Failure as a Service (FaaS): A Cloud Service for Large-Scale, Online Failure Drills

Haryadi S. Gunawi
Thanh Do
Joseph M. Hellerstein
Ion Stoica
Dhruba Borthakur
Jesse Robbins

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-87
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-87.html

July 28, 2011

Acknowledgement

This material is based upon work supported by the NSF/CRA Computing Innovation Fellowship and the National Science Foundation under grant numbers CCF-1016924 and CCF-1017073. We also thank Peter Alvaro for his feedback and comments. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

Failure as a Service (FaaS):
A Cloud Service for Large-Scale, Online Failure Drills

Haryadi S. Gunawi, Thanh Do*, Joseph M. Hellerstein, Ion Stoica,
Dhruba Borthakur†, and Jesse Robbins‡

University of California, Berkeley *University of Wisconsin, Madison †Facebook ‡Opscode

Abstract

Cloud computing is pervasive, but cloud service outages still take place. One might say that the computing forecast for tomorrow is "cloudy with a chance of failure." One main reason why major outages still occur is that there are many unknown large-scale failure scenarios in which recovery might fail. We propose a new type of cloud service, Failure as a Service (FaaS), which allows cloud services to routinely perform large-scale failure drills in real deployments.

1 Introduction

> *"The best way to avoid failure is to fail constantly."*
> *"Learn with real scale, not toy models."*
> – Netflix Engineers [21]

Cloud computing has matured. More and more local computing applications are replaced by easy-to-use on-demand services accessible via computer network (a.k.a. cloud services). Running behind these services are massive hardware infrastructures and complex management tasks (*e.g.*, lots of software upgrades) that can exhibit failures which, if not handled correctly, can lead to severe implications. In past outages, failures were often cascaded to other healthy clusters, dependent services went down, manual mistake-prone recovery code had to be quickly written during the outage, and users were frustrated and furious [2, 11, 12, 13, 22, 25, 26].

In many service outages in the past, the service providers believed that they had anticipated the failure scenarios and expected recovery to work. Such incorrect expectations could arise because cloud service deployment is complex and many scenarios might have not been tested. When a service becomes popular, for example, it suddenly must deal with more requests, machines, data, and failures. In other words, the scale of an actual deployment is typically orders of magnitude larger than the scale of a testing framework or a "toy model" [21]. As a result, *there are many unknown real-production scenarios in which a failure recovery might not work*.

The Netflix engineers' quote above sums up our motivation. Many real scenarios cannot be covered in offline testing, and thus a new paradigm has emerged: failures should be deliberately injected in actual deployments [17, 18]. For example, Amazon invented the "GameDay" exercise that injects real failures (*e.g.*, machine failures, power outages) into real production systems, which has proven effective in many ways: the exercise drives the identification and mitigation of risks and impacts from failures, builds confidence in recovering systems after failures and under stress, and allows the organization to schedule failures instead of waiting for unplanned, unscheduled failures.

Despite these benefits, production failure testing unfortunately remains uncommon outside of the few large organizations with the engineering resources, operational expertise, and managerial discipline to execute it. Also, to the best of our knowledge, no existing literature has laid out the concept, design space, and challenges of this new paradigm. As a result, cloud-service newcomers cannot reap the benefits of this new approach.

In this paper, we attempt to "commoditize" this new paradigm in a new type of service: Failure as a Service (FaaS), a cloud service for performing large-scale, online failure drills. The principle

Service Outage	Root Event → *"Supposedly Tolerable" Failure* → **Incorrect Recovery** → Major Outage
EBS [2]	Network misconfiguration → *Huge nodes partitioning* → **Re-mirroring storm** → many clusters collapsed
Gmail [11]	Upgrade event → *Some servers offline* → **Bad request routing** → All routing servers went down
Gmail [12]	Maintenance → *A datacenter (DC) offline* → **Bad cross-DC re-mirroring** → Many DCs went down
App Eng. [13]	Power failure → *25% machines of a DC offline* → **Bad failover** → All user apps in degraded states
PayPal [22]	Network failure → *Front-end systems offline* → **Late failover** → Global service interruption
Skype [25]	System overload → *30% supernodes failed* → **Positive feedback loop** → Almost all supernodes failed
Wikipedia [26]	Overheated DC → *The DC offline* → **Broken failover mechanism** → Global outage

Table 1: **Recent major outages of popular cloud services than lasted for hours to days.**

we promote here is to make failure a first-class citizen for cloud services. That is, rather than waiting for unexpected failures to happen, cloud services should run failure drills from time to time. When a drill finds a recovery problem, the drill can be cancelled, and major outage can be prevented.

In the following sections, we motivate our work further (§2), elaborate the concept of FaaS (§3), present the challenges of commoditizing this concept (§4), describe our design strategies (§5), and finally conclude (§7).

2 Extended Motivation

To strongly motivate our proposal, here we recap some major service outages that happened in the last two years and then discuss the lessons learned.

2.1 Major Service Outages

In September 2009, a fraction of Gmail's servers were taken offline for a routine upgrade (which is commonly done and "should be fine"). But due to some recent changes on the re-routing code, combined with the request load at that time, several routing servers refused to serve the extra load. This transferred the load to other servers, which then caused a ripple of overloaded servers, resulting in a global outage [11].

In February 2010, Google experienced a power failure that affected 25% of the machines in a datacenter. Google App Engine, designed to quickly recover from this type of failure, failed to do so this time. According to the report, the engineers "failed to plan" for this particular case [13].

In December 2010, 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", which led to a near complete outage [25].

2.2 Lessons Learned

From these outages and many others, it is obvious that many common events could lead to failures, events such as software upgrades, power failures, and increased load. Therefore, many cloud services already employ automated failure recovery protocols. However, they do not always work as expected, leading to severe implications. We illustrate further this problem by listing more outages in Table 1. All these outages began with root events that led to supposedly tolerable failures (in italic), however, the broken recovery protocols (in bold) gave rise to major outages.

One lesson learned here is that the correctness of recovery depends on the deployment scenarios (the load, executing code, environment, etc.). However, it is hard (or perhaps impossible) to test recovery for all possible scenarios, especially for large-scale deployments. As a result, the limit of a large-scale failure recovery is often unknown. In Google's case, the recovery did not work when 25% of the machines in the datacenter failed. Similarly in Skype's case when 30% of the supernodes died. What often happens here is that recovery tries to achieve high-availability (*e.g.*, by quickly re-mirroring lost data replicas or re-routing load to other live machines [11, 13, 25]) rather than sacrificing some availability (*e.g.*, returning errors to users). Such "greedy" recovery would then lead to major outages. On the other hand, "lazy" recovery has documented disadvantages [10, 22].

In summary, the outages we listed above are high-profile outages. We believe that many other cloud services face similar problems. A cloud service ideally should ensure that its recovery strategies work correctly (*e.g.*, not too aggressive/slow) for its deployment scenarios. We believe performing failure drills is a promising solution that can catch the type of recovery problems we mentioned above. In the following section, we elaborate our proposal.

3 Failure as a Service

Failure as a Service (FaaS) is a new type of cloud service for performing large-scale, online failure drills. The goal is analogous to that of fire drills. That is, before experiencing unexpected failure scenarios, a cloud service could perform failure drills from time to time to find out the real-deployment scenarios in which its recovery does not work. Below, we discuss our proposed service architecture (Figure 1), a use case of FaaS, the supported failure modes, and the three important characteristics of our proposed failure drills (large-scale, online, and a service).

• **Service Architecture:** Figure 1 illustrates our FaaS architecture. Target service is a cloud service (*e.g.*, HadoopFS) that exercises failure drills and runs on VMs. The monitoring service collects runtime information from the target service (*e.g.*, #user requests, #under-replicated files). This information are used for writing failure-drill specifications executed by the FaaS controller. The controller runs on multiple machines for fault-tolerance and sends drill commands to FaaS agents running on the same VMs as the target service. An agent could enable/disable virtual failures supported by FaaS-enabled VMs.

• **Use Case:** An example use case of FaaS is for maintenance events [11, 12]. Before completely taking some machines offline, a service provider could use FaaS to virtually disconnect the machines and monitor the recovery. If everything works as expected (*e.g.*, smooth failover, no aggressive re-mirroring, no violated SLAs), the provider could proceed with the real maintenance event with better confidence of success. But, if the drill finds a recovery problem, the drill can be cancelled, and ideally FaaS should quickly restore the system to a good state (*i.e.*, a safety challenge). Later, we discuss further this challenge (§4) and our solutions (§5).

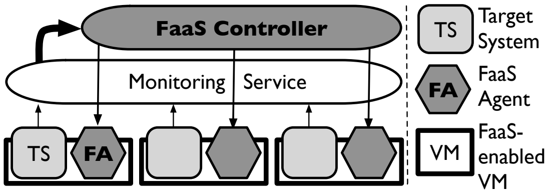

Figure 1: **FaaS Architecture.**

• **Failure Modes:** The goal of FaaS is to complement, not replace, offline testing. Thus, we focus on supporting failure modes that could happen simultaneously on a large number of components, failures such as machine crashes, network failures (packet loss/delay at node/rack/datacenter levels), disk failures (crash/slowdowns), and CPU overload.

All these failure modes represent some form of resource unavailability common in large-scale deployments and can bring severe implications if not anticipated. For example, Facebook engineers found, in real production, bugs in the Linux stack that caused 1GB NIC Card to transmit only 1 KB/sec, causing a chain reaction upstream in such a way that made 100-node cluster to be non-available for all practical purposes.

With these failure modes, we expect FaaS to uncover recovery issues initially caused by some resource unavailability. Currently, we do not intend to support I/O reorderings, Byzantine failures, system-specific API failures, and failures caused by software bugs.

• **Why Large-Scale?** Cloud service engineers already exercise many failure scenarios in offline testing, however, 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, Skype, with hundreds of millions of users worldwide, only emulates thousands of users in testing [25]. Facebook uses 100-machine testing clusters that mimic production data and workload of 3000-machine production clusters. Thus, scale is one main reason why recovery problems are still found in production.

• **Why Online?** Covering many large-scale failure scenarios in offline testing is infeasible for three reasons. First, such testing will incur big resource and monetary costs for generating large data and workloads and for powering up a large number of machines. Second, there are many unknown differences between testing setups and real deployments (*e.g.*, #machines, network topology). Although simulation is an option, problems found in simulations receive less attention than those found in production; as Netflix engineers said, they learn more from "real scale, not toy models" [21]. Finally, there are lots of complex dependencies between cloud services today. Cascading failures across services might not be observable in offline testing. For these reasons, there have been suggestions and new practices to deliberately inject failures in production [17, 18] (however, as we emphasized before, these work do not discuss in detail the concept, design space, and challenges).

• **Why as a Service?** Any cloud service that promises fault-tolerance must ensure correct recovery. However, young companies often put more emphasis on feature development and commit less resources for recovery testing. Yet, the risk is high; 60% of companies that lose customer data or cannot resume operations in 10 days are likely to shut down [8]. Thus, making failure drills an easy-to-use service will benefit many cloud providers.

Beyond functioning as online failure drills, we believe FaaS would play a big role in future cloud benchmarks. Recent work has laid out the important elements of cloud benchmarks [3, 9], and one of them is failure – cloud benchmarks should measure how various failures affect the benchmarked systems. However, available cloud benchmarks focus on performance and elasticity metrics. With FaaS, future cloud benchmarks will have the capability to produce availability metrics as well.

4 Challenges

This section elaborates the challenges of commoditizing FaaS. One major challenge is to *learn about failure implications without suffering through them*.

• **Safety:** The first major challenge is safety; we would like to check if customer data could be lost without really losing it, or if SLAs could be violated without violating them for a long period of time. Facebook currently exercises failures only in testing clusters, but not in real deployments, unless this safety challenge is properly addressed. In a related story, at a major venue, there was a "spike drill" where cloud services were challenged to deal with real massive increases in load [24]. However, not many participated because of the fear that they would fail, and indeed, one company that participated had to abort the drill. To emphasize our point here, FaaS should be a system that not just merely injects failures, but also comes with mechanisms that address this safety challenge.

This safety challenge gets harder due to the likelihood of having real failures *before* a drill. For example, due to some previous failure, there are single replicas still being re-mirrored. In such a scenario, the drill must be done more carefully (*e.g.*, by excluding from the drill the machines that store the single replicas). Increasing the challenge further is the possibility of real failures to occur *during* a drill. For example, a real network failure could partition the "commando" of the drill (the server that makes failure decisions) and the agents (*e.g.*, the machines that receive drill commands such as a "go-back-alive" command). Such a situation could unsafely place the agents in "limbo" for an indefinite period of time.

• **Performance:** To address the safety challenge, traditional mechanisms such as snapshots, VM forking, and process pair are considered heavyweight. On the other hand, FaaS safety mechanisms must not impose too much overhead.

• **Monetary Cost:** In the world of utility computing, performing large-scale failure drills will cost extra money; failure drills consume extra resources such as network bandwidth, storage space, and processing power. To estimate the cost, let's assume a machine costs $0.25/hour, raw disk storage $0.03/GB, and network bandwidth $0.01/GB across different regions (*e.g.*, west to east coast) and free within each [1]. A drill that shuts down 100 stateless servers (*e.g.*, front-end web servers) for 3 hours would only cost roughly $75 (if the recovery simply spawns another 100 machines). A drill that re-mirrors 200 TB of data (kills 100 storage servers with 2 TB/server) would cost $8000. It is unclear if cloud service providers want to incur this monetary cost.

• **Specifications and Integrations:** Every failure drill needs a specification that describes the conditions upon which the drill should start and stop, the failed resources, the injected failure types, and so on. For example, a drill can be specified to run during a peak load, virtually disconnect 100 storage servers, stop when the overall performance drops, and cancel halfway if the recovery looks correct. Drill specifications could be complex, however it is critical to have correct specifications; imagine a flawed specification that never stops the drill. At Facebook, the engineers are highly careful in turning up the scale of failure drills slowly and steadily (in testing clusters). This is only done manually so far; ideally, drill specifications should be statically checked.

Writing drill specifications requires rich information from other service components. To know if a drill is progressing well, FaaS needs information from monitoring service. To inject different failure types, FaaS must control local operating systems (or VMs). Cloud services that use FaaS might need to be modified to be drill aware. All these bring integration challenges.

• **Coverage:** Although all deployment scenarios cannot be covered, some coverage metrics are needed to express different specifications that explore different coverage. Unlike traditional metrics such as code/path coverage, FaaS needs new metrics such as failure and scale coverage. For example, one might write a set of specifications that kill different portions of available machines (*e.g.*, 1%, 10%) at different load (*e.g.*, day, night). Such coverage metrics also provide a means for FaaS users to measure the robustness of their systems; when a high-profile outage happens, many cloud service providers typically ask the question “Can my system survive the same failure scenario?”.

5 Design

In this section, we present our FaaS design plan and principles. We mainly focus on addressing the safety and monetary challenge.

• **Safe Failure Drills:** Regarding the safety and performance challenges, achieving all kinds of safety with a minimum overhead seems infeasible. Thus, we only guarantee two kinds of safety: (1) the service under drill can be rapidly restored to a good state after a bad drill and (2) there is no data loss. However, user-facing requests are not completely masked from the failure implications (*e.g.*, many requests experience increased latencies). To ensure the first safety, we are currently synthesizing VM forking and the decoupling of availability-related metadata and other data management (*i.e.*, we separate healthy and drill processes). Consider a scenario of a 1000-node system that experiences a remirroring storm that causes only 500 nodes to be connected. By forking the processes before the drill, the healthy processes (fully connected nodes) can quickly take over the drill processes (partially connected nodes) after the drill is cancelled. In order to observe the real failure implications, workloads are still directed to the drill processes. The performance overhead is small because the healthy processes do not need to process the workloads (*i.e.*, no double computation). There are more details not discussed here (*e.g.*, how healthy processes keep their metadata up-to-date without processing the workloads).

We also perform VM forking via failure taint propagation; imagine a drill that fails only 1% of all available nodes which only changes the states of the other 10%. In this case, there is no need to fork all the processes of all the nodes. To ensure that no data is lost, we mix-and-match classic techniques such as snapshotting and versioning that are suitable for failure-drill purposes. These mechanisms will be enabled only during the drill.

Beyond the main strategies described above, there are many more strategies that must be carefully designed to ensure safe failure drills. Here, we describe some. FaaS agents should have the capability to autonomously cancel a drill; imagine a case of network partitioning between the controller and the agents. FaaS-enabled VMs should employ just-in-time versioning file system; running versioning all the time might be expensive, but it should be enabled before a drill starts in order to prevent buggy recovery from accidentally deleting or overwriting files. The overall system must be drill aware; if a drill is ongoing, unsafe external events such as configuration change and software upgrade should be disabled. The FaaS controller should support fine-grained specifications; a specification can inject a large-scale failure (*e.g.*, disconnect a datacenter), but it should also be able to exclude some specific nodes from a drill (*e.g.*, the machines that store single replicas). FaaS-enabled VMs should enable fine-grained failures (*e.g.*, virtually disconnect specific connections); as shown in Figure 1, a drill can disconnect connections between the target service nodes, but should not do so between the agents. Whenever possible, the FaaS controller should be placed near "important" nodes; placing the controller near master nodes, for example, will enable early decision making (*e.g.*, cancel the drill) because master nodes have more important information than slave nodes.

• **Cheap Failure Drills:** One effective solution in addressing the cost challenge is by hybrid of live experimentation and simulation. For example, to cut the storage cost, some file writes could be simulated (*e.g.*, re-mirrored data is not stored on disk). To cut the network cost, some network transfer could be simulated (*e.g.*, re-mirrored data is not sent over the network). However, to observe the real impact of recovery, such simulation must be highly accurate. Accurately simulating the network of hundreds/thousands of machines is a difficult problem. Fortunately, network data transfer is cheap; it's usually free within a region and only costs $0.01/GB across regions [1]. Storage is expensive, but fortunately easier to simulate. For example, in the context of large-data re-mirroring, simulating mostly sequential writes is straightforward compared to simulating random disk accesses. Thus, simulating storage will greatly cut the overall cost but without sacrificing too much accuracy; the network does not have to be simulated.

This strategy however poses another problem: the performance of applications that need multiple data replicas (*e.g.*, MapReduce jobs) will not see any improvement during the drill because the recovered data is never written to the disk. To address this, we plan to explore strategies that could predict which data accessed by the applications during the drill. For these data, the storage simulation is disabled.

To further reduce the overall cost, we could write smarter drill specifications. For example, a specification can cancel a drill if the recovery looks correct half-way in the process (*e.g.*, #under-replicated files keep going down).

6 Related Work

In order to ensure good reliability, large-scale production systems are equipped with core services such as repair, provisioning, monitoring services [19, 23], and workload spikes generators [4]. We believe one missing piece is a failure service, and FaaS could fill this void. The implications of large-scale failures have been studied before in the context of P2P computing [7, 14, 16, 20]. However, these studies were mostly done in simulation or small-scale emulation and do not propose ideas presented in this paper. We also note that FaaS is different from Testing as a Service (TaaS); in TaaS, users upload their software to a cloud service to be analyzed [5]. FATE also only provides offline failure testing service [15]. DiCE is a framework that performs live/online model checking [6] Finally, existing online failure-injection frameworks (*e.g.*, Netflix's Chaos Monkey, Amazon GameDay, DevOps GameDay) motivated our proposal, but they are not generally available as a service and do not fully address the challenges (in §4) and opportunities of Failure as a Service.

7 Conclusion

Failure is part of "cloud's daily life". FaaS enables cloud services to routinely exercise large-scale failures online, which will strengthen individual, organizational, and cultural ability to anticipate, mitigate, respond to, and recover from failures.

8 Acknowledgments

This material is based upon work supported by the NSF/CRA Computing Innovation Fellowship and the National Science Foundation under grant numbers CCF-1016924 and CCF-1017073. We also thank Peter Alvaro for his feedback and comments.

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Amazon EC2. http://aws.amazon.com/ec2.

[2] Amazon.com. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. April 2011.

[3] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather tomorrow? Towards a Benchmark for the Cloud. In *Proceedings of the 2nd International Workshop on Testing Database Systems (DBTest '09)*, Providence, Rhode Island, June 2009.

[4] Peter Bodik, Armando Fox, Michael Franklin, Michael Jordan, and David Patterson. Characterizing, Modeling, and Generating Workload Spikes for Stateful Services. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, June 2010.

[5] George Candea, Stefan Bucur, and Cristian Zamfir. Automated Software Testing as a Service. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, June 2010.

[6] Marco Canini, Vojin Jovanović, Daniele Venzano, Boris Spasojević, Olivier Crameri, and Dejan Kostić. Toward Online Testing of Federated and Heterogeneous Distributed Systems. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC '11)*, Portland, Oregon, June 2011.

[7] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, M. Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006.

[8] Boston Computing. Data Loss Statistics. http://www.bostoncomputing.net/consultation/databackup/statistics/.

[9] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 2010 ACM Symposium on Cloud Computing (SoCC '10)*, Indianapolis, Indiana, June 2010.

[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

[11] GmailBlog. More on today's Gmail issue. http://gmailblog.blogspot.com/2009/09/more-on-todays-gmail-issue.html, September 2009.

[12] GmailBlog. Update on today's Gmail outage. http://gmailblog.blogspot.com/2009/02/update-on-todays-gmail-outage.html, February 2009.

[13] Google. Post-mortem for February 24th, 2010 outage. https://groups.google.com/group/google-appengine/browse_thread/thread/a7640a2743922dcf, February 2010.

[14] P. Krishna Gummadi, Ramakrishna Gummadi, Steven D. Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proceedings of SIGCOMM '03*, Karlsruhe, Germany, August 2003.

[15] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Koushik Sen. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, Massachusetts, March 2011.

[16] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, Massachusetts, May 2005.

[17] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009.

[18] Todd Hoff. Netflix: Continually Test by Failing Servers with Chaos Monkey. http://highscalability.com/display/Search?searchQuery=netflix+chaos+monkey, December 2010.

[19] Michael Isard. Autopilot: Automatic Data Center Management. *Operating Systems Review*, 41(2), April 2007.

[20] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, California, May 2006.

[21] Netflix. 5 Lessons Weve Learned Using AWS. http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html, December 2010.

[22] PayPal. Details on PayPals Site Outage today. https://www.thepaypalblog.com/2010/10/details-on-paypals-site-outage-today, October 2010.

[23] Ari Rabkin, Andy Konwinski, Mac Yang, Jerome Boulon, Runping Qi, and Eric Yang. Chukwa: a large-scale monitoring system. In *Cloud Computing and Its Applications 2008 (CCA '08)*, Chicago, IL, October 2008.

[24] Scott Ruthfield, Chris Bissell, and Ryan Nelson. Spike Night. http://velocityconf.com/velocity2009/public/schedule/detail/10207, June 2009.

[25] Skype.com. CIO update: Post-mortem on the Skype outage (December 2010). http://blogs.skype.com/en/2010/12/cio_update.html, December 2010.

[26] WikiMedia.com. Wikimedia Technical Blog: Global Outage (cooling failure and DNS). http://techblog.wikimedia.org/2010/03/global-outage-cooling-failure-and-dns, March 2010.