Graceful Restart Testing for Distributed Systems via LLM-Based Test Transformation

Shuai Wang
University of Illinois
Urbana-Champaign, IL, USA
swang516@illinois.edu

Yinfang Chen
Arizona State University
Tempe, AZ, USA
yinfang@asu.edu

Darko Marinov
University of Illinois
Urbana-Champaign, IL, USA
marinov@illinois.edu

Yongle Zhang
Purdue University
West Lafayette, IN, USA
yonglezh@purdue.edu

***Abstract*—Graceful node restarts are essential routines in distributed system operations. However, systematically testing graceful node restarts is challenging. A graceful restart must coordinate node shutdowns and rejoins, and restart correctness is highly sensitive to the workload, restart timing, and correctness oracles. This paper presents Palingenesis, a large language model (LLM)-based test transformation approach for graceful restart testing. To alleviate the burden of manually writing graceful restart tests, our key insight is to automate graceful restart testing via test transformation: we transform existing developer-written tests into restart tests, reusing their workloads and correctness oracles to validate graceful restart procedures. Palingenesis automatically transforms these tests into restart tests by using LLMs to infer test intent and lifecycle phases from test code and system context, then injecting graceful restarts while preserving the original workloads and test oracles. These transformed tests run with Palingenesis's runtime support, which performs system-specific restart procedures and refreshes cluster-derived references that become stale after the restart. We evaluate Palingenesis on seven widely used distributed systems, conducting 18316 restart test executions across 4459 transformed restart tests. The transformed restart tests reveal 17 new restart-related bugs: 6 fixed, 5 more confirmed, and the remainder are pending. On average, the transformation adds 2.27x execution overhead.**

***Index Terms*—software testing, distributed systems, test transformation, large language models**

I. Introduction

Graceful node restarts are essential routines in production distributed systems [1]–[6]. A graceful restart is a coordinated shutdown/rejoin protocol. Operators restart nodes for rolling upgrades, configuration changes, maintenance, performance tuning, and other operational tasks [4], [5], [7]–[11]. These graceful restarts, unlike crashes [12], [13], are "planned": operators initiate them as part of controlled rollouts, and the system expects nodes to execute shutdown protocols and rejoin the cluster cleanly. Before exiting, a node must *actively* execute shutdown logic (e.g., reject new requests, flush in-memory state, step down from leadership, drain resources, and hand off responsibilities); after restarting, it must rejoin the cluster with correct role-specific semantics [7], [14]–[19]. Bugs in this graceful restart protocol often stem from missing checks and inconsistent state on the shutdown/rejoin paths (e.g., accessing closed resources or rejoining with an uninitialized role), and are typically invisible under steady-state testing.

Automated robustness testing for distributed systems has long focused on crash failures by injecting faults and observing whether the system can recover [12], [20]–[26]. Recent work broadens robustness testing for distributed systems to partial-failure testing, greybox fault fuzzing, and synchronization-focused exploration [13], [27]–[30]. However, planned graceful restarts remain under-tested. Unlike crash-centric recovery, a planned graceful restart must validate the shutdown/rejoin protocol itself. As a result, restart-related bugs that only arise during shutdown or rejoin can remain undetected, even when the system has been tested extensively under crash failures. Moreover, the ability to detect restart-related bugs depends strongly on the testing workload: many such bugs surface only after the cluster has been driven into a restart-sensitive state, whereas restarting a freshly started or quiescent cluster often exercises only shallow restart paths. Obtaining such meaningful workloads remains challenging in practice because these bug-triggering states are feature-specific and costly to construct or explore systematically.

Many distributed systems already maintain extensive suites of *cluster tests*, often called MiniCluster-style tests, that start a small in-process (or local) cluster, drive realistic workloads, and assert externally visible outcomes [31]–[36]. Crucially for graceful restart testing, MiniCluster frameworks typically expose programmatic node-control operations (e.g., stopping, starting, restarting a node, and waiting for it to rejoin), making graceful restart an action that the test can trigger directly [31], [32]. Cluster tests are a natural fit for graceful restart validation because they execute real system components and protocols (closer to production behavior than unit tests) and already include correctness oracles, so restart-induced violations surface as test failures.

However, leveraging cluster tests for graceful restart testing is challenging for two reasons: (1) selecting execution points to inject restarts that trigger real bugs, and (2) adjusting test execution after the restart to avoid false alarms. First, cluster tests typically contain many execution steps and multiple nodes (e.g., one to ten nodes in HDFS [31]), so injecting a restart between every step for each combination of nodes creates a huge test space that is hard to cover exhaustively. Moreover, many such generated tests are ineffective because restart bugs surface only when the workload has driven the system into a particular state and the test continues after the restart to observe post-rejoin behavior. Restarting too early can be uninformative because little restart-sensitive state has accumulated; restarting

too late may miss failures because the relevant checks may already have executed. Selecting restart points therefore requires semantic understanding of the test intent, the cluster lifecycle phases (setup, workload, stabilization), and the project-specific helper abstractions that hide when key state transitions occur (e.g., leadership established or metadata stabilized).

Useful restart points thus occur at critical state-changing or stabilization steps of the behavior under test, while still leaving its later checks to execute after the restart. Existing techniques facing similar challenges often rely on program analysis with system-specific domain knowledge to identify critical operations worth testing [12], [24], [37]. In this work, we leverage the semantic reasoning capabilities of large language models (LLMs) [38]–[40] to infer program intent from implementation [41]–[43], identifying restart-sensitive execution steps and instrumenting them with restart operations.

The second challenge is that most cluster tests implicitly assume steady-state execution: nodes remain alive, references remain valid, and helper utilities behave as if the cluster context is stable until teardown. As a result, restarting nodes at arbitrary points often produces false alarms: failures that violate test assumptions rather than reveal restart bugs. Notably, restart injection must account for in-memory objects derived from the cluster (e.g., client handles or node-local references), which may become stale after restart; a useful restart test should keep exercising the system rather than failing immediately on stale handles.

Motivated by these two challenges, we propose a test-centric perspective: *graceful restart testing as automated test transformation*. We present Palingenesis, an end-to-end system that transforms existing cluster tests into restart tests by injecting graceful node restarts while preserving the original workloads and test oracles. The key idea is to treat graceful restart testing as a *semantic refactoring* problem with two requirements: selecting restart points that align with the test's intent and lifecycle, and keeping the transformed test executable across restart. Palingenesis meets the first requirement by using LLMs to infer test phases from the test code together with project context (e.g., cluster APIs, node definitions, and helper utilities), and then injects graceful restarts at restart points aligned with the test intent and critical state transitions. To satisfy the second requirement, Palingenesis provides a practical stale-reference refresher that tracks cluster-derived references during execution and refreshes them after restart when possible, reporting explicitly when not possible. The resulting restart tests can then be executed systematically to test graceful restart behavior under realistic workloads.

We conduct an evaluation on seven widely used Java-based distributed systems with MiniCluster-style tests, transforming 4459 cluster test methods and executing 18316 restart runs. The transformed tests expose 17 restart-related bugs: 6 fixed, 5 more confirmed, and the remaining reports are pending with patches submitted. The discovered bugs span shutdown, reboot/initialization, and post-restart recovery, and can manifest as node crashes, hangs, or safety-invariant violations.

This paper makes the following contributions:

- We automate graceful restart testing by reusing developer-written cluster tests and transforming them into restart tests by injecting graceful node restarts and reference refreshes, while preserving the original workloads and correctness oracles.
- We show how to use LLMs with project context to infer test intent and identify meaningful points for restart injection. Here a restart point is *meaningful* if the preceding workload has driven the cluster into a restart-sensitive state and the original oracle remains applicable afterwards (the test does not fail when no bug is present).
- We identify stale cluster-derived references as a key challenge in running transformed restart tests and provide a practical stale-reference refresher for test executions.
- We build Palingenesis, an end-to-end system that transforms cluster tests into restart tests and executes them with our runtime support. We evaluate Palingenesis on seven systems with 4459 transformed tests and 18316 restart runs, exposing 17 restart-related bugs: 6 fixed, 5 more confirmed, and the remainder are pending.
- Artifact: https://github.com/teamconfx/Palingenesis.

II. Background and Motivation

A. Graceful Restarts Are Common but Under-Tested

Graceful node restarts are essential routines in production distributed systems [1]–[5]. Operators restart nodes for planned maintenance, configuration changes, and upgrades via rolling procedures that preserve availability: ZooKeeper [9] documents rolling restart sequences, and Kubernetes relies on graceful pod termination during rolling updates [5], [7].

Automated robustness testing for distributed systems has long focused on crash failures and recovery [12], [13], [20]–[24], [27], [30]. These approaches model failure as an immediate stop and skip the system-specific shutdown and rejoin procedures, so they can miss bugs that arise when shutdown hooks run, responsibilities are handed off, or a rebooted node re-establishes role state. Our goal is complementary to crash-injection tools: we test correctness of the restart protocol.

Unlike a crash, a graceful restart requires a node to execute a system-specific shutdown protocol, suspending background work, persisting metadata, draining requests, and transferring responsibilities, before rejoining with correct membership, role, and reconstructed state. Correctness thus depends on recovery semantics and subtle state transitions around shutdown and rejoin, so the same workload can expose different bugs depending on restart timing, even if steady-state tests pass.

Figure 1 shows simplified ZooKeeper graceful shutdown: it tears down internal services and closes persistent state before the process exits. This path can fail in practice: Palingenesis found a bug, which we reported in ZOOKEEPER-5012, showing that QuorumPeer.shutdown() may call zkDb.close() without a null check, and reboot must still re-initialize components and rejoin with correct role-specific semantics. When zkDb is null, zkDb.close() raises a NullPointerException that escapes the surrounding catch block, which handles only IOException.

```
1:  public void shutdown() {
2:    running = false;
3:    ...
4:    if (leader != null)
5:       { leader.shutdown("quorum Peer shutdown"); }
6:    if (follower != null)
7:       { follower.shutdown(); }
8:    ...
9:    try {
10:     //Missing null check in shutdown: ZOOKEEPER-5012
11:     zkDb.close();
12:   } catch (IOException e) {...}
13: }
```

Fig. 1: ZooKeeper QuorumPeer graceful shutdown.

```
1: @Test
2: public void testFinalizedReplicas() throws Exception {
3:   Configuration conf = new HdfsConfiguration();
4:   MiniDFSCluster cluster = new MiniDFSCluster
                .Builder(conf).numDataNodes(3).build();
5:   ...
6:   try {
7:     createFiles("/test", (short) 3);
8:     ...
9:     util.waitReplication(fs, TopDir, (short) 3);
10:    cluster.restartDataNodes(); // restart all DataNodes
11:    ...
12:    checkFiles("/test");
13:  } finally { cluster.shutdown(); }
14: }
```

Fig. 2: A simplified HDFS cluster test that tests restart.

B. Cluster tests: Realistic Workloads with Built-in Oracles

Many distributed systems already maintain extensive suites of *cluster tests*: developer-written tests that start a small local cluster, run realistic workloads, and check correctness with developer-defined oracles, with Hadoop's MiniDFSCluster [31] and HBase's HBaseTestingUtility [32], which starts a MiniHBaseCluster, as representative examples. Cluster tests approximate production semantics with real components and protocols, run faster than full deployment tests, and already encode correctness oracles. Prior work on distributed systems testing similarly reuses such tests as workloads to validate different behaviors at scale [33]–[35], [44]–[46].

Some cluster tests already validate restart behavior, but only in narrow forms. For example, the HDFS test in Figure 2 restarts all DataNodes and checks that finalized replicas remain readable. It has an explicit restart procedure and oracle but covers one restart granularity (*restart all DataNodes*) and one workload pattern (create files, wait for replication, verify). Writing it took substantial manual effort, and many other restart-sensitive workloads in the suite never run under restarts.

C. Challenges of Restart Testing in cluster tests

We transform existing cluster tests that have no restarts into restart tests by injecting graceful node restarts while preserving the original workload and test oracles. The challenge is not injecting a restart anywhere but performing restarts consistent with the test intent and cluster lifecycle.

1) Restart point matters: Meaningful restart points align with the test intent and with the state-changing or stabilization steps the test performs on the cluster. A restart point must also be one at which the original oracle remains applicable and the test does not fail when no bug is present. Section III-A develops this requirement into an explicit validity condition on restart points. Figure 3 uses a real HBase cluster test to illustrate the challenges. The test starts a MiniHBaseCluster with BucketCache

```
1: @Test
2: public void testPrefetchPersistence() throws Exception {
3:   ...
4:   cluster = TEST_UTIL.startMiniHBaseCluster(option);
5:   Table table = TEST_UTIL.createTable(td, null);
6:   try {
7:     table.put(...);
8:     ...
9:     TEST_UTIL.flush(tableName);
10:    // Restart the HBase RegionServer
          right after flush: HBASE-28839
13:  }
14:  ...
15: }
```

Fig. 3: An existing HBase cluster test that can find a (previously known) restart bug (HBASE-28839) after restart injected.

persistence enabled, using one Master and one RegionServer, creates a table, writes and flushes data to create store files, and continues the workload, warming BucketCache, while checking expected outcomes. Without restart, the RegionServer stays alive, so cache persistence and restore are never exercised mid-test.

In this example, restarting the RegionServer immediately after the flush, before later operations warm BucketCache, can trigger a known restart bug (HBASE-28839). With BucketCache persistence enabled, shutdown persists cache metadata and the next startup restores it; if the cache is empty, the persisted state can be inconsistent for recovery and the restarted RegionServer can crash while rebuilding cache state. Restarting later can miss the bug, which shows that restart placement must expose restart-specific behavior that the original oracle can observe.

2) LLM-based restart-point selection: Choosing restart points requires semantic reasoning about the test's intent, the cluster lifecycle, and project-specific helper abstractions that hide the relevant state transitions; directly asking LLMs to "insert restarts" is brittle. An effective transformation must (i) infer the test's intent and implicit phase boundaries, (ii) invoke the correct restart procedure for project-specific cluster/node roles, and (iii) keep the transformed test executable across restart while preserving the original workload and oracles. Recent LLMs can support such semantic reasoning and the localized edits it requires [38], [39], [47]–[51], but these constraints must be enforced systematically.

3) Executability across restarts: A second, easily overlooked challenge is *reference staleness*. MiniCluster-style tests frequently cache cluster-derived objects, such as client handles, service stubs, or node-local components. A graceful shutdown tears these down and the node rejoins with fresh instances, so cached references become invalid (e.g., pointing to closed connections) and transformed tests can fail before exercising post-restart behavior, raising false alarms.

III. Methodology and Implementation

We show the workflow of Palingenesis in Figure 4. Starting from an existing cluster test, Palingenesis (1) assembles a prompt with the test, relevant project context, and the unified restart API (Sections III-B1 and III-B2); (2) uses LLMs to inject multiple restart points, generating restart tests that follow the Palingenesis template (Section III-B3); (3) compiles them, feeding compiler diagnostics back to the LLM to repair errors (Section III-B4); (4) executes them, activating one restart point per run (others are no-ops) and dispatching it to the

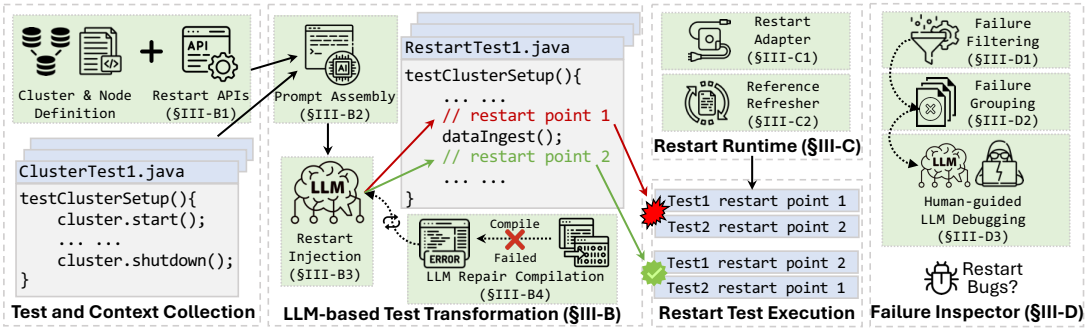


Fig. 4: End-to-end workflow of Palingenesis. Green blocks highlight the key components.

project restart adapter (Section III-C1); (5) keeps cluster-derived references valid across the restart with a stale-reference refresher (Section III-C2); and (6) sends failing executions to failure inspector, which filters, groups, and inspects whether a failure is a real restart bug (Section III-D).

A. Transformation Overview: From cluster test to restart test

Palingenesis turns a developer-written cluster test into a restart test by injecting graceful node restarts while preserving the original workload and oracles. Because a graceful restart models a planned reboot, not a crash, we assume the system should still satisfy the test's original invariants once recovery stabilizes; we avoid heuristic oracle relaxation, which could mask real bugs. The difficulty is that these tests were not written to survive a restart. A correct restart preserves durable state but may still discard volatile state, either *cross-node* (sessions, connections, credentials) or *node-local* (caches, in-memory records), and open a brief unavailability window. These effects can invalidate the invariants that certain oracles check, producing false alarms. An injection point is thus *valid* only if the test's suffix reuses no pre-restart volatile state, asserts only after recovery completes, and observes only state documented as surviving a planned restart. A failure at a valid point corresponds to a true bug, and a false alarm elsewhere.

Accordingly, the false alarms we observe (Section IV-B) fall into three classes: (1) *uninterrupted execution*, reuse of pre-restart sessions or connections; (2) *transient in-memory state*, checks on volatile in-memory state that a restart legitimately resets; and (3) *strict immediate availability*, assertions issued before recovery completes. We capture the validity conditions as a *restartability contract* that a developer or Palingenesis's transformer can check per candidate point. Fully automating this decision is future work. Palingenesis instead places restarts at semantically meaningful points (Section III-B) and routes failures to the failure inspector (Section III-D).

B. Component 1: LLM-based Cluster Test Transformer

Palingenesis's transformer takes a cluster test and produces a restart test by injecting multiple restart points into the test body while preserving the original workload and test oracle. Restart injection requires resolving three semantic constraints: test intent, cluster and node lifecycle/roles, and project-specific abstractions. Recent LLM-based systems can leverage test code and project context to propose such localized edits [38], [39], [42], [43], [47], [48], [51]. However, relying on LLMs alone to synthesize precise graceful restart procedures is brittle:

```
1: Palingenesis.at("<label>") // restart point label
2:   .on(clusterVar)    // cluster instance
3:   .restart("<role>") // node role
4:   .withIndex(<idx>)  // node index (if applicable)
5:   .execute(); // restart and wait for stabilization
```

Fig. 5: Unified restart API used by transformed restart tests.

cluster APIs and testing patterns differ across systems, and small syntactic differences can imply large semantic differences. Palingenesis constrains generated code to a unified restart API and validates it with a compilation-repair loop [40], [43].

1) Unified Restart Template API: Palingenesis exposes a unified API (Figure 5) for restart injection designed to be shared across distributed systems with MiniCluster-style test frameworks. The API lets a restart test request a graceful restart of a specific node role, and Palingenesis translates the request into the project-specific graceful shutdown and rejoin procedure for the cluster under test.

This API bridges transformation and runtime: it standardizes how restart logic is expressed, so the transformer generates a fixed call pattern instead of project-specific restart code; constraining the LLM to it substantially improves compilation success (quantified in Section IV).

When a restart test contains multiple restart points, Palingenesis executes *one* per run: it activates a single point (identified by its label and restart target) and treats all other injected calls as no-ops, isolating each injection's effect and yielding a test matrix of one restart point and target per run.

2) Prompt Construction and Context: For each target cluster test, Palingenesis constructs an LLM prompt (structure in Figure 6) that includes (1) the cluster test source code, (2) project code needed to understand the cluster instance, node roles, and test helpers (e.g., cluster setup/teardown utilities, topology configuration, and commonly used synchronization and client-access helpers), and (3) the unified restart API that the transformed test must use to perform restarts.

Providing project context is critical because restart placement is position-sensitive. Operationally, a useful restart point sits at a critical state-changing or stabilization step of the behavior under test while leaving the original checks to run after restart. Such points typically occur at lifecycle boundaries where durable state has advanced and the test has reached a stable checkpoint (e.g., after a write/flush/commit).

3) Restart Injection and Restart Test Generation: The LLM decides where to place restart points and which node(s) to restart, injecting a restart call at each selected point in the cluster test body. Figure 5 shows the injected code pattern.

```
You are a senior software engineer who will
transform mini cluster tests.
Goal: Given a cluster test, create a restart-aware
test that preserves the original workload and
oracles, while injecting multiple graceful restart
points using the unified restart API.

Inputs:
(A) Original cluster test code:${CLUSTER_TEST_CODE}
(B) Relevant project context (snippets to
    understand helpers and lifecycle):
    - Mini Cluster source code:${MINI_CLUSTER_CODE}
    - Helper utilities used by the test:${UTIL_CODE}
    - Nodes source code and definitions:${NODE_CODE}
(C) Unified restart API (must be used verbatim):
    Palingenesis.at("<label>").on(<clusterVar>).res
    tart("<role>").withIndex(<idx>).execute();

Task:
1) Identify the test phases (setup, workload,
   synchronization barriers, oracle checks,
   teardown) and cluster state-changing statements
2) Choose K restart positions that are
   semantically meaningful and stable (e.g.,
   before or after cluster operations), and avoid
   invalid placements (e.g., before node/cluster
   is ready or after node/cluster is shutdown)
3) Inject a restart at each selected position with:
   - a stable, descriptive label (e.g.,
   "after_flush", "before_write")
   - an explicit restart target (role + index)
4) Ensure the new test keeps the same workloads,
   with only restarts injected

Output requirements:
- Output ONLY the compilable test code diff
- Do NOT modify the original test workloads; keep
edits minimal
- Do NOT invent new APIs; only use the provided
context and the restart API
```

Fig. 6: Example prompt for cluster test transformation. Purple boxes are placeholders for the code snippets.

Palingenesis.at(...) labels the point for reporting and selective activation. on(...) binds the request to the test's cluster instance, restart(...) selects the node role, withIndex(...) picks an instance when several share that role, and execute() performs the graceful restart, blocking until it completes.

The LLM outputs a test diff that the transformer patches into the test. The restart test preserves the original logic and workload, adding only restart calls that follow the unified API, typically multiple labeled points with explicit targets (node role and index) for systematic, reproducible exploration. The result is *restart-aware but non-invasive*: with no restart point activated, all injected calls are no-ops and the restart test executes equivalently to the original.

4) Compilation-Repair Loop: Palingenesis then compiles the generated restart tests. On failure, it feeds the compiler diagnostics and current code back to the LLM for repair, looping until the test compiles or the repair budget is exhausted, after which the cluster test is reported not transformable.

C. Component 2: Runtime Support for Restarts

Palingenesis provides the runtime support for restart tests to perform restarts and keep the test logic executable afterwards: executing a restart inside a test requires a multi-step, system-specific shutdown and rejoin procedure, waiting for recovery, and maintaining the validity of cluster-derived references that the remaining logic depends on. These tasks are coupled to node roles and system conventions: roles often need different restart procedures, and each system defines its own healthy-recovery criteria.

To support uniform transformation across diverse systems while respecting role-specific restart behavior, Palingenesis provides (1) the unified restart template API used by the transformer (Section III-B1), (2) a project restart adapter mapping unified API calls to each project's standard, role-specific restart procedure (Section III-C1), and (3) a stale reference refresher preserving the executability of the remaining test logic after restart (Section III-C2).

1) Restart Adapter: A restart adapter connects the unified API to a concrete system, translating API calls into the system's shutdown and restart procedures. For a new project, the user supplies this thin integration layer (typically a few hundred lines) specifying how to gracefully shut down, restart, and rejoin each node role and how to wait for a healthy cluster. Palingenesis then forwards each unified-API request to the adapter's role-specific procedure.

In the MiniCluster setting, a graceful restart is an in-process shutdown/restart procedure that executes the system's intended shutdown and rejoin logic. When the project provides MiniCluster restart APIs, the adapter calls them; otherwise, it directly invokes the target role's shutdown and startup methods and then runs health checks until the restarted node and cluster stabilize. Leader behavior after restart follows the project's own MiniCluster restart semantics: in most systems a graceful restart restores the same node to its prior role, and Palingenesis preserves this project-specific behavior rather than imposing its own. A node that legitimately rejoins as a follower (i.e., loses leadership) per the system's own semantics is not flagged as a restart bug. Palingenesis targets application-level restart semantics used in planned maintenance and rolling upgrades, rather than OS-level process signals, which are out of scope.

2) Stale Reference Refresher: A key challenge in restart testing via cluster tests is reference staleness: cluster-derived objects obtained before restart (e.g., client handles, service stubs, or node-side references) can become invalid after graceful shutdown cleans up resources. For example, a cluster test may cache a node-local reference for inspection. After restarting that node, the old in-memory object is replaced, and later operations on the cached reference may fail or read stale state. Palingenesis therefore provides a stale-reference refresher that tracks such references and refreshes them after restart by re-invoking the same getter methods. The refresher is a pragmatic mechanism for *executability* in in-process MiniCluster restart testing: it keeps the remaining transformed test logic running after restart so recovery code paths and original oracles can still be exercised, but does not claim semantics-preserving reconstruction for all objects. Accordingly, the refresher is best-effort and fail-fast: it assumes recorded expression chains are idempotent and side-effect free (i.e., getter-like), and it either refreshes a tracked reference by replaying its recorded derivation chain or, when a chain is not replayable under this assumption, throws an explicit refresh exception rather than silently continuing with a stale or guessed reference. Crucially, the refresher does not mask restart semantics: it only reconstructs getter-like, side-effect-free, cluster-derived references (i.e., object identity and handles), never system state or oracle outcomes. The original workload and assertions are unchanged, so a genuine restart fault still surfaces through the

```
1: Client c = cluster.getClient();// Get client from root cluster
2: Table t = c.getTable("table1");// Get table from client
```

Fig. 7: Example of expression-chain capture for refreshing.

test's own oracle. When a reference cannot be reconstructed under the getter-like assumption, the refresher fails explicitly rather than substituting a guessed value. At high level, the refresher has three phases: (i) detect the cluster root object, (ii) track cluster-derived references, and (iii) replay these chains after restart to obtain fresh references.

Cluster root detection. The refresher first identifies the *cluster root* object representing the running cluster, the anchor for tracking cluster-derived references: it registers the object passed to the injected .on(cluster) call and, as a backup, marks any stored value whose runtime type matches a registry of known cluster types populated from the adapters.

Transitive reference tracking and expression chains. During test execution, the refresher tracks all objects transitively derived from a cluster root. Concretely, when instrumented code invokes a method on a tracked receiver, the refresher records the receiver, method name, and arguments, and associates the return value (if non-null) with an *expression chain* that extends the receiver's chain with the invoked method. For utility methods commonly used in cluster tests, the refresher applies pattern matching on class and method names to treat return values as cluster-derived if they are associated with the cluster root or a tracked reference. Arguments are handled explicitly: immutable values (e.g., primitives, primitive wrappers, and strings) are snapshotted at capture time, while object arguments that are themselves cluster-derived are represented by their own tracked chains and refreshed first. The result is a dependency graph that stores the expression chains needed to reconstruct each tracked reference after restart. Figure 7 shows an example of expression-chain capture. The refresher detects cluster as a root and records chains for derived references: (1) c: root.getClient(), and (2) t: (root.getClient()).getTable("table1"). The flow is record-then-replay: the refresher captures each derivation chain before the restart, and after the restart replays the chain against the new cluster root to obtain a fresh reference. On restart, it replays the chain for t against the new cluster root to obtain a fresh Table object, using the snapshotted immutable argument "table1". When a chain depends on another derived object, the refresher refreshes the dependency first, then replays the dependent chain.

Post-restart refresh and reference update. After a restart completes, the refresher refreshes cluster-derived references by replaying recorded expression chains. The refresher performs *lazy* refresh: it only replays a chain when the corresponding reference is accessed after restart. If a refresh fails, the refresher throws an explicit error instead of silently continuing. To avoid semantic changes, a manually maintained *blacklist* marks non-reconstructible types that are never refreshed. Accessing one after restart fails explicitly rather than returning a guess. Immutable primitive wrappers and strings are always treated as valid. Replay requires recorded chains to be idempotent and side-effect free: only getter-like derivations are refreshed, and non-deterministic, transient-state-dependent, or side-effecting patterns fail explicitly when a refresh is required.

D. Component 3: Failure Inspector

Failure inspector decides whether failures from many restart test executions correspond to real restart bugs. Restart injection produces generic noise and redundant failures where different tests or points expose the same root cause, so failure inspector filters noisy failures, groups the rest, and debugs one representative per group while covering all distinct root causes.

1) Failure Filtering: As a first step, failure inspector conservatively filters failures unlikely to reflect restart bugs and more likely to arise from generic testing noise or environmental issues. Concretely, it filters out three types of failures:

Palingenesis exceptions. Failures caused by exceptions from Palingenesis itself (e.g., RestartFrameworkException or StaleReferenceRefreshException) are excluded because they indicate framework issues (e.g., references are not correctly refreshed), not bugs in the system under test.

Pure timeouts. Failure inspector filters timeouts whose logs lack fatal signals such as crashes, corruption, or invariant violations. Most timeouts come from application-level waits in the original cluster test (e.g., waitFor(...)) and may reflect post-restart convergence latency rather than a bug: the unified restart API blocks until a basic healthy state but does not guarantee full convergence of asynchronous background protocols.

Resource exhaustion. Failure inspector filters failures caused by out-of-memory conditions or resource limits due mainly to testing environment constraints. Such failures are unlikely to reflect restart bugs in the system under test.

2) Failure Grouping: After filtering, failure inspector groups failures by a generalized stack trace: it keeps the deepest exception (the last Caused by:) with its first stack frame as the signature and strips variability such as line numbers and run-specific tokens (restart points, object identifiers, timestamps, UUIDs). Matching signatures form one group, reducing inspection cost when many runs share a root cause.

3) Human-guided LLM debugging: For each group, failure inspector selects one representative restart test and restart point. An LLM agent then reruns it to confirm reproducibility and inspects logs and traces to propose a root-cause hypothesis. Finally, a human inspector makes the bug or false-alarm decision. The LLM only proposes root-cause hypotheses and never makes the final call. For the representative of each group, an inspector receives these inputs: the failure logs and stack trace, the relevant system and restart test code, and the LLM's root-cause hypothesis. The inspector judges the representative against the restartability contract of Section III-A: the group is a real restart bug if the system/test cluster mishandles the restart, and a false alarm if it only violates an unsupported assumption, that is, its restart point meets one of the three exclusion conditions: reuse of volatile cross-node state established before the restart, an observation of only node-local volatile state, or an operation or assertion that fails before recovery completes. When the LLM's hypothesis is unconvincing, the inspector reruns the LLM analysis with tightened context (additional logs and the contract clause at issue) and makes the final decision.

E. Implementation

Palingenesis is implemented in Java with three components: a transformation driver that generates restart tests using LLMs, a runtime library that executes injected restarts and provides post-restart support, and an inspector that summarizes and groups failing executions (LLM configuration in Section IV-A). A Maven plugin extending Maven Surefire [52] integrates with existing build pipelines: it runs each transformed restart test once per injected restart point and target node, enabling only the selected restart call and treating the others as no-ops. For the evaluation projects, per-project restart adapters implement role-specific graceful shutdown, restart, and rejoin logic (Section III-C1), adding on average 250 lines of code per project. This line count is only a partial proxy for integration effort, which also requires understanding the project's restart APIs, node roles, health checks, and stabilization conditions. In practice, most of this logic comes directly from the project's existing MiniCluster and node-lifecycle APIs, which already encapsulate role-specific shutdown, restart, rejoin, and recovery checks. The adapter mainly maps the unified API onto these entry points. With code-agent assistance, a developer can integrate Palingenesis with a new system in about one hour. Maintenance cost is also low: these APIs are stable, changing on the order of once every 2.5 years per commit history, so an adapter rarely needs updating. The stale-reference refresher relies on getter-like, side-effect-free reconstruction (Section III-C2), which may need per-project attention but follows the same low-maintenance pattern. Adapters are discovered at runtime via Java's Service Provider Interface (SPI) [53] and loaded with ServiceLoader, and the stale-reference refresher is implemented as a Java agent via bytecode instrumentation (ASM) [54], [55].

IV. Evaluation

This section evaluates whether Palingenesis makes graceful restart testing practical for real distributed systems by answering the following research questions:

- **RQ1 (Bug Finding Effectiveness and Scope):** How good is Palingenesis at finding restart bugs?
- **RQ2 (Transformation Effectiveness and Reproducibility):** How effective and reproducible is the restart test transformation, and how sensitive to the choice of LLM?
- **RQ3 (Restart-Point Selection Validation):** How well does LLM-based restart injection compare to baselines?
- **RQ4 (Ablation Studies on Palingenesis Components):** How does the unified restart API affect transformation, and how does the reference refresher affect bug detection?
- **RQ5 (Bug Studies):** What are the bug root causes?
- **RQ6 (Execution Overhead):** What is the execution overhead of the transformed restart tests?

A. Evaluation Setup

We evaluate Palingenesis on seven open-source distributed systems shown in Table I. For each system, we identify MiniCluster-style test methods and retain those that pass *without* injected restarts. We run all experiments on CloudLab [56] d430 (2×8-core CPU, 64 GB RAM), Ubuntu 22.

Project	Version (commit)	# Cluster test methods
Accumulo	2.1.4 (e557fd9)	438
Cassandra	5.0.6 (4cba279)	518
Flink	2.2.0 (5a33689)	115
HBase	2.6.3 (01ac2f5)	903
HDFS	3.3.5 (706d882)	1759
YARN	3.3.5 (706d882)	160
ZooKeeper	3.9.4 (7246445)	566

TABLE I: Summary of the evaluated Projects.

LLM-based transformer configuration. The transformer runs the LLM with temperature 0.2 and at most 3 compilation-repair rounds per transformation, similar to prior LLM-based test transformation [42], [43]. We evaluate three LLMs: Claude Opus 4.5 [57], GPT-5.2 [58], and Gemini 3 Pro [59], repeating each transformation 3 times per model to account for nondeterminism. All questions except RQ2 use Claude Opus 4.5 as the default model. RQ2 reports per-model effectiveness.

Restart baselines. We compare Palingenesis against two baselines. Budget-matched *Random* samples restart points uniformly from syntactically valid statement boundaries in the original test AST (via JavaParser), matches the LLM transformer's restart points per test, and repeats sampling with 3 seeds. *Heuristic-based restart (HBR)* injects after common wait/synchronization helpers (e.g., waitFor*, await*, join*, sleep) and at cluster lifecycle boundaries (after startup, before shutdown), so its restart-point count varies by test. Unlike Palingenesis, HBR is only syntax-aware: it would inject after waitForReplication(...) even if later state-changing operations define the restart-sensitive phase. Both baselines use Palingenesis's restart runtime (adapters and refresher).

External reference. We also run CrashFuzz [13], a state-of-the-art crash/reboot testing technique, on HBase, HDFS, and ZooKeeper, the three projects from its original paper, with the same setup and workloads for 48 hours per project. We use this comparison as an external reference rather than a direct baseline: CrashFuzz targets crash/recovery semantics, not the planned graceful shutdown and rejoin procedure.

B. RQ1: How Many Bugs Does Palingenesis Find?

RQ1 evaluates whether Palingenesis can find restart bugs by transforming existing cluster tests into restart tests with controlled restart injections. Table III summarizes execution and failure-inspection results. Failures are reported as *failure groups* ("Failure reports") because one root cause can recur across executions. Across the seven projects, Palingenesis transforms 4459 cluster test methods into compilable restart tests and executes 18316 restart runs. The total depends on the number of existing cluster tests and injected restart points per method. HBase and HDFS contribute the most executions (7318 and 5420), reflecting many cluster test methods (903 and 1759), many injected restart points per method (8.10 and 3.08), richer node roles (e.g., NameNode, DataNode, JournalNode in HDFS; Master, RegionServer, ZooKeeperNode in HBase), and more complex test methods yielding more restart points.

As Table III shows, 652 of 18316 restart executions fail. Failure inspector filters them to 259 and groups these into 26 failure reports, keeping the inspection burden practical. In

Issue	Phase	Symptom	Buggy Loc.	CrashFuzz	Random	HBR	Palingenesis	Exec. (s)
HBASE-29804	Shutdown	Resource not unregistered during shutdown	System code	✗	✗	✓	✓	24.44
HDFS-17876	Shutdown	Node shutdown missing resource checker	System code	✗	✗	✓	✓	34.13
ZOOKEEPER-5012	Shutdown	Node shutdown missing resource checker	System code	✗	✗	✓	✓	12.89
ACCUMULO-6071	Reboot	Missing node role during reboot	Cluster infra	-	✗	✗	✓	26.30
ACCUMULO-6070	Reboot	No resource-access checker during reboot	Cluster infra	-	✓	✗	✓	21.16
HBASE-29857	Reboot	Improper resource-access checker in reboot	System code	✗	✗	✗	✓	34.36
HDFS-17870	Reboot	Missing field check in reboot initialization	Cluster infra	✗	✗	✗	✓	37.95
ZOOKEEPER-5013	Reboot	Node constructor missing field initialization	Cluster infra	✗	✗	✗	✓	35.08
ACCUMULO-6050	Recovery	Incorrect waiting logic for recovery	Test code	-	✗	✓	✓	77.55
CASSANDRA-21128	Recovery	No checker for resource access after restart	System code	-	✗	✗	✓	17.53
CASSANDRA-21097	Recovery	Inconsistent data-format assumption after restart	Cluster infra	-	✗	✗	✓	45.28
CASSANDRA-21096	Recovery	No checker for resource access after restart	Test code	-	✗	✗	✓	20.28
FLINK-38870	Recovery	Illegal application status after restart	System code	-	✓	✗	✓	10.45
HBASE-29805	Recovery	Resource access during recovery causes crash	System code	✓	✗	✗	✓	25.80
HDFS-17860	Recovery	No checker for closed resource access	System code	✗	✗	✗	✓	43.53
HDFS-17863	Recovery	Data not visible after restart	System code	✓	✓	✗	✓	45.40
ZOOKEEPER-5011	Recovery	No checker for closed resource access	System code	✗	✓	✗	✓	11.82

TABLE II: Restart bug inventory by phase, symptom, and buggy location. ✓: found by the technique; ✗: not found. CrashFuzz is evaluated only on its three originally supported projects (HBase, HDFS, ZooKeeper). Column "Exec. (s)": average execution time (seconds) of the transformed restart test(s) exposing the bug, averaged over reproducing runs.

Project	Restart tests	Restart execs	Failing execs	Fail. after filtering	Failure reports	Bugs	False alarms
Accumulo	438	1580	71	28	4	3	1
Cassandra	518	2162	91	36	4	3	1
Flink	115	244	22	8	3	1	2
HBase	903	7318	182	73	4	3	1
HDFS	1759	5420	211	84	5	4	1
YARN	160	248	18	8	2	0	2
ZooKeeper	566	1344	57	22	4	3	1
SUM	**4459**	**18316**	**652**	**259**	**26**	**17**	**9**

TABLE III: Palingenesis main restart testing results.

total, Palingenesis finds 17 distinct restart bugs across the seven projects, appearing in all projects except YARN. Table II summarizes them by project, failure phase, symptom, and root cause. The inspection takes roughly half an hour per report, or 1.7 bugs per hour (17 bugs over 10 hours), with failure inspector and LLM assistance. Developers with deeper system familiarity may inspect each group faster. To validate these inspection results, an additional LLM-based failure analyzer labeled each of the 26 failure reports as a real restart bug or a false alarm, using a fixed prompt and a fresh context per report. The LLM analyzer received the restartability contract of Section III-A and the same materials as the human inspector, with the first author's labels and issue identifiers removed. The LLM analyzer disagreed with the first author's labels on 6 of the 26 reports, in each case labeling a report a false alarm that the first author labeled a real restart bug. The resulting agreement, measured as Cohen's kappa, is 0.56. Notably, developers have confirmed four of the six disagreed reports as real bugs upstream (one already fixed), which externally supports the first author's labels and suggests that current LLMs remain conservative in failure inspection. Finally, a second author reviewed, double-checked, and agreed on *all* 26 labels, further validating the results.

To better understand the scope of the discovered bugs, we also compare against CrashFuzz, a state-of-the-art crash/reboot testing technique, on the three shared projects (ZooKeeper, HDFS, and HBase). CrashFuzz finds 2 overlapping bugs in HBase and HDFS but misses 8 bugs that Palingenesis finds: 3 (HBASE-29804, HDFS-17876, and ZOOKEEPER-5012) occur only on graceful shutdown paths outside its crash/reboot scope, 2 (HDFS-17870 and ZOOKEEPER-5013) are exposed only in the projects' existing testing cluster infrastructure, and the rest were conceptually reachable but not explored in the 48-hour campaign. This highlights the complementary value of Palingenesis's developer-written workloads and oracles, consistent with CrashFuzz's own note that workload coverage is a key limitation [13]. CrashFuzz also finds 3 additional crash-only bugs outside this paper's graceful-restart target, so we do not include them in our evaluation.

Beyond the 17 reported bugs, Palingenesis also identifies 9 failure groups caused not by restart bugs but by original test assumptions that do not survive restart. Each of these groups violates one of the three unsupported-assumption classes defined by the restartability contract in Section III-A: 4 groups are from *uninterrupted execution*, 2 from *transient in-memory state*, and 3 from *strict immediate availability*. The *uninterrupted-execution* groups reuse a pre-restart handle that the graceful shutdown invalidates: Cassandra reuses a client session and throws NoHostAvailableException after the transport stops, ZooKeeper reuses a pre-restart session and sees ConnectionLoss/SessionExpired, YARN reuses a stale AMRMToken/app-attempt after the ResourceManager restarts, and a Flink test built without HA leadership control relies on pre-restart JobManager configuration. The *transient-in-memory-state* groups check non-durable state a restart legitimately resets: an HBase test expects a region-split state not preserved across restart (HBase correctly refuses to split), and a YARN placement-constraint test reads scheduler-local state. The *strict-immediate-availability* groups assert before recovery completes: Accumulo and HDFS tests require tablet servers to be registered or the mini-cluster to be back the moment the restart call returns, and a Flink test enforces a strict availability oracle. These readiness assertions fail hard, surfacing as fatal failures rather than silent timeouts, so they are not removed by the timeout filter of Section III-D1 and instead reach inspection. These are application-level oracle mismatches in the original tests, distinct from the Palingenesis exceptions and the pure timeouts filtered before grouping (Section III-D1): unlike filtered timeouts, whose logs lack fatal signals, these false-alarm groups carry a fatal signal and

reach inspection, where they are judged false alarms against the restartability contract. These suggest that applying Palingenesis to a new system can include a one-time calibration to remove such false alarms from restart testing results: a developer applies the decision procedure of Section III-A to identify these oracle patterns, and adds them to a per-system filter list.

C. RQ2: Transformation Effectiveness and Reproducibility

RQ2 evaluates the effectiveness of the LLM-based transformer (Section III-B) in turning cluster tests into compilable restart tests. For each model, we run transformation 3 times and report (i) average LLM API latency per attempt, (ii) first-attempt transformation success rate, (iii) average API cost per attempt, and (iv) the total number of selected restart points. To isolate code-generation capability, we report *single-iteration* results (no compilation-repair loop, Section III-B4) on the same cluster test set for all models.

Table V summarizes the results for the three LLMs. All three achieve high first-attempt compilation success (94.2–96.5%): the unified restart API narrows the valid output space, making the task robust under Palingenesis's prompt structure and template constraints and keeping the differences among strong models relatively small. Claude Opus 4.5 has the best overall profile in our setting: the highest success rate (96.5%), the lowest average API latency (20.5 seconds), and the most injected restart executions, so we use it as the default model for the other research questions. Gemini 3 Pro has the lowest API cost (0.12 USD) but the highest latency (86.5 seconds) and the lowest success rate (94.2%). In practice, the model mainly needs to (i) select restart points consistent with test intent and (ii) apply Palingenesis's unified restart API. This matches the later RQ4 ablation: removing the unified API sharply reduces first-attempt success (Table VI), showing that the unified API is a key enabler of reliable transformation.

We also measure rerun stability of restart-point selection, the key source of nondeterminism. Table VII reports two views. Restart-point recurrence across the three repeated runs is high (Table VIIa): for Claude Opus 4.5, 91.6% of union restart points recur in all three runs and 98.3% in at least two runs. GPT-5.2 and Gemini 3 Pro show similarly high recurrence. Failure reports and bugs are similarly stable (Table VIIb): 20 of 26 failure reports recur in all three runs and the remaining 6 in two. For bugs, 14 of 17 recur in all three runs and the remaining 3 in two. Under the fixed prompt structure, unified restart API, and compilation validation, LLM-guided placement is not perfectly deterministic but highly reproducible.

D. RQ3: Restart-Point Selection Validation

RQ3 isolates the contribution of restart-point selection within Palingenesis by comparing Palingenesis against two baselines: Heuristic-based restart (HBR) and Random.

Table IV reports results for HBR (left) and Random (middle, averaged over 3 seeds). Across all projects, HBR finds 4 bugs (from 15 failure reports) and Random finds 4 (union), versus 17 bugs for LLM-guided selection (Table II). Both baselines also report fewer failure reports (15 and 17.67 vs. 26). Triggering restart bugs thus requires narrow injection points that demand semantic reasoning about test intent and phase boundaries: neither baseline finds HBASE-29805, which requires a restart between enabling and submitting a HBase region split.

We further estimate the chance that a uniformly random choice discovers each bug: the size of its triggering window (the statement boundaries where a restart triggers the bug) divided by all candidate boundaries in that test. Averaged over the 17 bugs, this probability is only 7.7%, explaining why budget-matched random sampling often misses bugs LLM-guided selection targets.

E. RQ4: Ablation Studies on Palingenesis Components

(a) Ablation: no unified restart API. We remove the unified restart-API specification and template constraints from the prompt, so the LLM must synthesize project-specific restart logic from code context; Table VI reports the results for the same three LLMs. Removing the unified restart API sharply reduces transformer effectiveness: the compilation success rate drops from 94.2%–96.5% (Table V) to 38.4%–42.1% (Table VI), and average latency and cost increase accordingly. This expands the space of valid outputs and increases code-generation errors and compilation failures.

(b) Ablation: no stale-reference refresher. We disable the stale-reference refresher while keeping other components unchanged. Restart testing quality degrades substantially: 73.1% (13392 out of 18316) of restart executions now fail, versus 3.6% with the refresher (Table III). Failure reports rise sharply from 26 to 550 (Table IV, right), and most are false positives from stale in-memory references, typically NullPointerException or IllegalStateException on cluster-derived objects captured before restart. More importantly, 9 of the 17 bugs become masked without the refresher, and only 8 bugs remain detectable. These remaining bugs are all graceful shutdown protocol violations and reboot failures that surface immediately during the restart process (see Section IV-F for details). Without the refresher, Palingenesis cannot reliably validate post-restart recovery correctness.

F. RQ5: Bug Study: When and Why Restarts Matter

We analyze the 17 bugs in Table II along two dimensions: when the restart failure occurs and where the buggy code resides. 3 bugs occur during shutdown, 5 during reboot/re-initialization, and 9 during post-restart recovery. By location, 10 are in system code, 5 in cluster infrastructure, and 2 in test code. These bugs often involve missing resource checks or cleanup, inconsistent post-restart state assumptions, or insufficient validation around post-restart operations. We highlight one representative bug from each category.

ZOOKEEPER-5012 (Shutdown protocol violation). Palingenesis uncovers a buggy condition in ZooKeeper's shutdown protocol: when QuorumPeer.shutdown() runs before the zkDb field is initialized, the path null-checks several fields but unconditionally calls zkDb.close(), causing a NullPointerException.

HDFS-17870 (Reboot failure to rejoin cluster). Palingenesis detects a defect in the HDFS MiniDFSCluster infrastructure

Project	Heuristic-based restart			Random (avg over 3 seeds)		No refresher	
	# Restart execs	# Failure reports	# Bugs	# Failure reports	# Bugs	# Failure reports	# Bugs
Accumulo	1228	3	1	5.33	0.67	39	2
Cassandra	1312	1	0	2.33	0.00	23	0
Flink	391	2	0	2.33	0.67	16	0
HBase	2881	2	1	1.33	0.0	229	2
HDFS	4772	3	1	2.67	0.67	178	2
YARN	495	1	0	1.33	0.0	14	0
ZooKeeper	1629	3	1	2.33	0.67	51	2
SUM	**12708**	**15**	**4**	**17.67**	**4 (union)**	**550**	**8**

TABLE IV: Restart test results for Heuristic-based restart (RQ3), Random baseline (RQ3), and ablation without stale-reference refresher (RQ4b). Random reports the mean over 3 seeds and the sum of bugs found across all seeds (union).

Model	Avg time per test (s)	Avg cost per test (USD)	Success rate	Total restart positions
Claude Opus 4.5	**20.5**	0.21	**96.5%**	**18316**
GPT-5.2	41.2	0.19	95.1%	17521
Gemini 3 Pro	86.5	**0.12**	94.2%	16892

TABLE V: LLM Comparison. Bold highlights the best.

Model	Avg time per test (s)	Avg cost per test (USD)	Success rate
Claude Opus 4.5	32.5	0.28	39.6%
GPT-5.2	53.2	0.24	42.1%
Gemini 3 Pro	106.4	0.19	38.4%

TABLE VI: RQ4a ablation (no unified restart APIs).

itself: restartDataNode() can trigger an out-of-bounds exception when the DataNode index exceeds the storageCapacities array length, so the node fails to reboot and rejoin the cluster.

HBASE-29805 (Recovery window race condition). Palingenesis identifies a bug where restarting a RegionServer in the brief window between enabling a region split and submitting the split procedure leaves the region OPEN with a temporarily null server location, causing a NullPointerException in the split path, which reads the location without a null check.

G. RQ6: Performance Overhead of Transformed restart tests

We evaluate the overhead of the transformed restart tests by comparing their average execution times with the original cluster tests. Figure 8 summarizes the per-project averages. Overall, the transformed restart tests incur an average execution overhead of 2.27x compared to the original cluster tests, raising the average per-test time from 18.28s to 41.47s. The per-project slowdown ranges from 1.16x for Flink to 4.68x for ZooKeeper. This overhead comes from the restart operations: stopping and starting components, waiting for them to reinitialize, and validating that they rejoin the cluster. Table II reports the execution time for each test that exposes a restart bug.

V. Discussion and Limitations

Refresher limitations beyond object identity. The stale-reference refresher makes in-process restart testing practical but cannot eliminate all restart-induced noise. RQ4b (Section IV-E) shows that without it most restart executions fail early on stale references and 9 of the 17 bugs become undetectable. Three limitations remain: it only refreshes references flowing through instrumented restart test bytecode, so references escaping to uninstrumented paths (external callbacks, lambda-generated classes) may stay stale; it excludes non-reconstructible resources such as I/O streams; and non-deterministic derivation chains may fail replay, raising an explicit refresh exception.

Model	All 3	Exactly 2	Exactly 1	Union	All 3 (%)	≥2
Claude Opus 4.5	16780	1218	318	18316	91.6%	98.3%
GPT-5.2	15690	1470	361	17521	89.6%	97.9%
Gemini 3 Pro	14910	1554	428	16892	88.3%	97.5%

(a) Restart-point recurrence.

Artifact	All 3	Exactly 2	Exactly 1	Union	All 3 (%)	≥2
Failure reports	20	6	0	26	76.9%	100%
Bug groups	14	3	0	17	82.4%	100%

(b) Failure and bug finding recurrence.

TABLE VII: Recurrence across the three repeated runs in RQ2.

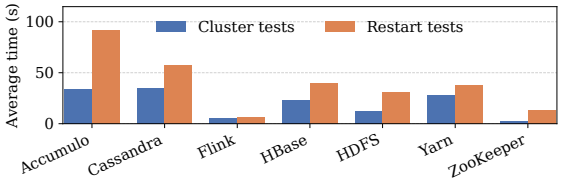


Fig. 8: Average execution time of original cluster tests and transformed restart tests across seven projects.

Oracle brittleness under restart. Oracle preservation is a principled design choice: a graceful restart models a planned shutdown and reboot, so after recovery stabilizes the system should satisfy the same safety invariants as in steady-state execution. Table III supports this design, with 17664 of 18316 restart runs passing under the original tests. Still, unchanged oracles have two limits. They can miss *silent failures* that violate intended invariants without triggering existing assertions or causing crashes, and they can raise false alarms when tests assume uninterrupted execution, immediate availability, or specific timing. We do not relax oracles with retries or eventual-consistency checks, because such relaxation can mask restart-related safety violations. Instead, Palingenesis preserves the original oracle outcomes and leaves false alarms visible for manual inspection. The second limit is exactly a violation of the restartability contract (Section III-A): a false alarm occurs when a restart is injected where the original test never intended to support one. Rather than weakening oracles to hide such failures, we keep them and rely on the contract to distinguish them from bugs during inspection.

Detection scope and false negatives. Palingenesis is

bounded by the coverage of the cluster tests it transforms: it can only exercise restart on code paths that the original cluster tests already reach, so bugs on uncovered paths yield false negatives. It also inherits the original tests' detection power, augmented with crash and hang detection, and so reports a restart bug only when a restart trips an existing oracle, a crash, or a hang. Consequently, it can miss *silent failures* that violate intended invariants without triggering an assertion or crash.

When to run restart testing given overhead. The transformed restart tests are slower than the original cluster tests because each execution includes an explicit shutdown, restart, and post-restart stabilization phase (Section IV-G). In practice, this makes Palingenesis a good fit for scheduled validation (e.g., nightly or weekly runs) and for targeted pre-merge checks on restart-sensitive components, rather than a drop-in replacement for running the original cluster test suite on every commit. When budget is limited, a natural strategy is to sample restart scenarios by prioritizing tests that cover changed code paths and restart windows most likely to expose restart bugs.

Future work. Several directions follow from Palingenesis's current scope. First, test generation could complement transformation to cover paths existing cluster tests never reach. We currently transform existing tests to preserve realistic workloads and developer-written oracles. Second, we used frontier models for stronger code generation. Open-source or fine-tuned LLMs could reduce cost and transformation nondeterminism. Third, each run activates a single restart point for cost and bug attribution, consistent with crash and reboot testing practice; combinations of restart points and consecutive restarts would expand the test matrix. Fourth, the contract's decision procedure could be automated via program analysis or LLMs.

VI. Related Work

Fault injection testing. Existing work has extensively explored fault injection for distributed systems. CrashFuzz [13] performs coverage-guided crash/reboot injection; DeMiner [27] predicts crash points from common-data-related operations; Legolas [28] exposes partial-failure bugs with inferred abstract states and fault hooks; DepState [30] targets phase-sensitive synchronization failures in relational DDBMSs; and Mallory [29] applies adaptive greybox fuzzing over node faults and network partitions. Like broader crash-centric fault-injection work [12], [22], [24]–[26], these systems target abrupt crashes, partial failures, or external-harness fault injection rather than planned graceful shutdown completion and rejoin semantics, and none transforms existing MiniCluster tests while preserving original workloads and oracles. Our work instead reuses those tests as realistic workloads with built-in oracles. The distinction is fundamental: a graceful restart exercises the *planned-termination* path, draining requests, running cleanup, persisting metadata, transferring leadership, and performing role-specific rejoin, whereas crash and reboot injection (e.g., CrashFuzz) skips this path and tests unplanned-restart recovery. The two are complementary: crash injection can miss bugs that occur only on the graceful shutdown and rejoin path.

Test transformation and LLM-based testing. Prior work transforms developer tests to explore under-tested behaviors [33]–[36], [45], [46], [60], a style recent LLM-driven work describes as *test amplification* [61], [62]. Unlike test generation tools such as Randoop [63] and EvoSuite [64], we reuse the domain knowledge in existing cluster tests, preserving workloads and oracles across restarts. Recent LLMs support code understanding and transformation [38], [47]–[50], and agentic systems tackle software engineering tasks [39], [41], [51]. Palingenesis applies them to a constrained transformation problem (restart-relevant checkpoint selection), using a unified template API plus compilation-guided repair, aligning with findings that constrained workflows improve reliability [40]. LLM-based testing also improves developer tests at industrial scale [42], uses generation-validation-repair loops [43], and guides test creation with mutants [65].

VII. Conclusions

Graceful restarts are routine in production distributed systems, yet remain under-tested. This paper frames graceful restart testing as an automated transformation task and presents Palingenesis, which uses LLMs to transform MiniCluster-style cluster tests into restart tests while preserving the original workload and oracles, with runtime support to execute them reliably under restarts. Across seven evaluated systems, Palingenesis transforms 4459 cluster tests and runs 18316 restart executions, uncovering 17 restart bugs: 6 fixed, 5 more confirmed, and the rest are pending.

Acknowledgements

We thank Tianyin Xu, Reyhaneh Jabbarvand, and Shizhuo Zhang for numerous discussions about testing distributed systems. We also thank the anonymous reviewers and the program committee chairs for their valuable comments and suggestions, which substantially improved this paper.

References

[1] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *EuroSys*, 2015.

[2] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *EuroSys*, 2013.

[3] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *NSDI*, 2011.

[4] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*, 2016.

[5] "Kubernetes documentation: Rolling update deployment," accessed: 2026-01-13. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-update-deployment

[6] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan, "Understanding and detecting software upgrade failures in distributed systems," in *SOSP*, 2021.

[7] "Kubernetes documentation: Pod termination," accessed: 2026-01-13. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-termination

[8] "HDFS rolling upgrade," accessed: 2026-01-13. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsRollingUpgrade.html

[9] "ZooKeeper guide," accessed: 2026-01-13. [Online]. Available: https://zookeeper.apache.org/doc/current/zookeeperAdmin.html

[10] A. Wolski and K. Laiho, *Rolling upgrades for continuous services*, 2004.

[11] J. Dean and L. A. Barroso, "The tail at scale," *ACM*, vol. 56, 2013.

[12] 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 *SOSP*, 2019.

[13] Y. Gao, W. Dou, D. Wang, W. Feng, J. Wei, H. Zhong, and T. Huang, "Coverage guided fault injection for cloud systems," in *ICSE*, 2023.

[14] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC*, 2014.

[15] L. Lamport, "The part-time parliament," *TCS*, 1998.

[16] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *DSN*, 2011.

[17] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *USENIX ATC*, 2010.

[18] M. Burrows, "The Chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," in *OSDI*, 2013.

[20] R. Chandra, R. Lefever, M. Cukier, and W. Sanders, "Loki: a state-driven fault injector for distributed systems," in *DSN*, 2000.

[21] R. Chandra, R. Lefever, K. Joshi, M. Cukier, and W. Sanders, "A global-state-triggered fault injector for distributed system evaluation," *TPDS*, 2004.

[22] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *Software*, 2016.

[23] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, and R. Padhye, "Service-level fault injection testing," in *SoCC*, 2021.

[24] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *SIGMOD*, 2015.

[25] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *SoCC*, 2014.

[26] 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 *OSDI*, 2014.

[27] Y. Gao, D. Wang, Q. Dai, W. Dou, and J. Wei, "Common data guided crash injection for cloud systems," in *ICSE Companion*, 2022.

[28] H. Wu, J. Pan, and P. Huang, "Efficient exposure of partial failure bugs in distributed systems with inferred abstract states," in *NSDI*, 2024.

[29] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Greybox fuzzing of distributed systems," in *CCS*, 2023.

[30] C. Fang, J. Liang, Z. Wu, J. Fu, Z. Jia, C. Huang, Y. Jiang, and S. Li, "DepState: Detecting synchronization failure bugs in distributed database management systems," in *ISSTA*, 2025.

[31] "HDFS MiniDFSCluster," accessed: 2026-01-13. [Online]. Available: https://github.com/apache/hadoop/blob/trunk/hadoop-hdfs-project/hadoop-hdfs/src/test/java/org/apache/hadoop/hdfs/MiniDFSCluster.java

[32] "HBase HBaseTestingUtility," accessed: 2026-01-13. [Online]. Available: https://github.com/apache/hbase/blob/master/hbase-testing-util/src/main/java/org/apache/hadoop/hbase/HBaseTestingUtility.java

[33] X. Sun, R. Cheng, J. Chen, E. Ang, O. Legunsen, and T. Xu, "Testing configuration changes in context to prevent production failures," in *OSDI*, 2020.

[34] S. Wang, X. Lian, D. Marinov, and T. Xu, "Test selection for unified regression testing," in *ICSE*, 2023.

[35] S. Ma, F. Zhou, M. D. Bond, and Y. Wang, "Finding heterogeneous-unsafe configuration parameters in cloud systems," in *EuroSys*, 2021.

[36] J. Li, S. Li, K. Li, F. Luo, H. Yu, S. Li, and X. Li, "ECFuzz: Effective configuration fuzzing for large-scale systems," in *ICSE*, 2024.

[37] F. Ma, Y. Chen, Y. Zhou, Z. Yan, H. Sun, and Y. Jiang, "Finding metadata inconsistencies in distributed file systems via {Cross-Node} operation modeling," in *USENIX Security*, 2025.

[38] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[39] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," 2024.

[40] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying LLM-based software engineering agents," in *FSE*, 2025.

[41] Z. Nan, Z. Guo, K. Liu, and X. Xia, "Test intention guided LLM-based unit test generation," in *ICSE*, 2025.

[42] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *FSE Companion*, 2024.

[43] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "ChatUniTest: A framework for LLM-based test generation," in *FSE Companion*, 2024.

[44] R. Cheng, L. Zhang, D. Marinov, and T. Xu, "Test-case prioritization for configuration testing," in *ISSTA*, 2021.

[45] S. Wang, X. Lian, Q. Li, D. Marinov, and T. Xu, "Ctest4J: A practical configuration testing framework for Java," in *FSE*, 2024.

[46] S. Wang, H. Wang, D. Marinov, T. Xu, and Y. Zhang, "ConfFuzz: Parameter-aware greybox fuzzing for configurable cloud systems," in *ASE*, 2026.

[47] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," 2020.

[48] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," 2021.

[49] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A generative model for code infilling and synthesis," 2023.

[50] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, T. Hubert, P. Choy, C. de Masson d'Autume, I. Babuschkin, X. Chen, P.-S. Huang, J. Welbl, S. Gowal, A. Cherepanov, J. Molloy, D. J. Mankowitz, E. Sutherland Robson, P. Kohli, N. de Freitas, K. Kavukcuoglu, and O. Vinyals, "Competition-level code generation with AlphaCode," *Science*, 2022.

[51] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "SWE-bench: Can language models resolve real-world GitHub issues?" 2024.

[52] The Apache Software Foundation, "Maven Surefire Plugin," 2025, accessed: 2026-01-25. [Online]. Available: https://maven.apache.org/surefire/maven-surefire-plugin/index.html

[53] Oracle, "Class ServiceLoader," Java Platform, Standard Edition Documentation, 2025, accessed: 2026-01-25. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ServiceLoader.html

[54] ——, "Package Java.lang.instrument," Java Platform, Standard Edition Documentation, 2025, accessed: 2026-01-25. [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/api/java.instrument/java/lang/instrument/package-summary.html

[55] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," 2002.

[56] 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 *ATC*, 2019.

[57] Anthropic, "Claude opus 4.5," 2026, accessed: 2026-01-25. [Online]. Available: https://www.anthropic.com/claude/opus

[58] OpenAI, "Openai: GPT-5.2," 2026, accessed: 2026-01-25. [Online]. Available: https://platform.openai.com/docs/models/gpt-5.2

[59] Google, "Google: Gemini models," 2026, accessed: 2026-01-25. [Online]. Available: https://ai.google.dev/gemini-api/docs/models/gemini

[60] S. Wang, "Unified automated reliability testing for configurable cloud systems," Ph.D. dissertation, University of Illinois Urbana-Champaign, 2026.

[61] R. Nooyens, J. Besjes, T. Bardakci, M. Beyazit, and S. Demeyer, "Test amplification for REST APIs via single and multi-agent LLM systems," 2025.

[62] J. Besjes, R. Nooyens, T. Bardakci, M. Beyazit, and S. Demeyer, "Agentic LLMs for REST API test amplification: A comparative study across cloud applications," 2025.

[63] C. Pacheco and M. D. Ernst, “Randoop: feedback-directed random testing for Java,” in *OOPSLA*, 2007.
[64] G. Fraser and A. Arcuri, “EvoSuite: automatic test suite generation for object-oriented software,” in *ESEC/FSE*, 2011.
[65] M. Harman, J. Ritchey, I. Harper, S. Sengupta, K. Mao, A. Gulati, C. Foster, and H. Robert, “Mutation-guided LLM-based test generation at meta,” in *FSE Companion*, 2025.