THE UNIVERSITY OF CHICAGO

IMPROVING TASK-RESILIENCE MECHANISMS IN SOFTWARE SYSTEMS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
UTSAV SETHI

CHICAGO, ILLINOIS
DECEMBER 2024

ABSTRACT

Modern software necessarily implements task-level mechanisms designed to anticipate and handle transient errors. Such mechanisms include retry, cancellation, checkpointing and timeout, and they are critical to the smooth operation of almost every type of software application - especially the distributed and large scale applications in use everywhere today.

At the same time, these mechanisms are not trivial to implement correctly, and prone to defects, for a variety of reasons: they require nuanced handling of partial execution states, are contingent on difficult-to-determine timing and error-handling policies, use non-standard implementations that are not well supported by existing libraries or frameworks, and are frequently disabled or excluded from application testing. Broken implementations are common and often result in severe software issues.

This dissertation aims to analyze and detect problems in two widely-used mechanisms: cancellation and retry. It conducts empirical studies of real-world problems associated with cancel and retry, and guided by these studies, develops approaches to detect policy and implementation-related problems in these mechanisms using static and complementary large language model (LLM) aided program analysis techniques. These techniques find hundreds of problems in popular open-source distributed applications.

ACKNOWLEDGMENTS

There are very many people without whose help and support this journey would not be possible. First of all, I would like to express my sincere gratitude to my advisor, Shan Lu, whose dedication, thoughtfulness and insight have been a constant source of inspiration and admiration, and whose encouragement was invaluable for my joining this research program in the first place and navigating all its twists and turns. I also want to also express my gratitude to committee members Madan Musuvathi, who provided critical exposure to real-world applications of software analysis research, and guided me on how to think about research problems creatively and constructively; and Haryadi Gunawi, whose feedback and support during my candidacy and in the latter part of this program sharpened and improved this work.

I would also like to thank my many collaborators and colleagues. Bogdan Stoica, Suman Nath, Junwen Yang, Jonathan Mace, and Haochen Pan were just a few among many who helped greatly with this work, and provided invaluable advice and good company along the way. And, it has also been a privilege to be part of a department with such wonderful faculty and staff, the interactions with whom were always helpful and meaningful.

Finally, I want to thank my family, whose support and love mean the world to me. I wish to dedicate this work to them.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS iv

LIST OF FIGURES vii

LIST OF TABLES viii

LIST OF LISTINGS ix

1 INTRODUCTION 1
1.1 Motivation 2
1.2 Thesis contribution 3
1.2.1 Understanding cancel and retry issues 4
1.2.2 Detecting cancel and retry problems 4
1.3 Outline 5

2 STUDY AND DETECTION OF TASK-CANCEL PROBLEMS 7
2.1 Introduction 7
2.2 Background 10
2.3 Methodology 13
2.4 Why Do Applications Cancel Tasks? 15
2.5 Root Causes of Cancel-Related Bugs 17
2.5.1 Cancel-initiation bugs 18
2.5.2 Cancel-propagation bugs 21
2.5.3 Cancel-fulfill bugs 27
2.5.4 Discussion: cancel mechanisms 33
2.6 Symptoms of Cancel-Related Bugs 36
2.7 Task Cancel Anti-Patterns 38
2.7.1 Unhandled Interrupt Exception (Java). 39
2.7.2 Interrupt API Misuse (Java). 41
2.7.3 Cancel not propagated to dependent tasks (Java) 42
2.7.4 Ignored cancellation tokens in loop (C#) 42
2.7.5 Token not passed - .NET analyzer (C#) 43
2.7.6 Anti-pattern limitations 43
2.8 Summary 44

3 STUDY AND LLM-AIDED DETECTION OF RETRY PROBLEMS AND IMPLEMENTATIONS 45
3.1 Introduction 45
3.2 Understanding Retry Issues 47
3.2.1 Methodology 47

3.2.2 IF retry should be performed . . . 49
3.2.3 WHEN retry should execute . . . 53
3.2.4 HOW to execute retry . . . 55
3.2.5 Other study findings . . . 56
3.3 Detecting retry locations in source code . . . 57
3.3.1 CodeQL retry-location detection . . . 57
3.3.2 LLM assisted retry-location detection . . . 60
3.3.3 Comparison: Retry Code Identification and Coverage . . . 61
3.4 Detecting retry bugs via static code analysis . . . 63
3.4.1 IF bug detection using CodeQL . . . 63
3.4.2 WHEN bug detection using GPT-4 . . . 67
3.5 Discussion . . . 69
3.6 Summary . . . 71

4 ADDITIONAL INVESTIGATIONS . . . 72
4.1 Making software documentation more useful using LLMs . . . 72
4.2 From comments to predicates . . . 74
4.2.1 Task Overview & Design. . . . 74
4.2.2 Evaluation . . . 76
4.3 From comments to locking rules . . . 77
4.3.1 Task Overview & Design. . . . 77
4.3.2 Evaluation . . . 80
4.4 Performance of latest models . . . 81

5 RELATED WORK . . . 83

6 CONCLUSIONS AND FUTURE WORK . . . 87

REFERENCES . . . 90

LIST OF FIGURES

2.1 Anti-pattern instances found in Java and C# applications 38

3.1 Retry code structures identified. 62
3.2 GPT-4 prompts for location and bug detection 68
3.3 Retry bugs reported by GPT-4 detection 68

4.1 Comments on parameters and exception-throwing conditions 73
4.2 An Example Prompt and Codex output 75
4.3 Codex output under an alternative prompt 76
4.4 Prompt for identifying locking rules 78
4.5 A comment for which GPT-4 performs worse 82

LIST OF TABLES

2.1 Task constructs and cancellation mechanisms 10
2.2 Applications included in our study 14
2.3 Reasons underneath Cancel-Feature Requests (CFR) 16
2.4 Cancel-related bugs: root causes 20
2.5 Cleanup issues breakdown 29
2.6 Cancel-related bugs: symptoms 37

3.1 Applications included in our study 47
3.2 Root causes of retry bugs 48
3.3 IF policy-outlier detection results 65

4.1 Tasks explored in this work 72
4.2 Specification Translation Precision and Recall 76
4.3 Accuracy in identifying comments that contain locking rules 80

LISTINGS

2.1 Handling cancel requests in Java 11
2.2 Handling cancel requests in C# 12
2.3 An example of late cancel 23
2.4 An example of dropped delivery 24
2.5 API Misuse Example 24
2.6 One type of invisible token 26
2.7 Unhandled Interrupt Exception CodeQL Script (partial) 40
3.1 Wrong Retry Policy - Recoverable error is not retried 50
3.2 Wrong Retry Policy - Non-recoverable error is retried 51
3.3 Wrong Retry Policy: Canceled task is retried 52
3.4 Missing delay between retry attempts 54
3.5 CodeQL script example: location detection 59
3.6 Example loop found by GPT but not CodeQL (Elasticsearch) 63
3.7 Retry IF bug detection: CodeQL script 64
3.8 Bug found by IF script: KeeperException not retried 66
5.1 Retry-framework example (Resilience4j) 85

CHAPTER 1

INTRODUCTION

It is inevitable that a software application will encounter unanticipated errors during the course of its operation - the sources of which may be hardware faults, software bugs, network issues, broken configurations, bad user input, and the like. Often these errors will arise in the middle of executing an application task, and cannot be entirely forseen or prevented, thus requiring the task to handle them in real-time.

To do so, software necessarily implements mechanisms specifically designed to handle and respond in real time to errors at the task execution level. Such mechanisms include retry, cancellation, checkpointing and timeout, and they have accordingly become ubiquitous in today's large-scale distributed and concurrent software applications.

Because these "task-resilience" mechanisms - as this work refers to them - are responsible for ensuring either successful completion of a task (if an error is recoverable), or alternatively the safe termination of a task without affecting the system at large, the quality and reliability of applications are highly dependent upon their correct implementation. Unfortunately, however, implementing them correctly is non-trivial, broken implementations remain common, and these broken implementations can result in severe or catastrophic impacts [37, 30, 57].

This work tries to remedy some of these problems through an in-depth investigation of two important task-resilience mechanisms - retry and cancellation. It conducts an empirical study of issues related to retry and cancel, proposes techniques to identify retry- and cancel-related problems using static program analysis, and evaluates how these techniques can be complemented and enhanced by large-language-model (LLM) informed software analysis.

1.1 Motivation

Retry and cancel are widely used to enable the resilient and flexible operation of modern distributed and concurrent software. Retry, long used by and familiar to any software developer, is a last line of defense: when an application encounters an error which is transient, simply re-executing the failed task with minimal or no modifications will often succeed.

For cases where an error is not recoverable or difficult to anticipate, cancel provides another avenue of handling: an application may decide (or give users an option) to cancel a task and its subtasks while minimizing effects on the broader system [64]. In addition, cancel can be used by applications to respond to adverse conditions before they explicitly become errors: for example, canceling resource-intensive tasks during periods of resource strain, or canceling low-priority conflicting tasks during a higher-priority operation (such as failover) [61].

Yet, despite their importance and seeming simplicity, retry and cancel are challenging to implement correctly in software systems, for a variety of reasons. They must not only handle events or errors that occur in the middle of task execution, but also ensure continuing regular operation of the system from these intermediary states. Understanding what intermediate state changes to revert and how can be challenging and error-prone [57]. Moreover, they frequently need to be coordinated across multiple dependent or related tasks, e.g. canceling dependent tasks when a parent is canceled. They may be implemented in ad-hoc ways, whose locations are difficult to identify using structural elements and to analyze. And it is not sufficient that code be syntactically or semantically correct: developers must choose correct policies - e.g. whether an error is transient and worth retrying - which are not always straightforward from code context.

Compounding these challenges is the fact that such functionalities are not straightforward to test: they often depend on transient errors or events which occur rarely and are

difficult to faithfully simulate and specially observe in today's unit testing frameworks. In addition these functionalities may not be viewed as a priority for testing since they deal with resilience behaviors and not core functionality. As a result they are often omitted from testing, resulting in regressions and unpredictable behavior in production code.

The results of incorrect behavior are various and often severe. Broken retry can lead to stuck jobs, large-scale performance degradation and service crashes or system failure. And broken cancel commonly results in resource leaks, performance issues, broken task APIs, and data corruption or loss [57, 8].

Despite their commonality in software, research on retry- and cancel-specific problems and attempts to improve them have been limited. Frameworks to support correct and configurable retry and cancel [66, 78] are unable to support all the diverse real-world modes that implementation of these functionalities may take. Nor are existing bug-detection tools designed or able to specifically target the correctness requirements and locations of many retry and cancel problems. Underlying all this is a gap in systematic and in-depth understanding of retry- and cancel-specific problems in modern systems.

1.2 Thesis contribution

Given the importance of these mechanisms and the challenges to correctness outlined above, this dissertation aims to provide a specific, systematic understanding of root cause problems in retry and cancel functionality with the goal of identifying principles and techniques for improving their correctness.

1.2.1 Understanding cancel and retry issues

The first step to improving these functionalities is to develop a broad and systematic understanding of issues that arise from broken implementations in real-world applications.

This dissertation conducts comprehensive studies on a large number of real-world cancel and retry bugs among 8 and 13 widely used open-source applications, respectively.

More specifically, this work contains an in-depth study of 156 cancel issues and introduce a new taxonomy of cancel problem types, based on cancel phase: **Initiation** (identifying a cancel event), **Propagation** (sending a cancel request to a task) and **Fulfillment** (ceasing execution safely).

Analogously, this work studies 70 retry issues across 8 popular Java open-source applications and classifies problems according to a taxonomy appropriate to retry: **IF** a retry should be performed in the first place; **WHEN** or how many times a retry is performed; and **HOW** implementations actually perform retry.

These studies yield interesting insights on cancel and retry bug patterns, and connections between problem types and differing modes of cancel and retry implementation. The findings motivate the solutions in the following section.

1.2.2 Detecting cancel and retry problems

Static techniques for cancel anti-pattern detection. Based on our study findings we develop a set of cancel anti-patterns which are indicative of various types of problematic cancel implementations. These anti-patterns correspond to problems in all phases of cancel mentioned above, as well as different language-supported cancel implementations.

Using these anti-patterns, this work implements static checkers that find hundreds of problems in recent versions of applications.

Static techniques for retry locations and bugs This section develops and evaluates traditional static techniques - e.g. using control-flow, data-flow and syntax analysis - for detecting loop-based retry implementations and a subset of retry-or-not problems (**IF**) as identified by our study, in these implementations. It also highlights limitations of these techniques.

Large language model-aided techniques for bug detection Retry is frequently implemented in ad-hoc ways that do not conform to any identifiable code patterns, limiting the ability to analyze them using traditional static techniques.

Recently developed large language models (LLMs) have created new opportunities to analyze program semantics using non-structural elements such as comments, context and naming conventions. So we also develop LLM prompts that successfully identify non-loop retry code locations, and evaluate the performance of LLM prompts at identifying additional policy-related (**WHEN**) bug types.

1.3 Outline

The core contributions of this dissertation are split into two major chapters. Chapter 2 describes a comprehensive study of real-world cancel problems, introduces terminology related to cancel as well as patterns of implementations across languages, and gives a description of anti-patterns and solutions. Chapter 3 similarly describes our study into retry problems, provides insights about the varying modes of retry implementations, and describes techniques and scripts for retry location identification and bug detection. Chapter 4 introduces additional investigations into LLM-informed software analyses for system reliability tasks. Chapter 5 introduces related work, followed by a conclusion and discussion of future work in Chapter 6.

Some of this work has also been published in conference papers: the material in Chapter

2 has been published in OSDI 2022 [83]; the material of Chapter 3 in SOSP 2024 [85]; and the material of Chapter 4 in HotOS 2023 [86]. In addition, there are other related projects I have participated in alongside this work, including an in-depth study and detection of data constraint problems in web applications (ICSE 2020 [94]); and tackling data inconsistency in mixed-version distributed systems (SOSP 2021 [101]). More details about those projects can be seen in corresponding papers.

CHAPTER 2

STUDY AND DETECTION OF TASK-CANCEL PROBLEMS

Modern software applications rely on the execution and coordination of many different kinds of tasks. Often overlooked is the need to sometimes prematurely terminate or cancel a task, either to accommodate a conflicting task, to manage system resources, or in response to system events or errors that make the task irrelevant. In this chapter, we study 62 cancel-feature requests and 156 cancel-related bugs across 13 popular distributed and concurrent systems written in Java, C#, and Go to understand why task cancel is needed, what are the challenges in implementing task cancel, and how severe are cancel-related failures. Guided by the study, we generalize a few cancel-related anti-patterns and implemented static checkers that found many code snippets matching these anti-patterns in the latest versions of these popular systems.

2.1 Introduction

Task cancellation is critical to the performance and availability of modern concurrent and distributed systems. Unlike fault handling, which reacts to the failure of a software or hardware component, task cancellation proactively stops the execution of a software component (i.e., a task) that no longer needs to run. Concurrent applications use task cancellation for better resource management, task coordination, and system responsiveness [64, 17, 61, 18]. For instance, when a user aborts a long-running operation, the underlying system may want to cancel the relevant tasks to save resources; when a high-priority request comes, a busy system may want to cancel a low-priority task for the greater good. Task cancellation is crucial for today's systems that concurrently execute a large number of complex and resource-consuming tasks under stringent quality of service requirements.

Unfortunately, supporting efficient and correct task cancellation in modern applications

is nontrivial. Tasks need to be designed such that they can be aborted at certain points of execution without undesirable side-effects (e.g., without corrupting the system state). Moreover, the application needs to decide when to safely cancel a task, and once decided to cancel, the decision needs to be correctly propagated to the target task to be canceled.[1] Last but not least, a system may contain dozens or hundreds of concurrent tasks, with complex dependencies among the tasks as well as on the system environment. If not carefully implemented, canceling a task may break a dependency or introduce concurrency errors such as races. It is therefore not surprising that implementing task cancellation can be error-prone.

As it stands, there have been no studies on task cancel problems in concurrent and distributed systems—how cancel is used and implemented, the various types of cancel-related bugs, the impact of those cancel-related bugs, and so on, although various other types of bugs and problems have been heavily studied for distributed systems [31, 58, 95, 50, 101].

This chapter attempts to provide an in-depth analysis of cancellation usage and problems in popular software applications across multiple languages, which we hope will help guide cancellation-related systems research and design.

Why do applications cancel tasks? To understand why cancellation may be desirable to system operation, we reviewed 62 *feature requests* in 13 popular open-source applications, such as HBase, Hive, Cassandra (Java); Roslyn, ASP.NET Core (C#); CockroachDB, and InfluxDB (Go).

We found that about half of the cancel-feature requests aim to terminate tasks that no longer produce useful results upon a change in system or user state (e.g., the finish of a related task and the end of a user session); close to half of the requests aim to improve operational flexibility and enable users to cancel a job, particularly the time-consuming ones, at any time; a small number of requests aim to enable stopping a low-priority task prematurely to support the launching and running of other more important tasks.

1. This is in contrast to fault-handling where the external environment decides *when* a fault is generated.

Our study confirms our understanding that task cancellation is a crucial feature that facilitates efficiency and operation flexibility in concurrent systems. It shows that the trigger of a cancel can be a variety of events (far beyond system shutdown and component failures), and the target of cancellation is often a small number of selective tasks (rarely bulk cancellation), which can all bring complexity to the implementation of task cancellation.

What causes cancel-related bugs? To understand the challenges in implementing task cancellation correctly, we studied 156 bug reports across the same set of 13 popular open-source applications in Java, C#, and Go to understand what are common cancellation-related bugs.

Our study shows that problems routinely occur at all phases of cancel: 1) deciding when and which task to cancel (about one third of the bugs), 2) propagating the cancel request from the initiator to the target task (about one quarter of the bugs), and 3) fulfilling the cancel in the target task (about one third of the bugs). Some classes of problems are particular to the type of mechanism used to issue cancel, such as bugs in the use of Java's `interrupt` API, and bugs in passing cancellation tokens through function parameters in C# and Go. Many other classes of problems are due to the overall complexity of implementing cancel, such as determining which tasks conflict, which system state changes must be reverted before task termination, etc. For each type of bugs, we discuss potential solutions to tackle them.

Impacts of cancel-related bugs. The impact of cancel bugs varies, but can in some cases be severe. Among issues with specified symptoms, a few common categories are resource leaks, performance issues, broken task APIs, data corruption or loss, and incorrect user reporting.

Cancellation anti-patterns. Through the study above, we have generalized and implemented static checkers for five cancel-related anti-patterns using the CodeQL [25] static analysis framework, including (1) missing interrupt handling inside a loop (Java); (2) using the wrong built-in API to check or reset the interrupt flag on threads (Java); (3) failure to

	Task	Task Cancellation
C#	`Task`,`Thread`	`CancellationToken` struct
Go	`goroutine`	`Context` type
Java	`Thread`	`interrupt()` on `Thread` itself

Table 2.1: Task constructs and cancellation mechanisms

propagate cancel to child tasks (Java); (4) ignoring cancel-token parameters (C#); and (5) not propagating cancel tokens (C#)[2]. We find around 200 instances of these anti-patterns across the latest versions of the 13 applications we studied, which further motivates future work to improve the support for correct cancel implementation.

2.2 Background

Task. This thesis defines a task as a unit of concurrent execution. As summarized in Table 2.1, in Java, all code that implements a `Runnable` interface qualifies (e.g., `Thread`). In C#, tasks are objects of type `Thread` or `Task`. In Go, execution inside a `goroutine` is a task [64, 28, 69]. Tasks are not limited to any specific programming model: for example, some issues we study involve tasks as part of an event-driven design. Some tasks execute with a clear end, like a user-request task launched by a server application; some execute with an open end and cease only on system shutdown or explicit request to terminate, like a task that provides an in-memory cache service for others. Tasks can also initiate work on other nodes, e.g. by issuing an RPC call.

Task Cancel. Cancel is the deliberate attempt of one task to terminate another task in a *cooperative* way. We will refer to the former as the cancel initiator and the latter as the cancel target. All the instances of cancel we study are *cooperative*, which means that the target task, upon receiving the request, chooses how and when to terminate [47]. Note that the alternate way of task cancel - *abortive*, where the initiator forces the target to

2. This particular checker is a re-implementation of an existing C# checker.

```
1   public void run() {
2        try { ...
3        } catch (InterruptedException e) {
4        // receiver handles the cancel request
5        }
6        ...
7        if (Thread.currentThread().isInterrupted()) {
8        // receiver handles the cancel request
9        }
10   }
```

Listing 2.1: Handling cancel requests in Java

terminate - is prone to semantic errors and is not supported by the three languages that our study focuses on (Java, C#, Go). For example, the abortive Java `Thread.stop()` method is deprecated now.

Cancel vs Fault Handling. Task cancel and fault handling have some similarities in that they both involve a task finishing earlier than expected, but they also have fundamental differences. Cancel can be considered part of the regular operation of the system: the conditions that cause cancel to be issued are known and expected with some regularity, such as to proactively prevent performance problems, as we will discuss in Section 2.4; the cancel process involves the cooperation between at least two running parties, the initiator and the target; after the cancel is conducted, the system is expected to remain functioning as normal or even at a higher capacity. This is in contrast to failure handling, in which failure events are unexpected; the handling is reactive after a component failure; and the expectation for system functioning may be lower - e.g. to function at reduced capacity, or to terminate safely.

Cancel mechanisms. Although the built-in cancel mechanisms in C#, Go, and Java take different forms, as listed in Table 2.1, they all essentially offer a "flag": the initiator sets the flag when requesting cancel, and the target can check the flag and respond to the cancel request.

Specifically, in Java, any thread can execute `t.interrupt()` to set an internal flag of thread `t`. Any code executing in thread `t` can use APIs like `isInterrupted()` to check this

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;
var mytask = Task.Run(() => {
// the receiver checks the token before starting
// to handle a potential cancel request
    ...
    if (token.IsCancellationRequested) {
    // receiver handles the cancel request
    }
}, token);
```

Listing 2.2: Handling cancel requests in C#

flag and see if an cancel request has been delivered to it. Alternatively, any execution of a blocking API, like `sleep()` or `poll()`, will throw an `InterruptedException` upon the setting of its thread's cancel flag, as shown in Listing 2.1.

C# and Go offer more flexible ways of cancel. Instead of limiting each thread to have one flag, they allow the software to declare any number of `CancellationToken` structs (C#) or `Context` variables (Go) that each contains a cancel flag. In C#, a `CancellationToken` object, generated from a `CancellationTokenSource` is typically passed through function parameters. An invocation of `Cancel()` on the token's source would set the flag inside the token object, which is visible to any task that has access to the token, as illustrated in Listing 2.2. Cancel in Go is similar: the `Context` type provides a `CancelFunc` to issue a cancel signal, which can be checked via ctx.Done() on the Context ctx. Like `CancellationToken`, `Context` is typically passed via function parameters. In the remainder of this chapter, we will refer to `Context` variables also as cancellation tokens for simplicity.

The `CancellationToken` in C# also allows registering a callback function to be called when the token is canceled. This functionality is rarely used in the applications that we study and hence will not be discussed.

Finally, developers can implement custom means of cancel. In many Java programs, shared Boolean variables are used as cancel flags. Threads explicitly read and write these flags to carry out cancel. This essentially allows multiple cancel flags for one thread and hence can embed more semantic information inside each flag. However, it is also prone to

bugs, as we will discuss in Section 2.5.

2.3 Methodology

Application selection. We study applications written in three different languages: Java, C# and Go, as shown in Table 2.2. These languages were chosen as they have widespread use of different built-in cancel mechanisms, and as such provide a useful point of comparison for this study.

In choosing which Java applications to study, we focus primarily on the most popular, as indicated by GitHub stars, open-source distributed applications in various categories, as listed in Table 2.2. Our selection is more limited for Go and C#, since there are much fewer applications written in these two languages on GitHub. For Go, we study applications that are analogous to categories studied in Java: InfluxDB and CockroachDB (distributed databases), and etcd (distributed application serving and coordination). For C#, there do not exist any widely-used applications in those categories. So, as an alternative, we chose the top 2 applications/frameworks, out of the 50 most popular C# applications on GitHub, that utilize cancel extensively: Roslyn (compiler suite) and ASP.NET core (web framework).

Cancellation Issue Study. For these selected applications, we checked their Jira issue trackers or GitHub issue-and-pull systems, if they do not use Jira. We searched for *resolved* and *valid* issues, up to June 2021, using the following keywords: *abort*, *cancel*, *interrupt*, and *terminate*. We then manually checked the reports to exclude issues that do not have a clear description or are unrelated to task cancel.

From the remaining, we get 156 issues that are labeled by developers as "bug" or are clearly fixing a bug, although not labeled. They will help us understand the root causes and symptoms of cancel-related bugs, as presented in Section 2.5 and 2.6. We should note that although an issue might belong to multiple root causes or symptom categories, it is classified by its primary category only, **without double-counting**. In addition, we study

Table 2.2: Applications included in our study

Application	Category	Stars	Bugs	CFR[2]
Java (distributed apps)				
Cassandra	Database	7K	14	2
Elasticsearch	Full-text search	57K	15	20
Hadoop[1]	Distri. storage; Distri. processing	12K	10	3
HBase	Database	4K	26	3
Hive	Data warehousing	4K	21	5
Kafka	Stream processing	20K	9	2
Solr/Lucene	Full-text search	4K	9	2
Spark	Data processing	31K	6	6
Java - subtotal			110	43
C# (single-instance apps)				
ASP.NET Core	Web framework	26K	6	1
Roslyn	Compiler	15K	14	8
C# - subtotal			20	9
Go (distributed apps)				
CockroachDB	Database	22K	12	6
etcd	Key-value store	38K	8	0
InfluxDB	Database	22K	6	4
Go - subtotal			26	10
Total			156	62

[1] Including Hadoop Common, HDFS, YARN, MapReduce
[2] Cancel-Feature Requests

62 issues that are requests to add the capability of canceling some tasks and are labeled as "improvement" or "feature", instead of "bug", and contain patches approved or already merged. They will help us understand the motivation of task cancel, as in Section 2.4.

We believe cancel problems are under reported, as cancel code can be difficult to exercise during testing. From the discussion in cancel-feature requests, we also see that the complexity in correctly implementing task cancel sometimes drives developers away from implementing cancel, which of course comes with performance and efficiency loss.

Threats to validity. Our study does not cover all task cancel mechanisms, and may not generalize to those issues and systems not covered in our benchmark suite. Particularly, we have skipped those cancel-feature requests and cancel-related bugs whose description is not clear enough for us to conduct further categorization. We may also have missed cancel-related requests or bugs whose reports do not contain the search keywords used by us. Furthermore, since there are many more issue reports and pull requests about adding cancel features than those about cancel-related bugs, we limit our study of cancel-feature requests to those that contain cancel-related keywords in the issue/pull titles. Thus, we likely have missed many requests that have those keywords in the issue/pull body, but not the title.

2.4 Why Do Applications Cancel Tasks?

To understand why tasks may require cancel and what triggers a task cancel, we studied 62 cancel-feature requests in Java, C#, and Go systems, following the methodology described in Section 2.3, and generalized three main reasons for task cancel as shown in Table 2.3.

Reason-A: Efficiency. Close to half of the cancel-feature requests originate from developers' efficiency concerns, as the computation of a task T no longer produces useful results upon (A1) a system shut-down, (A2) a user-session termination, or (A3) a particular system or user event. Among these three different cancel-trigger scenarios, A3 is the most common and triggers cancel at a finer granularity than A1 and A2. For example, when a user nav-

Table 2.3: Reasons underneath Cancel-Feature Requests (CFR)

Why should a task T be canceled?	**#CFR**
A. Efficiency: T no longer produces useful results	**30**
- A1. Upon system shutdown	5
- A2. Upon a user disconnection or time-out	6
- A3. Upon a system or user event	19
B. Flexibility: T is no longer wanted by users	**28**
- B1. Cancel through an API call	20
- B2. Cancel through user interface or keyboard	7
- B3. Cancel through timeout parameter	1
C. Priority: More important tasks need to run	**4**
Total	**62**

igates away from a web page P, the system still runs many tasks related to the user, but can cancel all the tasks initiated by page P (e.g., influxdb-19029); when one attempt of a task finishes, all other speculative or parallel attempts of this task can be canceled (e.g., SPARK-25773 and roslyn-8050); when a job is canceled or finished, its related tasks can be canceled (e.g., roslyn-25620 and roslyn-51816). In all these cases, continuing the execution of T does not affect functional correctness but wastes system resources and affects request latency, and which can be detrimental to system resilience.

Reason-B: Flexibility. Another common reason is to offer users the flexibility to prematurely terminate a user operation and all its related tasks, which contribute to about 40% of the cancel-feature requests. In a number of cases, the requests explicitly mention that the target task may take a long time (e.g., elasticsearch-72644 and elasticsearch-73818 and SOLR-6122) or even hang for unknown reasons (e.g., KAFKA-1506), and hence should be cancellable. In other cases, the exact reasons why a user may want to cancel a task is not explained. The requested cancel features typically get implemented as task-cancel commands or as handlers of certain user interface events, like the Ctrl+C keyboard combination.

Reason-C: Priority. Interestingly, sometimes, developers want to enable the system to sacrifice T for the benefit of other more important tasks. For example, in HDFS-2507, a

feature is added to cancel an ongoing checkpoint task of a standby NameNode when the active NameNode fails. This would allow the standby NameNode to immediately start the fail-over task instead of waiting for the long checkpointing to finish, minimizing the system downtime. Similar decisions of sacrificing long-running low-priority tasks for the benefit of high-priority tasks also occur in other systems (e.g., CASSANDRA-14397, elasticsearch-56009).

Observations. *Trigger variety.* A task cancel can be triggered by a variety of events, as shown in Table 2.3. This variety adds complexity to the implementation of cancel: the program may miss a trigger and fail to initiate the cancel. Even when a trigger is sensed, the trigger information may not be included in the cancel request, e.g., in Java's built-in cancel mechanism, making it difficult for the cancel handler to process the cancel request properly.

Fine granularity. Task cancel is often targeted; bulk cancel scenarios like system shutdown are rare. This fine granularity can make it difficult to decide which task to cancel.

Heavy coordination. In a system that involves many concurrent components, cancel may involve a lot of coordination across tasks: a task's cancel could be due to the launch, the progress, or the termination of another task. This heavy coordination requirement demands careful synchronization and shared-state clean-up during task cancel.

Proactive instead of reactive. Unlike fault handling, task cancel rarely reacts to an already exposed component failure. It is more about the system efficiency, request latency, operational flexibility, and resource balancing, which, although do not immediately precipitate system outages, are crucial to the service quality and robustness.

2.5 Root Causes of Cancel-Related Bugs

We divide the whole procedure of cancel into three phases, and categorize cancel bugs' root causes accordingly:

1) *Initiating Cancel* - the cancel initiator senses a cancel-trigger event and decides which task to cancel.

2) *Propagating Cancel* - the cancel request propagates from the initiator to the target.

3) *Fulfilling the Cancel* - the cancel target responds to the cancel request, releasing resources, restoring system states, and ending its own execution.

Note that there are 9 bugs caused by miscellaneous semantic errors that are not related to the core functionality of task cancel. We put them in the "Other" category in Table 3.2 and skip discussion about them below.

2.5.1 Cancel-initiation bugs

As discussed in Section 2.4, a variety of conditions might trigger a cancel. Deciding when to initiate a cancel to which target task is complex and susceptible to problems, contributing to about 30% of cancel-related bugs (Table 3.2).

In some cases, a cancel is not initiated when it should be, either because the system completely overlooks a cancel trigger ("Overlooking triggers") or because the system checks the existence of a cancel trigger incorrectly ("Broken trigger checking"). In other cases, a cancel is incorrectly or unnecessarily initiated ("Excess cancel"). We describe each type in more detail below.

Overlooking triggers

This type of bug occurs when a cancel should be initiated upon a specific trigger, but no logic exists to do so. This is the most common type of cancel-initiation bug, contributing to more than 20% of all the cancel-related bugs.

The most common scenario is that a running task T is canceled or has failed but a dependent task, which is no longer necessary, is not canceled. As an example, in SPARK-21738, expensive jobs would continue to run on a Spark cluster even after a user session was closed, wasting computation resources to produce irrelevant results. While Spark does provide support for canceling jobs, the system did not realize that a session closure should

be treated as a trigger for job cancel.

As another example, in roslyn-1086, the failure of a compilation task will prevent a "completion" event from ever being published to an event queue, while a task listening to the queue, *AnalyzerDriver*, will continue to run and wait for the event which will never arrive. The solution in this case was to include a reference to the *AnalyzerDriver* in the compilation task, which is canceled via cancellation token upon compilation failure.

Other types of triggers could also be overlooked. For example, in CASSANDRA-8805, developers realized that the launch of high-priority tasks like *repair* often gets blocked by long-running low-priority tasks like *index-summary redistribution*, as these tasks access *sstables* in a conflicting way and cannot run in parallel. To solve this problem, developers added the logic to allow any *repair* to check for and cancel any running *index-summary redistribution* tasks.

Note that bugs of this type share similar root causes with those cancel-feature requests for efficiency or priority reasons, which were discussed in Section 2.4. The difference seems to be the impact: the ones that cause more severe failure symptoms are reported as bugs, instead of feature requests.

The patches to these bugs are straightforward: adding the logic to initiate a cancel upon the occurrence of the trigger.

Lessons learned. A fundamental challenge here is to track the dependency relationship among all the concurrent tasks, a daunting task in modern concurrent and distributed systems: which tasks conflict with each other and cannot run in parallel; which tasks depend on which task and hence should not continue if the latter is canceled; which tasks are redundant copies of which task and hence should not continue if the latter finishes successfully; etc. In all systems that we have checked, this is conducted in an ad-hoc way. There is an unmet need for coherent tool/framework and possibly programming language support for capturing these dependencies.

Table 2.4: Cancel-related bugs: root causes

Root Cause Category	Java	C#	Go
Buggy cancel initiation			
- Overlooking triggers	22	3	9
- Broken trigger checking	7	0	0
- Excess cancel	7	1	0
Buggy cancel propagation			
- Untimely delivery	15	3	4
- Dropped cancel	17	5	2
Buggy cancel fulfill			
- Cancel not checked	8	0	4
- Cancel not carried out	6	0	0
- Defective cleanup	23	5	6
Other	5	3	1

One particular type of dependency, the parent-child relationship, is feasible to track through static program analysis. Consequently, we can build a static checker to automatically identify code snippets where the parent task is canceled, and yet no cancel is initiated towards the children tasks. We will present more details about this checker in Section 2.7.3.

Other types of dependencies, like *repair* versus *index-summary redistribution* or a speculative task versus the original task, depend on application-specific semantics and are much harder to track systematically. We noticed that these semantic-rich dependencies are often centered on some key shared data, like the `sstables` that are updated by conflicting tasks or the common `job-ID` shared between multiple job attempts (e.g., HIVE-12307). Consequently, future work may automatically infer task dependencies by analyzing access patterns on key data.

Broken trigger checking.

Sometimes, the program anticipates the existence of a trigger. However, it checks the trigger occurrence in a wrong way. For example, in SOLR-10525, if a duplicate task is submitted

while a previous instance of a task is still running, the previous instance should be canceled. However, the logic to recognize whether a previous instance of a task is running is incorrect and so a cancel is never issued, leading to the execution of duplicate tasks.

Lessons Learned. Many bugs of this type are related to checking whether a particular task is running. Often, the task performing the check does not have a direct reference to the task under check, and hence needs to refer to an intermediary, like a shared collection of task status. The logic to store and retrieve the task status information is custom implemented in each system and hence prone to bugs: some accesses to the task registry are not thread safe; different types of tasks may store their information in different ways in the collection and hence got mis-checked later; etc. Some standard library support would help.

Excess cancel.

Converse to "Overlooking triggers", sometimes triggers are correctly sensed and yet tasks are wrongly or unnecessarily canceled. For example, upon the launch of a task T, the software may incorrectly cancel tasks that are actually not conflicting with T (CASSANDRA-13142, CASSANDRA-15024) or tasks that are indeed conflicting but have higher priority than T (HBASE-17674). Upon the finish of a task T, the software may incorrectly cancel tasks which are related to T but whose results are still needed (roslyn-11470, HADOOP-6762).

Lessons Learned. Similar as "overlooking triggers", these bugs originate from the challenge of tracking the dependency among tasks. Future research should study how to track which tasks conflict with or depend on each other, potentially through data dependency analysis.

2.5.2 Cancel-propagation bugs

Once a cancel trigger is correctly sensed and the cancel target is correctly identified, the initiator issues a cancel request. For about a quarter of the cancel-related bugs in our study,

the propagation from the initiator to the target went wrong.

Untimely delivery

It is important that a cancel can be issued at any time to the cancel target without delays or mis-handling. However, this is often not the case when a custom cancel mechanism is used.

Cancel race. In many systems, a "task manager" is implemented to coordinate tasks and relay cancel requests: the cancel initiator notifies the task manager about its cancel request; the task manager then sends the request to the cancel target. In several Java and Go systems, such as Cassandra (CASSANDRA-9070), Spark (SPARK-4097), HBASE (HBASE-13146), InfluxDB (influxdb-9018), and etcd (etcd-8443), the implementation of task managers contain concurrency bugs that manifest when cancel is issued at a special moment, like shortly after the target task is submitted, or in parallel with another cancel request towards the same target. As a result of these bugs, cancel requests may be dropped.

Occasionally, such cancel-related concurrency bugs also occur when a standard cancel mechanism is used. For example, in aspnetcore-11757, a cancel initiator disposes a `CancellationTokenSource` right after it requests a cancel on the token. As a result, when the target task checks the token, a use-after-disposal error occurs.

Lessons Learned. It is alarming that similar cancel-concurrency bugs occur in so many different systems. On one hand, standard task-manager library support could help. On the other hand, existing concurrency bug detection and testing tools [27, 60, 45, 44, 54] should be applied to check the correctness of cancel-related implementation.

Late polling. As discussed in Section 2.2, many custom cancels are conducted through a shared flag variable. Unfortunately, without system support, such a cancel request cannot be delivered timely when the target task conducts frequent blocking operations. For example, Listing 2.3 illustrates a simplified version of bug SPARK-1582. A task checks whether a cancel is delivered to it at the beginning of every work-loop iteration through

```
// Cancel initiator
class Initiator {
  Task myTask;
  main() {
    ...
    myTask.cancelFlag = true;
  }
}

// Cancel recipient
class Task {
  public boolean cancelFlag = false;
  private BlockingQueue Bqueue;

  run() {
      while(cancelFlag == false) {
        ...
        Bqueue.take(); // blocks until an element is available
      }
  }
}
```

Listing 2.3: An example of late cancel (SPARK-1582)

a custom `cancelFlag` variable. Unfortunately, since every iteration of the loop executes a `BlockingQueue::take()` operation, the flag may not be checked for a long or even unlimited amount of time, causing severe delays in Spark job cancellation. Similar issues also exist in KAFKA-5697, KAFKA-5896, and others.

These problems are typically fixed by using a language built-in cancel mechanism instead of, or in addition to, the custom flag to carry out the cancel. In Java, the built-in `Thread.interrupt()` would terminate blocking operations such as `sleep()`, `BlockingQueue::take()`, and `poll()`, with an `InterruptException` thrown. In C# and Go, many system operations such as `sleep()` accept cancellation tokens as parameters, allowing the timely delivery of cancel.

Lessons Learned. The key takeaway here is to avoid using a custom cancel flag, particularly when the nearby code region conducts blocking operations. We can use static program analysis to identify these vulnerable custom-cancel loops and warn the developers. Having said that, the pervasive use of custom-cancel loops in Java programs is probably due to the limitation of Java's built-in cancel mechanism, which we will discuss more in Section 2.5.4.

```
// Cancel recipient
class Task {
  run() {
      ...
      commitSync() // interrupt lost inside commitSync
      ...
      if (isInterrupted()) {
        // cleanup steps here will not be performed
      }
  }
  commitSync() {
   sleep(1000); // unsets interrupted flag
    ...
    catch (InterruptedException ex) {
      // does not reset flag, cancel gets dropped
     }
  }
}
```

Listing 2.4: An example of dropped delivery (KAFKA-4375)

```
class Task {
  ...
  void checkStale() {
        ...
    // current thread is interrupted somewhere
     } catch (InterruptedException e) {
-      Thread.currentThread().interrupted(); // Wrong
+      Thread.currentThread().interrupt(); // Fixed
     }
  }
}
```

Listing 2.5: API Misuse Example (SOLR-8066)

Dropped cancel

Depending on the different cancellation mechanisms, a cancel request could be dropped before it propagates to the right target in different ways.

Cleared interrupt (Java). A tricky aspect of Java's built-in mechanism is that the interrupt received by a thread can be silently unset by methods along the call chain. As a result, the interrupt may fail to reach the code that is prepared to fulfill the cancel request, contributing to about 15% of cancel-related bugs in Java programs in our study.

For example, in KAFKA-4375, function `run` contains a well written cancel handler that stops child tasks and exits. Unfortunately, at run time, the cancel is often intercepted by the `sleep` method inside its callee `commitSync`, as shown in Listing 2.4. The Java `sleep` method, just like many other Java blocking methods, silently unset the interrupt and throw an `Interrupted Exception`. Without rethrowing the exception or resetting the interrupt flag, the interrupt is dropped before reaching the right handler in function `run`. Similar problems also occur in other systems, like HBASE-5243, HIVE-13858, HBASE-10650, HBASE-10651, HBASE-10652, etc. Patches for these bugs simply re-throw the interrupt in the catch block.

A related mistake is that developers sometimes get confused about a few similar Java APIs: `t.interrupt()` interrupts a thread `t`; `t.interrupted()` checks whether `t`'s interrupt flag is set *and* clears the flag; `t.isInterrupted()` conducts the same checking but *does not* clear the flag. When `interrupted()` is mistakenly used, the cancel could be dropped before reaching the intended cancel handler, as illustrated in Listing 2.5. This type of mistake occurred at multiple places across different systems (KAFKA-9415, KAFKA-5665,HBASE-10455, SOLR-8066). Patches for these problems are straightforward, as shown in Listing 2.5.

Lessons Learned. Many bugs of this type can be automatically detected. As we will discuss in Section 2.7.1 and 2.7.2, static checkers can search for the catch blocks of `Interrupted-`

```
// Cancel recipient
class SomeTask {
  private CancellationToken systemCancelToken;

  void doWork(CancellationToken userCancelToken) {
        ...
        libraryMethod(userCancelToken); // systemCancelToken invisible to
            libraryMethod
  }
}
```

Listing 2.6: One type of invisible token (aspnetcore-5936)

`Exception` that neither terminate the execution nor re-throw the exception, and search for incorrect use of the `interrupted()` API.

Invisible token (C#/Go). In C# and Go, once a cancel is issued on a cancellation token, the status of the token cannot be reverted. Consequently, the type of mistaken clearance in Java does not exist in C# or Go. However, a cancel request may still get dropped during its propagation: since the cancellation token is typically not a global object, developers need to pass the token through function parameters to ensure the token is available through the chain of method calls. If the token is not passed to a long-running function f, cancel would be greatly delayed until the execution returns to a caller of f that has access to the token. This contributes to close to 15% of cancel-related bugs in C# and Go.

Making things more complicated, unlike Java, C# and Go allow canceling a thread through different cancellation tokens, each representing different semantics—one token might represent requests from end users; one might represent requests from a periodic timer; and so on. As a result, programmers may pass some tokens to a function, but forget some others, causing certain cancel requests to be dropped, as shown in Listing 2.6. Note that, a function typically only allows one cancellation-token parameter. Consequently, the onus is on developers to be aware of what tokens exist in the current context and when or how to combine them into one token to pass to a callee function—not a trivial task.

Lessons Learned. This type of bug can be detected by static checkers: if a function f has a cancellation-token parameter, its caller function F should pass every cancellation token *tok*

visible in F to f. In fact, such a checker is included in the .NET SDK, a set of libraries that provide support for development for C#[65]. We apply this checker to the latest versions of ASP.NET Core and Roslyn, and report the results in Section 2.7.5.

2.5.3 Cancel-fulfill bugs

Once a cancel is correctly initiated and propagated to the target, the target task must process the cancel request, stopping its execution, releasing resources, and reverting or invalidating shared states so that other tasks, including a potential re-submission of the current task, can proceed correctly. This is unsurprisingly the most difficult aspect of cancel, contributing to about one third of all the bugs in our study.

Cancel not checked

Sometimes, a successfully delivered cancel request is not immediately checked by the target task, causing severe cancellation delays.

In Java, the complexity is that explicit cancel checking is not always needed. Once the internal cancel flag is set by the system, the target thread will throw an `InterruptedException` once it executes a blocking Java API like `sleep`, `poll`, and others. Consequently, if the target thread invokes some of these APIs from time to time, explicit checking is not needed. However, if a long-running code-region, like a loop, does not call any such APIs, explicit checks using APIs like `isInterrupted` or `interrupted` are needed. Lacking such explicit checks are the root causes behind several bugs in Java systems, like HIVE-16078 and HBASE-10575.

In C# and Go, similar problems occur if a long-running function never checks its parameter cancellation token.

Lessons Learned. For C# applications, we have implemented a static checker to detect this type of bug (Section 2.7.4). For Go applications, implementing an accurate checker is

difficult, as the `Context` variables contain many fields and could be used for many different purposes other than cancel. Automatically detecting this type of bug in Java programs is feasible. We leave this to future work.

Cancel not carried out

This type of bug occurs when the target task makes no attempt to stop its execution after it becomes aware of the delivered cancel request.

Our study has only seen this type of bugs in the context of the Java built-in mechanism. Specifically, an `InterruptedException` is thrown by a Java library API. This exception is caught by the caller function but the handling block is essentially empty. There are many bugs of this type (e.g., HBASE-3064, HBASE-10472, HIVE-15997, KAFKA-5833, KAFKA-1886).

Comparing with other cancel mechanisms, an `InterruptedException` contains the least semantic information—it is unclear which task initiated the cancel and for what reason. This may be why some of these catch blocks are empty.

Lessons Learned. Although the root cause here differs slightly from the "Cleared interrupt" bugs in Section 2.5.2[3], they both can be detected by a checker that searches for problematic catch blocks of `InterruptedException`, which we will discuss in Section 2.7.1.

Defective cleanups.

When responding to a cancel request, a task needs to not only stop itself, but also to release resources that it acquired earlier and clean up changes it made to shared data. Doing so in a coordinated, correct, and efficient way is challenging. Unsurprisingly, bugs that occur during this process are particularly common, contributing to more than 20% of all the bugs

3. The cancel-target task has no cancel handling across the call chain for bugs here, but has the right handling in a caller in "Cleared interrupt" bugs.

Table 2.5: Cleanup issues breakdown

	Count
What type of cleanup defect?	
- Incorrect: wrong API or cleanup semantics	10
- Incomplete: did not clean up all data	14
- Missing: no cleanup performed	4
- Unordered: clean up data in a wrong order	3
- Other	3
Where is data requiring cleanup located?	
- Heap	27
- Persistent data	7
How should data be cleaned up?	
- Invalidate, revert or reset data	13
- Release resource (lock, thread, etc.)	13
- Delete file from disk	2
- Other	6

in our study.

What went wrong? There are mainly four types of mistakes in a cancel cleanup, as shown in Table 2.5.

First, the cancel handler changes the values of some variables in an attempt at cleanup, but the resulting values lead to failures (10 bugs in our study). For example, in SOLR-8372, upon the cancel of a *recovery* task, the *update log* this *recovery* task has been working on should remain in "inactive" state until recovery is restarted. However the cleanup logic mistakenly puts the update log into "active" state, which had the serious consequence of potential data loss. The fix was simply not to make that state change.

Next is incomplete cleanup, where the task attempted to clean up data but did not do so comprehensively (14 bugs). For example, in CASSANDRA-7803, *compaction result* files were written during the *compaction* task. The files could be written in a regular location or a temporary location, depending on the configuration. The cleanup logic removed the regular files but not the temporary ones, which could quickly fill the disk and make the application

unusable.

Completely missing cleanup, where no steps are taken to clean up any data related to the task, occurred in 4 bugs. In HBASE-13877, a *TableFlushProcedure* task is canceled. However the task simply ceases execution without any additional steps taken. The data modified by the task (*Memstore Snapshot*) is not invalidated and may get reused by subsequent tasks, causing data corruption or data loss.

Finally, there are 3 bugs where the cleanup routine works on shared variables in an incorrect order, causing coordination problems with other tasks.

What data is at the center of defective cleanup? Unlike crash handling, cancel handling is carried out by the cancel target, an actively running task, and hence needs to clean up not only persistent but also heap data it has touched. In fact, for the majority of clean-up bugs (80%), heap, instead of persistent data, is the target of defective cleanup.

In our study, a canceled task T typically does not hold a close dependency with other running tasks—otherwise, T typically would not be canceled, or its dependent tasks would be canceled altogether. Consequently and fortunately, there is typically not too much heap data to clean. What needs to be cleaned are mainly low-level resources, such as locks or thread pools; or shared data structures related to system activities or persisted information. The latter includes things like task tracking, i.e. what tasks are running, have run, or about to run in the system, e.g. the `ZoneSubmissionTracker` object in Hadoop; pointers to persisted user data e.g. the `DataTracker` object in Cassandra, which maintains references to all database tables; and other system metrics or metadata, such as the `StorageMetrics` object in Cassandra which tracks disk usage, and the `RoutingNode` object in Elasticsearch, which maintains shard status information. This relatively focused target of cleanup may help future research to automate data cleanup.

Occasionally, a task which produces a large amount of intermediate results needs to be canceled. Fortunately, in most cases we have seen, the system already has a transaction-style

design, where all intermediate data is buffered in a cache. The cleanup only needs to update the cache meta-data correctly.

In the cases where persistent data is the target of defective cleanup, most often the data are temporary files local to a task, which are not properly deleted or invalidated. In three cases, however, the persistent data are shared by other system activities, and defects in cleaning up this data prevent the broader system from performing correctly.

What does the patch do? Most commonly, the patch releases resources, invalidates or reverts the data modified by the task. Releasing resources, such as locks, threads, and cancellation tokens, is straightforward. Often, the original task already has the correct resource release routine. However, upon a task cancel, that routine is short circuited. The patch simply makes sure the complete release routine is followed.

How to correctly invalidate or revert the data varies from case to case. Sometimes, the task needs not keep track of the modifications it has performed: for example, in CASSANDRA-5481, a task needs to reset a shared connection/cursor object on cancel, which does not require information about the history or the state of the task. But in other cases, a task must track information about modifications it has made: in CASSANDRA-15674, a task makes a single modification to *totalDiskSpaceUsed* on the shared *SystemMetrics* object, and should remember to decrement by this same value upon cancel. One challenge in performing this type of clean up is knowing, among the various heap data modified by a task, which requires cleaning and which type of cleaning.

Lessons Learned. As evidenced by the examples above, defective cleanups have severe consequences and are common. It is important to tackle these bugs.

Detecting the complete absence of cleanups is relatively easy. Whenever a cancel handler only ceases the execution and performs no cleanup, a warning should be issued. Some of these bugs can even be automatically fixed: in many cases, one just needs to re-throw the interrupt to the caller that contains the correct clean-up logic (e.g., HBASE-7711).

Some incomplete cleanups are caused by short-circuiting a correct clean-up routine. Particularly, exceptions may be thrown during the clean up, either due to unexpected task states or a system API hitting the original interrupt signal again. Incorrect handling of such a double-exception may skip the remainder of the cleanup routine, causing incomplete cleanups (HIVE-15997). Automated checkers can be developed to search for this type of bug.

Existing tools that detect resource leaks during exception handling [81] and cancellation-token leaks [63] can be applied to detect those resource leak problems.

Detecting incorrect cleanup or general missing cleanup is the most challenging and requires more research. One possible research direction is to consolidate cleanup steps to help detect and fix defective cleanups. In many bugs, the related cleanup steps were interspersed across the task. However, when they were combined or compared together, it was clear that they were not comprehensive or correct. Sometimes cleanup for one task should have been identical to another. For example, in SPARK-1396, a scheduler had two methods, *handleCancel* and *abortStage*. These should have performed the exact same cleanup steps, but for each method steps were implemented separately and non-comprehensively. The fix was to combine the cleanup logic so that it was shared. Or, the cleanup on task cancellation was very similar to the steps performed on task completion (e.g. removing a task from a registry when it is completed or canceled), and deficiencies were clear on consolidation.

Finally, given our observation that the target of cleanup is often a small set of system data structures, future research may use data-flow analysis to remind developers about what data should be cleaned, and to potentially synthesize invalidating/reverting methods for the small number of data structures that are the target of most cleanup.

2.5.4 Discussion: cancel mechanisms

Built-in mechanisms.

A natural question to ask is whether different built-in cancel mechanisms cause different cancel usage issues. Some types of bugs are common no matter what mechanism is used. For example, "overlooking triggers" contribute to 19% and 26% of bugs in Java and C#/Go, respectively; "defective cleanup" contribute to 20% and 24% of bugs in Java and C#/Go, respectively.

However, there are also many types of bugs that occur particularly often in Java systems, reflecting limitations of Java's built-in cancel mechanism:

1) "Cleared interrupt" bugs (Section 2.5.2) only occur in Java programs, as neither C# nor Go allows clearing an already issued cancel request. Note that, it is natural for Java to allow clearing a cancel signal received by a thread, because each thread has only one internal cancel flag no matter how many different cancel initiators and how many different cancel contexts there might be. This limitation also influences the next two types of bugs in Java.

2) The "Late polling" bugs (Section 2.5.2) in theory could exist in programs written in any languages, but were only seen in Java programs by us: the use of custom cancel-flag loops is very common in Java programs and yet very rare in C#/Go programs, probably due to the limitation of Java built-in cancel mechanism as discussed above.

3) "Cancel not carried out" bugs (Section 2.5.3) in theory could exist in programs written in any language, but were only seen by us in Java programs. We believe this is again related to the above limitation of Java cancel mechanism. In C# and Go, a nice effect of using a `CancellationToken` as one of a task's function parameters is that it makes clear from the function protocol that the task is designed to be cancellable. The rich semantics behind cancel tokens also helps developers decide how to treat each cancel request. In contrast, in Java, `interrupt()` is available on threads by default but there is no guarantee threads

respond to the interrupt, and indeed often do not.

Of course, the mechanisms in C# and Go are not perfect either. In addition to the common problems they face, such as "defective cleanup", they are particularly susceptible to "invisible token" problems (Section 2.5.2). Furthermore, the design of mixing cancel signals with other information in the `Context` variable in Go introduces challenges for both developers and researchers in designing cancel-related analysis tools.

Custom mechanisms.

Some of the systems we studied contain components specially built to assist with cancel functionality. These components offer features that may mitigate root cause cancel issues discussed previously, and so may be of interest. We share examples of a few such constructs here.

Cancellable Task interfaces. While Java threads by default provide a method to cancel tasks, i.e. built-in `interrupt()`, a few systems provide an alternative interface to be used by cancellable tasks. At a bare minimum these interfaces declare a "cancel" method that task developers must implement, in some cases encouraging developers to side-step built-in "interrupt" and associated problems.

For example, the *Interruptible* interface in Cassandra's "concurrent" package declares, in addition to the main task method `run()`, a method named `interrupt()` that requires implementation by developers. Though simple, this design advantageously makes explicit the task should be cancellable and actively requires cancel implementation, whereas for other task constructs, for example a generic thread, the need for cancel might not be apparent, and developers might not check for interrupts or passively ignore interrupt exceptions as we have seen. (And, an examination reveals all existing implementers of this interface do indeed handle cancel).

Some interfaces go further and include partial mechanism implementation. The ab-

stract class *CancellableTask* in Elasticsearch's *tasks* package provides a non-overridable, pre-implemented `cancel` method which sets a member field cancel flag `isCanceled` to false (and which task execution code should check). The class also includes the status method `isCanceled()`, which may help avoid misuse problems that occur when using the built-in API to check interrupted status. We must note, however, there is a downside to side-stepping built-in interrupt entirely: if the task uses built-in blocking Java methods - e.g. sleep - it will not be able to exit these methods prematurely, as we have seen.

Interfaces may also include post-cancellation methods that developers can implement to perform cleanup or other related tasks. *LifecycleTransaction* in Cassandra's *db* package provides, in addition to a cancel method, an `onAbort()` hook which is called after cancellation is processed. This may encourage developers to implement or consolidate cleanup logic, helping prevent missing or incorrect cleanup issues.

"Uninterruptible" interfaces. Conversely one system provides an "uncancellable" interface that allows users to run code sections without interruption: the *UninterruptibleThread* abstract class in Spark's "util" package allows users to define "uninterruptible" code sections that will complete in their entirety - if `interrupt()` is called on the thread, it will be suppressed until the uninterruptible code section completes. One area where this might be useful is for cleanup steps which must be executed in their entirety after the task is canceled: some issues we have seen arise from cleanup steps failing to complete due to interrupt during cleanup itself. An examination reveals that some implementations of this interface indeed use this functionality for cleanup. However, this design is susceptible to problems if not used carefully: if an uninterruptible code section uses an operation that blocks indefinitely, the thread may never respond to a cancellation request.

Task dependency tracking. One of the biggest categories of cancel issues is overlooking triggers, of which a common trigger is cancellation of a parent or associated task. Thus using constructs that track related or dependent tasks and help propagate cancel between them

may be valuable.

For example, some systems provide a task tracking service or "task manager" that maintains a list of scheduled or running tasks, usually by requiring that all task executions be launched through the manager. The task manager may additionally be designed to track task dependencies: e.g. the *TaskManager* shared class in Elasticsearch's "tasks" package require that submitted *Tasks* contain an "id" and "parentId". All task executions are initiated through the task manager using the manager's `register` or `registerAndExecute` methods. Running tasks and their children can thus be tracked and cancellations, which must also go through the manager (via `cancelTaskAndDescendants` method), can be propagated to all dependent tasks.

2.6 Symptoms of Cancel-Related Bugs

Not all the bug reports specify the exact failure symptoms. We categorize the ones that describe the symptoms in Table 2.6. As we can see, the symptoms vary, and can be severe.

Resource leaks. Resources acquired during task execution, including locks, buffers, and others, might not be released due to defective cleanup (Section 2.5.3). Furthermore, if a cancel does not take effect, the task thread itself may be leaked, which may be especially problematic if the thread pool has a fixed size. For example in SPARK-1582, work done by a Spark *Executor* thread was no longer needed, but a cancel was delayed (sometimes indefinitely) and the thread was not made available to perform other work.

Broken Task API. Unsurprisingly, incorrect cancellation might break the API used to submit or manage tasks. For example, in HDFS-12518, a critical task cannot be re-executed, due to the task not cleaning up its status when canceled. In SPARK-8132, *no* subsequent task for a multi-stage user job is able to be launched due to incorrect cleanup.

Data corruption/data loss. Many tasks might perform operations on user data, and a broken cancel can corrupt in-memory data used to service user requests, as well as cause

Table 2.6: Cancel-related bugs: symptoms

Symptom Category	Issues
Resource leaks	30
Performance issues	29
Broken task API	17
Data corruption/loss	5
Incorrect reporting	10
Unspecified	65
Total	156

persistent data to be lost - a very serious issue. For example, a silently dropped cancel signal in a callee led a caller to put incomplete (i.e. corrupted) in-memory values of user computations into a shared cache. Later user jobs would use these invalid values and give wrong results. (SPARK-1602).

Performance issues. While cancellation itself should generally lead to improved performance, as resources previously used by a task can be freed for other work, broken cancel handling can put the system in an unanticipated state that causes degraded performance or unresponsiveness.

In HIVE-13858 an interrupt signal was dropped, leading to an infinite loop in a task, which made access to a portion of system I/O impossible. This could cause unavailability of the entire cluster. Similarly, in CASSANDRA-11373, incomplete cleanup led to an infinite loop and CPU saturation.

In elasticsearch-75316, how frequently cancel would be used was underestimated, and inefficient cancel handling led to a 50x increase in latency for normal user requests. The patch was to make cancel handling more efficient.

Incorrect reporting to users. Lastly, mistakes in cancel functionality might lead to incorrect reports to users. For example a system might report to the user that a job has been canceled when in fact it was not (HIVE-14942, SPARK-18665, influxdb-13681). Or,

	HBase	Hive	Spark	Kafka	Solr	Cassandra	Hadoop	es	ASP.NET Core	Roslyn
Unhandled IE in loop (Java)	5	2	0	0	0	1	13	0	-	-
API misuse (Java)	3	2	0	7	5	0	0	0	-	-
Uncanceled child tasks (Java)	1	2	0	0	0	0	9	0	-	-
Ignored tokens (C#)*	-	-	-	-	-	-	-	-	34/112	120/179
Tokens not passed (C#)**	-	-	-	-	-	-	-	-	9	9

* Our analyzer result / CodeRush analyzer (simulated) result
** .NET analyzer (simulated) result

Figure 2.1: Anti-pattern instances found in Java and C# applications

conversely, the system might report that a job has not been canceled when indeed it has (SPARK-2666).

2.7 Task Cancel Anti-Patterns

Root causes of cancel bugs are varied and sometimes complex, but we find that a few types of bugs are associated with clear anti-patterns that are detectable by static code analysis. This section presents our experience of designing and evaluating a few anti-pattern checkers.

We have implemented a checker for each of the anti-patterns below using CodeQL [25], a publicly available static analysis tool. CodeQL takes as input *queries* which are a set of conditions on the application source code's call graph, control flow, dataflow graph and other information (e.g. object hierarchies). Queries are language specific, so for each anti-pattern and language, we have constructed a single query that describes the anti-pattern and can be run on all applications of that language, using CodeQL's command line tool or web interface. The results of queries are references to problematic section of source code (file and line number). The queries associated with each anti-pattern can be viewed at a publicly available repository [82].

Note that, code snippets that match an anti-pattern may not all cause severe failures, but are frequently harmful to the software in the long run if not fixed. We will discuss this in detail when we comment on the severity of each anti-pattern.

Also note that, these checkers mainly tackle low hanging fruits of cancel-related bugs, with more complicated bugs waiting to be tackled by future work. We are aware of similar checkers for the two C# anti-patterns, which we will discuss in details in Section 2.7.4 and 2.7.5. There may be similar checkers for the Java anti-patterns, although we are currently not aware of them. Our main goal here is to show that it is feasible to detect cancel-related code defects through simple static checking, and that many cancel-related defects exist even in the latest versions of these popular Java and C# applications.

2.7.1 Unhandled Interrupt Exception (Java).

Anti-pattern. An `InterruptedException` is caught inside a loop body, but in the catch block there is no handling - no control flow to exit the loop (i.e. no `break` statement, `return` statement or rethrown exception in the AST), and the interrupt flag is not reset via `t.interrupt()` on thread `t`. In addition, we also check via dataflow analysis that the thread is indeed interrupted somewhere in the codebase. A selection from the script is shown in Listing 3.5.

Rationale. This anti-pattern is closely related to "cleared interrupt" bugs (Section 2.5.2) and "cancel not carried out" bugs (Section 2.5.3). Its severity has been explained in these earlier sections. Note that, in this anti-pattern, we particularly look for problems inside a loop, as it is especially problematic there: without proper cancel handling inside a loop, a task may never cease execution or incur particularly long delays (HADOOP-6221,HBASE-3064).

Severity. There is one scenario where the impact of this anti-pattern may be mitigated: the program may use a custom cancel flag together with an `interrupt` call to cancel a task. In that case, an unhandled interrupt exception may not have a big impact, as long as the remainder of the loop iteration does not take long time to execute. Having said that, this type of implementation is still problematic and makes code maintenance difficult: what if an

```
import java
import semmle.code.java.dataflow.DataFlow

class Configuration extends DataFlow::Configuration {
  Configuration() { this = "Constructor Call to Interrupt Method Access
      Configuration" }

  override predicate isSource(DataFlow::Node source) {
    source
        .asExpr().(ConstructorCall).getConstructedType()
        .getASupertype*().hasQualifiedName("java.lang", "Runnable")
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call | call.getEnclosingStmt() = sink.asExpr().
        getEnclosingStmt() |
      call.(MethodAccess).getMethod() instanceof InterruptMethod
    )
  }
}

predicate propagatesCancelOrExits(RunMethod rm) {
    (exists(InterruptMethod im, MethodAccess ma |
        ma.getMethod() = im and
        ma.getEnclosingStmt().getEnclosingStmt() = cc.getBlock())
      or
      exists(Stmt exit | cc.getBlock().getAChild*() = exit |
        exit.(BreakStmt).(JumpStmt).getTarget() = lp
        or
        exit.(ContinueStmt).(JumpStmt).getTarget() = lp or
        exit.(ReturnStmt).getEnclosingStmt*() = lp.getBody()
      )
      or
      exists(cc.getBlock().getAStmt().(ThrowStmt))
    )
  )
}

from Configuration config, DataFlow::Node src, DataFlow::Node sink, RunMethod
    rm, IECatchClause cc
where
  config.hasFlow(src, sink) and
  exists(Expr ex | ex = src.asExpr().getAChildExpr*() |
    rm = ex.getType().(ClassOrInterface).getAMethod() or
    rm = ex.(FunctionalExpr).asMethod()
  ) and
  methodCalls(rm, cc) and
  not propagatesCancelOrExits(rm) and
  not hasExceptionHandling(cc) and
          isInsideLoop(cc)
select rm // to locate run() methods
```

Listing 2.7: Unhandled Interrupt Exception CodeQL Script (partial)

expensive operation is added near the end of the loop iteration? What if the task initiator deems the use of flag redundant in the presence of the interrupt call and removes the former?

Results. Our checker finds 21 cases of this anti-pattern in the latest versions of 4 Java applications in our benchmark suite (Table 2.1). Our manual checking of these 21 cases shows that 14 of them are truly instances of this anti-pattern; 2 of them are false positives (a corner case in CodeQL control-flow analysis misses the fact that the exception handler does stop the task execution); 5 of them may be considered false positives: the exception handler sets a flag, which defers the actual handling to a later point in the loop, which may or may not cause perceivable delay in the cancel handling.

2.7.2 Interrupt API Misuse (Java).

Anti-pattern. A thread calls `Thread.interrupted()` inside an `InterruptedException` catch block.

Rationale. This anti-pattern is inspired by a few API-misuse bugs discussed in Section 2.5.2 (e.g., Listing 2.5). When an `InterruptedException` is triggered by a library method in thread `t`, the interrupt flag is almost always cleared and should be reset by invoking `t.interrupt()` if the exception is to be handled by the caller. If a `t.interrupted()` is invoked instead, this is frequently a typo, as this API is designed to clear the interrupt flag, effectively a no-op inside the catch block. It may also be used inside a condition check, as it returns the status of the flag before clearing - `e.g. if (t.interrupted())`, - but when such checking occurs inside the catch block it is even worse, as library methods likely will have unset the flag before the check, and the logic inside the condition will never execute.

Severity. This API misuse can cause an interrupt to be dropped. Consequently, handling/cleanup logic that exists elsewhere may not be executed, causing functional problems.

Results. Our script finds 17 instances of this anti-pattern in 4 applications, as shown in Table 2.1. Our manual examination did not find any false positives.

2.7.3 Cancel not propagated to dependent tasks (Java)

Anti-pattern. A task instantiates a Java `Timer` and starts a child task (wrapped in a `TimerTask` interface) using a Java `Timer` object but does not cancel the `Timer` and `TimerTask`: either it does not maintain the reference to the `Timer` or it does not explicitly call `cancel()` on the `Timer` or `TimerTask`.

Rationale & Severity. This anti-pattern is related to some of the "Overlooking triggers" bugs discussed in Section 2.5.1. Java's built in `Timer` is one of the mechanisms used for scheduling single or periodic task executions on a separate thread. If the child task launched using the `Timer` (or `Timer` itself) is not canceled when the parent is canceled, then at a minimum, this lack of cancellation will leak resources. Note that, this anti-pattern focuses on `Timer`-based parent-child task dependency, because these type of child tasks are typically scheduled periodically and hence lead to more severe impact if not properly canceled.

Results. Our script finds 12 instances where a timer and associated tasks are started but not canceled. Three of these instances are false positives: in 2 cases, the reference to the `Timer` is embedded in a nested class, and hence is missed by our CodeQL-based static checking; in one case, the `Timer` task is only started during system shut down, and hence its leakage does not really cause problems.

2.7.4 Ignored cancellation tokens in loop (C#)

Anti-pattern. A method containing a loop accepts a `CancellationToken` parameter `ct`, but does not check the token via `ct.IsCancellationRequested`, `ct.CanBeCanceled` or `ct.ThrowIfCancellationRequested()`, anywhere inside a loop. Nor does it pass the token as an argument to any function calls inside the loop.

Rationale & Severity. The rationale of this anti-pattern has been discussed in Section 2.5.3. For a similar reason as discussed in Section 2.7.1, we focus on loops in this anti-pattern, for their bigger performance impact.

Results. Our analyzer found 154 cases of this anti-pattern (34 in ASP.NET Core and 120 in Roslyn). Manual checking finds 4 of these to be false positives: in 3 cases, a token is used via an indirect reference or reflection; in 1 case, a method that operates on a token instead of using it as a signal.

We also investigated a similar analyzer that is part of CodeRush [20], a popular debugging and code analysis extension for VisualStudio. The CodeRush analyzer warns if a token is not checked anywhere inside in a method. We have simulated the CodeRush analyzer using CodeQL and find 112 and 179 instances in ASP.NET Core and Roslyn, respectively. In one regard, our analyzer is stricter: if a token is checked somewhere in a method but not in a loop, our analyzer will flag it as a warning but the CodeRush analyzer will not. But, unlike the CodeRush analyzer, our analyzer does not check methods that do not contain loops.

2.7.5 Token not passed - .NET analyzer (C#)

We also applied an analyzer included as part of the .NET compiler platform (Roslyn). That Roslyn built-in analyzer checks if a `CancellationToken` is passed via parameter to a method M, but M does not pass the token to its calee C which optionally accepts a token parameter (optional arguments are a feature of the C# language). This anti-pattern is related to the "invisible token" bugs discussed in Section 2.5.2.

Simulating this anti-pattern using CodeQL, we find 9 instances each in the latest version of ASP.NET Core and Roslyn. Our manual checking finds no false positives.

2.7.6 Anti-pattern limitations

While these checkers have been inspired by and cover some of the bugs in our study, there are still many bugs that cannot be covered by our checkers, for various reasons. In some cases a bug manifests due to reasons logically different from those covered by our checkers: for example, a cancel is dropped due to a semantic bug in a custom mechanism, rather than

API misuse or an unhandled interrupt exception.

In other cases, conditions added to our antipatterns to reduce false positives thereby introduce false negatives: for example, we search for empty interrupt exception handling specifically inside loops, but empty handling outside loops can also cause bugs.

Finally, our checkers are designed around common usage patterns and may miss other valid forms of usage: for example, we assume a cancel-supporting method is one that accepts a context or token explicitly as a top-level parameter; our checkers will ignore methods where the context or token is passed implicitly, say as a member field of another parameter.

2.8 Summary

Task cancellation is critical to the efficiency, availability, and operational flexibility of concurrent systems. This chapter presents a comprehensive study about how task cancel is used and what type of bugs are related to task cancel in popular distributed and concurrent systems written in Java, C#, and Go. This study reveals the complexity of implementing correct and efficient task cancel, and motivates future research to offer better system support for task cancel.

CHAPTER 3

STUDY AND LLM-AIDED DETECTION OF RETRY PROBLEMS AND IMPLEMENTATIONS

Retry—the re-execution of a task on failure—is another common mechanism to enable resilient software systems. Yet, despite its commonality and long history, retry remains difficult to implement and test in modern systems.

Guided by a study of real-world retry issues, this chapter proposes a set of static techniques to detect retry locations and associated problems in software systems. In particular, we find that the ad-hoc nature of retry implementation in software systems poses challenges for traditional program analysis but can be enhanced by complementary large language model-based techniques.

3.1 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 is also a source of 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 [33, 40, 99, 15], 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 [37, 49, 32, 57].

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 [38, 66]. 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 do not provide 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.

The goal of this chapter is to 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 snippets in these incidents, we observe a broad diversity in

Table 3.1: 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

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.

Tackling retry bugs. Guided by these findings, we take a first step in enhancing the reliability of retry logic by developing static techniques enabled by large language models (LLMs) to identify retry locations and detect various types of retry-related bugs (IF and WHEN problems mentioned above) in both loop- and non-loop related retry.

In all, our techniques identify 86 distinct, previously unknown retry bugs in 8 Java applications across all three types of retry-bug root causes.

3.2 Understanding Retry Issues

3.2.1 Methodology

Our study looks into popular open-source distributed applications written in Java that cover various categories as listed in Table 3.1. For every application, we search for retry-

Table 3.2: 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

related issues, using a set of keywords (*retry*, *resubmit*, *reattempt*, and *reschedule*) in their issue-tracking systems (Jira or Github issue-and-pull system). We only look at issues that (1) are labeled by developers as *bugs*, *resolved*, and *valid*, (2) have been fixed or have a patch awaiting to merge,

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 3.2. We discuss each category in details below and we also discuss the typical failure symptoms associated with each type.

While we aimed to select a representative set of applications, the conclusions of our issue study may not generalize to other applications and systems. Also 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.

3.2.2 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.

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 3.1), it is not a surprise 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 the initialization (Issue Kafka-6829).

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

```
1
2 class CommitResponseHandler {
3   void handle(Error e, Future future) {
4     if (e == COORDINATOR_LOAD_IN_PROGRESS ||
5     +   e == UNKNOWN_TOPIC_OR_PARTITION
6     ) {
7         future.raise(RetryException());
8         return;
9     } else {
10        future.raise(DoNotRetryException());
11        return;
12    }
13  }
14 }
15
16 class ConsumerCoordinator {
17    void commit() {
18        ...
19        sendCommit(msg, new CommitResponseHandler())
20    }
21 }
```

Listing 3.1: Wrong Retry Policy - Recoverable error is not retried. +: the lines headed by '+' indicate developers' patch; the same applies to all figures in this chapter. (KAFKA-6829, *Queue-based mechanism*)

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 sync and was labeled as a critical high-severity bug requiring 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`

```
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 3.2: Wrong Retry Policy - Non-recoverable error is retried (HADOOP-16683, *Loop-based mechanism*)

as a non-recoverable error and does not retry it as shown in Listing 3.2. However, other code paths in Hadoop may wrap `AccessControl Exception` 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. 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 3.3.

```
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 3.3: 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 3.2).

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.

3.2.3 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 OOM errors.

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 3.4 illustrates such a problem in a state-machine based retry from HBase.

In `UnassignProcedure`, which redistributes "regions" among servers (a core functionality of the system), the `REGION_TRANSITION_DISPATCH` state involved a call to `markRegions-AsClosing`, 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 will move on to the next state `REGION_TRANSITION_FINISH`. However no delay was implemented between retries, which due to the rapid nature of failure would clog the `StateMachineExecutor` and prevent other procedures from making progress (and could not be fixed by restarting, as procedures retain state upon restart). A critical fix had to be deployed to implement a delay between retries, as shown in the listing.

```
1 public class UnassignProcedure extends Procedure {
2   void execute(State currentState) {
3     switch(currentState) {
4       case REGION_TRANSITION_DISPATCH:
5           try {
6               markRegionAsClosing();
7               // proceed to next state
8               setState(REGION_TRANSITION_FINISH);
9           } catch (Exception e) {
10              // Fix adds delay before implicit retry
11          +   long backoff = (1000 * Math.pow(2, attemptCount))
12          +   Thread.sleep(backoff);
13              return; // implicit retry
14          }
15      case REGION_TRANSITION_FINISH:
16          //...
17    }
18  }
19 }
```

Listing 3.4: Missing delay between retry attempts (HBASE-20492, *State-machine procedure*)

How to catch these bugs? Issues in this category can be detected by statically checking if a delay-related API (e.g. `Thread.sleep`) is invoked between two retry attempts. The key challenge is to identify which code snippets are conducting retry — no standard API is used for retry, as shown in earlier listings.

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? To statically detect cases of too many retry attempts (i.e. most issues in this category), the challenge is in identifying which code snippets are conducting retry and reasoning about the termination conditions of retry.

3.2.4 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 static program analysis.

3.2.5 Other study findings

Bug severity. The bugs' priorities as labeled by developers suggest that many of them are problematic or of extremely-high severity: 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 forms of more complex 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 3.3); 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 3.4).

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

3.3 Detecting retry locations in source code

The main challenge when detecting retry locations is that retry code does not have a unique code pattern for traditional program analysis to search for. For example, loop-based retry (listing 3.2) 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 3.1, listing 3.3) and state machines (listing 3.4) 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 both leverage non-structural code elements like variable names and comments, which we observed to be much better indicators for retry functionality than program structure alone.

3.3.1 CodeQL retry-location detection

The first technique identifies loop retry using control-flow analysis and naming conventions. (Note that, we leave the detection of non-loop retry to a second technique below - as we discuss later, there is no effective way to detect non-loop retry using traditional control/data-flow analysis.) The technique uses CodeQL [25], a query-based static analysis tool, to perform the analysis. The particular query used is shown in Listing 3.5.

The query identifies all loop structures (i.e. `for` and `while` statements in the application) using the built-in CodeQL type `LoopStmt`, and analyzes these structures for the following syntactic and control-flow conditions:

1. **Performs exception handling** (lines 1-3). Checks if the loop catches any exception

E (indicating exception-based failure checking and retry). `InterruptedException` is excluded as this built-in Java method does not typically indicate a failure.

2. **Allows loop re-entry from at least exception handling block** (lines 5-22). Analyzes loop control-flow to determine whether the header is reachable from at least one catch block, indicating a potential retry path. For example, the loop header in listing 3.2 is reachable from the catch block on Line 14, (although not from the catch block on Line 8).

3. **Contains retry-named keywords** (lines 24-31). Checks to see if the loop body or loop condition contains any string literals, variables, or methods whose name includes "retry" or "retries". For example, in listing 3.2, the loop also contains a counter variable named `retry`. We discuss the importance of keyword usage in Section 3.5.

The script can also support subsequent analysis steps and data-gathering: e.g. after we identify such a retry loop L, the script can check 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 can be considered a potential retry trigger. The call site of M inside loop L can be identified as a retried method. For example, in listing 3.2, Line 5 is identified as a retried method, as the callee method `connect` there could throw exception `ConnectException`, whose catch block on Line 14 can reach back to the loop header. Examples of subsequent analyses are given in Section 3.4.

Limitations. A limitation of CodeQL-based detection is that it is restricted to only retry-on-exception: retry that occurs on error-code (30% of issues in our dataset) is not supported. Research into how error-code checking is performed (and how to differentiate error codes from other result values) is left as future work.

Another limitation of CodeQL-based detection is that it targets only loop-based retry, because it is difficult to define general syntactic, control-flow or naming patterns which can adequately capture variances in non-loop retry forms that often occur in applications.

```
predicate hasExceptionHandling(LoopStmt loop) {
    exists(CatchClause cc | cc.getEnclosingStmt*() = loop and not cc.
        getACaughtType().hasQualifiedName("java.lang", "InterruptedException")
        )
}

predicate hasReentryFromCatch(LoopStmt loop) {
    exists(CatchClause cc | isCaughtTopLevel(loop, cc) and
            exists(Expr loopReentry, ControlFlowNode last |
                if exists(loop.(ForStmt).getAnUpdate())
                then loopReentry = loop.(ForStmt).getUpdate(0)
                else loopReentry = loop.getCondition()
            |
                last.getEnclosingStmt().getEnclosingStmt*() = cc.getBlock()
                    and
                hasSuccessor(last, loopReentry)
            ))
}

// Check if a path exists in CFG between node and expr
predicate hasSuccessor(ControlFlowNode node, Expr expr) {
    node.getASuccessor().(Expr).getParent*() = expr
        or
    hasSuccessor(node.getANormalSuccessor(), expr)
}

predicate hasRetryNamedVariable(LoopStmt loop) {
    (exists(Expr e | e.getAnEnclosingStmt() = l and (e instanceof VarAccess or
         e instanceof
        MethodAccess) and (e.toString().toLowerCase().matches("%retry%") or e.
            toString().toLowerCase().matches("%retries%")))
}

predicate hasRetryStringLiteral(LoopStmt loop) {
    exists (StringLiteral s | s.getAnEnclosingStmt() = l and (s.getValue().
        matches("%retry%") or s.getValue().matches("%retries%")))
}
```

Listing 3.5: CodeQL script example: location detection

In particular, applications use custom APIs and execution patterns for failure checking and task re-submission, which may only be identifiable as retry-related via contextual clues or comments. For example `handle` in Listing 3.1 (queue-based retry) can be reasonably inferred to retry: it accepts a Kafka-error-type parameter, and raises a custom Kafka `RetryException`. But these structural details are unique to this implementation - other implementations in Kafka (and in other applications) do not use this pattern.

Even if locations could be identified, extracting useful information - e.g. which errors lead to retry, or if resubmission occurs after a delay - are implementation-dependent and hard to encode using traditional program analysis techniques. The program analysis capabilities of recently-developed LLMs, however, can help overcome some of these limitations.

3.3.2 LLM assisted retry-location detection

We introduce a second technique that detects both loop and queue/state-machine retry using GPT-4, a recently released large language model. We find that some of the limitations of traditional program analysis mentioned above - i.e. identifying behavior which is hard to encode using syntactic or control flow patterns; or which relies on contextual clues such as method names or comments - can be well handled by the code analysis and comprehension capabilities of a large language model.

Prompt development. Because the GPT-4 model can accept any plain-text prompt, and searching the enormous space of possible prompts is expensive, we apply thoughtful constraints when designing our prompts. Our goal is to find the location of retry: i.e. the files in which contain retry, with guiding information on which specific methods contain retry-related logic.

Input data. We limit our prompt to a single unmodified source file. We chose to exclude file-preprocessing techniques (e.g. pruning, or selective code summarization [2]) for

simplicity (and control), and because the model performed sufficiently well on unmodified inputs.

Zero-shot vs n-shot. Prompts on a given input can be designed as n-shot or zero-shot prompts [9, 1, 46]. In the former, the user issues a query seeking a response for a particular input, along with the a set of n example input-response pairs. For zero-shot prompts, no examples are included. Because our input is a source file which is quite large, cost and token-limit considerations preclude including multiple examples in a prompt. Thus we restrict our design to zero-shot.

Query design. We manually experiment with various hints, word choices, formatting and system prompts. Hints are provided to exclude files that do not implement retry but contain other retry-associated behaviors (e.g. parsing retry-named configuration variables).

Other parameters. GPT-4 models accept other parameters to enable greater randomness (e.g. "temperature", and "frequency penalty.") As our goal is consistency and accuracy, and not randomness, we do not modify these parameters from their defaults.

The prompt that performed the best in our experiments is shown in Q1 of Figure 3.2.

Once GPT-4 reports a method C as one that has implemented retry (using a follow-up prompt of Q1 not shown in Figure 3.2), we can use additional CodeQL queries to perform additional analyses: such as identifying all methods invoked by C as potential retried methods and all exceptions thrown by them as potential retry triggers.

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.

3.3.3 Comparison: Retry Code Identification and Coverage

As shown in Figure 3.1, these two techniques identify 323 code structures across all 8 applications where retry logic is implemented. About 70% of them are loops (i.e., 239 retry loops

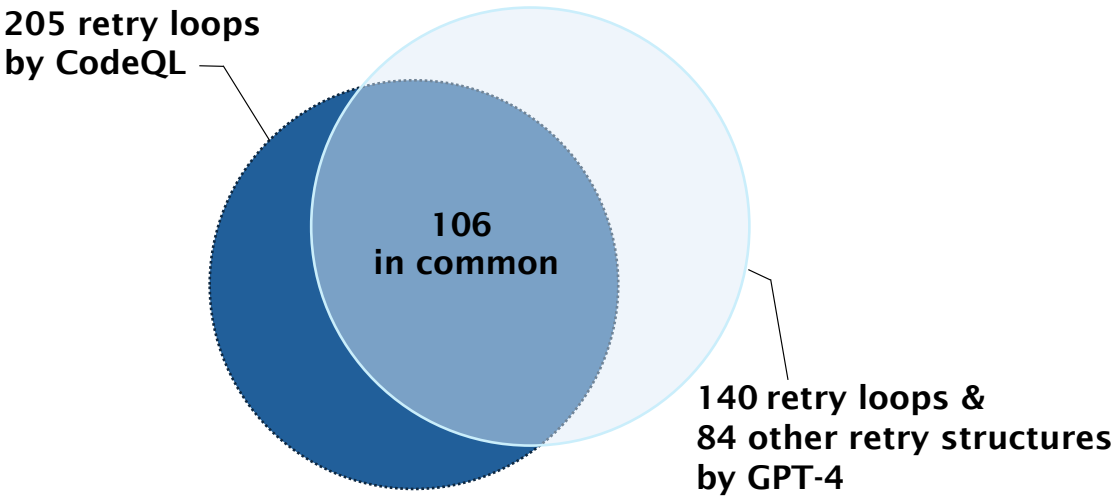

Figure 3.1: Retry code structures identified.

in total), while the rest implement retry through finite state machines and task re-enqueing.

Comparing between 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. In the remaining cases, the loops did not include a retry-named parameter. An example of this is Listing 3.6. Naturally, it missed retry loops that contain no string literals, variables, or methods whose name includes “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 reveal 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. These false positives in GPT-4’s retry identification

```
public class NamedPipeHelper {
  public InputStream openNamedPipeInputStream(..) throws IOException {
    ..
    // Try to open the file periodically until the timeout expires, then, if
    // it's still not available throw the exception from FileInputStream
    while (true) {
      try {
        PrivilegedInputPipeOpener privilegedInputPipeOpener = new
            PrivilegedInputPipeOpener(file);
        return AccessController.doPrivileged(privilegedInputPipeOpener);
      } catch (RuntimeException e) {
        if (timeoutMillisRemaining <= 0) {
          propagatePrivilegedException(e);
        }
        long thisSleep = Math.min(timeoutMillisRemaining, PAUSE_TIME_MS);
        timeoutMillisRemaining -= thisSleep;
        Thread.sleep(thisSleep);
      }
    }
  }
}
```

Listing 3.6: Example loop found by GPT but not CodeQL (Elasticsearch)

are connected with the false positives in its bug identification, discussed in Section 3.4.2.

3.4 Detecting retry bugs via static code analysis

As a next step, we investigate ways to combine the results of retry-location detection with additional static techniques in order to detect IF and WHEN retry bugs.

3.4.1 IF bug detection using CodeQL

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 section 3.3 and the exceptions that could be thrown in each loop are identified by analyzing signatures of

```
import java
import semmle.code.java.ControlFlowGraph

predicate isCaughtAndRetried(LoopStmt loop, RefType ex) {
        exists(CatchClause cc | isCaughtTopLevel(loop,cc)
                        and
                        not exists(CatchClause sup | sup != cc and isCaughtTopLevel(loop,
                            sup) and (sup.getACaughtType() = ex or sup.getACaughtType().
                            getADescendant()=ex)
                                and cc.getTry() = sup.getTry() and sup.getIndex() < cc.
                                    getIndex())
                        and
                        (cc.getACaughtType() = ex or (cc.getACaughtType() != ex and cc.
                            getACaughtType().getADescendant() = ex))
                        and
                        exists(MethodAccess ma | ma.getEnclosingStmt().getEnclosingStmt*()
                            = cc.getTry() and ma.getMethod().getAThrownExceptionType() = ex)
                        and
                        exists(Expr loopReentry, ControlFlowNode last |
                                if exists(loop.(ForStmt).getAnUpdate())
                                then loopReentry = loop.(ForStmt).getUpdate(0)
                                else loopReentry = loop.getCondition()
                        |
                                last.getEnclosingStmt().getEnclosingStmt*() = cc.getBlock()
                                     and
                                hasSuccessor(last, loopReentry)
                        ))

}


predicate isCaughtAndNotRetried(LoopStmt loop, RefType ex) {
  ...
}

predicate isNotCaught(LoopStmt loop, RefType ex) {
        exists(MethodAccess ma | ma.getAnEnclosingStmt() = loop and ma.getMethod().
            getAThrownExceptionType() = ex and
                        not exists(CatchClause cc | ma.getAnEnclosingStmt() = cc) and
                        not exists(CatchClause cc | ma.getAnEnclosingStmt() = cc.getTry()
                            and cc.getEnclosingStmt*() = loop and cc.getACaughtType().
                            getADescendant() = ex))

}
```

Listing 3.7: Retry IF bug detection: CodeQL script

Table 3.3: IF policy-outlier detection results

App	Exception	Majority behavior	# Outliers/Tot.
hbase	org.apache.zookeeper.KeeperException	retried	3/20
hadoop	java.lang.IllegalArgumentException	not retried	1/6
	java.io.FileNotFoundException	not retried	1/4
	org.apache.hadoop.util.ExitUtil.ExitException	not retried	1/3
hive	java.lang.IllegalArgumentException	not retried	1/3
	org.apache.thrift.transport.TTransportException	retried	1/3
cassandra	java.lang.IllegalStateException	not retried	1/3

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 section 3.3. The partial CodeQL script is shown in Listing 3.7.

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), this script would report the outliers as a reminder for developers to check the retry policy decision.

Results. We evaluate the technique on largely on the same set of applications used in our issue study. Note that (1) we used the latest version of each application as of March 2023; and (2) we exclude 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 scope for our IF-detection technique; and exclude Spark, because of incompatibilities with the CodeQL build tool. 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).

```
1 public Set<String> getAllWals() throws ReplicationException {
2   try {
3     for (int retry = 0;; retry++) {
4       int v0 = getQueuesZNodeCversion(); // Can throw KeeperException
5       List<ServerName> rss = getListOfReplicators0(); // Can throw
            KeeperException
6       Set<String> wals = new HashSet<>();
7       for (ServerName rs : rss) {
8         for (String queueId : getAllQueues0(rs)) {
9           wals.addAll(getWALsInQueue0(rs, queueId));
10        }
11      }
12      int v1 = getQueuesZNodeCversion(); // Can throw KeeperException
13      if (v0 == v1) {
14        return wals;
15      }
16      LOG.info("Replication queue node cversion changed from %d to %d, retry =
            %d", v0, v1,
17        retry);
18      }
19    } catch (KeeperException e) {
20      // Not retried
21      throw new ReplicationException("Failed to get all wals", e);
22    }
23 }
```

Listing 3.8: Bug found by IF script: KeeperException not retried

The script 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 $\leq \frac{1}{3}$). The cases are shown in Table 3.3. We have manually checked all cases and believe 8 of them to be truly problematic. These 8 cases come from 5 applications (2 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. An example of an outlier case detected by the script is shown in Listing 3.8

False positives. 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 unanalyzed by

our script.

3.4.2 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 3.4. Our WHEN bug-detection prompts are a series of yes/no interactions about possible missing cap or delay problems (Figure 3.2). 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. 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.

Results. The GPT-4 static checker reports 139 WHEN retry problems as shown in Figure 3.3. After carefully examining each of these reports we identify 79 of them as true bugs.

Our script may also return false negatives, mainly due to GPT-4 struggling at reasoning about large files and hence not even realizing the existence of retry, which we will elaborate in the next section.

False positives. WHEN bug detection using GPT-4 reports 60 cases that do not appear to be actual retry bugs (a false positive rate of 1.4 true bugs vs 1 false positive). 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

```
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 code 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- or 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 3.2: GPT-4 prompts for location and bug detection

.

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}

Figure 3.3: Retry bugs reported by GPT-4 detection script (subscripts: # of false positives)

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.

3.5 Discussion

Cost of GPT-4. Workflows using GPT-4 require making GPT-API calls that send text fragments - i.e. prompts combined with application source code - to the service provider for analysis. To execute these workflows, namely retry location identification and static WHEN bug detection (Figure 3.1, Figure 3.3), 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 writing time, the monetary cost of processing this volume of data using the GPT-4 API was about 8 USD per application. Costs can be further reduced through additional filtering steps, e.g. excluding from analysis files that clearly do not perform I/O.

Importance of keyword filtering If our script 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); and manual examination of these cases indicates that most, if not all, are not related to retry. Loops may be designed this way for many reasons: they 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 themselves may be informative rather than represent transient errors. Thus keyword search is important to eliminate these false positives.

Mitigating false positives. 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 [41, 42]

For the specific cases where GPT mistakenly identifies non- retry code to be retry-related, the effect of false positives can 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 always 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 by our scripts 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. Lastly, our scripts do not detect HOW bugs: our attempts at using the LLM to find these bugs were unsuccessful, as the semantic and subtle nature of these bugs appear to make them unsuitable for LLM detection.

That being said, for a bug whose root cause is covered by our scripts, a false negative could still occur when the retry location is missed by CodeQL and the LLM.

Broader system design considerations. Alongside bug detection tools, other 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.

3.6 Summary

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

CHAPTER 4

ADDITIONAL INVESTIGATIONS

This chapter introduces additional, related investigations that were completed alongside the work in prior chapters. They highlight the challenges and opportunities of applying LLM-aided software analysis to similar software maintenance tasks: namely the automatic extraction of both error-throwing predicates, and locking rules, from software documentation.

4.1 Making software documentation more useful using LLMs

Modern software systems have vast amounts of natural language components, such as code comments (e.g., Figure 4.1) and software manuals, which contain valuable information about system usage and behavior. Extracting such information can be used for system understanding, bug finding, failure diagnosis, configuration tuning, and many more tasks.

Unfortunately, it is difficult to automatically extract such information. Many techniques have been explored in the past, including replacing natural languages with domain-specific languages in writing comments [91], building task-specific machine learning models [7, 97, 76] and customized Natural Language Processing pipelines to process comments [87, 88], complementing documentation understanding with source code analysis [53, 36, 48], etc. These techniques are effective in specific tasks, but all fall short as a general solution that can be easily used to process a variety of natural language artifacts for a variety of purposes.

In this chapter, we explore whether the recent advancement of large language models

Input	Output	Previous techniques
Javadoc comments	Exception conditions in Java	Language&task-specific ML models [7, 97, 76]
Free-text comments	Lock usage rules in pre-defined templates	A pipeline of NLP tools [88, 87]

Table 4.1: Tasks explored in this work

```java
/* @param n the {@code long} to divide by
 * @return a {@link BigFraction} instance with
   the resulting values
 * @throws MathArithmeticException if the
   fraction to divide by is zero */
public BigFraction divide(final long n) {
    return divide(BigInteger.valueOf(n));
}
```

Figure 4.1: Comments on parameters and exception-throwing conditions (Apache Commons Math 3.6.1)

(LLM), such as the GPT series [77, 10, 71], can be leveraged to produce an easy-to-use and one-model-fit-all solution for processing existing natural language components of software systems. These language models have achieved great success on many natural language processing tasks, including translation, text completion, keyword extraction, and question answering, and have shown potential in providing coding assistance [26].

Specifically, we identify two representative tasks and investigate how (well) we can use a large language model, GPT-3 [10], to replace customized solutions originally designed for each task—an approach we refer to as HotGPT. These tasks process different natural language components of software systems, produce different types of output, support different types of system jobs (e.g., bug finding), and were previously solved by different solutions, as summarized in Table 4.1.

Our exploration shows the great potential of using LLMs to help diverse system tasks, achieving similar or higher accuracy than previous task-specific techniques. However, several pitfalls and challenges remain, which we describe in this chapter. We posit that building reliable tools that harness the capabilities of these models to analyze natural language software components is an exciting but open research problem.

4.2 From comments to predicates

4.2.1 Task Overview & Design.

Javadoc [70] is a widely used tool that generates HTML documentation from comments written in a format called *doc comment* or *doc string*. A typical line of Javadoc comment consists of a keyword headed by "@", called a block tag, and a natural language description in the topic defined by the block tag. An example is shown in Figure 4.1.

In the past, researchers have designed special ML models to transform the `@throws` part of Javadoc into exception-throw conditions in Java, which can then be used for automated run-time checking [7, 97]. These techniques involve task-specific NLP analysis and customized pattern-matching rules revolving around the grammar of the comment. Here, HotGPT aims to accomplish the same task using a language-agnostic large language model, Codex [14], a variant of GPT-3.

Prompt Design. A prompt is the text input to the language model. After many attempts, we settled down on a design that consists of three parts as shown in Figure 4.2:

1) An instruction text enclosed in `/* */`;

2) An example for Codex to learn from (Line 3–5 in Figure 4.2). For each software project to be processed, we randomly choose a function with `@throws` comment from it, and manually compose such an example for processing all other `@throws` comments.

3) The synthesis task for Codex, which includes the "Comment", the function "Signature", and an empty "output" line waiting to be filled. The Comment line and the Signature line are automatically extracted from program source code.

Although our prompt follows the generic structure recommended by GPT-3, an instruction, some optional examples, and the question, it took us many tries in the design.

The result of Codex was very sensitive to the instruction sentence. Some semantically similar instructions like "Convert this sentence to code" and "Extract specification from

```
1  /* Summarize the comment in Java code using
       signature provided. */
2
3  Comment: if the queue or transformer is null
4  Signature: transformQueue(java.util.Queue queue,
       commons.collections4.Transformer transformer
     )
5  output:queue==null || transformer==null
6
7  Comment: if the fraction to divide by is zero
8  Signature: divide (final long n)
9  output:
10 n==0
11
12 Comment:  if the fraction to divide by is zero
13 ...
```

Figure 4.2: An Example Prompt and Codex output (in green) For Method `divide()`

code" produced low-quality output, with at least 15% precision reduction. Some other instructions produced meaningful output, but required much effort in post-processing, which we will explain later.

We also tried not using the function signature but got poor results — knowing the type and parameter names helped Codex in synthesizing exception predicates; we tried having no example or multiple examples (e.g., up to 5), which unfortunately was both detrimental.

Codex output post-processing. Ideally, we want Codex to output exactly a predicate in Java that reflects the condition under which an exception is to be thrown. However, in practice, the output of Codex could be messy. When we used "Summarize the comment in Java code by signature" as the instruction, Codex tends to generate multiple lines of code, with the exception predicate embedded in an exception-throwing code structure made up by Codex (Figure 4.3). We had to write a parser to extract the exception predicate. With our final prompt, Codex tends to output the expected condition predicate first (e.g., `n==0` on Line 10 of Figure 4.2), and then part of the prompt after an empty line (e.g., Line 12 in Figure 4.2). Thus, we simply truncate the raw output and take the predicate line before the empty line.

```
...
output:
if (n == 0) {
 throw new IllegalArgumentException("Division by zero");
}
...
```

Figure 4.3: Codex output (in green) under an alternative prompt for Figure 4.2 example. Extra code parsing is needed to extract the exception predicate `n==0`.

	Jdoctor	C2S	HotGPT
Precision	0.97	0.98	0.96
Recall	0.79	0.91	1.00

Table 4.2: Specification Translation Precision and Recall

4.2.2 Evaluation

Methodology. We evaluate HotGPT on 6 well-maintained Java libraries, which were also used in the evaluation of Jdoctor [7] and C2S [97]—prior techniques that turn `@throws` Javadoc into predicates. These libraries were used by previous work partly because developers already provided corresponding code expressions for 60% of their 778 `@throws` comments in the Javadoc, which offers perfect ground truth for evaluation. So, like previous work, we focus on these `@throws` comments that have ground truth. We will measure *precision*, defined as the proportion of total translation that is correct: $\frac{C}{C+W}$, with C being the number of correctly translated predicates and W being the number of incorrectly translated predicates. Since the precision metric only penalizes wrong-output but not no-output, previous work [7, 97] also measured *recall*, computed as $\frac{C}{C+M}$, with M being the number of cases where the tool fails to output any predicate for a comment. We will use the same definition below.

Results. As shown in Table 4.2, HotGPT is effective at translating throw comments into code specifications. Comparing with Jdoctor and C2S, whose results are from their papers on exactly the same dataset, we achieve similar precision and better recall. It is worth

noticing that both Jdoctor and C2S rely on detailed analysis of the format of comments, both syntactically and semantically, to achieve high precision. Instead, we leverage the capability of Codex to conduct the translation, without conducting the task-specific analyses.

We also checked whether the results are sensitive to language models' hyper-parameters. The answer was not so much—much less than that under different prompt designs.

4.3 From comments to locking rules

4.3.1 Task Overview & Design.

Published in HotOS 2005, HotComments [88] pioneered extracting system rules from code comments using NLP techniques. Due to the limitation of NLP techniques at that time, HotComments took many steps: it manually identifies popular words that refer to locks (e.g., spinlock, rwlock) and replaces them with the word "lock"; it then breaks all comments to sentences, and uses a word splitter [90] to break a sentence into words; it then uses Part-of-Speech (POS) tagging and Semantic Role Labeling [90] to tell whether a word in a sentence is a verb or a noun, to distinguish main clauses from sub clauses, and to tell subjects from objects; it then determines whether a sentence contains a locking rule based on how the word "lock" is used in the sentence (e.g., used as a verb or a noun, appearing in the main clause or not, used as a subject or not, etc.). Finally, locking rules that fall into 4 carefully designed templates are extracted. In this task, we attempt to use GPT-3 to replace this long chain of NLP tools.

Prompt Design. We explored different designs in two directions: (1) a generic prompt that covers all four types of locking rules targeted by HotComments; (2) a set of dedicated prompts that each targets one type of rule.

Our final design of the generic prompt is shown in Figure 4.4. It informally introduces the concept of locks (we omitted this paragraph in the figure for space constraints); provides a

```
<... a background paragraph about locks ...>
Here are some templates about patterns of locks/semaphores. 1: Lock must,
   or must not, be held before entering function. 2: Lock must, or must
   not, be held before leaving function. 3: Lock A must, or must not, be
   held before Lock B. 4: Lock must, or must not, be held here.

Read this sentence: is_cpuset_subset(p, q) - Is cpuset p a subset of
   cpuset q? One cpuset is a subset of another if all its allowed CPUs and
    Memory Nodes are a subset of the other, and its exclusive flags are
   only set if the other's are set. Call holding manage_mutex.

Does the sentence describe constraint(s) about locks or semaphores using
   the above templates? If so, output: (1) the name of the lock/semaphore,
    (2) to hold or not to hold the lock/semaphore,(3) the condition(s) for
    holding or not holding, and(4) the template number that the condition
   belong to.

Name of lock/semaphore: manage_mutex
To hold or not to hold: Hold
Condition(s): Before entering is_cpuset_subset
Template number: 1
```

Figure 4.4: Prompt for identifying locking rules (GPT-3 output in green). (For space constraints, we omitted the first paragraph of our prompt that introduces lock background)

list of locking rules we are targeting, which come from HotComments [88]; puts the comment to be analyzed after "Read this sentence:"; and asks a series of questions.

This design came after several tries. Our initial design did not contain a background paragraph about locks. As a result, GPT-3 treated many irrelevant comments as related to locks, like "Hold reference count during initialization."—GPT-3 outputs that a lock named "reference count" should be held. Adding the background paragraph largely solved this problem. We initially did not include the four locking-rule templates from HotComments. As a result, GPT-3 identified many comments that are related to locks and yet difficult to use in correctness checking, like "For dynamic locks, a static lock_class_key variable is passed in through the mutex_init()".

In addition to the generic prompt, we also designed a collection of dedicated prompts. Each dedicated prompt is very simple, only including one question specifically designed for one HotComments locking rule, like "Does the following text explicitly specify that a lock or semaphore must, or must not, be held before function call or on entry (yes/no)?". The other three prompts ask "Does the following text explicitly specify that a lock or semaphore must, or must not, be held before exiting a function (yes/no)?", "Does the following text specify that a lock or semaphore must, or must not, be held before another lock (yes/no)?", and "Excluding on entry/call and exit, does the following text specify that a lock must, or must not, otherwise be held here (yes/no)?", respectively.

Although this design of dedicated prompts requires us to invoke GPT-3 multiple times upon each comment, we envision that it may provide some accuracy advantages over the generic prompt, as we have observed that GPT-3 is very consistent in answering yes/no questions and the answer, the literal "yes" or "no", is also easy to parse. Furthermore, the decomposition made our prompt design easier.

	arch	drivers	fs	kernel	mm
Positive	0.60	0.78	0.70	0.54	0.90
Negative	1.00	1.00	1.00	1.00	1.00

Table 4.3: Accuracy in identifying comments that contain locking rules. Positive (Negative) are the comments considered to (not) contain locking rules by the generic prompt.

4.3.2 Evaluation

Methodology. We evaluated HotGPT on the same five Linux modules used in HotComments: `arch`, `drivers`, `fs`, `kernel`, `mm`. Not knowing the exact Linux version used by HotComments, we chose the one released right before the deadline for HotOS 2007 (v2.6.19). We extracted all the multi-line comments from those five modules using Python library comment_parser [5], and applied GPT-3 to every comment.

To measure the accuracy, we randomly sampled 50 positive comments and 50 negative comments from each module and manually checked the answers of GPT-3. Here, we refer to positive (or negative) comments as those considered by the generic GPT-3 prompt as containing locking rules (or not).

Results. In total, the generic prompt identified 1461 locking rules (93 in `arch`, 778 in `drivers`, 469 in `fs`, 73 in `kernel`, and 48 in `mm`). In comparison, HotComments identified 538 locking rules (50 in `arch`, 263 in `drivers`, 180 in `fs`, 29 in `kernel`, and 16 in `mm`). Since HotGPT and HotComments may not have used the same version of Linux and HotComments paper did not mention the accuracy of their rule identification (the 1461 rules identified by GPT-3 contain false positives), it is unrealistic to expect them to identify the same number of rules. We found it encouraging that (1) the number of rules is of roughly the same magnitude; (2) the proportional distribution of rules across modules is similar.

As shown in Table 4.3, the generic GPT-3 prompt has 100% accuracy in judging a comment to not contain one of those locking rules (the "Negative" row); it has 54–90% accuracy in judging a comment to contain one of those four locking rules. This accuracy

imbalance is likely due to the majority of comments not containing a locking rule.

Once GPT-3 correctly identifies a rule-containing comment, its accuracy in identifying the lock name and differentiating locking from unlocking is very high, close to 100%, while its accuracy in pin-pointing the rule template ranges from 54% to 77% across the five kernel modules.

When we apply the dedicated GPT-3 prompts on those sampled positive comments, the average accuracy across the four dedicated prompts in each module ranges from 70% to 86%, an improvement from the generic prompt.

In summary, it is promising to replace the long chain of NLP tools used 15 years ago with a large language model (LLM). However, an LLM does not solve all the problems: those four rule templates designed by HotComments cannot be replaced yet. Furthermore, the accuracy of HotGPT is sufficient for bootstrapping rule extraction but is not sufficient yet to completely take humans out of the loop.

4.4 Performance of latest models

These investigations were performed using GPT-3; to understand whether and how our system tasks can benefit from the latest version of the LLM (GPT-4, at the time of writing), we re-run the experiments in Section 4.2 and 4.3 with GPT-4 and present the result highlight below.

From comments to predicates. Counterintuitively, HotGPT's precision slightly regressed from 96% down to 93% in translating Javadoc comments into predicates with GPT-4. On one hand, with GPT-4, HotGPT is able to correctly translate some comments with complicated conditions that it failed to translate before, such as "if a value of `array` is outside of `arange`". We believe this can be attributed to the increased model capabilities of [68]. On the other hand, with GPT-4, HotGPT sometimes translates comments into predicates without considering the method signature provided, producing incorrect predicates and causing

```
Comment: if the map is null
Signature: switchClosure(java.util.Map predicatesAndClosures)
```

Figure 4.5: A comment for which GPT-4 performs worse

a regression in overall accuracy. For example, for the task shown in Figure 4.5, HotGPT's output was "`predicatesAndClosures==null`", which is correct. However, with GPT-4, the new output becomes "`map==null`", which is incorrect.

From comments to locking rules. We see an increase in the accuracy of extracting lock-related rules from comments. This increase ranges from 6% to 15% across the four Linux modules. GPT-4 appears to consider comments more comprehensively and precisely than GPT-3.

CHAPTER 5

RELATED WORK

Failures in cloud systems. A large body of work is dedicated to studying failures in cloud systems [30, 57, 8, 37, 89, 24]. Some papers introduce new taxonomies for bugs in cloud and distributed systems [30, 57, 24], while others focus on new or understudied classes of bugs like metastable failures [8, 37] or cross-system defects [89]. This work sheds light on bugs specific to cancel and retry, and provides insights on why this functionality is difficult to implement, analyze, and reason about.

In addition, some works [31, 58] have identified error/fault handling to be a common cause, contributing to 18% of software-related failures in one study [31] and 31% of software-bug incidents in another study [58]. Both categorized error/fault handling problems into two or three major categories, including "error/fault detection", "error propagation", and "error handling". This taxonomy is similar to how we categorize some bugs (cancel-related) at the highest level. The similarity ends here. Since both previous studies focus on general cloud failures, neither goes deep into the error/fault handling problems. The examples of detection, propagation, and handling problems there are very different from the cancel initiation, propagation, and fulfillment bugs discussed in this dissertation.

Error handling and recovery code analysis. Because error handling is documented as one of the main root causes of production failures [29, 30, 96, 57], a sizable amount of work is dedicated to mitigating defects in these components. Some focus on static analysis to check error specifications [40], and error propagation [52, 79, 33]. Others focus on dynamic analysis and fault injection [6, 16, 62, 99, 3, 4, 11, 13, 29, 35]

Work focusing on the problem of empty exception handlers was discussed by Yuan et al. in the study of real-world failures of distributed systems and by Fu and Ryder in the context of analyzing exception-chain of Java programs [95, 23]. The cancel aspect of this work is orthogonal to their research, as we particularly focus on bugs related to task cancel.

As discussed in Section 2.5, only a small portion of cancel-related bugs are due to empty exception handlers — those 6 "Cancel not carried out" bugs in Java and some of those 16 "Dropped cancel" bugs in Java. Because of the task-cancel context, why these bugs' catch blocks are empty, how to fix them, their failure symptoms, and how to generalize them into anti-patterns are all different from generic empty handler problems (e.g., the anti-pattern in Section 2.7.1 does not just look for empty catch blocks).

While the cancel aspect of this work discusses how cancel signals may fail to propagate to the target tasks (Section 2.5.2) in concurrent systems, previous work studied how incomplete error propagation could occur in file systems and storage device drivers [34, 80]. Since previous work looks at propagation through function error-code return, it is orthogonal to our study.

I/O-targeted bug detection Recent research [15, 52] tackles the challenges of uncovering difficult-to-detect bugs from complex interactions between applications and their environments. These tools use fault injection, sometimes combined with a static analysis phase, to detect bugs in these interactions.

Legolas [52] focuses on partial failures in distributed systems, also called gray failures. It injects faults at the application level by throwing exceptions; and utilizes a static analysis phase focusing on `IOException` and its sub-types that are not caught or caught-but-rethrown, in order to find injection locations. That approach contrasts to this work, which focuses on retry-specific bug types; and identifies retry-specific locations that are more wide-ranging and more precisely targeted than those identified in Legolas.

Rainmaker [15] targets bugs in cloud-backed applications interacting with cloud services via REST APIs. Rainmaker intercepts API calls at the HTTP layer, and selectively returns transient errors like service unavailability and timeouts in order to expose bugs. While this technique can potentially trigger application retry, the oracles it uses cannot find the majority of bugs described in this work. For example IF and WHEN bugs are not covered;

```java
// Define retry policy
RetryConfig config = RetryConfig.custom()
    .maxAttempts(2)
    .waitDuration(Duration.ofMillis(100))
    .retryOnResult(response -> response.getStatus() == 500)
    .retryExceptions(TimeoutException.class)
    .build();

// Create function with auto-retry ("Client.httpRequest(..)")
Function<String> requestWithRetry = Retry.decorate(config,
    Client::httpRequest);

// Invoke function with auto-retry
requestWithRetry.apply("http://myurl.com");

// Save policy for reuse across application
RetryRegistry registry = RetryRegistry.ofDefaults();
registry.retry("StoredPolicy", config);
```

Listing 5.1: Retry-framework example (Resilience4j)

as well as bugs related to retrying non-HTTP requests.

Framework-based retry solutions So-called "resilience frameworks", such as Resilience4j (Java) and Polly (C#), provide templated methods and APIs that support applying retry configurations to existing code [66, 78]. Users can define and store a retry configuration that includes cap, delay, and error policy; and decorate user functions with the stored policy (see Listing 5.1 for an example).

However these frameworks support only simple implementations (i.e. loop-based retry); handle only simple policy conditions; and require the retry-able task to be isolated inside a single function, which is often not practicable. None of the applications we studied used such frameworks.

Even in cases where frameworks could in theory be used, they do not prevent most types of issues such as IF or HOW bugs. Therefore automated bug detection is still needed given these frameworks.

LLM-enabled software analysis. The emergence of large language models has created opportunities for software engineering research, with a very active body of work applying

LLMs to program analysis [75, 73, 12, 56, 84, 100, 67, 55, 74, 21, 92]; as well as related areas such as code generation [59, 98, 72], testing [51, 43, 19], repair [93, 22, 39], summarization [1, 2], and documentation [86].

Some works evaluate the performance of LLMs on detecting low-level bugs such as null dereferences and resource leaks [67]; or use-before-initialization bugs [56]. Other works evaluate LLM-assisted steps as part of static analysis workflows. Chapman et. al. [12] infers return value specifications of code segments using LLMs, then applies traditional static analysis techniques to detect inconsistencies. Other analyses look at comments [98] or logs [84] in combination with traditional techniques.

In contrast, this work utilizes capabilities of LLMs to detect a novel bug category: retry bugs. Retry bugs fundamentally differ from many of these low-level bugs in that they are related to a high-level, abstract, and common system mechanism designed to achieve certain system load and performance properties — retry. This inherent connection with the system-retry mechanism gives retry bugs unique bug locations, root causes, and symptoms from other bugs.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This thesis makes contributions to improve the reliability of distributed and concurrent software systems by investigating correctness problems in two critically important functionalities: cancel and retry.

The thesis performs the first systematic studies of cancel and retry problems in modern software, and reveals a variety of root-cause problem categories and anti-patterns. Guided by these studies we developed static detection scripts to detect a variety of problems in both cancel and retry. The thesis also investigates the potential of applying recently-developed large language models (LLMs) to detect retry mechanisms and bugs.

- This thesis provides the first systematic studies of cancel and retry in real-world popular distributed and concurrent applications. It creates taxonomies specific to each mechanism: for cancel, problems in each phase of **triggering**, **signaling**, and **fulfillment**; and for retry, whether a retry should be performed (**IF**), the amount of retries or delays between retry (**WHEN**), and errors in (**HOW**) a retry is performed.

- Guided by the root cause problems revealed by the studies, the thesis introduces a set of detection tools to identify problems in retry and cancel implementations using traditional program analysis techniques. It highlights methods to identify typical anti-patterns across Java-thread and cancel-token based cancel implementations, namely: unhandled interrupt exceptions inside loops; interrupt API misuse; uncanceled child tasks; and ignored tokens. In the case of retry, it introduces a statistical-based technique to detect error-policy problems in loop implementations.

- Consequent to the study finding that the structure of retry implementations varies widely across applications, and are often challenging to identify and analyze using traditional static techniques, the thesis also evaluates new, LLM-based methods to iden-

tify retry implementations and WHEN-type bugs. These techniques identify dozens of missing cap and delay bugs.

There are still many open problems that are left as future work.

Cancel-focused frameworks and developer tools. As discussed in Section 2.5.4, different built-in cancel mechanisms and language constructs offer different support and challenges to developers. While we present some examples of custom cancel constructs in Section 2.5.4, more extensive exploration and evaluation of cancel-related designs and models are needed.

Other kinds of developer tools may also assist in cancel implementation. For example, in Section 2.5.3 we describe how `InterruptedException` often contains the least semantic information about the source of cancel; it may be worth exploring whether developer tools, such as IDE plugins that detect and provide this contextual information, can help guide proper implementation.

Also, although this work presents static tools to detect certain classes of cancel bugs, there are still many cancel bugs that are not covered by our static checkers. More static or dynamic detection and diagnosis tools are needed.

Cancellation in other languages. Different languages may have attributes which affect what types of cancel issues manifest. For example, our study focuses on garbage-collected languages; languages with manual memory management (e.g. C++) may see other cancel issues, e.g. stemming from explicit deallocation.

LLM-analysis coverage and accuracy. While we expect the accuracy of our LLM-based static analysis to improve with future LLM models, other open questions on improving accuracy remain.

First, our work suggests that in some cases it may be helpful to present dependent code to a model (e.g. the body of a callee M), to evaluate the existence of delays or caps in callee

methods. How to choose which dependent code to include, and how best to present it to the model remains an open question. Our results also indicate that models performed more accurately on smaller files. More investigation is needed into accuracy improvements for larger files through source code summarization, prompt compression, or other techniques.

Also, certain problem types not covered by our tools could be a target for future applications of LLM code-analysis. For example, we do not investigate IF bugs concerning error codes as they are challenging to identify using traditional program analysis; yet this appears within the capabilities of LLM code-analysis. This thesis makes clear that LLMs enable effective analysis of bugs in higher-level system mechanisms; what other behaviors and mechanisms to apply them to, and how, will hopefully continue to be an exciting area for future research.

REFERENCES

[1] Toufique Ahmed and Premkumar Devanbu. Few-shot training LLMs for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[2] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T. Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, ICSE'24, pages 1004–1004, Los Alamitos, CA, USA, 2024. IEEE Computer Society.

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

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

[5] Jean-Ralph Aviles. comment_parser: Parse comments from various source files. Online document https://pypi.org/project/comment-parser/, 2022.

[6] Radu Banabic and George Candea. Fast black-box testing of system recovery code. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys'12, page 281–294, New York, NY, USA, 2012. Association for Computing Machinery.

[7] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 242–253, 2018.

[8] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 221–227, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language

models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

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

[12] Patrick J. Chapman, Cindy Rubio-González, and Aditya V. Thakur. Interleaving static analysis and LLM prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2024, New York, NY, USA, 2024. Association for Computing Machinery.

[13] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. Cofi: consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE'20, page 536–547, New York, NY, USA, 2021. Association for Computing Machinery.

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[15] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. Push-button reliability testing for cloud-backed applications with rainmaker. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023*, Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, pages 1701–1716, USA, 2023. USENIX Association.

[16] Maria Christakis, Patrick Emmisberger, Patrice Godefroid, and Peter Müller. A general framework for dynamic stub injection. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE'17, page 586–596. IEEE Press, 2017.

[17] Stephen Cleary. *Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming*. O'Reilly Media, 2019.

[18] Terry Crowley. How to think about cancellation. https://terrycrowley.medium.com/how-to-think-about-cancellation-3516fc342ae, Dec 2016.

[19] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 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*, ISSTA'23, page 423–435, New York, NY, USA, 2023. Association for Computing Machinery.

[20] DevExpress. *CodeRush analyzer*. CodeRush analyzer CRR0038, "The CancellationToken parameter is never used"
https://docs.devexpress.com/CodeRushForRoslyn/119693/static-code-analysis/analyzers-library/crr0038-the-cancellation-token-parameter-is-

[21] Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE'24, New York, NY, USA, 2024. Association for Computing Machinery.

[22] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 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*, ESEC/FSE'23, page 1229–1241, New York, NY, USA, 2023. Association for Computing Machinery.

[23] Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *29th International Conference on Software Engineering (ICSE'07)*, pages 230–239, 2007.

[24] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 126–141, New York, NY, USA, 2022. Association for Computing Machinery.

[25] GitHub. *CodeQL*. A query based static analysis tool
https://codeql.github.com.

[26] Github. Copilot: Your AI pair programmer. Online document https://github.com/features/copilot, 2023.

[27] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP*, pages 66–83. ACM, 2021.

[28] Google. *Go Programming Language*. A statically typed, compiled high-level programming language designed at Google
https://go.dev.

[29] 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. Fate and destini: a framework for cloud recovery testing. In *Proceedings*

of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11, page 238–252, USA, 2011. USENIX Association.

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

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

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

[33] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dussea, and Ben Liblit. Eio: error handling is occasionally correct. FAST'08, USA, 2008. USENIX Association.

[34] Haryadi S. Gunawi, Cindy Rubio-González, and Ben Liblit. EIO: Error handling is occasionally correct. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[35] Victor Heorhiadi, Shriram Rajagopalan, Hani Jamjoom, Michael K. Reiter, and Vyas Sekar. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, pages 57–66, 2016.

[36] Yigong Hu, Gongqi Huang, and Peng Huang. Automated reasoning and detection of specious configuration in large systems with symbolic execution. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 719–734, 2020.

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

[38] Netflix Hystrix. https://github.com/Netflix/Hystrix, Accessed: 2024-04-15. Accessed: 2024-04-15.

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

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

[41] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing prompts for accelerated inference of large language models. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13358–13376, Singapore, December 2023. Association for Computational Linguistics.

[42] Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1658–1677, Bangkok, Thailand, August 2024. Association for Computational Linguistics.

[43] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 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, pages 2312–2323, 2023.

[44] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy diagnosis of in-production concurrency bugs. In *SOSP*, 2017.

[45] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: crowdsourced data race detection. In Michael Kaminsky and Mike Dahlin, editors, *SOSP*, pages 406–422. ACM, 2013.

[46] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213. Curran Associates, Inc., 2022.

[47] Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer. How to cancel a task. In João M. Lourenço and Eitan Farchi, editors, *Multicore Software Engineering, Performance, and Tools*, pages 61–72, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[48] Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, and Dawson Engler. From uncertainty to belief: Inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 161–176, 2006.

[49] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 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*, OSDI'14, page 399–414, USA, 2014. USENIX Association.

[50] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In Tom Conte and Yuanyuan Zhou, editors, *ASPLOS*, pages 517–530. ACM, 2016.

[51] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 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, pages 919–931, 2023.

[52] Ao Li, Shan Lu, Suman Nath, Rohan Padhye, and Vyas Sekar. 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, USA, 2024. USENIX Association.

[53] Chi Li, Shu Wang, Henry Hoffmann, and Shan Lu. Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[54] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In Tim Brecht and Carey Williamson, editors, *SOSP*, pages 162–180. ACM, 2019.

[55] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 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*, ESEC/FSE 2023, page 2107–2111, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 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*, OOPSLA, New York, NY, USA, 2024. Association for Computing Machinery.

[57] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS'19, page 155–162, New York, NY, USA, 2019. Association for Computing Machinery.

[58] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pages 155–162, New York, NY, USA, 2019. ACM.

[59] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36 of *NeurIPS'23*, pages 21558–21572. Curran Associates, Inc., 2023.

[60] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. Automatically detecting and fixing concurrency bugs in Go software systems. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS*, 2021.

[61] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Towards General-Purpose resource management in shared cloud services. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*, Broomfield, CO, October 2014. USENIX Association.

[62] Paul D. Marinescu and George Candea. Lfi: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 379–388, 2009.

[63] Microsoft. Ca2000: Dispose objects before losing scope. https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca2000, 2021.

[64] Microsoft. Cancellation in managed threads. https://docs.microsoft.com/en-us/dotnet/standard/threading/cancellation-in-managed-threads, 2021.

[65] Microsoft. Code analysis in .net. https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/overview, 2021.

[66] Microsoft. Meet polly: The .net resilience library. https://www.pollydocs.org, Accessed: 2024-04-15. Accessed: 2024-04-15.

[67] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Effectiveness of ChatGPT for static analysis: How far are we? AIware 2024, New York, NY, USA, 2024. Association for Computing Machinery.

[68] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[69] Oracle. *Interface Runnable*. JavaDoc. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html.

[70] Oracle. How to write doc comments for the JavaDoc tool. Online document https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html, 2023.

[71] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:2203.02155*, 2022.

[72] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 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, pages 866–866, Los Alamitos, CA, USA, 2024. IEEE Computer Society.

[73] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning*, Proceedings of Machine Learning Research, 2023.

[74] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org, 2023.

[75] Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Sekhar Jana. Exploiting code symmetries for learning program semantics. In *International Conference on Machine Learning*, 2023.

[76] Hung Phan, Hoan Anh Nguyen, Tien N Nguyen, and Hridesh Rajan. Statistical learning for inference between implementations and documentation. In *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*, pages 27–30. IEEE, 2017.

[77] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[78] Resilience4j. Resilience4j fault tolerance library. https://github.com/resilience4j/resilience4j.

[79] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI'09, page 270–280, New York, NY, USA, 2009. Association for Computing Machinery.

[80] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. Error propagation analysis for file systems. *SIGPLAN Not.*, 44(6):270–280, jun 2009.

[81] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. Hector: Detecting resource-release omission faults in error-handling code for systems software. In *DSN*, 2013.

[82] Utsav Sethi, Haochen Pan, Shan Lu, Madanlal Musuvathi, and Suman Nath. *Cancellation Study Artifact (Github).* https://github.com/whoisutsav/cancellation-study-osdi.

[83] Utsav Sethi, Haochen Pan, Shan Lu, Madanlal Musuvathi, and Suman Nath. 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, pages 127–141, Carlsbad, CA, 2022. USENIX Association.

[84] Shiwen Shan, Yintong Huo, Yuxin Su, Yichen Li, Dan Li, and Zibin Zheng. Face it yourselves: An LLM-based two-stage strategy to localize configuration errors via logs. New York, NY, USA, 2024. Association for Computing Machinery.

[85] Bogdan Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. If at first you don't succeed, try, try, again...?: Insights and LLM-informed tooling for detecting retry bugs in software systems. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2024. Association for Computing Machinery.

[86] Yiming Su, Chengcheng Wan, Utsav Sethi, Shan Lu, Madan Musuvathi, and Suman 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*, HOTOS'23, page 87–93, New York, NY, USA, 2023. Association for Computing Machinery.

[87] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 145–158, 2007.

[88] Lin Tan, Ding Yuan, and Yuanyuan Zhou. HotComments: how to make program comments more useful? In *HotOS*, volume 7, pages 49–54, 2007.

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

[90] NLP tools. Online document http://l2r.cs.uiuc.edu/œcogcomp/tools.php, 2023.

[91] Alvaro Veizaga, Mauricio Alferez, Damiano Torre, Mehrdad Sabetzadeh, and Lionel Briand. On systematically building a controlled natural language for functional requirements. *Empirical Software Engineering*, 26(4), 2021.

[92] Ashwin Prasad Shivarpatna Venkatesh, Samkutty Sabu, Amir M. Mir, Sofia Reis, and Eric Bodden. The emergence of large language models in static analysis: A first look through micro-benchmarks. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, FORGE '24, page 35–39, New York, NY, USA, 2024. Association for Computing Machinery.

[93] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 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, pages 1482–1494, 2023.

[94] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. Managing data constraints in database-backed web applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20. Association for Computing Machinery, 2020.

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

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

[97] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2s: translating natural language comments to formal program specifications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 25–37, 2020.

[98] Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 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*, ESEC/FSE'23, page 695–707, New York, NY, USA, 2023. Association for Computing Machinery.

[99] Pingyu Zhang and Sebastian Elbaum. Amplifying tests to validate exception handling code. In *2012 34th International Conference on Software Engineering (ICSE)*, ICSE'12, pages 595–605, 2012.

[100] Yichi Zhang. Detecting code comment inconsistencies using LLM and program analysis. FSE 2024, New York, NY, USA, 2024. Association for Computing Machinery.

[101] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. Understanding and detecting software upgrade failures in distributed systems. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP*, pages 116–131. ACM, 2021.