What Bugs Live in the Cloud?
A Study of 3000+ Issues in Cloud Systems

Haryadi S. Gunawi, Mingzhe Hao,
Tanakorn Leesatapornwongsa,
and Tiratat Patana-anake

University of Chicago[*]

Thanh Do[**]

University of
Wisconsin–Madison

Jeffry Adityatama, Kurnia J. Eliazar,
Agung Laksono, Jeffrey F. Lukman,
Vincentius Martin, and Anang D. Satria

Surya University

Abstract

*We conduct a comprehensive study of development and deployment issues of six popular and important cloud systems (Hadoop MapReduce, HDFS, HBase, Cassandra, ZooKeeper and Flume). From the bug repositories, we review in total 21,399 submitted issues within a three-year period (2011-2014). Among these issues, we perform a deep analysis of 3655 "vital" issues (i.e., real issues affecting deployments) with a set of detailed classifications. We name the product of our one-year study Cloud Bug Study database (*CBSDB*) [9], with which we derive numerous interesting insights unique to cloud systems. To the best of our knowledge, our work is the largest bug study for cloud systems to date.*

1 Introduction

1.1 Motivation

As the cloud computing era becomes more mature, various scalable distributed systems such as scale-out computing frameworks [18, 40], distributed key-value stores [14, 19], scalable file systems [23, 41], synchronization services [13], and cluster management services [31, 48] have become a dominant part of software infrastructure running behind cloud data centers. These "cloud systems" are in constant and rapid developments and many new cloud system architectures have emerged every year in the last decade. Unlike single-server systems, cloud systems are considerably more complex as they must deal with a wide range of distributed components, hardware failures, users, and deployment scenarios. It is not a surprise that cloud systems periodically experience downtimes [7]. There is much room for improving cloud systems dependability.

This paper was started with a simple question: why are cloud systems not 100% dependable? In our attempt to provide an intelligent answer, the question has led us to many more intricate questions. Why is it hard to develop a fully reliable cloud systems? What bugs "live" in cloud systems? How should we properly classify bugs in cloud systems? Are there new classes of bugs unique to cloud systems? What types of bugs can only be found in deployment? Why existing tools (unit tests, model checkers, etc.) cannot capture those bugs prior to deployment? And finally, how should cloud dependability tools evolve in the near future?

To address all the important questions above, cloud outage articles from headline news are far from sufficient in providing the necessary details. Fortunately, as open source movements grow immensely, many cloud systems are open sourced, and most importantly they come with publicly-accessible issue repositories that contain bug reports, patches, and deep discussions among the developers. This provides an "oasis" of insights that helps us address our questions above.

1.2 Cloud Bug Study

This paper presents our one-year study of development and deployment issues of six popular and important cloud systems: Hadoop MapReduce [3], Hadoop File System (HDFS) [6], HBase [4], Cassandra [1], ZooKeeper [5], and Flume [2]. Collectively, our target systems represent a diverse range of cloud architectures.

We select issues submitted over a period of three years (1/1/2011-1/1/2014) for a total of 21,399 issues across our target systems. We review each issue to filter "vital" issues

[*] All authors are listed in institution and alphabetical order.

[**] Thanh Do is now with Microsoft Jim Gray Systems Lab.

SOCC '14, November 03 - 05 2014, Seattle, WA, USA
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3252-1/14/11...$15.00
http://dx.doi.org/10.1145/2670979.2670986

from "miscellaneous" ones. The former represents real issues affecting deployed systems while the latter involves non-vital issues related to maintenance, code refactoring, unit tests, documentation, etc.. In this paper, we only present findings from vital issues. In total, there are 3655 vital issues that we carefully study.

For each vital issue, we analyze the patches and all the developer responses for the issue. We then categorize vital issues with several classifications. First, we categorize them by *aspect* such as reliability, performance, availability, security, data consistency, scalability, topology and QoS. Second, we introduce *hardware type* and *failure mode* labels for issues that involve hardware failures. All types of hardware, disks, network, memory and processors, can fail and they can fail in different ways (stop, corrupt, or "limp"). Next, we dissect vital issues by a wide range of *software bug types* such as error handling, optimization, configuration, data race, hang, space, load and logic bugs. Fourth, we also study the issues by *implication* such as failed operations, performance problems, component downtimes, data loss, staleness, and corruption. In addition to all of these, we also add *bug scope* labels to measure bug impacts (a single machine, multiple machines, or the whole cluster).

The product of our classifications is Cloud Bug Study DB (CBSDB), a set of classification text files, data mining scripts, and graph utilities [9]. In total, we have added 25,309 annotations for vital issues in CBSDB. The combination of cloud bug repositories and CBSDB enables us (and future CBSDB users) to perform in-depth quantitative and qualitative analysis of cloud issues.

1.3 Findings

From our extensive study, we derive the following important findings that provide interesting insights into cloud systems development and deployment.

• **"New bugs on the block":** When broken down by issue aspects (§3), classical aspects such as reliability (45%), performance (22%), and availability (16%) are the dominant categories. In addition to this, we find "new" classes of bugs unique to cloud systems: data consistency (5%), scalability (2%), and topology (1%) bugs. Cloud dependability tools should evolve to capture these new problems.

• **"Killer bugs":** From studying large-scale issues, we find what we call as "killer bugs" (*i.e.*, bugs that simultaneously affect multiple nodes or the entire cluster). Their existence (139 issues in our study) implies that cascades of failures happen in subtle ways and the no single point of failure principle is not always upheld (§4).

• **Hardware can fail, but handling is not easy:** "Hardware can fail, and reliability should come from the software" is preached extensively within the cloud community, but we still find 13% of the issues are caused by hardware failures. The complexity comes from various causes: hardware can fail in different ways (stop, corrupt, or "limp"), hardware can fail at *any* time (*e.g.*, during a complex operation), and recovery itself can see another failure. Dealing with hardware failures at the software level remains a challenge (§5).

• **Vexing software bugs:** Cloud systems face a variety of software bugs: logic-specific (29%), error handling (18%), optimization (15%), configuration (14%), data race (12%), hang (4%), space (4%) and load (4%) issues. Exacerbating the problem is the fact that each bug type can lead to almost all kinds of implication such as failed operations, performance degradation, component downtimes, data loss, staleness, and corruption (§6).

• **Availability first, correctness second:** From our study, we make a conclusion that cloud systems favor availability over correctness (*e.g.*, reliability, data consistency). For example, we find cases where data inconsistencies, corruptions, or low-level failures are detected, but ignored so that the system can continue running. Although this could be dangerous because the future ramifications are unknown, perhaps running with incorrectness "looks better" than downtimes; users often review systems based on clear performance and availability metrics (*e.g.*, throughput, 99.9% availability) but not on undefined metrics (*e.g.*, hard-to-quantify correctness).

• **The need for multi-dimensional dependability tools:** As each kind of bugs can lead to many implications and vice versa (§5, §6), bug-finding tools should not be one dimensional. For example, a tool that captures error-handling mistakes based on failed operations is an incomplete tool; error-handling bugs can cause all kinds of implications (§6.1). Likewise, if a system attempts to ensure full dependability on just one axis (*e.g.*, no data loss), the system must deploy all bug-finding tools that can catch all hardware and software problems (§7). Furthermore, we find that many bugs are caused because of not only one but multiple types of problems. A prime example is distributed data races in conjunction with failures (§6.3); the consequence is that data-race detectors must include fault injections.

In summary, we perform a large-scale bug study of cloud systems and uncover interesting findings that we will present throughout the paper. We make CBSDB publicly available [9] to benefit the cloud community. Finally, to the best of our knowledge, we perform the largest bug study for cloud systems to date.

In the following sections, we first describe our methodology (§2), then present our findings based on aspects (§3), bug scopes (§4), hardware problems (§5), software bugs (§6) and implications (§7). We then demonstrate other use cases of CBSDB(§8), discuss related work (§9) and conclude (§10).

2 Methodology

In this section, we describe our methodology, specifically our choice of target systems, the base issue repositories, our issue classifications and the resulting database.

• **Target Systems:** To perform an interesting cloud bug study paper, we select six popular and important cloud systems that represent a diverse set of system architectures: Hadoop MapReduce [3] representing distributed computing frameworks, Hadoop File System (HDFS) [6] representing scalable storage systems, HBase [4] and Cassandra [1] representing distributed key-value stores (also known as NoSQL systems), ZooKeeper [5] representing synchronization services, and finally Flume [2] representing streaming systems. These systems are referred with different names (*e.g.*, data-center operating systems, IaaS/SaaS). For simplicity, we refer them as cloud systems.

• **Issue Repositories:** The development projects of our target systems are all hosted under Apache Software Foundation Projects [8] wherein each of them maintains a highly organized issue repository.[1] Each repository contains development and deployment issues submitted mostly by the developers or sometimes by a larger user community. The term “issue” is used here to represent both bugs and new features.

For every issue, the repository stores many “raw” labels, among which we find useful are: issue date, time to resolve (in days), bug priority level, patch availability, and number of developer responses. We download raw labels automatically using the provided web API. Regarding the *#responses* label, each issue contains developer responses that provide a wealth of information for understanding the issue; a complex issue or hard-to-find bug typically has a long discussion. Regarding the *bug priority* label, there are five priorities: trivial, minor, major, critical, and blocker. For simplicity, we label the first two as “minor” and the last three as “major”. Although we analyze all issues in our work, we only focus on major issues in this paper.

• **Issue Classifications:** To perform a meaningful study of cloud issues, we introduce several issue classifications as displayed in Table 1. The first classification that we perform is based on *issue type* (“miscellaneous” vs. “vital”). Vital issues pertain to system development and deployment problems that are marked with a major priority. Miscellaneous issues represent non-vital issues (*e.g.*, code maintenance, refactoring, unit tests, documentation). Real bugs that are easy to fix (*e.g.*, few line fix) tend to be labeled as a minor issue and hence are also marked as miscellaneous by us. We had to manually add our own issue-type classification because the major/minor raw labels do not suffice; many miscellaneous issues are also marked as “major” by the developers.

Classification	Labels
Issue Type	Vital, miscellaneous.
Aspect	Reliability, performance, availability, security, consistency, scalability, topology, QoS.
Bug scope	Single machine, multiple machines, entire cluster.
Hardware	Core/processor, disk, memory, network, node.
HW Failure	Corrupt, limp, stop.
Software	Logic, error handling, optimization, config, race, hang, space, load.
Implication	Failed operation, performance, component downtime, data loss, data staleness, data corruption.

Per-component Labels
Cassandra: Anti-entropy, boot, client, commit log, compaction, cross system, get, gossiper, hinted handoff, IO, memtable, migration, mutate, partitioner, snitch, sstable, streaming, tombstone.
Flume: Channel, collector, config provider, cross system, master/supervisor, sink, source.
HBase: Boot, client, commit log, compaction, coprocessor, cross system, fsck, IPC, master, memstore flush, namespace, read, region splitting, log splitting, region server, snapshot, write.
HDFS: Boot, client, datanode, fsck, HA, journaling, namenode, gateway, read, replication, ipc, snapshot, write.
MapReduce: AM, client, commit, history server, ipc, job tracker, log, map, NM, reduce, RM, security, shuffle, scheduler, speculative execution, task tracker.
Zookeeper: Atomic broadcast, client, leader election, snapshot.

Table 1: Issue Classifications and Component Labels.

We carefully read each issue (the discussion, patches, etc.) to decide whether the issue is vital. If an issue is vital we proceed with further classifications, otherwise it is labeled as miscellaneous and skipped in our study. This paper only presents findings from vital issues.

For every vital issue, we introduce *aspect* labels (more in §3). If the issue involves hardware problems, then we add information about the *hardware type* and *failure mode* (§5). Next, we pinpoint the *software bug types* (§6). Finally, we add *implication* labels (§7). As a note, an issue can have multiple aspect, hardware, software, and implication labels. Interestingly, we find that a bug can simultaneously affects multiple machines or even the entire cluster. For this purpose, we use *bug scope* labels (§4). In addition to generic classifications, we also add per-component labels to mark where the bugs live; this enables more interesting analysis (§8).

• **Cloud Bug Study DB (CBSDB):** The product of our classifications is stored in CBSDB, a set of raw text files, data mining scripts and graph utilities [9], which enables us (and future CBSDB users) to perform both quantitative and qualitative analysis of cloud issues.

As shown in Figure 1a, CBSDB contains a total of 21,399 issues submitted over a period of three years (1/1/2011-1/1/2014) which we analyze one by one. The majority of the issues are miscellaneous issues (83% on average across

[1] Hadoop MapReduce in particular has two repositories (Hadoop and MapReduce). The first one contains mostly development infrastructure (*e.g.*, UI, library) while the second one contains system issues. We use the latter.

the six systems). We then carefully annotate the vital issues (3655 in total) using our complete issue classifications. In total, we have added 25,309 labels for vital issues in CbsDB.

• **Threats to validity:** To improve the validity of our classifications, each issue is reviewed in at least two passes. If an ambiguity arises when we tag an issue, we discuss the ambiguity until we reach a unified conclusion. Each issue cited in this paper has been discussed by 3–4 people. Although these actions are by no means complete, we believe they help improving the accuracy of CbsDB significantly.

• **In-paper presentation:** The following sections are organized by issue classifications (Table 1). All the bugs discussed and graphs shown only come from vital issues. For each classification, we present both quantitative and qualitative findings and also introduce further sub-classifications. For each sub-classification, we cite some interesting issues as footnotes (*e.g.*, m2345). The footnotes contain hyperlinks; interested readers can click the links to read more discussions by the developers.

In the rest of this paper, "bugs/issues" imply vital issues, "the system(s)" implies a general reference to our target systems, "main protocols" imply user-facing operations (*e.g.*, read/write) and "operational protocols" imply non-main protocols such as background daemons (*e.g.*, gossiper) and administrative protocols (*e.g.*, node decommissioning). We also use several abbreviations.[1]

3 Issue Aspects

The first classification that we perform is by aspect. Figure 1b shows the distribution of the eight aspects listed in Table 1. Below we discuss each aspect; for interesting discussions, we focus on aspects such as data consistency, scalability, topology and QoS.[2]

3.1 Reliability, Performance, and Availability Aspects

Reliability (45%), performance (22%) and availability (16%) are the three largest categories of aspect. Since they represent classical system problems, we will weave interesting examples in later sections when we discuss killer bugs, hardware and software problems.

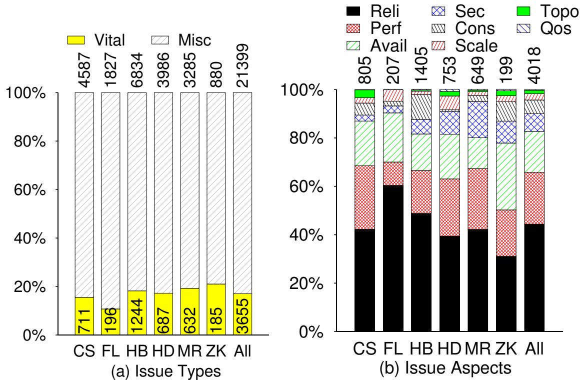


Figure 1: Issue Type and Aspect. *Figures (a) and (b) show the classification of issue types and aspects respectively. An issue can have multiple aspects.*

3.2 Data Consistency Aspect

Data consistency means that all nodes or replicas agree on the same value of a data (or eventually agree in the context of eventual consistency). In reality, there are several cases (5%) where data consistency is violated and users get stale data or the system's behavior becomes erratic. The root causes mainly come from logic bugs in operational protocols (43%), data races (29%) and failure handling problems (10%).[1] Below we expand these problems.

[a] **Buggy logic in operational protocols:** Besides the main read/write protocols, many other operational protocols (*e.g.*, bootstrap, cloning, fsck) touch and modify data, and bugs within them can cause data inconsistency. For example, when bootstrapping a node, Cassandra should fetch the necessary number of key-value replicas from multiple neighbors depending on the per-key consistency level, but a protocol bug fetches only one replica from the nearest neighbor. In another protocol, cross-DC synchronization, the compression algorithm fails to compress some key-values, catches the error, but allows the whole operation to proceed, silently leaving the two DCs with inconsistent views after the protocol "finishes". In HBase, due to connection problems during the cloning operation, metadata is cloned but the data is not. Beyond these examples, there are many other implementation bugs that make cloud systems treat deleted data as valid ones, miss some transactions logs, or forget to wipe out memoized values in some functions, all causing data staleness.

[b] **Concurrency bugs and node failures**: Intra-node data races are a major culprit of data inconsistency. As an example, data races between read and write operations in updating the cache can lead to older values written to the cache. Inter-node data races (§6.3) are also a major root cause; com-

[1] SPoF: single point of failure; OOM: out-of-memory; GC: garbage collection; WAL: write-ahead logging; DC: data center; AM: application master; NM: node manager; RM: resource manager; HA: high availability; CS: Cassandra; FL: Flume; HB: HBase; HD: HDFS; MR: MapReduce; ZK: ZooKeeper.

[2] We skip the discussion of security aspect as we lack expertise in that area. Interested readers can study security problems by downloading CbsDB [9] for further analysis.

[1] The correlation graphs between aspects and hardware/software problems are not shown due to space constraints.

§3.2: [a]c2434, c5391, hb6359, hb7352, m4342; [b]c3862, z1549.

plex re-ordering of asynchronous messages combined with node failures make systems enter incorrect states (*e.g.*, in ZooKeeper, committed transactions in different nodes have different views).

Summary: Operational protocols modify data replicas, but they tend to be less tested than the main protocols, and thus often carry data inconsistency bugs. We also find an interesting scenario: when cloud systems detect data inconsistencies (via some assertions), they often decide to continue running even with incorrectness and potential catastrophes in the future. Availability seems to be more important than consistency; perhaps, a downtime "looks worse" than running with incorrectness.

3.3 Scalability Aspect

Scalability aspect accounts for 2% of cloud issues. Although the number is small, scalability issues are interesting because they are hard to find in small-scale testing. Within this category, software optimization (29%), space (21%) and load (21%) problems are the dominant root causes. In terms of implications, performance problems (52%), component downtimes (27%), and failed operations (16%) are dominant. To understand deeper the root problems, we categorize scalability issues into four axes of scale: cluster size, data size, load, and failure.

[a] **Scale of cluster size:** Protocol algorithms must anticipate different cluster sizes, but algorithms can be quadratic or cubic with respect to the number of nodes. For example, in Cassandra, when a node changes its ring position, other affected nodes must perform a key-range recalculation with a complexity $\Omega(n^3)$. If the cluster has 100-300 nodes, this causes CPU "explosion" and eventually leads to nodes "flapping" (*i.e.*, live nodes are extremely busy and considered dead) and requires whole-cluster restart with manual tuning.

Elasticity is often not tested thoroughly; system administrators assume they can decommission/recommission any number of nodes, but this can be fatal. As an example, In HDFS, when a large number of nodes are decommissioned and recommissioned, a significant amount of extra load (millions of migrated blocks and log messages) bogs down critical services.

[b] **Scale of data size:** In the Big Data era, cloud systems must anticipate large data sizes, but it is often unclear what the limit is. For instance, in HBase, opening a big table with more than 100K regions undesirably takes tens of minutes due to an inefficient table look-up operation. In HDFS, the boot protocol does not scale with respect to the number of commit log segments, blocks (potentially millions), and inter-node reboot messages; whole-cluster reboot can take tens of minutes. Similarly in HBase, the log cleanup process is $O(n^2)$ with respect to the number of log files; this slow process causes HBase to fall behind in serving incoming requests. We also find cases where users submit queries on large data sets that lead to OOM at the server side; here, users must break large queries into smaller pieces (§6.5).

[c] **Scale of request load:** Cloud systems sometimes cannot serve large request loads of various kinds. For example, some HDFS users create thousands of small files in parallel causing OOM; HDFS does not expect this because it targets big files. In Cassandra, users can generate a storm of deletions that can block other important requests. In HDFS, when a job writing to 100,000 files is killed, HDFS experiences a burst of lease recovery that causes other important RPC calls such as lease renewals and heartbeats to be dropped. We also find small problems such as small leaks that can become significant during high load (§6.6).

[d] **Scale of failure:** At scale, a large number of components can fail at the same time, but recovery is often unprepared. For example, in MapReduce, recovering 16,000 failed mappers (if AM goes down) takes more than 7 hours because of an unoptimized communication to HDFS. In another case, when a large number of reducers report fetch failures, tasks are not relaunched until 2 hours (because of a threshold error). Also, an expensive $O(n^3)$ recovery (a triple for-loop processing) is magnified when MapReduce experiences thousands of task failures. In ZooKeeper, when 1000 clients simultaneously disconnect due to network failures, a session-close stampede causes other live clients to be disconnected due to delays in heartbeat responses.

Summary: Scalability problems surface late in deployment, and this is undesirable because users are affected. More research is needed to unearth scalability problems prior to deployment. We also find that main read/write protocols tend to be robust as they are implicitly tested all the time by live user workloads. On the other hand, operational protocols (recovery, boot, etc.) often carry scalability bugs. Therefore, operational protocols need to be tested frequently at scale (*e.g.*, with "live drills" [12, 32]). Generic solutions such as loose coupling, decentralization, batching and throttling are popular but sometimes they are not enough; some problems require fixing domain-specific algorithms.

3.4 Topology Aspect

In several cases (1%), protocols of cloud systems do not work properly on some network topology. We call this topology bugs; they are also intriguing as they are typically unseen in pre-deployment. Below we describe three matters that pertain to topology bugs.

§3.3: [a]hd4075, c3881, c6127; [b]c4415, hd2982, hb8778, hb9208; [c]c5456, hb4150, hd4479, hd5364; [d]m3711, m4772, m5043, z1049.

[a] **Cross-DC awareness:** Recently, geo-distributed systems have gained popularity [37, 47, 56]. In such settings, communication latency is higher and thus asynchrony is a fundamental attribute. However, some protocols are still synchronous in nature (*e.g.*, Cassandra hint delivery can take one day and stall a cross-DC Cassandra deployment). We also find various logic problems related to cross-DC. For example, in HBase, two DCs ping-pong replication requests infinitely and in Cassandra, time-dependent operations fail because of time drift between two DCs.

[b] **Rack awareness:** We find cases where operational protocols such as recovery are not rack aware. For example, when a mapper and a reducer run in separate racks with a flaky connection, MapReduce always judges that the mapper node (not the network) is the problem and hence blacklisted. MapReduce then re-runs the mapper in the same rack, and eventually all nodes in the rack are incorrectly blacklisted; a better recovery is to precisely identify cross-rack network problems. In an HDFS two-rack scenario, if the namenode and datanode-1 are in rack-A, a client and datanode-2 are in rack B, and the cross-rack network is saturated, the namenode will de-prioritize datanode-2, and thus forcing the client to connect via datanode-1 although in such a topology the communication between client and datanode-2 in rack B is more optimum.

[c] **New layering architecture:** As cloud systems mature, their architectures evolve (*e.g.*, virtual nodes are added in Cassandra and node groups in HDFS). These architectural changes are not always followed with proper changes in the affected protocols. For example, in Cassandra, with virtual nodes (vnode), the cluster topology and scale suddenly change (256 vnodes/machine is the default), but many Cassandra protocols still assume physical-node topology which leads to many scalability problems (*e.g.*, gossip protocol cannot deal with orders of magnitude increase in gossip messages). In HDFS, node groups lead to many changes in protocols such as failover, balancing, replication and migration.

Summary: Users expect cloud systems to run properly on many different topologies (*i.e.*, different number of nodes, racks, and datacenters, with different network conditions). Topology-related testing and verification are still minimal and should be a focus of future cloud dependability tools. Another emphasis is that changes in topology-related architectures are often not followed with direct changes in the affected protocols.

3.5 QoS Aspect

QoS is a fundamental requirement for multi-tenant systems [46, 49]. QoS problems in our study are relatively small (1%), however it should be viewed as an unsolved problem as opposed to a non-problem. Below, we highlight the two main QoS discussion points in our study.

[a] **Horizontal/intra-system QoS:** There are many issues about heavy operations affecting other operations that the developers can quickly fix with classic techniques such as admission control, prioritization and throttling. However, care must be taken when introducing thresholds in one protocol as it can negatively impact other protocols. For example, in HDFS, throttling at the streaming level causes requests to queue up at the RPC layer.

[b] **Vertical/cross-system QoS:** Herein lies the biggest challenge of QoS; developers can only control their own system but stackable cloud systems demand for an end-to-end QoS. For instance, HBase developers raise questions about how QoS at HBase will be translated at the HDFS or MapReduce layer. Also, some techniques such as throttling are hard to enforce all the way to the OS level (*e.g.*, disk QoS involve many metrics such as bandwidth, IOPS, or latency). There are also discussions about the impact of a lack of control to QoS accuracy (*e.g.*, it is hard to guarantee QoS when the developers cannot control Java GC).

Summary: Horizontal QoS is easier to guarantee as it only involves one system. Cloud systems must check that QoS enforcement in one protocol does not negatively impact others. Vertical QoS seems far from reach; developers do not have an end-to-end control and not many cross-system QoS abstractions have been proposed [46]. Increasing the challenge is the fact that open-source cloud systems were not born with QoS in mind and most likely must change radically.

4 "Killer Bugs"

By studying issues of large-scale systems, we have the opportunity to study what we call as "killer bugs", that is, bugs that simultaneously affect multiple nodes or even the entire cluster (Figure 2). This particular study is important because although the "no-SPoF" principle has been preached extensively within the cloud community, our study reveals that SPoF still exists in many forms, which we present below.

[a] **Positive feedback loop:** This is the case where failures happen, then recovery starts, but the recovery introduces more load and hence more failures [29, 32]. For example, in Cassandra, gossip traffic can increase significantly at scale caus-

§3.4: [a]c3577, c4761, c5179, hb7709; [b]c5424, hd3703, m1800; [c]c6127, hd3495, hd4240, hd5168.

§3.5: [a]hb9501, hd4412, hd5639, m1783; [b]c4705, hb4441, hd5499.

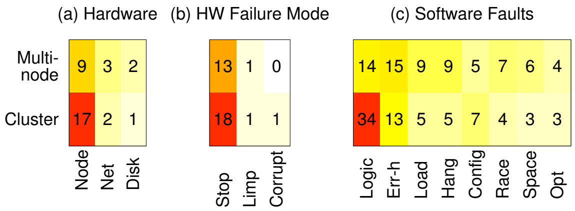


Figure 2: Killer bugs. *The figure shows heat maps of correlation between scope of killer bugs (multiple nodes or whole cluster) and hardware/software root causes. A killer bug can be caused by multiple root causes. The number in each cell represents the bug count.*

ing the cluster unstable for hours. As live nodes are incorrectly declared dead, administrators or elasticity tools might add more nodes to the cluster, which then causes more gossip traffic. Cloud systems should better identify positive feedback loops.

[b] **Buggy failover:** A key to no-SPoF is to detect failure and perform a failover. But, such guarantee breaks if the failover code itself is buggy. For example, in HBase, when a failover of metadata region server goes wrong, the whole cluster ceases to work because the whole-cluster metadata (META and ROOT tables) are not accessible; interestingly, we see this issue repeats four times in our study. Similarly in HA-HDFS, when a failover to a standby namenode breaks, all datanodes become unreachable. Our deeper analysis reveals that bugs in failover code surface when failover experiences another failure. Put simply, failover in failover is brittle [25]. A buggy failover is a killer bug when the affected components are a SPoF (*e.g.*, master node, metadata tables).

[c] **Repeated bugs after failover:** Another key to no-SPoF is that after a successful failover, the system should be able to resume the previously failed operation. This is true if the cause was a machine failure, but not true for a software bug. In other words, if after a failover the system must run the same buggy logic again, then the whole process will repeat and the entire cluster will eventually die. For example, In HBase, when a region server dies due to a bad handling of corrupt region files, HBase will failover to another live region server that will run the *same* code and will also die. As this repeats, all region servers go offline. We see this issue repeated three times in our study. Similar issues happen when a region server encounters different kinds of problems such as log-cleaning exception. To reduce the severity of these killer bugs, cloud systems must distinguish between hardware failures and software logic bugs. In the latter case, it is better to stop the failover rather than killing the entire cluster.

[d] **A small window of SPoF:** Another key to no-SPoF is ensuring failover ready all the time. We find few interesting cases where failover mechanisms are disabled briefly for some operational tasks. For example, in ZooKeeper, during dynamic cluster reconfiguration, heartbeat monitoring is disabled, and if the leader hangs at this point, a new leader cannot be elected.

[e] **Buggy start-up code:** Starting up a large-scale system is typically a complex operation, and if the start-up code fails then all the machines are unusable. For example, ZooKeeper leader election protocol is bug prone and can cause no leader to be elected; without a leader, ZooKeeper cluster cannot work. Similarly, we find issues with HDFS namenode start-up protocol.

[f] **Distributed deadlock:** Our study also unearths interesting cases of distributed deadlock where each node is waiting for other nodes to progress. For example, during start-up in Cassandra, it is possible that all nodes never enter a normal state as they keep gossiping. This coordination deadlock also happens in other Cassandra protocols such as migration and compaction and is typically caused by message re-orderings, network failures or software bugs. Distributed deadlock can also be caused by "silent hangs" (more in §6.4). For example, in HDFS, a disk hang on one node causes the whole write pipeline to hang. As catching local deadlock is challenging [51], it is more so for distributed deadlock.

[g] **Scalability and QoS bugs:** Examples presented in Sections 3.3 and 3.5 highlight that scalability and QoS bugs can also affect the entire cluster.

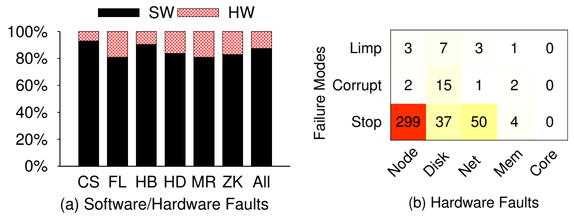
Summary: The concept of no-SPoF is not just about a simple failover. Our study reveals many forms of killer bugs that can cripple an entire cluster (potentially hundreds or thousands of nodes). Killer bugs should receive more attention in both offline testing and online failure management.

5 Hardware Issues

Figure 3a shows the percentage of issues that involve hardware failures. We perform this particular study because we are interested to know fundamental reasons why cloud systems cannot always deal with well-known hardware failure modes (Figure 3b).

[a] **Fail-stop:** Cloud systems are equipped with various mechanisms to handle fail-stop failures. There are several reasons why fail-stop recovery is not trivial. First, interleaving events and failures (*e.g.*, node up and down, message reordering) can force the system to enter unexpected states (more in §6.3). The use of coordination services (*e.g.*, ZooKeeper usage in HBase) does not simplify the problem (*e.g.*, many cross-system issues; more in §8). As mentioned before, a se-

§4: [a]c3831; [b]hb3446, hd4455; [c]hb3664, hb9737; [d]z1699; [e]hd2086, z1005; [f]c3832, c5244, hd5016; [g]c6127, m2214, z1049.



	Node	Disk	Net	Mem	Core
Limp	3	7	3	1	0
Corrupt	2	15	1	2	0
Stop	299	37	50	4	0

Figure 3: Hardware Faults. *Figure (a) shows the distribution of software (87%) and hardware (13%) faults. Figure (b) shows the heat map of correlation between hardware type and failure mode. The number in each cell is a bug count. If we see a component dies without any explanation, we assume it is a node failure. We did not find any report of CPU failure (perhaps because CPU failure translates to node failure).*

ries of multiple failures is often not handled properly and massive failures can happen (§3.3, §4).

[b] **Corruption:** It is widely known that hardware can corrupt data [11, 45] and thus end-to-end checksums are deployed. However, checksums are not a panacea. We find cases where detection is correct but not the recovery (*e.g.*, HDFS recovery accidentally removes the healthy copies and retains the corrupted ones). Software bugs can also make all copies of data corrupted, making end-to-end checksums irrelevant. We also find an interesting case where a bad hardware generates false alarms, causing healthy files marked as corrupted and triggering unnecessary large data recovery.

[c] **Limp Mode:** A good hardware can become a “limpware” [20]. We strengthen the case that cloud systems are not ready in dealing with limpware. For example, HDFS assumes disk I/Os will eventually finish, but when the disk degrades, a quorum of namenodes can hang. In an HBase deployment, developers observed a memory card that runs only at 25% of normal speed, causing backlogs, OOM, and crashes.

Summary: As hardware can fail, “reliability must come from software” [17]. Unfortunately, dealing with hardware failures is not trivial. Often, cloud systems focus on “what” can fail but not so much on the scale (§3.3) or the “when” (*e.g.*, subsequent failures during recovery). Beyond the fail-stop model, cloud systems must also address other failure modes such as corruption and limpware [16, 20, 21]. Reliability from software becomes more challenging.

6 Software Issues

We now discuss various kinds of software bugs that we find in cloud systems along with their implications. Figures 4a and 4b show the distribution of software bug types and bug implications respectively, and Figure 5 shows their correlation. Domain-specific bugs that cannot be classified into a general bug type are marked as “logic” bugs. Below, we discuss general bug types; we skip the discussion of logic and optimization bugs.

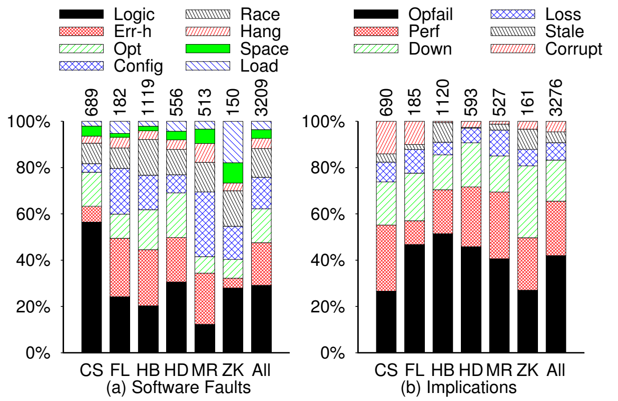


Figure 4: Software Faults and Implications. *Figures (a) and (b) show the breakdown of different software faults and implications respectively. For few vital issues that are about new features (i.e., a non-bug) without clear root causes and implications, no corresponding labels are added.*

6.1 Error Handling

Both hardware and software can fail, and thus error-handling code is a necessity. Unfortunately, it is a general knowledge that error-handling code introduces complexity and is prone to bugs [26, 55]. In our study, error-handling bugs are the 2nd largest category (18%) after logic bugs and can lead to all kinds of implications (Figure 5). Below, we break down error-handling bugs into three classes of problems.

[a] **Error/failure detection:** Before failures can be handled, they must be first detected, but detection is not always perfect. First, errors are often ignored (*e.g.*, ignored Java exceptions). Second, errors are often incorrectly detected. For example, in an HDFS protocol, network failure is considered as disk failure which can trigger a flurry of check-disk storm. In HDFS write pipeline, which node is problematic is hard to pinpoint. In MapReduce, a bad network between a mapper and a reducer is wrongly marked as the map node’s fault.

Failure detection in asynchronous protocols depends on timeouts. We find a large number of timeout issues. If timeout is too short, it can cause false positives (*e.g.*, nodes under high load are considered dead leading to an unnecessary replication storm). If timeout is too long, some operations appear to hang (§6.4). A more complex problem is cross-system timeouts. For example, a node is declared dead af-

§5: [a]c6531, hb9721, hd5438, m5489, z1294; [b]hd1371, hd2290, hd3004, hd3874, z1453; [c]hb3813, hd1595, hd3885, hd4859, hd5032.

§6.1: [a]hd3703, hd3874, hd3875, hd4581, hd4699, m1800; [b]hb4177, hd3703, hd4721, z1100; [c]c6364, hb4397, hd4239, m3993.

ter 10 minutes in HDFS and 30 seconds in HBase; this discrepancy causes problems for HBase. Here, one solution is to have multiple layers share the same failure detector [34], but this might be hard to realize across different groups of software developers.

[b] **Error propagation:** After an error is detected, the error might need to be propagated to and handled by upper layers. Interesting error-propagation problems arise in layered systems. For example, for some operations, HBase relies on HDFS to detect low-level errors and notify HBase. But when HDFS does not do so, HBase hangs. This example represents one major reason behind many silent failures across systems and components that we observe in our study.

[c] **Error handing:** After error information is propagated to the right layer or component, proper failure handling must happen. But, we find that failure handling can be "confused" (*e.g.*, not knowing what to do as the system enters some corner-case states), non-optimum (*e.g.*, not topology aware), coarse-grained (*e.g.*, a disk failure causes a whole node with 12 disks to be decommissioned), and best effort but not 100% correct (*e.g.*, in order to postpone downtime, Cassandra node will stay alive if data disk is dead and commit disk is alive).

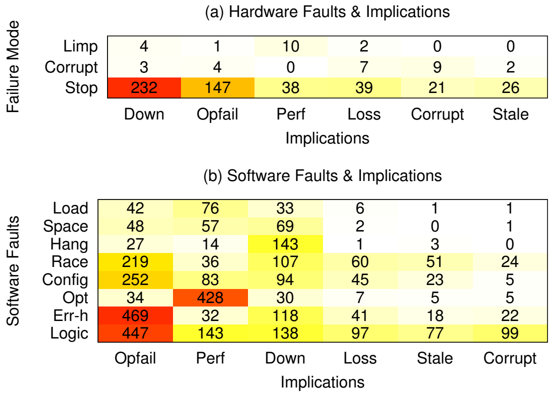
Summary: Dealing with errors involves correct error detection, propagation, and handling, but bugs can appear in each stage. We just touch few cases above, but with over 500 error-handling issues we tag in CBSDB, research community in dependability can analyze the issues deeper. Simple testing of error handling can uncover many flaws [55]. Overall, we find cloud systems code lacks of specifications of what error handling should do, but in most discussions the developers know what the code ideally should perform. This "specification gap" between systems code and developers needs to be narrowed.

6.2 Configuration

Configuration issues are the 4th largest category (14%) and recently become a hot topic of research [10, 30, 42, 52, 54]. Below we discuss two interesting types of configuration problems we find in our study.

[a] **Wrong configuration:** Setting configurations can be a tricky process. We find cases such as users providing wrong inputs, accidental deletion of configuration settings during upgrade, backward-compatibility issues when cloud systems move from static to dynamic online configurations, OS-compatibility issues, and incorrect values not appropriate for some workload.

Interestingly, configuration problems can lead to data loss and corruption (Figure 5). In MapReduce, users can accidentally set the same working directory for multiple jobs without warning from the system. Reducers' intermediate files can collide during merge because output files are not unique. In Flume, a simple mistake in line length limitation can cause corrupt messages.



(a) Hardware Faults & Implications

Failure Mode	Down	Opfail	Perf	Loss	Corrupt	Stale
Limp	4	1	10	2	0	0
Corrupt	3	4	0	7	9	2
Stop	232	147	38	39	21	26

Implications

(b) Software Faults & Implications

Software Faults	Opfail	Perf	Down	Loss	Stale	Corrupt
Load	42	76	33	6	1	1
Space	48	57	69	2	0	1
Hang	27	14	143	1	3	0
Race	219	36	107	60	51	24
Config	252	83	94	45	23	5
Opt	34	428	30	7	5	5
Err-h	469	32	118	41	18	22
Logic	447	143	138	97	77	99

Implications

Figure 5: Software/Hardware Faults & Implications. *The two figures represent heat maps of correlation between implications and hardware (and software) faults.*

[b] **Multiple configurations:** As users ask for more control, more configuration parameters are added over time, which by implication leads to more complexity. Not understanding the interaction of multiple parameters can be fatal. For example, an HBase administrator disables the WAL option and assumes in-memory data will be flushed to the disk periodically (but HBase does not do so when WAL is disabled). After learning that the HBase users lose the entire 2 weeks of data, HBase developers add more guards to prevent the case to happen again.

Summary: Configuration has become a significant deployment problem. Users, administrators, and even developers sometimes do not have a full understanding of how all the parameters interplay. They wish that the system gives them feedback about which configuration parameters matter in different deployments. We only touch few cases above, but we hope that the cloud community in configuration research can further analyze hundreds of configuration issues in CBSDB.

6.3 Data Races

Data races are a fundamental problem in any concurrent software systems and a major research topic in the last decade [39]. In our study, data races account for 12% of software bugs. Unlike non-distributed software, cloud systems are subject to not only local concurrency bugs (*e.g.*, due to thread interleaving) but also distributed concurrency bugs (*e.g.*, due to reordering of asynchronous messages). We find that many distributed data races surface when failures happen, such as the one in Figure 6. More examples can be found in CBSDB.[a] In our recent work [33], we elaborate this

§6.2: [a]f533, m5211, m5367; [b]hb5349, hb5450, hb5930.

m4819 **(1)** RM assigns an application to AM, **(2)** AM sees the completion of the application, **(3)** AM notifies Client and RM that the application completes, **(4)** AM crashes *before* RM receives the notification and unregisters the AM, **(5)** RM sees AM is down, and since it did not receive the notification, it re-runs another AM that writes the same files as previous AM **(6)** then *before* new AM finishes, client receives the notification and starts to consume the output, thus getting partial output ("corrupt" result).

Figure 6: "Distributed" data races. *The sequence of operations above represents a data race bug in distributed context due to concurrent messages.*

whole issue in great detail and present one solution towards the problem.

Summary: Numerous efforts in solving local concurrency bugs have been published in hundreds of papers. Unfortunately, distributed concurrency bugs have not received the same amount of attention. Yet, in our study, we find that distributed data races account for more than 50% of data race bugs. The developers see this as a vexing problem; an HBase developer wrote "do we have to rethink this entire [system]? There isn't a week going by without some new bugs about races between [several protocols]." Model checkers targeted for distributed system [28, 33, 53] are one solution, however we believe a larger and broader research is needed in this space (*e.g.*, how to extend numerous bug-finding techniques for local concurrency bugs for distributed context? how to reproduce them? [35]).

6.4 Hang

This section presents our findings behind hang-related issues which account for 4% of software bugs in our target systems. In software engineering, hang is usually attributed to deadlock [50], however, cloud developers have a broader meaning of hang. In particular, an operation that is supposedly short but takes a significantly long time to finish is considered hanging.

[a] **Silent hangs:** Timeouts are usually deployed around operations that can hang (*e.g.*, network operation). However, due to the complexity of large code base, some system components that can hang are sometimes overlooked (*i.e.*, timeoutless). For example, MapReduce developers did not deploy timeout for hanging AMs, but then they realized that AM is highly complex and can hang for many reasons such as RM exceptions, data races, failing tasks, and OOM.

We also find few cases of "false heartbeats". A node typically has a heartbeat thread and several worker threads. A buggy heartbeat thread can keep reporting good heartbeats even though some worker threads are hanging, and hence the silent hang. This hints that the relationship between heartbeat and actual progress can have a loophole.

[b] **Overlooked limp mode:** As mentioned before, hardware can exhibit a limp mode (§5). Although in many cases developers anticipate this (*e.g.*, network slowdown) and deploy timeouts (mainly around main protocols), there are some protocols that are overlooked. One prime example is job resource localization in MapReduce which is not regulated under speculative execution. Here, when a job downloads large JAR files over a degraded network, there is no timeout and failover. Similar stories can be found in other systems.

[c] **Unserved tasks:** Hang can also be defined as a situation where nodes are alive, but tasks stay in the queue forever and never get executed. We find this type of issue several times, mostly caused by deep data race and logic bugs that lead to corner-case states that prevent the system to perform a failover. For example, in MapReduce, tasks can get stuck in a "FailContainerCleanUp" stage that prevents the tasks to be relaunched elsewhere.

[d] **Distributed deadlock:** As described earlier, this problem can cause the whole cluster to hang (§4).

Summary: One expectation of the no-SPoF principle is that systems should not hang; a hang should be treated as a fail-stop and failed over. The cases above show that hang is still an intricate problem to solve.

6.5 Space

In cloud systems targeted for Big Data, space management (both memory and storage) becomes an important issue (4% of software problems). Below we describe prevalent issues surrounding space management.

[a] **Big data cleanup:** Big old data can easily take up space and require quick cleanup. Cloud systems must periodically clean a variety of big old data such as old error logs, commit logs, and temporary outputs. In many cases, cleanup procedures are still manually done by administrators. If not done in timely fashion, tight space can cause performance issues or downtimes.

[b] **Insufficient space:** Jobs/queries can read/write a large amount of data. We find cases where many jobs fail or hang in the middle of the execution when there is no available memory. Big jobs are not automatically sliced, and thus users must manually do so.

§6.3: [a]c2105, c3306, c4571, f543, hb6060, hb6299, hd2791, m4099, m4157, m4252, m5001, m5476, z1090, z1144, z1496, z1448.

§6.4: [a]m3596, m3355, m3274; [b]hd3166, hd4176, m2209, m4797; [c]hd5299, m3228, m4425, m4751, m4088; [d]See §4.

§6.5: [a]c3005, c3741, hb5611, hb9019, hb9208; [b]m2324, m5251, m5689; [c]c4708, hd3334, hd3373, m5351, z1163, z1431, z1494.

[c] **Resource leak:** Resource leak is a form of bad space management. We observe a variety of leaks such as memory, socket, file descriptor, and system-specific leaks (*e.g.*, streams). In many instances, leaks happen during recovery (*e.g.*, sockets are not closed after recovery). Surprisingly, memory leaks happen in "Java-based" cloud systems; it turns out that the developers do not favor Java GC (§6.6) and decide to manage the memory directly via C malloc or Java bytebuffer, but then careless memory management leads to memory leaks.

Summary: Big data space management is often a manual process. More work on automated space management seems to be needed [27].

6.6 Load

Load-related issues (4%) surface when cloud systems experience a high request load. Below we discuss several causes behind load issues.

[a] **Java GC:** Load-intensive systems cause frequent memory allocation/deallocation, forcing Java GC to run frequently. The problem is that Java GC (*e.g.*, "stop-the-world" GC) can pause 8-10 seconds per GB of heap; high-end servers with large memory can pause for minutes [36]. The developers address this with manual memory management (*e.g.*, with C malloc or Java bytebuffer). This provides stable performance but also potentially introduces memory leaks (§6.5).

[b] **Beyond limit:** Many times, cloud systems try to serve all requests even if the load is beyond their limit. This can cause problems such as backlogs and OOM, which then make the system die and cannot serve any requests. It seems better for cloud systems to know their limits and reject requests when overloaded.

[c] **Operational loads:** As alluded in Section 3.5, without horizontal QoS, request and operational loads can bog down other important operations. For example, we see many situations where error logging creates a large number of disk writes downgrading main operations. This kind of problem is largely found in deployment, perhaps because operational protocols are rarely tested offline with high load.

Summary: Java memory management simplifies users but is not suitable for load-intensive cloud systems. Research on stable and fast Java GC is ongoing [24]. As load issues are related to space management, work on dense data structures is critical [22, 44]. Load tests are hard to exercise in offline testing as live loads can be orders of magnitude higher. Live "spike" tests recently become an accepted practice [43], but they mainly test load sensitivities of main protocols; operational protocols should be load tested as well.

Cassandra		HBase		MapReduce	
Get	95	RegionServer	418	AM	141
Compaction	93	Master	239	NM	69
Mutate	90	Client	238	RM	63
Sstable	86	Cross	172	TaskTracker	60
Client	64	Coprocessors	57	JobTracker	51
Boot	52	Log splitting	46	Security	50
Cross	49	IPC	38	Reduce	44
Gossiper	47	Fsck	33	Client	43
Memtable	37	Snapshot	27	HistoryServer	35
Streaming	31	Commit log	24	Map	28

Table 2: Top-10 "Problematic" Components. *The table shows top-10 components with the highest count of vital issues in Cassandra, HBase and MapReduce.*

7 Implications

Throughout previous sections, we have implicitly presented how hardware and software failures can lead to a wide range of implications, specifically failed operations (42%), performance problems (23%), downtimes (18%), data loss (7%), corruption (5%), and staleness (5%), as shown in Figure 4b.

In reverse, Figure 5 also depicts how almost every implication can be caused by *all* kinds of hardware and software faults. As an implication, if a system attempts to ensure reliability on just one axis (*e.g.*, no data loss), the system must deploy all bug-finding tools that can catch all software fault types and ensure the correctness of all handlings of hardware failures. Building a highly dependable cloud system seems to be a distant goal.

8 Other Use Cases of CBSDB

In the main body of this paper, we present deep discussions of bug aspects, root causes (hardware and software) and implications, and due to space constraints, we unfortunately do not show deeper quantitative analysis. Nevertheless, CBSDB contains a set of rich classifications (§2) that can be correlated in various different ways which can enable a wide range of powerful bug analysis. In the last one year, we have created more than 50 per-system and aggregate graphs from mining CBSDB, some of which we demonstrate in this section. As we will make CBSDB public, we hope it encourages the larger cloud research community to perform further bug studies beyond what we have accomplished.

• **Longest time to resolve and most commented:** The raw bug repositories contain numeric fields such as time to resolve (TTR) and number of developer responses, which when combined with our annotations can enable a powerful analysis. For example, Figures 7a and 7b can help answer questions such as "which software bug types take the

§6.6: [a]c5506, c5521, hb4027, hb5347, hb10191, hd4879; [b]c4415, hb3421, hb5141, hb8143, m5060, c4918; [c]See §3.5.

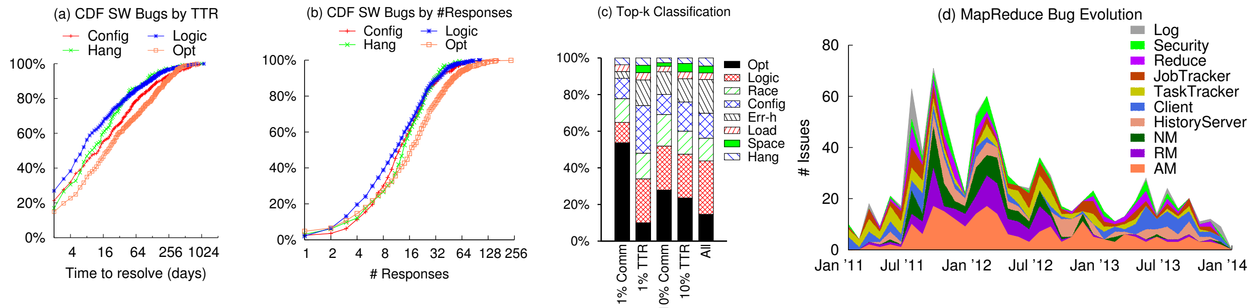


Figure 7: Case Studies. *Figures (a) and (b) show CDFs of software bug types (only four types for graph clarity) based on TTR and #responses. Figure (c) shows the distribution of software bugs as described in §8. Figure (d) shows MapReduce bug evolution.*

longest/shortest time to resolve or have the most/least number of responses?". The figures hint that optimization problems overall have larger TTR and #responses than logic bugs.

• **Top 1% or 10%:** Using CDFs from the previous analysis, we can further derive another empirical analysis as shown in Figure 7c which can answer questions such as "what is the distribution of software bug types in the top 1% (or 10%) of most responded (or longest-to-resolve) issues?". It is interesting to see that within the top 1% most responded bugs, optimization problems cover more than 50% of the issues, but only 10% within the top 1% longest-to-resolve issues.

• **Per-component analysis:** As shown in Table 1 we also add component labels for every system. These labels are useful to answer questions such as "which components have significant counts of issues?". Empirical data in Table 2 can help answer such question. We can see cross-system issues ("cross") are quite prevalent across our target systems. Thus, one can further analyze issues related to how multiple cloud systems interact (*e.g.*, HBase with HDFS, HBase with ZooKeeper). To perform a wide range of component-based analysis, CBSDB users can also correlate components with other bug classifications. For example, one can easily query which components carry data race bugs that lead to data loss.

• **Bug evolution:** Finally, CBSDB users can also analyze bug evolution. For example, Figure 7d shows how issues in different MapReduce components are laid out over time. The spike from mid-2011 to mid-2012 represents the period when Hadoop developers radically changed Hadoop architecture to Hadoop 2.0 (Yarn) [48].

9 Related Work

Throughout the entire paper, we cite related work around each discussion theme. In this section, we briefly discuss other bug-study papers.

Studies of bug/error reports from various systems have proven to be invaluable to the research community. For instance, Chou *et al.* [15] study more than 1000 operating system bugs found by static analysis tools applied to the Linux and OpenBSD kernels; Lu *et al.* [39] perform a comprehensive study of more than 100 local concurrency bugs; Yin *et al.* [54] study 546 real-world misconfiguration issues from a commercial storage system and four different open-source systems; finally, Lu *et al.* [38] cover eight years of Linux file-system changes across 5079 patches and study in detail 1800 of the patches. These studies bring significant contributions in improving software systems reliability.

10 Conclusion

We perform the first bug study of cloud systems. Our study brings new insights on some of the most vexing problems in cloud systems. We show a wide range of intricate bugs, many of which are unique to distributed cloud systems (*e.g.*, scalability, topology, and killer bugs). We believe our work is timely especially because cloud systems are considerably still "young" in their development. We hope (and believe) that our findings and the future availability of CBSDB can be beneficial for the cloud research community in diverse areas as well as to cloud system developers.

11 Acknowledgments

We thank Feng Qin, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. We would also like to thank Yohanes Surya and Teddy Mantoro for their support. This material is based upon work supported by the NSF (grant Nos. CCF-1321958, CCF-1336580 and CNS-1350499).

References

[1] Apache Cassandra Project. http://cassandra.apache.org.

[2] Apache Flume Project. http://flume.apache.org.

[3] Apache Hadoop Project. http://hadoop.apache.org.

[4] Apache HBase Project. http://hadoop.apache.org/hbase.

[5] Apache ZooKeeper Project. http://zookeeper.apache.org.

[6] HDFS Architecture. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[7] The 10 Biggest Cloud Outages Of 2013. http://www.crn.com/slide-shows/cloud/240165024/the-10-biggest-cloud-outages-of-2013.htm.

[8] The Apache Software Foundation. http://www.apache.org/.

[9] The Cloud Bug Study (CBS) Project. http://ucare.cs.uchicago.edu/projects/cbs.

[10] Mona Attariyan and Jason Flinn. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *OSDI '10*.

[11] Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08*.

[12] Cory Bennett and Ariel Tseitlin. Chaos Monkey Released Into The Wild. http://techblog.netflix.com, 2012.

[13] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems Export. In *OSDI '06*.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI '06*.

[15] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *SOSP '01*.

[16] Miguel Correia, Daniel Gomez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical Hardening of Crash-Tolerant Systems. In *USENIX ATC '12*.

[17] Jeffrey Dean. Underneath the Covers at Google: Current Systems and Future Directions. In *Google I/O '08*.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04*.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP '07*.

[20] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-Out Cloud Systems. In *SoCC '13*.

[21] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *FAST '13*.

[22] Bin Fan, David G. Andersen, and Michael Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *NSDI '13*.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*.

[24] Lokesh Gidra, Gal Thomas, Julien Sopena, and Marc Shapiro. A study of the Scalability of Stop-the-World Garbage Collectors on Multicore. In *ASPLOS '13*.

[25] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI '11*.

[26] Haryadi S. Gunawi, Cindy Rubio-Gonzalez, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *FAST '08*.

[27] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI '10*.

[28] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *SOSP '11*.

[29] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure Recovery: When the Cure Is Worse Than the Disease. In *HotOS XIV*, 2013.

[30] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No One Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics. In *SoCC '11*.

[31] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI '11*.

[32] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi. The Case for Drill-Ready Cloud Computing. In *SoCC '14*.

[33] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI '14*.

[34] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos Kawazoe Aguilera, and Michael Walfish. Detecting failures in distributed systems with the FALCON spy network. In *SOSP '11*.

[35] Kaituo Li, Pallavi Joshi, and Aarti Gupta. ReproLite : A Lightweight Tool to Quickly Reproduce Hard System Bug. In *SoCC '14*.

[36] Todd Lipcon. Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers, February 2011.

[37] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *SOSP '11*.

[38] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *FAST '13*.

[39] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics. In *ASPLOS '08*.

[40] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: A Timely Dataflow System. In *SOSP '13*.

[41] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *OSDI '12*.

[42] Ariel Rabkin and Randy Katz. Precomputing Possible Configuration Error Diagnoses. In *ASE '11*.

[43] Jesse Robbins, Kripa Krishnan, John Allspaw, and Tom Limoncelli. Resilience Engineering: Learning to Embrace Failure. *ACM Queue*, 10(9), September 2012.

[44] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured Memory for DRAM-based Storage. In *FAST '14*.

[45] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *SIGMETRICS '09*.

[46] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *OSDI '12*.

[47] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-Based Service Level Agreements for Cloud Storage. In *SOSP '13*.

[48] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC '13*.

[49] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling High-level SLOs on Shared Storage Systems. In *SoCC '12*.

[50] Xi Wang, Zhenyu Guo, Xuezheng Liu, Zhilei Xu, Haoxiang Lin, Xiaoge Wang, and Zheng Zhang. Hang analysis: fighting responsiveness bugs. In *EuroSys '08*.

[51] Yin Wang, Terence Kelly, Manjunath Kudlur, Stephane Lafortune, and Scott Mahlke. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *OSDI '08*.

[52] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do Not Blame Users for Misconfigurations. In *SOSP '13*.

[53] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09*.

[54] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *SOSP '11*.

[55] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-intensive Systems. In *OSDI '14*.

[56] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems. In *SOSP '13*.