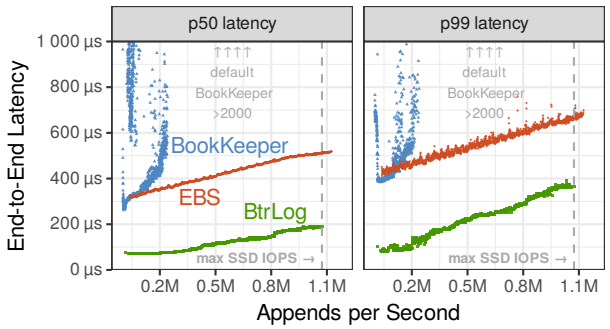


Figure 6: Latency with increasing load in EBS (♦), SSD-optimized BookKeeper (▲), and BtrLog (■).

Comparison systems. As discussed, Apache BookKeeper is the only alternative log system that eagerly assigns LSNs and executes appends with a single network round trip. Since other log systems have higher latency by design, our comparison focuses on Apache BookKeeper. We also compare BtrLog with Amazon EBS [14] in its low-latency, high-durability variant *io2*, which is a common choice for attaining durable writes on ephemeral cloud instances [69].

Experimental setup. We evaluate all systems on a cluster of three *c6id.metal* log nodes with local SSDs in the AWS *eu-central-1* region, plus a *c6in.metal* client node, which has enough network bandwidth to serve all log nodes simultaneously. The cluster uses "partitioned" placement groups, as recommended by AWS [1] for high-availability services. All measurements for an experiment are executed on the same cluster to ensure comparability. All nodes run Linux 6.14, and BtrLog uses instance-local SSDs as block devices.

BookKeeper setup. For BookKeeper, we measured both the default configuration and an optimized configuration for low-latency writes on SSDs. The default BookKeeper configuration buffers appends for up to 2 ms before group-flushing them to SSD. This buffering time is a fraction of typical HDD latency, but 60 times the write latency of a modern SSD. Our SSD-optimized BookKeeper configuration removes buffering entirely, aligns writes to 4 KiB, and disables the page cache. We combine SSDs in a RAID0 using the XFS file system, which has little overhead [37, 49].

Amazon EBS. EBS experiments directly attach an *io2* volume to the client node to achieve IOPS comparable with the BtrLog cluster. This volume is used as a block device, without a file system, similar to how BtrLog log nodes write to their local SSD.

6.1 Append Latency and Throughput

Open-loop latency under load. Response latency depends on both system configuration and load. For example, systems may batch disk writes from multiple requests to improve peak throughput at the cost of higher latency. Therefore, we vary the system load (appends per second), and record the end-to-end latency of each append. For comparability, we configure all systems to achieve the best possible latency by deactivating write batching. Our open-loop benchmark schedules requests with exponentially distributed inter-arrival times per log stream, i.e., as a Poisson point process. The client node adds more independent log streams over time, increasing system load and stress-testing many concurrent log streams.

Latency with increasing load. Figure 6 shows the median and 99th-percentile response latency for BookKeeper, EBS io2, and BtrLog with 128-byte appends, which we choose based on observed YCSB, TPC-C, and *pgbench* LSN write sizes. Each point corresponds to a 500 ms interval over which throughput and latency percentiles are measured. BookKeeper does not scale beyond 240,000 appends per second in either configuration (▲, ●), and its throughput and latency fluctuate strongly under load, producing the chaotic pattern in Figure 6. The default BookKeeper configuration is not visible in the figure because its latency never falls below 2 ms. EBS io2 (♦) shows steadily increasing write latency as load rises and consistently exhibits 4–5× higher median latency than the BtrLog prototype. At approximately 1 M appends/s, which is the IOPS limit of the instance-local SSD, both EBS and BtrLog latency deteriorate. Additional functionality such as authentication and encryption can be comfortably supported by BtrLog's remaining CPU cycles; stronger hardware-based protection using enclaves [32, 57, 58] is also an interesting avenue for future work.

Best-case latency. Let us now discuss interesting latency and system load results for each system using the data points shown in Figure 6. Across a large set of BookKeeper configurations, the best median latency we observed was 262 µs at 6,800 appends per second; latency deteriorates quickly as load increases. The best median latency on EBS was 318 µs, which increased steadily with higher load. BtrLog's best median latency is 70 µs (79 µs p99 latency) at a load of 35,500 appends per second. At half of the maximum system load, approximately 500 k appends per second, BtrLog achieves a median latency of 111 µs. This headroom would also allow adding a BtrLog gateway that exposes a simple authenticated HTTP or gRPC API. Even with an additional 40 µs of latency, such a design would remain substantially faster than EBS and BookKeeper.

Full system load. At the maximum load of 1 M appends/s (128-byte), EBS reaches 503 µs median latency and 651 µs p99 latency. Once the provisioned-IOPS volume is saturated, EBS latency increases beyond the range shown in Figure 6. BtrLog reaches 188 µs median latency at full load, limited by the 1.1 M IOPS of the underlying SSDs. To enable overload detection and to bound the latency of admitted requests, BtrLog limits its internal queue lengths and drops excess requests once the system is saturated [42]. As a result, it does not exhibit the typical "hockey stick" latency curve near full load. At peak throughput, BtrLog serves 2,186 concurrently active logs in this experiment, and log nodes use 71 GiB of main memory. While the maximum append rate is bounded by SSD IOPS, the number of logs stored is bounded by main memory because each log consumes 32 MiB on average. A production implementation should therefore flush segments of logs with very low append frequency to SSD or S3.

Impact of SSD latency. Applications with weaker durability requirements may run BtrLog with SSD writes disabled on log nodes: Each log node appends to log segments in memory and flushes them to S3 when full, but does not write its own WAL to SSD. This setting improves latency, since SSD write latency is similar to the network round-trip time in a single availability zone (35 µs vs 40 µs). We find that disabling SSDs reduces median latency by approximately 50% and improves tail latency across most throughput settings.

Comparison with EBS. The large latency gap between EBS and BtrLog has a plausible architectural explanation: EBS provides

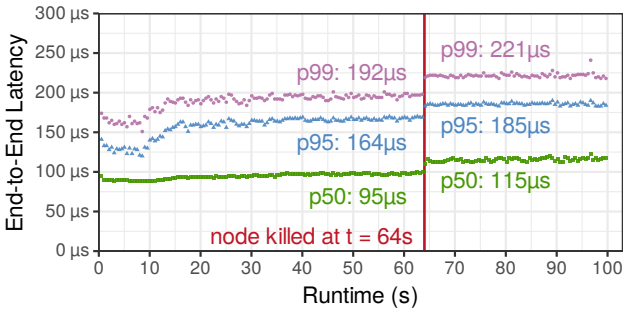


Figure 7: End-to-End latency before and after killing one log node at 400,000 log appends per second.

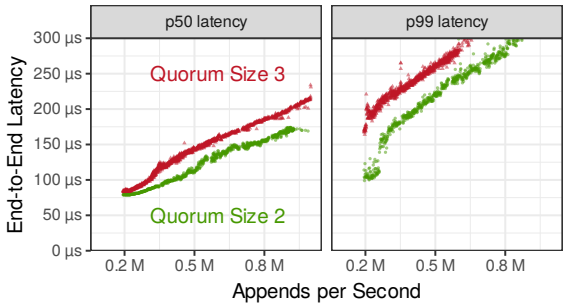


Figure 8: Impact of required quorum (2 or 3 nodes) on median and tail latency.

durability using chain replication [21]. As discussed in Section 2.2, chain replication requires more network hops than BtrLog's client-driven quorum replication on the append path. More broadly, remote block storage exposes a general-purpose block device abstraction rather than an append-only interface and therefore cannot optimize as aggressively for small WAL appends. To assess whether this behavior is specific to AWS, we also measured public block storage services on GCP and Azure and compared them to the corresponding network-plus-local-SSD baseline (evaluated with *sockperf* and *fio* using *n4-standard-2 + hyperdisk-balanced* in GCP *europe-west3*, and *Standard_D4s_v3 + PremiumV2_LRS* in Azure *germanywestcentral*). We find that all hyperscaler block storage services exhibit high write latency compared to the optimum:

	AWS	GCP	Azure
Network + SSD	76 µs	71 µs	301 µs
Remote Block Storage Service	311 µs	453 µs	776 µs
Factor	4.1×	6.4×	2.6×

These measurements suggest that block storage services in general are not optimized for low latency WAL appends.

Comparison with BookKeeper. Unlike EBS, BookKeeper and BtrLog share the same replication mechanism: both use client-driven quorum replication and can acknowledge appends in a single round trip. To assess whether their remaining latency gap is primarily due to language-level effects, we conducted microbenchmarks showing that Java's networking and SSD I/O primitives can achieve latency comparable to their Rust and C counterparts. We therefore attribute the gap mainly to implementation choices in BookKeeper, which was designed around millisecond-scale HDD latency rather than modern low-latency networks and NVMe SSDs.

6.2 Impact of Node Failure and Quorum

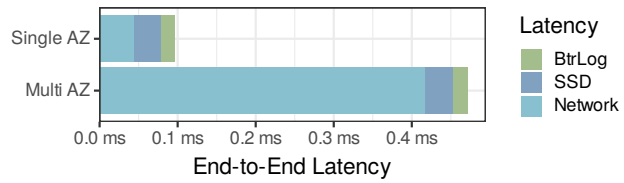
Impact of node failure. Our experimental setup with three log nodes can tolerate one node failure, since a quorum of two nodes suffices for appends. The following experiment tests the append availability and latency of the BtrLog prototype under node failure by manually stopping a random log node while the benchmark is running. The load is kept constant at approximately 400,000 appends per second. As Figure 7 shows, the client keeps successfully appending log entries on the BtrLog cluster after a node is killed at $t = 64s$. The median latency deteriorates from 95 µs to 115 µs (21%), and the p99 latency from 192 µs to 221 µs (15%).

Discussion: quorum latency. BtrLog's quorum-based algorithm does not need to wait for all responses to arrive, thereby hedging against network latency jitter. Killing one node effectively stops request hedging and causes the latency increase visible in Figure 7.

Impact of quorum size. BtrLog can be configured with different quorum sizes. The default number of responses required for a successful append operation is a majority, e.g., two out of three nodes. In the following experiment, we investigate the effect of disabling quorum for append requests, so that they require responses from all nodes. As Figure 8 shows, the quorum-based protocol significantly improves end-to-end latency: At half system load (500,000 appends/s), the quorum improves median latency by approximately 30 µs and p99 latency by approximately 50 µs.

6.3 Cross-AZ Latency

Network latency dominates. Previous experiments used a log cluster inside a single availability zone (AZ). This yields better latency and availability comparable to EBS, but may not provide enough availability for some use cases. To validate that BtrLog, like in a single AZ, achieves latency close to the hardware limit, we now spread the three-node cluster across multiple AZs of the same region (eu-central-1). In this configuration, the client sees idle-load median network latency of 270 µs, 440 µs, and 540 µs to the three log nodes as measured using sockperf [59]. Note that these numbers may vary by region. As the following figure shows, the overall append request latency in BtrLog is indeed dominated by cross-AZ network latency (400,000 appends per second):



As the previous single-AZ experiments showed, BtrLog achieves 4× better latency than EBS io2. In a multi-AZ configuration with three nodes, BtrLog achieves comparable latency (12% worse) to a single-AZ provisioned EBS volume at the same throughput while providing multi-AZ availability and a WAL-optimized interface.

Figure 9: YCSB-A transaction throughput with different WAL backends in LeanStore.

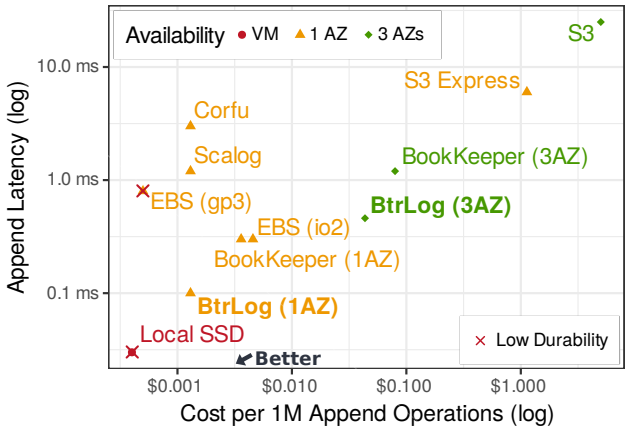


Figure 10: Comparison of append cost, append latency, and availability of potential WAL backends for the cloud. BtrLog shrinks the gap between local SSDs and alternative solutions.

6.4 End-to-End Database Performance

Integration in LeanStore. Experiments so far have used a custom open-loop benchmarking client designed to measure response latency as accurately as possible under different load settings. We now turn towards evaluating the performance of these systems in a transactional database, LeanStore [52]. LeanStore's autonomous commit protocol [64] uses multiple log streams (one per worker thread) that are flushed independently. This design was conceived for local SSDs, which require many parallel writes to saturate bandwidth, but it also fits the BookKeeper and BtrLog APIs, since their multi-tenant architecture supports multiple independent log streams. We modify LeanStore by swapping its WAL implementation with different backends: Custom-built backends for BookKeeper and BtrLog, and the default block-device backend for EBS.

Transaction Throughput. Figure 9 shows the measured YCSB-A (50% reads, 50% writes) throughput using different WAL backends in LeanStore. LeanStore using the BtrLog WAL backend achieves 2× higher throughput than LeanStore using the BookKeeper backend, 3× higher throughput than EBS gp3, and 1.25× higher throughput than EBS io2. These results show that systems previously using EBS io2 for their WAL can replace EBS with a non-provisioned, multi-tenant service while simultaneously increasing transaction throughput using a simple *append* interface.

6.5 WAL Backend Tradeoffs

Choosing a cloud WAL backend requires balancing append latency with cost and availability. While the previous results focused on latency, Figure 10 also compares cost and fault-tolerance for BtrLog and existing alternatives across all three dimensions.

Append latency and availability. We measure the 1 KiB append latency using each backend's native append interface in microbenchmarks. In contrast to Section 6.1, we optimize for throughput rather than latency to compare per-append costs: All systems batch four 1 KiB appends into one 4 KiB I/O. Corfu, Scalog, and BookKeeper benchmarks use the implementations provided by each GitHub repository [8, 24, 70]; S3 Standard and S3 Express benchmarks use the AWS C++ S3 SDK; EBS gp3 and io2 benchmarks use *fio* [9]. Corfu and Scalog run in a 400 Gbit/s lab environment; other measurements use AWS *c6id* instances in *eu-central-1*.

Availability. Instance-local NVMe SSD writes on EC2 take approximately 35 µs, but provide neither durability nor availability under instance failure. Figure 10 thus classifies them as low availability (red). Deployments replicated across nodes within a single AZ provide higher availability (orange), while deployments spanning multiple AZs, including S3 and the corresponding variants of BtrLog and BookKeeper, provide the highest availability (green).

Cost. To compare services with different pricing models, we normalize all costs to *per-append* cost, shown on the x-axis of Figure 10. Cloud-native services such as S3 and S3 Express already use per-operation pricing. For provisioned services (EBS) and hosted systems such as Apache BookKeeper and BtrLog, we derive per-append cost assuming full resource utilization.

Detailed cost calculation. For EBS, we assume peak provisioned performance (256,000 IOPS for io2 and 80,000 IOPS for gp3) and consider only IOPS cost, excluding storage-capacity cost. Provisioning 256,000 IOPS on EBS io2 costs $9,651.20 per month, or, assuming a 30-day month, $\$ 1.45 \times 10^{-8}$ per I/O operation. With full IOPS utilization, this corresponds to $0.0145 per 1 M I/O operations and thus $0.0036 per 1 M appends. For hosted systems, including BtrLog, Apache BookKeeper, Corfu, and Scalog, we use the same IOPS-based calculation and assume *c6id.metal* instances, each providing 1,073,336 IOPS across four local SSDs. Under this model, single-AZ BtrLog yields a cost of $0.00125 per 1 million appends, while its additional S3 PUT cost is negligible because appends are batched before upload (for 16 MB batches and 1 KB requests, the S3 PUT cost is approximately $\$ 3 \times 10^{-10}$ per append). Apache BookKeeper incurs higher cost because it dedicates SSD devices to read operations, reducing the IOPS available for writes. For multi-AZ deployments of BtrLog and BookKeeper, we additionally account for the larger deployment size (six rather than three nodes) and cross-AZ network transfer cost, again assuming 1 KB requests.

Discussion of results. As Figure 10 shows (on a logarithmic scale), BtrLog yields the best cost/latency tradeoff in both the single-AZ and multi-AZ availability classes. Note that these calculations assume the best case for all systems, even those that were not able to fully utilize the SSD IOPS budget in our experiments.

7 RELATED WORK

In addition to prior discussions, we review related work on cloud-native databases, shared-log designs, and cloud storage backends.

Disaggregated and cloud databases. Decoupling and optimizing logging functionality is a common pattern in modern cloud database systems. Amazon Aurora [76] uses a log-centric design with a highly coupled log and storage component to reduce network traffic and improve fault tolerance. Microsoft Socrates [6] implements a dedicated XLOG service with quorum-based commit, and Huawei TaurusDB [25] uses distinct log nodes (PLog) to reduce commit latency. OceanBase PALF [39] and FoundationDB [83] similarly utilize a dedicated log component in their design. Neon separates PostgreSQL compute from a replicated WAL service ("safekeepers") that provides durable WAL ingestion and supports failover [63]. Many of these systems, e.g., Aurora, Socrates, OceanBase, and Neon, also integrate object storage for low-cost storage. Although these systems highlight the need for a specialized logging component, their components are tightly coupled with the database engine and are not described sufficiently in the literature. In contrast, BtrLog aims to provide a reusable write-ahead logging system that can be integrated with database engines, supporting the development of modular and composable cloud database architectures [54, 79].

Shared logs and distributed log systems. Besides industrial systems, log abstractions have also been studied in academic research. Hyder proposed using a single totally ordered log on shared flash as the backend for OLTP [16], motivating systems such as Corfu [13]. However, shared-log systems including Corfu, Delos, and Scalog emphasize multi-writer semantics and throughput rather than low latency [11, 13, 27]. Milliscale [82] recently proposed using S3 Express for OLTP to achieve scalable multi-millisecond latency, while BtrLog targets scalable multi-microsecond latency. LazyLog [56] and SpecLog [17] explore lazy or speculative binding of records to log positions to reduce append latency, and FuzzyLog [55] relaxes ordering to partial orders to increase concurrency. These works, however, introduce new append semantics in which LSNs are unknown at commit time, making them incompatible with common recovery protocols such as ARIES.

Log-centric streaming systems. There has also been work on distributed log systems for streaming. LogDevice is a deprecated *distributed log system* developed by Facebook that was designed to support generic record-oriented and append-only use cases such as write-ahead logging for durability and stream processing [33]. Kafka [48] popularized the replicated commit-log model for high-throughput streaming and introduced tiered storage [29] to archive old log segments to object storage. Streaming-oriented systems such as Apache Pulsar [7] and Pravega [75], built on top of Apache BookKeeper [46] as a backend for durability, also provide data offloading to object storage. BtrLog complements these works by optimizing for databases instead of streaming use cases: it targets the single-writer, low-latency append pattern of database WAL and leverages ARIES-style semantics to minimize commit overhead, while still supporting asynchronous archival to object storage. Still, the prevalence of the log abstraction in other systems motivates us to explore using BtrLog for other latency-sensitive use cases.

Low-latency durable commits. Related work has studied how fast networks and memory can reduce commit latency. FaRM [28] and RAMCloud [68] replicate logs to remote DRAM to make commits durable with microsecond-scale overhead. QueryFresh [77] uses an RDMA-accessible NVRAM log as primary storage to enable fresh reads on standbys. In contrast to these works, BtrLog aims to achieve low-latency durability on commodity cloud infrastructure.

Using cloud storage for databases. Cloud storage characteristics shape database design beyond WAL. Durner et al. study how to use object storage for analytics despite higher latency and different access patterns [30]. Other benchmarking efforts characterize cloud storage services and database I/O behavior in cloud-native systems [81]. Early work evaluated alternative database architectures on cloud primitives such as EC2/EBS/S3 and quantified their performance implications [47]. More recent studies characterize cloud database system architecture tradeoffs [40, 84] and the implications of object vs. block storage latency/variance [73]. Instead of adapting database systems to different storage backends, BtrLog provides a specialized WAL storage backend that requires minimal database adaptation and transparently tiers storage for cold data.

8 SUMMARY & FUTURE WORK

Summary. We present BtrLog, a reusable single-writer logging service purpose-built for cloud-native database systems. It combines low-latency durable appends with low-cost, highly durable archival on cloud object storage. Our evaluation shows that BtrLog improves latency by up to 4× relative to the commonly used EBS io2, translating to higher end-to-end transaction throughput in OLTP systems. Unlike EBS, BtrLog does not require provisioned IOPS or capacity when run as a multi-tenant service.

Beyond database systems. Logging is a primitive with a multitude of applications, and although BtrLog is designed as a WAL backend for database systems, we believe its low latency and cost can also be advantageous in other use cases. Like BookKeeper, BtrLog can be used as a storage backend for streaming systems such as Apache Pulsar. Using an additional ordering layer, it can also support multi-writer semantics, similar to Corfu and Scalog.

Towards extensibility. BtrLog provides a reusable interface and is not coupled to a specific database engine. One can conceive of further BtrLog extension points in an open ecosystem. For example, custom log segment filtering and transformation functions on log nodes enable writing to S3 in application-specific formats [35], or verification metadata [31] for long-term storage. This can be achieved securely using WebAssembly functions, which have recently been used for future-proof storage formats [34, 80]. Custom archival backends, such as a page materialization service, enable optimizations for cloud-native OLTP systems, such as Socrates [6].

ACKNOWLEDGMENTS

We thank Carsten Binnig for fruitful discussions during the early stages of the project; Philipp Unterbrunner, Pat Helland, Anub Ghatage, and Michael Haubenschild for their valuable feedback and industry insights; and the anonymous reviewers for their feedback. Funded/Co-funded by the European Union (ERC, CODAC, 101041375). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

REFERENCES

[1] Amazon Web Services. 2025. Amazon EC2 Placement Groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-groups.html

[2] Amazon Web Services. 2025. Availability Zones. https://docs.aws.amazon.com/whitepapers/latest/aws-fault-isolation-boundaries/availability-zones.html

[3] Amazon Web Services. 2025. Placement strategies for your placement groups. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/placement-strategies.htm

[4] Amazon Web Services. 2026. Data persistence for Amazon EC2 instance store volumes. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-store-lifetime.html

[5] Amazon Web Services. 2026. Data protection in Amazon S3. https://docs.aws.amazon.com/AmazonS3/latest/userguide/DataDurability.html

[6] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD*. 1743–1756.

[7] Apache Software Foundation. 2020. Apache Pulsar. https://pulsar.apache.org/. Accessed: 2026-01-31.

[8] Apache Software Foundation. 2026. Apache BookKeeper Github Repository. https://github.com/apache/bookkeeper

[9] Jens Axboe. 2026. Flexible I/O Tester. https://github.com/axboe/fio

[10] Jens Axboe. September 13, 2023. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[11] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *OSDI*. 617–632.

[12] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A distributed shared log. *ACM Trans. Comput. Syst.* 31, 4 (2013), 10.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *NSDI*. 1–14.

[14] Jeff Barr. 2008. Amazon EBS (elastic block store) – bring us your data | Amazon Web Services. https://aws.amazon.com/blogs/aws/amazon-elastic/

[15] Jeff Barr. 2009. Introducing Amazon RDS – The Amazon Relational Database Service. https://aws.amazon.com/blogs/aws/introducing-rds-the-amazon-relational-database-service/

[16] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR*. 9–20.

[17] Shreesha G. Bhat, Tony Hong, Xuhao Luo, Jiyu Hu, Aishwarya Ganesan, and Ramnatthan Alagappan. 2025. Low End-to-End Latency atop a Speculative Shared Log with Fix-Ante Ordering. In *OSDI*. USENIX Association, 465–481.

[18] Thomas Bodner, Theo Radig, David Justen, Daniel Ritter, and Tilmann Rabl. 2025. An Empirical Evaluation of Serverless Cloud Infrastructure for Large-Scale Data Processing. In *EDBT*. 935–948.

[19] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *SOSP*. ACM, 836–850.

[20] Marc Brooker. 2024. AWS re:Invent 2024 - Deep dive into Amazon Aurora DSQL and its architecture (DAT427-NEW). https://www.youtube.com/watch?v=huGmR_mi5dQ&t=627s

[21] Marc Brooker, Tao Chen, and Fan Ping. 2020. Millions of Tiny Databases. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). 463–478. https://www.usenix.org/conference/nsdi20/presentation/brooker

[22] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (1996), 225–267.

[23] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*. 251–264.

[24] CorfuDB Organization. 2026. Corfu Github Repository. https://github.com/CorfuDB/CorfuDB

[25] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *SIGMOD*. 1463–1478.

[26] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*. 1243–1254.

[27] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert van Renesse. 2020. Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log. In *NSDI*. 325–338.

[28] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *NSDI*. 401–414.

[29] Satish Duggana, Sriharsha Chintalapani, Ying Zheng, and Suresh Srinivas. 2025. KIP-405: Kafka Tiered Storage. https://cwiki.apache.org/confluence/display/KAFKA/KIP-405%3A+Kafka+Tiered+Storage

[30] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782.

[31] Muhammad El-Hindi, Tobias Ziegler, and Carsten Binnig. 2023. Towards Merkle Trees for High-Performance Data Systems. In *VDBS@SIGMOD*. ACM, 28–33.

[32] Muhammad El-Hindi, Tobias Ziegler, Matthias Heinrich, Adrian Lutsch, Zheguang Zhao, and Carsten Binnig. 2022. Benchmarking the Second Generation of Intel SGX Hardware. In *DaMoN*. ACM, 5:1–5:8.

[33] Facebook Engineering. 2017. LogDevice: a distributed data store for logs. https://engineering.fb.com/2017/08/31/core-infra/logdevice-a-distributed-data-store-for-logs/. Accessed: 2026-01-31.

[34] Mateusz Gienieczko, Maximilian Kuschewski, Thomas Neumann, Viktor Leis, and Jana Giceva. 2025. AnyBlox: A Framework for Self-Decoding Datasets. *Proc. VLDB Endow.* 18, 11 (2025), 4017–4031.

[35] Pascal Ginter and Viktor Leis. 2026. Active Data Lakes: Regaining Physical Data Independence Without Losing Interoperability. *Proc. VLDB Endow* (2026).

[36] Google. 2026. AlloyDB for PostgreSQL. https://cloud.google.com/products/alloydb

[37] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[38] Gabriel Haas, Bohyun Lee, Philippe Bonnet, and Viktor Leis. 2025. SSD-iq: Uncovering the Hidden Side of SSD Performance. *Proc. VLDB Endow.* 18, 11 (2025), 4295–4308.

[39] Fusheng Han, Hao Liu, Bin Chen, Debin Jia, Jianfeng Zhou, Xuwang Teng, Chuanhui Yang, Huafeng Xi, Wei Tian, Shuning Tao, Sen Wang, Quanqing Xu, and Zhenkun Yang. 2024. PALF: Replicated Write-ahead Logging for Distributed Databases. *Proc. VLDB Endow.* 17, 12 (2024), 3745–3758.

[40] Michael Haubenschild and Viktor Leis. 2025. Oltp in the cloud: architectures, tradeoffs, and cost. *VLDB J.* 34, 4 (2025), 42.

[41] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*. 877–892.

[42] Rebecca Isaacs, Peter Alvaro, Rupak Majumdar, Kiran Kumar, Muniswamy Reddy, Mahmoud Salamati, and Sadegh Soudjani. 2025. Analyzing Metastable Failures. In *HotOS*. ACM, 172–178.

[43] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, and Carsten Binnig. 2025. A Wake-Up Call for Kernel-Bypass on Modern Hardware. In *DaMoN*. ACM, 14:1–14:5.

[44] Matthias Jasny, Muhammad El-Hindi, Tobias Ziegler, Viktor Leis, and Carsten Binnig. 2025. High-Performance DBMSs with io_uring: When and How to use it. *CoRR* abs/2512.04859 (2025). https://doi.org/10.48550/ARXIV.2512.04859 arXiv:2512.04859

[45] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *SOSP*. 691–707.

[46] Flavio Paiva Junqueira, Ivan Kelly, and Benjamin C. Reed. 2013. Durability with BookKeeper. *ACM SIGOPS Oper. Syst. Rev.* 47, 1 (2013), 9–15.

[47] Donald Kossmann, Tim Kraska, and Simon Loesing. 2010. An evaluation of alternative architectures for transaction processing in the cloud. In *SIGMOD*. 579–590.

[48] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[49] Maximilian Kuschewski, Jana Giceva, Thomas Neumann, and Viktor Leis. 2024. High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance. *Proc. ACM Manag. Data* 2, 6 (2024), 238:1–238:27.

[50] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.

[51] Bohyun Lee, Tobias Ziegler, and Viktor Leis. 2026. How to Write to SSDs. *Proceedings of the VLDB Endowment* 19 (2026), 1469–1482.

[52] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196.

[53] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (2019), 2263–2272.

[54] Feifei Li. 2023. Modernization of Databases in the Cloud Era: Building Databases that Run Like Legos. *Proc. VLDB Endow.* 16, 12 (2023), 4140–4151. https://doi.org/10.14778/3611540.3611639

[55] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J. Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. 2018. The FuzzyLog: A Partially Ordered Shared Log. In *OSDI*. USENIX Association, 357–372.

[56] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. 2024. LazyLog: A New Shared Log Abstraction for Low-Latency Applications. In *SOSP*. 296–312.

[57] Adrian Lutsch, Muhammad El-Hindi, Zsolt István, and Carsten Binnig. 2025. Towards High-performance and Trusted Cloud DBMSs. *Datenbank-Spektrum* 25, 1 (2025), 39–50.

[58] Adrian Lutsch, Christian Franck, Muhammad El-Hindi, Zsolt István, and Carsten Binnig. 2025. An Analysis of AWS Nitro Enclaves for Database Workloads. In *DaMoN*. ACM, 5:1–5:8.

[59] Mellanox. Accessed: April 22, 2026. https://github.com/Mellanox/sockperf

[60] Microsoft. 2024. Global data distribution with Azure Cosmos DB - under the hood. https://learn.microsoft.com/en-us/azure/cosmos-db/global-dist-under-the-hood. Accessed: 2026-01-28.

[61] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162.

[62] Neon. [n.d.]. Neon architecture. https://neon.com/blog/architecture-decisions-in-neon. Accessed: 2025-12-31.

[63] Neon. [n.d.]. Neon architecture. https://neon.com/docs/introduction/architecture-overview. Accessed: 2025-12-31.

[64] Lam-Duy Nguyen, Adnan Alhomssi, Tobias Ziegler, and Viktor Leis. 2025. Moving on From Group Commit: Autonomous Commit Enables High Throughput and Low Latency on NVMe SSDs. *Proc. ACM Manag. Data* 3, 3 (2025), 191:1–191:24.

[65] NVMe Express, Inc. Accessed: March 22, 2026. https://nvmexpress.org/specifications

[66] Brian M. Oki and Barbara Liskov. 1988. Viewstamped Replication: A General Primary Copy. In *PODC*. 8–17.

[67] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX ATC*. 305–319.

[68] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3 (2015), 7:1–7:55.

[69] SAP. [n.d.]. Storage Configuration for SAP HANA. https://docs.aws.amazon.com/sap/latest/sap-hana/hana-ops-storage-config.html. Accessed: 2024-12-08.

[70] Scalog Organization. 2019. Scalog Github Repository. https://github.com/scalog/scalog

[71] Simplyblock GmbH. Accessed: March 22, 2026. Simplyblock: Cloud-native storage in your data center. https://simplyblock.io/

[72] Till Steinert, Maximilian Kuschewski, and Viktor Leis. 2026. Cloudspecs: Cloud Hardware Evolution Through the Looking Glass. In *CIDR*.

[73] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12 (2019), 2170–2182.

[74] The Linux Foundation. Accessed: March 22, 2026. DAOS: The Open-Source Storage Platform for AI & HPC. https://daos.io/

[75] Raúl Gracia Tinedo, Flavio Junqueira, Tom Kaitchuck, and Sachin Joshi. 2023. Pravega: A Tiered Storage System for Data Streams. In *Proceedings of the 24th International Middleware Conference, Middleware 2023, Bologna, Italy, December 11-15, 2023*. 165–177.

[76] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD*. 1041–1052.

[77] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proc. VLDB Endow.* 11, 4 (2017), 406–419.

[78] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems. *Proc. VLDB Endow.* 14, 2 (2020), 189–201.

[79] Xiangyao Yu. 2025. Disaggregation: A New Architecture for Cloud Databases. *Proc. VLDB Endow.* 18, 12 (2025), 5527–5530.

[80] Xinyu Zeng, Ruijun Meng, Martin Prammer, Wes McKinney, Jignesh M. Patel, Andrew Pavlo, and Huanchen Zhang. 2025. F3: The Open-Source Data File Format for the Future. *Proc. ACM Manag. Data* 3, 4 (2025), 245:1–245:27.

[81] Jiashu Zhang, Wen Jiang, Bo Tang, Haoxiang Ma, Lixun Cao, Zhongbin Jiang, Yuanyuan Nie, Fan Wang, Lei Zhang, and Yuming Liang. 2023. CDSBen: Benchmarking the Performance of Storage Services in Cloud-native Database System at ByteDance. *Proc. VLDB Endow.* 16, 12 (2023), 3584–3596.

[82] Jiatang Zhou, Kaisong Huang, and Tianzheng Wang. 2026. Milliscale: Fast Commit on Low-Latency Object Storage. arXiv:2603.02108 [cs.DB] https://arxiv.org/abs/2603.02108

[83] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *SIGMOD*. 2653–2666.

[84] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem?. In *CIDR*.