Pilot Execution: Simulating Failure Recovery *In Situ* for Production Distributed Systems

Zhenyu Li   Angting Cai*   Chang Lou
University of Virginia   **University of California San Diego*

Abstract

Modern distributed systems rely on failure recovery to ensure availability and correctness—ironically, recovery itself often introduces severe and irreversible failures. In this paper, we first study 75 real-world recovery failures to understand common pitfalls in the recovery mechanisms. We find that the challenges primarily arise from cross-component interactions, which are difficult to expose in traditional approaches.

To address this gap, we introduce *pilot execution*, a new execution model that simulates dry-runs of recovery actions in production distributed systems to enable safe and predictable failure recovery. It enables systems and operators to observe recovery action effects before applying them, reducing the risk of cascading failures and unintended side effects.

We realize pilot execution with PILOT, an analysis framework with a runtime library that makes pilot execution easy to adopt. We evaluate PILOT on five large-scale distributed systems and show that PILOT uncovers 17 out of 20 recovery failures with modest overhead. Our use of PILOT also exposes an unknown recovery bug in the latest version of HBase.

1 Introduction

Cloud failures, even just for a brief period, could lead to substantial financial and operational losses. To react to failures, modern cloud systems employ recovery strategies such as client failover, replication, and checkpoint rollback, triggered automatically by monitoring tools or manually by operators. Unfortunately, *incorrect failure recovery* itself is known to be a major source of catastrophic incidents [47, 80, 91]. When recovery is triggered, the system is already in a vulnerable state, thus small mistakes in recovery logic can cause big issues. It is also known that recovery logic is often under-tested and error-prone, largely because of the complexity of writing and exercising tests for recovery paths [46].

Consequently, recovery actions are often *ineffective*, or even *amplify* failures instead, turning minor faults into cluster-wide disasters. Figure 1 illustrates a production incident [11] in an HBase cluster. When one region server attempts to read a write-ahead-log file, it crashes as the file was corrupted. HBase tries to recover the issue by transferring the workloads on that server to other servers, however, since the underlying file issue has not been resolved, the remaining servers also experience the same issue and this loop will eventually bring down all servers in the cluster. Such issues break the traditional fault-tolerance assumptions and cannot be simply tolerated by redundancy techniques [51].

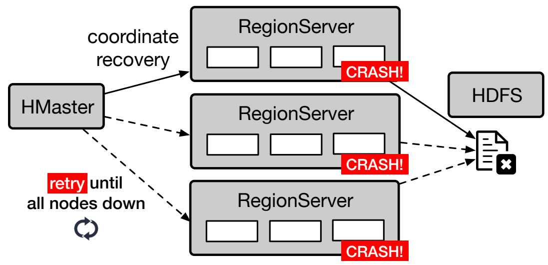

Figure 1. A production incident [11] caused by failed recovery in HBase clusters.

Real-world evidence [37, 48, 78, 84, 85] shows that recovery failures are prevalent and severe. In one incident [88], during a DDoS attack Azure tried to recover its traffic with filtering and rate limiting, but on the contrary a misconfiguration amplified the outage, disrupting Microsoft 365 services worldwide for 10 hours. In another incident [34], a brief network disruption in an AWS DynamoDB cluster triggered storage servers to renew their partition memberships, but enlarged payloads caused timeouts and triggered the recovery, which then amplified the event by repeatedly disqualifying servers and retried. It eventually overloaded the metadata service and escalating a local glitch into a region-wide outage.

Thus, there is a critical need to shift from executing inadequately validated recovery actions to *systematically validating recovery* strategies before deploying them in production. Despite extensive efforts to verify planned changes on abstract models [35, 38, 44], cloud developers still find it challenging to apply these techniques to complex, large-scale distributed systems, as abstract models usually struggle to capture low-level implementation issues.

In practice, developers often resort to rolling-style *in situ* testing on production clusters. A common strategy is *A/B testing* [52, 56, 71, 90]—along with similar approaches like canary and blue/green deployments—which applies recovery solutions to a subset of nodes first and monitors their impact before proceeding. Yet, this approach has a fundamental limitation: A/B testing does not assure that failures that manifest on one node will manifest (or be detected) similarly on other nodes, since nodes may have distinct hardware and workload characteristics. Worse, such tests can trigger cascading failures due to the absence of built-in safety mechanisms.

Is it possible to accurately and safely expose the potential issues of intended recovery actions before deploying? To understand the challenges of failure recovery in distributed systems, we first conducted a study of 75 recovery-related production incidents across eight widely deployed systems,

Property	Speculative execution	Pilot execution
Goal	Executes likely-needed work early to improve *performance*.	Executes recovery actions early to assess *safety* before applying.
Assumption	Assumes speculation is usually *correct*.	Assumes recovery may be *wrong*; the purpose is to expose bad recovery effects.
Meaning of failure	*Misprediction*.	*Desired signals* to veto or refine a recovery plan.
Interference	May *contend* for shared resources and perturb performance.	Pilot *aborts* on contention to avoid perturbing production.
Output	Produces a *candidate result* intended to become the real outcome.	Produces *evidence* (pass/fail signals, traces, metrics) for decision.
Scope	Typically *implicit* and continuous.	*Explicitly* invoked as part of the recovery workflow.

Table 1. Differences between speculative [55] and pilot execution.

including HDFS, HBase, etc. Our analysis reveals that recovery failures persist throughout the software lifecycle, often lead to severe consequences, and frequently involve complex inter-thread and inter-service interactions. Crucially, many of these failures could have been avoided with input adjustments, had their impact been evaluated ahead of time.

Therefore, we envision a *dry-run* approach for risky recovery in production recovery operations, inspired by the observation that many software tools offer a `--dry-run` option before applying important changes. For example, `Git` lets users preview changes before committing to avoid disrupting the repository. Such dry-run should reflect the consequences of the recovery action by exposing similar problems.

Our core insight is to issue a specially marked recovery request that follows the same end-to-end execution path as a real recovery (i.e., through the same handlers, queues, and service calls). As a result, the request encounters the same timeouts and exceptions that reveal these failures.

We propose *pilot execution*, a new execution model to validate the consequences of recovery actions before executing them in *production instances* (instead of testing clusters). Pilot execution conducts a dry-run of proposed recovery strategies; thus it is able to expose subtle issues deeply hidden in the implementation. Developers call pilot APIs at entries, run recovery in pilot mode, receive an end-to-end consequence report, then choose to commit, adjust inputs, or abort.

The high-level idea of executing code "ahead of time" may resemble speculative execution [55], a well-known technique in which a system predicts and executes instructions before results are needed. However, they are fundamentally different. Speculative execution is a performance optimization that assumes the future work (in our context, recovery) is correct. In contrast, our approach aims to improve reliability by executing candidate recovery actions to observe their (often faulty) consequences. Table 1 compares these two approaches.

We recognize that transparent failure recovery is extremely difficult in the general case [64]. We aim to provide a safe way for operators to preview consequences of some severe types of recovery failures, such as propagation failures [87].

We build PILOT, an analysis framework with a runtime library that makes pilot execution easy to adopt. Designing PILOT involves three main challenges. First, pilot execution should not interfere with original execution. However, it should still provide good observability. Instead of completely disengaging the whole environment by forking the process, we isolate pilot execution by spawning and mirroring execution on a group of *phantom threads* (ephemeral, in-process execution units). Phantom threads strike the middle ground: they remain inside the address space to share code and state while lightweight and easy to control.

Second, recovery logic spans threads and processes. When the pilot execution crosses a boundary (*e.g.*, an RPC), it must stay in pilot mode and new phantom threads must be tracked. Inspired by distributed tracing [66, 67], we design a lightweight *context propagation* mechanism: the runtime captures a small pilot context at the source, carries it across handoffs, and reactivates pilot execution at the destination by spawning a new phantom thread.

Additionally, recovery should be prompt to reduce system downtime. If a pilot execution takes too long, this amplifies the failure damage by extending the unavailability window. We reduce recovery delays by warming up the system during pilot execution, enabling a rapid transition to fast replay during actual recovery. We cache intermediate results from time-intensive operations to mitigate delays.

We have applied PILOT to five production-grade systems (Solr, HDFS, Cassandra, HBase and YARN), successfully uncovering 17 out of 20 severe real-world recovery failures, many of which prior testing methods failed to expose. PILOT adds 17.63% delay to recoveries over 10 seconds and only a few seconds to shorter ones. It also exposes an unknown recovery bug in the latest version of HBase.

Our main contributions include:

- We conduct a large-scale study of recovery failures that identifies common patterns, consequences, and root causes.
- We design pilot execution, a new model which enables dry-run simulation of recovery actions.
- We implement PILOT as an analysis framework with a runtime library, and evaluate it on real systems. Our evaluation demonstrates its effectiveness in exposing recovery failures with low overhead. PILOT is open sourced at https://github.com/LiftLab-UVA/PilotExecution.

2 Understanding Recovery Failures

Prior work has focused on particular patterns of recovery failures, including metastable failures [50] and vicious cycles [80, 86]. In this section, we take a broader view of recovery failures and distill key challenges to inspire solutions.

Software	Lang.	Category	Version	Date	Sampled
HDFS	Java	File Sys.	2.6.1-3.4.0	2014-2024	10
HBase	Java	Database	0.9.0-3.0.0	2010-2025	15
YARN	Java	Resource Mgr.	2.3.0-2.9.2	2014-2025	5
TiDB	Go	Database	1.2.0-1.12.0	2019-2025	5
Kafka	Scala	Streaming	1.0.0-3.3.0	2017-2024	10
Cassandra	Java	Database	0.8.0-5.0.0	2011-2024	15
RabbitMQ	Erlang	Message Bkr.	3.6.0-3.13.6	2016-2024	5
Mesos	C++	Resource Mgr.	0.21.0-1.1.2	2015-2022	10

Table 2. Sampled failure datasets from eight studied systems.

Methodology We analyze eight widely deployed, large-scale software systems spanning different functionalities and programming languages (Table 2). To gather relevant cases, we first extracted critical-priority reports from the systems' official issue trackers, specifically targeting incidents encountered in production environments. We then filter these reports using keywords like recovery, failover, retry, and related terms. This process resulted in a total of 75 recovery failures.

2.1 Findings

Finding 1: *Recovery failures persist throughout software lifespan.*

Some bugs take years to expose in production, and many others are newly introduced as the software evolves. Our analysis shows that 38% of recovery issues occurred since 2019, showing that correct recovery remains a challenge even for well-established systems.

Finding 2: *Both automatic and manual recovery mechanisms are prone to failures.*

More than half (61%) of failures happen in automatic recovery mechanisms. Within automatic recovery, failures in *state recovery* are the most common (25%), which reconstruct in-memory state from logs, snapshots, or replicas. *Primary-backup failover* (16%) also frequently encounters issues when promoting standby nodes to active status upon failures. The remaining automatic failures happen in *exception handling* (13%) and *consensus-based recovery* (7%). 39% of the studied failures happen in *manual recovery*, e.g., Cassandra nodetool commands [6] which repair data inconsistencies.

Finding 3: *Recovery failures lead to severe consequences.*

Besides *service unavailability* (41%), recovery failures also lead to: *Resource exhaustion* (14%), where recovery overloads system components, e.g., an incorrect error handler triggers a replication storm that overwhelms the server [14]; *Data loss* (13%), where incomplete recovery leaves the system in an undefined state; *Partial failures* (12%), where systems indefinitely hang when recovery fails halfway; *Performance degradation* (11%), where recovery generates enormous or uneven overhead; and *Data inconsistency* (9%) where recovery results in incorrect system state, e.g., when stale memstore snapshots are flushed with incorrect sequence IDs, generating HFiles that misrepresent the actual content [8].

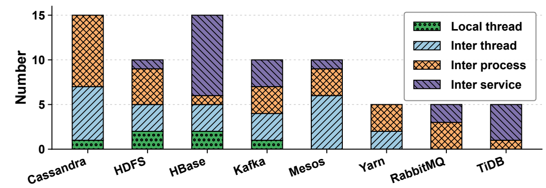

Figure 2. Failure scope distribution (Finding 5).

Finding 4: *The root causes of recovery failures are diverse.*

Error handling deficiencies are known to be error-prone and account for 28% of the studied cases. We found that state management issues (improper synchronization or handling of recovery state) turn out to be the most common root cause (37%). Another common cause is inappropriate resource lifecycle management (13%), i.e., incomplete cleanup of system resources. We also observed recovery exhausting system resources (9%) such as too frequent retries and other implementation flaws (6%) leading to failed recovery.

Finding 5: *Most recovery failures (92%) manifest beyond local thread scope.*

We categorize propagation into three types shown in Figure 2. *Inter-thread* (30%) issues involve coordination between threads within the same process. HDFS-16115 [16] illustrates this case, where a faulty error handler in one thread caused command pileup in another thread, leading to zombie operations. *Inter-process* (35%) issues affect coordination across processes in distributed system; in KAFKA-10832 [21], improper recovery handling by a broker corrupted the producer node. *Inter-service* (27%) issues span multiple services as in TiDB-963 [28], where misinterpreted failures led Kubernetes to incorrectly scale TiKV instances.

Finding 6: *63% of recovery failures are avoidable with minor tweaks to original recovery.*

Surprisingly, we found that many recovery failures could have been prevented without complex strategies. Tolerating non-critical errors during recovery (19%) would allow the recovery to complete successfully. Some recovery failures only manifest under specific event orderings, and developers may manually enforce certain sequences (12%). In one incident [24], RabbitMQ's failover recovery would fail if health checks ran during master node promotion. The developers fixed this by manually enforcing health checks after the master election was complete. Rebalancing recovery workloads across the system (18%) can help mitigate recovery induced overload. Finally, changing recovery-related configurations or parameters (14%) also helps.

2.2 Implication

While simulating recovery actions in situ may appear ambitious, our study findings highlight that it is both necessary and feasible. The persistence and severity of recovery failures (Findings 1-3) highlight that such issues are widespread and pose critical challenges requiring immediate attention. However, standard testing environments often miss these issues

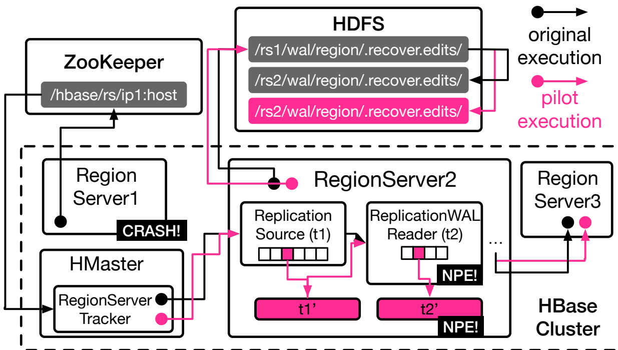

Figure 3. Overview of pilot execution in HBase-25898.

Pilot APIs
`RunId start(EntryPoint r, Policy pi)`
start a pilot run at entry point `r` under policy `pi` (e.g., time budget, stop conditions).
`Status observe(RunId runId, Consumer<Observation> sink)`
stream observations (events, exceptions) to `sink`.
`Status waitFinishOrAbort(RunId runId)`
block until the pilot execution finishes or aborts; return the terminal `Status`.
`Status abort(RunId runId)`
force to abort a pilot run; return the terminal `Status`.
`Summary report(RunId runId)`
retrieve a summary (path coverage and critical events) for the run.

Table 3. PILOT APIs.

due to their diverse root causes (Finding 4). The opportunity for better recovery strategies (Finding 6) underscores the need for a technique that allows systems to safely evaluate the consequences of recovery actions before applying them. Finally, recovery actions frequently involve cross-thread and cross-process interactions (Finding 5). This insight requires the solution to simulate complex recovery behavior across system boundaries.

3 Simulating Recovery with Pilot Execution

Design Goals Our goal is to provide a safe and accurate way to predict the effects of recovery actions before committing them in production. Drawing from our failure study (§2), we target risky recovery logic in distributed systems, where danger arises from cross-component interactions.

We design PILOT to satisfy four properties: (1) *Fidelity*: A PILOT run should traverse the same code paths as a real recovery so that it triggers the same issues; (2) *Safety*: PILOT should not perturb production execution or leave side effects; (3) *Observability*: PILOT should surface all relevant outcomes such as error signals, resource usage; and (4) *Performance*: PILOT should not significantly delay the actual recovery or incur heavy runtime overhead.

Problem Statement We consider a production distributed system S composed of many interacting components (e.g., processes, services), equipped with a set of recovery procedures $\mathcal{R}$. During an incident, operators or the system select a recovery procedure $R \in \mathcal{R}$ to apply to the live system. We define a *recovery failure* to be the case where executing R does not restore S to a healthy state, but instead causes safety or liveness violations. Our goal is to exercise the same recovery logic in an isolated pilot execution, using observable signals (e.g., logs and metrics) to predict the outcome without perturbing the production system and to aid operators in choosing safe recovery actions.

Simulating dry-runs for failure recovery is challenging in the general sense. PILOT targets production distributed systems whose recovery logic spans multiple components and is traceable through instrumentable surfaces. PILOT is designed to handle recovery failures that propagate or amplify faults, but is less effective for purely local bugs, long-term accumulative issues across many recovery rounds, or semantic violations that are not covered by built-in checkers.

Workflow In Figure 3, we illustrate the workflow on the motivating HBase example. A pilot run is a single instance of pilot execution parameterized by an entry point r_i and a policy (time budget, limits, stop conditions). Conceptually, a pilot run proceeds as follows:

- starts at r_i under the policy;
- mirrors the recovery path on a set of concurrent *phantom threads* $(t'_1, t'_2, \dots)$, without blocking or mutating production threads or states;
- propagates the execution across threads and processes so downstream work also executes in pilot mode;
- isolates side effects via state shadowing and proxy layer so writes are visible only to pilot participants;
- observes exceptions, timeouts, resource curves, and semantic-checker violations in a unified event stream;
- terminates and reclaims shadow resources once it finishes.

Below, we detail how pilot execution looks like in the motivating HBase incident. When the system encounters the error signal of RS1 crash, a special handler intercepts the exception before it can impact live servers. This triggers a new round of pilot execution initiated from the HMaster, which sends an RPC to RS2. The request is encoded with metadata identifying it as part of a pilot execution. Upon receiving the request, RS2's handler detects the metadata and diverts the execution into the pilot mode, where all operations are executed by a dedicated group of phantom threads. These phantom threads form a sandbox where state modifications and I/O operations remain invisible to production yet visible among pilot participants, isolating side effects from the production environment. For example, any updates to HDFS are redirected to a parallel shadow copy that is only visible to components running in pilot mode. The recovery logic proceeds as it would in a real run, triggering the same failure conditions. The system observes the failure signal and uses this insight to adjust the mitigation strategy (e.g., deleting the problematic log file after the simulation completes).

```
public class Repair extends NodeToolCmd {
+ @Arg boolean PilotMode;
  @Arg String keySpace, range, tableName;

  public void execute(…){
+   if (PilotMode) {
+    RunId id = Pilot.start(this::execute$pilot, policy);
+    if (Pilot.waitFinishOrAbort(id) == FAILED){
+       Pilot.report(id);
+       return;
+    }
+   }
    repair(keySpace, tableName, range);
    …
  }

+ public void execute$pilot(…){
+ …
+ }
}
```

Figure 4. Codes of enabling PILOT in Cassandra nodetool repair.

4 Pilot Execution Design

4.1 Interface

Table 3 summarizes the PILOT API. The PILOT runtime library is responsible for creating and managing all pilot runs throughout their lifecycle. We use the code example in Figure 4 to walk through the usage. This example shows a CLI-based entry point: we instrument the command handler `execute` and add a `-pilot` flag (annotated via the CLI framework). When an operator runs `nodetool repair -pilot -arg1 -arg2 ...`, the instrumented entry point invokes `start` to initiate a new pilot run. The runtime assigns a globally unique ID (to identify different rounds of pilot runs) and records it in the status registry as `/pilot-runs/<id>` as well as the related resources such as executing thread ID.

The caller thread then blocks on `waitFinishOrAbort`, which is fine as the actual job will be delegated. This blocking call avoids two concurrent recovery (original and pilot) interfere with each other. It continuously monitors the resource allocation status to detect pilot execution completion. This function waits until all execution units are removed from `/pilot-runs/<id>`. Meanwhile, developers have access to events and errors during pilot execution via `observe`.

After the pilot run finishes, PILOT outputs a summary which includes exceptions, time durations and resource usage via `report` call. Operators decide how to proceed: if the pilot succeeds, the recovery proceeds to commit; if it fails, they can make adjustments and launch additional pilot runs until finding a safe recovery strategy (e.g., in Cassandra cases [2, 3], changing `-keyspace`, `-table`, or `-range` avoided cascading failures and out-of-memory errors).

4.2 Managing Pilot via Phantom Threads

The goal of a pilot run is to rehearse recovery paths without interfering with other worker threads. Running the recovery logic on the original thread would block or perturb normal service, and make rollback hard; using process fork/containers is heavyweight and inconvenient to observe system states.

To address these limitations, we realize pilot execution by spawning and mirroring execution on a group of phantom threads, which are ephemeral, in-process execution units that

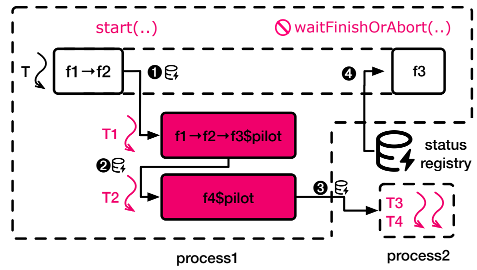

Figure 5. Lifecycle of phantom threads.

rehearse a prospective recovery path without changing the behavior of production threads. Phantom threads strike the middle ground: they remain inside the address space to share code and state (in a read-only way) while being lightweight and easy to control side effects.

4.2.1 Lifecycles of Phantom Threads

Assuming the current round pilot run is assigned ID 1 and registered under `/pilot-runs/1`, we show the complete lifecycle of a phantom thread in Figure 5. Phantom threads are created on demand. A typical example occurs when a new pilot run is activated via a `start` call (❶). The `start` API initializes the phantom thread by setting its executable code to a transformed version of the original function, which we denote using a `$pilot` suffix (e.g., `execute$pilot`). This transformed version is instrumented to ensure isolation and observability. Once initialized, the newly created phantom thread registers its thread ID with the status registry. Phantom threads can also be created when pilot execution propagates across thread (❷) and process (❸) boundaries (§4.3).

Most phantom threads are created from scratch at well-defined entry points (e.g., execute$pilot() or RPC handlers). However, when pilot execution affects an original thread blocked at a synchronization point, PILOT runtime initiates a new phantom thread and "fast forwards" it to match the original thread's state, ensuring the simulation accurately reflects this influence on the original thread. This is achieved by inserting lightweight hooks to record call stacks and local variables. The new phantom thread then uses these recordings to traverse the call stack, restoring local variables and jumping to call sites in each stack frame without replaying code, until reaching the affected site where it continues execution independently. We discuss more details on how the original thread's checkpoint is taken and restored in Appendix B.1.

Inside phantom threads, the execution runs a transformed version of functions that differ from those in the original thread. This is because a pilot run is designed to be a simulation; thus, we should not directly use original functions, especially those that cause dangerous effects (e.g., halt). PILOT runs the modified version to isolate their effects, which we discuss in §4.4. Phantom threads are generally safe from a resource-consumption perspective. They reuse existing code paths and maintain only a small amount of shadow state, so their CPU and memory overhead is modest compared to normal recovery traffic.

At the end of execution, phantom threads de-register their bookkeeping information. PILOT instruments phantom threads to remove their associated IDs from `/pilot-runs/1` in the status registry when phantom threads exit and update their status. The original thread resumes execution after the pilot run finishes. PILOT runtime makes the decision to revert the execution back. It periodically checks to align the status data in the status registry with current execution thread states; once it finds all phantom threads under `/pilot-runs/1` have been finished and removed, it notifies the blocking `waitFinishOrAbort` to resume normal execution (❹).

Phantom threads can also potentially run indefinitely without exiting, which wastes system resources (and more importantly, block normal recovery). To terminate phantom threads, we make phantom threads' liveness bidirectionally bound to registration data. Developers may force a pilot run to abort by invoking `abort` that deletes thread registration data in the status registry, triggering callbacks on all nodes to clean up the corresponding threads tracked in PILOT runtime library. Additionally, each pilot run has a maximum time budget. A timeout mechanism inside `waitFinishOrAbort` recursively cleans up the root node and its child nodes in storage associated with a pilot run when the timer fires.

4.2.2 Handling Synchronization

As phantom threads run concurrently with original threads, race conditions may occur: an original thread t and a phantom thread t' may simultaneously require the same lock L. If we allow t' to acquire the lock, it blocks the normal execution of t thus violating our safety guarantee. Meanwhile, completely ignoring the locking semantics produces inaccurate pilot results as some recovery operations need to wait until certain conditions.

To ensure accurate pilot runs without interfering with live execution, we introduce a speculative locking mechanism. The core principle is that phantom threads acquire locks "speculatively" — they always yield to original threads on conflict, immediately aborting pilot execution when contention is detected. This guarantees non-interference with production execution, at the cost of incompleteness.

Here is how speculative locks deal with three representative scenarios: (1) when only original/phantom threads involved: synchronization primitives behave same as originals; (2) when phantom held the lock and original acquires: the phantom thread releases the lock and pilot execution terminates immediately; (3) when original held the lock, phantom acquires: if an original thread t_1 holds the lock, a phantom thread t_2 waits until t_1 releases it. This wait does not block other original threads. Since phantom threads always yield to original threads, high contention may cause the pilot run to time out while waiting for the original thread to release the lock (a potential false positive). We accept this trade-off to prioritize non-interference with production. We described the detailed algorithm in Appendix B.2.

4.2.3 Fault Tolerance

PILOT runtime may include bugs and result in faults. Here we refer to those from PILOT itself instead of faults in execution (will discuss in §4.4). We consider two types of faults to handle.

First, phantom threads may spawn indefinitely if there exists a feedback loop. For example, if f_1 running in phantom thread T_1 spawns a new phantom thread T_2 to execute f_2, and this chain eventually attempts to re-invoke f_1 (forming a cycle $T_{1,f_1} \rightarrow T_{2,f_2} \rightarrow \cdots \rightarrow T_{n,f_1}$). To prevent this, PILOT detects cycles where pilot execution attempts to re-invoke an ancestral function that spawns phantom threads. Once PILOT identifies the circular dependency, it halts the spawn process. Additionally, there is an upper bound for the maximum number of phantom threads running in the system to prevent a pilot run from exhausting system resources. Since the propagation chain topology is typically shallow in practice, this detection incurs negligible overhead.

Second, PILOT ensures fault tolerance in resource reclamation by utilizing a distributed status registry to coordinate cleanup of phantom threads and their associated resources. The correctness of this registered data is critical, as it serves as the trigger for cleanup callbacks. PILOT implements the registry on top of Time-to-Live (TTL) node [22] feature of ZooKeeper [1]. These TTL nodes are tied to the liveness of the runtime's session via a heartbeat; if the PILOT runtime faults or the node fails, the heartbeat ceases, and ZooKeeper automatically expires the corresponding TTL nodes after a specified timeout.

4.3 Propagating Pilot Execution

Recovery logic often spans across multiple threads and processes. When pilot execution crosses such boundaries (e.g., making an RPC call), the system ensures that the execution continues under phantom threads and keeps track of these threads for proper management.

Inspired by prior distributed tracing works [66, 67], we design a *context-propagation* mechanism to track pilot execution. The core idea is to maintain lightweight metadata that travels with any execution, whether to another thread, a different process, or an external service. While the basic idea of "attaching context to a thread" is not new, the non-obvious part is doing this transparently in systems that heavily use thread pools and asynchronous callbacks. We need a lightweight, user-level execution mode and metadata (e.g., whether the current execution is pilot or live, which pilot run it belongs to, and where to redirect side effects) that can be queried by all instrumentation points.

Our runtime automatically captures the context at the source, propagates it through the handoff boundary (e.g., wrapping APIs, injecting headers), and activates pilot execution on the destination side by creating a phantom thread.

PILOT runtime initializes a lightweight `PilotContext` when a pilot run begins and attaches it to the originating phantom

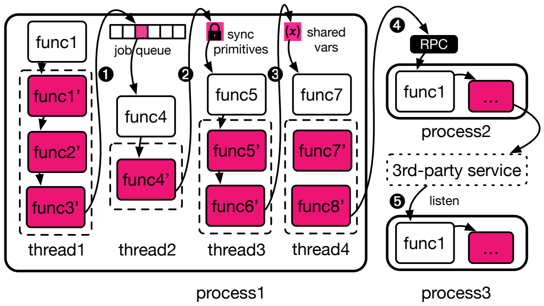

Figure 6. Propagating across thread/process boundaries.

thread as thread-local metadata. This metadata is intentionally compact so it can be propagated cheaply, including: (1) a globally unique pilot ID identifying the execution round, enabling any thread to determine execution mode—a non-null pilot ID indicates phantom thread execution, triggering redirection to pilot-version functions, while null indicates original thread execution; (2) span metadata for tracing pilot execution—each span contains fields: `spanID` (unique identifier for current execution segment), `parentSpanID` (linking to the parent span), and `timestamp` (for timeout management).

The propagation of `PilotContext` serves two key purposes. First, it enables global pilot mode awareness: when the context reaches a destination, PILOT runtime examines the incoming metadata and, if a valid `PilotContext` is present, it (i) recognizes that the request originates from a pilot execution, (ii) activates pilot mode by creating or reusing a phantom thread and binding the context to it, then executes pilot-version code paths rather than production logic, and (iii) reports bookkeeping information for phantom thread management as described in §4.2.1. Second, it enables distributed tracing of the recovery path: whenever propagation crosses thread or process boundaries, a new span is created with a fresh `spanID` while inheriting the sender's `spanID` as its `parentSpanID`, and reports provenance to the runtime.

We support both explicit (RPC, executor tasks) and implicit (shared variables, synchronization) handoffs (Figure 6). Implementation details for the five representative patterns are in Appendix B.3.

4.4 Isolating Induced Side-Effects

Pilot execution runs recovery logic in situ; without careful isolation, its side effects could leak into production state or third-party services and, conversely, over-isolation could make the pilot result unfaithful. In this section we discuss the isolation problem and our solution around two domains: program states and I/O.

4.4.1 Shadowing System States Pilot execution should leave no persistent changes to critical system state. One popular approach is encapsulating changes in the transactions and rolling back when needed. Transactional rollback presumes updates are cleanly encapsulated as transactions; in many distributed systems, state changes are spread across ad-hoc data

```
1   public class NodeFailoverWorker {
2       public Server server;
3 +     public Server server$pilot;
4       public void failover {
5           //...
6       }
7 +     public void failover$pilot{
8           try{
9               //...
10 -            transferQueue(server);
11 -            server = getFailoverTarget();
12 +            server$pilot = PilotUtil.copy(server);
13 +            transferQueue$pilot(server$pilot);
14 +            server$pilot = getFailoverTarget$pilot();
15              //...
16 -            Files.write(path, content);
17 +            Files$wrapper.write(path, content);
18              //...
19 -            ZookeeperClient.deleteNode(path);
20 +            ZooKeeperClient$wrapper.deleteNode(path);
21          }catch(IOException e){
22 -            Runtime.halt();
23 +            PilotUtil.report(e);
24          }
25      }
26  }
```

Figure 7. Codes of enabling PILOT in HBase node failover.

structures and services, so there is no uniform transactional boundary to revert reliably.

Instead, PILOT isolates recovery side effects by logically duplicating state, but doing so naively is impractical. Duplicating state via full copies or snapshots would roughly double memory and copy costs, which is unacceptable in production. We also cannot rely on OS page-level copy-on-write or simply clone an entire process or cluster: (i) the state is too large, (ii) recovery spans multiple components, and (iii) we must still run the recovery code in-place on the live system state. At the same time, recovery actions typically touch only a small fraction of the overall state, so fully cloning the system would be wasteful.

Consequently, our design adopts a selective copying approach with lazy materialization. Phantom threads reuse the live state and re-execute the same recovery code with side effects redirected into shadow state. This is enough to surface the main classes of recovery failures we target. PILOT augments each field with a corresponding shadow field and instruments the pilot functions so that when they first access a field, they perform a shallow copy—creating a reference to the shadow field rather than duplicating the entire object graph. All writes are then redirected recursively through nested hierarchies to their shadow counterparts, providing complete isolation without the overhead of deep copying.

Figure 7 illustrates how we instrument pilot version functions to achieve this: the pilot version first creates a shallow copy `server$pilot` (line 12), and subsequent accesses operate on this shallow copy (lines 13–14). For classes that cannot be instrumented, we fall back to deep copying. We find this design particularly useful when applied to large data structures such as `HRegionServer` (commonly used in HBase). This keeps memory overhead low and prevents any writes to production state.

4.4.2 Redirecting I/O Pilot execution must be hermetic: its effects stay within the simulated environment and never reach the external environment (local file systems, distributed stores, or third-party services). We require *scoped isolation*: pilot side effects are invisible to production yet fully visible within the pilot run. All pilot I/O is transparently redirected to pilot-scoped namespaces: for example, a file write by one pilot thread is seen by other pilot threads while the original thread continues to observe the unmodified file.

To isolate file system side effects during pilot execution, PILOT intercepts file operations and redirects them to a dedicated shadow directory. As shown in Figure 7, file operations in pilot functions are replaced with calls to `Files$wrapper` (lines 16–17), which employs a lazy copying strategy rather than eagerly duplicating original files: it initially replicates only the directory structure and metadata, deferring actual data copying until necessary. This optimization targets scenarios where recovery does not exhibit read-after-write behavior. When writing to an existing file, the wrapper avoids making a full copy and instead appends changes to a separate log. Only if the file is subsequently read during pilot execution does the wrapper reconstruct it by replaying the log over the original content. For files that are only written but never read back, this strategy significantly reduces I/O overhead. Newly created files during pilot execution are written directly into the shadow environment. Reads simply access the latest content in the shadow directory.

For third-party libraries, we apply a similar approach by redirecting all operations to isolated namespaces under a pilot-specific prefix (i.e., `/pilot/...`) that are only visible within pilot execution. We abstract the client-side operations of those third-party services into read/write operations and intercept the API calls (lines 19–20 of Figure 7). We also apply the lazy copy optimization described above in I/O isolation to further reduce the overhead. Supporting more complicated operation semantics remains a part of future work.

4.5 Observing and Reacting to Failures

PILOT runtime embeds monitoring components in pilot-version functions to observe and report recovery failures. For example, dangerous APIs that could terminate the system when a fatal exception is triggered are replaced with safe reporting APIs (lines 22–23 of Figure 7), allowing PILOT to capture failure signals without actual harm. Beyond explicit errors, PILOT tracks recovery duration via timers, monitors resource consumption (memory and I/O usage), and checks system health metrics. We plan to further add support from existing runtime checking tools [42, 62, 63] as future work.

Using context propagation, PILOT records each pilot-scoped function/RPC as a span and links spans by caller–callee to form a cross-thread/process execution tree (Figure 8) in the style of distributed tracing [66, 67]. Each span has a unique ID, its parent pointer, and minimal metadata stored in and carried by the `PilotContext`. Runtime monitors mark the active span when they observe anomalies (e.g., exceptions, timeouts,

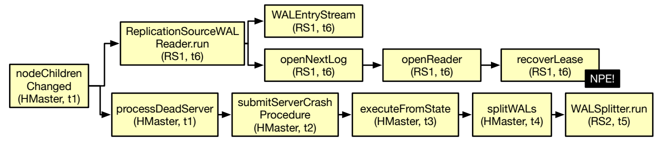

Figure 8. Execution tree.

or resource-limit violations). Upon a failure, the PILOT runtime materializes the tree and the precise path from the pilot entry point to the error, capturing involved components and how effects propagated.

Based on the feedback provided (monitoring metrics, anomalies, and traces), if the pilot run fails, the operator can iteratively start a new pilot run with adjustments to recovery command parameters (e.g., throttling rates or recovery range), environments (e.g., fixing buggy dependent services), or operation sequences (e.g., reordering recovery steps to avoid problematic interactions) to verify the updated recovery strategy. If the pilot run succeeds, operators can commit the recovery to the production environment.

PILOT's feedback is not always actionable: for example, when the problematic behavior is hard-coded in the recovery logic rather than configurable (e.g., a client library always retries failed RPCs with a fixed backoff). In such cases, fixing the issue requires code changes, not simply choosing a different configuration or recovery workflow.

4.6 Enabling Lightweight Pilot Execution

Recovery still needs to be fast enough to keep downtime small, but naively skipping work during pilot runs would hurt fidelity. Instead, PILOT lets pilot execution "warm up" the system by pre-computing the expensive yet deterministic parts of recovery so that the real run can simply reuse them. In practice, most of the delay comes from transferring large amounts of state (for example, log segments or index files) and from pure computations such as placement planning or Merkle-tree comparison. During pilot execution, we store transferred data in isolated directories and memoize the outputs of side-effect-free functions keyed by their inputs. When operators later commit recovery, the system validates that the relevant inputs have not changed (e.g., by re-checking checksums) and, if so, directly reuses the cached data and results. Our contribution here is not a new caching primitive, but how pilot execution systematically prepares recovery with intermediate state.

4.7 Code Transformation Support

Supporting pilot execution for existing systems requires transforming and instrumenting system codes— an error-prone process. We provide a static analysis framework to automate the process and ease developers' effort.

Figure 9 illustrates the overall workflow: at the offline phase, PILOT identifies recovery entries to insert pilot APIs. Recovery entry points fall into two categories: passive and proactive. Passive handlers include error-handling constructs (e.g., try-catch blocks) and listeners for third-party services. Proactive handlers are typically implemented as CLI commands, such as Cassandra's `nodetool repair`. Both of them

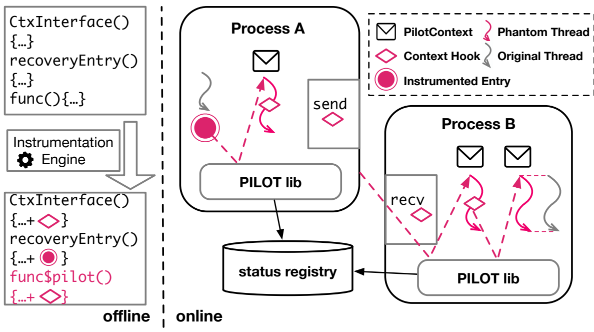

Figure 9. Offline and online workflow of PILOT.

have a clear pattern. PILOT further filters recovery entries with trivial handling logic (e.g., just logging the error message).

To support propagation, the tool injects hooks to carry PilotContext and activate phantom threads, while building lineage links (spans) to reconstruct cross-thread/process execution trees at runtime, leveraging standard telemetry substrates.

For isolation, the tool produces pilot-version functions f' for application code and redirects eligible calls to f' under pilot mode. Pilot versions embed the shims needed for propagation and isolation while keeping business logic unchanged. For state and I/O, the tool replaces unsafe operations with wrapper calls that go to shadow state and pilot-scoped namespaces; the wrappers are invoked only in f' paths.

At runtime, pilot execution starts from the instrumented recovery entries, and the injected hooks propagate `PilotContext` and activate phantom threads managed by the PILOT runtime library to execute pilot-version functions.

5 Implementation

We implemented PILOT with 13,500 SLOC. Our static analysis/instrumentation uses Soot [89] as the backend. Our context propagation is implemented on OpenTelemetry [23]. While our current implementation is based on Java, our techniques do not rely on language-specific features and can be conveniently ported to systems based on different languages. The tool includes a single script which automates the static transformation workflow and prepares the program package.

6 Evaluation

In this section, we answer several questions: (1) is pilot execution applicable for real-world distributed systems? (2) can PILOT detect and prevent recovery failures before they happen? (3) Does PILOT significantly delay real recovery? (4) how accurate is PILOT? (5) how large are the runtime overheads? The experiments are done on servers with 8 cores 2.0 GHz CPUs, 64 GB memory, running Ubuntu 22.04.

6.1 Manual Effort of Using PILOT

Most analysis and instrumentation by PILOT is automatic, though two types of manual effort are required. First, the operator needs to provide a configuration file that specifies metadata such as source file paths and instrumentation scope.

	SL	HF	CS	HB	YN
LOC (K)	650	550	435	1220	442
Entry Points	26	12	34	60	24
Context Hooks	794	463	323	1191	313

Table 4. Overview of evaluated systems, including size (LOC in thousands) and number of identified instrumentation points. *SL*: Solr; *HF*: HDFS; *CS*: Cassandra; *HB*: HBase; *YN*: YARN.

Second, custom communication protocols will require minor modifications to include context metadata in the message headers to support context propagation. These manual changes are 41 lines of code in Cassandra, 89 in HBase, and 50 in Hadoop (HDFS and YARN share the same RPC framework). Solr uses a standard HTTP library for communication so there is no need for manual modification.

6.2 Applying PILOT to Real-World Systems

To evaluate the practicality of our approach, we applied PILOT to five real-world distributed systems: Solr, HDFS, Cassandra, HBase and YARN. These systems are widely deployed in production and are known to suffer from subtle recovery-related failures. We selected these systems to represent diverse architectures (e.g., master-worker and peer-to-peer) and recovery patterns. As the result, our tool successfully enabled pilot execution for all five systems. In Table 4, we show how large the codebase of the system is, and the number of recovery entry points instrumented to enable pilot execution.

6.3 Runtime Detection and Recovery

Methodology PILOT is designed to be a simulation tool to expose recovery failures before happening. The key metric for evaluating it is once the recovery action is triggered in a running system and will cause a failure, whether and how quickly can the tool detect the failure. This contrasts with bug-detection tools (*e.g.*, fuzzing), which are designed to induce bugs by generating triggering conditions like crafted inputs or specific thread interleavings.

Failure Benchmark We constructed our evaluation dataset by reproducing twenty **real-world** recovery failures across five systems. Table 7 in the appendix shows the list of evaluated cases. Our selection process began by querying each system's issue tracker with keywords such as "recovery failure", "error handler", "retry", "failover", and "mitigation failure". This search returned 9,913 issues in total (Solr: 1,899; HDFS: 1,837; Cassandra: 1,309; HBase: 3,665; YARN: 1,203). From these, we randomly sampled 500 issues for closer inspection and examined their descriptions to confirm whether they reflected genuine recovery failures. We excluded cases that were low priority, not triggered by recovery, or lacked complex interactions, yielding 35 candidate failures. We attempted to reproduce each and successfully recreated 20, while the remaining 15 lacked sufficient detail for reproduction. These 20 cases form our evaluation dataset. All of these failures led to severe consequences. Each case took one week on average to reproduce. Seventy-five percent of cases were **not** included in our motivational study.

6.3.1 Detecting Recovery Failures

Baselines Many prior works [60, 90] are network-layer systems that target failures in the datacenter fabric instead of the application-level recovery logic. We compare PILOT against four baselines that span the main families of runtime validation that operators likely use for our target systems:

- We evaluate *built-in dry-run* when available (e.g., Cassandra's repair previews inconsistent ranges before fixing). For systems without such support, we implement ad hoc dry-runs that output recovery plans without execution.
- We implement *A/B testing* by limiting recovery actions to a subset of nodes (e.g., single range mode in Cassandra).
- We evaluate failure detectors proposed by [80] which log system events and correlate repeated requests with errors to find *vicious cycles*, a common cause of recovery failures.
- We implement *validation slice* approach based on a prior OSDI work [71], which creates a validation environment by extending online services with new nodes and validates workloads on these nodes before migrating to the whole cluster. This is different from A/B testing as the validation slice is isolated from the production cluster.

Results The results are shown in Table 5. PILOT successfully detected seventeen out of twenty evaluated issues. In contrast, baseline approaches, even all four combined only detect nine failures. The built-in preview can reveal issues in the planning phase but fails to expose issues in the execution phase. A/B testing is non-deterministic and highly depends on how the nodes of test set are selected. For example, for CS3 and CS4, the timeout-related issues could only be triggered if the specific problematic node is included in the selected test group. Failed A/B tests can corrupt the test node or trigger cascading failures, as in HB3 where the faulty node's recovery spread and caused others to fail. The vicious cycle detector detects retrying issues but does not address other types. Validation slice emulates recovery only on a small subset, thus missing most bugs induced by complex interactions.

PILOT fails to detect three of twenty recovery failures. In SL3, recovery is triggered multiple times and repeatedly spawns threads that accumulate until causing stack overflow after creating forty thousand ZooKeeper nodes—PILOT only validates a single round recovery and cannot capture this accumulating effect that manifests progressively after multiple rounds. HF3 demonstrates a failure of omission where the ClosedChannelException is silently swallowed, leaving volumes partially initialized, but since the recovery's bug lies in what error handler doesn't do rather than what it does, PILOT cannot detect this issue. HB2 exhibits a subtle semantic violation where regions must clear states after failed log replay—an implicit invariant that our checker did not cover.

6.3.2 Recovery Latency

Setup Validating should not significantly delay the real recovery. To measure the delay brought by pilot execution, we evaluated recovery time for each issue under five scenarios. We first evaluate the recovery time in the buggy version, which is defined as the time from the start of recovery until a recovery failure manifests without (❶) and with pilot execution (❷). This shows how fast pilot execution exposes the issue compared to directly performing recovery. We then measure, in the correct version in which the issue in the recovery code has been fixed, whether pilot execution will incur significant delay by comparing recovery time without pilot execution (❸), with pilot execution (❹), as well as the total time of the pilot run followed by the subsequent real recovery once determined as safe (❺).

Results Table 6 summarizes the evaluation results. Under the execution with buggy version, pilot execution takes a similar amount of time compared to the original recovery, incurring 3.99% latency increase on average (median: 2.12%, σ = 4.56%), caused by instrumentation. Under the execution with correct version, we found that the combination of base and pilot execution incurs additional latency ranging from seconds to dozens of seconds. Thanks to our caching optimization mechanism, the actual recovery after pilot execution is significantly accelerated. The total time is far less than the sum of the two individual phases and only slightly slower than normal recovery. The overhead averages 17.63% for normal recoveries longer than 10 seconds (median: 15.50%, σ = 6.87%), and remains within a few seconds for shorter recoveries. We consider the overhead a worthwhile tradeoff for the effectiveness and safety assurance we received with this approach. For HF3 and HB4 we are unable to collect their recovery latency under the correct version as the code fix removes the whole error handler.

6.4 Preventing Recovery Failures

We examined whether insights from pilot execution could guide subtle adjustments to recovery logic that would avoid the recovery failures. Once the pilot execution exposes potential errors, the re-execution applies the following strategy: (1) using alternative recovery options if available, (2) slightly adjusting recovery schedules to avoid race conditions and resource conflicts and (3) throttling recovery operations to prevent resource exhaustion and cascading failures (4) cleaning up problematic persistent data to enable correct recover. Our experiments indicated that 14 out of the 20 issues (SL1, SL2, SL3, HF2, HF4, HF5, CS1, CS2, HB3, HB4, YN1, YN2, YN3, YN4) can proceed safely with this approach. We discuss concrete case studies in the appendix §A.2.

6.5 Exposing A New Recovery Bug

PILOT is designed as a runtime testing tool instead of a bug finding technique. Nevertheless, in our experiments it helps expose a new recovery bug in the latest version of HBase. In HBase, a Scan/Get RPC allocating an overly large array can cause an OOM error. To prevent retries from spreading the failure, developers added an array-size check that throws a

	SL1	SL2	SL3	HF1	HF2	HF3	HF4	HF5	CS1	CS2	CS3	CS4	HB1	HB2	HB3	HB4	YN1	YN2	YN3	YN4
Built-in	0.11	✗	✗	✗	✗	✗	✗	✗	✗	✗	5.98	6.24	✗	✗	✗	✗	✗	✗	✗	✗
A/B Test	0.13	5.16	✗	✗	✗	✗	✗	✗	ND	✗	ND	ND	✗	✗	ND	✗	✗	✗	ND	ND
VC [81]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	14.65	14.97	✗	✗	✗	✗
VS [71]	15.72	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	30.01	✗	✗	9.76	9.17
PILOT	0.14	5.17	✗	1.31	1.53	✗	53.89	6.60	223.70	71.30	6.00	6.27	1.31	✗	13.22	12.58	46.93	2.52	3.93	2.67

Table 5. Detection time (sec) on 20 real-world recovery failures. ✗ = Not detected. *ND* = Non-deterministic.

	SL1	SL2	SL3	HF1	HF2	HF3	HF4	HF5	CS1	CS2	CS3	CS4	HB1	HB2	HB3	HB4	YN1	YN2	YN3	YN4
Base ❶	0.13	5.14	✗	1.27	1.47	✗	52.77	6.55	215.94	68.24	5.98	6.24	1.09	✗	12.98	12.03	44.00	2.49	3.86	2.56
Pilot ❷	0.14	5.17	✗	1.31	1.53	✗	53.89	6.60	223.73	71.30	6.00	6.27	1.31	✗	13.22	12.58	46.93	2.52	3.93	2.67
Base ❸	18.64	20.64	11.08	1.65	1.91	ϕ	54.06	8.57	234.55	258.59	161.74	180.91	1.74	18.93	14.62	ϕ	0.60	3.71	6.66	6.26
Pilot ❹	19.97	22.35	12.10	1.75	2.00	ϕ	55.88	9.43	245.98	272.85	169.85	191.56	2.08	20.14	15.86	ϕ	1.24	4.02	7.95	7.69
B+P ❺	21.27	24.03	13.98	2.70	3.69	ϕ	57.24	10.63	269.65	301.81	186.94	204.70	3.35	23.57	18.83	ϕ	1.45	5.37	9.88	9.44

Table 6. Recovery time (sec) across configurations (❶❷=buggy, ❸❹❺=fixed; ✗=not detected, ϕ=not reproducible).

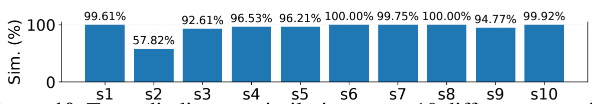

Figure 10. Tree edit distance similarity across 10 different scenarios.

DoNotRetryException. Still, in our fault injection tests PILOT reports pilot-run errors. The output span tree shows, when the reservoir configuration is disabled, DoNotRetryException does not reach the client, leading to endless retry loops. Our proposed fix [12] was confirmed by the developers within hours, marked as "critical" (P1, highest priority), and promptly merged into the master branch.

6.6 Accuracy of Context Propagation

To evaluate the accuracy of context propagation, we compare execution trees (§4.3) for both pilot and normal recovery. We employ the Tree Edit Distance algorithm [69, 79], a standard metric for trace comparison to compute similarity scores. We sample two recovery scenarios per system: one from our evaluated failures in Table 7 (s1-s5) and another from a random recovery entry (s6-s10). We show results in Figure 10. Overall, in most cases the recovery path in pilot runs shares a high similarity with the actual recovery. The remaining divergences stem from three sources: non-deterministic operations, time-sensitive functions affected by pilot-induced delays, and concurrent state modifications that occur after pilot execution creates its shadow version. In HDFS, one case drops to 57.82% because recovery tasks were triggered after a chain of multiple shared states, which fall out of our tracking scope.

6.7 Performance and Overhead

Offline Performance We measure the static performance of PILOT 's workflow. PILOT first analyzes target systems and then generates the instrumented/transformed function codes. The analysis/generation times (in seconds) are: Solr 75/115, HDFS 47/94, Cassandra 51/82, HBase 63/179, and YARN 92/120. All finish within minutes.

Online Setup We evaluated PILOT runtime overhead using standard benchmarks: YCSB for Cassandra/HBase (40 clients, 100K requests, 50% reads), DFSIO for YARN (400 files of 10MB each), the built-in NNBenchWithoutMR for HDFS (100 files, each containing 160 blocks of 1MB), and Apache JMeter for Solr (600K search queries, 300 clients).

For each system, we evaluated under two scenarios. First, we measured overhead during normal operation without fault injection. Second, we injected faults to trigger pilot execution for recovery reactions to these faults by: (1) creating network partition with Jepsen [20], a widely used distributed fuzzing framework; (2) periodically killing a subset of nodes; and (3) modifying files at random offsets to simulate corrupted files.

Online Results Figure 11 summarizes the results. When pilot execution is not triggered (Figures 11(a), (c), and (e)), PILOT incurs negligible overhead on throughput (0.28%–1.35%), memory (0.40%–3.93%), and CPU usage (0%–1.54%). We then collect data for three types of faults injected, with 5 data points for each fault type, totaling 15 data points. In Figures 11(b), (d), and (f), we show that boxes span the 25th–75th percentiles, the black line marks the median, the white circle marks the mean, whiskers extend to 1.5× the interquartile range, and red dots denote outliers. When pilot execution is triggered, the system experiences moderate overhead in throughput (1.85%–8.38%), memory (3.51%–10.66%), and CPU usage (5.21%–17.23%) due to extra isolation and instrumentation overhead.

6.8 False Alarms

We evaluated the false positive rate of PILOT under the same workloads with the same faults injected as §6.7. Pilot execution will be triggered under fault injection, and for every alarm it raises, we scrutinize whether it is a false positive, defined as a case where the recovery is benign but the pilot run still reports an error. The false positive rate is measured as the number of false alarms divided by the total number of alarms raised. The observed rates under different types of fault injection are: Cassandra (0-0%), HBase (0-1.2%), HDFS (0-0.7%), YARN (0-0.2%), and Solr (0-0.2%). Overall, PILOT incurs a low false alarm rate. False alarms mainly come from overly strict checkers that classify tolerable exceptions and

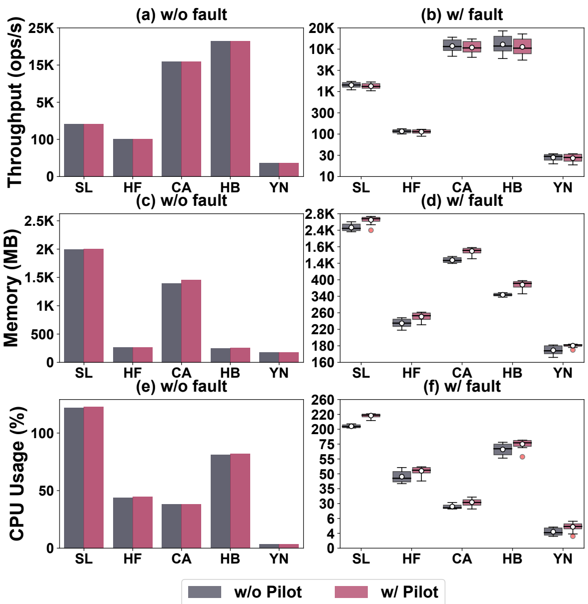

Figure 11. Throughput, CPU and Memory Overhead.

transient timeouts as failures, even though these error signals did not ultimately cause user-visible damage in practice.

7 Limitations and Future Work

First, fully capturing and modeling all the interactions is inherently complex for production systems. PILOT cannot track consequences of "skipped" actions (e.g., lease renewal not triggered), as well as interactions caused by subtle performance shifts. Second, PILOT does not guarantee that it can reliably reproduce nondeterministic failures. We plan to investigate using collected traces during pilot runs to enforce deterministic interleavings. Finally, customizing effective recovery strategies remains an open challenge. We envision integrating LLM agents to generate tailored candidate recovery-policy adjustments from pilot outputs (e.g., tuning retry backoffs based on logs). PILOT acts as the gatekeeper: the candidate is accepted only if it passes pilot validation for safety.

8 Related Work

Failure recovery in distributed systems is well-studied [39, 40, 45, 53, 54, 75, 76], yet many techniques treat failures in isolation and overlook cross-component interactions, a major source of recovery bugs [33, 65, 68]. Prior work categorizes recovery misbehaviors observed in production [47] and highlights common causes such as metastability under overload [50], vicious cycles from unbounded retries [80, 86]. Our study examines diverse failure patterns.

A line of work models distributed system dependency to understand correlated impacts brought by system changes/updates. INDaaS [93] and CloudCanary [92] construct fault graphs from dependency data and perform audits to prevent correlated failures. Some works [35, 38, 44] formally verify that network properties still hold after configuration changes. Our approach focuses on collecting empirical evidence from real execution and complements the modeling approach.

A/B testing is a popular approach to assess recovery effects without risking the whole cluster [52, 56, 90]. Nagaraja *et al.* [71] propose to create a validation slice as an extension of the existing cluster to validate operator actions, which shares a similar goal with us. CrystalNet [60] emulates production networks with real device firmware in isolated sandboxes. NetPilot [90] addresses failures in datacenter networks by iterating candidate devices and applying mitigation. A fundamental limitation of such emulation is that its fidelity is inherently constrained by scale and setup, and it can still trigger cascading issues that lead to failures. In contrast, pilot execution uses in situ runs on the live system for accuracy, while relying on isolation mechanisms to ensure safety.

Several systems mask errors after they occur: failure-oblivious computing [83] discards invalid writes and fabricates return values to keep execution going; Rx [82] rolls back to a recent checkpoint and re-executes under modified conditions; Shadow Filesystem [61] redirects execution to a verified system implementation during faults. These techniques mitigate symptoms post hoc but do not surface recovery hazards early. In contrast, we expose issues before they impact production.

Speculative execution pre-computes work to hide latency and improve performance, originating in processors [36, 49, 58, 73, 77] and later adapted to software systems [43, 57, 59, 72]. Examples include Speculator [73], which continues execution while remote I/O is pending, Xsyncfs [74], which optimizes synchronous I/O in file systems by proactively deferring output until commit, and SWARM [70], which speculatively guesses timestamps for replication and auto-commits upon verification to achieve single-roundtrip latency. They have distinct goals and challenges compared to our approach, as we focus on improving reliability rather than performance.

9 Conclusion

As cloud systems become increasingly complex, correctly recovering systems from failures requires a more rigorous approach. In this work we first present a study of real-world recovery failures in popular distributed systems. Based on the findings, we introduce PILOT, a new execution model to validate failure recovery consequences before applying them to production systems. Our evaluation results show that PILOT is effective and efficient for real-world distributed systems.

Acknowledgment

We thank our shepherd Behnaz Arzani and anonymous reviewers for insightful reviews. This work was supported in part by CNS-2441284, and a 4VA research grant. This project has benefitted from the Microsoft Accelerating Foundation Models Research (AFMR) grant program. We also thank CloudLab [41] and Google for providing computing resources.

References

[1] Apache ZooKeeper releases. https://zookeeper.apache.org/releases.html.

[2] CASSANDRA-13938: Default repair is broken, crashes other nodes participating in repair (in trunk). https://issues.apache.org/jira/browse/CASSANDRA-13938.

[3] CASSANDRA-14096: Cassandra 3.11.1 repair causes out of memory. https://issues.apache.org/jira/browse/CASSANDRA-14096.

[4] CASSANDRA-6415: Snapshot repair blocks for ever if something happens to the "i made my snapshot" response. https://issues.apache.org/jira/browse/CASSANDRA-6415.

[5] CASSANDRA-7560: 'nodetool repair -pr' leads to indefinitely hanging antientropysession. https://issues.apache.org/jira/browse/CASSANDRA-7560.

[6] CASSANDRA: nodetool repair. https://cassandra.apache.org/doc/4.0/cassandra/operating/repair.html.

[7] HBASE-13567: [dlr] region stuck in recovering mode. https://issues.apache.org/jira/browse/HBASE-13567.

[8] HBASE-13877: Interrupt to flush from tableflushprocedure causes dataloss in itbll. https://issues.apache.org/jira/browse/HBASE-13877.

[9] HBASE-14598: Bytebufferoutputstream grows its heapbytebuffer beyond jvm limitations. https://issues.apache.org/jira/browse/HBASE-14598.

[10] HBASE-19980: Nullpointerexception when restoring a snapshot after splitting a region. https://issues.apache.org/jira/browse/HBASE-19980.

[11] HBASE-25898: Rs getting aborted due to npe in replication walentrystream. https://issues.apache.org/jira/browse/HBASE-25898.

[12] HBASE-28589: Server side donotretryexception not propagated to client. https://issues.apache.org/jira/browse/HBASE-28589.

[13] HDFS-10320: Rack failures may result in nn terminate. https://issues.apache.org/jira/browse/HDFS-10320.

[14] HDFS-12914: Block report leases cause missing blocks until next report. https://issues.apache.org/jira/browse/HDFS-12914.

[15] HDFS-14459: Closedchannelexception silently ignored in fsvolumelist.addblockpool(). https://issues.apache.org/jira/browse/HDFS-14459.

[16] HDFS-16115: Asynchronously handle bpserviceactor command mechanism may result in bpserviceactor never fails even commandprocessingthread is closed with fatal error. https://issues.apache.org/jira/browse/HDFS-16115.

[17] HDFS-16689: Standby namenode crashes when transitioning to active with in-progress tailer. https://issues.apache.org/jira/browse/HDFS-16689.

[18] HDFS-4937: Replicationmonitor can infinite-loop in blockplacementpolicydefault#chooserandom(). https://issues.apache.org/jira/browse/HDFS-4937.

[19] HDFS-9908: Datanode should tolerate disk scan failure during nn handshake. https://issues.apache.org/jira/browse/HDFS-9908.

[20] Jepsen: Distributed systems safety research. https://jepsen.io/.

[21] KAFKA-10832: Recovery logic is using incorrect producerstatemanager instance when updating producers. https://issues.apache.org/jira/browse/KAFKA-10832.

[22] Nodes and ephemeral nodes - apache zookeeper. https://zookeeper.apache.org/doc/r3.4.6/zookeeperOver.html.

[23] OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability. https://opentelemetry.io/.

[24] RABBITMQ-658: [ocf ha] do not check cluster health if master is not elected. https://github.com/rabbitmq/rabbitmq-server/pull/658.

[25] SOLR-10914: Recoverystrategy's sendpreprecoverycmd can get stuck for 5 minutes if leader is unloaded. https://issues.apache.org/jira/browse/SOLR-10914.

[26] SOLR-17515: Recovery fails in solr 9.7.0 if basic-auth is enabled. https://issues.apache.org/jira/browse/SOLR-17515.

[27] SOLR-6056: Zookeeper crash jvm stack oom because of recover strategy. https://issues.apache.org/jira/browse/SOLR-6056.

[28] TiDB-963: Tikv cluster not respecting crd replica configuration. https://github.com/pingcap/tidb-operator/issues/963.

[29] YARN-2816: Nm fail to start with npe during container recovery. https://issues.apache.org/jira/browse/YARN-2816.

[30] YARN-4347: Resource manager fails with null pointer exception. https://issues.apache.org/jira/browse/YARN-4347.

[31] YARN-6403: Invalid local resource request can raise npe and make nm exit. https://issues.apache.org/jira/browse/YARN-6403.

[32] YARN-7382: Nosuchelementexception in fairscheduler after failover causes rm crash. https://issues.apache.org/jira/browse/YARN-7382.

[33] R. Alagappan, A. Ganesan, E. Lee, A. Albarghouthi, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Protocol-Aware recovery for Consensus-Based storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 15–32. USENIX Association, Feb. 2018.

[34] Amazon. Aws dynamodb service event in the us-east region. https://aws.amazon.com/cn/message/5467D2/.

[35] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, Los Angeles, CA, USA, 2017.

[36] P. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[37] C. Breck. Kubernetes liveness and readiness probes: How to avoid shooting yourself in the foot. https://blog.colinbreck.com/kubernetes-liveness-and-readiness-probes-how-to-avoid-shooting-yourself-in-the-foot/.

[38] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein. Lessons from the evolution of the batfish configuration analysis tool. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 122–135, New York, NY, USA, 2023.

[39] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, page 3, San Francisco, CA, 2004.

[40] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[41] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[42] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, Dec. 2007.

[43] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano. A survey on thread-level speculation techniques. *ACM Comput. Surv.*, 49(2), June 2016.

[44] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483. USENIX Association, May 2015.

[45] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 149–166. USENIX Association, Feb. 2017.

[46] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dussea, and B. Liblit. EIO: Error handling is occasionally correct. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 14:1–14:16, San Jose, California, 2008.

[47] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 8–8, Santa Ana Pueblo, New Mexcio, 2013.

[48] Helius. A complete history of solana outages: Causes, fixes, and lessons learnt. https://www.helius.dev/blog/solana-outages-complete-history.

[49] G. Hu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Foreactor: Exploiting storage i/o parallelism with explicit speculation, 2024.

[50] L. Huang, M. Magnusson, A. B. Muralikrishna, S. Estyak, R. Isaacs, A. Aghayev, T. Zhu, and A. Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, 2022.

[51] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI. ACM, May 2017.

[52] R. Kohavi and S. Thomke. The surprising power of online experiments. *Harvard business review*, 95(5):74–82, 2017.

[53] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. In *Proceedings of 1986 ACM Fall Joint Computer Conference*, ACM '86, page 1150–1158, Dallas, Texas, USA, 1986.

[54] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[55] B. W. Lampson. Lazy and speculative execution in computer systems. *SIGPLAN Not.*, 43(9):1–2, Sept. 2008.

[56] S. Levy, R. Yao, Y. Wu, Y. Dang, P. Huang, Z. Mu, P. Zhao, T. Ramani, N. Govindraju, X. Li, Q. Lin, G. L. Shafriri, and M. Chintalapati. Predictive and adaptive failure mitigation to avert production cloud vm interruptions. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20. USENIX, November 2020.

[57] N. Li, A. Kalaba, M. J. Freedman, W. Lloyd, and A. Levy. Speculative recovery: Cheap, highly available fault tolerance with disaggregated storage. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 271–286. USENIX Association, July 2022.

[58] S. S. Liao, P. H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J. P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, page 117–128, Berlin, Germany, 2002.

[59] G. Liargkovas, K. Kallas, M. Greenberg, and N. Vasilakis. Executing shell scripts in the wrong order, correctly. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 103–109, Providence, RI, USA, 2023.

[60] H. H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. P. Lopes, A. Rybalchenko, G. Lu, and L. Yuan. Crystalnet: Faithfully emulating large production networks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 599–613, Shanghai, China, 2017.

[61] J. Liu, X. Hao, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and T. Chajed. Shadow filesystems: Recovering from filesystem runtime errors via robust alternative execution. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, page 15–22, Santa Clara, CA, USA, 2024.

[62] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, page 423–437. USENIX Association, 2008.

[63] C. Lou, P. Huang, and S. Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '20, pages 559–574. USENIX Association, Feb. 2020.

[64] D. E. Lowell, S. Chandra, and P. Chen. Exploring failure transparency and the limits of generic recovery. In *Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*. USENIX Association, Oct. 2000.

[65] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Yang, and L. You. Crashtuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 114–130, Huntsville, Ontario, Canada, 2019.

[66] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 589–603, Oakland, CA, 2015.

[67] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 378–393, Monterey, California, 2015.

[68] P. D. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Trans. Comput. Syst.*, 29(4), Dec. 2011.

[69] L. Meng, F. Ji, Y. Sun, and T. Wang. Detecting anomalies in microservices with execution trace comparison. *Future Generation Computer Systems*, 116:291–301, 2021.

[70] A. Murat, C. Burgelin, A. Xygkis, I. Zablotchi, M. K. Aguilera, and R. Guerraoui. Swarm: Replicating shared disaggregated-memory data in no time. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 24–45, Austin, TX, USA, 2024.

[71] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and dealing with operator mistakes in internet services. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, Dec. 2004.

[72] R. Netravali and J. Mickens. Reverb: Speculative debugging for web applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 428–440, Santa Cruz, CA, USA, 2019.

[73] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 191–205, Brighton, United Kingdom, 2005.

[74] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 1–14, Seattle, Washington, 2006.

[75] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, Philadelphia, PA, 2014.

[76] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), Aug. 2015.

[77] A. E. Papathanasiou and M. L. Scott. Aggressive prefetching: an idea whose time has come. In *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, HOTOS'05, page 6, Santa Fe, NM, 2005.

[78] Parse.ly. Kafkapocalypse: a postmortem on our service outage. https://www.parse.ly/kafkapocalypse/.

[79] M. Pawlik and N. Augsten. Rted: a robust algorithm for the tree edit distance. *arXiv preprint arXiv:1201.0230*, 2011.

[80] S. Qian, W. Fan, L. Tan, and Y. Zhang. Vicious cycles in distributed software systems. In *2023 38th IEEE/ACM International Conference*

on Automated Software Engineering (ASE), pages 91–103. IEEE, 2023.

[81] S. Qian, W. Fan, L. Tan, and Y. Zhang. Vicious cycles in distributed software systems. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ASE '23, page 91–103, Echternach, Luxembourg, 2024.

[82] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, page 235–248, Brighton, United Kingdom, 2005.

[83] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and J. William S. Beebee. Enhancing server availability and security through Failure-Oblivious computing. In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. USENIX Association, Dec. 2004.

[84] Spotify. Incident management at spotify. https://engineering.atspotify.com/2013/6/incident-management-at-spotify.

[85] Srcco. Liveness probes are dangerous. https://srcco.de/posts/kubernetes-liveness-probes-are-dangerous.html.

[86] B. A. Stoica, U. Sethi, Y. Su, C. Zhou, S. Lu, J. Mace, M. Musuvathi, and S. 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, page 63–78, Austin, TX, USA, 2024.

[87] L. Tang, C. Bhandari, Y. Zhang, A. Karanika, S. Ji, I. Gupta, and T. 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, page 433–451, Rome, Italy, 2023.

[88] TechInformed. Microsoft confirms ddos cyberattack behind second outage. https://techinformed.com/microsoft-confirms-ddos-cyberattack-behind-second-it-outage/.

[89] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13, Mississauga, Ontario, Canada, 1999.

[90] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang. Netpilot: automating datacenter network failure mitigation. *SIGCOMM Comput. Commun. Rev.*, 42(4):419–430, aug 2012.

[91] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 249–265, Broomfield, CO, 2014.

[92] E. Zhai, A. Chen, R. Piskac, M. Balakrishnan, B. Tian, B. Song, and H. Zhang. Check before you change: Preventing correlated failures in service updates. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 575–589. USENIX Association, Feb. 2020.

[93] E. Zhai, R. Chen, D. I. Wolinsky, and B. Ford. Heading off correlated failures through independence-as-a-service. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 317–334, Broomfield, CO, 2014.

Id.	Feature	Symptom	Studied
SL1 [26]	Follower recovery	State inconsistency	N
SL2 [25]	Follower recovery	Partial failure	N
SL3 [27]	Core recovery	Zombie (whole-cluster)	N
HF1 [17]	Namenode failover	Service unavailability	Y
HF2 [19]	Error handler	Cluster initialization failure	N
HF3 [15]	Error handler	Incorrect results	N
HF4 [13]	Datanode recovery	Service unavailability	N
HF5 [18]	Datanode recovery	Partial failure	N
CS1 [2]	Node recovery	Crash (cascading)	N
CS2 [3]	Node recovery	Out-of-Memory	Y
CS3 [5]	Inconsistency	Partial failure	N
CS4 [4]	Inconsistency	Partial failure	N
HB1 [10]	Snapshot Recovery	State inconsistency	Y
HB2 [7]	RegionServer failover	State inconsistency	N
HB3 [11]	RegionServer failover	Crash (cascading)	Y
HB4 [9]	Scan Retry	Crash (cascading)	N
YN1 [32]	ResourceManager failover	Service unavailability	Y
YN2 [30]	ResourceManager failover	Service unavailability	N
YN3 [31]	NodeManager Recovery	Crash	N
YN4 [29]	NodeManager Recovery	Crash	N

Table 7. 20 real-world recovery failures used in our evaluation. *SL*: Solr; *HF*: HDFS; *CS*: Cassandra; *HB*: HBase; *YN*: YARN.

Appendix A Evaluation Clarifications

A.1 Evaluated Recovery Failures

We show the complete set of evaluated failures in Table 7.

A.2 Preventing Recovery Failure – Case Studies

We discuss several concrete cases from the result in §6.4.

HB3. A RegionServer crash was caused by an empty but corrupted WAL file. Upon detection, HMaster initiated failover, transferring the failed RegionServer's responsibilities to others. However, since the corrupted WAL remained untouched, each recovery attempt triggered a NullPointerException (NPE), cascading across the cluster and eventually crashing all RegionServers. PILOT intercepts this NPE during pilot execution and replaces the dangerous `Runtime.halt()` call, which is invoked when NPE is thrown, with a safe reporting API of PILOT runtime lib. By flagging the faulty WAL file early, PILOT enables intervention—specifically, by deleting the empty file—allowing recovery to proceed cleanly without further data loss or crashes.

CS2. The recovery process requires each node to construct and transmit MerkleTrees to the coordinator node to fix data inconsistency. When inconsistencies are substantial, these MerkleTrees can become excessively large, leading to OutOfMemoryError crashes during recovery. Using pilot execution, PILOT successfully anticipates and reports this failure condition before it manifests. Based on insights from the pilot run, the real execution switches to use: (1) subrange repairs, which divide token ranges into smaller segments, and (2) table level repairs, which scope the repair operation to individual tables rather than entire keyspaces. Both strategies significantly reduce MerkleTree size and prevent OOM conditions.

YN1. The system encounters a timing-sensitive issue during Resource Manager (RM) failover. If the RM is switched over when a MapReduce job is being executed, the `FairScheduler` of RM may attempt to access an empty internal list used to denote containers' scheduler keys during container reassignment for failover, resulting in a `NoSuchElementException` that crashes the newly recovered RM. PILOT detects this subtle timing-related failure by simulating the recovery path and observing the unsafe condition. As a workaround, PILOT reschedules RM failover and avoids the issue.

Appendix B Implementation Details

B.1 Spawning Phantom Threads from Original Threads (Micro-Fork)

§4.2.1 discusses how we support spawning phantom threads from original threads, which we call micro fork. We further elaborate on the details of this mechanism here. For checkpoint capture, PILOT leverages Soot to transform compiled bytecode. Soot first lifts bytecode into an intermediate representation (IR), where PILOT inserts hooks to capture local variables before each call site. The instrumented IR is then lowered back to bytecode. Operating on compiled bytecode makes the instrumentation independent of source code structure and compiler options.

Static instrumentation PILOT employs interprocedural analysis to identify threads that may be influenced by pilot runs, particularly long-running threads that contain synchronization primitives or monitor shared states within loops. For those threads, PILOT instruments all functions along the call paths leading to these potential influence points. Figure 12 illustrates this mechanism using the motivating HBase example. The `AssignmentThread` t2 waits on region reassignment signals through `cv$wrapper.await()`, representing a potential influence point. As shown on the left side of the figure, our instrumentation inserts lightweight hooks: before each function invocation, a frame is pushed to record the call site and local variables; after the function returns, the frame is popped. This maintains a continuous call stack checkpoint. Consequently, when the `AssignmentThread` blocks at `cv$wrapper.await()`, the PILOT runtime library holds a complete checkpoint of its call stack, ready for micro-fork.

Runtime behavior At runtime, consider when a phantom thread t1 triggers `cv$wrapper.signal()` to coordinate with the `AssignmentThread` t2 to complete the reassignment logic. As illustrated in Figure 12, our condition variable wrapper, described in §4.3, propagates `PilotContext` and initiates a micro-fork through `PilotUtil.microFork(t2, pilotCtx)`. The runtime library creates a corresponding phantom thread t3 of `AssignmentThread` t2, and binds the propagated `PilotContext` to it, with the `isFastForward` flag set to `true`.

Phantom thread t3 then reconstructs the original thread's execution state by fast-forwarding through the recorded call stack. As illustrated on the right side of Figure 12, `run$pilot()` function checks the `isFastForward` flag and, if set, uses `PilotUtil.stackInfo.getTop()` to retrieve the next call site from the stack, then jumps directly to that function (e.g., `func1$pilot()`) while bypassing intermediate code. Before entering each function, the system restores local variables from the call stack and pops the corresponding frame. This process continues until the call stack is empty—precisely at the synchronization point where the original thread t2 was influenced. At this point, the `isFastForward` flag is cleared, allowing the phantom threads to execute the region assignment logic entirely within pilot versions and continue normal pilot execution.

This design guarantees that original threads remain unaffected (the `AssignmentThread` stays safely blocked), while pilot run accurately simulates the complete reassignment behavior that would occur in actual recovery.

Algorithm 1: Speculative Lock

```
State Variables:
lockSeq ← 0, currentHolder ← 0, delegate ← original lock

procedure ACQUIRE_LOCK(t)
if t.state = ORIGINAL then
    LOCK(delegate)
    if lockSeq > currentHolder then
        ABORTPILOTRUN()
else if t.state = PHANTOM then
    t.mySeq ← lockSeq
    lockSeq ← lockSeq +1
    while (currentHolder < t.mySeq) ∨ ISLOCKED(delegate) do
        YIELD

procedure RELEASE_LOCK(t)
if t.state = ORIGINAL then
    UNLOCK(delegate)
else if t.state = PHANTOM then
    currentHolder ← currentHolder + 1
```

B.2 Speculative Lock

PILOT implements this mechanism by adding an auxiliary queue to each lock and instrumenting the synchronization primitive interfaces. We show the logic in Algorithm 1. It maintains three state variables: *lockSeq* is an atomic integer which marks the sequence id for the next acquire. *currentHolder* is an atomic integer which marks the sequence id of the current lock holder. *delegate* points to the original lock. An original thread simply acquires the real lock (*delegate*) and runs as usual, aborting any ongoing pilot run if phantom threads are queued or active. A phantom thread, when requesting the lock, takes a sequence number from *lockSeq* and waits until it is its turn and the real lock is free (checked via ISLOCKED(*delegate*)). Once both conditions are satisfied, the phantom enters a speculative critical section without touching the real lock. When it finishes, it increments *currentHolder*

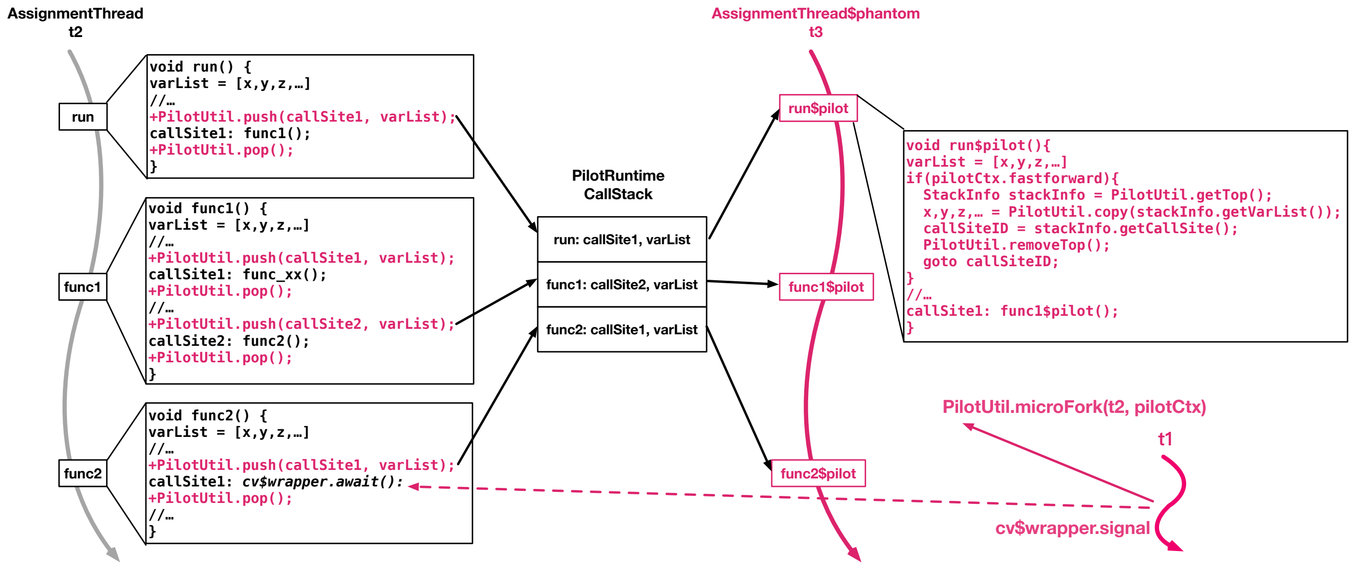

Figure 12. Code examples (simplified) for micro-fork used in HBase-25898.

so the next phantom can proceed. This approach ensures original threads are never delayed.

B.3 Propagation Mechanism

Propagating `PilotContext` requires distinguishing between *explicit handoff* and *implicit handoff* to track pilot execution and construct causal relationships. Figure 6 illustrates five representative scenarios.

Explicit Handoff Explicit handoff occurs at well-defined interfaces where a clear sender-receiver relationship exists. `PilotContext` can be directly attached at these boundaries.

Task-Executor Model (❶). Many recovery actions enqueue asynchronous tasks that later run on executor threads, breaking direct thread lineage and losing pilot intent. PILOT instruments standard task interfaces (e.g., `Runnable`/`Callable`/`Future`) to carry context from the submitting pilot thread. When the task executes, the runtime inspects the context: if valid, it activates a phantom thread and redirects to pilot-version handlers; otherwise, execution proceeds normally.

Synchronous Cross-Process Communication (❹). RPC/HTTP boundaries sever thread identity and process state, so pilot intent must traverse message channels. PILOT injects `PilotContext` into outbound metadata (e.g., HTTP headers, gRPC metadata, or custom RPC fields), piggybacking on standard interceptors or extending existing serialization hooks. At receiver entry points (e.g., servlet filters, gRPC service stubs), the runtime detects the context, spawns or reuses a phantom thread bound to it, and dispatches pilot-version handlers.

Asynchronous Cross-Service Communication (❺). Some third-party services (e.g., ZooKeeper) deliver callbacks asynchronously, where the causality between a pilot-originating write and a later callback event is explicit but temporally decoupled. PILOT introduces a thin client-side proxy: on pilot-originating operations, it persists the current `PilotContext` to a side metadata channel (e.g., a sibling znode); on watch delivery, it intercepts the callback, retrieves the matching context, and activates a phantom thread to handle the event in pilot mode.

Implicit Handoff Implicit handoff occurs when pilot execution alters shared state that original threads observe, causing them to change their behavior without direct communication. PILOT intercepts these state modifications to propagate `PilotContext`.

Synchronization Primitives (❷). Condition variables, latches, and similar primitives are a form of shared state which can cause pilot-induced wakeups to affect original threads. PILOT replaces common synchronization primitives with wrapper variants that record waiters and signallers while carrying context. When a phantom thread signals, the wrapper dequeues the next waiter: if it is an original thread, PILOT creates a new phantom thread with the signaling context bound to it, continuing the awakened path in pilot mode. This transforms the wait queue into an explicit propagation bridge while preserving the isolation of original threads.

Shared Variable Monitoring (❸). Systems often have threads that poll or branch on shared variables (e.g., status flags) that pilot execution may update, implicitly handing off control through state changes. PILOT uses static analysis to identify such variables (typically lock-protected or concurrent types) and augments them with an embedded `PilotContext` slot, injecting guards at access sites. When pilot execution writes to such a variable, it marks the variable with its context. Subsequently, when an original thread reads this pilot-marked state, the guard creates a phantom thread that continues execution in pilot mode.