Check for updates

Too Big NOT to Fail

PAT HELLAND, SIMON WEAVER, AND ED HARRIS

**EMBRACE FAILURE
SO IT DOESN'T
EMBRACE YOU**

Web-scale computing means running hundreds of thousands of servers. It requires a fundamentally different approach from smaller environments. Consistent server hardware and data-center plans, as well as consistent and simple configurations, are essential. Everything is designed to expect and embrace failure without human intervention. Operations must be largely autonomous. Software must assume crashes. Development, test, and deployment must have integrated and automated solutions.

This article offers a brief survey of many of the techniques so well known to those working at web scale and yet frequently surprising to others.

Web-scale infrastructure implies LOTS of servers working together—often tens or hundreds of thousands

of servers all working toward the same goal. How can the complexity of these environments be managed? How can commonality and simplicity be introduced?

WHAT'S WEB-SCALE INFRASTRUCTURE, ANYWAY?
A lot of effort goes in to achieving uniform and fungible hardware resources in Web-scale data centers. If you can have just one kind of server with the same CPU, same DRAM, same storage, and the same network capacity, then any server is as good as the next server. When all the servers are the same, there is a single pool of spares and a single resource to allocate.

You want to treat your hardware resources like a bag of nails. Any single nail doesn't matter very much. It's the aggregate number of useful nails that gets the house built. Typically, you buy too many nails, expecting a certain failure rate. When a nail bends or breaks, you don't get too emotional about it.

Similarly, when a server in the data center breaks, you can't get emotional. It happens all the time.

Variety leads to complexity. Complexity leads to unpredictability and a lack of web-site availability.

The law of large numbers is probably true
The law of large numbers says that individual pieces may be unreliable but the aggregate expected failure rate is predictable. While you can't know the exact outcome of the toss of a pair of dice, you can know the expected rate over time. The more tosses you have, the better the confidence in the average.

Similarly, in a large data center, the more servers you run, the easier it is to plan for expected failures. The larger the set of servers, the more likely something is broken. Figure 1 shows a typical server failure rate.

It's predictability that matters, not reliability. Stuff can fail… Stuff *will* fail. With tens of thousands of servers, lots of stuff fails every day! You just need to predict how often.

HARDWARE AT WEB SCALE

Typical web-scale data centers think about hardware differently from typical IT shops. They avoid variety and stick to common hardware as often as possible. It's assumed that each piece of hardware may very well fail and the software running on it needs to work successfully when that happens. Furthermore, the inexorable pace of improvement means it's not cost effective to keep hardware too long. The new stuff will offer more "bang for the buck" and for the electricity it consumes.

FIGURE 1: **TYPICAL SERVER FAILURE RATE**

Assume 4 SATA Disks per Server
Assume 4% disk failure rate per year
4 disks * 4% per disk means 16% of the servers fail from disk
Assume 4% miscellaneous failures (e.g. power supplies, etc)
20 % of servers fail each year
1 in 1825 servers fails each day   (1825 = 5 * 365)
50,000 servers ➜ expect 27 servers to fail each day

Bespoke is baroque

When something is *bespoke*, you tweak and modify it until it is perfect. In some environments, systems and servers are designed with their own special configuration.

Baroque is an architectural style with a lot of ornate detail and tremendous variety. In a large-scale environment, bespoke becomes baroque. The individual details of the various types of servers become overwhelming. The aggregate system with its large number of server types is full of detail and complexity.

Attack of the clones

In a typical data center, you pick a standard server configuration and insist everyone use the same type. Just like Henry Ford's Model-T, you can have any color you like as long as it's black.

With one SKU (stock keeping unit) to order, you gain huge leverage with vendors in buying servers. In addition, there's a single pool of spares for that SKU.

Now, there are some exceptions. Each company is phasing in a new server type while phasing out an old one. Also, it's common to have a very few special servers—such as one for compute loads and one for storage.

Still, tightly controlling the variety of server types is essential.

The short life of hardware in the data center

Messing with stuff in the data center causes problems. Upgrading servers can cause inconsistencies. Just don't do it! When repairing, you really only want to replace the server with an identical spare and then repave its

software. Maybe the broken server can be fixed and become a spare.

Servers and other gear provide less value over time. New servers offer more computation and storage for the same form factor and same electricity. The value for the electrical cost diminishes.

Data-center hardware is typically decommissioned and discarded (or returned to its lessor) after three years. It's just not worth keeping.

> *Datacenter servers and roast beef are*
> *worth a lot less after a few years.*

That means there's a lot of pressure to place new servers into production quickly. Say it takes two months to commission, activate, and load data into new servers. In addition, it may take one month to decommission the servers and get the data out of them. That's three months out of a total three-year lifetime not productively used. The life cycle of servers is a big financial concern.

OPERATIONS AT WEB SCALE
Operations at web scale are very different from operations at smaller scale. It's not practical to be hands on. This leads to autonomous data-center management.

People don't scale—dermatological issues notwithstanding

Say your operations staff can manage 100 servers per person. This is very typical in a DevOps environment. Each server needs attention to validate state, handle failures,

and perform repairs. Frequently, the servers come in many flavors and you want to optimize the hardware for each server. Does everyone understand all the server types? What are their different operational tasks?

Fifty thousand servers at 100 servers per person require 500 operators. This gets out of hand very quickly as you scale.

Zen and the art of data-center maintenance

To support web-scale data centers, we've had to evolve from Ops to DevOps to NoOps, as detailed in figure 2. Historically, with manual operations, people not only decided what to do, but also did all the actual work required to operate the servers. DevOps is a huge step forward as it automates the grunt work of operations. Still, this is not adequate when scaling to tens of thousands of

FIGURE 2: **OPS, DEVOPS, AND NO OPS**

TASKS	MANUAL OPERATIONS "OPS"	AUTOMATED OPERATIONS "DEVOPS"	AUTONOMOUS OPERATIONS "NO OPS"
Who sets the goals?	Human	Human	Human
Who decides when to start?	Human	Human	Machine
Who adjudicates priorities?	Human	Human	Machine
Who does the work?	Human	Machine	Machine
Who generates validation data?	Human	Machine	Machine
Who interprets validation data?	Human	Human	Machine
Who handles failures?	Human	Human	Machine
Who handles exceptions?	Human	Human	Human

servers. In a NoOps or autonomously managed system, the workflow and control over the operational tasks are also automated.

SOFTWARE AT WEB SCALE
Software must embrace failures with pools of stateless servers and special storage servers designed to cope with the loss of replicas.

Stateless servers and whack-a-mole

Stateless servers accept requests, may read data from other servers, may write data to other servers, and then return results. When a stateless server dies, its work must be restarted. Stateless servers are designed to fail.

Stateless servers must be idempotent. Restarting the work must still give the correct answer. Learning about idempotent behavior is an essential part of large-scale systems. Idempotence is not that hard!

Frequently, stateless servers run as a pool of servers with the number increasing or decreasing as the demand fluctuates. When a server fails, the demand increases on its siblings, and that likely will cause a replacement to pop back to life. Just like the arcade game whack-a-mole, as soon as you hit one, another one pops up.

To each according to its need

Concurrent requests for a stateless service need assignment to servers. Load balancing requests across a pool of servers is much like spraying work.

When requests to servers are taking longer than hoped, it is likely because there's a queue waiting for the individual

servers. If you're spraying work across more servers, then the per-server queue is shorter. This will result in a faster response.

Adding servers to a server pool usually lowers the response time for requests. Removing servers reclaims resources. Given a response-time goal, this can be done by an automated robot.

### Avoiding memory loss from traumatic server injury

Most distributed storage systems keep each piece of data on three separate servers in three separate racks in a data center. The choice of three replicas is a function of the durability goals, the MTBF (mean time between failures), and the MTTR (mean time to repair).

```
Availability = MTBF / (MTBF + MTTR)
```

Since we assume one in five servers fail every year, our MTBF (Mean Time Between Failures) is relatively short. To improve availability, we either need a longer MTBF or a short MTTR (Mean Time To Repair). By shortening our MTTR, we can dramatically improve our availability and data durability.

Assume the data contained in each server is cut into pieces and the pieces have their additional replicas on many different servers, as shown in figure 3. For example, the data on server A is cut into 100 pieces. Each of those pieces has its secondary and tertiary replicas on different servers, perhaps as many as 200 total in addition to server A. If server A fails, the other 100 secondary servers will try to find a new place on potentially yet another 100

FIGURE 3: **DATA REPLICATION**
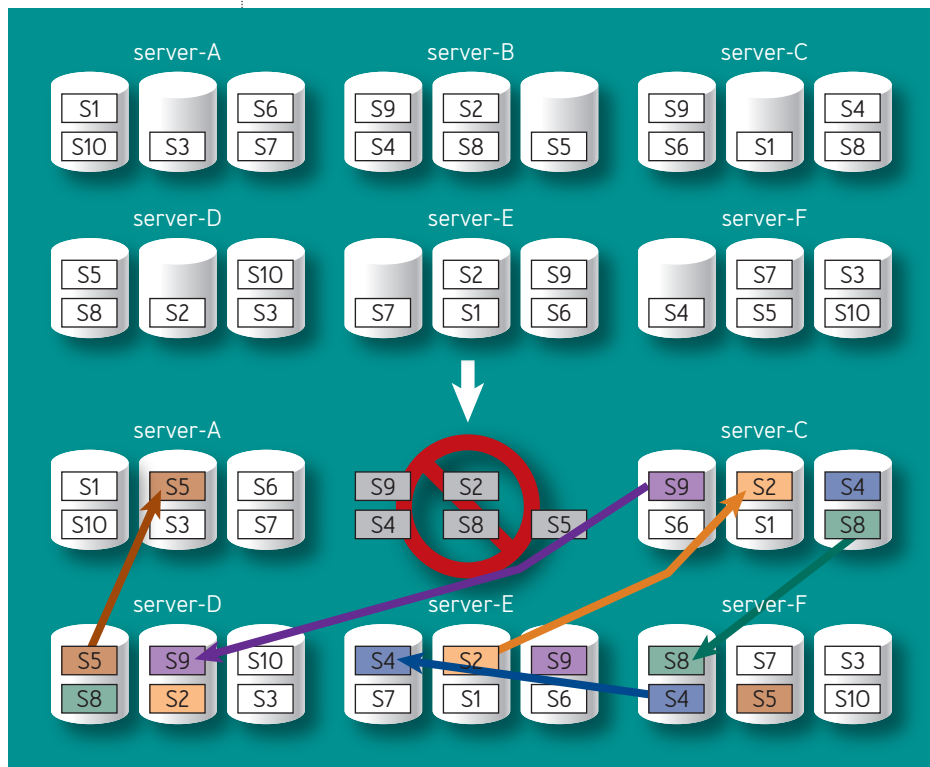
servers. In the limit, this parallelism can reduce the MTTR by 100 times.

Notice in figure 3 that each slot of data in server-B (S9, S2, S4, S8, and S5) has two other replicas on different servers. If server-B fails, each of these slots is replicated onto a new third server. The placement of the new third replica preserves the guarantee of each replica lives in a separate server.

This approach is tried and true with GFS (Google File System),[1] HDFS (Hadoop Distributed File System),[3] Microsoft Bing's Cosmos distributed file system,[2] and others leveraging this technique.

Large data centers need to consider two essential aspects of this approach. First, software-driven robotic recovery is essential for high availability. Humans take too long and would be overwhelmed by managing the failures. Second, the larger the cluster, the more storage servers are possible. The more storage servers, the smaller the pieces smeared around the cluster and the faster the recovery time.

DEVELOPMENT, TEST, AND DEPLOYMENT AT WEB SCALE
Development and testing in large-scale web environments are best suited for sharing resources with production. Successfully developing, testing, and deploying software presents many challenges. To top it off, you are pretty much constantly working to keep up with the changing environment.

The isolation ward
It's important to concentrate data-center resources. Separating data centers for development and test from production may be tempting, but that ends up creating problems. Managing demand and resources is difficult when they are separate. It is also hard to ensure that production is in sync with dev/test and that you are testing what you will run in production.

Developing and testing using production data centers means sharing resources. You must separate the

resources using containers or VMs (virtual machines). Your customers will be using the same servers as your testers so you must isolate the production data. In general, development and test personnel must not have access to customer data to comply with the security team.

While ensuring safe isolation of workloads presents many challenges, the benefits outweigh the hassles. You don't need dedicated data centers for dev/test, and resources can ebb and flow.

Sign-off, rollout, canaries, and rollback

The management of software change in a large-scale environment has complexities, too. Formalized approvals, rollout via a secure path, automated watchdogs looking for problems, and automated rollback are all essential.

Formal approvals involve release rules. There will be code review and automated test suites. The official signoff typically involves at least two people.

Rollout to production includes code signing whereby a cryptographic signature is created to verify the integrity of the software. The software is released to the needed servers. Sometimes, tens of thousands of nodes may be receiving the new version. Each node is told the code signature, and it verifies that the correct bits are there. The old version of the software will be kept side by side with the new one.

A few servers are assigned to be *canaries* that will try the new version before all the others. Like the little birds taken into underground mines to check for dangerous gases (the canaries would die of the gas before the miners, who would then skedaddle to safety), in software releases,

the automated robot tries a few servers first, and only when successful, rolls out more.

Rollback is the automated mechanism to undo the deployment of a new version. Each server keeps the old version of the software and can pop back to it when directed.

CONCLUSION
Running large data centers requires some fundamental changes in approach. Everything must be simpler, automated, and expect failure.

A simple model for server and service behavior
Web-scale systems run on three important premises:
➡ Expect failure. Any component may fail and the system continues automatically without human intervention.
➡ Minimal levers. The underlying system provides support for deployment and rollback. When a server is sick, it's better to kill the whole server than deal with partial failures.
➡ Software control. If it can't be controlled by software, don't let it happen. Use version control for everything with human-readable configuration.

Reliability is in the service, not the servers!

Embrace failure so it doesn't embrace you!
Running hundreds of thousands of servers requires a different approach. It requires consistent hardware (servers and network) with minimal variety. Data centers

must have simple and predictable configurations. They must embrace failure and automatically place services onto the hardware as needed.

Operations must evolve from manual to automated to autonomous. Humans should set the goals and handle major exceptions. The system does the rest.

Software must embrace failures with pools of stateless servers and special storage servers designed to cope with the loss of replicas.

Integrated development, test, and deployment are built deeply into the system with controlled and automated deployment of software.

References

1. Ghemawat, S., Gobioff, H., Leung, S.-T. 2003. The Google file system. Proceedings of the 19th ACM Symposium on Operating Systems Principles: 29-43.
2. Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, Jingren Zhou 2008. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. Proceedings of ACM VLDB 2008. http://www.vldb.org/pvldb/1/1454166.pdf
3. Shvachko, H., Kuang, H., Radia, S., Chansler, R. 2010. Hadoop Distributed File System. Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies: 1-10.

Pat Helland *has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978.*

For recreation, he occasionally writes technical papers. He currently works at Salesforce.

Ed Harris *leads the Infrastructure Compute team at Salesforce.com, building the compute, network, and storage substrate for the world's most trusted enterprise cloud. Prior to Salesforce, Ed worked in the Bing Infrastructure at Microsoft, working on the Cosmos big data service.*

Simon Weaver *has been devising and building distributed and autonomous systems for 18 years solving problems in the fields of big data, lights-out infrastructure, and artificial intelligence. Simon currently works at Salesforce.*