Cancellation in Systems: An Empirical Study of Task Cancellation Patterns and Failures

Utsav Sethi and Haochen Pan, *University of Chicago;*
Shan Lu, *University of Chicago and Microsoft;* Madanlal Musuvathi
and Suman Nath, *Microsoft Research*

https://www.usenix.org/conference/osdi22/presentation/sethi

This paper is included in the Proceedings of the
16th USENIX Symposium on Operating Systems
Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Cancellation in Systems

An Empirical Study of Task Cancellation Patterns and Failures

Utsav Sethi
University of Chicago
usethi@uchicago.edu

Haochen Pan
University of Chicago
haochenpan@uchicago.edu

Shan Lu
University of Chicago/Microsoft
shanlu@uchicago.edu

Madanlal Musuvathi
Microsoft Research
madanm@microsoft.com

Suman Nath
Microsoft Research
Suman.Nath@microsoft.com

Abstract

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 or user events that make the task irrelevant. In this paper, we studied 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 generalized 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. We hope this study will help guide better and more systematic approaches to task cancellation.

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 [6, 7, 20, 22]. 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 [11, 16, 18, 26, 27].

This paper 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.

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

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

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

Table 1. Task constructs and cancellation mechanisms

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 [1] 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 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 Background

Task. This paper defines a task as a unit of concurrent execution. As summarized in Table 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 [4, 5, 22]. 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.

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

```java
public void run() {
      try { ...
      } catch (InterruptedException e) {
      // receiver handles the cancel request
      }
      ...
      if (Thread.currentThread().isInterrupted()) {
      // receiver handles the cancel request
      }
  }
```

Listing 1. Handling cancel requests in Java

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 [15]. Note that the alternate way of task cancel - *abortive*, where the initiator forces the target to 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 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 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.

```
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. Handling cancel requests in C#

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 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 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. 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 the paper, 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 in this paper.

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 5.

3 Methodology

Application selection. We study applications written in three different languages: Java, C# and Go, as shown in Table 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. 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).

Table 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

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 5 and 6. We should note that although an issue might belong to multiple root causes or symptom categories, it

Table 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**

is classified by its primary category only, **without double-counting**. In addition, we study 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 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.

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 3, and generalized three main reasons for task cancel as shown in Table 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 navigates 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.

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 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.

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 4 and skip discussion about them below.

5.1 Cancel-initiation bugs

As discussed in Section 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 4).

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.

5.1.1 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.

Table 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

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 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.

<u>*Lessons learned.*</u> 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.

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 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.

5.1.2 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.

5.1.3 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.

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.

```
// 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 3. An example of late cancel (SPARK-1582)

5.2.1 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 [10, 13, 14, 17, 19] should be applied to check the correctness of cancel-related implementation.

Late polling. As discussed in Section 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 3 illustrates a simplified version of bug SPARK-1582. A task checks

```
// 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 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 5. API Misuse Example (SOLR-8066)

whether a cancel is delivered to it at the beginning of every work-loop iteration through 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 5.4.

5.2.2 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 4. The Java `sleep` method, just like many other Java blocking methods, silently unset the interrupt and throw an `InterruptedException`. 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 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 5.

Lessons Learned. Many bugs of this type can be automatically detected. As we will discuss in Section 7.1 and 7.2, static checkers can search for the catch blocks of `Interrupted-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

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

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

Listing 6. One type of invisible token (aspnetcore-5936)

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 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#[23]. We apply this checker to the latest versions of ASP.NET Core and Roslyn, and report the results in Section 7.5.

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.

5.3.1 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.

Table 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

Lessons Learned. For C# applications, we have implemented a static checker to detect this type of bug (Section 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.

5.3.2 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 5.2.2[3], they both can be detected by a checker that searches for problematic catch blocks of `InterruptedException`, which we will discuss in Section 7.1.

5.3.3 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

[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.

efficient way is challenging. Unsurprisingly, bugs that occur during this process are particularly common, contributing to more than 20% of all the bugs in our study.

What went wrong? There are mainly four types of mistakes in a cancel cleanup, as shown in Table 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 metadata 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 [25] and cancellation-token leaks [21] 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.

5.4 Discussion: cancel mechanisms

5.4.1 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 5.2.2) 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 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 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 5.2.2). 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.

5.4.2 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 abstract 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.

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 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 5.3.3). 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.

Table 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

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

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

Table 7. Anti-pattern instances found in Java and C# applications

	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

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 [1], 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 [3].

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 7.4 and 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.

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.

Rationale. This anti-pattern is closely related to "cleared interrupt" bugs (Section 5.2.2) and "cancel not carried out" bugs (Section 5.3.2). 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 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 7). 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.

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 5.2.2 (e.g., Listing 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 7. Our manual examination did not find any false positives.

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 5.1.1. 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.

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 5.3.1. For a similar reason as discussed in Section 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 [2], 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.

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 5.2.2.

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.

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.

8 Related Work

Our study is the first empirical study of task cancellation patterns and failures in concurrent systems to the best of our knowledge. Nevertheless, several related works have discussed general exception handling problems in the past.

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 [8, 26]. Our work is orthogonal to their research, as we particularly focus on bugs related to task cancel. As discussed in Section 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 7.1 does not just look for empty catch blocks).

While our work discusses how cancel signals may fail to propagate to the target tasks (Section 5.2) in concurrent systems, previous work studied how incomplete error propagation could occur in file systems and storage device drivers [12, 24]. Since previous work looks at propagation through function error-code return, it is orthogonal to our study.

Past studies about general cloud system failures [11, 18] have identified error/fault handling to be a common cause, contributing to 18% of software-related failures in one study [11] and 31% of software-bug incidents in another study [18]. 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 cancel-related bugs 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 paper.

A Java textbook [9] has listed five possible reasons behind task cancel: (a) user-requested cancel, (b) time-limited activities, (c) application events, (d) errors, (e) shutdown. In our cancel feature study, we want to see what are the common reasons and trigger events behind task cancel in modern concurrent systems. Our study led to a categorization (Table 3) that is related but not the same as the textbook listing.

9 Future Research Directions

In this section we highlight a few potential areas for future research.

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.

Cancel programming models and language features. As discussed in Section 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 5.4, more extensive exploration and evaluation of cancel-related designs and models are needed.

Bug-detection and other developer tools. 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.

Other kinds of developer tools may also assist in cancel implementation. For example, in Section 5.3.2 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.

10 Conclusions

Task cancellation is critical to the efficiency, availability, and operational flexibility of concurrent systems. This paper 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.

11 Acknowledgements

We thank the reviewers for their insightful comments, and Mahesh Balakrishnan for shepherding this work. The authors' research is supported by NSF (grants CCF-2119184, CCF-2028427, CNS-1956180, CCF-1837120, CNS-1764039), the CERES Center for Unstoppable Computing, the Marian and Stuart Rice Research Award, and gifts from Microsoft and Facebook.

References

[1] [n.d.]. CodeQL. https://codeql.github.com
[2] [n.d.]. CRR0038 - The CancellationToken parameter is never used. https://docs.devexpress.com/CodeRushForRoslyn/119693/static-code-analysis/analyzers-library/crr0038-the-cancellation-token-parameter-is-never-used
[3] [n.d.]. Github - cancellation-study-osdi. https://github.com/whoisutsav/cancellation-study-osdi
[4] [n.d.]. Golang. https://go.dev
[5] [n.d.]. Runnable. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Runnable.html
[6] Stephen Cleary. 2019. *Concurrency in C# Cookbook: Asynchronous, Parallel, and Multithreaded Programming.* O'Reilly Media.
[7] Terry Crowley. 2016. How to Think About Cancellation. https://terrycrowley.medium.com/how-to-think-about-cancellation-3516fc342ae
[8] Chen Fu and Barbara G. Ryder. 2007. Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In *29th International Conference on Software Engineering (ICSE'07).* 230–239. https://doi.org/10.1109/ICSE.2007.35
[9] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. 2006. *Java concurrency in practice.* Pearson Education.

[10] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis. In *SOSP*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 66–83.

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

[12] Haryadi S. Gunawi, Cindy Rubio-González, and Ben Liblit. 2008. EIO: Error Handling is Occasionally Correct. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*. USENIX Association, San Jose, CA. https://www.usenix.org/conference/fast-08/eio-error-handling-occasionally-correct

[13] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. 2017. Lazy Diagnosis of In-Production Concurrency Bugs. In *SOSP*.

[14] Baris Kasikci, Cristian Zamfir, and George Candea. 2013. RaceMob: crowdsourced data race detection. In *SOSP*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 406–422.

[15] Alexey Kolesnichenko, Sebastian Nanz, and Bertrand Meyer. 2013. How to Cancel a Task. In *Multicore Software Engineering, Performance, and Tools*, João M. Lourenço and Eitan Farchi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–72.

[16] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *ASPLOS*, Tom Conte and Yuanyuan Zhou (Eds.). ACM, 517–530.

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

[18] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What Bugs Cause Production Cloud Incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. ACM, New York, NY, USA, 155–162. https://doi.org/10.1145/3317550.3321438

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

[20] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2014. Towards General-Purpose Resource Management in Shared Cloud Services. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*. USENIX Association, Broomfield, CO. https://www.usenix.org/conference/hotdep14/workshop-program/presentation/mace

[21] Microsoft. 2021. CA2000: Dispose objects before losing scope. https://docs.microsoft.com/en-us/dotnet/fundamentals/code-analysis/quality-rules/ca2000

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

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

[24] Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau. 2009. Error Propagation Analysis for File Systems. *SIGPLAN Not.* 44, 6 (jun 2009), 270–280. https://doi.org/10.1145/1543135.1542506

[25] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L. Lawall, and Gilles Muller. 2013. Hector: Detecting Resource-Release Omission Faults in error-handling code for systems software. In *DSN*.

[26] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[27] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *SOSP*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 116–131.