RESEARCH-ARTICLE

Once Bitten, Still Shy: Can We Prevent Cloud Systems from Repeating Their Mistakes?

DIMAS SHIDQI PARIKESIT, University of Virginia, Charlottesville, VA, United States

CHANG LOU, University of Virginia, Charlottesville, VA, United States

Published: 17 November 2025

HotNets '25: 24th ACM Workshop on Hot Topics in Networks
November 17 - 18, 2025
MD, College Park, USA

Conference Sponsors:
SIGCOMM

https://doi.org/10.1145/3772356.3772385
ISBN: 9798400722806

Once Bitten, Still Shy: Can We Prevent Cloud Systems from Repeating Their Mistakes?

Dimas Parikesit
University of Virginia
Charlottesville, VA, USA

Chang Lou
University of Virginia
Charlottesville, VA, USA

Abstract

Cloud systems constantly experience changes. Unfortunately, these changes often introduce regression failures, breaking the same features or functionalities repeatedly. Such failures disrupt cloud availability and waste developers' efforts in re-investigating similar incidents. In this position paper, we argue that regression failures can be effectively prevented by enforcing *low-level semantics*, a new class of intermediate rules empirically inferred from past incidents, yet capable of offering partial correctness guarantees. Our experience shows that such rules are valuable to strengthen system correctness guarantees and expose new bugs.

CCS Concepts

• **Computer systems organization** → **Reliability**; • **Software and its engineering** → **Software verification and validation**.

Keywords

Cloud systems; Regression failures; Large language models; Symbolic execution

ACM Reference Format:
Dimas Parikesit and Chang Lou. 2025. Once Bitten, Still Shy: Can We Prevent Cloud Systems from Repeating Their Mistakes? . In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3772356.3772385

1 Introduction

Cloud systems experience constant changes. To support dynamic workloads and customer requests, developers continuously submit updates to add new features, adjust configurations, fix bugs, *etc.* On a typical workday, Google Cloud Platform sees around 16,000 changes committed by developers [51], not even including those submitted by automated systems. These updates unavoidably introduce new failures and hurt cloud reliability.

Unfortunately, many of these are regression failures [46]–issues that have been fixed by developers but are broken again due to changes, as shown in Figure 1. In practice, developers are frustrated to find themselves dealing with the same kinds of issues over and over again [1, 2, 10–12, 14, 15]. Those wasted efforts and computing resources could have been saved.

Meanwhile, *isn't testing meant to catch and prevent these very issues?* Indeed, developers often add regression tests after failures happen. However, their efforts did not successfully transform into guarantees. A recent study [44] shows that 68% of the studied failures violate old semantics, which have existed since the first major stable release of the system. Tests often focus on specific bugs rather than the underlying deeper causes. Consequently, similar issues may still be introduced.

Thus, we need a more reliable approach to guarantee that regression failures would not occur. Verifying software systems [22, 30, 34, 36, 39, 40, 48, 55, 61–63] is a tempting approach. Nevertheless, cloud systems usually have a very large code base containing tens of thousands of lines of code with complicated interactions between components. Today, it remains impractical to fully verify large production cloud systems.

In this position paper, we ask this question: *what would it take to guarantee that after a failure is addressed, similar root causes can never reappear in a modern cloud codebase?* To answer this question, we first perform a study of regression failures across popular distributed systems. We find that the recurrences often stem from violations of low-level semantics–hidden, implementation-centric rules that developers implicitly rely on. These semantics are expressive enough to capture the correctness expectations of the high-level system properties, yet grounded in the observable low-level behavior of the code implementation. While formally verifying the entire system is costly and often impractical, low-level semantics offer an ideal middle ground—providing meaningful guarantees with significantly lower overhead.

Manually authoring these rules is certainly feasible, meanwhile, the workflow is painfully slow and notoriously brittle.

HotNets '25, College Park, MD, USA

ACM ISBN 979-8-4007-2280-6/25/11
https://doi.org/10.1145/3772356.3772385

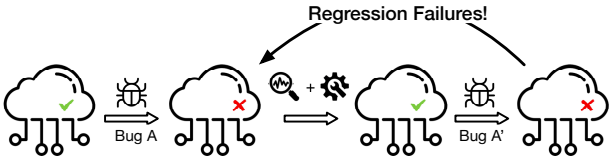

Figure 1: Regression failures in cloud systems.

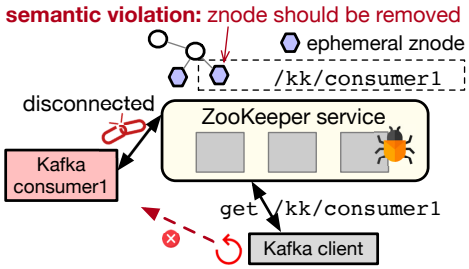

Figure 2: A real-world failure example in ZooKeeper.

Each rule must first be reverse-engineered from scattered issue tickets, code comments, and developer intuition. Worse, crafting high-fidelity rules and encoding them in a form that a checker or verification pass can consume is a fragile, labor-intensive process, which discourages comprehensive coverage and leaves gaps exactly where reliability matters most.

We envision a new cloud system development workflow where every failure, once fixed, automatically becomes an executable contract that shields the system from ever repeating the same mistake. To spark that conversation, we have built LISA, a lightweight prototype that automatically (i) extracts candidate semantics from LLM output, (ii) converts them into executable assertions, and (iii) enforces those assertions in CI/CD pipelines. Even in its current form, LISA uncovered two previously unknown, community-confirmed bugs in the latest releases of HBase and HDFS. We conclude by outlining the open research questions that must be answered to turn this vision into production reality.

2 Background and Motivation

2.1 Why Regression Failures Happen?

A central motivation for this work is understanding why cloud developers often repeat the same mistake during the development process. To shed lights into this question, we collect and analyze 16 regression cases from widely used cloud systems, including ZooKeeper, HDFS, HBase, and Cassandra. Each case includes one original bug and at least one new (regression) bugs. In total we study 34 software bugs. For each case, we examined developer discussions and code changes (implementation fix and newly added tests) to understand the lessons they learned after failures happened and how they affect future occurrences. Our study yields some interesting findings.

```
private void pRequest2TxnCreate(..){
  ..
  validateCreateRequest(path,
    createMode, request, ttl);
  DataTree.createStat(hdr.getZxid(),
    hdr.getTime(), ephemeralOwner);
  ..
    if (session == null) {
    if (session == null
      || session.isClosing()){
      throw new KeeperException();
    }
```

ZK-1208

```
public boolean touchSession(..){
  SessionImpl s = sessionsById
    .get(sessionId);
  if (s == null) {
  if (s == null || s.isClosing()){
    return false;
  }
  ..
  set.sessions.add(s);
  return true;
}
```

(one year later)

ZK-1496

Figure 3: The same semantics violated in both incidents.

We use one feature from ZooKeeper, ephemeral nodes, as an example. Ephemeral nodes are temporary records that automatically disappear when the client session ends. This makes it a convenient function that allows applications to detect service availability without manual cleanup.

Despite its importance, this feature has been a recurring source of regression failures. According to our survey, this feature has been associated with 46 related bugs over the past 14 years. Figure 2 demonstrates one real-world incident [16]. In this incident, developers deployed Kafka, a popular streaming service in their cluster and used ephemeral nodes to register consumer addresses for Kafka. However, a concurrency bug in ZooKeeper allowed the creation of an ephemeral node on a closing session, resulting in stale data that persisted even after the session terminated. Clients continued to query the dead address, leading to system-wide errors. Developers patched the issue by adding a check to prevent node creation on closing sessions. However, a year later, a similar failure [17] occurred (Figure 3). Although the original bug was fixed, another execution path that was not covered by existing tests allowed ephemeral node creation on a closing session again. Again, this caused the whole kafka cluster to get stuck in zombie mode.

This case is a representative example that reflects the gap. When the first incident happened, developers had already learned that "an ephemeral node may be created on a closing session, which is problematic". They inspected the failure, traced it to a race in the `PrepRequestProcessor`, and inserted the guard shown in Figure 3. In their heads–and often only in their heads–this became a semantic rule about ZooKeeper's

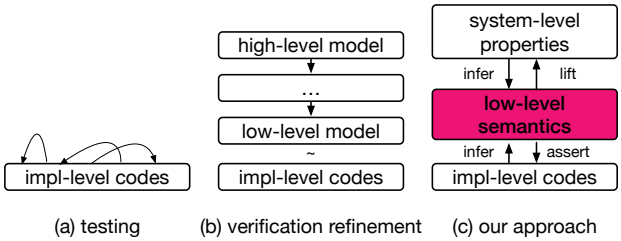

Figure 4: Comparison with alternative approaches.

correctness: "Reject the create request if the session is closing" Unfortunately, that rule was captured only informally (in Jira comments and code-review notes) and reflected in a single test that exercised the exact workload of the original bug. The knowledge gained during incident response is rarely promoted to a machine-checkable format in general scenarios. Such tests are brittle–they encode one execution trace rather than the underlying semantic invariant–and they run only under the limited workloads available in CI. When the system later evolved, another path in the request pipeline reached the same node creation logic without hitting the original guard, and the invariant was silently broken again.

This case reflects a recurring pattern in our study: regression failures often reappear because semantic assumptions are embedded in ad hoc tests rather than captured as explicit rules. Even when such rules are articulated, developers lack a systematic way to validate them across executions. Addressing this gap requires mechanisms to extract the implicit rules developers rely on during debugging and to enforce them continuously, regardless of workload.

2.2 What Can Be Done Differently?

We propose an approach to proactively prevent regression failures. We observe that many failures happen due to some paths being executed under special triggering conditions. Take the case in Figure 3 as an example. Both failures are triggered by omission to set the protect flag for creating data nodes. If developers enforce such rules after fixing the first issue, the failure reoccurrence could have be avoided. Thus, our key idea is to extract these rules from historical failures and systematically enforce the rules for updated system codes thus preventing future issues.

Comparison with alternative approaches Our approach occupies the middle ground between testing and verification as shown in Figure 4. Traditional testing validates concrete executions directly against the implementation: each regression test encodes one scenario, so coverage remains sparse and latent assumptions (*e.g.,* an ephemeral node is never created on a closing session) stay implicit. Consequently, fixes can regress as soon as code evolves outside the test scope.

On the other hand, refinement-based verification attacks the opposite extreme: developers first craft an abstract high-level model of the system, then prove via refinement that every implementation path preserves the model's properties. This yields powerful guarantees, but at the cost of heavyweight specifications, manual proof effort, and proofs that must be rebuilt after each non-trivial change—barriers that have limited uptake in large cloud codebases.

In contrast, we automatically infer low-level semantics from past bug fixes, then assert these semantics with symbolic execution across the code base. These semantics are expressive enough to capture the root cause of clustered regressions, yet concrete enough to be checked mechanically, providing broader coverage than tests and lower overhead than refinement.

A potential concern is that many distributed system bugs stem from multi-components interactions, making formal reasoning difficult to scale. Fortunately, our study suggests that the semantics developers care about do not always require reasoning over fully distributed or global states. In many cases, we can enforce correctness using localized semantic rules. For example, in the ephemeral node case, although the underlying semantics span request handling, session management, and data storage, the inferred local rules alone can already provide strong guarantees and successfully expose a large number of regression failures.

Challenges A central challenge is identifying the appropriate level of abstraction for low-level semantics. If the semantics are too high-level, they become difficult to assert and monitor directly in the system. If the semantics are too low-level, they risk being overly tied to syntactic details and may fail to capture meaningful behaviors or connect with higher-level properties. The semantics must strike a balance—serving as a bridge between code implementation and system properties defined in the specification, while remaining expressive enough to support diverse correctness expectations.

Another challenge is to support diverse types of system properties. System failures often reflect violations of nuanced, context-specific semantics. However, there is no uniform framework for extracting such semantics, as different failures may encode different assumptions or guarantees. Recovering invariants from historical failures requires careful interpretation of system context and developer intent.

After semantics are extracted, systematically checking them across a large, evolving cloud codebase is nontrivial. Cloud systems often contain complex control paths and interactions, making it difficult to ensure that all relevant execution paths are exercised and verified against the intended semantics.

Opportunities Two observations motivated this project. First, modern cloud systems have culminated in a rich repository of semantic resources through decades of iterative development and deployment. Modern cloud systems possess a significant volume of test cases—on average 1,309 files among

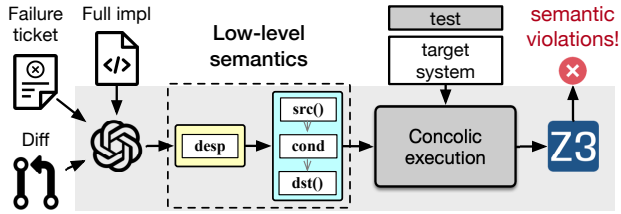

Figure 5: System workflow.

studied systems—with satisfactory code coverage. Historical failures are well-documented, with bug fixes, regression tests, and detailed discussions. These resources are invaluable for extracting insights to guide semantic checker construction. Second, systematically reasoning semantics in large systems software (*e.g.,* ZooKeeper, HDFS, Kubernetes) is made possible with advances in program analysis, formal methods, and machine learning [22, 24, 37, 42, 54, 65]. For example, symbolic execution has recently been applied to MySQL [37, 38], previously thought to be too complex for such technique.

3 System Design

We demonstrate the system workflow at a high level in Figure 5. The workflow starts with iterating over each failure ticket, collecting the buggy code version, developer discussion, code patch, and regression test, and feeding this bundle to an LLM. Based on the collected information, the LLM proposes likely low-level semantics as natural language descriptions. Our tool then employs a hybrid strategy of using static analysis assisted by LLMs to translate semantic descriptions into a machine-checkable format. At the end of the workflow, the system uses symbolic execution to assert these semantics across all reachable paths and confirms their correctness by feeding them into SMT solvers (*e.g.,* Z3).

3.1 Inferring and Translating Low-Level Semantics

Definition A low-level semantic includes two components. The first component is a concise description in natural language. The second component is a safety contract, where s is the target statement (or basic block) identified from a past bug fix and σ denotes the program state. Concretely, we restrict P,Q to conjunctions of implementation-local predicates of the form such as state relations ($v = c$) and resources ($handle.isOpen$).

For the ZooKeeper bug, the recovered rule is:

$$<session.isClosing == false> createEphemeralNode <>$$

To capture the implicit behavioral contracts embedded in production code, our system first infers low-level semantics from historical failures using large language models (LLMs). We designed our prompt based on the process of how experienced developers deal with failure cases. Given a past failure ticket consisting of the textual description, code diff, and the source code after the patch, we instruct the LLM to extract both high-level and low-level semantic relations in the modified logic. Listing 1 shows the current prompt we use.

```
You are an AI assistant that extracts violated low
    -level semantics from a past system failure.

You will receive three inputs:
   Failure description and developer discussion
   Code patch (the diff)
   Source code after the patch has been applied

Here are the steps you will take:
   1. Identify the root cause of this failure
   2. Identify the high-level semantics: a single
        concise statement describing the
   system-level behavioral change introduced by
        this pull request.
   3. Identify the low-level semantics: a single
        concise statement describing the
   implementation-local invariant that must hold
        so that a corresponding high-level
   property cannot be violated.
   4. Translate the low-level semantics into a
        checkable format:
   - one condition statement (predicates over
        concrete state and control-flow that needs
        to be checked)
   - one target statement (the code statement
        where the condition should be checked)
   5. Describe the reasoning for choosing those
        statements
   6. Repeat previous steps until all unique
        checks have been reasoned

   Output your answer in the exact format:
   {"high_level_semantics": "<description>",
    "low_level_semantics": {
      "description": "<concise_description>",
      "target_statement": "<code_text>",
      "condition_statement": "<predicates>", ...},
    "reasoning": "<summary>" ...}
```

Listing 1: LLM Prompt for low-level semantics inference

The inference operates in two phases. In the first phase, the LLM identifies the high-level semantics (*e.g.,* "Every ephemeral node is deleted once its client session is fully disconnected") and the low-level semantics (*e.g.,* "No client may create an ephemeral node when the session is in the CLOSING state") described in the failure ticket. In the second phase, the LLM maps the low-level semantics to actual boolean

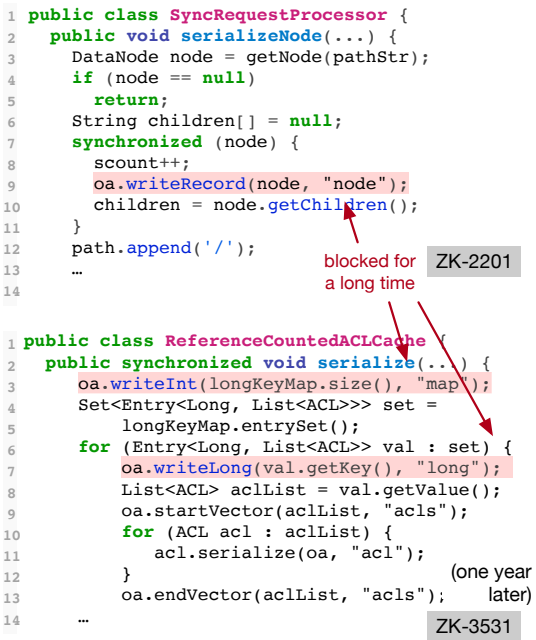

Figure 6: Example: low-level semantics should be generalized.

checks across the patch and surrounding source code. The LLM is prompted to output JSON-formatted pairs: the condition statement (predicates over concrete state and control-flow that needs to be checked), and the target statement (the code statement where the condition should be checked).

We made two discoveries while tuning our prompts.

First, asking the LLM to "just list the low-level semantics" yielded poor accuracy; forcing it to walk through its reasoning—linking code changes to developer intent and then to a semantic rule—anchored the answer in both implementation details and high-level properties.

Second, accuracy improved markedly once we clarified the notion of low-level semantics in the prompt itself, refining the definition and supplying concrete examples. To ensure robustness, we augment the LLM prompt with examples, enforce structural form on the outputs, and incorporate contextual embeddings (*e.g.,* test case summaries) via retrieval-augmented generation (RAG) to overcome input size limitations. This allows us to accurately infer semantics even when critical context is spread across multiple files or layers of abstraction.

The direct outputs often focus on specific functions or code paths, limiting generality. Figure 6 shows such an example. This regression involves a stuck serialization call inside a synchronized block. The first failure manifested as a zombie cluster where write operations were silently blocked. After developers fixed the issue, a similar bug reappeared a year later in a different serialization function. This highlights the need to generalize semantics beyond specific functions to a broader class of serialization patterns. Naively broadening the scope risks introducing many false positives. A more robust way is to abstract these rules to reflect system-level behaviors—*e.g.,* "no blocking I/O within synchronized blocks"-—which better capture the underlying intent and apply across code changes.

3.2 Asserting Semantic Rules with Concolic Execution

Once low-level semantics are inferred, our system verifies whether other parts of the system adhere to these semantics. While a more straightforward approach is to use static analysis to leverage code patterns, we find that many semantic failures have diverse patterns and often require substantial domain knowledge. Instead, we use concolic execution [53], a technique that systematically explores a program with concrete input and records the conditions for each execution path. It does not rely on summarized code patterns thus more generic towards different failures.

Complex systems have a vast space of execution paths, making exhaustive checking impractical. To focus on paths relevant to a given semantic, we identify those leading to the target statement it constrains. We do this by statically building a call graph and traversing all paths to each target. The result is an execution tree rooted at the target statement, with leaves representing entry functions for each path.

The tree can still be huge, so we prune further: the concolic engine follows only branches whose guards involve variables relevant to the semantic. We obtain that variable set by prompting an LLM–given the semantic's Boolean condition and the path's source code–to map the condition's placeholders to concrete variables; other branches are skipped.

In addition to selective branch exploration, our tool injects a code snippet right after all selected branches that will check the path condition of each relevant variable (not the concrete variable value) against the inferred condition in our semantic. To check whether these conditions adhere to the conditions of the semantic, we transform these trace conditions into Z3 formulas and compare them with the expected checker formulas derived from the semantics, and then check whether the trace does not fulfill the complement of the checker formula. Using the complement of the formula is important to guard against missing checks being treated as true conditions.

For example, the condition for an ephemeral node to be created is that the session must exist, not in closing state, and must not have expired, which is represented as the formula `(s != null && s.isClosing()== false && s.ttl > 0)` which has a complement of `(s == null || s.isClosing()== true || s.ttl <= 0)`. If we have a trace where the ephemeral node is created with a condition of `(s == null)` or `(s!=null && s.isClosing()== false)`, these traces fulfill the complement of the checker formula, so they violate our semantic. Missing `s.ttl` condition is treated as true value, so the second condition

is treated the same way as `(s!=null && s.isClosing()== false && s.ttl<=0)`. If we have `(s!=null && s.isClosing() == false && s.ttl>0)`, this trace does not fulfill the complement of the checker formula, so it adheres to our semantic.

Our execution tree now consists only of paths that are relevant to the semantics we are checking. Do note that "fixed" paths, which are paths that have been fixed in the pull request are also a part of this tree. We do not exclude these paths because these paths will act as our sanity check, where we want at least one path in this execution tree that will give verified result when checked against the semantic. The next step is to verify our execution tree using concolic execution.

Instead of doing execution with random inputs, our tool utilizes existing tests to act as our input because mature systems usually have extensive tests that cover all features in the system. The key question to this approach is how to select the relevant tests to act as concrete input? Our system automatically selects relevant tests for each path using LLM-based similarity search over test embeddings. This similarity search works by creating an embedding of all the tests using an embedding model. Then, an LLM model will be given an input of an execution path and then asked to identify the features involved by this execution path (remember that the execution path contains an entry function that is specific to a feature) and the condition for the feature to take this execution path. After that, the LLM will choose test cases that closely resembles the identified condition of the feature. These related tests are over-approximations of each paths which is then used as concrete input for the concolic execution engine.

Finally, we run these tests on the concolic execution engine. During execution, the result of the injected code snippets will determine whether the execution path is verified or not. If there are any execution paths that are not run, it either means the test suite does not have enough coverage, or the LLM misses the related tests. Developers should provide the final verdict for both cases.

To handle semantic mismatches between symbolic and static expressions, we implemented a normalization process between symbolic outputs and the LLM-inferred semantics. This normalization is done as an extension to detecting relevant variables, where we replace constant variables with their actual value rather than ignoring them. After detecting the class of the relevant variables in the LLM-inferred semantics, our tool automatically converts the LLM-inferred semantics into the format of our symbolic execution output.

4 Preliminary Results

We implemented an early prototype of LISA, based on the design described earlier. LISA extracts low-level semantics from past failures using an LLM and then asserts them through symbolic execution. We use OpenAI text-embedding-3-large [13] as our embedding model, OpenAI o4-mini [9] as the LLM model for our inference backend and build on WeBridge [37], a symbolic execution framework for Java applications. For call graph analysis, we employ Soot [56]. Applying LISA to a small set of historical failures, we identified two previously unknown bugs in HBase and HDFS.

Bug #1 [5] In HBase it is crucial to prevent expired snapshots from being used to avoid stale data. There have been existing efforts [3, 4] adding snapshot expiration checks in different scenarios to avoid such issues. However, in practice, users still observed expired snapshots returning to clients successfully without generating any alarms. Our tool found that such protection is not consistent and reported new execution paths missing such checks in the latest hbase version (5dafa9e). We propose to add timestamp checks to other paths, and the solution has been accepted by hbase developers.

Bug #2 [8] In HDFS if the block report of the observer namenode is delayed, one or more of the listing results would return blocks without any location. Missing locations indicate that the observer namenode is not up-to-date compared to the active namenode. In previous incidents [6, 7], a check was added in a few locations that checks whether the found blocks have valid locations. LISA found that these checks do not provide full coverage in the latest HDFS version (e8a64d0) and reported a new bug. We propose to complete the coverage of location checks. HDFS developers have approved the fix.

5 Open Questions

Can we make LLM-generated semantics reliable? We currently employ LLMs to infer low-level semantics. However, LLMs introduce two risks: (i) non-determinism—results may vary across runs, undermining reproducibility, and (ii) hallucination—generated semantics may be plausible-sounding but incorrect. These limitations are particularly problematic in the context of system verification where consistency and correctness are critical. To address this, we consider incorporating a cross-checking mechanism that validates mined semantics against test cases, ensuring that inferred rules are grounded in actual system behavior.

Can we provide better interface for developers to encode low-level semantics? Besides mining low-level semantics from existing resources, another approach is to enable developers to explicitly express these semantic rules in a more effective way. We plan to explore a solution that provides developers with a structured prompt template to describe expected behaviors in natural language. These descriptions can then be paired with LLM-assisted suggestions that generate corresponding formal rules or symbolic assertions.

Can we verify high-level system properties by composing multiple validated low-level semantics? Low-level semantics might serve as building blocks for higher-level guarantee. Our

long-term goal is to logically composing multiple low-level semantic rules and merge partial insights, so that it could provide a more complete, high-level form of system correctness guarantee. This is extremely challenging with current techniques; therefore, we plan to begin with a preliminary study on the collected cases as an initial step.

6 Related Work

Testing the correctness of distributed system implementations [27, 31, 32, 43, 45, 59, 66] has uncovered countless production bugs, yet it remains a best-effort approach: coverage is never complete, and a bug that slips through In contrast, we enforce insights learned from the past, aiming to stop the same class of failures from ever re-appearing.

There have been increasing interests in applying formal methods to enhance the correctness of distributed systems [22, 33, 40, 47, 64]. Meanwhile, the extensive cost of formal methods prevents its wide use in practice. Instead of full formal verification, this work explores a new approach by fusing lightweight formal methods with the generative power of LLMs to enable affordable and realistic guarantees of system reliability.

A line of work [18, 20, 25, 26, 29, 35, 41, 60] automatically mines likely invariants from software execution traces. Due to the high inaccuracy of statistics-based approach, they mainly support simple forms of relations and only captures violations at runtime, which is already late. Our approach aims to prevent the issues before they manifest in production.

Some works [21, 28, 49] explore extending regressions to systematically discover new issues. BESA [19] uses a combination of backward propagation and alias-aware analysis to find new null-pointer dereferences similar to known issues. Different from the relatively shallow, syntactic properties (*e.g.,* $variable \neq null$ or $flag == 0$) targeted at by existing works, we focus on richer behavioral semantics that span multiple methods or components in large-scale and complex cloud systems.

Recently LLMs have become increasingly popular in assisting developers to analyze existing resources for system insights [23, 50, 52, 57, 58]. LISA introduces a new abstraction that encodes low-level system semantics by exposing the right interface between natural language insights and machine-checkable guarantees, then supplies a full workflow that enforce them in development.

7 Conclusion

Regression failures constantly happen and cost unnecessary cloud resources, as well as developer efforts to debug and fix the issues. In this paper, we propose enforcing low-level semantics to prevent regression failures. By combining the reasoning capability of LLMs and verification from symbolic execution, this approach not only exposes unknown bugs to developers but also provides stronger guarantee that similar issues will not repeat. Our preliminary results with popular distributed systems show the promise of this approach.

Acknowledgment

We thank our shepherd Jon Crowcroft and the anonymous reviewers for their insightful reviews. We thank our undergraduate interns Julian Chandra Sutadi and Elva Chen for their help in collecting preliminary data. This research project has benefited from the Microsoft Accelerating Foundation Models Research (AFMR) grant program. Chang Lou is supported by the National Science Foundation grant CNS-2441284.

References

[1] Aws outage analysis december 7, 2021. https://www.thousandeyes.com/blog/aws-outage-analysis-dec-7-2021.

[2] Github hit with multiple back-to-back outages. https://www.zdnet.com/article/github-hit-with-multiple-back-to-back-outages/.

[3] HBASE-27671: Client should not be able to restore/clone a snapshot after it's ttl has expired. https://issues.apache.org/jira/browse/HBASE-27671.

[4] HBASE-28704: The expired snapshot can be read by copytable or exportsnapshot. https://issues.apache.org/jira/browse/HBASE-28704.

[5] Hbase-29296: Missing critical snapshot expiration checks. https://issues.apache.org/jira/browse/HBASE-29296.

[6] HDFS-13924: Handle blockmissingexception when reading from observer. https://issues.apache.org/jira/browse/HDFS-13924.

[7] HDFS-16732: Avoid get location from observer when the block report is delayed. https://issues.apache.org/jira/browse/HDFS-16732.

[8] Hdfs-17768: Observer namenode network delay causing empty block location for getbatchedlisting. https://issues.apache.org/jira/browse/HDFS-17768.

[9] Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/.

[10] It glue incident #dgrgk4kvz4n7. https://status.itglue.com/incidents/dgrgk4kvz4n7.

[11] It glue incident #z8r5glspbq3m. https://status.itglue.com/incidents/z8r5glspbq3m.

[12] Microsoft hit with teams, microsoft 365 outage issues. https://www.crn.com/news/cloud/microsoft-hit-with-teams-microsoft-365-outage-issues.

[13] New embedding models and api updates. https://openai.com/index/new-embedding-models-and-api-updates/.

[14] Psn appears to be down for ps4 and ps5 right now. https://www.androidcentral.com/psn-appears-be-down-ps4-and-ps5-right-now.

[15] Psn outage: Playstation network down for ps5 and ps4. https://www.androidcentral.com/psn-outage-playstation-network-down-ps5-and-ps4.

[16] ZooKeeper-1208: Ephemeral node not removed after the client session is long gone. https://issues.apache.org/jira/browse/ZOOKEEPER-1208.

[17] ZooKeeper-1496: Ephemeral node not getting cleared even after client has exited. https://issues.apache.org/jira/browse/ZOOKEEPER-1496.

[18] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, page 4–16, Portland, Oregon, 2002.

[19] J.-J. Bai. Besa: Extending bugs triggered by runtime testing via static analysis. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 1077–1091, Rotterdam, Netherlands,

2025.

[20] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with csight. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 468–479, Hyderabad, India, 2014.

[21] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. Precision reuse for efficient regression verification. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 389–399, Saint Petersburg, Russia, 2013.

[22] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, Virtual Event, Germany, 2021.

[23] Y. Chen, H. Xie, M. Ma, Y. Kang, X. Gao, L. Shi, Y. Cao, X. Gao, H. Fan, M. Wen, J. Zeng, S. Ghosh, X. Zhang, C. Zhang, Q. Lin, S. Rajmohan, D. Zhang, and T. Xu. Automatic root cause analysis via large language models for cloud incidents. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 674–688, Athens, Greece, 2024.

[24] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[25] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, page 5–14, Dubrovnik, Croatia, 2007.

[26] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 281–290, Leipzig, Germany, 2008.

[27] D. Domingo and S. Kannan. pFSCK: Accelerating file system checking and repair for modern storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 113–126. USENIX Association, Feb. 2021.

[28] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 349–360, Vasteras, Sweden, 2014.

[29] A. Fioraldi, D. C. D'Elia, and D. Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, Aug. 2021.

[30] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 328–343, Belgrade, Serbia, 2017.

[31] X. Fu, W.-H. Kim, A. P. Shreepathi, M. Ismail, S. Wadkar, D. Lee, and C. Min. Witcher: Systematic crash consistency testing for non-volatile memory key-value stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 100–115, Virtual Event, Germany, 2021.

[32] S. Gong, D. Peng, D. Altınbüken, P. Fonseca, and P. Maniatis. Snowcat: Efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 35–51, Koblenz, Germany, 2023.

[33] F. Hackett, J. Rowe, and M. A. Kuppe. Understanding inconsistency in azure cosmos db with tla+. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '23, page 1–12, Melbourne, Australia, 2023.

[34] T. Hance, A. Lattuada, C. Hawblitzel, J. Howell, R. Johnson, and B. Parno. Storage systems are distributed systems (so verify them that Way!). In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 99–115. USENIX Association, Nov. 2020.

[35] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 291–301, Orlando, Florida, 2002.

[36] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 1–17, Monterey, California, 2015.

[37] G. Hu, Z. Wang, C. Tang, J. Shen, Z. Dong, S. Yao, and H. Chen. Webridge: Synthesizing stored procedures for large-scale real-world web applications. *Proc. ACM Manag. Data*, 2(1), mar 2024.

[38] Y. Hu, G. Huang, and P. Huang. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, OSDI '20. USENIX, November 2020.

[39] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, Big Sky, Montana, USA, 2009.

[40] A. Lattuada, T. Hance, J. Bosamiya, M. Brun, C. Cho, H. LeBlanc, P. Srinivasan, R. Achermann, T. Chajed, C. Hawblitzel, J. Howell, J. R. Lorch, O. Padon, and B. Parno. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 438–454, Austin, TX, USA, 2024.

[41] C. Lee, F. Chen, and G. Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 591–600, Waikiki, Honolulu, HI, USA, 2011.

[42] A. Li, S. Lu, S. Nath, R. Padhye, and V. Sekar. {ExChain}: Exception dependency analysis for root cause diagnosis. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 2047–2062, 2024.

[43] G. Li, L. Zhou, N. Yu, Y. Ding, M. Ying, and Y. Xie. Projection-based runtime assertions for testing and debugging quantum programs. *Proc. ACM Program. Lang.*, 4(OOPSLA), Nov. 2020.

[44] C. Lou, Y. Jing, and P. Huang. Demystifying and checking silent semantic violations in large distributed systems. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '22, pages 91–107, July 2022.

[45] T. Lyu, L. Zhang, Z. Feng, Y. Pan, Y. Ren, M. Xu, M. Payer, and S. Kashyap. Monarch: A fuzzing framework for distributed file systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 529–543. USENIX Association, July 2024.

[46] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, page 241–251, Newport Beach, CA, USA, 2004.

[47] L. Ouyang, X. Sun, R. Tang, Y. Huang, M. Jivrajani, X. Ma, and T. Xu. Multi-grained specifications for distributed system model checking and verification. In *Proceedings of the Twentieth European Conference on Computer Systems*, EuroSys '25, page 379–395, Rotterdam, Netherlands, 2025.

[48] A. Panda, M. Sagiv, and S. Shenker. Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 30–36, Whistler, BC, Canada, 2017.

[49] F. Pastore, L. Mariani, A. E. J. Hyvärinen, G. Fedyukovich, N. Sharygina, S. Sehestedt, and A. Muhammad. Verification-aided regression testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 37–48, San Jose, CA, USA, 2014.

[50] C. Pei, Z. Wang, F. Liu, Z. Li, Y. Liu, X. He, R. Kang, T. Zhang, J. Chen, J. Li, G. Xie, and D. Pei. Flow-of-action: Sop enhanced llm-based multi-agent system for root cause analysis. In *Companion Proceedings of the ACM on Web Conference 2025*, WWW '25, page 422–431, Sydney NSW, Australia, 2025.

[51] R. Potvin and J. Levenberg. Why google stores billions of lines of code in a single repository. *Commun. ACM*, 59(7):78–87, June 2016.

[52] D. Roy, X. Zhang, R. Bhave, C. Bansal, P. Las-Casas, R. Fonseca, and S. Rajmohan. Exploring llm-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 208–219, Porto de Galinhas, Brazil, 2024.

[53] K. Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[54] Y. Su, C. Wan, U. Sethi, S. Lu, M. Musuvathi, and S. Nath. Hotgpt: How to make software documentation more useful with a large language model? In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, pages 87–93, 2023.

[55] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, and T. Xu. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666. USENIX Association, July 2024.

[56] 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.

[57] Y. Wang and K. P. Birman. Diagnosing and resolving cloud platform instability with multi-modal rag llms. In *Proceedings of the 5th Workshop on Machine Learning and Systems*, EuroMLSys '25, page 139–147, World Trade Center, Rotterdam, Netherlands, 2025.

[58] Z. Wang, Z. Liu, Y. Zhang, A. Zhong, J. Wang, F. Yin, L. Fan, L. Wu, and Q. Wen. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, CIKM '24, page 4966–4974, Boise, ID, USA, 2024.

[59] M. Wu, M. Lu, H. Cui, J. Chen, Y. Zhang, and L. Zhang. Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, page 56–68, Melbourne, Victoria, Australia, 2023.

[60] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 117–132, Big Sky, Montana, USA, 2009.

[61] X. Xu, Y. Yuan, Z. Kincaid, A. Krishnamurthy, R. Mahajan, D. Walker, and E. Zhai. Relational network verification. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 213–227, Sydney, NSW, Australia, 2024.

[62] J. Yao, R. Tao, R. Gu, and J. Nieh. DuoAI: Fast, automated inference of inductive invariants for verifying distributed protocols. In M. K. Aguilera and H. Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 485–501. USENIX Association, 2022.

[63] J. Yao, R. Tao, R. Gu, J. Nieh, S. Jana, and G. Ryan. DistAI: Data-driven automated invariant learning for distributed protocols. In A. D. Brown and J. R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 405–421. USENIX Association, 2021.

[64] T. N. Zhang, K. Singh, T. Chajed, M. Kapritsos, and B. Parno. Basilisk: Using Provenance Invariants to Automate Proofs of Undecidable Protocols. pages 1–17.

[65] N. Zheng, M. Liu, Y. Xiang, L. Song, D. Li, F. Han, N. Wang, Y. Ma, Z. Liang, D. Cai, E. Zhai, X. Liu, and X. Jin. Automated verification of an in-production dns authoritative engine. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 80–95, Koblenz, Germany, 2023.

[66] L. Zhong, C. Xiang, H. Huang, B. Shen, E. Mugnier, and Y. Zhou. Effective bug detection with unused definitions. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 720–735, Athens, Greece, 2024.