CORNELIA DAVIS

Realizing Software Reliability in the Face of Infrastructure Instability

Cornelia Davis, Pivotal Software

Cloud computing has brought with it the utilization of off the shelf, commodity hardware that has higher failure rates than the systems that have been used in enterprises for the last several decades. Coupled with increasingly complex, highly-distributed, constantly-changing data center environments that can no longer

 2325-6095/17/$33.00 © 2017 IEEE

be treated as deterministic systems, this forces us to change the way that we view the stability of that infrastructure. Our aim is to provide digital solutions that remain stable in the face of this infrastructure uncertainty and we achieve this by utilizing specific design patterns and operational practices. At the core of the new way of working is a philosophical view that change is the rule rather than the exception. US entertainment company Netflix has fully embraced this new mindset and this served them well during a major outage experienced by their cloud infrastructure provider, Amazon Web Services (AWS). In this piece, we study how Netflix was able to avoid any significant impact while many other well established, technically savvy AWS users were not.

In the 1980s the computer industry experienced a massive transformation when core software system architectures changed from being mainframe-based to client-server. This shift changed virtually everything in software from the hardware, to software designs, to the practices for development and operation of that software.

Today we are in the midst of the next radical change as the industry moves from client-server to cloud-native. Sometimes called the third platform, cloud-native is characterized by highly dynamic systems, where every element, from the hardware and operating system, to networks and software deployments, are in constant flux. Whereas for second platform systems (client-server), software was written with an expectation that the systems that it executed on were quite stable, in the new world software must be written, deployed, and managed in a manner that anticipates change. That is, the software that runs on a highly distributed, constantly changing infrastructure must have zero downtime even while the lower layers are shifting about. In fact, applications must have zero downtime even while the application itself is changing, due either to an upgrade being performed, or to the application itself experiencing trouble (there's a bug!).

Just as the shift from first platform (mainframe) to second (client-server) changed everything about the way that software is constructed and managed, so does the shift to third platform. Software practitioners must learn a whole new set of design patterns as well as master new software engineering and management tools and methodologies to remain effective. Ultimately our aim is to provide reliable digital solutions even while the infrastructure they are running on is unstable.

It's not Amazon's Fault

On Sunday, September 20, 2015 Amazon Web Service (AWS) experienced a significant outage. With an increasing number of companies running mission critical workloads, even their core customer facing services on AWS, such an outage can subsequently result in far reaching system outages. In this instance, Netflix, Airbnb, Nest, IMDb, and more all experienced down time, impacting their customers and ultimately their business's bottom lines. The core outage lasted more than 5 hours (or even beyond, depending on how you count), with even longer AWS customer downtimes before they had their systems fully functional.

If you are Nest, you are paying AWS because you want to focus on creating value for your customers, not on infrastructure concerns. As a part of the

deal, AWS is responsible for keeping their systems up, enabling you to keep yours functioning just as well. So, if AWS experiences downtime, it would be very easy to blame Amazon for your resulting outage.

But then you would be wrong. Amazon is not to blame for your outage.

Wait! Please hear me out. My assertion gets right to the heart of the matter and will explain one of the key characteristics of cloud-native.

First, let me clear up one thing. I am not suggesting that Amazon and other cloud providers have no responsibility for keeping their systems functioning well—they obviously do. And if a provider does not meet certain service levels, their customers can and will find alternatives.

What I am asserting is that the applications you have running on a cloud-based infrastructure can be more stable than the infrastructure itself. How is that possible? By employing certain architectural patterns in your software designs and by following particular operational practices. At the core of both of these things is a mindset that failure, and more generally, change, is not an exceptional circumstance, but rather something that should be anticipated.

Let's look at an exemplar. Netflix was one of the many companies affected by the AWS outage of September 2015 and with it being, by one measure (the amount of internet bandwidth consumed—36%!), the top Internet site in the US, Netflix downtime affects a lot of people. But Netflix had this to say about the outage:[1]

> Netflix did experience a brief availability blip in the affected Region, but we sidestepped any significant impact because Chaos Kong exercises prepare us for incidents like this. By running experiments on a regular basis that simulate a Regional outage, we were able to identify any systemic weaknesses early and fix them. When US-EAST-1 actually became unavailable, our system was already strong enough to handle a traffic failover.

That's right, they were able to very quickly recover from the AWS outage, being fully functional only minutes after the incident began. That is, Netflix, still running on AWS, was fully functional even as the AWS outage continued.

The above quote holds many hints as to what they've done to achieve this quality of service, and I'd like to study those here. I will start at the end and work back to the start.

Where the Responsibility Lies

In the above passage, Netflix is describing a number of practices that allowed them to *"identify any systemic weaknesses"*. What is most interesting about this specific phrase is that the *"system"* they are referring to is not that of Amazon, but rather their own, and it reflects acknowledgement of something foundational in the cloud—that things are always changing and their software must account for that constant change.

Over the last several decades we've seen ample evidence that an operational model that is predicated on a belief that our environment changes only when we intentionally and knowingly initiate such changes simply doesn't work. Reacting to unexpected changes dominates time spent by IT and traditional software development lifecycle (SDLC) processes that depend on estimates and predictions have proven problematic.

But what, exactly, does it mean to design for failure, or design for change? Comprehensive coverage is beyond the scope of this article, but the following are some of the most foundational ideas:

- Digital solutions are formed as a composition of independent, loosely coupled software components, often called microservices. The software design intentionally builds bulkheads between the components so that a failure in one part does not cascade through the entire system. If there is a problem with displaying images on your ecommerce site, your users should still be able to complete their purchase by providing payment information.
- All apps, or microservices, have many instances deployed so that functionality is maintained, albeit with reduced capacity, when one or more of the instances goes down for any reason (unplanned or planned). Even if the root cause of failure is a very slow memory leak in the app itself, the likelihood that all of 100 deployed instances should crash at the same time is almost zero, so with multiple instances some application capacity is always available. Of course, as soon as we have multiple instances we need a load balancer to distribute traffic across them; there are several different deployment configurations for these, with different tradeoffs.
- State is very deliberately separated from compute by keeping it out of the apps and binding those to stateful backing services. Patterns for redundancy are relatively easy for stateless apps—essentially the aforementioned multiple

instances—and when apps are stateless, platforms such as Kubernetes and Cloud Foundry can auto-recover failed instances. Implementing redundancy in data services is far more complex and solutions are very specific to the unique characteristics of the store. The deliberate separation allows for software developers to leverage common redundancy patterns through large portions of their software, and depend on specialists in the data services to implement the needed redundancy patterns there.

- Cloud-native software is highly distributed and components that might, in the past, have run within the same process now require a network hop when one calls the other. As we know, one of the fallacies of distributed computing is that the network is always stable; to compensate for the instability:
  - A consuming service implements retries, so that when it receives no response from a call it's made to a service, it tries again. If the network was only momentarily unstable, the subsequent retry stands a good chance of succeeding.
  - Because a consumer might fail to receive a response even though the called service received and executed on the request, and will therefore retry the invocation, a producing service must be implemented in such a manner that multiple invocations from the same consumer are not harmful—that is, they must be idemponent.
  - Retries can be dangerous, however; a momentary network outage can cause a retry storm and inadvertent denial of service attack (which ironically was the root cause of the Amazon outage of September 2015).[2] To avoid this, the system overall must also implement things such as circuit breakers.

Reflecting back on these design-for-failure patterns we see that redundancy is key—redundant app instances, redundant copies of data, redundant invocations. But that redundancy must be carefully applied, and that brings us to another excerpt from the Netflix quote.

Understand Where the Failure Domains Are

Turning back to the quote above, in it Netflix mentioned that they had performed activities that *"simulate a Regional outage"*—the word I want to hone in on in this phrase is *"Regional"*. No single system can be guaranteed to be functional 100% of the time and as we've just concluded, redundancy is essential to surviving the inevitable failures. When deploying into these cloud environments, the trick, however, is to ensure that your redundancy spans failure boundaries—if all of your redundant app instances are running on the same host and that host goes down, the redundancy serves no purpose.

The cloud providers themselves (public or private) recognize this and as a result have provided abstractions that map to failure domains. For example, the availability zone (AZ) is an all but ubiquitous concept that usually maps to physical entities such as hardware racks or perhaps a network subnet. The largest cloud providers extend the model further with a concept of "region," which usually corresponds to a data center. Notice that these providers themselves recognize the value of redundancy, even at the infrastructure level, giving their users access to multiple AZs and multiple regions. When infrastructure experiences problems, failures are usually constrained within these abstractions. If a bad firewall rule cuts an entire subnet off from the rest of the data center, other subnets will still be available. If hurricane wipes out all power sources to a data center, it takes that region off line but other regions will be unaffected. And the infrastructure software itself is carefully deployed so that any failures within it will stay within those bounds.

Amazon surfaces both of these abstractions to their users, and this played prominently in the minimal disruption that Netflix experienced in September 2015. Figure 1 depicts both abstractions—AZs and regions—and theorizes how redundant (or not) software deployments might have been made against them before the outage. (Note: I have no knowledge of how any of these companies' digital offerings were deployed into the AWS infrastructure—I show this only as an educational illustration.)

When *"When US-EAST-1 actually became unavailable,"* (Figure 2) it is easy to see how Netflix might have fared far better than other AWS users. This is far from accidental—Netflix deliberately took on the responsibility of building and operating their software to leverage the abstractions available to them. Both their developers and their operators have a deep understanding of what these abstractions are and how to leverage them to achieve the outcomes they seek (usually resilience).

The reason that many other web properties suffered long outages when US-East-1 went down in this particular instance, however, is that despite having access to multiple regions, users hadn't taken advantage of them. Building solutions that leverage more than one region, and establishing the

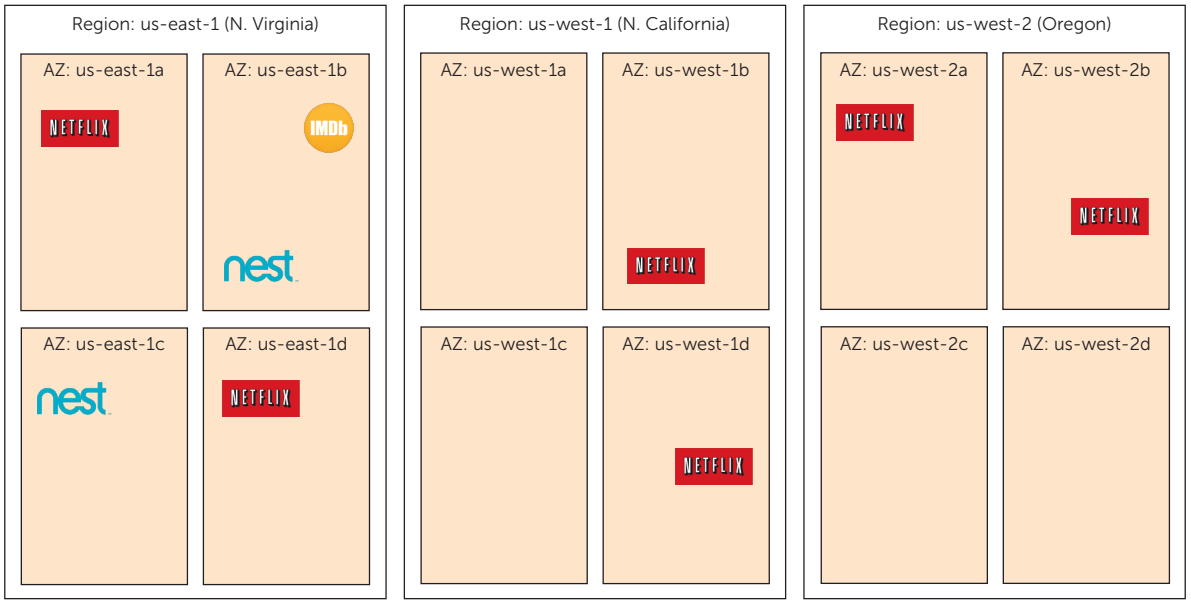

FIGURE 1. Applications deployed onto *Amazon Web Services* may be deployed into a single *availability zone* (AZ) (IMDb), multiple AZs (*Nest*) but only a single region, or in multiple AZs and multiple regions (Netflix). This will provide very different resiliency profiles.

appropriate management practices thereof, is harder than doing so for a single region, and it is therefore work that is often deferred—with dire consequences. But there is a different approach that can take the burden off the application developer and operator.

The most sophisticated platforms can hide what are effectively infrastructure abstractions, AZs, and regions, and implement the required distribution patterns on behalf of their users. Cloud Foundry, for example, can be deployed and configured so that any application deployed into it will have its instances evenly distributed across all availability zones. Of course, the individuals responsible for establishing the platform must be aware of the specific failure domains so that the automatic distribution is done right, but that responsibility is then limited to a small team and empowers many application teams to achieve the resilience they need with no additional cognitive load.

Let me now turn to another word in this excerpt: *"simulate."*

Regularly Experiment (in Production!)

Use of automated testing as a part of the SDLC is becoming fairly pervasive in the industry, with many organizations going so far as to practice test-driven development where they write the tests even before the implementation. And while there is no debate as to the value of a test suite that primarily focuses on expected inputs and outputs, a new practice has emerged in the last few years, and it not a coincidence that the heroes of our story have been at the forefront of innovation in this space.

They said: *"Chaos Kong exercises prepare us for incidents like this. By running experiments on a regular basis that simulate a Regional outage, we were able to identify any systemic weaknesses early and fix them."* Chaos Kong is one part of a suite of tools Netflix has built to test their software's operation in the event of outages of the infrastructure that it runs on. Harkening back to the main point of the previous section, different testing tools that make up the Netflix Simian Army are cast against different failure domains.[3] Chaos Kong simulates region outages. Chaos Gorilla simulates availability zone failures. Chaos Monkey actually kills servers.

Intuitively, it seems rather obvious that regularly exercising tools such as these will help teams find weaknesses in their implementations. But when we begin to put these things in practice we'll be faced with many questions, the most basic being: when will we execute these tests and where? Let's look at the latter first.

Over the last several decades, most IT organizations have established extensive preproduction environments in which they run their tests. The theory is that these "exact" replicas of production

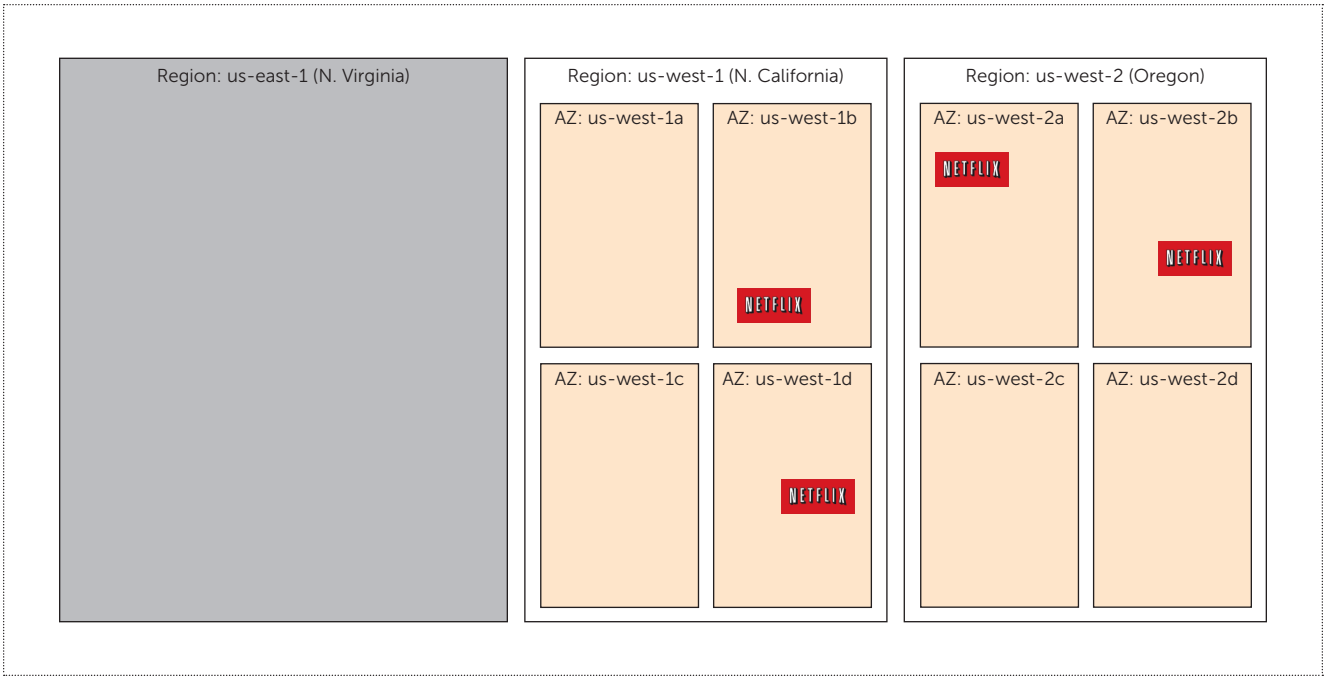

FIGURE 2. If applications are properly architected and deployed, digital solutions can survive even a broad outage such as an entire region.

will allow us to sufficiently test our software so that the unknowns are minimized in production environments. The effectiveness of this approach is debatable however, even in systems of the past—establishing a replica that is sufficiently close to the production environment is difficult and very expensive. But when we move to the cloud the number of moving parts and the frequency with which they experience change has grown by several orders of magnitude making it completely impossible to create that "exact replica of production." In fact, some researchers are formalizing models showing that we can no longer treat software running in cloud and on prem data centers as deterministic systems.[4]

The conclusion we then reach is that in order for the results of Simian Army like tests to be meaningful, we must run them in production environments. Yes, I am suggesting that you experiment in production. Twenty years ago, this would have been considered insane, but the born-in-the-cloud software companies like Facebook, Twitter, Google, and Netflix have shown that it works. We hear stories of a developer who deploys code into production on her first day on the job. There are a number of factors that make this possible, but one is that production environments have safety nets that allow for safe experimentation.

Take, for example, the case we've been studying here. Obviously, Netflix is not actually taking an AWS region down when they run their Chaos Kong exercises; rather they simulate it by intercepting requests routed to an application running in a specific region and returning an error. This is done in software of course, and the safety net allows them to immediately flip a switch and disable the interceptor if things go wrong. Sure, for a matter of seconds or maybe a few minutes some of their users may be disrupted, but a few minutes of downtime during a predicable moment is a good tradeoff for a long outage when totally unprepared. I'll draw your attention to one more word in the above excerpt from the Netflix quote: *"early."* They see the value in finding any *"systemic weakness"* well before they result in unanticipated and difficult to remediate outages.

When is a good time to run experiments then? While my recommendation is not specific to cloud-based software, it is provocative enough that it bears mention in this context. Simulations of catastrophic events should be done during normal working hours. Running tests during off hours, if there even are such periods anymore, will likely miss failure scenarios that only exist when your system is under full load. And, quite pragmatically, you want the many teams responsible for building and operating the software solutions that could be impacted to be readily available in the event of failure.

Conclusion

The events of September 20, 2015 were quite catastrophic for quite a number of well-established, highly trafficked web sites, yet Netflix viewed it as

"a brief availability blip". For them the recovery was rapid and low ceremony and their users experienced little disruption. This is not attributed to luck, rather, Netflix deeply understands what it means to run software in the cloud.

They recognize that the software architectures of the last decades do not work well in the cloud, an environment that is constantly changing and is more distributed than ever before. They've invested in understanding and even defining new cloud patterns and have implemented frameworks and toolsets that allow them to manage the added complexity and even exploit the advantages that extreme distribution can bring.

Moving to the cloud is not merely a matter of deploying existing software into Internet accessible data centers. Cloud-native software is defined by how you compute, not about where you compute.

Writing software for the cloud demands that we treat change as the rule, rather than the exception. It is this that allows us to produce software that runs more reliably than the infrastructure that it is deployed to.

References

1. A. Basiri et al., "Chaos Engineering Upgraded," Sep. 2015; http://techblog.netflix.com/2015/09/chaos-engineering-upgraded.html.
2. C. Babcock, "Amazon Disruption Produces Cloud Outage Spiral," Sep. 2015; http://www.informationweek.com/cloud/infrastructure-as-a-service/amazon-disruption-produces-cloud-outage-spiral/d/d-id/1322279.
3. Y. Izrailevsky and A. Tseitlin, "The Netflix Simian Army," Jul. 2011; https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbab116.
4. S. Dhanagopal, "Converged Virtualized Data Center Networks—Reasons for Non-Deterministic Nature and Possible Solutions," *Proc. IEEE 5th International Conference on Advanced Networks and Telecommunication Systems (ANTS)*, 2011.

CORNELIA DAVIS *is sr. director of technology at Pivotal, where she works on the technology strategy for both Pivotal and for Pivotal customers. Through engagement across Pivotal's broad customer base, she develops core cloud platform strategies that drive significant change in enterprise organizations, and influence the Pivotal Cloud Foundry evolution. Currently she is working on ways to bring the various cloud-computing models of Infrastructure as a Service, Application as a Service, Container as a Service, and Function as a Service together into a comprehensive offering that allows IT organizations to function at the highest levels. She is the author of the book Cloud Native: Designing Change-tolerant Software by Manning Publications (https://www.manning.com/books/cloud-native). An industry veteran with almost three decades of experience in image processing, scientific visualization, distributed systems and web application architectures, and cloud-native platforms, she holds the BS and MS in computer science from California State University, Northridge and further studied theory of computing and programming languages at Indiana University. Contact her at cornelia@corneliadavis.com or cdavis@pivotal.io.*