If At First You Don't Succeed, Try, Try, Again...?

Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems

Bogdan A. Stoica†*
bastoica@uchicago.edu

Utsav Sethi†*
usethi@uchicago.edu

Yiming Su†
yimingsu@uchicago.edu

Cyrus Zhou†
zhouzk@uchicago.edu

Shan Lu†‡
shanlu@microsoft.com

Jonathan Mace‡
jonathanmace@microsoft.com

Madanlal Musuvathi‡
madanm@microsoft.com

Suman Nath‡
suman.nath@microsoft.com

†University of Chicago
Chicago, IL, USA

‡Microsoft Research
Redmond, WA, USA

Abstract

Retry—the re-execution of a task on failure—is a common mechanism to enable resilient software systems. Yet, despite its commonality and long history, retry remains difficult to implement and test. Guided by our study of real-world retry issues, we propose a novel suite of static and dynamic techniques to detect retry problems in software. We find that the ad-hoc nature of retry implementation in software systems poses challenges for traditional program analysis but can be well handled by large language models; we also find that careful repurposing existing unit tests can, along with fault injection, expose various types of retry problems.

1 Introduction

Retry is a commonly used mechanism to improve the resilience of software systems. It is well understood that many task errors encountered by a software system are transient, and that re-executing the task with minimal or no modifications will succeed. However, retry can also cause serious or even catastrophic problems. Retry is oftentimes the last line of defense against various software bugs, hardware faults, and configuration problems at run time. Unfortunately, like other fault-tolerance mechanisms [10, 29, 34, 67], retry functionality is commonly under-tested and thus prone to problems slipping into production. Indeed, recent studies have identified a substantial portion of cloud incidents related to broken or unsafe fault-handling mechanisms, including that of retry [28, 31, 40, 45].

Despite its seeming simplicity, it is challenging to implement retry correctly. First, there are policy-level challenges regarding whether a task error is worth retrying and when to retry it. Often it is unclear which errors are transient and hence recoverable, and such retry-or-not policies require maintenance as applications evolve. It is also difficult to get the timing of retry correct: a system that retries too quickly or too frequently might overwhelm resources, while one that retries too slowly could lead to unacceptable delays in processing. Second, there are also mechanism-level challenges: how systems should perform retry — how to track job status, how to clean up the program state after an incomplete task, and how to launch a job again (and again) – continues to be prone to defects. These requirements are made more challenging by the fact that retry is not always a "simple loop": forms of retry that utilize asynchronous task re-enqueing, or circular workflow steps, whose implementation may be complex and difficult to identify, are common.

In recent years, a number of "resilience frameworks" or "fault tolerance libraries" have been developed to improve the resiliency of distributed applications, a major component of which has been configurable support for retry [23, 32]. But such frameworks, while helpful in some ways, cannot solve all policy or mechanism problems. While they support configuration of policy aspects (such as providing automated retry-on-error), they provide no help in deciding the policies, e.g. which errors should be retried; nor can they prevent issues in how retry is implemented. Moreover, their design can only support simple retry implementations. Instead, non-loop retry modes and retrying complex tasks—which are common—are difficult to support.

Testing retry logic presents similar challenges. To ensure reliability prior to deployment, developers typically run applications in a controlled, small-scale testing environment. However, recreating retry conditions requires developers to first, faithfully simulate *transient* errors that typically occur in production, and second, write specialized tests that exercise retry code paths with high-enough coverage and specially designed test oracles. Both are challenging and do not exist in today's unit testing framework.

*Both authors contributed equally to this research.

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11.
https://doi.org/10.1145/3694715.3695971

The goal of this paper is to systematically study and characterize real-world retry bugs; and provide a solution to help improve this pervasive and critically important functionality in software systems.

Understanding the retry challenge. By thoroughly studying 70 retry-related incident reports from 8 popular open-source applications in Java, we find that the root causes of retry-related incidents are about equally common regarding (1) IF to retry a task upon an error (36%), (2) WHEN and how many times a task is retried (33%), and (3) HOW to properly retry without leaking resources or corrupting application states (31%).

By inspecting the retry code fragments in these incidents, we observe a broad diversity in how retry mechanisms are implemented, making it difficult to automatically identify them. There is no dedicated retry API in any of the cases we studied. In about 55% of the cases, the retry functionality is implemented as a simple loop, while in 45% of cases it is implemented as a non-loop structure, either as a finite state machine or using asynchronous task re-enqueing. Instead, we find comments, log messages, variable names, and error codes to offer the clearest evidence of a retry code structure.

By running and analyzing all unit tests of these 8 applications (thousands to tens of thousands for each application), we confirm that existing unit tests are poor at exposing retry bugs. Based on our analysis, close to 10% of the unit tests invoke a part of the application that could trigger retry. However, retry almost never occurs during unit testing as transient errors are extremely rare. About 0.1%–0.5% of unit tests in these applications contain a mechanism to deterministically inject transient errors, but they only test a tiny portion of retry logic (e.g., most of them check whether the injected exception can be caught or not) and are not capable of catching those most common retry bugs discussed above.

Tackling retry bugs. Guided by these findings, we take a first step in enhancing the reliability of retry logic by developing Wasabi [63], a toolkit combining static program analysis, large language models, fault injection, and unit testing to tackle all three types of retry-related bugs (IF, WHEN, and HOW problems mentioned above) in both loop- and non-loop related retry. This suite of techniques uniquely enables bug-finding at the software mechanism level (i.e. retry), for which traditional program analyses are a poor fit.

Wasabi operates in two workflows—a *dynamic testing* workflow and a *static checking* workflow—that complement each other. In the dynamic testing workflow, Wasabi automatically alters the execution of unit tests using fault injection to exercise retry logic and expose retry bugs. Wasabi does not require developers to create specialized tests, bug oracles, or other bug-finding policies. Instead, it uses large language models (GPT-4) and traditional static analysis to identify the locations of retry and the trigger exceptions of retry in the source code. This enhances traditional program analysis with the *fuzzy code comprehension* capabilities of large language models, allowing for more nuanced detection of retry logic. It then formulates and executes a fault injection plan, leveraging existing unit tests to probe these locations under simulated faults. Finally, Wasabi comes with a set of test oracles that are specially designed to identify the manifestation of retry bugs.

Table 1. Applications included in our study

Application	Category	Stars	Bugs
Elasticsearch	Full-text search	66K	11
Hadoop[1]	Distr. storage/processing	14K	15
HBase	Database	5K	15
Hive	Data warehousing	5K	11
Kafka	Stream processing	26K	9
Spark	Data processing	37K	9

[1] Includes Hadoop Common, HDFS and Yarn

In the static checking workflow, Wasabi employs a combination of static control flow checks (CodeQL) and large language models (GPT-4) to identify retry-related bugs directly from source code. This approach extends the bug-finding capabilities of unit testing by allowing Wasabi to check retry code not covered by existing unit tests. On the other hand, it can incur more false positives than unit testing and miss bugs that are related to system run-time states.

In all, Wasabi identifies more than 100 distinct, previously unknown retry bugs in eight Java applications across all three types of retry-bug root causes. In particular, Wasabi identifies 42 retry bugs by re-purposing existing unit testing, and 87 through static analysis, with 20 bugs detected by both. Detailed comparison shows that re-purposed unit testing, traditional static code analysis, and large language models each have their own limitations and can well complement each other in the task of detecting retry related bugs.

2 Understanding Retry Issues

2.1 Methodology

Our study examines popular open-source distributed applications written in Java that cover various categories as listed in Table 1. For every application, we search for retry-related issues by using a set of keywords (*retry*, *resubmit*, *reattempt*, and *reschedule*) in their issue-tracking systems (Jira or Github issue-and-pull system). We look only at issues that (1) are labeled by developers as *bugs*, *resolved*, and *valid*, (2) have been fixed or have a patch awaiting merging, and (3) were reported within the time range of Apr 2018 — Nov 2023.

For every issue, we examine in detail the issue description, developer comments, patches and related source code, and linked issues if any. We divide retry issues into three categories based on their root causes, as listed in Table 2. We discuss each category in detail below and we also discuss the typical failure symptoms associated with each type.

Table 2. Root causes of retry bugs

Root Cause Category	# of Issues
IF retry should be performed	
- Wrong retry policy	17
- Missing or disabled retry mechanism	8
WHEN retry should be performed	
- Delay problem	10
- Cap problem	13
HOW to execute retry	
- Improper state reset	12
- Broken/raced job tracking	8
- Other	2
Total	70

While we aimed to select a representative set of applications, the conclusions of our issue study may not generalize to other applications and systems. Moreover, keep in mind that we have skipped issues whose descriptions are not clear for us to fully understand, as well as possible retry issues whose reports do not contain the keywords searched by us.

2.2 IF retry should be performed

Application logic must be selective about whether to retry: some errors are not transient and may require a different mitigation approach. However, deciding IF a failed task merits retrying can be challenging as we will see below.

2.2.1 Wrong retry policy. About a quarter (17) of the studied bugs are caused by incorrect retry policy: for 8 of them, recoverable errors were not retried, causing stuck jobs or even large scale performance degradation and system failure; for 9, non-recoverable errors were retried, which led to increased job latency or unresponsive client APIs.

Recoverable errors are not retried. In some cases, an application has a long list of error codes or exceptions; which of them could be returned by which functions and which reflect transient errors and hence should be retried are difficult for developers to track. For example, in Kafka, after a message is processed and committed, a response handler will check the error code, if any, and decide if retry is needed. Given the asynchronous nature of the execution, the large number of application-wide error codes in Kafka (74 in total), and the fact that message-processing and response-handling are located in different classes (Listing 1), it is not surprising that developers forgot to include error-code UNKNOWN_TOPIC_OR_PARTITION in the retry logic of the response handler — this error occurs when a message is committed during broker initialization, which can be recovered when the commit is tried again after initialization (Issue Kafka-6829).

```
class CommitResponseHandler {
  void handle(Error e, Future future) {
    if (e == COORDINATOR_LOAD_IN_PROGRESS ||
    +   e == UNKNOWN_TOPIC_OR_PARTITION
    ) {
        future.raise(RetryException());
        return;
    } else {
        future.raise(DoNotRetryException());
        return;
    }
  }
}

class ConsumerCoordinator {
    void commit() {
        ...
        sendCommit(msg, new CommitResponseHandler())
    }
}
```

Listing 1. Wrong Retry Policy - Recoverable error is not retried. +: the lines headed by '+' indicate developers' patch; the same applies to all figures in the paper. (KAFKA-6829, *Queue-based mechanism*)

Even if the list of recoverable error codes/exceptions is correct, it can be challenging to maintain such a list during the changes of applications and libraries. An example is HBASE-25743. HBase relies on the Zookeeper library for coordination. At some point, the Zookeeper library was upgraded and would return a new transient error, KeeperException.RequestTimeout, but this change was not noticed/fixed in HBase for over one year. A similar problem occurs in KAFKA-12339: an internal library was modified to return a new transient exception type UnknownTopicOrPartitionException, and yet the code calling this library was not changed to retry upon this new exception. This issue obstructed the worker from running during synchronization and was labeled as a critical, high-severity bug requiring an immediate hot-fix.

Non-recoverable errors are retried. In many cases, the granularity of error codes/exceptions is too coarse, with non-recoverable errors bundled with recoverable ones. For example, in HADOOP-16580, the Hadoop Common module defines a retry policy in which Java's IOException is retried. However, this decision is not granular enough: IOException encompasses a subclass AccessControlException, which indicates a permission failure and should not be retried.

Such wrong bundling could also occur during error propagation. In HADOOP-16683, function setupConnection correctly considers AccessControlException as a non-recoverable error and does not retry it as shown in Listing 2. However, other code paths in Hadoop may wrap AccessControlException inside the more general HadoopException, with the latter always getting retried. The patch has to unwrap HadoopException to differentiate non-recoverable errors from recoverable ones.

Another common mistake is to bundle task-cancel with recoverable errors, causing "cancel" to fail and resource waste. For example, in Elasticsearch, users can submit analytics jobs whose results are periodically persisted.

```
class WebHdfsFileSystem {
  private HttpResponse run() throws IOException {
    for(int retry=0; retry < maxAttempts; retry++) {
      try {
          HttpURLConnection conn = connect(url);
          HttpResponse response = getResponse(conn);
          return response;
      } catch (AccessControlException e) {
          break;
      // AccessControlException may be wrapped. Fix
      + } catch (HadoopException he) {
      +   if (he.getCause() instanceof AccessControlException)
      +     break;
      } catch (ConnectException ce) {
      }
      Thread.sleep(1000);
    }
    return null;
  }

  private HttpUrlConnection connect(URL url) throws AccessControlException, ConnectException;

  private getResponse(HttpURLConnection conn) throws IOException;
}
```

Listing 2. Wrong Retry Policy - Non-recoverable error is retried (HADOOP-16683, *Loop-based mechanism*)

If the job is cancelled, however, the ResultsPersisterService treats the cancellation as a recoverable error and keeps re-trying to write results indefinitely (ElasticSearch-53687). In HIVE-23894, a TezTask is submitted to a task queue inside a task processor; however if the TezTask is canceled, the task processor will mistakenly consider the task as failed and re-submit it to the queue. The fix again was to check the canceled flag inside the task, isShutdown, as shown in Listing 3.

```
class TezTask {
    boolean isShutdown;
    void execute();
    ...
}

class TaskProcessor {
    Queue taskQueue;
    void run() {
        Task task = taskQueue.take();
        try {
            task.execute();
        } catch (Exception e) {
        // FIX: only retry if not canceled
        +   if (task.isShutdown == false) {
                taskQueue.renqueue(task);
            }
        }
    }
}
```

Listing 3. Wrong Retry Policy: Canceled task is retried (HIVE-23894, *Queue-based Mechanism*)

How to catch these bugs? These types of bugs often manifest through various and hard-to-predict changes in task/system performance and behavior, and hence are challenging to identify through dynamic techniques. However, static analysis might help: statically extracting and identifying inconsistent error-retry policies may be feasible, particularly for retry implementations with straightforward control flows, e.g. for or while loops (Listing 2).

2.2.2 Missing or disabled mechanism.

In a few cases, developers did not realize the opportunity of retry in a component with the retry mechanisms completely missing or disabled. For example, in Hive, failures to fetch data segments from a node could be retried by checking other nodes that may contain redundant data. However, developers did not implement such retry initially, which hurts the robustness of related queries (HIVE-20349).

These bugs have similar symptoms as “recoverable errors not retried” bugs discussed above, but are much harder to automatically detect or fix — developers’ domain knowledge is needed to tell whether implementing a retry is feasible.

2.3 WHEN retry should execute

About one third of retry bugs that we studied are related to the timing of retry. Sometimes, the retry may be overly aggressive with no delay between each retry attempt (10 issues). This would lead to request flooding at server nodes, causing large-scale performance degradation or even service crashes. Sometimes, retry attempts are conducted endlessly, without a cap on the total number/duration of retry attempts (13 issues). Infinite retry attempts can cause jobs to hang and even application crashes due to out-of-memory errors.

2.3.1 Delay problems.

Missing delay occurs in all types of retry code structures, loop retry, queue-based retry, and state-machine based retry, and may lead to severe consequences.

Listing 4 illustrates such a problem in a state-machine based retry from HBase. In UnassignProcedure, which re-distributes ‘regions’ among servers (a core functionality), the REGION_TRANSITION_DISPATCH state involved a call to markRegionsAsClosing, which could fail when region server meta information was awaiting update. The exception was caught, and state deliberately left unchanged, so that this step could be retried by the executor. In contrast, if there is no exception, the state machine moves on to the next state, REGION_TRANSITION_FINISH. However, no delay was implemented between retries which, due to the rapid nature of failure, would congest the StateMachineExecutor preventing other procedures from progressing. Such behavior cannot be fixed by restarting, as procedures retain state upon restart. Instead, a critical fix had to be deployed to implement a delay between retries, as shown in the listing.

```
public class UnassignProcedure extends Procedure {
  void execute(State currentState) {
    switch(currentState) {
      case REGION_TRANSITION_DISPATCH:
          try {
              markRegionAsClosing();
              // proceed to next state
              setState(REGION_TRANSITION_FINISH);
          } catch (Exception e) {
              // Fix adds delay before implicit retry
          +   long backoff = (1000 * Math.pow(2,
               attemptCount))
          +   Thread.sleep(backoff);
              return; // implicit retry
          }
      case REGION_TRANSITION_FINISH:
          //...
    }
  }
}
```

Listing 4. Missing delay between retry attempts (HBASE-20492, *State-machine procedure*)

How to catch these bugs? Since delay is typically implemented by standard APIs (`Thread.sleep`, `TimeUnit.sleep`, etc.), issues in this category can be detected by checking if a certain API is invoked between two retry attempts either statically or dynamically. The key challenge is to identify which code snippets are conducting retry—no standard API is used for retry, as shown in earlier listings.

2.3.2 Cap problems. Infinite retry attempts should always be avoided. Sometimes, developers simply forgot to put any cap on retry. In other cases, configuration problems led to infinite retries. In HDFS, `dfs.mover.retry.max.attempts` configures the maximum number of retries for a mover job. In HDFS-15439, developers realized that setting this configuration to a negative value would unfortunately allow infinite retries — HDFS gave up on retry only when the number of retry attempts equals the configuration, which would never happen when the latter has a negative value.

Occasionally, broken attempt or time tracking may mistakenly cap retries below user-configured values. In YARN-8362, a state-machine procedure would re-attempt a transition on failure up to a max count value; however the counter variable was incremented twice, both during state transition and a subsequent status check, effectively causing max retries to be half the value configured by the user. There are similar problems when timeouts are not tracked correctly.

How to catch these bugs? Most issues in this category lead to too many retry attempts, which can be caught by testing if (1) a checker can identify each retry attempt (i.e., distinguishing a retry attempt from a normal, fresh task); (2) the reason for a task to fail can persist for a long time during testing. As for static checking, the challenge will be identifying which code snippets are conducting retry and reasoning about the termination conditions of retry.

2.4 HOW to execute retry

Conducting retry correctly is difficult, as it often involves complicated job status tracking and system state cleanup. Various semantic bugs in retry execution led to symptoms like data corruption, resource leaks, request failures, etc.

The most common problem occurs in the meta-data maintenance of job retry. Coordinating jobs in distributed systems is a challenge and retry introduces additional complexity. For example, in Spark, jobs are composed of stages, and the job manager may retry a stage when it does not respond within a certain threshold (labeled as 'zombie'). However, zombie stages could still progress and update status in a map `stageIdToNumTasks`, used by the job manager to link stages to running tasks. Because original and retried stages shared the same `stageId` key, modifications by both would corrupt this map, leading to stuck jobs (SPARK-27630).

Another common source of bugs is incomplete or incorrect state reset during retry: failed jobs may have performed partial work or state changes, which need to be properly reset before retry. For example, HBase uses a state-machine procedure to truncate tables. One of the state machine steps, `CREATE_FS_LAYOUT`, creates a new set of files in HDFS for the table after the table's data has been truncated/deleted. If this step fails to write all files, it is retried; however files previously written are not cleaned up, so attempts to rewrite these files will fail and prevent the entire procedure from succeeding (HBASE-20616).

How to catch these bugs? The semantic bugs behind these How-to-retry bugs differ a lot from each other. They do not conform to a unified code structure and hence are difficult to detect using program analysis. We hypothesize that unit testing could be helpful to expose this type of bug, just like how unit testing is used to expose generic semantic bugs. We will explore this in the next section.

2.5 Other study findings

Bug severity. The priority labels offered by developers suggest that many of these bugs are problematic: those with the highest-priority label, "blocker", account for 5% of our dataset, and likewise "critical" 10%, "major" 65%, and "minor" 5%, with the remaining 10% unlabeled. This is not surprising: as we have highlighted earlier, broken retry often leads to serious problems including data corruption, severely impaired functionality, and application crashes.

Retry mechanisms. As shown by examples earlier, these issues involve not just simple loop-based retry but also other more complex forms of retry. Specifically, 25% of issues deal with asynchronous task re-enqueueing, where a request is defined inside a "task" or "message" object that is re-submitted to a queue for retry (e.g. Listing 3); and 20% of issues involve a special case of asynchronous retry which we call "state-machine" retry, where a framework allows tasks to be defined as a series of states, and supports retry by re-transitioning to the current state upon errors (e.g., Listing 4).

All these three types of retry mechanisms impose challenges for automated retry-bug detection and retry-code testing. The latter two mechanisms often obscure retry logic or disperse it across files, making it difficult to understand

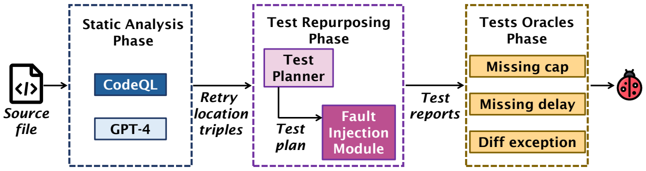

Figure 1. Repurposed unit testing of Wasabi.

if retry is performed and when. Even for simple loop-based retry, it may be difficult to differentiate a loop with retry from those that contain no retry. We will present in detail how we tackle these challenges in the next section.

Unit tests. For 42 out of the 70 retry bugs in our study, regression unit tests were added by developers after corresponding bugs were resolved.

Across these 8 applications, about 0.1%–0.5% of their test cases are labeled to be related to retry (348 unit tests in total). Most of these unit tests focus on a tiny portion of retry logic: the unit test throws a retry trigger exception and then checks if the exception can be caught and delivered to the right catch block. There are also many of these unit tests that exercise miscellaneous retry helper functions, like getter/setter functions of retry-related configurations, the retry condition checking functions, the delay back-off functions, etc. Finally, some unit tests check whether the execution, after one retry attempt triggered by an exception explicitly thrown by it, satisfies a specific safety property. These are typically regression tests for specific HOW retry bugs.

Overall, we can see that developers value the usage of unit tests to capture retry bugs. However, developers need help in the design/synthesis of unit tests that can systematically exercise core retry logic in their applications with good coverage and diverse test oracles.

3 Wasabi

Given the different characteristics of IF-, WHEN-, and HOW-retry bugs, as discussed above, we have designed (1) a testing workflow that repurposes existing unit tests to expose WHEN and HOW retry bugs (§3.1) and (2) a set of static analysis routines and prompts that detect IF and WHEN retry bugs (§3.2). We refer to them together as Wasabi.

3.1 Exposing retry bugs through unit tests

Modern systems all have extensive unit tests, typically thousands or more, carefully designed by developers. These unit tests offer a good opportunity to systematically exercise retry logic and expose bugs of various root-causes and symptoms.

However, directly running existing unit tests would not work for two key reasons. First, the triggers of retry are typically low-probability events like networking failures, and hence rarely occur during in-house testing. Second, even if a retry is triggered and a problematic retry logic is exercised, the problem may not be caught by existing test oracles that were designed without retry in mind. With these challenges in mind, Wasabi uses the following components to repurpose existing unit tests and expose retry bugs:

1. A static analysis routine that processes program source code and reports where retry may happen (§3.1.1);
2. A fault injection module that simulates an exception to trigger retry when exercising unit tests (§3.1.2);
3. A set of test oracles specially designed to catch retry-related bugs (§3.1.3);
4. A test planner that coordinates among all unit tests so that all retry locations in unit tests will be exercised through fault injection in a cost-efficient way (§3.1.4).

For the ease of discussion, we refer to a function M whose execution error causes itself to be re-executed as a *retried method*, such as `connect` and `getResponse` in listing 2, `task.execute` in listing 3, and `markRegionAsClosing` in listing 4. We refer to the caller function of M that catches the error and conducts the retry as the *coordinator method* (e.g., function `run` in listing 2 and listing 3, and `execute` in listing 4). The execution error of the retried method typically is reported to the coordinator through exceptions (70% of the cases in our study) or error codes (30%). The current prototype of Wasabi focuses on exceptions, which we refer to as *retry triggers*. Finally, we refer to the call site of the retried method inside the coordinator method as a *retry location*.

3.1.1 Identifying retry locations. The main challenge is that retry code does not have a unique code pattern for traditional program analysis to search for. For example, loop-based retry (listing 2) all involve try-catch blocks in loops, but there are also many loops with try-catch that do not offer retry — a method could be repeatedly invoked in a loop for processing different inputs, instead of for retry. Non-loop retry that involves queues (listing 1, listing 3) and state machines (listing 4) are even harder — it is difficult to tell whether a method would be re-executed from this code, not to mention whether the re-execution is for retry or not.

To tackle this challenge, we design two complementary techniques that leverage both non-structural code elements like variable names and comments, which we observed to be much better indicators of retry functionality than program structure alone. The output of both techniques is a list of retry-location triplets: a coordinator method C, a retried method M, and a retry-trigger exception E, as defined above.

The first technique identifies loop retry using control-flow analysis and naming conventions. Note that we leave the detection of non-loop retry to the second technique below, as there is no effective way to detect non-loop retry using traditional control/data-flow analysis.

Using CodeQL [22], a query-based static analysis tool, we first apply traditional control-flow analysis to identify every loop whose header is reachable from at least one catch block inside the loop body—an indication of potential retry through exception handling. We then check if the loop body or loop condition contains any string literals, variables, or methods whose name include "retry" or "retries". For example, the loop header in listing 2 is reachable from the catch block on Line 14, although not from the catch block on Line 8; the loop also contains a counter variable named `retry`.

Once we identify such a retry loop L, the static analysis checks the prototype of every method M invoked in the loop to see whether M could throw an exception E so that the header of loop L is reachable from the catch block of E. If such an exception E is found, E is considered a potential retry trigger, the call site of M inside loop L is identified as a retry location, and the method that contains this call site is a coordinator method. For example, in Listing 2, Line 5 is identified as a retry location, as the callee method `connect` there could throw exception `ConnectException`, whose catch block on Line 14 can reach back to the loop header.

The second technique covers both loop and queue/state-machine retry mainly using large language models. Specifically, we prompt GPT-4 to identify potential retry logic, with the prompt explicitly reminding GPT-4 of different ways to implement retry (Q1 in Figure 2), and feed GPT-4 one file at a time across the entire code base. Once GPT-4 reports a method C as one that implements retry (using a follow-up prompt of Q1 not shown in Figure 2), we use a simple CodeQL query to identify all methods invoked by C as potential retried methods and all exceptions thrown by them as potential retry triggers. Note that we go back to CodeQL for the latter step, because callee methods are often implemented in a different file from the caller, which hinders GPT-4 regarding which callee may throw what exceptions.

Our retry location identification is neither sound nor complete. It may report a retried method M and its exception E that actually cannot be caught by the caller to trigger the retry of M. Fortunately, our test oracles handle these analysis inaccuracies, preventing them from causing false bug reports, as explained in §3.1.3. As we will see in the evaluation (Section 4.2), our retry-location identification is mostly accurate in practice.

3.1.2 Simulating a trigger exception.

Once a retry location L is identified, Wasabi can instrument a unit test that exercises L so that a retry-trigger exception is thrown during the test. Of course, static analysis may report that multiple exceptions could be thrown at a retry location and trigger retry, in which case Wasabi creates multiple unit *retry* tests, each with one type of exception injected.

We implement this instrumentation using AspectJ, which allows us to register a handler, called pointcut in AspectJ, that gets executed right before a callee method specified by us (i.e., a retried method) is invoked by a caller method specified by us (i.e., a coordinator method). As illustrated in listing 5, the handler takes as argument the callee method name, the caller method name, and the exception to be thrown. The handler simply (1) throws the exception if this particular injection point was reached less than K times, and (2) writes a corresponding entry into the test log.

```
onCallAt(callee, caller, exception) {
  int callCount = hashTable.get({callee, caller, exception})
  if (callCount < K) {
    log.debug("Injected {exception} {callCount + 1} times, at callsite {callee} invoked from {caller}");
    hashTable.set({callee,caller,expcetion},callCount+1);
    throw new exception()
  }
}
```

Listing 5. Exception Throwing Handler Pseudocode

To decide the value of K, we consider how many retry attempts are needed to expose a retry bug. For each unit test that targets a specific retry location and retry trigger, Wasabi runs it twice with two different settings of K: 1 and 100. The latter number is chosen to safely exceed all application-configured thresholds. This exercises different aspects of retry: with $K = 1$, the unit test can exercise the code after the retry and get a chance to expose HOW retry bugs—e.g., incorrect state clean-up (§2.4); with $K = 100$, the unit test can exercise the retry cap and delay mechanism (§2.3), yet the application code after retry may not execute if the application times out.

3.1.3 Retry test oracles.

Every unit test comes with test oracles often in the form of `assert` statements, so that software bugs can manifest as test failures. However, retry bugs may not violate existing test oracles — they might degrade application performance, but not triggering crashes; or they might lead to unexpected data corruption that does not violate existing assertions. Furthermore, a violation of existing test oracles may not reflect true bugs under our exception injection scheme, as we will explain later.

To tackle this challenge, we have designed three retry-specific, application-agnostic test oracles.

"Missing cap" oracle. This oracle classifies test runs based on the number of retry attempts, and is meant to catch 'Cap problems' of WHEN-retry bugs. In theory, missing cap retry bugs are characterized by an unbounded number of retry attempts (§2.3.2). In practice, retry caps are typically no more than 20 retry attempts or 10 minutes [14–21]. In the current prototype of Wasabi, each test-run log is examined to see if any fault injection handler (listing 5) has thrown exceptions for 100 times or if any unit test has run for more than 15 minutes. If so, a missing-cap bug is reported.

Rare cases where a callee method is invoked 100 times for reasons other than retry (e.g., handling different tasks) could lead to false positives in Wasabi. Similarly, while a unit test might correctly run for over 15 minutes, this is extremely rare as they, in our experience, typically terminate within approximately 30 seconds.

"Missing delay" oracle. This oracle is to catch 'Delay problems' in WHEN-retry bugs (§2.3.1). It checks testing-run logs to see if there were any delays between consecutive retry attempts, and reports missing-delay bugs if there is no delay in-between consecutive retry attempts.

Wasabi registers an AspectJ handler to generate a log entry together with the call stack right before every call of a sleep API (i.e., `Thread.sleep`, `Object.wait`, `TimeUnit.sleep`, `TimeUnit.timedWait`, `TimeUnit.scheduled ExecutionTime`, `Timer.wait`, and `Timer.schedule`). After each test, Wasabi checks the log to see if there is a `sleep` call in-between two consecutive fault injections from the same retry location. Wasabi compares the call stack to only consider a sleep issued from the corresponding coordinator method.

"Different exception" oracle. This oracle checks exception(s) thrown by the test code, not by Wasabi handlers, and flags an execution as potentially buggy when the exception is *different* from the one Wasabi injected. This oracle is meant to find `HOW` type of retry bugs (§2.4).

To maintain accuracy, our oracle intentionally avoids reporting correct behavior where, after a few retry attempts, the unit test gives up and re-throws the same exception that was initially injected by Wasabi. While this causes the unit test to crash, it is typically correct behavior as it allows the upper layer to handle the error. Additionally, the oracle avoids reporting false positives resulting from our static analysis inaccuracies (§3.1.1): if an injected exception is not a retry trigger, the unit test will crash and throw the injected exception without being flagged as a bug.

3.1.4 Tests preparation.

There are a few remaining roadblocks for Wasabi to effectively utilize unit tests.

Restoring default retry configurations. Developers sometimes deliberately restrict retry in unit tests, by explicitly overriding the default configuration of the maximum number of retry attempts to 0, 1, or 2 (in about 10% of the test cases that cover retry logic based on our study). To counteract developer-imposed restrictions on retry logic, we use a Python script to scan every unit test and pinpoint instances where retry parameters are altered. Next, we override these values with the default ones in the application's configuration files or documentation. This ensures that Wasabi's retry testing accurately reflects the intended retry behavior without artificial limitations.

Fault-injection planning. A naive testing plan that injects trigger exceptions at every retry location in every unit test would lead to insufficient testing of some retry locations and redundant testing of others: (1) when one unit test contains multiple retry locations (e.g., the invocations of `connect` and `getResponse` on line 5 and 6 in Listing 2), injecting exceptions at an early location could cause later retry locations to be skipped — the coordinator method may terminate after several retry attempts at the early location; (2) one retry location and its corresponding coordinator method are often covered by multiple unit tests; repeatedly conducting retry attempts at the same location across different unit tests is often a waste of testing resources.

To improve this naive plan, before any fault injection, Wasabi instruments every retry location reported by its static analysis, and runs the entire test suite once to figure out which retry locations are covered by every unit test at run time. Wasabi then conducts a *test planning*, with the resulting plan being a list of {unit-test, retry location} pairs. Wasabi's planning makes sure every retry location that could be covered by the test suite appears exactly once in the plan list; Wasabi also tries to maximize the number of unique unit tests covered by this plan, although there is no guarantee. Specifically, Wasabi iterates through every unit test one by one. For each test t, Wasabi identifies its first retry location ℓ, if any, that is not yet covered, adds $\{t, \ell\}$ to the plan, and moves on to the next test case. After iterating through every test case once, if there are still uncovered retry locations, Wasabi iterates through every unit test again (and again), until every retry location is included in the plan.

For each pair in the test-plan list, Wasabi uses AspectJ to turn a unit test into a series of unit *retry* tests that exercise retry triggered by different exceptions associated with that retry location, as discussed in §3.1.2.

3.2 Detecting retry bugs through static analysis

Wasabi's unit testing depends on the quality and coverage of existing test suites. Moreover, Wasabi's test oracles do not cover IF bugs. To fill in these gaps, we include static analysis routines/prompts as part of Wasabi's detection suite.

3.2.1 WHEN bug detection using GPT-4.

To identify additional WHEN bugs, we use a LLM prompt-based design. This allows us to find bugs in both loop and non-loop retry forms, such as the one in Listing 4. Our WHEN bug-detection prompts are a series of yes/no interactions about possible missing cap or delay problems, along with the contents of a single application source file (Figure 2). The prompts include clarifications to improve detection of different types of retry-related behaviors, such as asynchronous scheduling-based delay, as well as clauses to reduce the incidence of false positives—e.g. exclusion of non-retry related timeouts. The prompts were arrived at by experimenting with different hints, word choices, and code formatting. We also include an additional prompt to address one more type of false positive: GPT-4 will often label cases that implement spin-lock- or polling-related functionality as retry. As these do not conform to our definition of retry (i.e. re-execution on error), we use this prompt to exclude these cases.

3.2.2 IF bug detection using CodeQL.

Wasabi's fault injection and unit testing can help discover WHEN and HOW bugs, but not IF bugs — we cannot tell whether a task error is recoverable or not when the error is injected.

```
Q1. Does the following code perform retry anywhere
    ? Answer (Yes) or (No).
- Say NO if the file only _defines_ or _creates_
    retry policies, or only passes retry
    parameters to other builders/constructors.
- Say NO if the file does not check for exception
    or errors before retry.
**Remember that retry mechanisms can be
    implemented through for or while loops or
    data structures like state machines and
    queues.**
< Entire file contents >

Q2. Does the sleep before retrying or
    resubmitting the request? Answer (Yes)or(No).
  **Remember that delay might be implemented
      through scheduling after an interval or
      some other mechanism.**

Q3. Does the code have a cap OR time limit on the
    number times a request is retried or
    resubmitted? Answer (Yes) or (No).
  **Remember that timeouts or caps should be
      specifically applied to retry and not other
       behaviors**

// Used to exclude poll/spin-lock-related cases
Q4. Do any of the retry-containing methods either
    call "compareAndSet" or contain poll-related
    behavior? Answer (Yes) or (No)
```

Figure 2. GPT-4 prompts used in Wasabi

.

Therefore, as a part of the Wasabi toolkit, we propose a static technique that reports *likely* IF bugs in a statistical way — if an exception is (not) retried in most places across a codebase but not (is) in few cases, even though the retry mechanism was there, those outliers are flagged as potential IF bugs. Here, we focus on traditional CodeQL-based static analysis and loop-based exception retry mechanism.

For each given exception E, our analysis counts the number of retry loops N_E where E could be thrown — the retry loops are identified by CodeQL as discussed in §3.1.1 and the exceptions that could be thrown in each loop are identified by analyzing signatures of callee methods of the loop. We then count the subset of these cases R_E where the exception is retried, by analyzing whether there exists an exception-catching basic block with a branch that returns control to the start of the loop, as discussed in §3.1.1.

We use the application-wide retry ratio, $\frac{R_E}{N_E}$, to infer recoverability of the exception E and identify outliers: when this ratio is very close to 1 (or 0) but is not equal to 1 (or 0), Wasabi would report the outliers as a reminder for developers to check the retry policy decision.

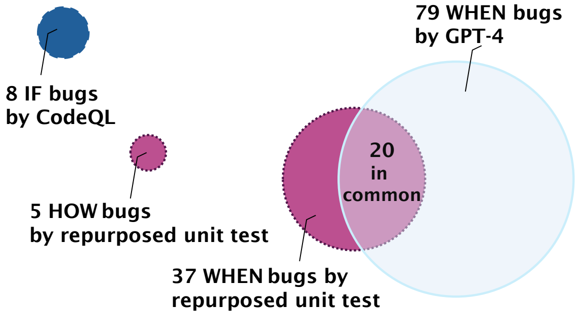

Figure 3. Bugs found by Wasabi unit testing and static checking illustrated as different circles.

4 Evaluation

We evaluate Wasabi largely on the same set of applications used in our issue study. The difference is that (1) we used the latest version of each application as of March 2023 when we started designing Wasabi; and (2) we excluded Kafka and Spark from the set, and instead added MapReduce and Cassandra. We excluded Kafka because its retry logic is predominantly driven by error codes and application state, rather than exception handling, and hence is out of the scope of Wasabi; we excluded Spark, because of an incompatibility with the AspectJ Maven plugin [56]. Overall, we used 8 applications in our evaluation: Hadoop-Common (HA), HDFS (HD), MapReduce (MA), Yarn (YA), HBase (HB), Hive (HI), Cassandra (CA), and ElasticSearch (EL).

We ran each application on an Ubuntu 22.04.3 LTS machine, with a 12-core Intel i7-8700 CPU, 32 GB of RAM, and 512 GB of disk space. We relied on both Java 11 and 17, Maven 3.6.3, AspectJ 1.9.19, and the AspectJ Maven plugin 1.13.1 [56]. Additionally, ElasticSearch requires Gradle 4.4.1, instead of Maven, as the build system.

4.1 Results on Bug Detection

In total, Wasabi reports 191 *distinct* retry problems across all 8 applications evaluated, with its repurposed unit tests reporting 63 WHEN and HOW retry problems, its GPT-4 static checker reporting 139 WHEN retry problems, and its CodeQL checker reporting 9 IF retry problems. We have carefully examined every Wasabi report and identified 109 of them as true bugs — Wasabi unit testing has a false positive rate of 2 true bugs vs. 1 false positive, and Wasabi static analysis has a rate of 1.4 true bugs vs 1 false positive. We are releasing our tool and a detailed description of every bug reported by our tool together with our paper [63].

We illustrate the distribution of the 109 reports that we deem as true bugs in Figure 3, and explain them in detail below. §4.3 will discuss the 82 false positives in detail.

Wasabi unit testing. As illustrated in Figure 3, Wasabi identifies 42 bugs through its repurposed unit testing: 20 are missing-cap WHEN problems (i.e., infinite retry); 17 missing-delay WHEN problems (i.e., aggressive retry without delay); and 5 issues related to HOW retry is implemented. Those 5 HOW-retry bugs were exposed by injecting a retry-trigger exception just once, while the other bugs were exposed by injecting retry exceptions 1,000 times, as discussed in §3.1.2.

As an example, after WASABI injects `SocketException` once during the unit test *testSaveAndLoadErasureCodingPolicies* of HDFS, the test fails with `NullPointerException`. WASABI's "different exception" oracle flagged this as a potential bug. After inspection, we realized that when a transient error happens too early in function `createBlockReader`, not all objects are properly allocated. However, the `catch` block that handles `SocketException` assumes all objects were constructed and attempts to log the current program state. This logging results in a `NULL` pointer dereference.

Note that, one bug can cause multiple WASABI unit test runs to fail. For example, the above bug also caused crashes when exceptions were thrown at two other retry locations in the same retry loop. The "different exception" oracle considers two crash failures as the same bug if they have the same crash stack; the "missing cap" and "missing delay" oracles group test reports based on retry code structures (e.g., only one missing cap/delay bug is counted for each retry loop).

Without WASABI, the probability of exposing *any* of these bugs during unit testing is extremely low, if at all. Indeed, we have run the original test suite without WASABI many times, and *none* of the problematic retry logic is exercised. Even if these bugs are exposed during testing by luck, they cannot be effectively handled by existing test oracles, which we will discuss more in §4.4.

WASABI static checking. WASABI's GPT-4 based bug detection, presented in §3.2.1, finds 79 WHEN bugs. Of these, 20 are also found by fault injection. Comparing GPT-4 and WASABI's repurposed unit testing, the false negatives of the latter are mainly due to the lack of code coverage of existing unit tests, while the false negatives of the former are mainly due to GPT-4 struggling to reason about large files and hence not even realizing the existence of retry, which we will elaborate in the next sub-section. Also note that, although GPT-4 identified more WHEN bugs than unit testing (79 vs. 37), it also reported more false positives (60 vs. 16). Overall, GPT-4 and unit testing complement each other well.

As discussed in §3.2.2, WASABI's IF bug detection uses CodeQL to compute the application-wide retry ratio for each exception. WASABI finds 9 outlier cases in total where an exception is mostly but not always retried (i.e., retry ratio $\geq \frac{2}{3}$), or the other way around (i.e., retry ratio $\leq \frac{1}{3}$). We have manually checked all cases and believe 8 of them to be truly problematic. These 8 cases come from 5 applications (1 in Hadoop, 1 in Yarn, 3 in HBase, 2 in Hive, and 1 in Cassandra), and involve these 5 exceptions with their retry ratio in parentheses: `Zookeeper.KeeperException` (17/20), `Thift.TTransportException` (2/3), IllegalArgumentException (2/9), Hadoop.ExitException (1/3) and IllegalStateException (1/3). For example, `KeeperException` can be thrown due to transient network errors such as timeout or connection loss, and is retried in 17 out of 20 places where it is caught inside a retry loop.

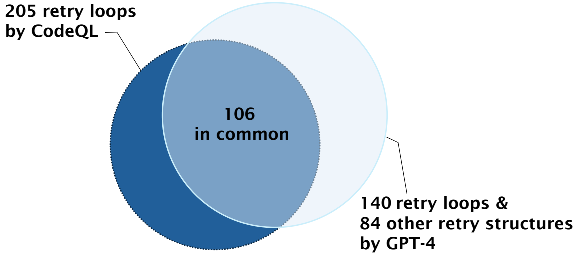

Figure 4. Retry code structures identified.

4.2 Retry Code Identification and Coverage

Retry code identified. As shown in Figure 4, WASABI identifies 323 code structures across all 8 applications where retry logic is implemented. About 70% of them are loops (i.e., 239 retry loops in total), while the rest implement retry through finite state machines and task re-enqueuing.

Comparing the two approaches, CodeQL cannot detect non-loop retry but did manage to identify more than 85% of the retry loops reported by the two techniques. Naturally, it missed retry loops that contain no string literals, variables, or methods whose name include "retry" or "retries". GPT-4 has the advantage of identifying non-loop retry, but it missed 100 retry loops. Our investigation showed that these are located in 53 different large files. On average, these files are almost twice as large (mean: 10,539, median: 9,304) than those where GPT-4 does identify retry logic.

Both occasionally mislabel locations. A manual examination of 40 sampled retry loops identified by CodeQL reveals 3 false positives: an attempt to obtain a lock and failure logging if unobtainable after *n* "retries"; an attempt to generate a unique string and failure after *n* "retries"; and token-by-token parsing of a request which may contain a "retryOnConflict" parameter. The locations found by GPT-4 have a slightly higher false-positive rate: of 100 sampled locations, we find 16 false positives, which contain re-execution behavior such as iterating through queues, or status-update polling; as well as object parsing or construction that contain a retry-named parameter. Fortunately, these false positives do not affect the accuracy of WASABI unit testing: injecting exceptions at non-retry code would cause a test to crash with the same exception, which would be labeled as non-bug by WASABI's "different exception" oracle. However, the false positives in GPT-4's retry identification are connected with the false positives in its bug identification.

Table 3. Retry bugs reported by Wasabi unit testing (subscripts: # of false positives; -: no report).

Retry Bug Type	Hadoop	HDFS	MapReduce	Yarn	HBase	Hive	Cassandra	ElasticSearch	Total
WHEN bugs: missing cap	2_1	7_2	-	1_1	13_2	3_1	1_0	1_1	28_8
WHEN bugs: missing delay	3_2	6_3	5_1	-	6_2	2_0	2_0	1_0	25_8
HOW retry bugs	-	4_2	-	-	4_2	2_1	-	-	10_5
Total	5_3	17_7	5_1	1_1	23_6	7_2	3_0	2_1	63_{21}

Table 4. Retry bugs reported by Wasabi GPT-4 detector (subscripts: # of false positives)

Retry Bug Type	Hadoop	HDFS	MapReduce	Yarn	HBase	Hive	Cassandra	ElasticSearch	Total
WHEN bugs: missing cap	3_3	9_4	3_3	2_0	16_5	7_6	10_4	10_8	60_{33}
WHEN bugs: missing delay	7_4	9_2	4_1	4_0	16_4	17_6	5_1	17_9	79_{27}
Total	10_7	18_6	7_4	6_0	32_9	24_{12}	15_5	27_{17}	139_{60}

Table 5. The number of static retry code structures identified and covered in Wasabi unit tests

App.	HA	HD	MA	YA	HB	HI	CA	EL
Identified	38	41	16	18	98	59	15	38
Tested	12	27	12	11	48	14	6	5

Retry code covered in testing. Table 5 shows the breakdown of retry code structures identified in each application, as well as how many of these retry code structures get covered by Wasabi in its unit testing. Two key factors contributed to some retry structures not covered by Wasabi unit testing: (1) as discussed earlier, Wasabi unit testing focuses on exception-triggered retry only and hence cannot cover error-code triggered retry; (2) some retry code structures are not covered by any existing unit test. For example, Hive (HI) and ElasticSearch (EL) have a large portion of error-code related retry, and hence have the lowest retry coverage. For the remaining 6 applications, Wasabi unit testing is able to cover 32% – 75% of the retry code structures.

4.3 Cost and False Positives

Cost of Wasabi. For most of these 8 applications, Wasabi unit testing took around 10 hours, with HBase taking the most time (close to 20 hours). The majority of the time is spent on running the test cases, with less than 1% spent on static analysis or post-mortem log processing. The test run time can be further broken down into two parts. First, the time to run every test in the test suite once to figure out which test case covers which retry location, as part of the Wasabi test planning (§3.1.4). This takes 18%–32% of the total run time — all these applications come with thousands or tens of thousands of unit tests, as shown in Table 6, that take more than an hour to run. Second, the time to run all Wasabi repurposed unit tests with injected exceptions, which takes the remainder of the test run time. Since Wasabi is designed for in-house testing, we consider the overheads acceptable.

Table 6. Details of Wasabi unit testing

App.	# Unit Tests		# Wasabi Test Runs	
	Total	CoverRetry	w/o planning	w/ planning
HA	7296	841	9156	54
HD	7642	405	7834	110
MA	1468	393	2940	48
YA	5757	764	4764	42
HB	7052	1438	4248	158
HI	35289	1505	2506	36
CA	5439	952	1132	26
EL	12045	1388	1802	28

Note that, exceptions and exception handling are very costly, not to mention that one of Wasabi's fault-injection policies is to throw up to 100 exceptions or terminate a unit test at 15 minutes. Wasabi's repurposed unit testing only increases the original unit testing time by 2X–5X, instead of hundreds to thousands of times, because only a portion of unit tests actually cover retry locations (4%–27% across all applications as shown in Table 6). More importantly, Wasabi's planning stage makes sure that retry locations are not repeatedly tested across different unit tests (§3.1.4), which helps cut the number of fault-injection testing runs by 27X–170X, as shown by the last two columns of Table 6.

Cost of GPT-4. To execute the workflows that involve GPT-4, namely retry location identification and static WHEN bug detection (Figure 4, Table 4), the median number of GPT-API calls we made for each application was about 2600 (1 call per file and follow ups). The median amount of data sent through these API calls is around 16MB and 3.3M tokens for each application. At publication time, the monetary cost of processing this volume of data using the GPT-4 API was about 8 USD per application. Costs may be further reduced through additional filtering steps, e.g., excluding from analysis files that clearly do not perform I/O.

False positives of unit testing. Through the repurposed unit testing, Wasabi reported 63 bugs in total. Our investigation shows 21 of them to be false positives.

There are 5 false positives in Wasabi's report of HOW bugs. In all 5 cases, applications caught the injected exception, wrapped it inside a general exception, and the general exception lead to a crash. These failures were wrongly labeled as HOW bugs by Wasabi's "different exception" oracle. Future work can prune these false positives by analyzing the exception-propagation and the exception-wrapping chain.

There are 8 false positives in Wasabi's report of missing-delay WHEN bugs. For all these cases, Wasabi's judgement is correct — the application indeed did not take a break between consecutive retry attempts. However, our manual inspection found that the delay may not be necessary as small changes were made between consecutive retry attempts. For example, in HDFS, when a file-block fetching fails, the retry will ping a different replica node. In this case, we conservatively consider Wasabi's bug report as a false positive.

Finally, there are 8 false positives in Wasabi's report of missing-cap WHEN bugs. In these cases, the application actually has a cap for the number of retry attempts, and chooses to propagate the exception to an upper level after the cap is hit. However, the upper level turns out to be the test harness, which ignores the exception and continues the unit test. Making things worse for Wasabi, in these unit tests, the method associated with the fault injection handler is invoked by the test harness many times to handle different tasks. As a result, the 100 retry attempts were hit across many tasks and triggered a false missing-cap bug report. Future work that makes Wasabi more aware of the call context should be able to resolve most of these false positives.

False positives of static bug detection. WHEN bug detection using GPT-4 reports 60 cases that do not appear to be actual retry bugs. In 29 cases, GPT-4 labels non-retry-related files as containing retry. For example, the prompt that asks GPT-4 to differentiate poll- or lock-behavior from retry is not always successful. In 16 cases, the false positive appears to be caused by limitations of single-file input: for example, a retry reported to be missing delay, but does call a sleep-containing helper method defined in a different file. Lastly in 15 cases, GPT-4 appears to wrongly comprehend code behavior. For example, identifying a missing cap when there is indeed an explicit comparison and exit condition on attempts.

IF bug detection using CodeQL incorrectly reports one case: it declares `FileNotFoundException` to be retried in 1/4 cases, when it is actually never retried. The wrong outlier result is due to ancilliary boolean variable-based control flow not analyzed by our script.

4.4 Other results

If Wasabi did not use keyword filtering to support CodeQL, it would have reported 3.5x more retry loops across 8 applications (i.e., 725 vs. 205). Manual check of these cases indicates that most, if not all, are not related to retry: these loops may iterate through lists of items, poll for status updates, or repeatedly execute a periodic task; catch blocks may be used to simply track or log errors, or ignored; and exceptions may be informative rather than represent transient errors.

The three test oracles (§3.1.3) are crucial for Wasabi unit testing. Without these test oracles, there will be a large number of additional false negatives (i.e., all missing-delay bugs and the majority of the missing-cap bugs will be missed), and false positives. For example, about 90% of test crashes encountered by Wasabi testing are caused by unit tests rethrowing the injected exceptions, and are correctly filtered out by Wasabi test oracles.

4.5 Discussion

Mitigating false positives. Most of Wasabi's unit testing false positives may be removed through further analysis of the call and exception contexts, as discussed above.

Many of the static detection false positives may be removed by collating the results of static detection with unit testing results. For code segments not covered by unit tests, other avenues for reducing false positives include: 1) appending the content of a method M callee function from a different file into the prompt referencing M, or 2) reducing the token size of large files using prompt-compression techniques [35, 36]. For the specific cases where GPT mistakenly identifies non-retry code to be retry-related, the effect of false positives may be mitigated by presenting every bug report in two parts: which code snippet is considered as retry (easily reviewable by developers), and a description of the bug. We also expect the accuracy of our LLM-based static analysis to improve with future LLM models.

Note on false negatives. It is difficult to precisely measure the false negatives of a bug-detection tool. Looking at the root-cause categories listed in Table 2, those "missing or disabled retry mechanism" bugs are not covered in the design of Wasabi and will cause false negatives. Furthermore, if a bug is caused by software mis-configuration, which happens to about 10% of the cases in our dataset, it will be missed by Wasabi unless the unit tests use the same mis-configured setting. A false negative could occur even for a bug whose root cause is covered by Wasabi when 1) existing unit tests do not cover related code and/or 2) the retry location is missed by CodeQL and the LLM.

Broader system design considerations. Some design considerations would improve the quality of retry and system at large. For one, the systems we studied display an overall lack of consistency in encoding retry-errors: applications will retry based on error-code in some instances and on exceptions in others (even within the same file); wrap exceptions in an ad-hoc way; or use too-general errors, making accurate retry-or-not decisions difficult. Retry structures are also widely inconsistent—a single application might include a range of unique local implementations of queue or state-machine based retry. Reducing variance of retry structures and error definitions would help improve the correctness and maintainability of retry-related code.

Another consideration is that application testing frameworks and conventions are ill-suited to isolated and systematic retry testing. For example, tests will frequently disable retry or enforce restrictive timeouts, which are difficult to override without tedious workarounds; or use error-mitigating procedures in test harness code that conceal genuine retry problems. More flexible testing frameworks with built-in support for retry-test configurations would reduce the burden of implementing comprehensive retry tests.

5 Related Work

Error handling is one of the main root causes of production failures [26, 27, 45, 65]. Some past work used static analysis to check error specifications [34], error propagation [29, 42, 58], and task cancellation anti-patterns [59]. Others used dynamic analysis and, in particular, fault injection [3–6, 8, 9, 11, 26, 30, 37, 39, 40, 47–52, 55, 61, 67]. These techniques are *not* tailored to automatically expose retry-related bugs. Consequently, they differ from Wasabi in several aspects.

First, Wasabi faces the unique challenge of identifying retry-specific rather than general error handling problems. As discussed earlier, traditional program-dependency based static analysis is not suitable to identify retry functionality. Meanwhile, code constructs related to generic error handling like exception blocks, yield a wide instrumentation code space where most of the code is not related to retry.

Second, by targeting retry bugs, Wasabi has different fault injection policies regarding where and what exceptions to inject as well as different test oracles from previous works. For example, prior work injects domain-specific faults like network partitions, disk faults, database read/write inconsistencies [3, 5, 26, 37, 39, 48, 50, 61], targets specific system components like cluster management controllers [61] and microservices [51], or focuses on certain failure scenarios like crash-recovery [26, 37, 47] and crash-consistency [3, 52, 55], which are typically triggered by non-recoverable instead of recoverable errors. Others ask users to specify or customize injection rules [4, 11, 30, 40, 67] or test oracles [8, 9], instead of automating these tasks like Wasabi.

This difference also applies to recent research [10, 42]: focusing on different types of bugs from Wasabi, their design differs from Wasabi's in terms of where and how to inject faults, how to judge the existence of a bug, and the types of tests used to drive fault injection.

Rainmaker [10] targets bugs triggered by transient errors in applications interacting with cloud services via REST APIs. Rainmaker intercepts API calls at the HTTP layer. Its fault injection follows a taxonomy of HTTP-specific bug patterns and code-coverage metrics. Its oracles use existing test assertions to detect issues such as inconsistent exceptions, silent state divergence, and unhandled transient errors.

Legolas [42] focuses on partial failures in distributed systems, also called gray failures. Like Wasabi, but unlike Rainmaker, it injects faults at the application level by throwing exceptions. However, it differs from Wasabi in almost all other aspects of the design because the two tools target different types of bugs. Specifically, Legolas strategically injects faults to maximize a system's abstract state coverage, whereas Wasabi injects faults at retry locations. It uses existing system-crash and gray-failure checkers to judge whether a bug has occurred, unlike the retry-specific oracles used by Wasabi. To catch gray failures, Legolas needs to use system tests instead of unit tests like Wasabi.

The rise of LLMs has brought opportunities to software engineering research, with an emergent set of papers applying LLMs to code generation [46, 53, 66], testing [12, 38, 41], repair [24, 33, 64], analysis [13, 43, 44, 54], summarization [1, 2], and documentation [60]. Wasabi is orthogonal to these works by using LLMs to pinpoint a common yet unstructured functionality—retry logic—in large codebases.

Much work is dedicated to studying failures in cloud systems [7, 25, 27, 31, 45, 59, 62]. Some papers introduce new taxonomies for bugs in cloud and distributed systems [25, 27, 45], while others focus on new or understudied classes of bugs like metastable failures [7, 31], cancellation issues [59], or cross-system defects [62]. Our paper sheds light on retry bugs and provides insights on why retry functionality is difficult to implement, test, analyze, and reason about.

6 Conclusion

Retry is a widely-used functionality to handle transient failures frequently encountered by software systems. This paper introduces Wasabi, a novel suite of techniques that detect common retry problems using repurposed unit testing and static checking, guided by a comprehensive study of retry bugs found in popular distributed applications. This work highlights the potential of combining complementary static, dynamic and LLM-based approaches to identify and improve retry implementations.

7 Acknowledgments

We thank the reviewers for their insightful comments and Junfeng Yang for shepherding this work. The authors' research is supported by NSF (CNS-2313190, CCF-2119184, CNS-1956180), the Chameleon Cloud Project [57], an Eckhardt Fellowship, and two University of Chicago Quad Undergraduate Research grants.

References

[1] Toufique Ahmed and Premkumar Devanbu. 2023. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (, Rochester, MI, USA,) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 177, 5 pages. https://doi.org/10.1145/3551349.3559555

[2] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. 2024. Automatic Semantic Augmentation of Language Model Prompts (for Code Summarization). In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE) (ICSE'24)*. IEEE Computer Society, Los Alamitos, CA, USA, 1004–1004. https://doi.ieeecomputersociety.org/

[3] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 151–167.

[4] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 51–68.

[5] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 331–346. https://doi.org/10.1145/2723372.2723711

[6] Radu Banabic and George Candea. 2012. Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys'12)*. Association for Computing Machinery, New York, NY, USA, 281–294. https://doi.org/10.1145/2168836.2168865

[7] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. 2021. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 221–227. https://doi.org/10.1145/3458336.3465286

[8] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE way to test openflow applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, USA, 10.

[9] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2021. CoFI: consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE'20)*. Association for Computing Machinery, New York, NY, USA, 536–547. https://doi.org/10.1145/3324884.3416548

[10] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. 2023. Push-Button Reliability Testing for Cloud-Backed Applications with Rainmaker. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023 (Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023)*. USENIX Association, USA, 1701–1716.

[11] Maria Christakis, Patrick Emmisberger, Patrice Godefroid, and Peter Müller. 2017. A general framework for dynamic stub injection. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE'17)*. IEEE Press, 586–596. https://doi.org/10.1109/ICSE.2017.60

[12] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA'23)*. Association for Computing Machinery, New York, NY, USA, 423–435. https://doi.org/10.1145/3597926.3598067

[13] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024. TRACED: Execution-aware Pre-training for Source Code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (, Lisbon, Portugal,) *(ICSE'24)*. Association for Computing Machinery, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/3597503.3608140

[14] Apache Cassandra Docs. Accessed: April 2024. https://cassandra.apache.org/doc/stable/cassandra/configuration/cass_yaml_file.html.

[15] Apache Cassandra Docs. Accessed: April 2024. https://www.elastic.co/guide/en/elasticsearch/hadoop/8.13/configuration.html.

[16] Apache HDFS Docs. Accessed: April 2024. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[17] Apache Hive Docs. Accessed: April 2024. https://cwiki.apache.org/confluence/display/Hive/Configuration+Properties.

[18] Apache HBase Docs. Accessed: April 2024. https://hbase.apache.org/book.html.

[19] Apache MapReduce Docs. Accessed: April 2024. https://hadoop.apache.org/docs/r3.1.0/hadoop-project-dist/hadoop-common/core-default.xml.

[20] Apache MapReduce Docs. Accessed: April 2024. https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml.

[21] Apache Yarn Docs. Accessed: April 2024. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/yarn-service/Configurations.html.

[22] CodeQL Documentation. Accessed: April 2024. https://codeql.github.com/docs/.

[23] Polly documentation. Accessed: April 2024. https://www.pollydocs.org.

[24] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE'23)*. Association for Computing Machinery, New York, NY, USA, 1229–1241. https://doi.org/10.1145/3611643.3616243

[25] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. 2022. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 126–141. https://doi.org/10.1145/3542929.3563482

[26] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: a framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 238–252.

[27] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670986

[28] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service

Outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC'16)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/2987550.2987583

[29] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. 2008. EIO: error handling is occasionally correct *(FAST'08)*. USENIX Association, USA, Article 14, 16 pages.

[30] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. 2016. Gremlin: Systematic Resilience Testing of Microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 57–66. https://doi.org/10.1109/ICDCS.2016.11

[31] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22) (OSDI'22)*. USENIX Association, Carlsbad, CA, 73–90. https://www.usenix.org/conference/osdi22/presentation/huang-lexiang

[32] Netflix Hystrix. Accessed: April 2024. https://github.com/Netflix/Hystrix.

[33] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE'22)*. Association for Computing Machinery, New York, NY, USA, 1219–1231. https://doi.org/10.1145/3510003.3510203

[34] Suman Jana, Yuan Kang, Samuel Roth, and Baishakhi Ray. 2016. Automatically detecting error handling bugs using error specifications. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 345–362.

[35] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 13358–13376. https://doi.org/10.18653/v1/2023.emnlp-main.825

[36] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2024. LongLLMLingua: Accelerating and Enhancing LLMs in Long Context Scenarios via Prompt Compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 1658–1677. https://aclanthology.org/2024.acl-long.91

[37] Xiaoen Ju, Livio Soares, Kang G. Shin, Kyung Dong Ryu, and Dilma Da Silva. 2013. On fault resilience of OpenStack. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (Santa Clara, California) *(SOCC'13)*. Association for Computing Machinery, New York, NY, USA, Article 2, 16 pages. https://doi.org/10.1145/2523616.2523622

[38] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (ICSE'23)*. 2312–2323. https://doi.org/10.1109/ICSE48619.2023.00194

[39] Kyle Kingsbury and Peter Alvaro. 2020. Elle: inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.* 14, 3 (nov 2020), 268–280. https://doi.org/10.14778/3430915.3430918

[40] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 399–414.

[41] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (ICSE'23)*. 919–931. https://doi.org/10.1109/ICSE48619.2023.00085

[42] Ao Li, Shan Lu, Suman Nath, Rohan Padhye, and Vyas Sekar. 2024. ExChain: Exception Dependency Analysis for Root Cause Diagnosis. In *Proceedings of the 21th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024 (Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024)*. USENIX Association, USA.

[43] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. Assisting Static Analysis with Large Language Models: A ChatGPT Experiment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 2107–2111. https://doi.org/10.1145/3611643.3613078

[44] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Pasadena, California, USA) *(OOPSLA'24)*. Association for Computing Machinery, New York, NY, USA.

[45] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What bugs cause production cloud incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS'19)*. Association for Computing Machinery, New York, NY, USA, 155–162. https://doi.org/10.1145/3317550.3321438

[46] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems (NeurIPS'23, Vol. 36)*, A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.). Curran Associates, Inc., 21558–21572. https://proceedings.neurips.cc/paper_files/paper/2023/file/43e9d647ccd3e4b7b5baab53f0368686-Paper-Conference.pdf

[47] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP'19)*. Association for Computing Machinery, New York, NY, USA, 114–130. https://doi.org/10.1145/3341301.3359645

[48] Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.* 2, POPL, Article 46, 24 pages. https://doi.org/10.1145/3158134

[49] Paul D. Marinescu, Radu Banabic, and George Candea. 2010. An extensible technique for high-precision testing of recovery code. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 23.

[50] Paul D. Marinescu and George Candea. 2009. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems and Networks*. 379–388. https://doi.org/10.1109/DSN.2009.5270313

[51] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC'21)*. Association for Computing Machinery, New York, NY, USA, 388–402. https://doi.org/10.1145/3472883.3487005

[52] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 13th USENIX*

Conference on Operating Systems Design and Implementation (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 33–50.

[53] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in Translation: A Study of Bugs Introduced by Large Language Models while Translating Code. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE) (ICSE'24)*. IEEE Computer Society, Los Alamitos, CA, USA, 866–866. https://doi.ieeecomputersociety.org/

[54] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?. In *Proceedings of the 40th International Conference on Machine Learning* (, Honolulu, Hawaii, USA,) *(ICML'23)*. JMLR.org, Article 1144, 25 pages.

[55] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All file systems are not created equal: on the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 433–448.

[56] AspectJ Maven Plugin. Accessed: April 2024. https://www.mojohaus.org/aspectj-maven-plugin/.

[57] The Chameleon Cloud Project. Accessed: September 2024. https://chameleoncloud.org/.

[58] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI'09)*. Association for Computing Machinery, New York, NY, USA, 270–280. https://doi.org/10.1145/1542476.1542506

[59] Utsav Sethi, Haochen Pan, Shan Lu, Madanlal Musuvathi, and Suman Nath. 2022. Cancellation in Systems: An Empirical Study of Task Cancellation Patterns and Failures. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, Carlsbad, CA, 127–141. https://www.usenix.org/conference/osdi22/presentation/sethi

[60] Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman Nath. 2023. 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* (Providence, RI, USA) *(HOTOS'23)*. Association for Computing Machinery, New York, NY, USA, 87–93. https://doi.org/10.1145/3593856.3595910

[61] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 143–159. https://www.usenix.org/conference/osdi22/presentation/sun

[62] Lilia Tang, Chaitanya Bhandari, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. 2023. Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys'23)*. Association for Computing Machinery, New York, NY, USA, 433–451. https://doi.org/10.1145/3552326.3587448

[63] The Wasabi Toolkit. Release: September 2024. https://github.com/bastoica/wasabi.

[64] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE) (ICSE'23)*. 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[65] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 249–265.

[66] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2023. Multilingual Code Co-evolution using Large Language Models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (, San Francisco, CA, USA,) *(ESEC/FSE'23)*. Association for Computing Machinery, New York, NY, USA, 695–707. https://doi.org/10.1145/3611643.3616350

[67] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying tests to validate exception handling code. In *2012 34th International Conference on Software Engineering (ICSE) (ICSE'12)*. 595–605. https://doi.org/10.1109/ICSE.2012.6227157