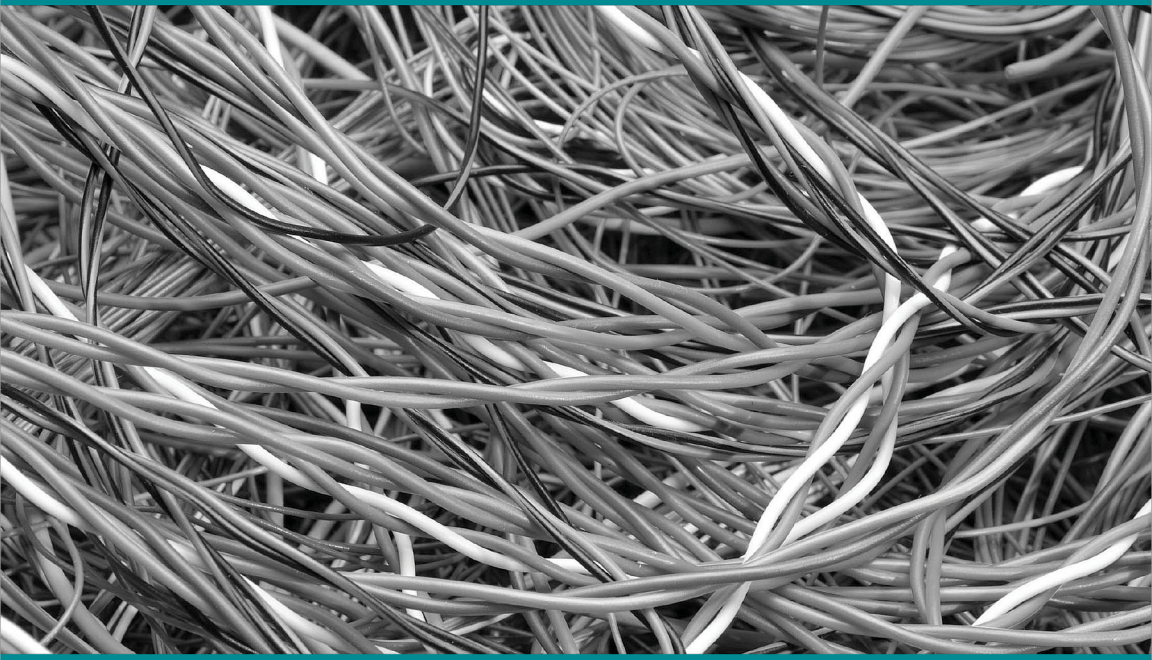
O'REILLY®

Everything Is Distributed

Essays On Performance and Operations



Courtney Nash
& Mike Loukides

Everything Is Distributed

Courtney Nash and Mike Loukides

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Everything Is Distributed

by Courtney Nash and Mike Loukides

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Anderson
Production Editor: Kara Ebrahim
Proofreader: Kara Ebrahim
Cover Designer: Ellie Volckhausen
Interior Designer: David Futato
Illustrator: Rebecca Demarest

September 2014: First Edition

Revision History for the First Edition:

2014-08-26: First release

2015-03-24: Second release

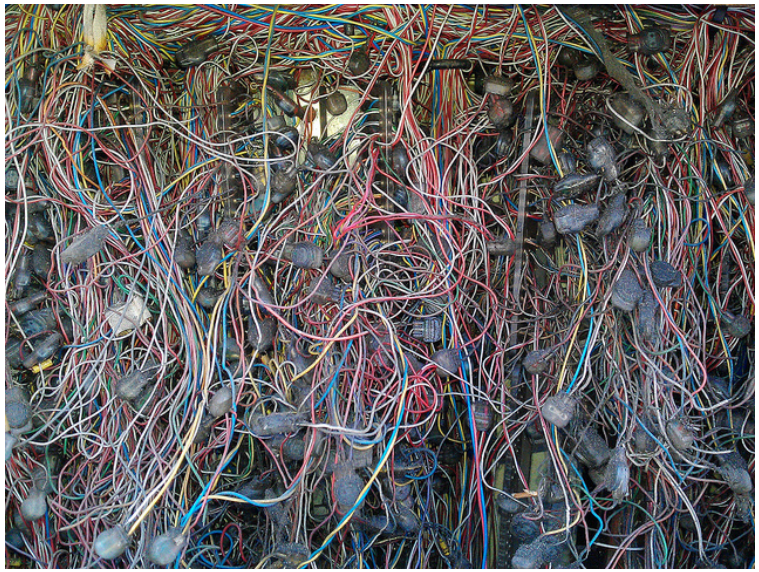
ISBN: 978-1-491-91247-8

[LSI]

Table of Contents

Everything Is Distributed

Courtney Nash



> What is surprising is not that there are so many accidents. It is that there are so few. The thing that amazes you is not that your system goes down sometimes, it's that it is up at all.
>
> — Richard Cook

In September 2007, Jean Bookout, 76, was driving her Toyota Camry down an unfamiliar road in Oklahoma, with her friend Barbara Schwarz seated next to her on the passenger side. Suddenly, the Camry began to accelerate on its own. Bookout tried hitting the brakes, applying the emergency brake, but the car continued to accelerate. The car eventually collided with an embankment, injuring Bookout and

killing Schwarz. In a subsequent legal case, lawyers for Toyota pointed to the most common of culprits in these types of accidents: human error. "Sometimes people make mistakes while driving their cars," one of the lawyers claimed. Bookout was older, the road was unfamiliar, these tragic things happen.

However, a recently concluded product liability case against Toyota has turned up a very different cause: a stack overflow error in Toyota's software for the Camry. This is noteworthy for two reasons: first, the oft-cited culprit in accidents—human error—proved not to be the cause (a problematic premise in its own right), and second, it demonstrates how we have definitively crossed a threshold from software failures causing minor annoyances or (potentially large) corporate revenue losses into the realm of human safety.

It might be easy to dismiss this case as something minor: a fairly vanilla software bug that (so far) appears to be contained to a specific car model. But the extrapolation is far more interesting. Consider the self-driving car, development for which is well underway already. We take out the purported culprit for so many accidents, human error, and the premise is that a self-driving car is, in many respects, safer than a traditional car. But what happens if a failure that's completely out of the car's control occurs? What if the data feed that's helping the car to recognize stop lights fails? What if Google Maps tells it to do something stupid that turns out to be dangerous?

We have reached a point in software development where we can no longer understand, see, or control all the component parts, both technical and social/organizational—they are increasingly complex and distributed. The business of software itself has become a distributed, complex system. How do we develop and manage systems that are too large to understand, too complex to control, and that fail in unpredictable ways?

Embracing Failure

Distributed systems once were the territory of computer science PhDs and software architects tucked off in a corner somewhere. That's no longer the case. Just because you write code on a laptop and don't have to care about message passing and lockouts doesn't mean you don't have to worry about distributed systems. How many API calls to external services are you making? Is your code going to end up on desktop sites and mobile devices—do you even know all the possible

devices? What do you know now about the network constraints that may be present when your app is actually run? Do you know what your bottlenecks will be at a certain level of scale?

One thing we know from classic distributed computing theory is that distributed systems fail more often, and the failures often tend to be partial in nature. Such failures are not just harder to diagnose and predict; they're likely to be not reproducible—a given third-party data feed goes down or you get screwed by a router in a town you've never even heard of before. You're always fighting the intermittent failure, so is this a losing battle?

The solution to grappling with complex distributed systems is not simply more testing, or Agile processes. It's not DevOps, or continuous delivery. No one single thing or approach could prevent something like the Toyota incident from happening again. In fact, it's almost a given that something like that *will* happen again. The answer is to embrace that failures of an unthinkable variety are possible—a vast sea of unknown unknowns—and to change how we think about the systems we are building, not to mention the systems within which we already operate.

Think Globally, Develop Locally

Okay, so anyone who writes or deploys software needs to think more like a distributed systems engineer. But what does that even mean? In reality, it boils down to moving past a single-computer mode of thinking. Until very recently, we've been able to rely on a computer being a relatively deterministic thing. You write code that runs on one machine, you can make assumptions about what, say, the memory lookup is. But nothing really runs on one computer any more—the cloud is the computer now. It's akin to a living system, something that is constantly changing, especially as companies move toward continuous delivery as the new normal.

So, you have to start by assuming the system in which your software runs will fail. Then you need hypotheses about why and how, and ways to collect data on those hypotheses. This isn't just saying "we need more testing," however. The traditional nature of testing presumes you can delineate all the cases that require testing, which is fundamentally impossible in distributed systems. (That's not to say that testing isn't important, but it isn't a panacea, either.) When you're in a distributed environment and most of the failure modes are things you can't predict

in advance and can't test for, monitoring is the only way to understand your application's behavior.

Data Are the Lingua Franca of Distributed Systems

If we take the living-organism-as-complex-system metaphor a bit further, it's one thing to diagnose what caused a stroke after the fact versus to catch it early in the process of happening. Sure, you can look at the data retrospectively and see the signs were there, but what you want is an early warning system, a way to see the failure as it's starting, and intervene as quickly as possible. Digging through averaged historical time series data only tells you what went wrong, that one time. And in dealing with distributed systems, you've got plenty more to worry about than just pinging a server to see if it's up. There's been an explosion in tools and technologies around measurement and monitoring, and I'll avoid getting into the weeds on that here, but what matters is that, along with becoming intimately familiar with how histograms are generally preferable to averages when it comes to looking at your application and system data, developers can no longer think of monitoring as purely the domain of the embattled system administrator.

Humans in the Machine

There are no complex software systems without people. Any discussion of distributed systems and managing complexity ultimately must acknowledge the roles people play in the systems we design and run. Humans are an integral part of the complex systems we create, and we are largely responsible for both their variability and their resilience (or lack thereof). As designers, builders, and operators of complex systems, we are influenced by a risk-averse culture, whether we know it or not. In trying to avoid failures (in processes, products, or large systems), we have primarily leaned toward exhaustive requirements and creating tight couplings in order to have "control," but this often leads to brittle systems that are in fact more prone to break or fail.

And when they do fail, we seek blame. We ruthlessly hunt down the so-called "cause" of the failure—a process that is often, in reality, more about assuaging psychological guilt and unease than uncovering why things really happened the way they did and avoiding the same

outcome in the future. Such activities typically result in more controls, engendering increased brittleness in the system. The reality is that most large failures are the result of a string of micro-failures leading up to the final event. There is no root cause. We'd do better to stop looking for one, but trying to do so is fighting a steep uphill battle against cultural expectations and strong, deeply ingrained psychological instincts.

The processes and methodologies that worked adequately in the '80s, but were already crumbling in the '90s, have completely collapsed. We're now exploring new territory, new models for building, deploying, and maintaining software—and, indeed, organizations themselves.

Photo by Mark Skipper, used under a Creative Commons license.

Learn More

Planning for failure

- Bloomberg on the Toyota acceleration case
- An analysis of the case from NHTSA
- The role of human error in the incident
- "Resilience In Complex Adaptive Systems: Operating At The Edge Of Failure", Velocity New York 2013 keynote by Richard Cook
- "What, Where and When is the Risk in System Design?", Velocity Santa Clara 2013 Keynote by Johan Bergström
- *Learning from First Responders: When Your Systems Have to Work*, free ebook by Dylan Richard

Managing complexity

- *In Search of Certainty*, by Mark Burgess
- "Beyond Automation with CFEngine 3", video by Mark Burgess
- *Continuous Quality* (O'Reilly), by Jeff Sussna
- *Building Anti-Fragile Systems and Teams* (O'Reilly), by Dave Zwieback

Beyond the Stack

Mike Loukides

The shape of software development has changed radically in the last two decades. We've seen many changes: the Internet, the Web, virtualization, and cloud computing. All of these changes point toward a fundamental new reality: all computing has become distributed computing. The age of standalone applications has disappeared, and applications that run on a single computer are almost inconceivable. Distributed is the default; and whether an application is running on Amazon Web Services (AWS), on a private cloud, or even on a desktop or a mobile phone, it depends on the behavior of other systems and services that aren't under the developer's control.

In the past few years, a new toolset has grown up to support the development of massively distributed applications. We call this new toolset the Distributed Developer's Stack (DDS). It is orthogonal to the more traditional world of servers, frameworks, and operating systems; it isn't a replacement for the aged LAMP stack, but a set of tools to make development manageable in a highly distributed environment.

The DDS is more of a meta-stack than a "stack" in the traditional sense. It's not prescriptive; we don't care whether you use AWS or OpenStack, whether you use Git or Mercurial. We do care that you develop for the cloud, and that you use a distributed version control system. The DDS is about the requirements for working effectively in the second decade of the 21st century. The specific tools have evolved, and will continue to evolve, and we expect you to evolve, too.

Cloud as Platform

AWS has revolutionized software development. It's simple for a startup to allocate as many servers as it needs, tailored to its requirements, at low cost. A developer at an established company can short-circuit traditional IT procurement channels, and assemble a server farm in minutes using nothing more than a credit card.

Even applications that don't use AWS or some other cloud implementation are distributed. The simplest web page requires a server, a web browser to view it, DNS servers for hostname resolution, and any number of switches and routers to move bits from one place to another. A web application that's only slightly more complex relies on authentication servers, databases, and other web services for real-time data. All these are externalities that make even the simplest application into a distributed system. A power outage, router failure, or even a bad cable in a city you've never heard of can take your application down.

I'm not arguing that the sky is falling because … cloud. But it is critically important to understand what the cloud means for the systems we deploy and operate. As the number of systems involved in an application grows, the number of failure modes grows combinatorially. An application running over 10 servers isn't 10 times as complex as an application running on a single server; it's thousands of times more complex.

The cloud is with us to stay. Whether it's public or private, AWS, OpenStack, Microsoft Azure, or Google Compute Engine, applications will run in the cloud for the foreseeable future. We have to deal with it.

Development as a Distributed Process

We've made many advances in source control over the years, but until recently we've never dealt with the fact that software development itself is distributed. Our models have been based on the idea of lone "programmers" writing monolithic "programs" that run on isolated "machines." We have had build tools, source control archives, and other tools to make the process easier, but none of these tools really recognize that projects require teams. Developers would work on their part of the project, then try to resolve the mess in a massive "integration" stage in which all the separate pieces are assembled.

The version control system Git recognizes that a team of developers is fundamentally a distributed system, and that the natural process of software development is to create branches, or forks, then merge those branches back into a master repository. All developers have their own local codebase, branching from master. When they're ready, they merge their their changes and push them back to master; at this point, other members of the team can pull the changes to update their own code bases. Each developer's work is decoupled from others; team members can work asynchronously, distributed in time as well as in space.

Continuous integration tools like Jenkins and its predecessor, Hudson, were among the first tools to recognize the paradigm shift. Continuous integration reflects the reality that, when development is distributed, integrating the work of all the developers has to be a constant process. It can't be postponed until a major release is finished. It's important to move forward in small, incremental steps, making sure that the project always builds and works.

Facilitating collaboration on a team of distributed developers will never be a simple problem. But it's a problem that becomes much more tractable with tools that recognize the nature of distributed development, rather than trying to maintain the myth of the solitary programmer.

Infrastructure as Code

Infrastructure as code has been a slogan at the Velocity Conference for some years now. But what does that mean?

Cloud computing lets developers allocate servers as easily as they allocate memory. But as any '90s sysadmin knows, the tough part isn't taking the server out of the box, it's getting it set up and configured correctly. And that's a pain whether you're sitting at a console terminal with a green screen or ssh'ed into a virtual box a thousand miles away. It's an even bigger pain when you've grown from a single server or a small cluster to hundreds of AWS nodes distributed around the world.

In the last decade, we've seen a proliferation of tools to solve this problem. Chef, Puppet, CFEngine, Ansible, SaltStack, and other tools capture system configurations in scripts, automating the configuration of computer systems, whether physical or virtual. The ability to allocate machines dynamically and configure them automatically

changes our relationship to computing resources. In the old days, when something went wrong, a sysadmin had to nurse the system back to health, whether by rebooting, reinstalling software, replacing a disk drive, or something else. When something was broken, you had to fix it. That still may be true of our laptops or phones, but it's no longer true of our production infrastructure. If something goes wrong with a server on AWS, you delete it, and start another one. It's easier, simpler, quicker, cheaper. A small operations staff can manage thousands, or tens of thousands, of servers. With the appropriate monitoring tools, it's even possible to automate the process of identifying a malfunctioning server, stopping it, deleting it, and allocating a new one.

If configuration is code, then configuration must be considered part of the software development process. It's not enough to develop software on your laptop, and expect operations staff to build systems on which to deploy. Development and deployment aren't separate processes; they're two sides of the same thing.

Containerization as Deployment

Containers are the most recent addition to the stack. Containers go a step beyond virtualization: a system like Docker lets you build a package that is exactly what you need to deploy your software: no more, and no less. This package is analogous to the standard shipping container that revolutionized transportation several decades ago. Rather than carefully loading a transport ship with pianos, nuts, barrels of oil, and what have you, these things are stacked into standard containers that are guaranteed to fit together, that can be loaded and unloaded easily, placed not only onto the ship but also onto trucks and trains, and never opened until they reach their destination.

Containers are special because they always run the same way. You can package your application in a Docker container and run it on your laptop; you can ship it to Amazon and run it on an AWS instance; you can ship it to a private OpenStack cloud and run it there; you can even run it in on a server in your machine room, if you still have one. The container has everything needed to run the code correctly. You don't have to worry about someone upgrading the operating system, installing a new version of Apache or nginx, replacing a library with a "better" version, or any number of things that can result in unpleasant surprises. Of course, you're now responsible for keeping your containers patched with the latest operating systems and libraries; you

can't rely on the sysadmins. But you're in control of the process: your software will always run in exactly the environment you specify. And given the many ways software can fail in a distributed environment, eliminating one source of failure is a good thing.

Monitoring as Testing

In a massively distributed system, software can fail in many ways that you can't test for. Test-driven development won't tell you how your applications will respond when a router fails. No acceptance test will tell you how your application will perform under a load that's 1,000 times the maximum you expected. Testing may occasionally flush out a race condition that you hadn't noticed, but that's the exception rather than the rule.

Netflix's Chaos Monkey shows how radical the problem is. Because systematic testing can never find all the problems in a distributed system, Netflix resorts to random vandalism. Chaos Monkey (along with other members of Netflix's Simian Army) periodically terminates random services in Netflix's AWS cloud, potentially causing failures in their production systems. These failures mostly go unnoticed, because Netflix developers have learned to build systems that are robust and resilient in the face of failure. But on occasion, Chaos Monkey reveals a problem that probably couldn't have been discovered through any other means.

Monitoring is the next step beyond testing; it's really continuous runtime testing for distributed systems where testing is impossible. Monitoring tools such as Riemann, statsd, and Graphite tell you how your systems are handling real-world conditions. They're the tools that let you know if a router has failed, if your servers have died, or if they're not holding up under an unexpected load. Back in the '60s and '70s, computers periodically "crashed," and system administrators would scurry around figuring out what happened and getting them rebooted. We no longer have the luxury of waiting for failures to happen, then guessing about what went wrong. Monitoring tools enable us to see problems coming, and when necessary, to analyze what happened after the fact.

Monitoring also lets the developer understand what features are being used, and which are not, and applications that are deployed as cloud services lend themselves easily to A/B testing. Rather than designing a monolithic piece of software, you start with what Eric Ries calls a

minimum viable product—the smallest possible product that will give you validated learning about what the customer really wants and responds to—and then build out from there. You start with a hypothesis about user needs, and constantly measure and learn how better to meet those needs. Software design itself becomes iterative.

Is This DevOps?

No. The DDS stack is about the tools for working in a highly distributed environment. These tools are frequently used by people in the DevOps movement, but it's important not to mistake the tools for the substance. DevOps is about the culture of software development, starting with developers and operations staff, but in a larger sense, across companies as a whole. Perhaps the best statement of that is Velocity speaker Jeff Sussna's (*@jeffsussna*) post "Empathy: The Essence of DevOps".

Most globally, DevOps is about the realization that software development is a business process, all businesses are software businesses, and all businesses are ultimately human enterprises. To mistake the tools for the cultural change is the essence of cargo culting.

The CIO of Fidelity Investments once remarked to Tim O'Reilly: "We know about all the latest software development tools. What we don't know is how to organize ourselves to use them." DevOps is part of the answer to that business question: how should the modern enterprise be organized to take advantage of the way software systems work now? But it's not just integration of development and IT operations. It's also integration of development and marketing, business modeling and measurement, and, in a public sector context, policy making and implementation.

Why Now?

All software is "web software," even the software that doesn't look like web software. We've become used to gigantic web applications running across millions of servers; Google and Facebook are in the forefront of our consciousness. But the Web has penetrated to surprising places. You might not think of enterprise applications as "web software," but it's increasingly common for internal enterprise applications to have a web interface. The fact that it's all behind a firewall is irrelevant.

Likewise, we've heard many times that mobile is the future, and the Web is dead. Maybe, if "the Web" means Firefox and Chrome. But the first time the Web died, Nat Torkington (*@gnat*) said: "I've heard that the Web is dead. But all the applications that have killed it are accessing services using HTTP over port 80." A small number of relatively uninteresting mobile applications are truly standalone, but most of them are accessing data services. And those services are web services; they're using HTTP, running on Apache, and pushing JSON documents around. Dead or not, the Web has won.

The Web has done more than win, though. The Web has forced all applications to become distributed. Our model is no longer Microsoft Word, Adobe InDesign, or even the original VMWare. We're no longer talking products in shrink-wrapped boxes, or even enterprise software delivered in massive deployments, we're talking products like Gmail and Netflix that are updated and delivered in real time from thousands of servers. These products rely on services that aren't under the developer's control, they run on servers that are spread across many data centers on all continents, and they run on a dizzying variety of platforms.

The future of software development is bound up with distributed systems, and all the complexity and indeterminacy that entails. We've started to develop the tools necessary to make distributed systems tractable. If you're part of a software development or operations team, you need to know about them.

Learn More

Cloud computing

- *The Enterprise Cloud: Lessons Learned* (O'Reilly), by James Bond
- *AWS System Administration* (O'Reilly), by Mike Ryan
- *OpenStack Operations Guide* (O'Reilly), by Tom Fifield, Diane Fleming, Anne Gentle, Lorin Hochstein, Jonathan Proulx, Everett Toews, and Joe Topjian
- *eCommerce in the Cloud* (O'Reilly), by Kelly Goetsch
- *Resilience and Reliability on AWS* (O'Reilly), by Jurg van Vliet, Flavia Paganelli, and Jasper Geurtsen

- *60 Recipes for Apache CloudStack* (O'Reilly), by Sebastien Goasguen

Distributed development

- *Jenkins: The Definitive Guide* (O'Reilly), by John Ferguson Smart
- *Version Control with Git* (O'Reilly), by Jon Loeliger and Matthew McCullough
- *Git Pocket Guide* (O'Reilly), by Richard E. Silverman
- *Building Microservices* (O'Reilly), by Sam Newman

Infrastructure as code

- Adam Jacob's chapter on "Infrastructure as Code" from *Web Operations* (O'Reilly), by John Allspaw and Jesse Robbins
- *Learning Chef* (O'Reilly), by Mischa Taylor and Seth Vargo
- *Customizing Chef* (O'Reilly), by Jon Cowie
- *Test-Driven Infrastructure with Chef* (O'Reilly), by Stephen Nelson-Smith
- *Vagrant: Up and Running* (O'Reilly, by Mitchell Hashimoto
- "Beyond Automation with CFEngine 3", video by Mark Burgess
- *Salt Essentials* (O'Reilly), by Craig Sebenik
- *Learning MCollective* (O'Reilly), by Jo Rhett
- "Ansible - Python-Powered Radically Simple IT Automation", presentation by Michael DeHaan, PyCon 2014

Containerization

- Getting Started with Docker, a set of Docker tutorials from OSCON 2014
- Interview with James Turnbull (Docker) at Velocity Santa Clara 2014

Monitoring and testing

- *Using WebPagetest* (O'Reilly), by Rick Viscomi, Andy Davies, and Marcel Duran
- *Monitoring with Ganglia* (O'Reilly), by Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, and Daniel Pocock
- *Feedback Control for Computer Systems* (O'Reilly), by Philipp K. Janert
- *Complete Web Monitoring* (O'Reilly), by Alistair Croll and Sean Power
- "Beyond Averages", Velocity New York 2013 presentation by Dan Kuebrich
- *Lightweight Systems for Realtime Monitoring*, free ebook by Sam Newman

Revisiting DevOps

Mike Loukides

It's always easy to think of DevOps (or of any software industry paradigm) in terms of the tools you use; in particular, it's very easy to think that if you use Chef or Puppet for automated configuration, Jenkins for continuous integration, and some cloud provider for on-demand server power, that you're doing DevOps. But DevOps isn't about tools; it's about culture, and it extends far beyond the cubicles of developers and operators.

Empathy

As Jeff Sussna says in "Empathy: The Essence of DevOps":

> …it's not about making developers and sysadmins report to the same VP. It's not about automating all your configuration procedures. It's not about tipping up a Jenkins server, or running your applications in the cloud, or releasing your code on GitHub. It's not even about letting your developers deploy their code to a PaaS. The true essence of DevOps is empathy.

By "empathy," Jeff means an intimate understanding between the development and operations teams. Tricks like co-locating dev and ops desks, placing them under the same management, and so on, are just a means to an end: the real goal is communications between organizations that can easily become antagonistic. Indeed, the origin of our Velocity conference was the realization that, although the development and operations teams were frequently antagonists, they spoke the same language and had the same goals.

Promise Theory

In his blog post "The Promises of DevOps", Mark Burgess discusses the connection between DevOps and promise theory. Promise theory is a radically different take on management: rather than basing management on a top-down, command-and-control network of requirements, promise theory builds services from networks of local promises. Components of a system (which may be a machine or a human) aren't presented with a list of "requirements" that they must deliver; they are asked to make "promises" about what they are able to deliver. Promises are local commitments: a developer commits to writing a specific piece of code by a specific date, operations staff commits to keeping servers running within certain parameters.

Promise theory doesn't naively assume that all promises will be kept. Humans break their promises all the time; machines (which can also be agents in a network of promises) just break. But with promise theory, agents are aware of the commitments they're making, and their promises are more likely to reflect what they're capable of performing. As Burgess explains:

> Dev promises things that Ops like; Ops promises things that Dev likes. Both of them promise to keep the supply chain working at a certain rate, i.e., Dev supplies at a rate that Ops can promise to deploy. By choosing to express this as promises, we know the estimates were made with accurate information by the agent responsible, not by external wishful thinkers without a clue.

And a well-formed network of promises includes contingencies and backups. What happens if Actor A doesn't deliver on Promise X? It may be counterintuitive, but a web of promises exposes its weak links much more readily than a top-down chain of command. Networks of promises provide services that are more robust and reliable than command and control management pushed down from above. As Tim Ottinger puts it in a pair of Tweets:

> Some people waste their time trying to make a perfect, efficient machine with human cogs.
>
> — Tim Ottinger (*@tottinge*) June 12, 2014
>
> Generally people would be better off with a productive, dynamic community of talented human beings.
>
> — Tim Ottinger (*@tottinge*) June 12, 2014

That's the difference between top-down management and promise theory in a nutshell: are you building a machine made of human cogs, or a community of talent?

Burgess is completely clear that DevOps isn't about tools and technologies. "Cooperation has nothing to do with computers or programming. The principles of cooperation are universal matters of information exchange, and intent." Cooperation, information exchange, and networks of intent are first and foremost cultural issues. Likewise, Sussna's concept of "empathy" is about understanding (again, information exchange), and understanding is a cultural issue.

Blameless Postmortems

It's one thing to talk about cultural change and understanding; it's something different to put it into practice. To make this concrete, let's talk about one particular practice: blameless postmortems at Etsy. As John Allspaw writes, if a postmortem analysis is about understanding what actually happened, it's essential to do so in an atmosphere where employees can give an account of events "without fear of punishment or retribution." A postmortem is about information exchange and empathy (to use Sussna's word). If we can't find out what happened, we have no hope of building systems that are more resilient.

Blameless postmortems are all the more important because of another aspect of modern computing. Top-down management has long insisted that, when there's a failure, it must be traced to a single root cause, which usually ends up being "human error." But for complex systems, there is no root cause. This is an extremely important point: as we've pointed out, all systems are distributed, and all systems are complex systems. And almost all failures are the result of "perfect storms" of unrelated events, not single failures or errors that could or should have been anticipated. As Allspaw puts it, paraphrasing Richard Cook, "failures in complex systems require multiple contributing causes, each necessary but only jointly sufficient."

Beyond DevOps

DevOps isn't just about Dev and Ops. It's about corporate management as a whole.

If you've ever worked in a company where the project wasn't over until the blame was assigned (as I have), you know that the short-term result

of "single root cause" thinking is blame and shame for the individual "responsible." The long-term result is a solution that inevitably makes the organization more brittle and failure-prone, and less agile, less able to adapt to changing circumstances. Without a culture of understanding and empathy, it is impossible to get to real causes and to build systems that are more resilient.

The conclusions we're coming to are far-reaching. We've been discussing cultural change and DevOps, but we have hardly mentioned computing systems, software developers, or infrastructure engineers. It doesn't matter a bit whether the postmortem is about a server outage or bad lending practices; the same principles apply.

If all companies are software companies, then all companies have to learn how to manage their online operations. But beyond that: on the Web, we've seen dramatic decreases in product development time and dramatic increases in reliability and performance. Can those increases in productivity be extended through the whole enterprise, not just the online group? We believe so. Can practices like blameless postmortems make corporations more resilient in the face of failure, in addition to improving the lives of employees at every level? We believe so. Adoption of DevOps principles across the enterprise, and not just in the "online group," will be a slow process, but it's a necessary process. In five or ten years, we'll look back at who survived and who thrived, and we'll see that the enterprises that have built communities of collaboration, mutual respect, and understanding have outperformed their competition.

DevOps isn't just about Dev and Ops. It's about corporate management as a whole; it's about the entire corporate culture, from the janitors (who promise to keep the building clean) to the CEO (who promises to keep the company funded and the paychecks coming). Promise theory has emerged as the intellectual framework underpinning that change in culture. And Velocity is where we are discussing those changes.

See you in Beijing or Barcelona!

Learn More

DevOps

- "10+ Deploys Per Day: Dev and Ops Cooperation at Flickr", Velocity 2009 presentation by John Allspaw and Paul Hammond
- *DevOps in Practice*, free ebook from J. Paul Reed
- *The Phoenix Project*, by Gene Kim, Kevin Behr, and George Spafford
- *Lean Enterprise: Adopting Continuous Delivery, DevOps, and Lean Startup at Scale* (O'Reilly), by Jez Humble, Barry O'Reilly, and Joanne Molesky
- *Continuous Delivery*, by Jez Humble
- "What is DevOps?", blog post by Mike Loukides
- *Building a DevOps Culture*, free ebook by Mandi Walls
- *Training DevOps Staff*, free ebook by Mandi Walls
- *5 Unsung Tools of DevOps*, free ebook by Jonathan Thurman
- Interview with Gene Kim, Velocity Santa Clara 2014

Empathy and promise theory

- *Continuous Quality* (O'Reilly), by Jeff Sussna
- "Empathy: The Essence of DevOps", blog post by Jeff Sussna
- "The Promises of DevOps", blog post by Mark Burgess
- Interview with Mark Burgess, Velocity Santa Clara 2014
- *Promise Theory: Principles and Applications*, by Jan A. Bergstra and Mark Burgess

Blameless postmortems

- *Being Blameless* (O'Reilly), by Dave Zwieback
- *The Human Side of Postmortems*, free ebook by Dave Zwieback
- John Allspaw's post on blameless postmortems
- Interview with John Allspaw on blameless postmortems, Velocity SC 2014

Performance Is User Experience

Lara Swanson and Courtney Nash

Despite a wealth of research, writing, and even media coverage of the pain/cost of slow websites and apps, the Web is barely getting faster—depending on who you ask, it may even be getting slower. Back at Velocity 2009, a groundbreaking presentation by Google and Microsoft engineers showed how serious performance is: imperceptibly small increases in response time cause users to move to another site. If response time is over a second, a measurable percentage of users just click away. If your site takes four seconds to load, forget it: you don't exist. A fast website is not just a technology challenge, it is a user experience imperative.

The Slow Web

Three primary factors contribute to the continuing problem of the "slow web":

- Lack of general awareness of importance of performance among web developers, especially in more beginner/intermediate roles
- Design-heavy requirements (images and video) that increase page size. New techniques like parallax design, responsive web design, etc. can be significant performance hogs
- Third-party elements (scripts, APIs, social sharing features) aren't under your control, and can wreak havoc on performance

Web developers, designers, and frontend engineers all need to think about performance from a more holistic perspective. They have to master the basics (e.g., JavaScript minimization, network round-trip

reduction, image compression, etc.) and devise strategies to make the end-user experience seem as fast and seamless as possible. They need to exercise discipline around what to add to their pages: the latest video or special JavaScript effect is likely to be counterproductive if it makes the page slower and drives users away.

The Human Impact

Web performance *is* user experience. Fast page load time builds trust in your site; it yields more returning visitors, more users choosing your site over a competitor's site, and more people trusting your brand. Users expect pages to load in two seconds, and after three seconds, up to 40% of users will abandon your site. Similar results have been noted by major sites like Amazon, who found that 100 milliseconds of additional page load time decreased sales by one percent, and Google, who lost 20% of revenue and traffic due to half a second increase in page load time. Akamai has also reported that 75% of online shoppers who experience an issue such as freezing, crashing, taking too long to load, or having a convoluted checkout process will not buy from that site.

Web performance impacts more than just ecommerce sites; improvements from page speed optimization apply to any kind of site. Users will return to faster sites, evidenced in a study by Google Maps that noted an increase in returning traffic when the Google Maps homepage weight was reduced from 100 KB to 80 KB. Additionally, page load time is factored into search engine results, bumping faster sites higher in the results list than slower sites.

Page load time also has a significant impact on mobile users' experience. Lara Swanson's team at Etsy found an increased bounce rate of 12% on mobile devices when they added 160 KB of images to a page. DoubleClick removed one client-side redirect and saw a 12% increase in clickthrough rate on mobile devices. In another study, researchers found that if Amazon changed all of their images to compressed JPEG files, it would save 20% of the energy needed to load the page on a phone, and on Facebook it would save 30%.

The bottom line is that your efforts to optimize your site have an effect on the entire experience for your users, including battery life.

These numbers matter because collectively we are designing sites with increasingly rich content—lots of dynamic elements, larger JavaScript

files, beautiful animations, complex graphics, etc. You may focus on optimizing design and layout, but those can come at a tradeoff with page speed. Some responsively designed sites are irresponsible with the amount of JavaScript and images they use to reformat a site for smaller screen sizes.

Think about your most recent design. How many different font weights were used? How many images did you use? How large were the image files, and what file formats did you use? How did your design affect the plan for markup and CSS structure?

It's Not Just the Desktop: It's Mobile, Too

In the past, developers relied on the assumption that people didn't expect mobile sites or apps to be as fast as the desktop. For a brief romantic period (probably before the iPhone took off), this may have been true. But the opposite holds, and strongly, now. If anything, users expect their mobile devices to be faster than their desktops.

Expectations for website and app performance on mobile devices is even more stringent than for desktops. They don't care about network constraints or how many server-client roundtrips you have to make —they want things to load in under 4 seconds. Initial irritation starts to set in at 1 second. As such, this is an area where the focus on user experience must be even stronger, notably finding ways to make mobile experiences feel faster, even if they really aren't.

The problems of the modern development or operations team are difficult enough without dealing with the performance of devices you don't control, don't even know about, and possibly can't even test, communicating over networks that may be slow or unreliable. But that's the world we live in, and those are the challenges we face.

Selling It to Your Organization

It's one thing to know performance is important, and to understand how to address performance problems technically. It's a whole other challenge to convince management to invest time in improving it. Culture change may be the single most challenging aspect of implementing performance improvements, and it involves helping upper management as well as your peers understand the importance of performance's impact on your site's user experience.

Start by educating those around you. Teach them not only how to positively affect page load time and perceived performance, but also why performance is an important focus for your organization. Share studies that detail the impact that performance has on business metrics, or build your own experiments that show how a site speedup can positively affect bounce rates, returning visitors, and other metrics that your coworkers and upper management at your company care about.

Incentivize upper management to give you and others an opportunity to work on improving the performance of your site. Run multiple page speed tests using different locations and devices and share the filmstrip or video versions with them. How does your site perform on a mobile network, or on another continent? Comparing the videos of your desktop page speed to what a mobile or global user may see will help those around you *feel* what your users are likely experiencing. Another tactic to incentivize upper management is comparing the video of your site's page load time to that of a competitor's. How does your site stack up? Could you be losing visitors to another site because it outperforms yours?

Develop performance budgets for new projects and publicize your site's speed internally. Teach lunch and learns. Incorporate performance into designers' and developers' daily workflows using automated testing and dashboards. Empower people to understand how their work directly impacts your site's end user experience, especially the effect that they have on performance.

Learn More

The slow Web

- *Web Page Size, Speed, and Performance*, free ebook by Terrence Dorsey
- "Mobile Web Stress: Understanding the Neurological Impact of Poor Performance", webcast by Tammy Everts

Frontend performance

- *Getting Started with Web Performance*(O'Reilly), by Daniel Austin
- *Designing for Performance* (O'Reilly), by Lara Swanson

- "Web performance is user experience", blog post by Lara Swanson
- *High Performance Websites* (O'Reilly), by Steve Souders
- *Even Faster Websites* (O'Reilly), by Steve Souders
- *High Performance Browser Networking* (O'Reilly), by Ilya Grigorik
- *Web Performance Daybook Volume 2* (O'Reilly), by Stoyan Stefanov
- "Achieving Rapid Response Times in Large Online Services", presentation by Jeff Dean, Velocity 2014]

Mobile performance

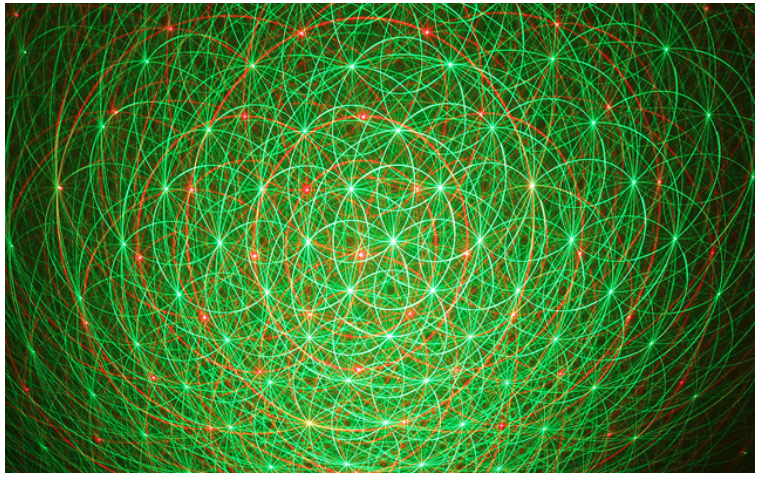
- *Programming the Mobile Web, 2nd Edition* (O'Reilly), by Maximiliano Firtman
- *High Performance iOS Apps* (O'Reilly), by Gaurav Vaish
- *Responsive & Fast* (O'Reilly), by Guy Podjarny
- *High Performance Responsive Design* (O'Reilly), by Tom Barker
- "Speed Up Mobile Delivery by Squeezing Out Network Latency", webcast by Steve Miller-Jones

Selling performance in your organization

- *Art of Application Performance Testing, 2nd Edition* (O'Reilly), by Ian Molyneux
- "4 Steps to a culture of performance", blog post by Mehdi Daoudi
- *Designing for Performance* (O'Reilly), by Lara Swanson

From the Network Interface to the Database

Mike Loukides



From the beginning, the Velocity Conference has focused on web performance and operations—specifically, web operations. This focus has been fairly narrow: browser performance dominated the discussion of "web performance," and interactions between developers and IT staff dominated operations.

Web Ops and Performance

These limits weren't bad. Perceived performance really is dominated by the browser—how fast you can get resources (HTML, images, CSS files, JavaScript libraries) over the network to the browser, and how

fast the browser can execute those resources. How long before a user stops waiting for your page to load and clicks away? How do you make a page useable as quickly as possible, even before all the resources have loaded? Those discussions were groundbreaking and surprising: users are incredibly sensitive to page speed.

That's not to say that Velocity hasn't looked at the rest of the application stack; there's been an occasional glance in the direction of the database and an even more occasional glance at the middleware. But the database and middleware have, at least historically, played a bit part. And while the focus of Velocity has been frontend tuning, speakers like Baron Schwartz haven't let us ignore the database entirely.

The web operations side of Velocity has been more diverse: integrating the work of developers and ops staff, moving from waterfall practices to "agile" development, developing a culture of continuous deployment—these have been major milestones in the Velocity story. We're proud that a healthy "devops" movement grew out of Velocity. But here, too, there's certainly a bigger story to tell.

In the operations world, it's never been possible to abstract a system from what's happening at the lowest levels. In the past few years, we've learned that all applications are distributed. So, there have been sessions on resilient systems, accepting failure, and blameless postmortems: that's a start. There's a lot of system between the web server and the browser. For that matter, there's a lot of system between the web server and the point where the bits leave the building. For that matter squared, what building? A building you own, or a building Amazon owns in a city you can't name?

While Velocity has been a pioneer in recognizing the distributed nature of modern computing as well as the tools and the cultural changes needed to deal effectively with a distributed world, we've only taken occasional glances at the infrastructure that lies behind the server. We've only taken occasional looks at data centers themselves. And I don't think we've ever discussed routing issues or routers, though a router failure can sink your application just as badly as a dead server.

Broadening the Scope

So, in our ongoing exploration of web performance and operations, we're going to broaden the scope. I don't think we'll be doing less of anything than we are today, but we do have to take a step back and

look at the bigger picture. If everything is distributed, it makes no sense to look at part of a distributed system and skip the rest. In particular:

- What are the performance and operational implications of our low-level network infrastructure? What happens when you hand off your packets to "the network"? There are many important questions here. How can you use new technologies like software-defined networks and network function virtualization to make your infrastructure more reliable? What roles do CDNs and other intermediaries play in delivering data to our users? How does the advent of a "slow lane" for those of us who aren't rich enough to negotiate with Comcast, Verizon, and AT&T affect our applications? We don't have to like it, but we're naive if we don't think those issues will affect us.
- What are the performance and operational implications of middleware and databases? While the backend of the application doesn't have the same millisecond-by-millisecond effect on performance, it has a huge effect on scalability. And an application that hasn't scaled well is very slow. We're not talking about the extra second of latency that makes some users frustrated and drives them away: we're talking about being dead in the water during the Christmas rush. An application that's slow because it can't handle peak loads is slow in an entirely different way from an application that downloads 5 MB of JavaScript libraries and images before it's useable, but the users don't care either way. It's slow, and they're going elsewhere.

All systems are distributed systems. And in that sense, there's nothing really new here. Rather than focusing narrowly on a few key components of our distributed systems, we're extending the scope to include the whole thing, even the parts we don't know about or see. Furthermore, as our distributed systems evolve, we see how they fit into Velocity's historical themes. Configuration is code, but that's a nasty argument to make when the "code" you're talking about is a mess of Cisco IOS configuration files. Software-defined networks (and their descendants) turn network configuration into software. And in a virtualized, cloud-oriented world, automated database scaling is also a matter of software.

The historical mission of Velocity has been to unite web developers and operations, to get both sides to realize they're on the same team and talking the same language. Over the years, the number of people

we've invited to the conversation has grown: a few years ago, we started to address mobile developers. Now, the conversation is becoming even wider. We trust we won't lose focus. But if there's one thing we've learned over the years, silos are nobody's friend, and it doesn't matter who's living in the silo, whether it's a DBA or a router administrator.

Everything is distributed. And when everything is distributed, everyone has a stake in the conversation. We're looking forward to burning down a few more silos, and inviting even more people into the Velocity tent. It's a big tent indeed.

Photo by Ian Barbour, used under a Creative Commons license.