What Bugs Live in the Cloud?

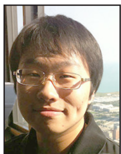
A Study of Issues in Scalable Distributed Systems

HARYADI S. GUNAWI, THANH DO, AGUNG LAKSONO, MINGZHE HAO,
TANAKORN LEESATAPORNWONGSA, JEFFREY F. LUKMAN, AND
RIZA O. SUMINTO

Haryadi Gunawi is a Neubauer Family Assistant Professor in the Department of Computer Science at the University of Chicago where he leads the UCARE Lab (U Chicago systems research on Availability, Reliability, and Efficiency). He received his PhD from the University of Wisconsin–Madison and was awarded an Honorable Mention for the 2009 ACM Doctoral Dissertation Award.
haryadi@cs.uchicago.edu

Thanh Do is a Researcher at Microsoft Jim Gray Systems Lab. His research focuses on the intersection of systems and data management.
thdo@microsoft.com

Agung Laksono is a Researcher in SCORE Lab at Surya University Indonesia. He received his BS from Sepuluh Nopember Institute of Technology (ITS). His research interest is in cloud computing and software engineering.
agung.laksono@surya.ac.id



Mingzhe Hao is a PhD student in the Computer Science Department at the University of Chicago. As a member of UCARE Lab, he is passionate about building performance-predictable cloud storage systems. hmz20000@uchicago.edu

We performed a detailed study of development and deployment issues of six open-source scalable distributed systems (scale-out systems) by analyzing 3655 vital issues reported within a three-year span [4]. The results of our study should be useful to system developers and operators, systems researchers, and tool builders in advancing the reliability of future scale-out systems. The database of our Cloud Bug Study (CbsDB) is publicly available [1].

As the cloud computing era becomes more mature, various scale-out systems—including distributed computing frameworks, key-value stores, file systems, synchronization services, streaming systems, and cluster management services—have become a dominant part of software infrastructure running behind cloud datacenters. These systems are considerably complex as they must deal with a wide range of distributed components, hardware failures, users, and deployment scenarios. Bugs in scale-out systems are a major cause of cloud service outages.

In this study, we focused on six popular and important scale-out systems: Hadoop, HDFS, HBase, Cassandra, ZooKeeper, and Flume, which collectively represent a diverse set of scale-out architectures. A comprehensive study of bugs in scale-out systems can provide intelligent answers to many dependability questions. For example, why are scale-out systems not 100% dependable? Why is it hard to develop fully reliable cloud systems? What types of bugs live in scale-out systems, and how often do they appear? Why can't existing tools capture these bugs, and how should dependability tools evolve in the near future?

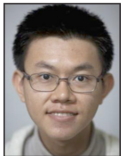
The answers to these questions are useful for different communities. System developers can learn about a wide variety of failures in the field and come up with better system designs. System operators can gain further understandings of distributed operations that are fragile to failure. For system researchers, this study provides bug benchmarks that they can use to evaluate their techniques. This study also motivates researchers to address new large-scale reliability challenges. Finally, tool builders can understand the limitations they work within and advance current tools.

In the rest of this article, we will present our high-level findings by focusing on new interesting types of bugs that we believe require more attention. At the end of this article, we will provide more samples of CbsDB use cases. The full scope of our study and specific bug examples can be found in our conference paper [4].

Findings

Before presenting specific types of bugs, we summarize our important findings.

New bugs in town: As shown in Figure 1, classical issues such as reliability (45%), performance (22%), and availability (16%) are the dominant categories. In addition, new classes of bugs unique to scale-out systems have emerged: *data consistency* (5%), *scalability* (2%), *topology* (1%), and *QoS* (1%) aspects.


Tanakorn Leesatapornwongsa is a PhD student in the Department of Computer Science at University of Chicago. He is a member of the UCARE Lab and is interested in addressing scalability and distributed concurrency problems. tanakorn@cs.uchicago.edu

Jeffrey Ferrari Lukman is a Researcher in SCORE Lab at Surya University, Indonesia. He is joining University of Chicago as a PhD student in computer science in Fall 2015. His research interest is in distributed systems reliability. jeffrey.ferrari@surya.ac.id

Riza Suminto received a BS in computer science from Gadjah Mada University in 2010. In 2013, he joined the University of Chicago to pursue his PhD in computer science. He is currently a member of the UCARE Lab and is interested in addressing performance bugs in cloud systems. riza@cs.uchicago.edu

Classification	Labels
Aspect	Reliability, performance, availability, security, consistency, scalability, topology, QoS
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
Impact scope	Single machine, multiple machines, entire cluster

Table 1: Issue classifications

Handling diverse hardware failures is not easy: "Hardware can fail, and reliability should come from the software" has been preached extensively, but handling diverse hardware failures such as fail stop, corruption, and "limpware[3]," including the timing of failures, is not straightforward (13% of the issues relate to hardware faults).

Vexing software bugs: The 87% of issues that pertain to software bugs consist of logic (29%), error-code handling (18%), optimization (15%), configuration (14%), data race (12%), hang (4%), space (4%) and load (4%) issues, as shown in Figure 3a.

In this article, we will delve into three interesting types of software bugs: (1) *single-point-of-failure* (SPoF) bugs, which can simultaneously affect multiple nodes or the entire cluster; (2) *distributed concurrency bugs,* caused by nondeterministic distributed events such as message reorderings and failure timings; and (3) *performance logic bugs,* which can cause significant performance degradation of the system.

Less-tested operational protocols: User-facing read/write protocols are continuously exercised in deployment and thus tend to be robust. Conversely, operational protocols (e.g., bootstrap logic, failure recovery, rebalancing) are rarely run and not rigorously tested. Bugs often linger in operational protocols.

A wide range of implications: Exacerbating the problem is the fact that each bug type can lead to almost all kinds of implication such as failed operations (42%), performance problems (23%), component downtimes (18%), data loss (7%), corruption (5%), and staleness (5%), as shown in Figure 3b.

The need for multi-dimensional dependability tools: As each kind of bug can lead to many implications and vice versa (Figure 4), bug-finding tools should not be one-dimensional.

Methodology

The six systems we studied 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 the questions we listed above. From the issues repository of each system, we collected issues (bugs and new features) submitted over a period of three years (2011–2014) for a total of 21,399 issues. We manually labeled "vital" those issues pertaining to system development and deployment problems and marked them as high priority. We ignored non-vital issues related to maintenance, code refactoring, unit tests, documentation, and minor easy-to-fix bugs. This left us with 3655 vital issues that we then studied and tagged with our issue classifications as shown in Table 1. In each classification, an issue can

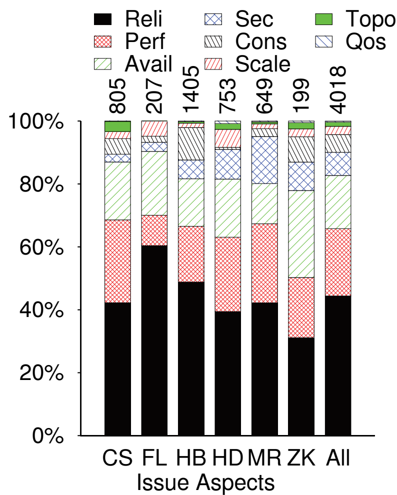


Figure 1: Issue Aspects. CS: *Cassandra; FL: Flume; HB: HBase; HD: HDFS; MR: MapReduce; ZK: ZooKeeper; All: Average*

have multiple sub-classifications. The product of our study is named Cloud Bug Study database (CbsDB) and is publicly available [1]. With CbsDB, users can perform both quantitative and qualitative analysis of cloud bugs.

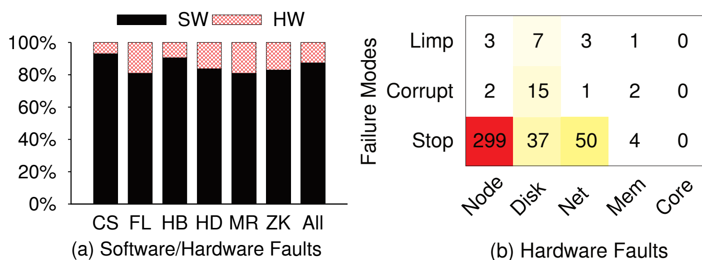
Issue Aspects

The first classification that we use is by aspect. Figure 1 shows the distribution of the eight aspects listed in Table 1. Reliability (45%), performance (22%), and availability (16%) aspects are the three largest categories. They are caused by diverse hardware-related and software bugs that we will discuss in subsequent sections. We also found many vital issues related to security (8%) and QoS (1%). Below, we pay attention to two interesting aspects distinct to scale-out systems: *distributed data consistency* and *scalability bugs*.

Data Consistency

Users demand data consistency, which implies that all nodes or replicas should agree on the same value of data (or eventually agree in the context of eventual consistency). In reality, several cases (5%) show data consistency is violated where users get stale data or the system's behavior becomes erratic. Data consistency bugs are mostly caused by the two following problems:

1. *Buggy logic in operational protocols:* Besides the main read/write protocols, many other operational protocols (e.g., bootstrap, background synchronization, cloning, fsck) touch and modify data, and bugs within them can cause data inconsistency. For example, in the Cassandra cross-datacenter (DC) synchronization protocol, the compression algorithm fails to compress some key-values, but Cassandra allows the whole operation to proceed, silently leaving the two DCs with inconsistent views.



Figures 2a and b: Hardware faults

2. *Concurrency bugs and node failures:* Intra-node (local) 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 (distributed) data races are also a major root cause; complex reordering of asynchronous messages combined with node failures make systems enter incorrect states.

In summary, operational protocols modify data replicas, but they often carry data inconsistency bugs. Robust systems require all protocols to be heavily tested. In addition, more research is needed to address complex distributed concurrency bugs (as we will discuss later).

Scalability

Scalability issues, although small in number (2%), are interesting because they are hard to find in small-scale testing. We categorize scalability issues into four axes of scale: cluster size, data size, load, and failure.

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" (that is, live nodes are extremely busy and considered dead) and requires whole-cluster restart with manual tuning.

Scale of data size: Big Data 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.

Scale of request load: Large request loads of various kinds are sometimes unanticipated. For example, in HDFS, creation of thousands of small files in parallel causes out-of-memory problems (OOM), and in Cassandra, users can generate a storm of deletions that can block other important requests.

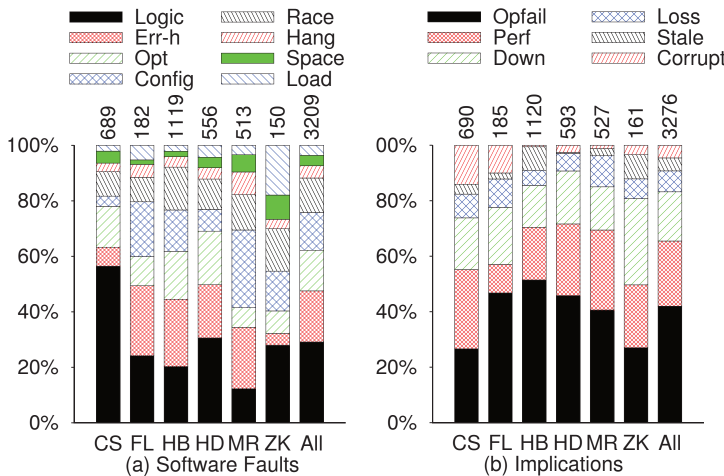


Figure 3a and b: Software bug types and implications

Scale of failure: At scale, a large number of components can fail at the same time, but some recovery protocols handle large scale failures poorly. In one example, when 16,000 mappers failed, Hadoop required over seven hours to recover because of unoptimized communication in HDFS.

In summary, scalability problems surface undesirably late in deployment. Similar to an earlier summary, we find the main read/write protocols scale robustly, but operational protocols (recovery, boot, etc.), on the other hand, often carry scalability bugs. One approach to solve this is via operational "live drills" [5], which should be performed frequently in deployment. Another research challenge is to develop scalability bug finders that can find scalability bugs without using large resources in testing.

Hardware Issues

Next we categorize issues based on hardware vs. software faults. Figure 2a shows the percentage of issues that involve hardware (13%) and software (87%) faults. Figure 2b shows the heat map of correlation between hardware type and failure mode; the number in each cell is a bug count.

While fail stop and corruption are well-known failure modes, there is an overlooked hardware failure mode, *limpware* [3], hardware whose performance degrades significantly. For example, in an HBase deployment, a memory card ran only at 25% of normal speed, causing backlogs, OOM, and crashes.

Software Issues

Figure 3a shows the distribution of software bug types. The average distributions of software issues are: logic (29%), error handling (18%), optimization (15%), configuration (14%), data race/concurrency (12%), hang (4%), space (4%), and load (4%) issues.

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 4: Counts of software bugs and implications

Figure 3b depicts respective software bug implications. The average distributions for the implications are: failed operations (42%), performance problems (23%), downtimes (18%), data loss (7%), corruption (5%), and staleness (5%).

Figure 4 presents an interesting heat map of correlation between software bugs and their implications. Each kind of bug can lead to many implications and vice versa. If a system attempts to ensure reliability on just one axis (e.g., no data loss), the system must deploy various bug-finding tools that can catch different types of software bugs. Therefore, there is a need for multi-dimensional dependability tools.

For interested readers, discussions of the software issues above are discussed in our full paper [4]. Below we focus our discussions on three interesting distributed system bugs: single-point-of-failure (SPoF), distributed concurrency, and performance logic bugs.

SPoF Bugs

One interesting type of bug that we find is "single-point-of-failure (SPoF)" bugs. These bugs can simultaneously affect multiple nodes or even the entire cluster. The presence of these bugs implies that although the "no-SPoF" principle has been preached extensively, SPoF still exists in many forms.

Positive feedback loop: This is the case where failures happen, then recovery starts, but the recovery introduces more load and hence more failures. For example, busy gossip traffic can incorrectly declare live nodes dead, which then causes administrators or elasticity tools to add more nodes, which then causes more gossip traffic.

Buggy failover: A key to no-SPoF is to detect failure and perform a failover. But such a guarantee breaks if the failover code itself is buggy. For example, in HDFS, when a failover to a standby name node breaks, all data nodes become unreachable.

Repeated bugs after failover: Here, a buggy operation leads to a node crash triggering a failover. After the failover, the other node will repeat the same buggy logic, again crashing the node. The whole process will repeat and the entire cluster will eventually die.

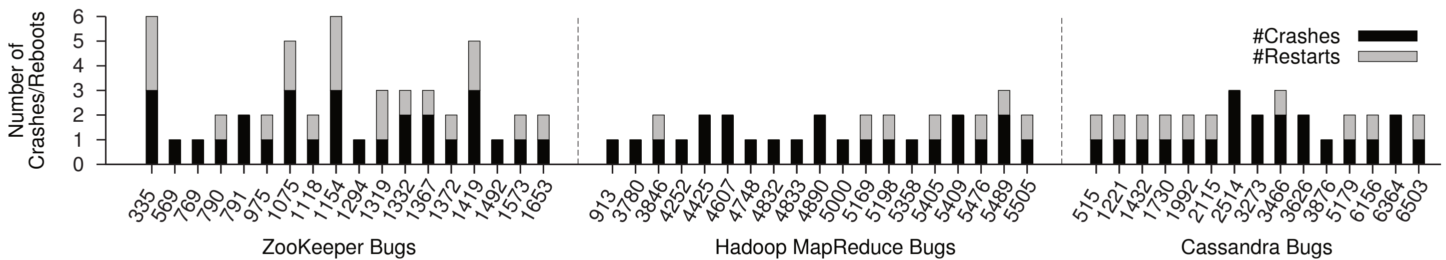


Figure 5: "Deep" distributed concurrency bugs. The x-axis lists bug numbers and the y-axis represents the number of crashes and reboots to unearth deep distributed concurrency bugs.

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

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. As an example, a buggy ZooKeeper leader election protocol can cause no leader to be elected.

Distributed deadlock: This is the case 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 corner-case situation is typically caused by message reorderings, network failures, or software bugs.

Scalability and QoS bugs: Examples presented earlier also highlight that scalability and QoS bugs can affect the entire cluster.

In summary, the concept of no-SPoF is not just about a simple failover. Many forms of SPoF bugs exist, and they can cripple an entire cluster (potentially hundreds or thousands of nodes). Scale-out systems must also be self-aware and make decisions to stop recovery operations that can worsen the cluster condition (for example, in the first two cases above). Future tools must address the five challenges of unearthing various forms of SPoF bugs.

Distributed Concurrency Bugs

Data races are a fundamental problem in any concurrent software system and a major research topic over the last decade. In our study, data races account for 12% of software bugs. Unlike nondistributed software, cloud systems are subject to not only local concurrency bugs (e.g., thread interleaving) but also *distributed concurrency bugs* (e.g., reordering of asynchronous messages). Our finding is that around 50% of data race bugs are distributed concurrency bugs and 50% are local concurrency bugs.

As an extreme example, let's consider the following distributed concurrency bug in ZooKeeper that happens on a long sequence of messages including failure events that must happen in a specific order.

ZooKeeper Bug #335: (1) Nodes A, B, C start with latest txid #10 and elect B as leader; (2) *B crashes*; (3) Leader election rerun, and C becomes leader; (4) Client writes data; A and C commit new txid-value pair {#11:X}; (5) *A crashes before committing tx #11;* (6) C loses quorum; (7) *C crashes;* (8) *A reboots and B reboots;* (9) A becomes leader; (10) Client updates data; A and B commit a new txid-value pair {#11:Y}; (11*) C reboots after A's new tx commit*; (12) C synchronizes with A; C notifies A of {#11:X}; (13) A replies to C the "diff" starting with tx 12 (excluding tx {#11:Y}!); (14) Violation: permanent data inconsistency as A and B have {#11:Y} and C has {#11:X}.

The bug above is what we categorize as a distributed concurrency bug. To unearth this type of bug, testing and verification tools must permute a large number of events, crashes, and reboots as well as network events (messages). Figure 5 lists more samples of distributed concurrency bugs. The point of the figure is to show that many of them were induced by multiple crashes and reboots at nondeterministic timings. Distributed concurrency bugs plague many many protocols, including leader election, atomic broadcast, speculative execution, job/task trackers, resource/application managers, gossiper, and many others. These bugs can cause failed jobs, node unavailability, data loss, inconsistency, and corruption.

For local concurrency bugs, numerous efforts have been published in hundreds of papers. Unfortunately, distributed concurrency bugs have not received the same amount of attention. We observed that distributed concurrency bugs are typically found in deployment (via logs) or manually. 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]."

For this reason, we recently built an advanced semantic-aware model checker (SAMC) [6], a software (implementation-level) model checker targeted for distributed systems. It works by rapidly exercising unique sequences of events (e.g., different reorderings of messages, crashes, and reboots at different timings), and thereby pushing the target system into corner-case situations and unearthing hard-to-find bugs. SAMC is available for download [2].

Scenario Type	Possible Conditions
DLC: Data Locality	(1) Read from remote disk, (2) read from local disk,...
DSR: Data Source	(1) Some tasks read from same data node, (2) all tasks read from different data nodes,...
JCH: Job Characteristic	MapReduce is (1) many-to-all, (2) all-to-many, (3) large fan-in, (4) large fan-out,...
FTY: Fault Type	(1) Slow node/NIC, (2) node disconnect/packet drop, (3) disk error/out of space, (4) rack switch,...
FPL: Fault Placement	Slow down fault injection at the (1) source data node, (2) mapper, (3) reducer,...
FGR: Fault Granularity	(1) Single disk/NIC, (2) single node (dead node), (3) entire rack (network switch),...
FTM: Fault Timing	(1) During shuffling, (2) during 95% of task completion,...

Table 2: A partial anatomy of scenario root causes of performance bugs

Performance Bugs

Another notorious type of bug are performance bugs, which can cause a system to under-deliver the expected performance (e.g., a job takes 10x longer than usual). Conversation with several cloud engineers reflects that performance stability is often more important than performance optimization.

To dissect the root-cause anatomy of performance bugs, we performed a deeper study of vital performance bugs in Hadoop [7]. We found that the root causes of performance bugs are *complex deployment scenarios* that the system failed to anticipate. Table 2 shows a partial root-cause anatomy that we built. The table shows some of the *scenario types* such as "Data Source (DSR)" and *specific conditions* such as "some tasks read from the same data node (DSR_1)."

A performance bug typically appears in a specific scenario. For example, we found cases of untriggered speculative execution when the original task and the backup task read from the same slow remote data node (which can be represented as the combination of DSR_1 & FTY_1 & FPL_1 & DLC_1 as described in Table 2) or when all reducers must read from a mapper remotely and the mapper is slow (JCH_1 & FTY_1 & FPL_2). If one of the conditions is not true, the performance bug might not surface.

These examples point to the fact that performance anomalies are hard to find and reproduce. Scale-out systems make many nondeterministic choices (e.g., task placement, data source selection) that depend on deployment conditions. On top of that, external conditions such as hardware faults can happen in different forms and places.

The challenge is clear: to unearth performance bugs, we need to exercise the target system against many possible deployment scenarios. Unfortunately, performance regression testing is time-consuming and does not cover the complete scenarios. What is missing is fast, pre-deployment detection of performance bugs in distributed systems. One viable approach is the use of formal modeling tools (with time simulation) such as Colored Petri Nets (CPN) and TLA+/PlusCal. To be practical, the next big challenge is to automatically generate formal models that truly reflect the original systems code [7].

Other Use Cases of CbsDB

CbsDB [1] contains a set of rich classifications that can be correlated in various different ways which can enable a wide range of powerful bug analyses. For example, CbsDB can provide answers to questions such as: Which software bug types take the longest/shortest time to resolve (TTR) or have the most/least number of responses? What is the distribution of software bug types in the top 1% (or 10%) of most responded to (or longest-to-resolve) issues? Which components have significant counts of issues? How does bug count evolve over time? More details regarding CbsDB use cases can be found in our full paper [4].

Conclusion

At scale, hardware is not a single point of failure, but software is. A software bug can cause catastrophic failures including downtimes, corruption, and data loss. Our study brings new insights on some of the most intricate bugs in scale-out systems that we hope can be beneficial for the cloud research community in diverse areas as well as to scale-out system developers.

Acknowledgments

This material is based upon work supported by the NSF (grant nos. CCF-1321958, CCF-1336580, and CNS-1350499) as well as generous support from NetApp. We also thank other members of UCARE and SCORE labs, Tiratat Patana-anake, Jeffry Adityatama, Kurnia Eliazar, Anang Satria, and Vincentius Martin for their contributions in this project.

References

[1] http://ucare.cs.uchicago.edu/projects/cbs/.

[2] http://ucare.cs.uchicago.edu/projects/samc/.

[3] 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 *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC ’13), 2013*.

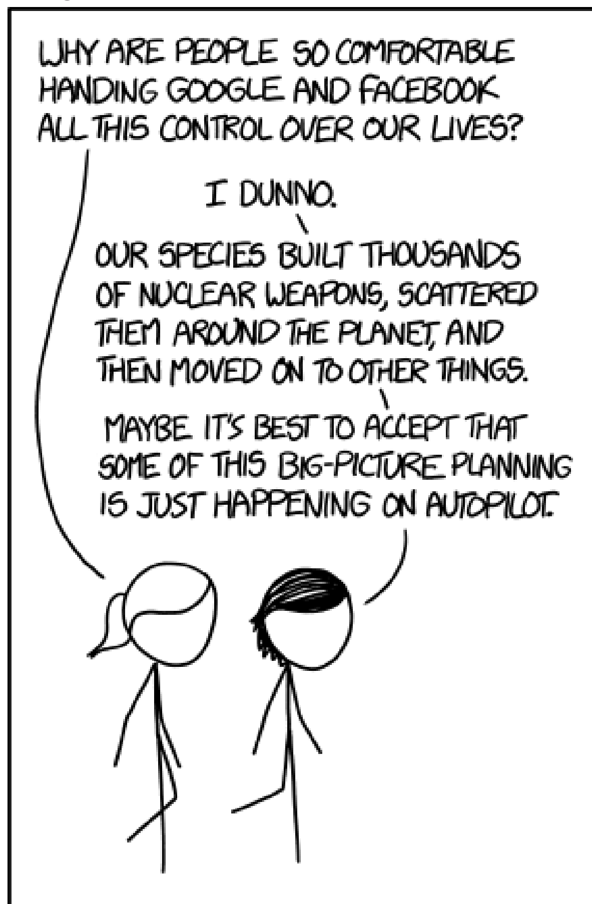
[4] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang Satria, “What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems,” in *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC ’14)* 2014: http://dx.doi.org/10.1145/2670979.2670986.

[5] Tanakorn Leesatapornwongsa and Haryadi S. Gunawi, “The Case for Drill-Ready Cloud Computing,” in *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC ’14), 2014*.

[6] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi, “SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems,” in *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI ’14), 2014*.

[7] Riza O. Suminto, Agung Laksono, Anang D. Satria, Thanh Do, and Haryadi S. Gunawi, “Towards Pre-Deployment Detection of Performance Failures in Cloud Distributed Systems,” in *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud ’15),* 2015.

XKCD



xkcd.com