Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems

Lilia Tang*
University of Illinois
Urbana-Champaign, IL, USA
liliat2@illinois.edu

Chaitanya Bhandari*
University of Illinois
Urbana-Champaign, IL, USA
cbb1996@illinois.edu

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

Anna Karanika
University of Illinois
Urbana-Champaign, IL, USA
annak8@illinois.edu

Shuyang Ji
University of Illinois
Urbana-Champaign, IL, USA
sji15@illinois.edu

Indranil Gupta
University of Illinois
Urbana-Champaign, IL, USA
indy@illinois.edu

Tianyin Xu
University of Illinois
Urbana-Champaign, IL, USA
tyxu@illinois.edu

Abstract

Modern cloud systems are orchestrations of independent and interacting (sub-)systems, each specializing in important services (e.g., data processing, storage, resource management, etc.). Hence, cloud system reliability is affected not only by the reliability of each individual system, but also by the interplay between these systems. We observe that many recent production incidents of cloud systems are manifested through *interactions* across the system boundaries. However, there is a lack of systematic understanding of this emerging mode of failures, which we term as *cross-system interaction failures* (or *CSI failures*). This hinders the development of better design, integration practices, and new tooling.

In this paper, we discuss cross-system interaction failures based on analyses of (1) 11 CSI-failure-induced cloud incidents of Google, Azure, and AWS, and (2) 120 CSI failure cases of seven widely co-deployed open-source systems. We focus on understanding discrepancies between interacting systems as the root causes of CSI failures—CSI failures cannot be understood by analyzing one single system in isolation. This paper draws attention to this emerging failure mode, provides a comprehensive understanding of CSI failure patterns, and discusses potential approaches for mitigation. We advocate for cross-system testing and verification and demonstrate its potential by cross-testing the Spark-Hive data plane and exposing 15 new discrepancies.

CCS Concepts: • **Computer systems organization → Reliability**; • **Software and its engineering → Software testing and debugging**.

Keywords: Cross-system interaction, failure study, root cause analysis, cloud system

ACM Reference Format:
Lilia Tang*, Chaitanya Bhandari*, Yongle Zhang, Anna Karanika, Shuyang Ji, Indranil Gupta, and Tianyin Xu. 2023. Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems. In *Eighteenth European Conference on Computer Systems (EuroSys '23), May 8–12, 2023, Rome, Italy.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3552326.3587448

1 Introduction

Modern cloud systems are orchestrations of independent and interacting (sub-)systems, each specializing in important services (data processing, storage, resource management, etc.). For example, to run data analytics jobs, Spark needs to interface with at least a storage system (e.g., HDFS, Cassandra, or Alluxio) and a cluster management system (e.g., YARN, Mesos, or Kubernetes), which further interfaces with downstream systems. In fact, Spark has been interfaced with tens of different data/storage systems [11, 27]. Other cloud systems such as Flink, Storm, and OpenStack require similar levels of orchestration [36, 91].

We expect this practice of system and data orchestration to become more prevalent and fine-grained, driven by recent trends such as Sky Computing [102] and Hybrid Cloud [59, 106]. Furthermore, recent movements such as

*Co-primary authors.

EuroSys '23, May 8–12, 2023, Rome, Italy

ACM ISBN 978-1-4503-9487-1/23/05…$15.00
https://doi.org/10.1145/3552326.3587448

serverless computing and microservices decouple monolithic systems into interacting, fine-grained ones.

Given the heterogeneity and complexity of cloud systems, every (sub-)system tends to be developed and maintained independently, by different projects and teams. This segregation differs from traditional cloud system projects such as Hadoop, which encapsulates multiple subsystems for data processing (MapReduce), storage (HDFS), resource management (YARN), and previously a key-value store (HBase) and ML framework (Submarine) (before their spin-offs), with a common runtime and library (Hadoop Common [20]).

Despite the benefits of decoupling and orchestration, an emerging software failure model manifests through interactions of independent and interacting systems, which we term *cross-system interaction failures* (or, *CSI failures*) in this paper. Different from failures of individual systems, the root cause of a CSI failure is not contained within one system, but resides in the interactions between multiple systems—each system inspected in isolation behaves correctly as per its specification. CSI failures escape existing unit testing and integration testing which cover functionalities and interactions across components *within* a system.

Our study reveals that CSI failures have significant presence among root causes of cloud incidents (§3). For example, in a recent Google Cloud Platform (GCP) incident [25], Google's User-ID system suffered an outage due to cross-system interaction between its monitoring system and a quota system. The root cause was a discrepancy in the monitoring data—a deregistered monitor reported a value "0" for the resource usage to the quota system, which misinterpreted zero as the expected load of the User-ID system. Consequently, the quota system incorrectly decreased the resource quota of the User-ID system, resulting in a major GCP outage. The outage had a widespread impact on other upstream services such as YouTube and Gmail [18, 19, 28].

Addressing CSI failures is critical to cloud system reliability, because reliability arises not only from the correctness of each individual system, but also from the correctness of their interactions. However, there is a lack of systematic understanding of this emerging mode of failures. An early cloud failure study [62] mentioned "cross-system bugs", but left them for future work. In fact, cross-system bugs in [62] are different from CSI failures—most of them are not CSI bugs, but occurred within an orchestrated system (Table 1). A recent study on Azure cloud incidents [79] revealed the challenges of data interactions and reported data-format issues as an emerging failure root cause, which corroborates our analysis of data-related CSI failures (§6.1). However, we find that CSI failures are broader and diverse.

CSI failures are not limited only to cloud systems. The infamous incident of the Mars Climate Orbiter [31] was a classic CSI failure caused by a unit discrepancy between two subsystems: metric units by NASA and US Customary units by spacecraft builder Lockheed Martin. This expensive incident drove much research on unit safety and type systems (e.g., [73, 93, 100]). However, the heterogeneity and composability of today's cloud systems significantly magnifies and diversifies CSI issues beyond a unit or type mismatch.

In this paper, we discuss cross-system interaction failures based on an analysis of (1) eleven CSI-failure-induced cloud incidents of Google, Azure, and AWS, and (2) 120 CSI failure cases of seven widely co-deployed, commonly interacting open-source systems. Our analysis reveals over a dozen informative findings with concrete implications for new practices and research directions in combating CSI failures. Our goal is to draw attention to the emerging failure mode, provide a comprehensive, holistic understanding of various CSI failure patterns, and discuss potential solutions in terms of better practices of API design, system integration, as well as testing and verification.

The paper makes the following contributions:

- **Prevalence (§3 and §4).** Our study shows CSI failures' prevalence in both proprietary and open-source cloud systems. A large fraction (20%) of cloud incidents are caused by CSI failures. Since cloud vendors typically only report the most severe incidents, this finding highlights CSI failures' catastrophic consequences. In open-source systems too we find that CSI failures are common (37%).
- **Failure location (§5.1)**. Contrary to traditional cloud bugs [38, 123], data- and management-plane interactions are major contributors to CSI failures (51% and 32% respectively); control-plane interactions are not as dominant as reported in early studies on traditional cloud bugs.
- **Symptoms (§5.2)**. We find that most (74%) CSI failures are manifested through crashing behavior, which indicates that existing fault tolerance or recovery mechanisms are insufficient or ineffective in handling CSI issues.
- **Root causes (§6)**. Most (82%) data-plane CSI failures are caused by discrepancies of metadata: over two-thirds (69%) are about typical metadata including addressing/naming and data schemas, and 13% are about custom metadata such as non-POSIX file properties. The remaining 18% are rooted in discrepancies of data operation semantics. On a different perspective, we find that many (25%) data-plane CSI failures can be caused by *ad-hoc* data serialization.
  For management-plane CSI failures, most failure-inducing configuration issues are about coherently configuring multiple interacting systems (e.g., 60% occur due to silent configuration overwriting across systems), which is fundamentally different from traditional configuration issues.
  We also find that cross-system actions triggered by monitoring data can cause crashing failures.
- **Fix strategies (§7)**. We find that 42% of CSI failures' fixes are in the form of condition checking and error handling. Unfortunately, we find that common fixes do not fix the error-prone interactions.

- **Case study of cross-system testing (§8)**. Our findings suggest that pre-deployment cross-system testing could be an effective practice to prevent CSI failures. As a feasibility study, we develop a cross-system integration testing tool targeting Spark and Hive interfaces and show that our tool can capture data-plane CSI failures across Spark and Hive. It exposes 15 new discrepancies on the Spark-Hive data plane, among which nine have been acknowledged by the developers.

The research artifact is available at https://github.com/xlab-uiuc/csi-ae. We hope that our work will encourage more research into this emerging software failure mode.

2 Definition

2.1 CSI Failure Mode

A CSI failure mode manifests through *interactions* between independent and interacting systems, instead of within a single system. The interaction may take various forms, including SQL queries, filesystem reads/writes, resource allocations, etc. We refer to the system that initializes the interaction as the *upstream* and the one that responds to the interaction as the *downstream*. The upstream system requests the services provided by the downstream system. Each system is typically maintained independently and could function with other interchangeable upstream or downstream systems. We do not consider resource contention (co-located systems competing for hardware resources) as a form of interaction. A key characteristic of CSI failures is that the root causes are discrepancies between the upstream and downstream systems, i.e., neither the upstream nor the downstream is buggy based on their own specifications. Hence, CSI failures cannot be analyzed in the context of one involved system in isolation.

Difference from dependency failures. CSI failures are fundamentally different from *dependency failures* [105, 107]. In dependency failures, the downstream system fails independently from the upstream. From the upstream's perspective, the failure lies in the lack of fault tolerance or error handling; in other words, the failure does not lie in the interaction.

In CSI failures, the downstream system could be well available, yet the upstream would still not be correctly serviced by the downstream due to discrepancies in the interaction. The interaction may also affect the downstream (e.g., overloading the downstream or triggering unexpected behavior).

Dependency failures can be exposed by simulating unavailability of dependent services using fault injection and chaos engineering approaches [35, 42], while exposing CSI failures requires generating faulty interactions.

Difference from library interaction failures. If we treat the downstream system as a library of the upstream, CSI failures resemble library interaction failures [72, 108]. However, CSI failures are different from library interaction failures in many ways beyond differences in complexity. For example, library interaction failures are typically rooted in incompatible API and data types [44, 110], while CSI failures have more diverse root causes (§6). More fundamentally, a library is a part of the system that integrates it and even runs in the same address space. In software testing, library code is no different from the other parts of system code and is extensively exercised by existing tests. However, testing cross-system interaction is more expensive, because it requires deploying multiple interacting systems. Furthermore, systems typically use specific libraries for a given purpose. However, upstream systems are often designed to support multiple downstream systems so that users can configure the interacting systems based on their needs.

2.2 Failure Planes

We observe that patterns of CSI failures are highly correlated with the logical "planes" of the interaction, and interactions on different planes have drastically different characteristics and implications to potential solutions. Therefore, we organize our discussion around CSI failures of *control*, *data*, and *management* planes. The concepts of control/data/management planes originated from networking literature [57, 60, 111] and have been used to refer to the construction of cloud systems in recent years [38, 47, 52, 71, 75]. For concreteness, we define each plane as follows:

- *Control plane* embodies the system core control logic, such as scheduling, resource allocation, coordination, fault tolerance, recovery, etc.
- *Data plane* embodies the components for data operations, in forms of tables, files, tuples, and streams.
- *Management plane* embodies the components for system configuration and monitoring.

Each plane could be composed of components of multiple interacting systems and thus is subjective to CSI failures.

2.3 Examples

We present three representative CSI failures on the control, data, and management planes, respectively.

Control Plane. Figure 1 shows a CSI failure on the control plane between Flink and YARN (documented in FLINK-12342). Flink uses a YARN API to request containers. It maintains a count C for the number of containers it requires. This count is decreased after YARN returns the number of containers successfully allocated. In this case, Flink periodically requests C containers every 500ms. However, when C is large, it takes YARN more than 500ms to allocate the container. Since the request is not acknowledged by YARN within 500ms, Flink requests the aggregated number of pending containers plus another C containers to YARN, overloading it. This resulted in thousands of container requests. In essence, the discrepancy lies in the container request/response semantics of Flink and YARN. Flink's usage of the YARN API assumes a synchronous interaction, in which the request is served and

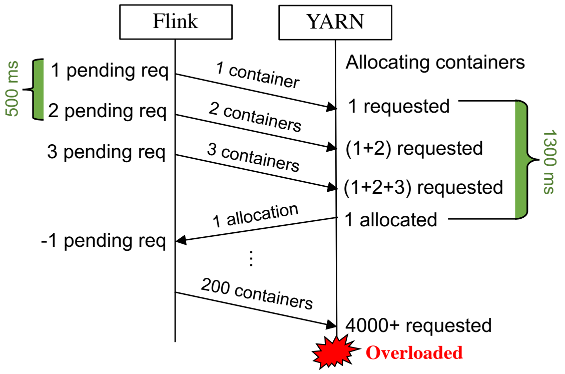

Figure 1. A control-plane CSI failure between Flink and YARN (FLINK-12342), where the discrepancy lies in the interaction model (sync vs. async) of container allocations.

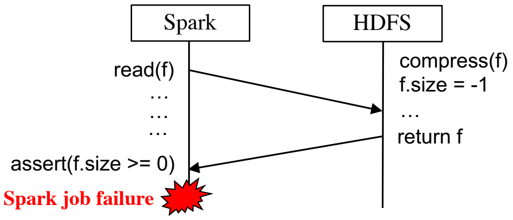

Figure 2. A data-plane CSI failure between Spark and HDFS (SPARK-27239); the discrepancy lies in interpretation of file size.

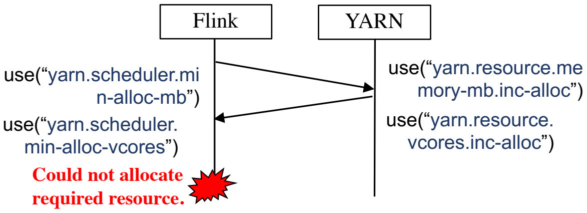
Flink
YARN
use("yarn.scheduler.min-alloc-mb")
use("yarn.scheduler.min-alloc-vcores")
Could not allocate required resource.
use("yarn.resource.memory-mb.inc-alloc")
use("yarn.resource.vcores.inc-alloc")

Figure 3. A management-plane CSI failure between Flink and YARN (FLINK-19141); the discrepancy lies in the misinterpretation of the configuration values used by Flink and YARN.

returned within the time interval. However, this assumption is broken if the time interval is smaller than the time needed by processing at the YARN side.

Data Plane. Figure 2 shows a CSI failure on the data plane between Spark and HDFS (SPARK-27239). The discrepancy is that Spark asserts that the size of a valid file it reads from HDFS should be nonnegative, while HDFS sets the file size to be −1 for compressed data. This discrepancy causes Spark job failures if the job operates on compressed data.

Management Plane. Figure 3 shows a CSI failure on the management plane between Flink and YARN (FLINK-19141). The discrepancy lies in the interpretation of the configuration values used by Flink and YARN. Essentially, the two configuration parameters are used by different YARN schedulers with inconsistent semantics.

3 Cloud Incidents Induced by CSI Failures

We first study CSI failures in the field by collecting and analyzing public incident reports of three public cloud services: Google Cloud Platform (GCP) [17], Microsoft Azure [14], and Amazon Web Services (AWS) [13]. We collected in total 55 cloud incident reports. For GCP and Azure, we sampled 20 recent incidents; for AWS, we collected all 15 incidents with post-event summaries [13]. Due to the often limited scope of provided information, we use our best judgment in identifying CSI failures based on the definitions in §2.

Finding 1: *Among 55 cloud incidents, 11 (20%) were caused by CSI failures, showing their catastrophic consequences.*

The CSI-failure-induced cloud incidents lasted from 10 minutes [24] to 19 hours [14], with a median of 106 minutes. Many of them (8/11) further impaired other external production services that depend on them. For example, the GCP incident described in §1 is such a case.

Postmortem reports typically provide limited information about the failure. Some reports reveal that the failed interactions happen on different planes, including control plane (e.g., scheduling [23]), data plane (e.g., metadata queries [24]), and management plane (e.g., configuration updates [26], monitoring [25]). However, we do not find detailed information on the failed interactions and corresponding root causes, especially at the source-code level. We also do not find detailed information on how the CSI failures are fixed. In fact, only 4/11 reports mentioned code fixes related to the interactions. The other reports either did not mention interaction-related fixes, or mentioned generic solutions such as more rigorous deployment process and more comprehensive testing.

Implication. CSI failures are under-studied, especially given their prevalence and severity. Specifically, studies on open-source systems are needed to understand the detailed failure characteristics (e.g., root causes), because public reports from cloud providers are limited.

4 Open-Source CSI Failure Dataset

While publicly available incident reports provide high-level information to identify CSI failures, they lack details to understand concrete interactions and failure patterns at the source-code level. Therefore, we collected a CSI failure dataset of open-source systems that are commonly deployed together from their issue databases.

Collecting a CSI failure dataset is challenging. Unlike failures of a standalone system that are documented in their own issue database, CSI failures do not have a centralized database. Instead, a CSI failure could be documented by any of the involved systems. Moreover, there is no label related to CSI in existing issue databases; in fact, no label indicates whether a failure involves multiple systems. We addressed these challenges and collected 120 CSI failures across seven mature, widely-used open-source systems.

Upstream	Downstream	Interaction	# CSI failures
Spark	Hive	Data (Hive tables)	26
Spark	YARN	Control (resource management)	19
Spark	HDFS	Data (files)	8
Spark	Kafka	Data (streaming)	5
Flink	Kafka	Data (streaming)	12
Flink	YARN	Control (resource management)	14
Flink	Hive	Data (Hive tables)	8
Flink	HDFS	Data (file systems)	3
Hive	Spark	Control (compute)	6
Hive	HBase	Data (key-value store)	3
Hive	HDFS	Data (files)	6
Hive	Kafka	Data (streaming)	1
Hive	YARN	Control (resource management)	2
HBase	HDFS	Data (file systems)	4
YARN	HDFS	Data (file systems)	3
Total			120

Table 1. Target systems, their interactions, and the number of corresponding CSI failures studied in this paper.

Methodology. We study CSI failures between any two or more of seven cloud systems: Spark [10], Hive [6], YARN [4], HDFS [21], Flink [3], Kafka [7], and HBase [5]. The systems are selected using a data-driven approach. We focus on systems that use JIRA [12] as their issue database so that we can apply our search scripts uniformly. We combine all the issue databases and query for issues that include multiple system names in the issue title or issue summary. We use a heuristic to collect issues: if an issue documents a CSI failure, the title/summary should mention the involved systems. We then select the seven most commonly occurring systems from the selected issues. The relationships between the target systems are listed in Table 1.

CSI failure collection. Not every issue that involves multiple systems in the title/summary is a CSI failure (an issue of a single system could mention co-located systems). Furthermore, not every cross-system failure is a CSI failure (see §2). To collect CSI failures, we randomly sampled 360 issues that include at least two target systems in title/summary and manually inspected them, yielding 120 CSI failures (Table 1). For the remaining ones, 26 are dependency failures, and the rest are not cross-system issues.

We only consider issues that (1) are resolved because it is hard to draw conclusions for open issues, (2) have severity of "Blocker", "Critical", or "Major" – we do not consider "Minor" issues or "Improvements", and (3) occur in the wild, which excludes issues exposed during testing. We borrow a heuristic from [120] to filter out developer-reported issues.

With these criteria, we collected 1428 issues for the seven target systems, from which we randomly sampled 360 issues. Each issue was labeled by two team members independently. If the labels differed, a senior team member was consulted until a consensus was reached. Collection started from a pilot study of 40 issues synchronously discussed and labeled by all team members. Data collection took about 180 person-hours, involving six team members.

Comparison with the CBS dataset. In order to have a comparable reference, we also studied the Cloud Bug Study (CBS) dataset released in 2014 [32, 62]. CBS contains JIRA issues of six Hadoop-based systems (MapReduce, HDFS, HBase, Cassandra, ZooKeeper, and Flume) from 1/1/2011–1/1/2014. Conveniently, the CBS contains a `cross` label to indicate whether an issue is a potential cross-system issue. Applying the same collection criteria described above, we collected 105 issues: 39 are CSI failures, 15 are dependency failures, and the remaining are not cross-system issues.

5 General Characteristics

5.1 Failure Plane

We now analyze CSI failures by failure planes (defined in §2.2), because different planes have drastically different CSI failure characteristics and implications to potential solutions. For example, data-plane CSI failures are mostly about data/metadata interoperability, while management-plane CSI failures are mainly about inconsistent configurations and monitoring. Solutions to these CSI failures could be tailored to the nature of failures on each plane, except for a few common patterns (e.g., misusing downstream APIs is one that occurs across all planes).

Finding 2: *Data- and management-plane interactions contribute to significant percentages of CSI failures: 51% of CSI failures in our dataset manifest at the data plane, and 32% of CSI failures manifest at the management plane. Control-plane interactions contribute to 17%. See Table 2.*

Plane	# (%) Fail.
Control	20 (17%)
Data	61 (51%)
Management	39 (32%)
Total	120 (100%)

Table 2. Cat. by planes.

The results corroborate recent studies [79, 115, 126], e.g., it is reported that 21% of cloud incidents at Azure were caused by inconsistent data formats across different components and versions [79]. We find that data-plane CSI failures are more diverse (§6.1).

Control-plane interactions are not as dominating as reported in early studies (e.g., 99% in [38, 123]). The main reasons are 1) the practice of decoupling the control and data planes in modern cloud systems, and 2) the increasing heterogeneity of the data plane. As shown in Table 1, Spark, Flink, and Hive all support multiple data stores with different data models (tables, files, tuples, and streams).

We compare our results with CSI failures in the CBS dataset, which is dominated by Hadoop issues, including MapReduce, HDFS, and HBase (HBase was a component of the Hadoop project). In the CBS dataset, control-plane CSI failures contribute to 69% of 39 studied CSI failures, much higher than the percentage in our dataset.

We also observe a significant percentage (32%) of CSI failures on the management plane, including configuration and monitoring failures. Specifically, configuration values and monitoring data fail to cooperate across systems. The management planes of cloud systems are often overlooked.

Implication. The results draw attention to cross-system data- and management-plane interactions. We believe that cloud computing stacks could be orchestrated with even more heterogeneous, complex data and management planes. Ensuring CSI correctness of these two planes is crucial.

5.2 Failure Symptoms

Finding 3: *Existing fault tolerance or recovery mechanisms are insufficient or ineffective in handling CSI failures, resulting in diverse failure impacts. Most (89/120) CSI failures are manifested through crashing behavior. See Table 3.*

	Impact	#
Control	Runtime crash/hang	8
	Startup failure	4
	Performance issue	3
	Data loss	2
	Unexpected behavior	3
Data	Job/task failure	47
	Job/task startup	6
	Wrong results	3
	Performance issues	2
	Resource leak	2
	Usability issue	1
Mgmt	Job/task crash/hang	24
	Reduced observability	8
	Unexpected behavior	5
	Performance issue	2

Table 3. Failure symptoms.

Failure impacts are based on how interactions influence the entire system and whether the failed interactions can be tolerated or mitigated. All studied systems employ extensive fault tolerance and recovery mechanisms, such as state and data replication [66–68, 116], checkpointing [54], auto-restart [74], recomputation [122], etc. While those mechanisms offer reliability for each system, they rarely protect their interactions.

We do not observe specific redundancy or recovery mechanisms for cross-system interactions. In this sense, interactions are single points of failure (SPOFs), despite redundancy in components and data. Compared with dependency failures which developers often write handling code for, CSI failures are mostly unexpected.

Implication. Crashing behavior indicates opportunities for proactive techniques, such as testing, to detect CSI issues and prevent CSI failures, e.g., as long as a test could exercise buggy interactions, it would expose CSI failures without specialized test oracles. From an observability perspective, it indicates that most of the reported CSI failures are not particularly harder to diagnose, compared with other types of failures (e.g., silent failures [70, 80, 82]).

Given that cross-system interactions are often SPOFs, there is a need to develop specialized fault-tolerance techniques for CSI. One direction is to leverage existing redundant interfaces and components or build new redundancy to address failing interactions.

5.3 Why existing tests are not enough?

Mature systems have abundant test cases, including both unit and integration tests. Unit tests do not cover cross-system interaction. Most existing integration tests only cover multiple components *within* a system, *not* across interacting systems. Currently, given a deployment of two independent and interacting systems A and B, it is unclear which tests from each system's test suite can help test their interactions, and what interactions would be covered (if any test is available). Even if there exists an integration test from A that covers certain interactions with B, it may not cover the deployed versions and configurations of B, and vice versa. As a case study, we analyzed all integration tests of Spark and found that only 6% of them cross-test dependent systems (e.g., the ones shown in Table 1). All cross-tested systems are of a specific version, which could be different from co-deployed versions.

We believe that cross-system testing and verification would greatly reduce CSI failures. However, effectively and efficiently conducting cross-system testing and verification remains an open problem. Applying traditional integration testing techniques by treating each system as a component is not an effective or cost-efficient technique given the variety of possible integrated systems and the complexities of each of their APIs.

6 Discrepancies: The Root Causes

We focus on understanding the discrepancies that manifest through interactions of the involved systems. As discussed in §2, the root causes of CSI failures are not traditional *faults* like software bugs and misconfigurations [50, 62, 76, 83, 84, 88, 103, 112, 113], but are discrepancies.

6.1 Data-Plane Discrepancies

Data-plane CSI failures are rooted in discrepancies of data operations between interacting systems. We look into their many facets to understand their discrepancy patterns.

Finding 4: *Discrepancies of data-plane CSI failures lie in many different data properties. The majority (50/61) of data-plane CSI failures are caused by metadata, namely typical metadata (42/61) such as addresses/names and data schemas, and custom metadata (8/61). The others (11/61) are caused by custom properties and API semantics. See Table 4.*

Typical metadata, including names/addresses and data schemas, look basic and innocuous, but still cause the majority (42/61) of CSI failures. For example, Flink inserts a `PROCTIME`-typed value as the `TIMESTAMP` type in Hive, but fails to translate it back (FLINK-17189). These properties perhaps can be easily unified in one project with canonical interfaces and centralized implementations. However, it is challenging to consistently manage all properties *across multiple systems*, due to the lack of systematic tooling and framework support for cross-system interactions.

Property	Description	# Fail.
Address	Name/identifier/address of the data	10
Schema	Data schema	32
⊢ Structure	Structure representation and serialization	⊢ 14
⊢ Value	Values and their interpretation (e.g., type, encoding)	⊢ 18
Custom Property	Custom metadata explicitly defined by the data store (e.g., IsCompressed, isPresentLocally)	8
API semantics	Data operation semantics (e.g., concurrency support, data element ordering in a collection)	11
Total		61

Table 4. Data properties in which discrepancies of data-plane CSI failures are rooted; these properties apply to different forms of data (table/file/stream/tuple).

Data abstraction	Address	Schema		Custom prop.	API semantics	Total
		Struct.	Value			
Table	1	13	16	0	5	35
File	8	0	0	8	2	18
Stream	1	1	2	0	4	8
KV Tuple	0	0	0	0	0	0
Total	10	14	18	8	11	61

Table 5. Data abstractions in which discrepancies of data-plane CSI failures are rooted.

Patterns	Description and Example	# Fail.
Type Confusion	Data is serialized/deserialized or type-casted in a conflicting way by the interacting systems. Ex. FLINK-17189: Flink did not translate TIMESTAMP of Hive Catalog to PROCTIME.	12
Unsupported Operations	One of the interacting systems fails to support certain data operations. Ex. SPARK-18910: Spark SQL did not support UDFs stored as jar files in HDFS.	15
Unspoken Convention	The interacting systems use different conventions for data operation. Ex. SPARK-21686: Spark failed to read column names in ORC files written by Hive	9
Undefined Values	Undefined values interpreted differently by the interacting systems. Ex. −1 represents compressed files (Figure 2)	7
Wrong API Assumptions	The consumer of data makes wrong assumptions (e.g., concurrency/order) about the data operation. Ex. SPARK-19361: Spark assumes Kafka offsets always increment by 1, which is not always true.	18
Total		61

Table 6. Discrepancy patterns and corresponding examples of data-plane CSI failures.

The eight custom metadata issues are *file properties* that are not defined by POSIX file system APIs. These properties include content (e.g., compressed or encrypted), location (e.g., local or remote), permission for impersonation, etc. Without POSIX-compliance by design [22, 58], it is natural for cloud storage systems to extend file properties in order to expose custom information. However, custom properties could be error-prone, especially when they require special handling by upstream systems. For example, in a few cases upstream systems have to operate on files stored in local and remote storage differently (FLINK-13758), or be aware of an overloaded field (e.g., Figure 2). Lastly, non-standard data APIs often have implicit semantics (e.g., [1, 22, 33]), which also exist in traditional systems that follow POSIX or SQL standards [94, 98], but are magnified by the heterogeneity and composability of cloud APIs.

Implication. Our results indicate that many data-plane CSI failures, especially those induced by metadata properties, could potentially be prevented by cross-testing data-plane interactions of involved systems *collectively.* Such cross-testing should take a data-centric view to achieve high coverage over the metadata/data properties listed in Table 5.

We explore the feasibility of cross-testing data-plane cross-system interactions in §8.

Finding 5: *Complicated data abstractions (e.g., tables) are more error-prone to CSI failures, compared with simple data abstractions. 57% (35/61) of data-plane CSI failures are induced by table-related operations. None are induced by key-value tuple operations. See Table 5.*

Table operations induced the most data-plane CSI failures, due to challenges in consistently transforming data schema, including both structure and value properties. We find that achieving interoperability between two different table-related data schemas is very challenging, due to unspoken conventions, undefined values, unsupported but expected operations, type confusion, and wrong assumptions made by data APIs (see Table 6). However, such interoperability is unavoidable. For example, to read Hive table data, Spark implements 45 unique object transformers.

Compared with tables, other simpler data abstractions have fewer CSI failures. File operations have a number of addressing failures due to heterogeneous file-path and URI conventions. This again shows that complexity without standardization or coordination is error-prone.

Data streams have few CSI failures and key-value tuples have none. However, using simple data abstractions is not always an option, because many applications need structured data with SQL-like queries. For example, Flink supports SQL by creating tables on top of data streams [16]. In Table 5, CSI failures are classified as "Stream" before table creation and as "Table" after that.

Implications: *Many discrepancy patterns of data-plane CSI failures are due to a lack of enforced data and data-operation specifications across the involved systems (Table 6).* Formal data and data-operation specifications beyond basic types and structures are not yet common in today's cloud-system engineering practices, and existing component testing does not cover CSI data discrepancies. As a result, it is hard to prevent the discrepancy patterns shown in Table 6. Many discrepancies are introduced during software evolution, i.e., code changes that affect the consistency between the interacting systems. For example, SPARK-21150 was caused by a code change that loses case sensitivity. Given the prevalence of data discrepancies, we posit that data specifications are needed for data-plane orchestration across system boundaries, especially for complex data abstractions.

Finding 6: *25% (15/61) data-plane CSI failures are root-caused by data serialization.*

We observe that developers often implement *ad-hoc* serialization on raw schemas instead of using a unified procedure. Such practice is error-prone and hard to manage. For example, SPARK-21686 was caused by the discrepancy between Spark's and Hive's custom serializers used when exchanging ORC tables. This is because for complicated, performance-critical data structures developers typically implement custom data layouts as well as *ad-hoc* serialization and deserialization with read and write optimization (e.g., Spark's serializer has ORC-specific read optimizations).

Implications. Replacing *ad-hoc* serialization with serialization libraries could prevent many data-plane CSI failures. Since popular serialization libraries (e.g., Google Protocol Buffers [29]) mainly support simple data abstractions, we posit the need for a unified serialization library that supports complicated data abstractions.

6.2 Management-Plane Discrepancies

The management plane consists of system components for *configuration*, which control and customize system behavior, and *monitoring*, which provide observability.

6.2.1 Configuration. The majority (30/39) of management-plane CSI failures are caused by configuration issues, and the remaining are related to monitoring.

Finding 7: *CSI-failure-inducing configuration issues are very different from traditional configuration issues of individual systems. The former is mostly about failures of coherently configuring multiple involved systems, while the latter is mainly on correctness checking of erroneous configuration values.*

Prior work has extensively studied misconfigurations of computer systems [40, 41, 88, 103, 112–114, 117, 125]. Most of them focus on configurations of individual systems. The essential research problem is to understand the impact of configuration values (in terms of correctness and performance), for misconfiguration detection and troubleshooting.

We find that CSI-failure inducing configuration issues have fundamentally different characteristics—they are mostly about failures of coherently configuring multiple involved systems, as shown in Table 7.

Pattern	Description and Example	# Fail.
Ignorance	Configuration settings are incorrectly ignored. Ex. SPARK-10181: Spark's Hive client ignored Kerberos configuration (keytab and principal).	12
Unexpected override	Configuration settings are incorrectly overruled. Ex. SPARK-16901: Spark incorrectly overwrote Hive's configuration when merging with Hadoop configuration.	6
Inconsist. context	Configuration values are wrong in a CSI context Ex. FLINK-19141: Flink and YARN use inconsistent resource alloc configurations (Fig. 3).	10
Mishandling configuration values	Configuration errors that break the CSI code. Ex. SPARK-15046: Spark ApplicationMaster on YARN treats an interval configuration as numeric, which is allowed to be `86400079ms`.	2
Total		30

Table 7. Discrepancy patterns and corresponding examples of configuration-related CSI failures.

In 18/30 cases, configurations are silently ignored or unexpectedly overruled. All these configurations are expected to be used by the upstream system to configure its interactions with the downstream system. A recurring pattern is that the configuration value is lost during transformation or merges with other configurations, where it is hard to know whether the values should be kept or overwritten.

In 10/39 cases, configurations are wrong in a specific CSI context (which could be correct in a different context). For example, in FLINK-19141, the resource allocation configurations are used inconsistently when a different YARN scheduler is used. The same configuration parameters have different semantics for different YARN schedulers. It is challenging to maintain correctness when the configurations are overloaded in the interactions between different systems.

Finding 8: *Parameter-related configuration issues are the majority (21/30) of configuration-induced CSI failures. The rest (9/30) are in configuration components of the involved systems.*

Prior studies categorize configuration issues into *parameter*, *component*, and *compatibility (versions)* [117]. It is reported that parameter-related issues are the majority of real-world configuration problems. We find similar characteristics in CSI failures. Specifically, 70% (21/30) of the configuration-related CSI failures were caused by parameter-related issues. All but one of these CSI-failure inducing configuration parameters were complex string-typed parameters.

Component-related configuration issues are not specific to any one configuration parameter, but lie in configuration management between interacting systems. For instance, in HIVE-11250, Hive ignores all updates to the Spark configuration via `RemoteHiveSparkClient`, due to a bug that does not set the update flag correctly.

Implications. Few works on configuration management have touched cross-system configuration. Efforts so far have focused on the correctness and compatibility of configuration parameters [49, 86, 97]. However, our results show that a more fundamental problem is to build a consistent configuration plane across multiple systems.

Currently, the configuration plane consists of a collection of configuration files from different systems. How configuration values interact is opaque. As a consequence, seemingly basic issues like incorrect ignorance and overruling are common in configuration CSI failures (Table 7). Many of these issues are not as simple as a lack of a handler or dead code addressed by prior work [43, 95, 113]; instead, configuration values are lost when propagating or merging with other configurations. Traceability of how configuration values are applied across systems could be useful.

Cross-system configuration testing [86, 103], i.e., cross-testing multiple systems under deployment (or to-be-deployed) configurations, could expose configuration-related CSI failures. One challenge is to capture unexpected behavior, as CSI misconfigurations often do not have immediate exceptions.

6.2.2 Monitoring.

We observe two patterns. First, CSI failures impair observability due to 1) not storing the expected metrics/logs, 2) not propagating the expected status code, 3) incorrectly reporting or interpreting metrics and logs between systems. For example, in SPARK-10851, Spark's R runner does not throw the right exception to YARN when an application fails, but instead exits silently; in SPARK-3627, Spark reports success for failed YARN jobs.

Second, CSI failures are caused by the discrepancies in the policies that trigger monitoring actions. For example, in FLINK-887, Flink's JobManager running as a YARN container is killed by YARN's pmem monitor if it does not appropriately adjust its JVM memory configuration.

Finding 9: *Monitoring-related CSIs are critical to reliability, especially when monitoring data is used for critical actions.*

Implication. Monitoring logic that could trigger cross-system termination actions such as kill commands should be tested to avoid unexpected crashing behaviors.

6.3 Control-Plane Discrepancies

The control plane carries out a diverse set of operations—service registration, scheduling, load balancing, resource management, etc. Surprisingly, they contribute to fewer CSI failures than the data and management planes.

Finding 10: *Most control-plane CSI failures are rooted in discrepancies of implicit properties, including implicit API semantics and state/resource inconsistencies. See Table 8.*

The majority of control-plane CSI failures are caused by the violation of implicit API semantics (such as thread safety, synchrony, ordering, etc.), which are hard to check effectively. Another common discrepancy pattern involves inconsistent views of the interacting systems' states or resources, which lead to non-cooperation between the systems.

Pattern	Description and example	# Fail.
API semantic violation	Upstream violates semantics of downstream APIs. Ex. FLINK-12342: Flink uses container-request API asynchronously (Figure 1).	13
State/ resource inconsist.	Interacting systems have inconsistent views of the system states or resources; Ex. HBASE-537: HBase wrongly assumed HDFS NameNode readiness when it was in safe mode.	5
Feature inconsist.	Upstream assumes feature consistency across different downstream versions/configurations. Ex. YARN-9724: Spark assumed availability of getYarnClusterMetrics APIs in all YARN modes	2
Total		20

Table 8. Discrepancy patterns and corresponding examples of control-plane CSI failures.

The inconsistent views are caused by diverse reasons, including asynchrony-induced stale states due to concurrent events (e.g., HBASE-16621), unawareness of safe/stealth mode (e.g., HBASE-537), and inconsistent resource calculations between two systems (e.g., SPARK-2604). Only asynchrony is a fundamental distributed systems challenge [46, 104]; the others are caused by a lack of cooperation.

Feature inconsistencies are common software engineering flaws, which are in the same vein as unsupported operations in data-plane CSI failures (Table 6).

Finding 11: *API misuses, despite being a classic problem, are still common defects and contribute to the majority (13/20) of control-plane CSI failures. The main patterns are implicit semantic violation (8/13) and incorrect invocation context (5/13).*

Implicit API semantics such as synchronous call requirements (e.g., Figure 1), concurrency, ordering, and even thread-safety are hard to check and reason about in practice, despite recent progress [65, 78]. The challenges of adhering to API semantics are further magnified across the system boundary, due to a lack of machine-checkable specifications and a shortage of systematic testing and verification.

The other five API misuse cases are caused by API invocation in a wrong context. For example, in FLINK-5542, an API used for reading local vcore information is used in a global context, causing misinformation of available cores. In FLINK-4155, partition discovery should be done in a Flink context to interact with Kafka; however, it is invoked in a client context, which may not have access to the Kafka cluster.

Implications. Despite decades of research in API specification checking [39, 77, 99, 109], it is still a challenging problem to detect API misuses or semantic violations, especially for implicit and hard-to-check API semantics. CSI failures provide another motivation to rethink API specifications at the system boundaries. An arguably more tangible solution is

Fix Pattern	Description and example	# Fai.
Checking	Check specific conditions to avoid CSI issues Ex. SPARK-27239: Include the -1 as a valid file size in the checking code (Figure 4).	38
Error handling	Add/improve exceptions handling of CSI issues Ex. FLINK-3081: Add try-catch block to capture exceptions thrown by CSI operations (Figure ??).	8
Interaction	Fix cross-system interaction code Ex. FLINK-12342: Change the interaction from sync to async (Figure 5, Resolution#3)	69
Others	No merged fixes or document-only fixes	5
Total		120

Table 9. Fix patterns of the evaluated CSI failures.

to design simple and consistent control-plane APIs. Kubernetes presents one good example of such a design, where a unified API and object-metadata structure ensures semantic consistency and transparency [47].

7 Fixes

We discuss code fixes for the studied CSI failures. All but five CSI failures have merged fixes (Table 9). Amongst these five cases, three do not have merged fixes and two only enhance documentation. We focus on the 115 issues with fixes.

Finding 12: *In 40% (46/115) CSI failures, the merged fixes improve condition checking and error handling instead of repairing the failed interactions.*

Table 9 details the fix patterns. A check or error handling fix could prevent a particular CSI failure by avoiding the failure condition or by gracefully reacting, but cannot fundamentally prevent CSI failures of similar kinds, and especially over other systems. Figure 4 shows the fix for the CSI failure in Figure 2 by including -1 as a valid value in the checking code. However, given the undefined value and the overloaded file length, other upstream systems may be vulnerable to the same CSI failure.

In fact, in all but one CSI failure, the fixes were implemented by the upstream system, because (1) the upstream has more incentives to fix the problem and (2) it is easier to fix it in the upstream than the downstream where changes need to accommodate for backward compatibility. In the only exception (YARN-9724 in Table 8), the downstream fixed an API contract violation, which was considered a bug.

In the remaining 58% of the CSI failures, the fixes focus on improving the failure-inducing interactions. However, not every fix aimed at improving the interaction fundamentally resolves the CSI issue. For example, the fix for YARN-2790 moves token renewal in YARN close to the operation on HDFS where it is consumed, to reduce the likelihood of token expiration. Nonetheless, expiration can still happen due to a small timeout value or delays in early operations.

```
- require(length >= 0, s"length ($length)
   cannot be negative")
+ require(length >= -1, s"length ($length)
   cannot be smaller than -1")
/* spark/.../rdd/InputFileBlockHolder.scala */
```

Figure 4. The fix for SPARK-27239 (Figure 2), which includes -1 as a valid file size in the checking code.

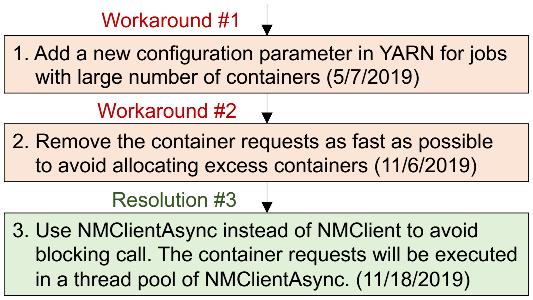
Workaround #1
1. Add a new configuration parameter in YARN for jobs with large number of containers (5/7/2019)
Workaround #2
2. Remove the container requests as fast as possible to avoid allocating excess containers (11/6/2019)
Resolution #3
3. Use NMClientAsync instead of NMClient to avoid blocking call. The container requests will be executed in a thread pool of NMClientAsync. (11/18/2019)

Figure 5. The fix process for FLINK-12342 (Figure 1) which patched workarounds before a more fundamental resolution.

Note that fixing the interaction typically takes more effort than adding checks and error handling. In a few cases, we observe that developers first added checks or handling as workarounds for the CSI failure and later fixed the interaction. Figure 5 shows the fix process for the CSI failure described in Figure 1. The immediate workarounds are to make the 500ms interval configurable and to decrement the pending container count immediately, which can reduce the occurrence of asynchronous interactions. Eventually, the developers rewrote the container startup logic in Flink to make it asynchronous as a more fundamental fix.

Implication. The common fix patterns of checking and error handling by one upstream indicate error-proneness of similar CSI failures in other upstreams. One potential technique is to extend existing precondition analysis and error propagation analysis to be able to cross the system boundaries and apply them to enhance existing checking and error handling code.

Finding 13: *In 69% (79/115) CSI failures, fixes were applied to code in the upstream system specific to interaction with a downstream system. Furthermore, among these 79 cases, fixes for 68 (86%) cases resided in dedicated "connector" modules.*

Many systems maintain code for cross-system interactions in modularized connectors (also named handlers and clients). Take Flink as an example. Its connectors to Hive and Kafka are maintained in flink-connectors with 32 other connectors; the connector to HDFS resides in flink-filesystems along with connectors for other downstream file systems; the connector to YARN is in flink-yarn. Connector code contributes to less than 5% of the entire codebase, but is the target of fixing more than half of the studied CSI issues. In the other 11 (out of 79) cases, the fixes were applied to code specific to the downstream system but not in any connector modules. In those cases, the CSI code lacks modularity.

In the remaining 36 cases, the fixes were applied to code not specific to the downstream system being interacted with, but to generic code that is used to interact with multiple downstream systems. For example, in SPARK-10122, the fix corrected an issue in PySpark's core streaming module that lost a data attribute during compaction. Such issues could potentially affect multiple downstream systems.

Implication. With current fix practices, we expect CSI failures to continue being prevalent. A significant number of CSI failures were fixed as afterthoughts to the failures; the fixes were not sufficiently general (Finding 13) and therefore, the same CSI issues could be experienced in a different upstream system (Finding 12). We show that the connectors could be an effective starting point for CSI testing and verification. The connectors consist of relatively little code, but are associated with many CSI failures.

8 Cross-System Testing: A Case Study

We present a case study on cross-system-testing of the Spark-Hive data plane. We have two goals. First, we evaluate whether CSI issues, especially those on data planes, are still common and recurring, given the fixes of the studied CSI failures. We choose to test the Spark-Hive data plane because it had the most issues in our dataset (Table 1), as well as the most fixes.

Second, we assess whether cross-system testing is a feasible solution to detect pre-deployment CSI issues and prevent failures. We apply the simplest form of cross-system data-plane testing—check whether the two systems, Spark and Hive, each process data consistently by writing the data and then reading it through various interfaces of the two systems.

Our results are specific to the Spark-Hive data plane. Developing a more general tool is our future work.

8.1 Methodology

Setup. Figure 6 illustrates the setup, which writes and reads data across three different interfaces (SparkSQL, DataFrame, and HiveQL). Spark (upstream) uses Hive (downstream) as a backend datastore. Spark uses a series of connectors to transform the data into a form compatible with Hive. Hive further transforms data based on the requested data format, including ORC [8], Parquet [9], or Avro [2]. We use Spark v3.2.1 (the latest version at the time of writing) for Spark-to-Spark tests. We use Hive v3.1.2 (the latest version at the time of writing) and Spark v2.3.0 for both Spark-to-Hive and Hive-to-Spark tests, because Spark does not support an external Hive instance beyond v2.3.0.

Test inputs. We generate input data based on the publicly documented specifications of each interface [15, 30]. The generated inputs cover *all* the data types that are supported by each interface. These inputs include both valid and invalid data—the former are used to test expected behavior, while the latter are used to test error handling behavior. In total, we generated 422 and 116 values that cover the data types; 210 are valid and 212 are invalid.

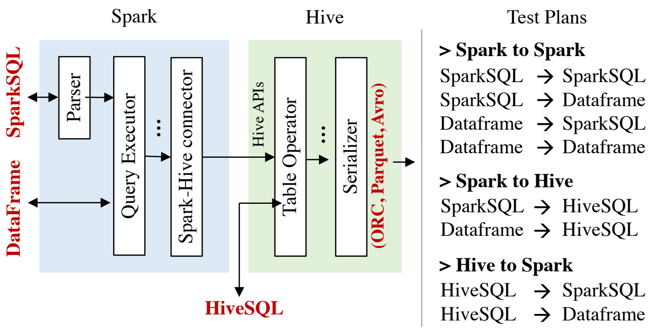

Figure 6. Setup of cross-system testing for the Spark-Hive data-plane, which checks whether data is processed consistently.

Test oracles. We apply three basic oracles to capture potential discrepancies: (1) *Write-Read (WR):* For valid data, the data read from the query should be the data written earlier. The read following the write can be performed by a different interface. (2) *Error handling (EH):* For invalid data, the data should be either rejected or corrected with feedback (e.g., log messages) during the writes. (3) *Differential (Diff):* For valid and/or invalid data, results/behavior should be consistent across interfaces and backend formats (ORC/Avro/Parquet). The read values between different write-read interface pairs are compared.

Artifact. The testing scripts and data are available at https://github.com/xlab-uiuc/csi-test-ae.

8.2 Results and Findings

The simple testing described in §8.1 exposed 15 distinct discrepancies on the Spark-Hive data plane, caused by inconsistent data processing across Spark and Hive. We reported 13 of them. So far, nine have been acknowledged, among which two are confirmed as bugs. For the rest, developers pointed out that the discrepancies can be resolved by custom configurations including the two unreported discrepancies. The discrepancies caused the following problems:

Cannot read what was written (2/15). In one case (SPARK-39075), if DataFrame is used to write and then read data through Avro, data with the BYTE or SHORT type cannot be read after writing. The root cause is that Avro converts BYTE/SHORT into INT upon serialization; however, upon deserialization, it misses the case to convert INT back to BYTE/SHORT, but throws `IncompatibleSchemaException`. In another case (SPARK-39158), valid decimal values written from DataFrame cannot be read from HiveQL.

Type violations (2/15). When writing and then reading from SparkSQL, BYTE/SHORT values will be converted into

INT, which violates its original type (HIVE-26533, SPARK-40409). Moreover, the conversion also creates a side effect of violating case sensitivity of the column name, with a warning message of "not case preserving." This is because when reading data from Hive, Spark falls back to Hive schema (which is case-insensitive) if it fails to use Spark's native format (which is case-sensitive).

Exposing internal configurations of the downstream to the upstream (5/15). The serializers (Avro/ORC/Parquet) are not fully compatible. For example, Avro does not support non-string keys in maps; ORC and Parquet support it (HIVE-26531); `spark.sql.hive.caseSensitiveInferenceMode` (a Spark configuration) only works with ORC and Parquet, but not Avro. As a result, the serializer has to be specified when a table is created and cannot be easily changed. This means (1) internal configuration of downstream systems cannot be transparent to the upstream, and (2) if the downstream changes the serializer, it will break the upstream jobs.

Inconsistent error behavior across interfaces (7/15). For example, decimal values with too much precision throw an exception by SparkSQL but evaluate to NULL by DataFrame (SPARK-40439).

Relying on custom (non-default) configurations (8/15). For example, SPARK-40439 mentioned that the above can be resolved by setting `spark.sql.storeAssignmentPolicy` to `legacy`. However, it is a daunting task to achieve consistent system behavior by fine-tuning configurations (e.g., SparkSQL alone has 350+ configuration parameters). Testing systems under the deployment configuration (not the default configuration) could potentially expose such discrepancies.

Finding 15: *Data-plane CSI issues are still common and recurring across the latest versions of Spark and Hive, despite many being reported and fixed.*

Implication. It is critical to understand and resolve inconsistencies at deployment time to prevent potential CSI failures. *Cross-testing co-deployed, interacting systems under deployment configuration could be an effective approach to prevent CSI failures, especially data-plane ones.*

9 Threats to Validity

The open-source CSI dataset may not reflect all CSI failures in deployed systems. Specifically, CSI failures caused by misoperation, such as with capacity planning and load management, are unlikely to be reported to developers. For example, we did not observe metastable failures in our dataset, despite the fact that CSI failures by definition include metastable failures that occur across systems. Closed-source distributed systems may have different characteristics.

Also, CSI failures with explicit symptoms (e.g., crashing behavior) are more likely to be observed and reported than the silent ones. This could be a source of bias in our dataset.

For the open-source CSI dataset, we selected systems based on their occurrence in our heuristic sample from the aggregated JIRA database (§4). Many such issues had data processing systems as upstream systems. Other systems could have different characteristics, especially those with simpler interfaces and fewer subsystems. We also excluded issues with "minor" severity, which could have different characteristics.

Our sampling is imperfect due to the lack of a labeled dataset (§4). We used heuristics to identify multi-system issues, from which we selected systems and identified failures.

10 Discussion

There is no silver bullet for CSI failures. As evidence, despite decades of research, data/type inconsistencies and API misuses remain largely unsolved problems in practice and have been causing many CSI failures. The emerging cross-system interactions magnify the integration challenges. We now discuss promising directions on CSI issues.

Cross-system testing, verification, and model checking. One intuitive way of addressing CSI issues is to treat different (sub-)systems as traditional software modules and to apply existing testing, verification, or model checking techniques. As demonstrated in §8, we believe such efforts can potentially detect many CSI issues and can be performed in a continuous manner (e.g., before changing related code and configurations in production). Today's DevOps practices make it possible to perform white-box techniques.

One challenge is to scale traditional techniques for unit and module levels (e.g., symbolic execution, concolic testing) to the (sub-)system level. Finding 14 points to analyzing connectors for the interaction as a good starting point, as they constitute a small subset of the entire codebase.

Another promising direction is cross-system testing with feedback-based fuzzing. Recently, Schumilo et al. [101] scaled feedback-based fuzz testing to network-based systems with fast virtual machine snapshot and restore techniques. Though applied to single-node network systems, their *hypervisor-based snapshot fuzzing* is a promising technique to scale feedback-based fuzzing to cross-system testing.

Unification and standardization. One direction to address heterogeneity is unification—building a standardized layer that abstracts away system-specific details. For example, a unified serialization library that can handle conversion between the system's data and all desired data formats could simplify the data translation process between systems. However, market motivation remains a challenge [102]. Furthermore, standardization may not be a panacea to all CSI issues, as reflected by the POSIX and SQL standards.

Rethinking data/API specifications. Many studied CSI failures can potentially be addressed with comprehensive, machine-checkable data/API specifications. Yet, specification engineering is costly, and specification mining remains immature [77]. CSI failures can serve as another motivation to rethink data/API specifications, with more targeted, specialized techniques.

Serialization library for complicated data abstraction. Many CSI failures reside in *ad-hoc* (de)serialization for complex data abstractions (e.g., tables) with custom performance optimizations. For example, Spark's deserializer implements certain read optimizations for ORC tables that Hive's deserializer does not. One could develop and promote unified serialization libraries that support complex data abstractions with workload-specific read and write optimizations.

Change analysis for cross-system interactions. Many CSI issues are introduced during software evolution. We observe that system interfaces constantly need changes when new features/configurations are added. Traditional regression testing [119] does not address the interface problem. New techniques are needed for reasoning about impacts of changes regarding cross-system interactions.

CSI fault tolerance. We find that existing system implementations do not have effective mechanisms to tolerate CSI issues (see §5.2). Mostly, upstream systems treat CSI failures in the same way as dependency failures. However, such practices miss the opportunities of tolerating CSI failures, because the downstream systems are well available. A potential direction is to leverage the diversity of existing interfaces to build interaction redundancy across systems.

11 Related Work

Cloud system failure analysis. Many studies [34, 37, 45, 51, 55, 62–64, 70, 76, 79, 81, 87, 92, 96, 120, 126] analyzed cloud and distributed system failures, with different focuses/perspectives, including fail-slow behavior [64, 118], gray and partial failures [70, 81], network partitions [34, 37], upgrade failures [124, 126], and metastable failures [45, 69]. However, few of them discussed CSI failures comprehensively.

Liu et al., [79] reported data-format issues as the major root causes of cloud incidents in Azure. The reported prevalence corroborates our results. We provide a deep analysis on data-plane CSI failures including the reported data-format issues. We show that CSI failures are much broader.

Gunawi et al., [62] observed cross-system failures in the CBS dataset (see §4). Unfortunately, they left the analysis of cross-system failures as future work. Note that only 37% (39/105) of their cross-system failures are CSI failures.

Studies on software bugs and misconfigurations. There has been rich literature that studies software bugs and misconfigurations, including those in cloud systems [48, 56, 62, 76, 88, 103] and other software systems such as OS kernels, file systems, server systems, etc. [50, 83, 84, 90, 117].

As we have shown in the paper, CSI issues are not traditional bugs. Often, there is no strictly defined bug in either the upstream or the downstream system. Regarding configuration, as discussed in §6.2.1, most of configuration-related CSI failures are not caused by erroneous values, but by inconsistently configuring involving systems.

Integration testing. Recent work, e.g., DUPTester [126] and ZebraConf [86] leverages existing integration tests to detect software upgrade bugs and misconfigurations. The key idea is to test a system with components under different versions, a common state under continuous, rolling upgrades. Such integration testing cannot directly address CSI issues, as CSI issues may not be caused by version incompatibility during software upgrades. However, systematic cross-system integration testing can be extended to combat CSI failures.

Cross-checking multiple implementations that follow same specifications is a powerful principle to detect semantic bugs [53, 65, 89, 121]. Juxta [89] cross-checks multiple Linux file system implementations that obey same API semantics. The idea can be extended and applied to the cross-system interactions, if multiple implementations are available.

API specifications and invariant mining. Techniques for inferring API specifications and invariants have been studied [61, 85, 99, 115]. Existing API specification mining techniques are not mature enough to be used for bug finding due to false information [77]. I4 [85] infers inductive invariants to verify distributed protocols, which can potentially address CSI failures due to discrepancies in protocol implementations. Akita infers API models by monitoring the API traffic to find regressions [115].

Type safety and unit correctness. Type systems are developed to ensure type safety, including specialized solutions for unit correctness [73, 93, 100]. CSI discrepancies are broader and more diverse than type/unit inconsistencies.

12 Concluding Remarks

This paper presents the first in-depth analysis of cross-system interaction (CSI) failures from proprietary and open-source distributed systems. We found that CSI failures account for a significant portion (20%) of the most catastrophic cloud incidents, indicating a demand for research efforts on understanding and addressing CSI failures. Our study revealed over a dozen findings with concrete implications. In particular, we discussed approaches to investigate for cross-system testing and analysis, standardization and specification of cross-system interaction, and CSI-specific fault tolerance. As a feasibility study, we developed a data-plane cross-system testing tool for Spark and Hive, and it exposed previously unknown CSI issues. We expect CSI failures to become more common, given the growing complexity of cloud systems today and practice of composing them into versatile computing stacks. We believe future research can leverage the guidance provided by our study to combat CSI failures.

Acknowledgement

We thank anonymous reviewers and our shepherd, Roberto Natella, for their insightful comments. We thank Darko Marinov, Justin Meza, Xudong Sun, and Lalith Suresh for valuable feedback and discussions that helped improve our work. We thank Jack Chen, Xudong Sun, Jinghao Jia, and Le Xu who helped with a pivot study of CSI failures. This work was funded in part by NSF CNS-2130560, CNS-2145295, IIS-1909577, CNS-1908888, a Facebook Core Systems Faculty Research gift, a Capital One gift, and a Microsoft gift.

References

[1] ANSI Compliance. https://spark.apache.org/docs/latest/sql-ref-ansi-compliance.html.

[2] Apache Avro. https://avro.apache.org/.

[3] Apache Flink. https://flink.apache.org/.

[4] Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[5] Apache HBase. https://hbase.apache.org/book.html.

[6] Apache Hive. https://cwiki.apache.org/confluence/display/Hive/Tutorial.

[7] Apache Kafka. https://kafka.apache.org/documentation/.

[8] Apache ORC: The smallest, fastest columnar storage for Hadoop workloads. https://orc.apache.org/.

[9] Apache Parquet. https://parquet.apache.org/.

[10] Apache Spark Codebase. https://github.com/apache/spark.

[11] Apache Spark Website. https://spark.apache.org/.

[12] ASF JIRA. https://issues.apache.org/jira/secure/Dashboard.jspa.

[13] AWS Post-Event Summaries. https://aws.amazon.com/premiumsupport/technology/pes/.

[14] Azure status history. https://status.azure.com/en-us/status/history/.

[15] Data types (Databricks SQL). https://docs.databricks.com/sql/language-manual/sql-ref-datatypes.html#data-types-databricks-sql.

[16] Dynamic Tables. https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/table/concepts/dynamic_tables/.

[17] Google Cloud Service Health. https://status.cloud.google.com/summary.

[18] Google was hit with massive outage, including youtube, gmail and google classroom | cnn business. https://www.cnn.com/2020/12/14/tech/google-youtube-gmail-down/index.html.

[19] Google's apps crash in a worldwide outage. - the new york times. https://www.nytimes.com/2020/12/14/business/google-down-worldwide.html.

[20] Hadoop Common. https://github.com/apache/hadoop/tree/trunk/hadoop-common-project.

[21] Hadoop Distributed File System. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[22] Implicit assumptions of the Hadoop FileSystem APIs. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/filesystem/introduction.html#Implicit_assumptions_of_the_Hadoop_FileSystem_APIs.

[23] Incident affecting Google App Engine. https://status.cloud.google.com/incidents/NuaWbbv8n8V8PMHNR7kT.

[24] Incident affecting Google BigQuery. https://status.cloud.google.com/incidents/qq7VS3aLtp6Nmgs5Nux4.

[25] Incident affecting Google Cloud Infrastructure Components, Google Cloud Support, Google Cloud Console, Google BigQuery, Google Cloud Storage, Google Cloud Networking, Google Kubernetes Engine, Virtual Private Cloud (VPC). https://status.cloud.google.com/incidents/cFXPsFUnUELR8U2bQeGz.

[26] Incident affecting Google Compute Engine, Google Cloud Networking, Access Approval, Google App Engine. https://status.cloud.google.com/incidents/1tX748pbxW2JjTUuTJsx.

[27] Integration with Cloud Infrastructures. https://spark.apache.org/docs/latest/cloud-integration.html.

[28] Massive google outage takes millions offline. https://www.forbes.com/sites/paulmonckton/2020/12/14/massive-google-outage-takes-millions-offline/?sh=40f33d060ad1.

[29] Protocol buffers guide. https://developers.google.com/protocol-buffers/docs/proto.

[30] Spark SQL Guide. https://spark.apache.org/docs/latest/sql-programming-guide.html.

[31] Mars Climate Orbiter Mishap Investigation Board Phase I Report. https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf, Nov. 1999.

[32] CBS: Cloud Bug Study Database. https://ucare.cs.uchicago.edu/projects/cbs/, 2014.

[33] Apache Hive SQL Conformance. https://cwiki.apache.org/confluence/display/Hive/Apache+Hive+SQL+Conformance, Nov. 2018.

[34] Alfatafta, M., Alkhatib, B., Alquraan, A., and Al-Kiswany, S. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[35] Allspaw, J. Fault injection in production: Making the case for resilience testing. *Communications of the ACM (CACM) 55*, 10 (Oct 2012), 48–52.

[36] Alluxio Docs. The Need for a New Data Orchestration Platform. https://www.alluxio.io/data-orchestration/.

[37] Alquraan, A., Takruri, H., Alfatafta, M., and Al-Kiswany, S. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[38] Altekar, G., and Stoica, I. Focus Replay Debugging Effort on the Control Plane. In *Proceedings of the 6th Workshop on Hot Topics in System Dependability (HotDep'10)* (Oct. 2010).

[39] Amann, S., Nguyen, H. A., Nadi, S., Nguyen, T. N., and Mezini, M. A Systematic Evaluation of Static API-Misuse Detectors. *IEEE Transactions on Software Engineering 45*, 12 (Dec. 2019), 1170–1188.

[40] Attariyan, M., Chow, M., and Flinn, J. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)* (Oct. 2012).

[41] Attariyan, M., and Flinn, J. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Oct. 2010).

[42] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., and Rosenthal, C. Chaos engineering. *IEEE Software 33*, 3 (May 2016), 35–41.

[43] Behrang, F., Cohen, M. B., and Orso, A. Users Beware: Preference Inconsistencies Ahead. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Aug. 2015).

[44] Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)* (Nov. 2016).

[45] Bronson, N., Aghayev, A., Charapko, A., and Zhu, T. Metastable Failures in Distributed Systems. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)* (May 2021).

[46] Brooker, M. The Fundamental Mechanism of Scaling. http://brooker.co.za/blog/2021/01/22/cloud-scale.html, 2020.

[47] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes. *Communications of the ACM 59*, 5 (May 2016), 50–57.

[48] Chen, H., Dou, W., Jiang, Y., and Qin, F. Understanding Exception-Related Bugs in Large-Scale Cloud Systems. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)* (Nov. 2019).

[49] Chen, Q., Wang, T., Legunsen, O., Li, S., and Xu, T. Understanding and Discovering Software Configuration Dependencies in Cloud and Datacenter Systems. In *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'20)* (November 2020).

[50] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)* (Oct. 2001).

[51] Cotroneo, D., De Simone, L., Liguori, P., Natella, R., and Bidokhti, N. How Bad Can a Bug Get? An Empirical Analysis of Software Failures in the OpenStack Cloud Computing Platform. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)* (Aug. 2019).

[52] Databricks Docs. Databricks architecture overview. https://docs.databricks.com/getting-started/overview.html.

[53] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP'01)* (Oct. 2001).

[54] Flink Docs. Checkpointing. https://nightlies.apache.org/flink/flink-docs-release-1.14/docs/dev/datastream/fault-tolerance/checkpointing/.

[55] Ford, D., Labelle, F., Popovici, F. I., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)* (Oct. 2010).

[56] Gao, Y., Dou, W., Qin, F., Gao, C., Wang, D., Wei, J., Huang, R., Zhou, L., and Wu, Y. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)* (Nov. 2018).

[57] Gember-Jacobson, A., Wu, W., Li, X., Akella, A., and Mahajan, R. Management Plane Analytics. In *Proceedings of the 2015 Internet Measurement Conference (IMC'15)* (Oct. 2015).

[58] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)* (Oct. 2003).

[59] Google Cloud. What is a hybrid cloud? https://cloud.google.com/learn/what-is-hybrid-cloud.

[60] Govindan, R., Minei, I., Kallahalla, M., Koley, B., and Vahdat, A. Evolve or Die: High-Availability Design Principles Drawn from Google's Network Infrastructure. In *Proceedings of the 2011 ACM SIGCOMM Conference (SIGCOMM'11)* (Aug. 2016).

[61] Grant, S., Cech, H., and Beschastnikh, I. Inferring and Asserting Distributed System Invariants. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)* (May 2018).

[62] Gunawi, H. S., Hao, M., Leesatapornwongsa, T., Patana-anake, T., Do, T., Adityatama, J., Eliazar, K. J., Laksono, A., Lukman, J. F., Martin, V., and Satria, A. D. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC'14)* (Nov. 2014).

[63] Gunawi, H. S., Hao, M., Suminto, R. O., Laksono, A., Satria, A. D., Adityatama, J., and Eliazar, K. J. Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages. In *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC'16)* (Oct. 2016).

[64] Gunawi, H. S., Suminto, R. O., Sears, R., Golliher, C., Sundararaman, S., Lin, X., Emami, T., Sheng, W., Bidokhti, N., McCaffrey, C., Grider, G., Fields, P. M., Harms, K., Ross, R. B., Jacobson, A., Ricci, R., Webb, K., Alvaro, P., Runesha, H. B., Hao, M., and Li, H. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)* (Feb. 2018).

[65] Gyori, A., Lambeth, B., Shi, A., Legunsen, O., and Marinov, D. NonDex: A Tool for Detecting and Debugging Wrong Assumptions on Java API Specifications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)* (Nov. 2016).

[66] HBase Docs. HBase Cluster Replication. https://hbase.apache.org/book.html#_cluster_replication.

[67] HBase Docs. HBase Write Ahead Log. https://hbase.apache.org/book.html#wal.

[68] HDFS Docs. Data Replication. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html#Data_Replication.

[69] Huang, L., Magnusson, M., Muralikrishna, A. B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T., and Charapko, A. Metastable Failures in the Wild. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[70] Huang, P., Guo, C., Zhou, L., Lorch, J. R., Dang, Y., Chintalapati, M., and Yao, R. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS-XVI)* (May 2017).

[71] Istio Docs. Architecture. https://istio.io/latest/docs/ops/deployment/architecture/.

[72] Jia, Z., Li, S., Yu, T., Zeng, C., Xu, E., Liu, X., Wang, J., and Liao, X. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)* (May 2021).

[73] Jiang, L., and Su, Z. Osprey: A Practical Type System for Validating Dimensional Unit Correctness of C Programs. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)* (May 2006).

[74] Kafka Docs. Auto Restart. https://kafka.apache.org/documentation/streams/architecture#streams_architecture_recovery.

[75] Kubernetes Docs. Control Plane Components. https://kubernetes.io/docs/concepts/overview/components/.

[76] Leesatapornwongsa, T., Lukman, J. F., Lu, S., and Gunawi, H. S. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)* (Mar. 2016).

[77] Legunsen, O., Hassan, W. U., Xu, X., Rosu, G., and Marinov, D. How Good are the Specs? A Study of the Bug-Finding Effectiveness of Existing Java API Specifications. In *Proceedings of the 31th IEEE/ACM International Conference on Automated Software Engineering (ASE'16)* (2016).

[78] Li, G., Lu, S., Musuvathi, M., Nath, S., and Padhye, R. Efficient Scalable Thread-Safety-Violation Detection. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (2019).

[79] Liu, H., Lu, S., Musuvathi, M., and Nath, S. What Bugs Cause Production Cloud Incidents? In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS'19)* (Nov. 2019).

[80] Lou, C., Chen, C., Huang, P., Dang, Y., Qin, S., Yang, X., Li, X., Lin, Q., and Chintalapati, M. RESIN: A Holistic Service for Dealing with Memory Leaks in Production Cloud Infrastructure. In *Proceedings*

of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22) (July 2022).

[81] Lou, C., Huang, P., and Smith, S. Understanding, Detecting and Localizing Partial Failures in Large System Software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Feb. 2020).

[82] Lou, C., Jing, Y., and Huang, P. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)* (July 2022).

[83] Lu, L., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Lu, S. A Study of Linux File System Evolution. *ACM Trans. Storage 10*, 1 (Jan. 2014).

[84] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGARCH Comput. Archit. News 36*, 1 (Mar. 2008), 329–339.

[85] Ma, H., Goel, A., Jeannin, J.-B., Kapritsos, M., Kasikci, B., and Sakallah, K. A. I4: Incremental Inference of Inductive Invariants for Verification of Distributed Protocols. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (2019).

[86] Ma, S., Zhou, F., Bond, M. D., and Wang, Y. Finding Heterogeneous-Unsafe Configuration Parameters in Cloud Systems. In *Proceedings of the 16th ACM European Conference on Computer Systems (EuroSys'21)* (Apr. 2021).

[87] Maurer, B. Fail at Scale: Reliability in the Face of Rapid Change. *Communications of the ACM 58*, 11 (Nov. 2015), 44–49.

[88] Mehta, S., Bhagwan, R., Kumar, R., Ashok, B., Bansal, C., Maddila, C., Bird, C., Asthana, S., and Kumar, A. Rex: Preventing Bugs and Misconfiguration in Large Services using Correlated Change Analysis. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Feb. 2020).

[89] Min, C., Kashyap, S., Lee, B., Song, C., and Kim, T. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP'15)* (Oct. 2015).

[90] Nagaraja, K., Oliveira, F., Bianchini, R., Martin, R. P., and Nguyen, T. D. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)* (Dec. 2004).

[91] OpenStack Docs. Logical architecture. https://docs.openstack.org/install-guide/get-started-logical-architecture.html.

[92] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. Why Do Internet Services Fail, and What Can Be Done About It? In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS'03)* (Mar. 2003).

[93] Ore, J.-P., Detweiler, C., and Elbaum, S. Phriky-Units: A Lightweight, Annotation-Free Physical Unit Inconsistency Detection Tool. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)* (2017).

[94] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)* (Oct. 2014).

[95] Rabkin, A., and Katz, R. Static Extraction of Program Configuration Options. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)* (May 2011).

[96] Rabkin, A., and Katz, R. How Hadoop Clusters Break. *IEEE Software Magazine 30*, 4 (July 2013), 88–94.

[97] Ramachandran, V., Gupta, M., Sethi, M., and Chowdhury, S. R. Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications. In *Proceedings of the 6th International Conference on Autonomic Computing and Communications (ICAC'09)* (June 2009).

[98] Rigger, M., and Su, Z. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[99] Robillard, M. P., Bodden, E., Kawrykow, D., Mezini, M., and Ratchford, T. Automated API Property Inference Techniques. *IEEE Transactions on Software Engineering 39*, 5 (May 2013), 613–637.

[100] Rosu, G., and Chen, F. Certifying Measurement Unit Safety Policy. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)* (Oct. 2003).

[101] Schumilo, S., Aschermann, C., Jemmett, A., Abbasi, A., and Holz, T. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys'22)* (Apr. 2022).

[102] Stoica, I., and Shenker, S. From Cloud Computing to Sky Computing. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS'21)* (May 2021).

[103] Sun, X., Cheng, R., Chen, J., Ang, E., Legunsen, O., and Xu, T. Testing Configuration Changes in Context to Prevent Production Failures. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)* (Nov. 2020).

[104] Sun, X., Suresh, L., Ganesan, A., Alagappan, R., Gasch, M., Tang, L., and Xu, T. Reasoning about modern datacenter infrastructures using partial histories. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)* (May 2021).

[105] Treynor, B., Dahlin, M., Rau, V., and Beyer, B. The Calculus of Service Availability. *Communications of the ACM (CACM) 60*, 9 (Sept. 2017), 42–47.

[106] van Renesse, R., Weatherspoon, H., Shen, Z., and Song, W. The Supercloud: Applying Internet Design Principles to Interconnecting Clouds. In *IEEE Internet Computing (IEEE Internet Computing'18)* (Mar. 2018).

[107] Veeraraghavan, K., Meza, J., Michelson, S., Panneerselvam, S., Gyori, A., Chou, D., Margulis, S., Obenshain, D., Padmanabha, S., Shah, A., Song, Y. J., and Xu, T. Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)* (Oct. 2018).

[108] Wang, Y., Wen, M., Liu, Y., Wang, Y., Li, Z., Wang, C., Yu, H., Cheung, S.-C., Xu, C., and Zhu, Z. Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)* (July 2020).

[109] Wen, M., Liu, Y., Wu, R., Xie, X., Cheung, S.-C., and Su, Z. Exposing Library API Misuses via Mutation Analysis. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)* (May 2019).

[110] Xia, H., Zhang, Y., Zhou, Y., Chen, X., Wang, Y., Zhang, X., Cui, S., Hong, G., Zhang, X., Yang, M., and Yang, Z. How Android Developers Handle Evolution-Induced API Compatibility Issues: A Large-Scale Study. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)* (July 2020).

[111] Xia, W., Wen, Y., Foh, C. H., Niyato, D., and Xie, H. A Survey on Software-Defined Networking. *IEEE Communications Surveys & Tutorials 17*, 1 (June 2014), 27–51.

[112] Xu, T., Jin, X., Huang, P., Zhou, Y., Lu, S., Jin, L., and Pasupathy, S. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)* (Nov. 2016).

[113] Xu, T., Zhang, J., Huang, P., Zheng, J., Sheng, T., Yuan, D., Zhou, Y., and Pasupathy, S. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)* (Nov. 2013).

[114] Xu, T., and Zhou, Y. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR) 47*, 4 (July 2015).

[115] Yang, J. Modeling API Traffic to Catch Breaking Changes. https://www.akitasoftware.com/blog-posts/modeling-api-traffic-to-catch-breaking-changes, 2021.

[116] YARN Docs. YARN ResourceManager HA. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html.

[117] Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L. N., and Pasupathy, S. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)* (Oct. 2011).

[118] Yoo, A., Wang, Y., Sinha, R., Mu, S., and Xu, T. Fail-slow fault tolerance needs programming support. In *Proceedings of the 18th Workshop on Hot Topics in Operating Systems (HotOS-XVIII)* (May 2021).

[119] Yoo, S., and Harman, M. Regression Testing Minimisation, Selection and Prioritization: A Survey. *Software Testing, Verification, and Reliability 22*, 2 (Mar. 2012), 67–120.

[120] Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M. 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)* (Oct. 2014).

[121] Yun, I., Min, C., Si, X., Jang, Y., Kim, T., and Naik, M. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)* (Aug. 2016).

[122] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)* (Apr. 2012).

[123] Zamfir, C., Altekar, G., and Stoica:, I. Automating the Debugging of Datacenter Applications with ADDA. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)* (June 2013).

[124] Zhai, E., Chen, A., Piskac, R., Balakrishnan, M., Tian, B., Song, B., and Zhang, H. Check before You Change: Preventing Correlated Failures in Service Updates. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)* (Feb. 2020).

[125] Zhang, J., Renganarayana, L., Zhang, X., Ge, N., Bala, V., Xu, T., and Zhou, Y. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)* (Mar. 2014).

[126] Zhang, Y., Yang, J., Jin, Z., Sethi, U., Rodrigues, K., Lu, S., and Yuan, D. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP'21)* (2021).

A Artifact Appendix

A.1 Abstract

The artifact can be found at https://github.com/xlab-uiuc/csi-ae and https://github.com/xlab-uiuc/csi-test-ae. The first repository contains the main dataset and the Jupyter notebook required to reproduce the key finding statistics and table data in Sections 1-7. The second repository contains the scripts required to reproduce the results in Section 8.

A.2 Description & Requirements

A.2.1 How to access. The artifact can be found at https://github.com/xlab-uiuc/csi-ae.

A.2.2 Hardware dependencies. None.

A.2.3 Software dependencies. The study artifact can be accessed on a browser through the provided Binder setup or run through an online or local Jupyter client.

The case study artifact runs on a Docker container, and has been tested on Ubuntu 18/20 and OS X. Installation instructions for Docker Engine are here: https://docs.docker.com/engine/install/. All other dependencies are installed in the Docker container.

A.2.4 Benchmarks. None.

A.3 Set-up

Clone the repository and submodules:

```
git clone --recursive
https://github.com/xlab-uiuc/csi-ae.git
```

A.3.1 Study Evaluation (C1, E1 below). The tables and reproduced findings can be viewed with this Jupyter notebook: https://github.com/xlab-uiuc/csi-ae/blob/main/reproduce_study.ipynb. Additionally, you can run the Jupyter notebooks here: https://mybinder.org/v2/gh/xlab-uiuc/csi-ae/HEAD. Navigating to this link would bring up an ephemeral Jupyter environment. Once the Jupyter environment is initialized, browse to `reproduce_study.ipynb` and execute it to reproduce all the tables and findings in the paper.

A.3.2 Case Study Experiment (C2, E2 below). The easiest way to run the experiments is to pull the Docker image from Docker Hub:

```
docker pull chaitanyabhandari/\
csi-eurosys23-ae:linux-amd64
```

or

```
docker pull chaitanyabhandari/\
csi-eurosys23-ae:linux-arm64-v8
```

depending on your architecture (`dpkg --print-architecture`).

You can also build the Docker image with the provided Dockerfile:

```
cd csi-test-ae
docker build -t csi-test-ae .
```

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1): The study dataset corroborates the 13 findings discussed in Sections 1-7. This is shown through the dataset and scripts in (E1).*
  1. *Among 55 cloud incidents, 11 (20%) were caused by CSI failures, showing their catastrophic consequences.*
  2. *Data- and management-plane interactions contribute to significant percentages of CSI failures: 51% of CSI failures in our dataset manifest at the data plane, and*

32% of CSI failures manifest at the management plane. Control-plane interactions contribute to 17%.

3. *Existing fault tolerance or recovery mechanisms are insufficient or ineffective in handling CSI failures, resulting in diverse failure impacts. Most (89/120) CSI failures are manifested through crashing behavior.*
4. *Discrepancies of data-plane CSI failures lie in many different data properties. The majority (50/61) of data-plane CSI failures are caused by metadata, namely typical metadata (42/61) such as addresses/names and data schemas, and custom metadata (8/61). The others (11/61) are caused by custom properties and API semantics.*
5. *Complicated data abstractions (e.g., tables) are more error-prone to CSI failures, compared with simple data abstractions. 57% (35/61) of data-plane CSI failures are induced by table-related operations. None are induced by key-value tuple operations.*
6. *25% (15/61) data-plane CSI failures are root-caused by data serialization.*
7. *CSI-failure-inducing configuration issues are very different from traditional configuration issues of individual systems. The former is mostly about failures of coherently configuring multiple involved systems, while the latter is mainly on correctness checking of erroneous configuration values.*
8. *Parameter-related configuration issues are the majority (21/30) of configuration-induced CSI failures. The rest (9/30) are in configuration components of the involved systems.*
9. *Monitoring-related CSIs are critical to reliability, especially when monitoring data is used for critical actions.*
10. *Most control-plane CSI failures are rooted in discrepancies of implicit properties, including implicit API semantics and state/resource inconsistencies.*
11. *API misuses, despite being a classic problem, are still common defects and contribute to the majority (13/20) of control-plane CSI failures. The main patterns are implicit semantic violation (8/13) and incorrect invocation context (5/13).*
12. *In 40% (46/115) CSI failures, the merged fixes improve condition checking and error handling instead of repairing the failed interactions.*
13. *In 69% (79/115) CSI failures, fixes were applied to code in the upstream system specific to interaction with a downstream system. Furthermore, among these 79 cases, fixes for 68 (86%) cases resided in dedicated "connector" modules.*

- *(C2): The case study uncovers discrepancies between the Spark and Hive data plane exposing the following problems. This is proven by the experiment in (E2).*
  - Cannot read what was written
  - Type violations
  - Exposing internal configurations of the downstream to the upstream
  - Inconsistent error behavior across interfaces
  - Relying on custom (non-default) configurations

A.4.2 Experiments.

Experiment (E1): Dataset (C1) [15-30 human-minutes]: Verify the data used to produce statistics for the paper.

[Preparation] The Jupyter notebook for the study data can be directly accessed here: https://mybinder.org/v2/gh/xlab-uiuc/csi-ae/HEAD

[Execution] The cells on `reproduce_study.ipynb` should already be executed in the provided binder, but can be re-executed using the run buttons on the top bar.

[Results] Note that any issue mentioned in the paper can be accessed through appending the issue id to https://issues.apache.org/jira/browse/, e.g. https://issues.apache.org/jira/browse/FLINK-12342.

For each finding and table in the paper, you can compare the data from the corresponding cells in the notebook to the statistics and tables in the paper. Note that Finding 9 does not appear in the notebook but can be validated qualitatively from Section 6.2.2 or from FLINK-887 which is discussed in that section and is represented in the study.

Experiment (E2): Case study (C2) [2-3 human hours + 2-3 compute-hours]: Performs the Spark-Hive data plane testing experiments.

[Preparation] Run the docker container.

```
docker run -it chaitanyabhandari/\
csi-eurosys23-ae:linux-amd64
```

or

```
docker run -it chaitanyabhandari/\
csi-eurosys23-ae:linux-arm64-v8
```

[Execution] In the prepared Docker container, run the following scripts. Each takes around 30-60 mins.

```
./spark_e2e.sh
./spark_hive_oneway.sh
./hive_spark_oneway.sh
```

[Results] The scripts should output results in `logs/<script_name>/<timestamp>`. The relevant files to inspect are the 2-3 `*failed.json` files in each experiment's directory which correspond to the test failures produced by each test oracle for each experiment. Each log specifies the input, read/write interface and data format combinations which failed the test.

- `difft`: Failed tests will have differing outputs between combinations of interface and data format.
- `wr`: Failed tests will have differing values between what is written and what is read. Inputs should be on valid data.

- `eh`: Failed tests show that the value was successfully inserted and read back. Inputs should be on invalid data.

The test failures indicated in the below list can be found in these files. The reported issues which are associated with the test failure are also listed. There will be many more test failures produced than the ones listed, but they correspond to the same discrepancies as those shown. For many inputs, multiple oracles will have failed tests.

1. `ss_difft` 0: SPARK-39075
2. `sh_difft` 68: SPARK-39158
3. `ss_difft` 6 (see `spark_e2e/log_w_sql_r_sql_avro` for matching warning): HIVE-26533, SPARK-40409
4. `ss_difft` 420: HIVE-26531
5. `ss_difft` 70: SPARK-40439
6. `sh_difft` 131: HIVE-26528
7. `sh_difft` 163: Same root cause as #6 but exhibits different behavior
8. `ss_difft` 69: SPARK-40616
9. `ss_difft` 8: SPARK-40525
10. `ss_difft` 71: SPARK-40624
11. `ss_difft` 14: Addressed with the same config as #10
12. `ss_difft` 48: SPARK-40629
13. `ss_difft` 86: Addressed with config `spark.sql.legacy.charVarcharAsString`
14. `ss_difft` 116: SPARK-40637
15. `ss_eh` 198 (see `ss_difft_all.json` 198 for input details): SPARK-40630

The following list maps the characteristics discussed in Section 8.2 of the submission to the aforementioned test failures.

- Cannot read what was written **(2/15)**: 1, 2
- Type violations **(2/15)**: 3, 8
- Exposing internal configurations of the downstream to the upstream **(5/15)**: 1, 2, 3, 4, 6
- Inconsistent error behavior across interfaces **(7/15)**: 1, 5, 9, 10, 11, 12, 13
- Relying on custom (non-default) configurations **(8/15)**: 5, 8, 9, 10, 11, 12, 13, 15