The Case for Drill-Ready Cloud Computing

Tanakorn Leesatapornwongsa and Haryadi S. Gunawi
University of Chicago

1 Introduction

"The best way to avoid failure is to fail constantly. Learn with real scale, not toy models."
– Netflix engineers [40]

As cloud computing has matured, more and more local applications are replaced by easy-to-use on-demand services accessible via computer networks (a.k.a. cloud services). Running behind these services are massive hardware infrastructures and complex management tasks (*e.g.*, recovery, software upgrades) that if not tested thoroughly can exhibit failures that lead to major service disruptions. Some researchers estimate that 568 hours of downtime at 13 well-known cloud services since 2007 had an economic impact of more than $70 million [18]. Others predict worse: for every hour it is not up and running, a cloud service can take a hit between $1 to 5 million [32]. Moreover, an outage of a popular service can shutdown other dependent services [11, 37, 59], leading to many more frustrated and furious users.

The situation is not getting better due to the bleak future of offline testing. The success of online web services has boosted the "ship early, ship often" trend [46] where features are deployed quickly with little beta testing. Developers do not have the luxury of time to test a system against many failure scenarios. Therefore, in offline testing, the scale of the workloads, environments, and injected failures are often orders of magnitude smaller than the scale of real deployments. Nonetheless, service providers often believe that the recovery will work correctly in many failure scenarios. In reality, past service outages prove that such expectations are often not met.

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA
ACM 978-1-4503-3252-1.
http://dx.doi.org/10.1145/2670979.2670992

In this paper, we raise this fundamental question: *how can we ensure that cloud services work robustly against many failure scenarios in real deployments?* The Netflix's quote above sums up a new method to improve dependability. As many failure scenarios cannot be covered in offline testing, failures should be *deliberately* injected online in actual deployments. We name this method *"failure drill"*. The principle here is to make failure a first-class citizen: rather than waiting for unplanned failures to happen, cloud services should plan and schedule failure drills from time to time (analogous to a routine exercise of fire drills), thereby unearthing in-production recovery issues early before they lead to major outages.

Despite these benefits, failure drill unfortunately remains a "controversial" idea, mainly because no service provider would like to report to their clients "a failure drill that we scheduled has caused an outage/data loss/performance disruption." The risk is too high. For example, 60% of companies that lose customer data or cannot resume operations in 10 days are likely to go out of business [14]. As a result, this method is only accepted "psychologically", but not practically. That is, many believe it is important but few really make it a common practice.

Our vision is to make failure drills prevalent (*i.e.*, a routine practice in deployments of thousands of cloud systems). To the best of our knowledge, no existing literature has laid out efforts and solutions in this space. Until this happens, cloud providers cannot reap the power of failure drills. To address this pressing issue, we propose *drill-ready cloud computing*, a new reliability paradigm that enables cloud systems to perform failure drills in an easy, safe and efficient manners (analogous to a proper fire drill preparation).

To illustrate this, imagine a usage scenario where an administrator would verify if the system can survive 25% of machine failures. Given a drill-ready system, the administrator can write a simple drill specification. The system then can prepare itself for the drill (*e.g.*, spawn into normal and drill modes) and "virtually" kill the machines in the drill mode to trigger actual recovery, automatically monitor the recovery process, carefully ensure isolation (to prevent data loss and performance disruption), and finish the drill if recovery works fine or cancel the drill and restore the normal state if a new recovery anomaly is found. All of the hurdles of exercising failure drills are now built into a drill-ready system and removed from the responsibility of the administrator.

Service Outage	Root Event → *"Supposedly Tolerable" Failure* → **Incorrect Recovery** → Major Outage
EBS [51]	Network misconfiguration → *Huge nodes partitioning* → **Re-mirroring storm** → many clusters collapsed
Gmail [53]	Upgrade event → *Some servers offline* → **Bad request routing** → All routing servers went down
App Eng. [54]	Power failure → *25% machines of a DC offline* → **Bad failover** → All user apps in degraded states
Skype [56]	System overload → *30% supernodes failed* → **Positive feedback loop** → Almost all supernodes failed
Google Drive [55]	bug in network control → *Some network offline* → **Timeouts during failover** → 33% requests affected for 3 hours
Outlook [58]	Caching service failures → *Failover to backend servers* → **Request flooding at backend** → 7-hour service outage
Pertino [41]	Network failure → *Network partition* → **Bad migration** → 6-hour service disruption
Joyent [57]	Operator error→ *Whole DC reboots* → **PXE request flooding in whole-DC boot** → Boot takes 1-hour or more
Yahoo Mail [36]	Hardware failue → *Some servers offline* → **Buggy failover to back-up systems** → 1% of users affected for days

Table 1: Major outages of popular cloud services than lasted for hours to days.

In subsequent sections, we present an extended motivation (§2) and our proposed building blocks of drill-ready clouds: safety, efficiency, usability, and generality (§3). We note that the purpose of this paper is not to propose a specific solution, but simply to present a new dependability paradigm for cloud computing and the challenges that come with it.

2 Extended Motivation

In this section, we raise a basic question: *what is missing in cloud dependability research?* To answer this, we journey through the history of dependability research in the last decade, as summarized in Figure 1.

2.1 Fault-Tolerant Systems

As failures are commonplace, fault-tolerance becomes a necessity, not an option. Almost all deployed cloud systems are equipped with complex fault-tolerant protocols that handle a variety of failures.

Limitations: Complex failure recovery protocols are unfortunately hard to implement correctly, especially in low-level system languages like C/C++ and Java. Performance optimization code also often downgrades the robustness of theoretically-proven protocols. As a result, recovery code is susceptible to bugs [16, 21, 23, 31].

2.2 Offline Testing

Undoubtedly, recovery code must be thoroughly tested before deployed, which is the goal of offline testing. Many various forms of offline recovery testing have been proposed, ranging from fault injections, model checking, stress testing, static analysis, failure modeling, and running "mini clusters" to emulate production scenarios.

Limitations: Although offline testing provides a certain guarantee of recovery correctness, the scale of the injected failures, workloads, and environments (*e.g.*, #machines) are often orders of magnitude smaller than the scale of real deployments. For example, with hundreds of millions of users worldwide, Skype only emulates thousands of users [56]. Similarly, Facebook uses a 100-machine cluster that mimics the workload of 3000-machine production clusters [20]. Furthermore, only few companies deploy mini clusters for testing purposes, while most others forego such luxury.

2.3 Service Outages

• **Lessons from outage headlines:** Even with offline testing, recovery does not always work as expected and could lead to service outages. We illustrate this problem by listing some high-profile outages in Table 1; they exhibit a certain pattern: some root events led to "supposedly tolerable" failures (in italic); however, the broken recovery (in bold) gave rise to major outages. For example, A fraction of Gmail's servers were taken offline for a routine upgrade (which reportedly "should be fine"), but due to some recent changes on the re-routing code, several servers refused to serve the extra load causing a ripple of overloaded servers and resulting in a global outage [53]. Similarly, Skype faced some overload that caused 30% of the supernodes to go down. The rest of the supernodes were not able to handle the extra responsibility, creating a "positive feedback loop" that led to a near complete outage [56]. More recently, a bug in a network control caused a portion of Google's network capacity to go offline, triggering re-routing and load recovery to other servers which then caused unexpected increase in latency that triggered a second bug that caused errors and timeouts on 33% of the user requests [55]. Microsoft ActiveSync service went down for 7 hours because of failures on their caching service which then swamped the slower backend servers with high request load [58]. Overall, the reality is clear: outages due to large-scale recovery issues still plague many cloud services regularly [1, 42], and they are still an unsolved problem.

• **Lessons from Cloud Bug Study:** Recently, we conduct an in-depth and large-scale study of cloud development and deployment issues from the bug repositories of six open-source cloud systems [22]. From this study, we also found numerous bugs that are hard to find in offline testing. Below are just a few samples.

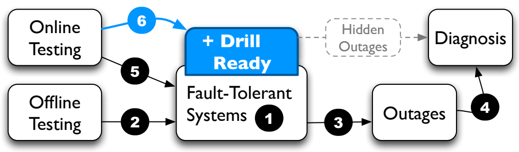

Figure 1: Dependability research.

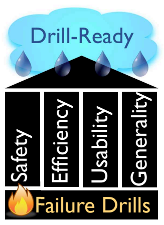

Figure 2: Drill Ready.

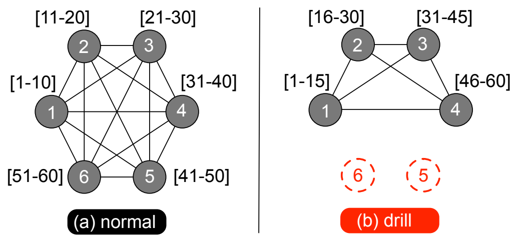

Figure 3: Normal vs. drill topology.

First, at scale, a large number of components can simultaneously fail. For example, a bad recovery in Hadoop took more than 7 hours to recover 16,000 failed mappers. In a ZooKeeper case, 1000 clients were simultaneously disconnected, leading to a close-request stampede that bogged down other important requests. In an HDFS case, jobs writing to 100,000 files got killed causing lease recovery stampede that crippled the cluster.

Second, there exists cases of positive feedback loop where recovery inadvertently introduces more load and hence more failures. For example, in Cassandra, heavy gossip traffic at scale caused live nodes declared dead incorrectly; adding more nodes to replace the "dead" nodes caused more gossip traffic. In ZooKeeper, snapshot synchronization on very large data set triggered a timeout which led to another synchronization; as this continues on infinitely, incomplete snapshots accumulated on disks and the cluster ran out of disk space.

Finally, we also see problems with simultaneous whole-cluster reboot causing high reboot traffic, long global lock contention, and distributed deadlock. In summary, all the bugs above were found in deployment. We believe it is hard, if not impossible, for offline testing to cover such large failures, loads, and data sets.

2.4 Diagnosis

As service outages and disruptions continued to occur, the last decade witnessed growing research in diagnosis [10, 17, 35, 44, 47], which assists users, developers and administrators to pinpoint and reproduce root causes of outages or disruptions.

Limitations: Although valuable, diagnosis is limited in two ways. First, diagnosis detects symptoms that lead to disruptions but does not necessary *prevent* the disruptions. Preventing disruptions is still the responsibility of the fault-tolerant protocols built in the system. Second, diagnosis is a *passive* entity. It waits for disruption symptoms to appear after which it can begin diagnosing. In the literature, diagnosis techniques are mostly evaluated based on issues that already happened in the past.

2.5 Online Testing and Drills

Due to all the limitations above, there is a revamp of online testing where tests are run directly on production systems. Two factors make this idea attractive: users outnumber testers and in-production systems enter deep scenarios not reachable in offline testing. Many instances of online testing have been proposed [12, 49, 60], but they do not exercise failures. Failure drill is an instance of online testing, and large web services realize its importance; large service providers have learned from experience the importance of "failing often" [40].

Limitations: At the current state, failure drill remains controversial due to its risky nature (no guarantee of safety, efficiency, etc.). Let's consider Netflix's Chaos Monkey tool [9] which can automatically kill virtual machines in a probabilistic manner. The catch is that it requires engineers to "be alert and able to respond" [9]. What if the engineers respond too late (*e.g.*, some data is already lost) and cannot revert back to the normal pre-drill state? In a related story, at a major venue, there was a "load spike drill" where cloud services were challenged with load spikes and failures [5, 45]. However, not many providers participated because of the fear that they would fail, and indeed, one company that participated had to manually abort the drill [20].

In summary, failure drill is only common within a few large organizations with the engineering resources, operational expertise, and managerial discipline to execute it [20]. In fact, to the best of our knowledge, we know only a handful companies that perform this in practice (*e.g.*, Amazon, Google, Microsoft, Netflix), and none of them describe their strategies in detail; we suspect they have many skilled operators on stand by. Our goal is clear: failure drill should be a common practice, ideally in every cloud service deployment.

2.6 A Missing Piece

In the context of failure drill, unfortunately *the journey through cloud dependability research seems to end at the last step above*. We are not aware of any published work that attempts to address the risks, opportunities, and fundamental challenges of exercising failure drills. We therefore advocate the continuation of this journey into a new territory: drill-ready cloud computing, an era where failure drills become a regular, non-risky, non-disruptive, easy-to-do, and automated operation.

3 Drill-Ready Clouds

To realize this new paradigm, we identify the four fundamental building blocks of drill-ready clouds: *safety, efficiency, usability* and *generality,* as illustrated in Figure 2. We now describe each of the building blocks along with our proposed solutions.

3.1 Safety

Analogous to a proper fire drill preparation, a drill-ready system should guarantee safety. The challenge is how to learn about failure implications *without* suffering through them. A drill would like to check if customer data could be lost without really losing it, or if SLOs could be violated without violating them for a long period of time. Also, if a drill goes bad, the system must automatically cancel the drill and quickly restore to the healthy state. This safety challenge gets harder due to the possibility of real failures occurring during a drill. Guaranteeing safety is arguably the most fundamental task. Below we propose our safety solutions.

• **Drill State Isolation:** To exercise a drill safely, a system must be drill-aware. This can be accomplished by supporting two modes (normal and drill modes) and isolating the normal state management from the drill one. Let's consider a key-value cluster that contains a simple ring of six nodes with key space [1-60] as shown in Figure 3a (2^{64} key space in practice). In this architecture, every node monitors the availability of all nodes (as shown by the edges). If a drill virtually removes nodes 5 and 6, the system will maintain two topologies: normal and drill topologies. In the drill topology, the key-range responsibility for each node will be re-balanced as shown in Figure 3b, and thus key operations (get and put) will be routed differently in these two modes.

The separation of normal and drill states provides several benefits. First, we can easily select which users/requests (*e.g.*, "gold" vs. "bronze" users) would be rerouted to the drill topology. Second, resource accounting between the two modes can be done easily; the system can ensure resource requirements (*e.g.*, disk or network bandwidth) of the normal mode is not jeopardized. Finally, upon drill cancellation, the system can quickly fall back to the normal state. Any requests affected by the drill will be migrated back to the normal state in an efficient way (more in Section 3.2). This technique is different from the expensive use of cloning where normal state is copied to other "test nodes" [13, 29, 60]. In our case, there are no test nodes and both modes know the existence of each other.

• **Self Cancellation:** As we target distributed systems, there will be a drill master (the "commando") that sends drill commands to other nodes (the agents). In this context, safety requires the anticipation of real failures during the drill. For example, a master or a network failure could partition the commando from the agents. Such a situation could unsafely place the agents in "limbo" for an indefinite period of time. Therefore, self drill cancellation must be installed on every node. An agent can autonomously revert back to the normal mode if it cannot reach the commando or if a drill becomes harmful.

• **Safe Drill Specifications:** For better usability, drills should be written in some form of declarative specifications (more in §3.3). A drill specification defines what failures to inject, how long the drill should take, on what conditions the drill should be cancelled, and some other drill properties. Before a drill specification is executed, its safety must be verified, especially because specifications are written by human administrators. Imagine an administrator who forgets to write cancellation conditions; a harmful drill can run in a prolonged period. Or, consider a specification that virtually kills a node that stores the last surviving replica of some data (while the background re-replication is still ongoing); here, the data will "not exist" in the drill mode. Another motivating story comes from Facebook where the engineers are highly careful in turning up the scale of failure drills slowly and steadily (even in their mini test clusters) [20]. Here, the safety precaution is only done manually. With a clear specification language, safety checks can be automated via some formal analysis tool built on top. A simple example is a check that always requires a cancellation condition to exist (more in §3.3).

• **Delete/Overwrite Prevention:** To prevent data loss, a drill-ready system must carefully monitor data deletion, overwrites/merging, and metadata changes (altered metadata can make data inaccessible, and hence "lost"). Runtime guards can be added to prevent accidental deletion/merging in drill mode. Just-in-time metadata versioning can also be added to prevent accidental changes to data pointers. Here, versioning is only performed on metadata and is only run during a drill, which is far more efficient than all-time full data versioning.

• **SLA Protection:** Imagine a scenario where some user requests (under drill) are served longer than the 95-percentile latency. To prevent such SLA violation, SLA protection must be added. If the SLA is not met in drill mode, the requests can be forwarded to the normal mode, and this event recorded in error logs. This way the system administrator captures the impact of the drill without affecting the real users.

3.2 Efficiency

Exercising drills could lead to extra data migration, which then consume extra resources such as network bandwidth and storage space. As we target data-intensive systems, efficiency-oblivious drills will incur lots of performance overhead. Moreover, in the world of utility computing, monetary overhead must be accounted (extra compute cycles, I/Os, and storage space will be charged). We believe there is a large design space of drill optimization techniques.

• **Domain-Specific Workload Sampling:** Cloning real workloads could consume double resources. A viable solution is to sample some requests to participate in a drill, for example based on some domain-specific workload classification (*e.g.*, sample free customers but not the paying ones). Sampling reduces drill resource consumption but it does not reflect real failure scenarios where *all* requests are affected. To emulate this, a drill can generate extra artificial workload using load spike mechanisms (as long as SLA is not broken).

• **Low-Overhead Drill Setup and Cleanup:** Starting and finishing a drill incurs setup and cleanup costs. To illustrate this, let's quantify the setup cost of the drill in Figure 3b where two nodes are removed. The drill setup leads to key migration (*e.g.*, key range [11-15] must be moved from node 2 to node 1). This setup cost however might be unnecessary as it *depends* on the objective of the drill. For example, this particular drill can be used for two different objectives: (1) to measure the impact of background key migration or (2) to measure the performance of future requests when the cluster is downsized by two nodes. In the second case, a full key migration is unnecessary and uneconomical (*e.g.*, imagine key 11 is migrated to node 1 but is never used by future requests during the drill).

Now let's consider the cleanup cost. The drill topology in Figure 3b will vanish as the drill ends, and any update requests affected by the drill topology must now be moved to its normal location (*e.g.*, update on key 11 that was rerouted to node 1 during the drill must now move to node 2). Imagine thousands of affected keys that must be migrated back. Here, the cleanup procedure can suddenly generate a harmful bursty load that could negatively affect normal workload or could prolong the cancellation phase of a bad drill.

We believe a key to optimization is the knowledge of drill objectives. For example, for the second objective above, key migration can be done lazily on demand (*e.g.*, a key is migrated when it is requested in the drill mode). Therefore, drill objectives must be specified as part of the drill specification. Furthermore, to ensure non-disruptive cleanup, the system can decouple metadata and data migration where metadata (small size) is migrated directly for consistency and data (large size) is migrated lazily on demand. For example, given an updated key 11, we will put a data pointer `key11`$\rightarrow$`node1` in node 2 such that the value of key 11 can be fetched correctly when `get(11)` is requested in the future; in other words, cleanup can be piggybacked with real requests.

• **Cheap Drill Specifications:** To further reduce drill cost, one can write "smarter and cheaper" drill specifications. For example, if a recovery (*e.g.*, file re-mirroring) seems to be going well half-way in the process or user service-level agreements are not violated, the drill can stop and report success.

• **Monetary Cost Reduction via Simulation:** Exercising drills in the world of utility computing (*e.g.*, Amazon EC2) will incur monetary overhead. Let's assume a machine costs \$0.25/hour and storage space \$0.03/GB/month [2]. A drill that shuts down 100 stateless front-end servers for 3 hours would only cost roughly \$75 (as recovery simply spawns another 100 machines). A drill that kills 100 2-TB storage servers would re-mirror 200 TB of data which would cost \$6000 if not done carefully (and if daily/weekly rent is not allowed). Thus, whenever appropriate, the system can mix simulation and live experimentation. For example, to cut storage cost, some file writes could be simulated (*e.g.*, re-mirrored data caused by a drill does not require on-disk space), and similarly for network. Accurately simulating the network of hundreds of machines is a difficult problem. Fortunately, network data transfer is cheap; it is free within a DC and only costs \$0.02/GB across DCs [2]. Storage is expensive, but fortunately easier to simulate due to its localized nature.

3.3 Usability

To be highly usable, a drill-ready system should allow an administrator to express some form of drill specification. Below we describe approaches to improve drill usability.

• **Declarative Drill Specification Language:** An administrator might want to run a drill that starts during a peak load, virtually disconnects 5% of the machines, stops when SLA is broken in the last 1 minute or cancels halfway with success if recovery progresses well after 10 minutes. Furthermore, the same administrator might desire to run another set of drills with larger failures (*e.g.*, 10%, 20%) at different load (*e.g.*, morning, noon). To simplify usability, these drills should be written in a form of executable specifications.

Based on our past experience, one ideal solution is to use a declarative logic language with runtime support. For example, we have used a Datalog-based framework to express recovery specifications [21]. Others have illustrated additional benefits of the language framework, such as the ability to run distributed systems [7], the ease of adding verification checks [50, 61], and the extensibility of adding new constructs and runtime supports [34, 62]. In the context of drills, the specification language should define important drill-specific constructs such as the condition upon which the drill should start and stop, the failed resources including the quantity, and the affected requests, all of which will be executed by the language runtime in a continuous fashion. Unlike a simple start-up script, drill specifications are always re-evaluated as new feeds come (*e.g.*, new recovery progress that comes from the live monitoring tool).

• **Drill Coverage Metrics and Crowdsourcing:** Ideally, an administrator should run unique drills from time to time in order to cover different failure scenarios. To improve usability, there is a need to define *drill coverage metrics*. Unlike traditional metrics such as code path/branch coverage, new evaluation metrics are required, metrics such as failure scale (*e.g.*, kill 1% or 10% of the machines) and request load (*e.g.*, during day or night). When a high-profile outage happens, other service providers typically ask the question "Can my system survive the same failure scenario?". We believe drill coverage metrics will provide a means to measure the fault tolerance of deployed systems.

To improve usability further, *drill crowdsourcing* will be an invaluable and effective approach. If two organizations run the same system with the same configuration, they do not need to run the same drill twice, but in fact they can run different drills and improve coverage. To support this, each type of system (*e.g.*, Hadoop, HBase, Cassandra) should have a community database containing coverage metrics such as system version number, #deployment nodes, rack topology, specific system configuration (*e.g.*, consistent, vs, random hashing), failure size, peak load, and the outcomes of the drill. A third-party service can be used to protect the privacy of deployment metrics. With this information, an organization can exercise new unique drills (*e.g.*, change failure size to 20% while retaining the other attributes). We believe drill crowdsourcing is unique compared to other crowdsourcing contexts [6, 25, 26].

3.4 Generality

Cloud services periodically must run operational tasks such as software upgrades and configuration changes. These operational codes are rarely run and are "fragile", and thus could lead to outages. Therefore, the paradigm of drill-ready computing can be made general beyond failure drills, specifically by including *operational drills* such as elasticity, configuration, upgrade, and perhaps security attack drills. Below are several operational drills that can be built with similar foundations we discussed in previous sections; these drills below have similarity with failure drills.

• **Elasticity Drill:** Many distributed systems are designed to be elastic [30, 38, 48]; a cluster can scale up the number of nodes on the fly based on load demand. This however involves complex background operations such as load rebalancing, data reassignments, and many other tasks that can fail in the middle of the process [22]. In many systems, administrators are expected to perform these operations manually and correctly via command lines [3, 4]. Fortunately, elasticity drills are similar to failure drills; while the former is about scaling out resources, the latter is about reducing resources.

• **Configuration Change Drill:** Configuration change is a major root cause of service disruptions [22, 24, 51, 52]. In our view, configuration changes are similar to background operations described above; it modifies system states and is hard to undo. Therefore, we believe similar mechanisms can be applied here. Recent work proposes post-mortem troubleshooting techniques to pinpoint misconfiguration root causes [43, 8]. On the contrary, config change drills attempt to prevent real consequences of possible misconfigurations.

• **Software Upgrade Drill:** To prevent a complete outage, current practice advocates for rolling upgrades where updates are applied to a subset of nodes at a time [15, 39]. If an upgrade fails, it must be rolled back. However, it is hard to guarantee that a rollback is possible (*e.g.*, an upgrade that leads to a deadlock). We believe the idea of software upgrade drills will improve the current practice for upgrades.

• **Security Attack Drill:** Many recent work have improved the resiliency of software infrastructure against security attacks [19, 28, 27, 33]. We believe it is interesting to test how security defenses work under failures in production; this way we can unearth security vulnerabilities under failures.

4 Conclusion

A decade ago, the concept of failure drill perhaps was considered outrageous. Today, failure is the norm; all pieces of hardware can fail. It is part of "the cloud's daily life". Therefore, failure drill is an accepted concept, but unfortunately only "psychologically", not practically. Our hope is to make it pervasive in practice: every cloud service will routinely exercise failure drills in a safe, efficient, and easy manner. To reach this, we continue the journey of cloud reliability research towards the paradigm of drill-ready clouds.

We have accomplished some initial work on drill master and client architecture and drill state isolation. The details are unfortunately beyond the scope of this vision paper where our primary goal is to deliver this new concept of drill-ready clouds and the many system challenges that come with it.

5 Acknowledgments

We thank Sriram Rao, our shepherd, and the anonymous reviewers for their tremendous feedback and comments. This material is based upon work supported by the NSF (grant Nos. CCF-1321958, CCF-1336580, and CNS-1350499).

References

[1] http://cloutage.org.

[2] Amazon Web Services. http://aws.amazon.com.

[3] Apache HBase Operational Management. http://hbase.apache.org/book/ops_mgt.html.

[4] Cassandra Operations. http://wiki.apache.org/cassandra/Operations.

[5] DevOps GameDay. https://github.com/cloudworkshop/devopsgameday/wiki.

[6] Open Sourced Vulnerability Database. http://www.osvdb.org.

[7] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell C. Sears. BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *EuroSys '10*.

[8] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *OSDI '12*.

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

[10] Sapan Bhatia, Abhishek Kumar, Marc E. Fiuczynski, and Larry Peterson. Lightweight, High-Resolution Monitoring for Troubleshooting Production Systems. In *OSDI '08*.

[11] Henry Blodget. Amazon's Cloud Crash Disaster Permanently Destroyed Many Customers' Data. http://www.businessinsider.com, 2011.

[12] Andrew Bosworth. Building and testing at Facebook. http://www.facebook.com/Engineering, 2012.

[13] Marco Canini, Vojin Jovanović, Daniele Venzano, Boris Spasojević, Olivier Crameri, and Dejan Kostić. Toward Online Testing of Federated and Heterogeneous Distributed Systems. In *USENIX ATC '11*.

[14] Boston Computing. Data Loss Statistics. http://www.bostoncomputing.net.

[15] Olivier Crameri, Nikola Knezevic, Dejan Kostic, Ricardo Bianchini, and Willy Zwaenepoel. Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System. In *SOSP '07*.

[16] 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*.

[17] U. Erlingsson, M. Peinado, S. Peter, and M. Budiu. Fay: Extensible Distributed Tracing from Kernels to Clusters. In *SOSP '11*.

[18] Loek Essers. Cloud Failures Cost More Than $70 Million Since 2007, Researchers Estimate. http://www.pcworld.com, 2012.

[19] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazieres, John C. Mitchell, and Alejandro Russo. Hails: Protecting Data Privacy in Untrusted Web Applications. In *OSDI '12*.

[20] Haryadi S. Gunawi, Thanh Do, Joseph M. Hellerstein, Ion Stoica, Dhruba Borthakur, and Jesse Robbins. Failure as a Service (FaaS): A Cloud Service for Large-Scale, Online Failure Drills. UC Berkeley Technical Report UCB/EECS-2011-87.

[21] 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*.

[22] 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 *SoCC '14*.

[23] 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.

[24] Weihang Jiang, Chongfeng Hu, Shankar Pasupathy, Arkady Kanevsky, Zhenmin Li, and Yuanyuan Zhou. Understanding Customer Problem Troubleshooting from Storage System Logs. In *FAST '09*.

[25] Baris Kasikci, Cristian Zamfir, and George Candea. RaceMob: Crowdsourced Data Race Detection. In *SOSP '13*.

[26] Emre Kiciman and Benjamin Livshits. Ajaxscope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *SOSP '07*.

[27] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich CSAIL. Efficient Patch-based Auditing for Web Application Vulnerabilities. In *OSDI '12*.

[28] Taesoo Kim, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Intrusion Recovery Using Selective Re-execution. In *OSDI '10*.

[29] Oren Laadan, Nicolas Viennot, Chia che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive Detection of Process Races in Deployed Systems. In *SOSP '11*.

[30] H. Andres Lagar-Cavilla, Joseph A. Whitney, Adin Scannell, Stephen M. Rumble, Philip Patchin, Eyal de Lara, Michael Brudno, and M. Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. In *EuroSys '09*.

[31] 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*.

[32] David Linthicum. Calculating the true cost of cloud outages. http://www.infoworld.com, 2013.

[33] Lionel Litty, H. Andres Lagar-Cavilla, and David Lie. Computer Meteorology: Monitoring Compute Clouds. In *HotOS XII*, 2009.

[34] Changbin Liu, Boon Thau Loo, and Yun Mao. Declarative Automated Cloud Resource Orchestration. In *SoCC '11*.

[35] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3S: Debugging Deployed Distributed Systems. In *NSDI '08*.

[36] Marissa Mayer. An Update on Yahoo Mail, December 2013.

[37] Rich Miller. Amazon Cloud Outage KOs Reddit, Foursquare and Others. http://www.datacenterknowledge.com, 2012.

[38] Michael J. Mior and Eyal de Lara. FlurryDB: A Dynamically Scalable Relational Database with Virtual Machine Cloning. In *SYSTOR '11*.

[39] Iulian Neamtiu and Tudor Dumitras. Cloud Software Upgrades: Challenges and Opportunities. In *MESOCA '11*.

[40] Netflix. 5 Lessons We've Learned Using AWS. http://techblog.netflix.com, December 2010.

[41] Pertino. April 1st Service Disruption Postmortem, April 2013.

[42] Ken Presti. 6 Devastating Cloud Outages Over The Last 6 Months. http://www.crn.com, 2013.

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

[44] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI '06*.

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

[46] Chuck Rossi. Ship early and ship twice as often. https://www.facebook.com/Engineering, 2012.

[47] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *NSDI '11*.

[48] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *SoCC '11*.

[49] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? . In *OSDI '10*.

[50] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. In *EuroSys '06*.

[51] AWS Team. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. http://aws.amazon.com/message/65648, 2011.

[52] AWS Team. Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region. http://aws.amazon.com/message/680587, 2012.

[53] Gmail Team. More on today's Gmail issue. http://gmailblog.blogspot.com, September 2009.

[54] Google AppEngine Team. Post-mortem for February 24th, 2010 outage. https://groups.google.com/group/google-appengine, February 2010.

[55] Google Apps Team. GoogleApps IncidentReport, March 2013.

[56] Skype Team. CIO update: Post-mortem on the Skype outage (December 2010). http://blogs.skype.com, December 2010.

[57] The Joyent Team. Postmortem for outage of us-east-1, May 2014.

[58] The Verge. Microsoft apologizes for Outlook, ActiveSync downtime, says error overloaded servers, August 2013.

[59] Christina Warren. How Facebook killed the Internet. http://www.cnn.com, 2013.

[60] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed. Distributed Systems. In *NSDI '09*.

[61] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure Network Provenance. In *SOSP '11*.

[62] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, , and Yun Mao. Efficient Querying and Maintenance of Network Provenance at Internet-Scale. In *SIGMOD '10*.