

LiteLib: Containing Cascading Failures in Stateful Services with Lite Replicas

Yi Chen Site Fan Rishika Varma Kalidindi Po-Yu Hou Maizhe Zhang Peng Huang

University of Michigan

Abstract

Stateful applications are the backbone of cloud services, yet their failures frequently lead to prolonged disruption and trigger cascading and metastable failures. Replication incurs high resource overhead and slow failover, while techniques like checkpointing or load shedding either suffer from stale data or low service capacity that fails to contain failure impact.

This paper presents LiteLib, a library that enables *lite replicas*, compact versions of stateful applications that maintain only critical state and core operations. A lite replica provides bounded yet high service capacity during failures until the application fully recovers. LiteLib transparently maintains the lite replica's state during normal operation and switches to it during failures while logging updates to replay after recovery.

We integrate LiteLib with five datacenter applications (LevelDB, Memcached, Redis, HDFS, and MySQL), and evaluate it in standalone and microservice deployments. The lite replicas accelerate recovery by 46.7×, serve over 50% of user-level requests during failures, and mitigate cascading and metastable failures, while incurring low overhead and requiring only 2% of the original code size to implement.

1 Introduction

Stateful applications like caches, databases, and stream processing engines form the backbone of cloud services. Even brief downtime can disrupt many requests and clients. Yet these applications are prone to software failures including crashes and hangs [12, 30–32, 34, 43–45, 57, 62, 64, 66], which make the service unavailable while the underlying hardware remains healthy. Studies show that software errors are the major causes of cloud service outages [28, 32, 33, 44]. Recovering stateful applications typically requires state reconstruction and warm-up, which inherently takes time and leaves a service gap even after the process restarts.

Moreover, this service gap can cause disruption beyond the failed component. Stateful applications often serve as downstream dependencies in microservice systems [47]. During recovery, retries and traffic redirection can overload other healthy components, triggering *cascading failures* [10, 37, 49, 51]. These effects can also create sustained degradation after the initial trigger has ended, resulting in *metastable failures* [14, 36].

Despite decades of research on fault tolerance, existing mechanisms leave fundamental gaps. *Replication-based mechanisms* improve availability but incur high coordination delays [11, 42, 55]. They also suffer from slow failover, as clients must reconnect to new replicas, often causing abrupt load shifts that trigger cascading failures. Moreover, since replicas use the same code, a deterministic bug can cause the other replicas to fail as well. Active-passive replication further requires duplicating resources. Active-active replication partially alleviates resource underutilization by allowing replicas to serve requests during normal operation. However, the system typically operates near capacity in steady state, leaving little to absorb failures. When a replica fails, traffic is redistributed to remaining replicas that are already heavily loaded, leading to performance degradation. Crucially, during recovery, the hardware resources of the host running the failed process remain available but are not exploited.

Checkpointing [58, 61, 67] lowers resource cost, but it leaves a long service gap while restoring a checkpoint. It resumes operation from a stale snapshot, compromising response quality. *Conventional caching* improves performance in normal operation but typically only supports read-only requests. Thus, it neither preserves enough service capacity to prevent cascading failures nor handles state synchronization after recovery. *Load shedding* [10] prevents overload in other components but does not provide capacity for the failed component itself, so requests targeting that service continue to fail until it recovers. As a result, in practice, software failures of stateful components continue to cause prolonged service disruption even when these mechanisms are in place.

These limitations, together with the prevalence of software failures in cloud services, raise a key research question: *can we provide high service continuity during software failures while remaining resource-efficient in normal operation?*

In answering this question, we make two observations:

- **Skewed operation usage**: While stateful applications support a broad range of operations, in practice, requests commonly exercise only a small subset. For instance, Meta restricts its services to a limited set of simple SQL queries [15]. At Twitter, over 90% of requests involve `Get` and `Set` operations, whereas others, such as `Prepend`, `Incr`, and `Cas`, are rarely used [70]. Similarly, in Google's authorization

system [56], `Check`, `Read`, and `Expand` operations occur orders of magnitude more frequently than `Write`.

- **Skewed and correlated state access**: Similarly, the state accessed by requests tends to be both heavy-tailed and correlated [56, 59, 70]. For example, Scribe [40], a production message queue application, supports replicating popular data on ephemeral storage. Cao *et al.* [17] observed that, in Meta's database, keys with the most Get queries also tend to have the most Put queries.

These characteristics highlight an opportunity: keeping only a subset of state and functionality during failures can still serve many clients and requests, minimize failover time, absorb retries, limit failure propagation, and avoid staleness.

Building on these insights, we propose **lite replicas**, a new approach for cost-effective mitigation and containment of software failures. A lite replica is a compact version of an application. It runs in the same endpoint and activates immediately upon failure with *fresh* state. Unlike conventional replication, it provides just a subset of services while the application recovers, analogous to a spare tire. By focusing on frequently used operations and state, it achieves enough capacity to reduce service disruption and mitigate metastable failures while avoiding high overheads.

To make this approach practical, we design LiteLib, a library that addresses the challenges in building and managing lite replicas. Developers use the simple abstractions and APIs of LiteLib to specify the core application state and operations, while the library handles the heavy lifting automatically.

LiteLib operates in two modes. In normal mode, it maintains up-to-date state for the lite replica with low overhead. It *passively* updates this state through transparent interception with lightweight hooks and eBPF. To limit resource usage, LiteLib keeps a State Store with a configurable memory quota. Upon a failure, LiteLib switches to emergency mode. It ensures seamless failover through automatic socket transfer and then continues serving requests from its state.

A key capability of LiteLib is that it goes beyond merely serving cached responses. Because it maintains the underlying application state during normal mode, the lite replica can serve both read-only and state-changing requests in emergency mode as well as execute core business logic and even interact with other components. Supporting writes is especially important. Failures often cause backlogged writes that, if dropped or rerouted, can cascade into larger outages.

To support this capability, LiteLib provides logging for state-changing requests served during emergency mode. Once the application recovers, LiteLib synchronizes the updates through replay, after which it transitions back to normal mode. This ensures that the failure is masked and clients have a consistent view of the application state.

LiteLib supports different deployment scenarios. It works for single-component services as well as stateful components in microservices. Integrating LiteLib into one component is transparent to clients and dependent services. Lite replicas complement rather than replace conventional replication. Replication remains essential for handling hardware failures across machines. LiteLib is specialized for software failures within a node, enabling instantaneous failover and much lower client-visible disruption by leveraging local resources during recovery. Combining both approaches allows a system to achieve fast local recovery from software faults while being robust to broader failures. We demonstrate this capability by applying LiteLib to HDFS, a distributed system.

We apply LiteLib to develop lite replicas for five popular, large datacenter stateful applications: LevelDB, Memcached, Redis, HDFS DataNode, and MySQL. Despite the complexity and scale of these applications, the lite replicas remain compact (on average only 2% of the original code size). Moreover, since lite replicas focus on core functionality, they do not require frequent changes across application updates, amortizing the development effort. They also share little code with the full version, making them less prone to shared bugs.

Despite their compactness, the lite replicas achieve substantial gains in mitigating failure impact. During failures, LiteLib serves 54% of end-to-end user-level requests while maintaining a goodput comparable to the original service. The recovery time to reach 90% throughput is accelerated by 46.7×. LiteLib incurs only modest overhead: on average, 6.4% latency, 27.5% CPU, and 21.3% memory.

In comparison, active-passive replication requires double the memory and CPU resources yet still incurs failover delays, *e.g.*, around 30 s for Redis. Active-active replication still incurs failover delays and fails to mitigate cascading failures. Checkpointing causes substantial stale data and replay delays, *e.g.*, 56% stale responses for LevelDB failures. In a microservice setting with cascading failures, LiteLib reduces load shift by 13.7× and prevents latency from continuously increasing, mitigating the failure.

This paper makes the following contributions:

- We propose lite replicas, a new approach to achieve cost-effective service failure containment for stateful software.
- We design a library runtime, LiteLib, to facilitate the construction and management of lite replicas.
- We evaluate LiteLib on real-world applications and microservices to demonstrate its benefits and efficiency.

The source code of LiteLib is available at https://github.com/OrderLab/LiteLib.

2 Motivation

We use cascading failures to show how a single failure of a stateful application can cause prolonged service disruption despite existing fault-tolerance mechanisms. We also show how LiteLib mitigates such failures.

A Single Failure Escalates. Stateful applications, such as caches and databases, are critical dependencies in microservices. To tolerate failures, these applications are commonly

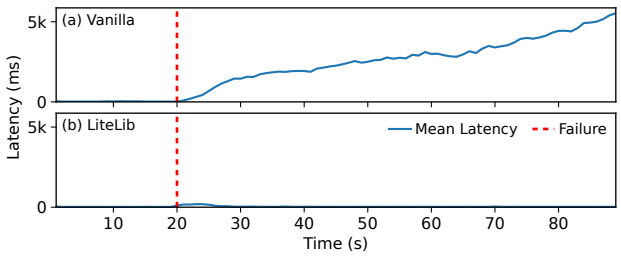


Figure 1. User-level latency before and after the failure.

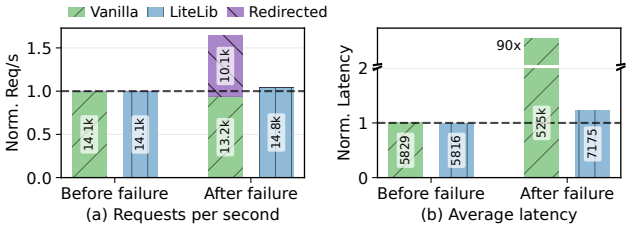


Figure 2. Performance for the healthy instance. The value on each bar in (b) represents the average Memcached latency in microseconds.

replicated across multiple instances. However, even with replication, when one instance fails, client retries and traffic redirection often cause load amplification, where redirected requests overwhelm remaining instances and lead to sustained service degradation. To illustrate this challenge, we use the Social Network setup from the DeathStar Benchmark [27] (its dependency graph is shown in Appendix A). We deploy Mcrouter [53] in front of two Memcached instances to broadcast write requests and load balance read requests, forming an *active-active* serving configuration.

Figure 1(a) shows the client latency when one Memcached instance crashes (the experimental setup is detailed in §7). Despite the fault-tolerant configuration, client latency continues to increase and shows no sign of stabilizing by the end of the experiment. The goodput—successful responses that meet the SLO—is effectively *zero*.

As Figure 2 shows, after the failure, the healthy instance experiences a 65% increase in workload due to the failover. This overload causes the instance's latency to increase by 90×, which propagates to the phenomenon observed in Figure 1(a). The throughput also drops by 7.7% within the approximately one-minute duration of the experiment, resulting in over 13.3K user requests not being served due to the failure.

In summary, while replication helps maintain availability, it does so by shifting *all* failed and new requests to remaining instances and increasing their load. Yet, the resources of the host running the failed instance remain available but unused.

Lite Replica Mitigates the Failure and Escalation. As depicted in Figure 1(b), with LiteLib deployed, the service quickly stabilizes after the failure. This is because the lite replica continues to serve both read and write requests with low latency during the failure, limiting the load increase on the healthy instance to 4.7% and containing the failure impact. Thanks to the characteristics of stateful applications we mentioned in §1, the lite replica achieves this with only 15.6% additional memory and 1% of the original code.

We further measure the success rate of user-level requests in the worst case, where even the remaining replica and the database are unavailable. Although a single user-level request consists of tens of correlated requests to the lite replica, the end-to-end success rate averages 43%, increases to 54% after one minute, and continues to improve over time. Notably, this rate can increase further with a larger memory quota.

We also evaluate other existing failure mitigation solutions. One commonly used technique is load shedding. Our experiments show that at least 40% of requests must be dropped to prevent latency from increasing continuously. However, this significantly compromises the system's goodput, which remains much lower than that achieved with LiteLib. Another technique is graceful degradation by dynamically disabling heavyweight features [50]. In our evaluation, neither of the two mitigation strategies is sufficient. For load shedding, dropping writes to a healthy instance stops cache updates, causing read requests to bypass the cache and overwhelm the backend database. For graceful degradation, disabling the client-side `ComposePost` feature reduces write load, but the remaining read requests alone can still exceed system capacity.

Our key insight is that even *partial* service capacity can substantially mitigate the impact of a software failure. Unlike existing approaches, LiteLib leverages the otherwise unused *local* resources of the failed instance and converts them into usable service capacity. Providing *full* capacity through complete replication of functionality and state is often unnecessary and prohibitively costly; it pays off only for long failures, ideally permanent ones. Production software usually takes a bounded time to recover [28, 33]. Thus, we need *sufficient* capacity only until recovery finishes.

3 LiteLib Design

We design LiteLib, a library for building and managing *lite replicas* to minimize the impact of failures and achieve high service continuity at a fraction of the cost of full replication.

3.1 Scope and Opportunities

LiteLib targets software failures in stateful, latency-sensitive applications where the process fails—whether due to a crash or an operator- or watchdog-initiated [46] application restart (*e.g.*, to resolve a hang or memory leak)—while the underlying node remains healthy. Such failures are common and can cause significant disruption even when the failure is brief. LiteLib does *not* address hardware or OS failures that make the entire node unavailable, which can be handled by conventional replication. LiteLib enables fast local failure containment for the common case and reduces the service gap to milliseconds (§7) without requiring expensive cross-machine replication.

LiteLib is most effective for applications with three properties: (1) A small set of operations dominates request traffic. By supporting only these operations, a lite replica can preserve

a large fraction of service capacity with minimal implementation effort. (2) A relatively small portion of the state is accessed frequently, and frequently accessed items tend to be updated frequently. Thus, a lite replica can maintain bounded state, consuming only moderate resources during normal operation. (3) Application state can be decomposed into multiple independent units, such as key-value pairs for caching services, rows for databases, and messages for messaging systems.

Many production stateful applications exhibit such properties. While some may not, such as telemetry backends, compute-intensive analytics pipelines, and maintenance tasks, their failures are less likely to significantly disrupt users. LiteLib does not target these or stateless applications.

3.2 Design Choices

LiteLib co-locates a lite replica with the application, which we term the *full version*. This co-location allows us to leverage the unused local resources of the failed instance. However, co-location does not by itself ensure low runtime overhead for state synchronization. A key design challenge is how to efficiently synchronize the state information between them.

A straightforward approach is to fetch state from the full version at the time of failure, minimizing resource overhead during normal operation. However, this reactive approach relies on the full version being operational even in case of failure, which limits the kinds of failures LiteLib can handle and introduces a significant service gap. Instead, LiteLib takes a proactive approach by extracting and maintaining a bounded set of application state during normal operation. This allows the lite replica to immediately assume service responsibilities when the full version fails, while incurring low memory overhead.

There are two ways to obtain state from the full version before failure: (1) *traffic observation*, which extracts state from the client traffic to and from the full version, and (2) *in-process hooks*, which push state updates from the full version to a separate lite replica process during request handling. Traffic observation is more transparent, while in-process hooks incur lower resource overhead. LiteLib supports both and exposes them as operator-selectable options (§5.1).

A third approach is to co-design with state machine replication to obtain state information. When applications already expose shared logs [25, 48], this can be effective and even beneficial. However, such mechanisms are often absent in microservice architectures. Introducing a local state machine replication layer between a single writer (the full version) and a single reader (the lite replica) is intrusive and adds unnecessary abstraction and overhead.

3.3 LiteLib Overview

LiteLib consists of four main modules (Figure 3): *Network*, *Serving*, *State Store & Logging*, and *Replay*. Together, they operate as a lite replica. LiteLib also provides a *LitePlugin*, which is invoked by the full version for auxiliary operations.

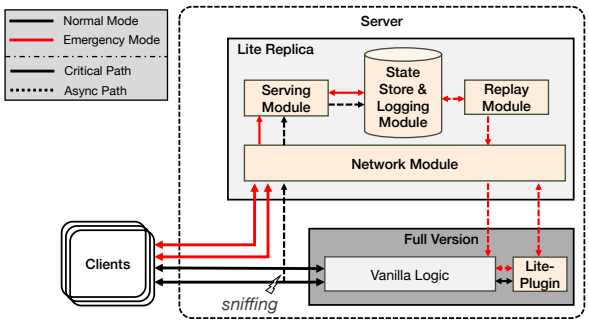


Figure 3. LiteLib overview.

LiteLib operates in two modes: *normal mode* when the full version is healthy, and *emergency mode* during failure.

To efficiently extract state, LiteLib deploys an eBPF program between the clients and the full version (§3.4). In normal mode, clients connect directly to the full version, which handles all requests. By monitoring these requests and responses, the userspace lite replica collects and maintains relevant data in anticipation of potential emergencies.

When the full version fails, client-server connections remain intact through our socket migration and mode transition mechanism (§3.5), enabled by the LitePlugin. During emergency mode, the Serving Module attempts to handle incoming requests using the available data in the State Store (§3.6), while efficiently logging all state-changing requests.

Once the full version is restored, the Replay Module synchronizes all the state changes with the recovered full version before transitioning back to normal mode (§3.7). Throughout this process, the Serving Module keeps serving requests.

To ease development and support diverse applications, LiteLib provides simple abstractions and a small set of callbacks that developers need to implement (§4). These callbacks define how to extract state information from requests and responses, how to handle requests for the subset of supported operations in emergency mode, *etc.*

3.4 State Extraction in Normal Mode

A lite replica must maintain sufficient state in normal mode so it can immediately serve requests when the application fails. This raises two key challenges: avoiding extensive application changes and keeping runtime overhead low.

Our insight is that application APIs and protocols evolve far more slowly than internal implementations. Thus, LiteLib treats the full version as a black box and extracts the necessary state from network traffic. This approach greatly reduces development effort while remaining robust to application code updates. The decoupling also allows the lite replica to use a customized state representation tailored for its own use case (serving requests in emergency mode).

By default, LiteLib leverages eBPF to efficiently capture information between clients and the full version. Compared to a proxy-based setup, this approach incurs minimal overhead. As Figure 4 shows, in the critical path (❶ and ❷), network

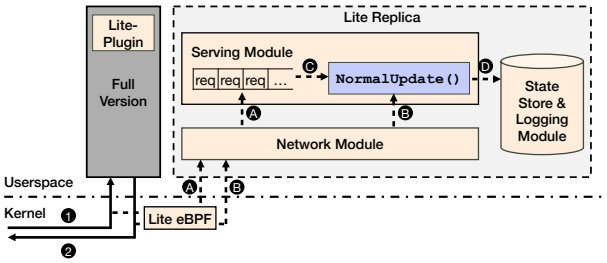


Figure 4. Normal mode workflow. The solid lines represent the critical path, while the dashed lines represent the non-critical path.

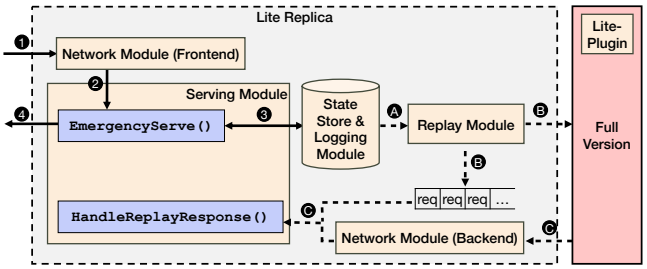


Figure 5. Serving in emergency mode and replaying requests after the full version recovers. The solid lines represent the critical path, while the dashed lines represent the non-critical path.

flows normally to and from the full version. A lightweight eBPF program attached by LiteLib identifies *state-changing* requests (*e.g.*, `SET`, `APPEND`, `DELETE`)—we also call them *write requests*—and their responses, and asynchronously forwards a copy to the lite replica. It filters out read-only requests that do not affect lite state. Request boundaries and types can typically be identified using fields such as length and type in the request header. When the application uses complex serialization formats or the traffic is encrypted, the eBPF program's ability to perform in-kernel filtering will be limited. In such cases, deserialization and filtering logic is delegated to the userspace lite replica. Since the lite replica operates within the same trust domain as the full version and has access to the necessary credentials (*e.g.*, TLS session keys), it can decrypt and interpret traffic that eBPF cannot.

In the lite replica, off the critical path, the Serving Module pushes the request into the *pending* request queue (❹). Once the response arrives (❺), the request is dequeued (❻). Through the application-specific `NormalUpdate()` callback (Listing 1), LiteLib extracts the semantics of requests and responses, and updates the State Store (❼). Note that LiteLib extracts state changes only after receiving responses, ensuring that it remains consistent with the full version's committed state and correctly handles non-deterministic operations.

For developers willing to make trade-offs between effort and overhead, LiteLib provides alternative deployment options (embedded and proxy), which are described in §5.1. The embedded option can also cover state changes that responses do not reveal, such as hidden non-deterministic outcomes.

3.5 Seamless Transition upon Failure

When the full version fails, LiteLib transitions into emergency mode. This transition must be fast and nearly imperceptible to clients to minimize the service gap and avoid retry storms [36, 62]. Ideally, client connections remain open and uninterrupted, preserving the illusion of a stable and unchanged backend. Meanwhile, the program handling those requests behind the connection switches to LiteLib.

To achieve this, LiteLib migrates all client connection sockets to the lite replica during the transition (§5.2). Upon failure, the signal handler in the full version invokes the LitePlugin, which transfers all client sockets to the lite replica via a Unix domain socket and prevents the full version's failure handler from writing error messages to them. This mechanism assumes failures that still allow signal delivery and handler execution (*e.g.*, a crash that raises `SIGSEGV`). For failures that prevent handler execution, LiteLib relies on a simple fallback: clients detect the broken connection and reconnect to the same port, which is now bound to the lite replica.

3.6 Serving Requests in Emergency Mode

In emergency mode, the lite replica follows a *serving invariant*: it serves a request only when all state needed for the response is present in the State Store, and rejects the rest. In normal mode, if the lite replica cannot derive an object's new value from an observed request (*e.g.*, an unsupported operation), it removes the object, or keeps it but marks its affected fields unknown (§6.2), so it never holds a stale value.

As Figure 5 shows, LiteLib processes requests using its State Store. Upon receiving a request (❶), the Network Module forwards it to the Serving Module (❷). The `EmergencyServe()` callback (§4) retrieves the necessary state from the State Store and serves the request using it (❸, ❹). Meanwhile, it logs the request for replay (§3.7).

If required state is missing, LiteLib returns an application-level error or closes the connection (§6.2), so clients observe a failed request rather than an incorrect response.

3.7 State Synchronization after Recovery

Supporting write requests is a key feature of LiteLib. To prevent clients from observing stale data or inconsistencies, the recovered full version's state must be brought up to date before normal operation resumes. To achieve this, LiteLib logs all write requests during emergency mode, and replays them once the full version recovers.

The key challenge is efficient synchronization: the more writes the lite replica handles, the more state must be synchronized, yet the full version cannot safely resume until synchronization completes. Thus, the Logging and Replay modules aim to minimize log volume and replay latency.

Logging. Naively logging all requests and replaying them in order is inefficient: it incurs high memory overhead and makes timely replay difficult, especially when client throughput approaches system capacity and new requests continue to arrive during the replay process.

LiteLib addresses this challenge with *log compaction*. Our insight is that only the version resulting from the final modification to an object matters when reconstructing the post-failure state. This collapses multiple updates into one, reducing storage and replay overhead.

To support replay, LiteLib uses two types of log entries: *Action Nodes* and *Result Nodes*. An Action Node records the information required to re-execute a request, typically the request itself. A Result Node instead references an object in the State Store, avoiding repeated re-execution when multiple requests modify the same object. Developers choose a log entry type for each write request type based on its semantics: a Result Node for single-object or non-deterministic updates (*e.g.*, popping a random member from a set), so that repeated writes compact into one replay request and nothing is re-executed, and an Action Node for request types that can update many objects or whose effects cannot be fully represented by objects retained by the lite replica (*e.g.*, range updates that may affect objects outside the State Store).

For request types that use Result Nodes, LiteLib marks updated objects as `REPLAY_REQUIRED` and creates a Result Node for them in emergency mode. Objects referenced by Result Nodes are exempt from eviction until replayed. When an Action Node reads or updates such an object, LiteLib copies its current value into the Result Node and *pins* it (Appendix B). Requests logged as Action Nodes still update tracked objects, so they remain servable in emergency mode.

Each new Result Node replaces the previous node for the same object, unless that node is pinned. This collapses multiple writes into one replay request, so *compaction occurs during logging*. A pinned node is exempt from the compaction to preserve the intermediate version needed for replay. Otherwise, replacing it with a later value could change the action's effects. At replay time, LiteLib converts each Result Node into a write request for the full version, using its pinned copy if present and the linked State Store object otherwise.

To maintain the order of requests, LiteLib uses a global linked list that contains all log entries, ordered chronologically based on the corresponding modification requests. Additionally, a separate linked list is maintained for each client connection to record which client sent each request.

Consider two Redis clients. LiteLib receives the following requests in order in emergency mode:

```
1 DELETE K2 (Client 2)
2 MULTI     (Client 1)
3 SET K1 V1 (Client 2)
4 SET K1 V2 (Client 1)
5 SET K3 V3 (Client 1)
6 EXEC      (Client 1)
```

Figure 6 illustrates the log entries recorded before the `EXEC` request (commit a transaction). Two linked lists formed by the `Connection` nodes represent the clients that have sent requests. The `Chronological` node records the sequence of requests received by the server. For the third request, which is a modification, LiteLib automatically records it in a Result Node. While the first request is also a modification, it is recorded in an Action Node because a delete leaves no object for a Result Node to reference. All requests from Client 1 are also recorded in Action Nodes, because they are part of an uncommitted transaction.

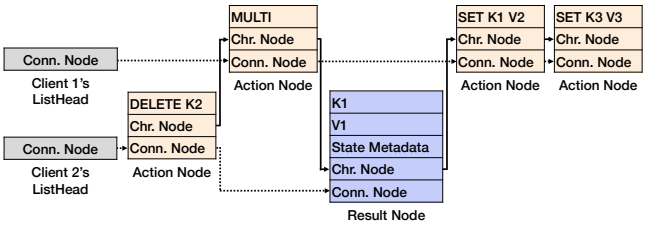


Figure 6. Logging example (before EXEC).

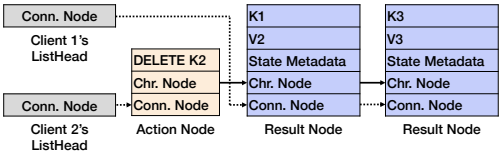


Figure 7. Logging example (after EXEC).

After the `EXEC` request, all requests from Client 1 are committed. Consequently, LiteLib removes all Action Nodes associated with Client 1 from the linked lists. Additionally, the two `SET` operations from Client 1 are automatically recorded in Result Nodes and modifications to the same object are compacted into a single Result Node. The Action Nodes are then updated as Figure 7 shows.

Replaying. When the full version recovers, LiteLib must replay all logged writes. During the replay, the lite replica must continue serving client requests to maintain service continuity. Thus, LiteLib employs a *double replay* procedure.

As Figure 5 shows, the replay occurs *in parallel* with request handling. The Replay Module goes over all the log entries in the State Store & Logging Module in the order of the `Chronological` nodes (Ⓐ). It creates one connection to the full version for each active or closed client connection (Ⓑ). All entries initiated by the same client connection are replayed over the same connection to the full version. This ensures that the replayed requests preserve per-client ordering and consistency. The Replay Module pushes each replay request to a pending request queue. When the Network Module receives responses from the full version, `HandleReplayResponse()` processes them to detect and handle errors during replay (Ⓒ).

In some cases, incoming throughput may exceed replay throughput. To address this, LiteLib implements admission control using sliding windows [24] to estimate incoming and replay throughput. When replay throughput falls below a user-specified multiple of incoming throughput, admission control is triggered and new modification requests are rejected to prevent unbounded replay. This multiple thus controls replay time, at the cost of dropping some modification requests.

Recovering. Once all log entries have been replayed, LiteLib stops admitting new requests and waits for in-flight requests (typically few) to complete. After that, LiteLib replays the

entries added by these requests for full synchronization. Then, LiteLib switches back to normal mode by migrating all client connections and listen sockets to the recovered full version.

3.8 Deployment Across System Architectures

Besides microservices, LiteLib is also applicable to traditional monolithic software, including replicated systems. Since the lite replica observes all replication protocol-related network traffic and handles it during failures, it remains active and synchronized with other replicas. This design ensures compatibility with distributed systems.

In case of hardware or kernel failures, conventional remote replicas can still take over to ensure availability. This layered deployment allows the original replication protocol to act as a fallback during failures beyond the scope of LiteLib.

3.9 Correctness Overview

LiteLib preserves correctness across failures under two configurations. Appendix B gives the detailed proofs.

Strict mode (response equivalence). In strict mode, the full version exposes a sequence ID (`seq`) that reflects its serialization order. Most services already expose a notion of a global serialization position for replication (*e.g.*, the GTID in MySQL and the replication offset in Redis), which totally orders committed requests. If the target service does not already export such a value, the full version can increment a counter where it linearizes a request (*e.g.*, when Redis appends the request to its log) and carry the value in a reserved field of the response or in a header attached to each response it sends to LiteLib. LiteLib strips `seq` before returning the response, so the network protocol and client-visible semantics remain unchanged. Single-writer services (*e.g.*, an HDFS DataNode with at most one active writer per block) already serialize requests deterministically and need no explicit `seq`.

During normal operation, LiteLib derives state updates only from processed responses. Requests whose effects are not externally observable (*e.g.*, those whose responses were never sent) are not reflected in the lite replica's state. At failure, LiteLib sets up a cutover barrier by stopping forwarding and draining all observable responses. This lets the lite replica start from a well-defined prefix state consistent with the full version's committed and externally visible state (s_F^{fail}).

There are cases where external observations diverge from the internal state of the full version. For example, the full version may recover to a state different from s_F^{fail}, or a request may be committed but return an error due to transient failures [21]. Such inconsistencies stem from the original system's failure and retry semantics, and are not introduced by LiteLib.

During emergency mode, the lite replica serves requests only when all required state is present and derived from committed responses, and applies the same semantics as the full version. After recovery, LiteLib replays all accepted updates in order. Dependency-aware log compaction preserves only necessary state while ensuring correctness. Assuming the full version recovers to s_F^{fail}, replaying all emergency-mode updates brings it to an equivalent state.

As a result, (1) responses served during emergency mode match those of the full version under the same prefix state, and (2) the system converges after recovery.

Relaxed mode. When `seq` is unavailable, LiteLib provides weaker guarantees: after recovery and replay, the full version eventually reflects all accepted emergency-mode writes, as long as each write is either *base-independent* (*e.g.*, an unconditional overwrite, a delete, or an increment) or recorded as a Result Node. Other writes, such as conditional updates, require strict mode (Appendix B.2).

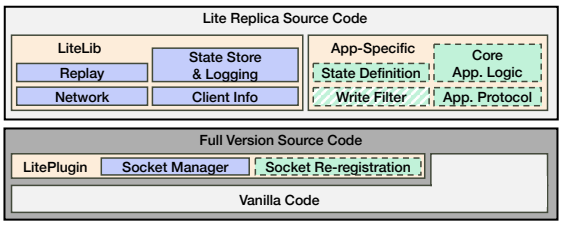


Figure 8. Code architecture of LiteLib. The vanilla code (gray) comprises the majority of the codebase. The LiteLib modules (purple) are shared across all applications. The application-specific callbacks (green with dotted borders) constitute the smallest portion, among which the write filter callback, used by the eBPF program to identify state-changing requests, is optional.

```
1  class Req {int Deserialize(uint8_t*);};
2  class Resp {int Deserialize(uint8_t*);};
3  void NormalUpdate(resp,req,state_store) {
4    if (resp == kOK && req.op == kSet)
5      state_store.set(req.key, req.value);
6  }
7  Resp EmergencyServe(req,state_store,log,flow_control) {
8    if (req.op == kGet) return state_store.get(req.key);
9    else if (req.op == kSet && !flow_control)
10     return state_store.set(req.key, req.value);
11   else return kError;
12 }
13 void ReinstallEventHandler(fd,client_or_listener) {
14   ev_set(client_or_listener.event, fd, Handler);
15 }
```

Listing 1. Key application-specific code for Memcached. Protocol definitions (lines 1–2) and core logic (lines 3–12) of the lite replica, and socket re-registration in the LitePlugin (lines 13–15).

4 Interfaces

Developers build a lite replica with LiteLib by implementing a few callbacks that let it extract state in normal mode and serve requests in emergency mode (Figure 8). Listing 1 shows the key application-specific code for Memcached.

The `EmergencyServe()` callback is a key component of the LiteLib library, as it handles client requests during emergency mode. In this function, developers must define how to use state to serve requests. It lets developers decide the level of service in emergency mode by choosing which operations to serve and which to reject. More examples are in §6.2.

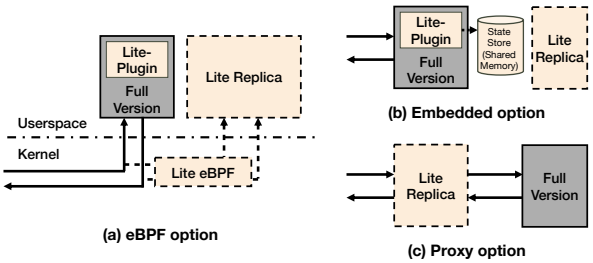


Figure 9. Three options of LiteLib. The solid lines represent the critical path, while the dashed lines represent the non-critical path.

LiteLib requires minor changes to the full version. Developers must implement `ReinstallEventHandler()` to restore network handlers after replay and subsequent socket replacement by the LitePlugin (§5.2), which typically requires only tens of lines of code.

5 Implementation

LiteLib is implemented in C/C++ in approximately 5.7K SLOC, using Boost C++ Library 1.87.0 and Libevent 2.1.12.

5.1 Deployment Options

LiteLib offers three design and deployment options as shown in Figure 9. They share the core design but differ in how they extract state in normal mode. Operators can choose and switch options based on their specific requirements.

The *eBPF option* extracts state without modifying the full version as already described in §3.4. Because this option is off the critical path, the full version is not affected even in the worst case where the lite replica crashes during normal mode.

The *embedded option* incurs lower CPU overhead. The full version invokes hooks provided by the LitePlugin whenever a write request is processed. The LitePlugin runs as threads within the full version and updates the State Store in shared memory, which is used by the lite replica during emergency mode. The mode transition process is identical to that of the *eBPF option*. By avoiding additional network I/O, it reduces CPU overhead. The trade-off is that explicit modifications to the full version are needed to invoke the LitePlugin hook `UpdateState()` during write request handling. Direct updates from the modified full version enable strict mode (Appendix B.1) and simplify support for encrypted traffic.

The *proxy option* requires no changes to the full version and supports near-instant mode switching through a simple mode flag. In normal mode, the lite replica runs as a local proxy; in emergency mode, it serves requests directly. It extracts state from proxied traffic, eliminating the need for integration with the full version. The trade-off is higher CPU and latency overhead, as all requests incur an extra network hop and pass through the lite replica on the critical path.

5.2 Socket Replacement Mechanism

LiteLib migrates all client connections and listen sockets during mode transitions. It introduces a new interface, `socket_replace()` (Figure 10), which replaces the socket at

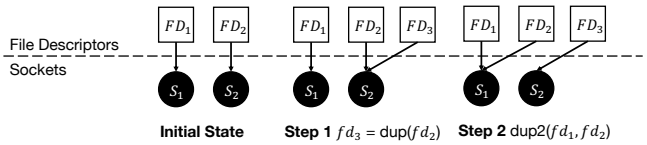


Figure 10. Workflow of $fd_3 = socket_replace\ (fd_1, fd_2)$.

fd_2 with fd_1 and returns a new fd_3 that refers to the original socket previously at fd_2.

On switching to emergency mode, the LitePlugin replaces client and listen sockets (S_2) with placeholders (S_1), detaching the full version from clients, and transfers the original sockets (fd_3) to the lite replica. On switching back, it replaces the replay connections (S_2) with the actual client connections (S_1) and closes the replay sockets (fd_3).

6 Building a Lite Replica with LiteLib

6.1 General Guidelines

To apply LiteLib to a new application, developers (1) select operations to support and (2) implement them and their state.

Choosing operations. Developers should analyze the workload and prioritize frequently used operations. At this stage, the operation's cost is not the primary concern since the implementation can often be simplified and LiteLib already keeps state extraction off the critical path. Instead, the goal is to ensure the selected operations preserve enough capacity to contain failure impact. We observed several common patterns across our use cases: (1) reads and simple writes dominate production traffic; (2) complex queries and administrative commands can be deferred; (3) infrequent operations that do not affect the lite replica's maintained state can be rejected with an error. Otherwise, they must either be implemented or trigger invalidation of the affected state; (4) authentication, although infrequent, is required to admit clients. The lite replica only needs to authenticate existing users and can omit credential management operations (*e.g.*, user creation), which can be deferred to the full version after recovery.

Implementing operations and state. The lite replica need not replicate the full implementation. Instead, it should adhere to the functional specification while using simpler designs. Since it maintains less state and is not required to match peak performance, complex optimizations and data structures can often be omitted. For example, MySQL's query optimizer need not be reimplemented. Similarly, simpler concurrency control (*e.g.*, coarse-grained locking) is often adequate.

The maintained state is determined by supported operations: state required only by unsupported operations can be omitted, allowing large portions of internal state to be eliminated (*e.g.*, optimizer statistics or secondary indexes). The lite replica's state representation need not match the full version. Developers can adopt lightweight designs, such as using the State Store Map interface for key-value-like workloads or simplified representations for more complex systems.

The serving invariant (§3.6) requires that a request be served only when all state required to compute a correct response is present in the State Store. Otherwise, it should be rejected.

6.2 Use Cases

Memcached and Redis. Their lite replicas support the core `GET`, `SET`, and `DELETE` operations. The Redis lite replica additionally handles basic transactions (`MULTI`/`EXEC`) by buffering commands within the lite replica and applying them atomically upon commit. In normal mode, they insert hot key-value pairs into the State Store upon observing successful `SET` responses, while read-only operations are skipped. In emergency mode, both adhere to the serving invariant. For `GET` requests on missing keys, the lite replica returns "item not found". For `SET` requests, the lite replica accepts the write and logs it for replay. This behavior aligns with caching semantics, which do not guarantee the retrieval of previously stored items but ensure retrieved values reflect the most recent `SET`.

Advanced features such as Bloom filters, geospatial indexes, HyperLogLog, streaming, Lua scripting, Redis modules, and some administrative commands such as debug and slowlog are not supported. These operations are rarely on the hot path but account for more than 50K lines of code.

LevelDB. A key distinction between a caching service and a database service is that databases guarantee the retrieval of stored information. The lite replica directly closes the connection if the requested key is not in the State Store. This is because LiteLib cannot distinguish whether the key does not exist or the key-value pair was evicted in normal mode due to memory constraints. The connection reset mimics the behavior of the baseline system, where clients experience a connection loss when the full version is down.

The lite replica does not support iterators, range scans, or snapshots. These operations are rejected with an error. This simplification eliminates the need for LevelDB's internal memtable, SSTable management, compaction logic, and versioning, over 5K lines of complex C++ code. The lite replica's state representation is a simple hash map.

HDFS DataNode. The lite replica maintains the namespace and block information in the State Store and manages file blocks on disk. Additionally, to support the replication mechanism, it periodically sends heartbeats to the NameNode.

MySQL. This is the most complex application we applied LiteLib to. The lite replica supports simple `SELECT`, `INSERT`, `UPDATE`, and `DELETE` statements on hot rows. It leverages the full version's query cache and maintains a row cache, which stores row fields to assist query cache maintenance and directly serves single-row queries. In normal mode, the lite replica extracts row data from QUERY responses and stores it in the State Store. If it cannot determine whether a row is affected or lacks the state to compute its value, it conservatively marks columns as `UNKNOWN` or invalidates the row. The full version shares its query cache with the lite replica via shared memory, and the lite replica keeps its own local query cache. The full version invalidates its query cache per table on every write; the lite replica instead captures these writes and invalidates only the affected rows using the row cache.

In emergency mode, the lite replica migrates the shared query cache. For data queries, it returns results from the query cache when available; if the required state is missing or `UNKNOWN`, it returns an error. For modification requests, it updates the row cache and adjusts affected query cache entries accordingly. If unknown columns prevent a definitive update, it either invalidates the affected entries or rejects the request.

The lite replica omits the query optimizer and many advanced features such as stored procedures, triggers, event schedulers, and full-text search. These are rarely used in typical OLTP workloads. It also does not support transactions beyond single-statement auto-commit.

7 Evaluation

Our evaluation answers several key questions: (1) What is the development effort for lite replicas? (§7.1) (2) Can LiteLib reduce the service gap (§7.2.1), and serve enough requests during failures to contain the impact of the failure (§7.2.2)? (3) How much overhead does LiteLib incur (§7.3)?

We also evaluate diverse workloads (Appendix C.2), the impact of memory quota configuration (Appendix C.3), and overhead under different loads (Appendix C.4).

Experiment Setup. All experiments are conducted on CloudLab c220g5 bare-metal machines running Ubuntu 22.04. Checkpointing baselines use CRIU [3].

LevelDB. We employ a wrapper [2] that transforms LevelDB into a server. The workload consists of 20% writes at 40K req/s, generated by a Zipfian distribution over 64 persistent connections. Each request has a 1 s timeout and up to 5 retries. Keys are 16 bytes, values 100 bytes, and the database is preloaded with six million entries. The write buffer is 512 MB, and the service runs on 3 cores.

Memcached. We adopt a setup similar to [36], where Memcached serves as a write-through cache for a MySQL database with 34.6 million entries. Memcached 1.6.14, MySQL 8.0.39, and Nginx run on three separate machines. Requests follow a Zipfian distribution with a 5% write ratio.

Redis. We evaluate both vanilla Redis 7.4.1 and Redis with a passive replica. Three Redis Sentinel instances with default settings enable automatic failover. We use YCSB with an 80/20 read-write mix on a 1M-record dataset (1 KB values).

HDFS. We deploy an HDFS 3.3.6 cluster with one NameNode and three DataNodes on separate machines, with a replication factor of 2. The TestDFSIO benchmark writes ten 1000 MB files to the file system.

MySQL. We use Sysbench to evaluate MySQL 5.7 under four deployments: standalone, active-passive replication (a primary and a replica on separate machines, with ProxySQL and Orchestrator on a third), active-active replication (MySQL NDB Cluster 7.6 with replication factor 2, where each machine

Software	Vanilla	Lite Replica	Patch[1]	Dev. Time
Memcached	60K	567 (0.9%)	63 (0.1%)	3 days
Redis	123K	801 (0.6%)	149 (0.1%)	5 days
LevelDB	23K	924 (3.9%)	89 (0.4%)	5 days
HDFS D.N.	56K[2]	1,574 (2.8%)	N/A[3]	2 weeks
MySQL	1.3M[4]	3,092 (0.2%)	243 (<0.1%)	3 weeks

Table 1. Code sizes of lite replicas and full versions, and their development time. [1] Patch to the full version supporting the *eBPF* and *embedded* options. [2] Underestimates: only counts code related to the DataNode in `org.apache.hadoop.hdfs`. [3] N/A: the *eBPF* and *embedded* options are not available for Java in the current prototype. [4] Underestimates: only C/C++ code in `libmysqld`, `sql`, `sql-common`, and `storage` is counted.

hosts one SQL node and one data node, plus a separate management node), and LiteLib. For active-active replication, we evaluate (1) ProxySQL as an external load balancer, and (2) client-side support for multiple endpoints. The workload issues random reads and writes under a Zipfian distribution.

DeathStar. We use the DeathStarBench Social Network (v0.4.1), a realistic microservice application with the Facebook Reed98 dataset [60]. The workload consists of 90% timeline reads and 10% post compositions, with requests following a Zipfian distribution to approximate real user traffic [27]. Each Memcached server is allocated one CPU core.

Correctness Validation. For each lite replica, we first run the application's existing test suite against the lite replica in emergency mode, with expected errors for unsupported operations. We also conduct fault-injection experiments that crash the full version and verify that the clients observe consistent behavior by comparing the responses during failure and after recovery with a failure-free baseline. Across all tested scenarios, lite replicas either served correct responses or rejected requests with well-defined errors.

7.1 Development Effort

Table 1 shows the lines of code (LoC) for each lite replica along with development time. The LiteLib library (5.7K SLOC) is shared across applications. The lite replicas average only 2% of the original code size. Besides supporting fewer operations with simpler implementations, the lite replicas stay compact because they reuse LiteLib's State Store & Logging Module. For HDFS, about half of the lite replica's code is dedicated to the network protocol and can be reduced by using third-party libraries. Modifications to the full version are minimal. Most add event registration callbacks (for *eBPF*), while the rest copy key data from complex request objects to the lite replica's simpler ones (for *embedded*).

In terms of development time, all except HDFS and MySQL require relatively little effort. For HDFS, most of the effort is spent understanding and implementing the network protocol. For MySQL, the effort is dominated by designing an appropriate state representation. This time would likely be shorter if the implementation were done by the original developers.

App.	Solution	Settings	Gap
Redis	A.P. Repl.	Failure detection: 30s (default)	33 s
		Failure detection: 5s	8 s
	LiteLib	Deploy Option: Proxy	1 ms
		Deploy Option: Embedded	13 ms
MySQL	A.P. Repl.	Binlog Repl.	5 s
	A.A. Repl.	NDB Cluster (Proxy Failover)	55 ms
	A.A. Repl.	NDB Cluster (Client Failover)	52 ms
	LiteLib	Deploy Option: Proxy	9 ms

Table 2. Service gap across applications. A.P. Repl. and A.A. Repl. denote active-passive and active-active replication, respectively.

Because the service is largely treated as a black box, the effort to implement a lite replica is primarily a one-time investment, as service APIs tend to remain stable.

7.2 Service Continuity under Failures

We evaluate service continuity along two orthogonal dimensions: (1) service gap (§7.2.1), which captures periods during which requests cannot make progress, and (2) service performance under and after failures (§7.2.2), which characterizes how much service remains available, how quickly performance recovers after a failure occurs, and how effectively LiteLib limits cascading and metastable failure effects.

7.2.1 Service Gap We measure the service gap from the server-side perspective. As shown in Table 2, active-passive replication incurs service gaps of several seconds. This gap typically consists of two components: (1) failure detection latency, and (2) coordination overhead to promote a replica. MySQL experiences a service gap of several seconds due to failure detection. Moreover, in deployments without a front-end proxy (*e.g.*, Redis), clients must additionally interact with sentinels to re-establish connections after failover. Active-active replication avoids coordination delays, but still relies on network timeouts for failure detection, incurring service gaps of over 40 ms.

By comparison, LiteLib avoids both sources of delay. Because LiteLib is deployed locally, it does not wait for network-based failure detection, nor does it require distributed coordination to promote a new primary. As a result, failover under LiteLib incurs only millisecond-scale service gaps. In the DeathStar experiment, Mcrouter reports no interrupted client connections during failover.

7.2.2 Performance and Recovery Table 3 shows that LiteLib provides substantial capacity during failures, reducing failure impact. We discuss each application below, with more details in Appendix C.1.

Metastable failures' sustaining effects can be categorized into two types: (1) workload amplification and (2) capacity degradation amplification [36]. We evaluate workload amplification in LevelDB, where it is triggered by retry storms, and evaluate capacity degradation in Memcached, where the effective capacity is reduced due to cache loss.

App.	Solution	Effective Service Capacity from Failed Node	Performance Impact on Clients and Healthy Nodes
LevelDB	Baseline	0 throughput; 300K+ unserved	+83% avg. latency
	Checkpoint	780K stale responses in 30 s after failure	212× avg. latency even w/o failures
	LiteLib	143K (67%) served during failure	Stable avg. latency
Memcached	Baseline	0% hit rate post-restart	3,224 s to 90% throughput
	Checkpoint	2K stale responses	7 s to 90% throughput
	LiteLib	33.8K additional end-to-end requests served	69 s to 90% throughput
Redis	Baseline	0 throughput	Sustain 27.0% of pre-failure throughput within 1 min.
	A.P. Repl.	20.6K stale responses	Sustain 45.5% of pre-failure throughput within 1 min.
	LiteLib	881.8K served during failure	Sustain 49.1% of pre-failure throughput within 1 min.
HDFS	A.A. Repl.	0 throughput	55.7% write shift; +59.9% healthy DataNode CPU time
	LiteLib	Stable throughput	No write shift; +22.5% healthy DataNode CPU time
MySQL	A.P. Repl.	0 throughput	3× traffic surge during failover
	A.A. Repl.	0 throughput	Healthy node experiences 43.5% more load; total throughput drops by 28.8%
	LiteLib	48.6% of requests are served during failure	Cascading failure avoided under modeled load shift

Table 3. Service performance and recovery behavior under failures across applications. A.P. Repl. and A.A. Repl. denote active-passive and active-active replication, respectively.

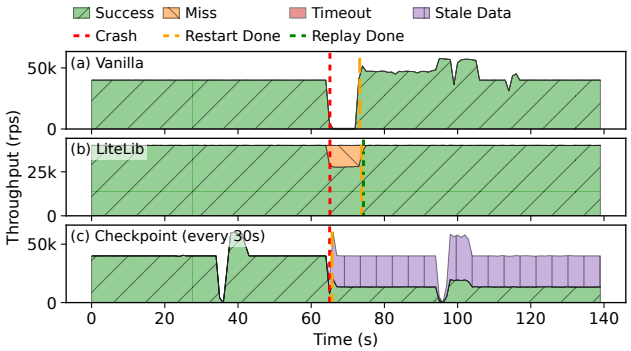


Figure 11. Impact of a LevelDB crash on throughput.

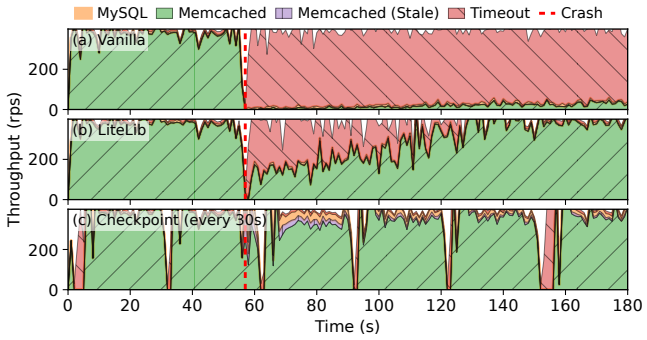


Figure 12. Impact of a Memcached crash on the throughput of a database system with Memcached deployed as a cache.

LevelDB. In the vanilla setup (Figure 11(a)), even with immediate restart, LevelDB requires 8.2 s to replay its write-ahead log. Retries then overload the system, elevating latency and sustaining poor performance throughout the one-minute experiment, signaling a metastable failure.

With LiteLib (Figure 11(b)), much of the workload is absorbed during downtime, reducing retry pressure and preventing sustained latency increases. Thanks to LiteLib's log compaction, the lite replica replays only 35K requests in 512 ms, fewer than the full version would have processed for the same writes without the failure. During replay, client-observed throughput also does not degrade. Checkpointing every 30 s blocks LevelDB for 2.4 s per snapshot, which explains its latency increase even without failures (Table 3).

Memcached. In the vanilla setup (Figure 12(a)), the crash at 57 s causes complete data loss, driving the cache miss rate to 100% upon restart. Throughput recovery depends on cache warm-up, but heavy load on the backend database leads to widespread timeouts. As a result, throughput reaches only 10.8% of the maximum at 180 s. With checkpointing every 30 s (Figure 12(c)), throughput recovers to 90% of its maximum within 7 s after failure but produces over 2K stale responses.

When LiteLib is deployed (Figure 12(b)), the lite replica performs a special *full-state replay* for caches: it writes back all objects in its State Store (18.3% of cache entries, 256K) to the restarted full version within 18 s, containing the failure impact. Notably, the Memcached throughput observed by clients immediately before replay completion (while still served by the lite replica) and immediately after (when service switches back to the full version) shows no observable difference, indicating no significant service degradation during replay.

Redis. Even with an immediate restart, vanilla Redis takes 45 s to restore its dataset. Active-passive replication returns stale responses because Redis replicates asynchronously. LiteLib cannot provide full capacity as replication does, but it serves requests from up-to-date state and sustains the highest throughput after the failure (Table 3).

HDFS. HDFS replicates blocks across DataNodes, so a DataNode failure loses no data, but its writes shift to the healthy DataNodes and raise their load, which risks cascading failures.

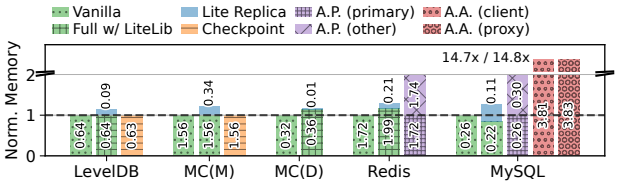


Figure 13. Memory usage overhead in normal operation. The value in each bar represents the absolute memory usage (GB). MC(M) and MC(D) refer to the Memcached used in the Metastable experiment (Figure 12) and the DeathStar experiment (§2), respectively.

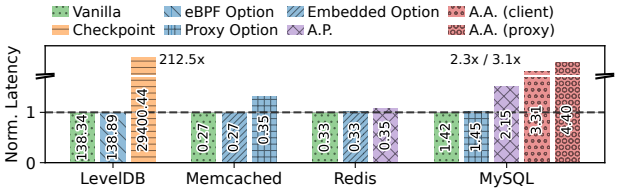


Figure 14. Latency overhead in normal operation. The value on each bar represents the average latency in milliseconds.

The lite replica keeps sending heartbeats to the NameNode and handles writes as a regular DataNode, so writes continue to reach it and the healthy DataNodes see no extra writes (Table 3). This shows that LiteLib can work alongside an application's own replication and mitigate cascading failures.

MySQL. In an active-active setting, when one replica fails, redirecting all requests to the survivor causes queue buildup and latency to exceed timeouts, indicating a cascading failure.

Reducing the load to 3.8K requests per second—approximating a scenario where the lite replica absorbs 48.6% of the shifted traffic—keeps the survivor stable without queue buildup. This suggests that LiteLib mitigates cascading failures by containing load amplification. Similar behavior appears in active-passive settings, even before considering the 3× surge during failover.

7.3 Performance and Resource Overhead

We evaluate the lite replica's memory usage, latency, and CPU utilization in normal mode to quantify overhead. Memory usage is quota-based and configurable to application needs; here, we adopt the same configuration as in previous experiments.

Memory. A significant benefit of LiteLib is its low memory usage relative to the service continuity it provides under failures, as shown in Figure 13. It incurs 21.3% average memory overhead, whereas replication almost doubles memory usage. We also measure how the memory quota influences the benefits in Appendix C.3.

Latency. Figure 14 shows average latency without failures. LiteLib adds minimal overhead: under the *eBPF* and *embedded* options, latency increases by less than 1%. The *proxy option* incurs modest overhead, reaching 2.1% in complex services like MySQL where application processing dominates.

CPU. Figure 15 shows that LiteLib increases CPU usage by 27.5% on average. In HDFS, the lite replica incurs a 34.5%

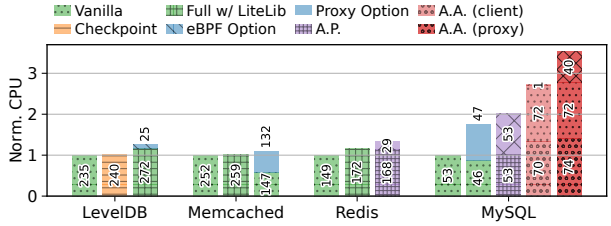


Figure 15. CPU utilization overhead in normal operation. The value on each bar represents the absolute CPU utilization (%). For active-active or active-passive replication, the stacked segments (from bottom to top) represent the first replica, the second replica, and auxiliary processes (*e.g.*, manager and proxy).

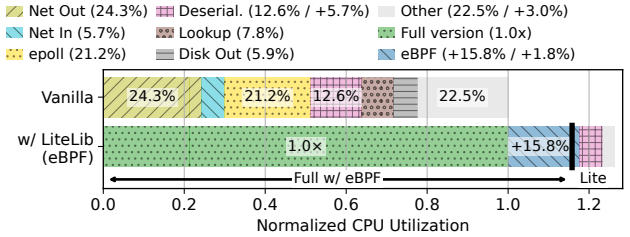


Figure 16. CPU utilization breakdown in LevelDB (*eBPF* option). All values are normalized to the CPU utilization of the vanilla full version. In the legend, a value without "+" is a component's share of the vanilla full version, and a value with "+" is CPU utilization added by LiteLib. Where an entry has two values, the first applies to the full version and the second to the lite replica.

overhead and shows room for improvement, as the packet transfer and parsing code is not yet optimized.

Overall, runtime overhead is small for three reasons: (1) the lite replica maintains less state, making operations cheaper, and processes them asynchronously; (2) in normal mode, it only updates state, allowing read operations to be skipped, which further limits overhead since most workloads are not write-dominated; and (3) network I/O is a major bottleneck in the full version, and traditional replication incurs additional communication overhead, while the lite replica is deployed locally with the full version, making communication much cheaper. Figure 16 breaks down LevelDB's CPU utilization under the *eBPF* option. Network I/O takes more than half of the vanilla full version's CPU time. With LiteLib, the eBPF program adds 15.8% to the full version, and the lite replica adds another 10.5%, mostly to deserialize the forwarded requests and responses, for a total increase of 26.3%.

8 Limitations

LiteLib is not suitable for all application types. This includes computation-intensive applications (*e.g.*, image processing, data analytics). The lite replica would incur similar CPU costs to the full version yet preserve little meaningful state to provide benefit during failures. LiteLib is less straightforward for applications that rely on active participation in distributed consensus protocols. A lite replica can observe protocol traffic to maintain its state log and serve reads or buffer writes, but it cannot safely assume active roles such

as leader election, voting, or proposing new values. Nodes whose sole purpose is active coordination should rely on native replication. Our HDFS DataNode use case shows a middle ground: the lite replica handles data-path I/O while deferring full protocol participation until recovery. For applications with non-deterministic operations, correctness requires that resulting state changes be fully reflected in responses; otherwise, the lite replica cannot fully synchronize with the full version in normal mode unless the *embedded* option passes additional information.

The lite replica may contain implementation bugs, which can lead to incorrect results in emergency mode. However, its smaller codebase makes it easier to test and validate, and it can reuse the full version's test suites. Moreover, such bugs do not affect normal operation and only propagate if incorrect results are logged and replayed.

The lite replica also introduces a potential new failure point. With the *eBPF* option, it is off the critical path, so failures do not affect the full version. Crashes of the *proxy* option are temporary, as clients can fall back to the full version. With the *embedded* option, the LitePlugin runs as separate threads but shares the full version's failure domain. This option trades isolation for lower overhead, although the plugin's simpler design may reduce the risk.

Finally, LiteLib does not address all metastable failures. To reduce coordination complexity, the lite replica is not designed to serve requests concurrently with the full version, and thus cannot break failure loops when a degraded full version still offers higher capacity. However, by serving requests during downtime, it enables restart opportunities with minimal service gaps and reduced workload amplification.

9 Related Work

There is a rich body of work on tolerating software failures through redundancy. The most common approach is replication, including process pairs [31] and virtual machines [13]. For fast failover, replicas are actively synchronized, often in lockstep. It is effective but costly both in performance and resource usage. Since the primary needs to wait for the backup to perform the same operation, a significant delay is introduced to clients. Running the backup also requires a separate machine (remote replication) or the same amount of resources as the primary. Shared logs coordinate logging across nodes and are often co-designed with replication. They are helpful for versatility [6, 8] and performance [48]. Logging solutions are also used for reliability and recovery [7, 29, 38, 54, 71]. However, they remain multi-node systems with higher overhead, and the backup runs identical code, making it vulnerable to the same deterministic failures.

Another common approach is checkpointing [3, 18, 63, 68, 69]. Compared to replication, it incurs a smaller performance penalty and lower resource usage. It also allows a buggy process to roll back to a working state. However, the checkpointed state is from the past, incurring significant data loss.

Client-side data caching typically allows continued client access during server failures by serving only reads from cached data within a time bound. Some systems also support writes [41], but may expose different views to clients during failures or incur substantial consistency-maintenance costs.

N-version programming [5, 20] tolerates design flaws and deterministic bugs by developing and running multiple functionally equivalent versions and voting on their outputs. LiteLib addresses a different problem: cost-effective failure containment during failures.

Hystrix [52] is a library to provide latency tolerance and fault tolerance by implementing the circuit breaker pattern, load shedding, and other mechanisms. Similarly, the Google SRE Book [10] discusses strategies to address cascading failures, such as load shedding and graceful degradation. These techniques focus on reducing the workload to prevent overload. LiteLib enhances capacity during service interruptions and can work together with these solutions.

RPCShield [19] statically detects microservice dependencies whose failures propagate to their callers, so that developers can identify cascading-failure risks before deployment. LiteLib addresses a later stage: it contains the impact of the failures that still occur.

Overload control systems such as Breakwater [22] and Protego [23] admit requests based on queueing delay and on throughput gains under lock contention, respectively, while Atropos [35] cancels the requests that monopolize critical resources. Defcon [50] mitigates overload by disabling less-critical features. These systems keep a running server within capacity and are orthogonal to LiteLib, which adds capacity while the server recovers.

Slicer [4] provides degraded service using a read-only backup distributor. While both use simplified replicas, their goals differ: LiteLib is a general library that enables transparent, seamless failover across systems and supports writes.

Another line of work shortens recovery itself [16, 39]. LiteLib covers the downtime that remains, so faster recovery and LiteLib complement each other.

10 Conclusion

We propose lite replicas, a new method for efficiently containing failure impact, along with a library that enables developers to apply this approach to their stateful applications. We apply LiteLib to widely used datacenter software components. Our extensive evaluations show that LiteLib provides high service continuity during failures, mitigates cascading and metastable failures, and incurs low overhead with modest effort. It offers a new design point in high-availability solutions.

Acknowledgments

We thank the anonymous reviewers and the shepherd for their valuable feedback. Our experiments use machine resources in CloudLab [26]. This work was supported in part by NSF grants CNS-2317698, CNS-2317751, and CCF-2318937.

References

[1] DeathStarBench. https://github.com/delimitrou/DeathStarBench. Accessed: 2026-10-05.

[2] Redis-leveldb. https://github.com/KDr2/redis-leveldb. Accessed: 2026-10-05.

[3] Checkpoint/restore in userspace (CRIU). https://criu.org/Main_Page, 2012. Accessed: 2026-10-05.

[4] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-Sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, November 2016. USENIX Association.

[5] A. Avizienis. The N-Version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12):1491–1501, December 1985.

[6] 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. Virtual consensus in Delos. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 617–632, Virtual Event, November 2020. USENIX Association.

[7] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Transactions on Computer Systems*, 31(4):10:1–10:24, December 2013.

[8] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 325–340, Farmington, PA, USA, November 2013. Association for Computing Machinery.

[9] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, Bordeaux, France, April 2015. Association for Computing Machinery.

[10] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 1st edition, 2016.

[11] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *ACM SIGOPS Operating Systems Review*, 21(5):123–138, November 1987.

[12] Michael D. Bond and Kathryn S. McKinley. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–72, San Jose, CA, USA, October 2006. Association for Computing Machinery.

[13] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 1–11, Copper Mountain, CO, USA, December 1995. Association for Computing Machinery.

[14] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, pages 221–227, Ann Arbor, MI, USA, June 2021. Association for Computing Machinery.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, June 2013. USENIX Association.

[16] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 04)*, pages 31–44, San Francisco, CA, December 2004. USENIX Association.

[17] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[18] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, August 2017.

[19] Milind Chabbi, Sonal Mahajan, Ivan Beschastnikh, René Just, Yuxin Wang, Yufan Xu, Elton Pinto, Seemanta Saha, Manu Sridharan, Abhishek Jha, Sandeep Koushik Sheshadri, and Mayank Bansal. RPC-Shield: Defending microservices against cascading failures. In *Proceedings of the ACM SIGOPS 32nd Symposium on Operating Systems Principles*, SOSP '26, pages 802–826, Prague, Czech Republic, September 2026. Association for Computing Machinery.

[20] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, 'Highlights from Twenty-Five Years'*, pages 113–119, Pasadena, CA, USA, June 1995. IEEE.

[21] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. Push-Button reliability testing for Cloud-Backed applications with Rainmaker. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1701–1716, Boston, MA, April 2023. USENIX Association.

[22] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with Breakwater. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314, Virtual Event, November 2020. USENIX Association.

[23] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Protego: Overload control for applications with unpredictable lock contention. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 725–738, Boston, MA, April 2023. USENIX Association.

[24] Julien Desgats. How we built rate limiting capable of scaling to millions of domains. https://blog.cloudflare.com/counting-things-a-lot-of-different-things, June 2017. Accessed: 2026-10-05.

[25] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: seamless reconfiguration and total order in a scalable shared log. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 325–338, Santa Clara, CA, February 2020. USENIX Association.

[26] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 1–14, Renton, WA, July 2019. USENIX Association.

[27] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their

hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 3–18, Providence, RI, USA, April 2019. Association for Computing Machinery.

[28] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? An empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 126–141, San Francisco, CA, USA, November 2022. Association for Computing Machinery.

[29] Dimitra Giantsidi, Emmanouil Giortamis, Nathaniel Tornow, Florin Dinu, and Pramod Bhatotia. Flexlog: A shared log for stateful serverless computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, pages 195–209, Orlando, FL, USA, June 2023. Association for Computing Machinery.

[30] Amogha Udupa Shankaranarayana Gopal, Raveendra Soori, Michael Ferdman, and Dongyoon Lee. TAILCHECK: A lightweight heap overflow detection mechanism with page protection and tagged pointers. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 535–552, Boston, MA, July 2023. USENIX Association.

[31] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the Fifth Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, Los Angeles, CA, USA, January 1986. IEEE Computer Society.

[32] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '14, pages 1–14, Seattle, WA, USA, November 2014. Association for Computing Machinery.

[33] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing? Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 1–16, Santa Clara, CA, USA, October 2016. Association for Computing Machinery.

[34] Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 156–164, Boston, MA, USA, October 2004. Association for Computing Machinery.

[35] Yigong Hu, Zeyin Zhang, Yicheng Liu, Yile Gu, Shuangyu Lei, Baris Kasikci, and Peng Huang. Mitigating application resource overload with targeted task cancellation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, pages 270–285, Seoul, Republic of Korea, October 2025. Association for Computing Machinery.

[36] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, Carlsbad, CA, July 2022. USENIX Association.

[37] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 150–155, Whistler, BC, Canada, May 2017. Association for Computing Machinery.

[38] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 691–707, Virtual Event, Germany, October 2021. Association for Computing Machinery.

[39] Yuzhuo Jing, Yuqi Mai, Angting Cai, Yi Chen, Wanning He, Xiaoyang Qian, Peter M. Chen, and Peng Huang. Optimistic recovery for high-availability software via partial process state preservation. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, SOSP '25, pages 305–321, Seoul, Republic of Korea, October 2025. Association for Computing Machinery.

[40] Manos Karpathiotakis, Vlassios Rizopoulos, Basri Kahveci, Tiziano Carotti, Artem Gelum, Hazem Nada, and Yuri Dolgov. Scribe: How Meta transports terabytes per second in real time. *Proceedings of the VLDB Endowment*, 18(12):4817–4830, August 2025.

[41] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[42] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[43] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, pages 25–33, San Jose, CA, USA, October 2006. Association for Computing Machinery.

[44] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 155–162, Bertinoro, Italy, May 2019. Association for Computing Machinery.

[45] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. RESIN: A holistic service for dealing with memory leaks in production cloud infrastructure. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 109–125, Carlsbad, CA, July 2022. USENIX Association.

[46] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[47] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 412–426, Seattle, WA, USA, November 2021. Association for Computing Machinery.

[48] Xuhao Luo, Shreesha G. Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. LazyLog: A new shared log abstraction for low-latency applications. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pages 296–312, Austin, TX, USA, November 2024. Association for Computing Machinery.

[49] Cong Ma and Matt Ranney. Failure mitigation for microservices: An intro to Aperture. https://careersatdoordash.com/blog/failure-mitigation-for-microservices-an-intro-to-aperture/, March 2023. Accessed: 2026-10-05.

[50] Justin J. Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, David Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard. Defcon: Preventing overload with graceful feature degradation. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 607–622, Boston, MA, July 2023. USENIX Association.

[51] Modern Networks. Cloud outages and the resilience reckoning: What businesses must rethink. https://modern-networks.co.uk/news/cloud-outages-and-resilience, November 2025. Accessed: 2026-10-05.

[52] Netflix. Hystrix. https://github.com/Netflix/Hystrix, 2012. Accessed: 2026-10-05.

[53] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, April 2013. USENIX Association.

[54] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, August 2017.

[55] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[56] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christina D. Richards, and Mengzhi Wang. Zanzibar: Google's consistent, global authorization system. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 33–46, Renton, WA, July 2019. USENIX Association.

[57] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, David Oppenheimer, Naveen Sastry, William Tetzlaff, Jonathan Traupman, and Noah Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, EECS Department, University of California, Berkeley, March 2002.

[58] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of the USENIX Winter 1995 Technical Conference*, pages 213–223, New Orleans, LA, USA, January 1995. USENIX Association.

[59] Anna Povzner, Prince Mahajan, Jason Gustafson, Jun Rao, Ismael Juma, Feng Min, Shriram Sridharan, Nikhil Bhatia, Gopi Attaluri, Adithya Chandra, Stanislav Kozlovski, Rajini Sivaram, Lucas Bradstreet, Bob Barrett, Dhruvil Shah, David Jacot, David Arthur, Manveer Chawla, Ron Dagostino, Colin McCabe, Manikumar Reddy Obili, Kowshik Prakasam, Jose Garcia Sancio, Vikas Singh, Alok Nikhil, and Kamal Gupta. Kora: A cloud-native event streaming platform for Kafka. *Proceedings of the VLDB Endowment*, 16(12):3822–3834, August 2023.

[60] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI '15, pages 4292–4293, Austin, TX, USA, January 2015. AAAI Press.

[61] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX ATC 04)*, pages 29–44, Boston, MA, June 2004. USENIX Association.

[62] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. If at first you don't succeed, try, try, again...? Insights and LLM-informed tooling for detecting retry bugs in software systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, pages 63–78, Austin, TX, USA, November 2024. Association for Computing Machinery.

[63] Foteini Strati, Michal Friedman, and Ana Klimovic. PCcheck: Persistent concurrent checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, pages 811–827, Rotterdam, Netherlands, March 2025. Association for Computing Machinery.

[64] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. Fail through the cracks: Cross-system interaction failures in modern cloud systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, pages 433–451, Rome, Italy, May 2023. Association for Computing Machinery.

[65] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, Copper Mountain, CO, USA, December 1995. Association for Computing Machinery.

[66] Teng Wang, Zhouyang Jia, Shanshan Li, Si Zheng, Yue Yu, Erci Xu, Shaoliang Peng, and Xiangke Liao. Understanding and detecting on-the-fly configuration bugs. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pages 628–639, Melbourne, Victoria, Australia, May 2023. IEEE Press.

[67] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala. Checkpointing and its applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 22–31, Pasadena, CA, USA, June 1995. IEEE Computer Society.

[68] Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, T. S. Eugene Ng, and Yida Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 364–381, Koblenz, Germany, October 2023. Association for Computing Machinery.

[69] Chen Xu, Rudi Poepsel Lemaitre, Juan Soto, and Volker Markl. Fault-tolerance for distributed iterative dataflows in action. *Proceedings of the VLDB Endowment*, 11(12):1990–1993, August 2018.

[70] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, Virtual Event, November 2020. USENIX Association.

[71] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, Virtual Event, November 2020. USENIX Association.

A Motivation Details

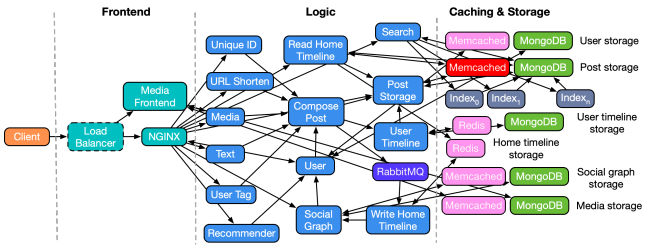


Figure 17. DeathStar Social Network microservice architecture. Adapted from the DeathStarBench repository [1] with the failing component recolored. The red Memcached service is the component experiencing the failure. It consists of two Memcached instances, with a Mcrouter instance acting as a load balancer in front of them.

Figure 17 shows the architecture of the microservices we use for the motivating experiment in §2.

B Correctness Proof

This appendix proves that LiteLib preserves correctness across failures under two consistency configurations.

Strict mode (response equivalence) To provide response equivalence, LiteLib enables a *strict mode* in which the full version exposes a *sequence ID* (`seq`) that identifies its global serialization position for each completed request. Using `seq`, LiteLib (i) aligns its State Store with a well-defined cutover-prefix state when entering emergency mode, (ii) serves emergency-mode requests only after draining and processing all previously observed responses (eliminating divergence due to in-flight responses), and (iii) replays all accepted emergency-mode updates back to the full version in a manner consistent with the service serialization order.

Section 3.9 describes how the full version exposes `seq` without changing the external network protocol or client-visible semantics, and why single-writer services need no explicit `seq`. Exposing `seq` does not alter service behavior; it merely reveals an internal serialization position that already exists logically within the system.

Relaxed mode When response equivalence is not required, LiteLib can operate in a *relaxed mode* that does not rely on `seq`. In this configuration, LiteLib provides a weak consistency guarantee centered on *replay reconciliation*: after the full version recovers, replay eventually makes the full version reflect the effects of emergency-mode accepted writes for the objects modified during emergency mode, under a restricted class of writes (formalized in Appendix B.2). Base-dependent operations, whose outcomes may change when replayed from a different base state, require additional mechanisms to ensure correctness; we discuss such mechanisms in Appendix B.2.

Service model Let $\mathcal{R}$ be the set of requests and $\mathcal{O}$ the set of responses. We model the service as a state machine $M = (S, s_0, \delta, \omega)$, where S is the set of states, s_0 is the initial state, $\delta : S \times \mathcal{R} \to S$ is the transition function, and $\omega : S \times \mathcal{R} \to \mathcal{O}$ is the response function.

Requests and orders Let $\prec_F$ denote the full version's serialization order over requests.

Timeline and states Figure 18 defines four named points and the corresponding state variables used throughout the proofs.

Assumptions Our correctness arguments rely on the following assumptions about the application and LiteLib.

1. **Internal-external state consistency (full version).** At any given point, we assume that the full version's internal state is consistent with respect to externally observable effects across normal operations and failures. This includes application failures and transient failures in normal mode. After a failure, the application's recovery procedure restores a state that is *crash-consistent* with respect to client-visible external observations: any request whose response was successfully observable before the crash has its effects reflected in the recovered state (*e.g.*, via WAL/redo). Similarly, transient failures either do not produce state changes without externally observable effects, or such cases are mitigated so that the state remains consistent with external observations. If a request is committed but its response is not delivered before the failure, or is delivered as an error due to transient failures, the client may observe ambiguity about whether the operation took effect. This reflects inherent ambiguity in the base service's failure and retry semantics, rather than a new issue introduced by LiteLib. LiteLib preserves the underlying service's failure semantics rather than creating new ones.
2. **Deterministic operations.** The transition and response functions are deterministic.
3. **Correct lite semantics (for observed and served requests).** For any request it observes in normal mode or serves in emergency mode, the lite replica applies the same transition function as the full version and, when serving, the same response function. In emergency mode, requests that cannot be handled with the same semantics or lack the necessary state are rejected with an error, without modifying state.
4. **Correct logging.** Each accepted write request is recorded via either an Action Node or a Result Node. Result Nodes are used only when the resulting state update fully captures all side effects required for correct replay; otherwise, an Action Node is used.
5. **Replay completion.** LiteLib's admission control ensures replay eventually completes.
6. **(Strict mode only) Seq order.** The full version assigns sequence IDs consistent with $\prec_F$: if $r_1 \prec_F r_2$, then $\mathrm{seq}(r_1) < \mathrm{seq}(r_2)$, and LiteLib applies full-version effects in increasing `seq` order.

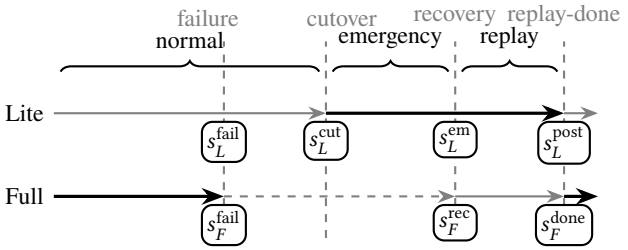


Figure 18. Timeline and states around a crash. Thick segments indicate the component currently serving requests.

B.1 Strict Mode: Response Equivalence and State Convergence

Normal-mode response processing and in-flight responses In normal mode, the full version serves requests, and LiteLib

updates its own State Store only after it *receives* and *processes* the corresponding responses. Response processing at LiteLib is asynchronous: responses may arrive and be applied to the State Store later than when the full version executes the requests. When `seq` is available, LiteLib processes a response only if its `seq` immediately follows, in the full version's serialization order, the `seq` of the last processed response. This enforces prefix application in serialization order and prevents reordering at LiteLib. We call a request *in-flight* if the full version has sent its response but LiteLib has not yet processed it.

Cutover to emergency mode The point at which LiteLib enters emergency mode depends on the deployment option. With the *eBPF* and *embedded* options, clients connect to the full version directly, and the transition happens when the LitePlugin, invoked from the full version's signal handler, transfers all client connections to the lite replica. With the *proxy* option, clients connect to LiteLib, and the transition happens when all connections between LiteLib and the full version are lost. With any option, operators can also trigger the transition manually. Beyond this point, no further responses from the full version can be observed, which naturally induces a *cutover barrier*.

Upon detecting the full version's failure, LiteLib stops forwarding new requests and drains all responses that are *observable* at LiteLib (*i.e.*, those forwarded before the failure). LiteLib processes all such responses and applies their effects to the State Store, taking it from s_L^{fail} to s_L^{cut}, before serving any emergency-mode request. This ensures that emergency-mode execution begins from a well-defined applied prefix.

Lemma 1 (Cutover alignment with observed responses)**.** *In strict mode, immediately before LiteLib serves the first request in emergency mode, s_L^{cut} is consistent with the full version's state s_F^{fail} on every object held in the State Store.*

Proof. By construction of the cutover barrier, LiteLib processes *all* responses before serving any emergency-mode request, yielding s_L^{cut}. In strict mode, these responses carry sequence IDs consistent with the full version's serialization order, so applying them in increasing `seq` order yields the same effects as executing the corresponding prefix at the full version. Since the State Store is updated only through processed responses and the lite replica derives the same updates implied by the full version's semantics for those responses, s_L^{cut} matches s_F^{fail} on every object held in the State Store. □

Emergency-mode execution During emergency mode, LiteLib serves a subsequence of requests. LiteLib serves a request only if all state required by $\delta(s, r)$ and $\omega(s, r)$ is present (§3.6); otherwise, the request is rejected.

Theorem 1 (Emergency-mode response equivalence)**.** *In strict mode, let $r_1, \ldots, r_n$ be the requests LiteLib serves in emergency mode, in order. For every i, the response LiteLib returns to r_i is identical to the response the full version would have produced had it, starting from s_L^{cut}, executed $r_1, \ldots, r_i$ in that order.*

Proof. By induction on i. Before r_1, Lemma 1 gives that the lite state and the full version's state agree on every object held in the State Store. Assume they agree after $r_1, \ldots, r_{i-1}$. LiteLib serves r_i only when all state it reads is present, so r_i observes the same pre-state in both executions; since the lite replica implements the same δ and ω for served requests, it produces the same response and the same state update, so the states agree after r_i as well. □

Logging, compaction, and dependencies LiteLib logs all write requests that it accepts during emergency mode. To reduce log volume, LiteLib applies log compaction: for each object, the final modification performed during emergency mode is retained, either as an Action Node that records the original request (and its effects) or as a Result Node that records the resulting state update. Intermediate Result Nodes that are overwritten are removed from the log; Action Nodes are never removed by compaction.

Log compaction is safe only when it does not remove information needed to replay later requests. To capture this, LiteLib tracks dependencies between Action Nodes and Result Nodes. Intuitively, an Action Node may require the exact state produced by an earlier update as the base for a subsequent operation. When such a dependency exists, LiteLib pins the corresponding Result Node.

Formally, let $R(x)$ denote a Result Node that records the resulting state update to object x, and let A denote an Action Node. We say A *depends* on $R(x)$ if correct replay of A requires the full version to reflect the state update recorded by $R(x)$ before replaying A. If any Action Node depends on a Result Node $R(x)$, LiteLib marks $R(x)$ as *pinned*. A pinned Result Node keeps a copy of x's value at the time it is pinned, which is the value replayed for it. Pinned Result Nodes are not overwritten or removed by later modifications to the same object.

Pinned-result preservation For any object x, LiteLib removes or overwrites a Result Node for x only if that Result Node is not pinned.

Replaying Upon recovery of the full version, when the lite replica is in state s_L^{em}, LiteLib replays the compacted log while continuing to serve requests. Replay preserves per-connection ordering and the relative order of modifications recorded by LiteLib.

Because the lite replica keeps serving during replay, the log may grow: writes accepted during replay are appended in chronological order and replayed in the same pass; those from requests still being served by the lite replica when LiteLib stops admitting new requests are replayed in a second round (§3.7). Compaction during replay removes only Result Nodes superseded by such later entries, so every removed entry is

either already replayed or replaced by a later entry. Throughout, L denotes the compacted log at the time replay completes.

Lemma 2 (Prefix replay consistency)**.** *Let L be the compacted replay log in replay order. After replaying any prefix $L[1..k]$, the full version's state is consistent with LiteLib's State Store after the effects of the entries in $L[1..k]$ have been applied. The only effects that may be absent are updates removed by compaction because they are superseded by later entries for the same object. Such updates are safe to omit, because any Result Node needed by a later Action Node is pinned and preserved in L.*

Proof. We prove the lemma by induction on k.

Base case ($k = 0$). Before replay begins, the recovered full version restores to s_F^{rec} (Figure 18). By Assumption 1, s_F^{rec} is crash-consistent with externally observable effects prior to the crash. In particular, it is consistent with s_L^{cut}, and thus provides a valid base for replay.

Inductive step. Assume the claim holds after replaying $L[1..k-1]$. Consider the k-th replay entry.

Case 1: The k-th entry is an Action Node. By dependency tracking, all Result Nodes required by this Action Node are pinned and therefore have not been removed or overwritten. Replaying the Action Node in order produces the same effects as during emergency mode at LiteLib.

Case 2: The k-th entry is a Result Node for object x. Replaying it updates x to the recorded resulting state.

In both cases, after applying the k-th entry, the full version remains consistent with LiteLib for all replayed effects that remain semantically relevant; any remaining differences are confined to updates that will be overwritten by a later entry and are not required by subsequent replay steps. □

Theorem 2 (Convergence after recovery)**.** *After the full version recovers and LiteLib completes replay, the full version reaches a state s_F^{done} that reflects the complete net effect of all write requests accepted by LiteLib from cutover until replay completes. Moreover, s_F^{done} is consistent with LiteLib's post-replay state s_L^{post} on every object held in the State Store (Figure 18).*

Proof. By Lemma 2, after replaying the entire log, there are no remaining later entries that can supersede prior updates. Thus, the caveat of Lemma 2 no longer applies, and the full version's state is consistent with LiteLib for all replayed effects. Since LiteLib logs all accepted writes and replay preserves ordering constraints, the full version reflects the complete net effect of all writes accepted from cutover until replay completes. □

B.2 Relaxed Mode: Scoped Eventual Consistency

Relaxed contract In relaxed mode, LiteLib does not attempt to match the full version's global order $\prec_F$. Emergency-mode responses are computed on the lite replica's local view (*i.e.*, they are well-defined with respect to the current State Store), but need not match any single global serialization of the full version.

After recovery, LiteLib reconciles emergency-mode writes via replay. Since replay may re-execute operations from a different base state s_F^{rec}, a clean guarantee requires restricting the class of emergency-mode writes.

Scope and write classes Let Scope be the set of objects modified by emergency-mode accepted writes, *i.e.*, the set of objects that appear in the compacted replay log.

We call a write request *base-dependent* if replaying the original request can produce different outcomes when starting from different base states (*e.g.*, many conditional updates that depend on the exact observed value or version). We call a write *base-independent* if its committed effect can be represented and replayed without depending on the current value at replay time (*e.g.*, unconditional overwrite, delete, or a commutative or base-independent delta such as an increment-by-1 update).

Key idea In relaxed mode, replay starts from the recovered full state s_F^{rec}, which may differ from the lite replica's view at cutover. To ensure that replay reproduces the effects accepted during emergency mode, LiteLib logs each accepted write in a *replay-safe* form. In most cases, an Action Node records the original request, and the request itself is replay-safe (*e.g.*, a commutative delta such as "+1"), so replaying the Action Node yields the same effect even if the base state differs. When a write's effect is more naturally represented as a concrete state update (or to enable log compaction), LiteLib records a Result Node that captures the accepted resulting update and replays it directly.

Operations that are *not* replay-safe as raw requests (*i.e.*, whose effects depend on the exact observed base state, such as many conditional updates) must either be transformed into replay-safe forms (*e.g.*, deltas or CRDT-style updates) or handled via a stronger path.

Emergency-mode response staleness Because the full version is unavailable during emergency mode, the lite replica may serve requests from a view that is behind (or otherwise differs from) the full version's failure-time state. We capture this as *staleness*: emergency responses are always semantically consistent with the lite state they execute on, but may be stale relative to the full version.

Lemma 3 (Emergency-mode response validity)**.** *For every request r served by LiteLib in emergency mode, the returned response equals $\omega(s, r)$ for the lite state s on which LiteLib executes r. Consequently, the response is semantically valid under the service model for that state; any mismatch with the full version is attributable to staleness* (i.e.*, the lite view differs from the full version's failure-time state), not to producing an arbitrary response.*

Proof. By the relaxed contract and the "Correct lite semantics" assumption, whenever LiteLib serves r it computes the response using the same response function ω on its current lite

state s, and advances its state using δ. Therefore, the returned response is exactly $\omega(s, r)$. If s differs from the full version's failure-time state, this manifests as staleness of the view, but the response remains semantically well-defined. □

Theorem 3 (Scoped consistency)**.** *Assume every accepted write is logged in a replay-safe form: as an Action Node whose request is replay-safe (*e.g.*, commutative or base-independent), or as a Result Node that fully captures the accepted resulting update for replay. After the full version recovers to* s_F^{rec} *and* LiteLib *completes replay, for every object* $x \in$ Scope*, the full version reflects the net effect of all accepted writes on* x *as recorded in the compacted log.*

Proof. By definition, the compacted log records the net effect of accepted writes on each $x \in$ Scope. Under the assumption that each log entry is replay-safe, replay applies each recorded effect without relying on matching the lite replica's base state at cutover: Action Nodes correspond to replay-safe requests whose effects are invariant to the base state (*e.g.*, commutative deltas), and Result Nodes directly enforce the accepted resulting updates. Since replay eventually completes (Replay completion) and the full version is not serving requests during replay, the recorded net effect is eventually reflected at the full version for each $x \in$ Scope. □

Base-dependent operations and stronger requirements Supporting base-dependent operations under weak consistency is a well-studied problem (*e.g.*, disconnected operation with reconciliation or selective coordination [9, 65]). LiteLib does not implement such mechanisms in relaxed mode. If an application requires stronger semantics for base-dependent operations, the recommended approach is to enable strict mode (sequence IDs), which provides a simpler and stronger end-to-end guarantee via global ordering and response equivalence.

C Evaluation Details

We include additional details and results that are omitted in the main paper due to space constraints.

C.1 Service Performance and Recovery under Failures

LevelDB Figure 19 shows the system's average latency throughout the incident, which reflects the metastable effect of the system. In the vanilla setup, the elevated latency persists until the experiment concludes one minute later. In contrast, with LiteLib, the average latency remains stable before and after the failure.

Each checkpoint (§7.2.2) takes 2.4 s to capture LevelDB's in-memory state and persist its database files to disk. During this period, no requests are processed, so service availability drops to 92%.

HDFS DataNode The per-task throughput reported by TestDFSIO remains largely unchanged after a DataNode failure, because it abstracts away concurrency effects. The failure impact instead appears as the load shift to the healthy DataNodes (§7.2.2).

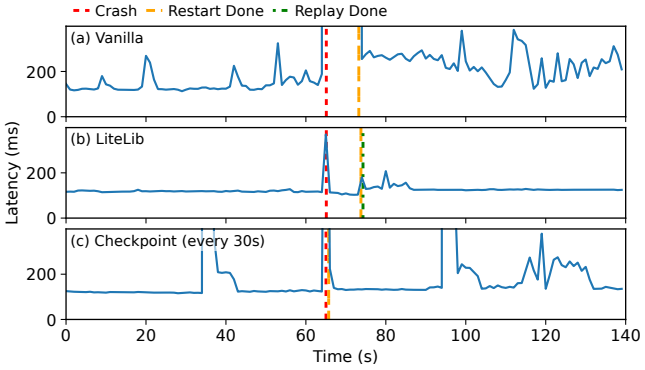


Figure 19. Impact of a LevelDB crash on latency.

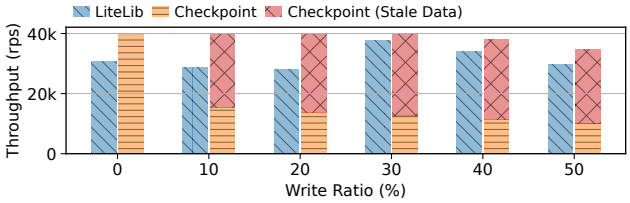


Figure 20. Throughput of LevelDB after failures with LiteLib and checkpointing under different workloads.

DeathStar In addition to the results presented in §2, we also evaluate how LiteLib performs when there is a single Memcached instance deployed without a Mcrouter in front. In this setup, the vanilla version is modified to continue serving requests from the database when the cache becomes unavailable. Upon failure, client-side latency increases by a factor of more than 570 by the end of the one-minute experiment, accompanied by a 30% drop in throughput. In contrast, with the support of LiteLib, latency increases by only 4.8%, and the throughput drop is fully eliminated.

C.2 Service Capacity under Various Workloads

Figure 20 shows the throughput of LevelDB with LiteLib and checkpointing under different write ratios. The throughput provided by the lite replica in emergency mode is 31.5K req/s on average and remains relatively stable across different workloads. This stability is because the lite replica is designed to handle both read and write requests when the full version is unavailable.

With checkpointing, 56.0% of responses are stale on average. As the write ratio increases, the throughput remains stable; however, a growing proportion of responses provided by the restored snapshot service becomes stale. Reducing the stale data requires increasing the checkpointing frequency, which, in turn, introduces even higher latency and availability overheads. This suggests that checkpointing is unsuitable for scenarios with high write ratios.

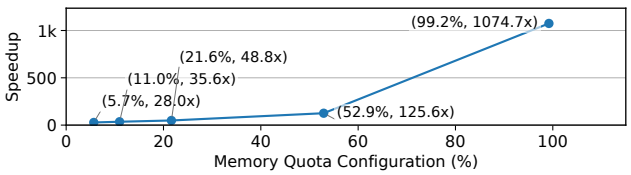


Figure 21. Memcached recovery speedup versus memory quota.

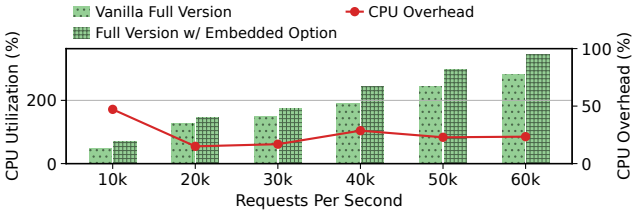


Figure 22. CPU overhead of Redis with LiteLib under different loads.

C.3 Flexible Memory Quota Configuration

LiteLib provides flexibility by allowing a configurable memory quota for the lite replica's State Store. This parameter affects the recovery speed and memory usage during normal operation.

Figure 21 illustrates the recovery speed of Memcached when LiteLib is deployed with varying memory quotas. As the quota increases, the recovery speed improves. For example, when the quota is set to 5.7% of the full version's memory usage, recovery to 90% of the pre-failure throughput is 28× faster than in the vanilla service without LiteLib. This shows that LiteLib can significantly reduce recovery time with minimal memory usage.

Similarly, when the memory quota is increased to 52.9%, recovery is 125.6× faster. This configuration is suitable when maintaining service capacity and performance is critical, and memory resource constraints are less stringent.

LiteLib offers a flexible trade-off between recovery speed and memory quota. Operators can choose the quota based on their specific requirements. Furthermore, even with a small memory quota, LiteLib achieves significant benefits, ensuring the trade-off is not a hard choice to make.

C.4 Overhead under Different Loads

We evaluate the CPU utilization overhead of LiteLib across varying throughput levels, as shown in Figure 22. The results indicate that the CPU utilization overhead of LiteLib remains around 20% across all throughput levels, except at the lowest one. These findings show that LiteLib scales with the vanilla service while keeping CPU overhead low, even under high CPU demand, so it can support high-throughput workloads.