DevOps: The Ops Perspective

by Don Jones

Table of Contents

ReadMe	1.1
About this Book	1.2
What is DevOps?	1.3
What Does DevOps Look Like?	1.4
Operational Capabilities of a DevOps Environment	1.5
IT Ops Skills in a DevOps Environment	1.6
Operations as Development	1.7
DevOps Doesn't Exclude Anyone	1.8
A DevOps Reading List	1.9

"DevOps" is such a popular term these days - but what's it actually mean to an Ops person? This high-level book attempts to put DevOps into perspective with real-world examples and descriptions.

DevOps: The Ops Perspective

By Don Jones

"DevOps" is such a popular term these days - but what's it actually mean to an Ops person? This high-level book attempts to put DevOps into perspective with real-world examples and descriptions.

Was this book helpful? The author(s) kindly ask(s) that you make a tax-deductible (in the US; check your laws if you live elsewhere) donation of any amount to The DevOps Collective to support their ongoing work.

Check for Updates! Our ebooks are often updated with new and corrected content. We make them available in three ways:

- Our main, authoritative GitHub organization, with a repo for each book. Visit https://github.com/devops-collective-inc/
- Our GitBook page, where you can browse books online, or download as PDF, EPUB, or MOBI. Using the online reader, you can link to specific chapters. Visit https://www.gitbook.com/@devopscollective
- On LeanPub, where you can download as PDF, EPUB, or MOBI (login required), and "purchase" the books to make a donation to DevOps Collective. You can also choose to be notified of updates. Visit https://leanpub.com/u/devopscollective

GitBook and LeanPub have slightly different PDF formatting output, so you can choose the one you prefer. LeanPub can also notify you when we push updates. Our main GitHub repo is authoritative; repositories on other sites are usually just mirrors used for the publishing process. GitBook will usually contain our latest version, including not-yet-finished bits; LeanPub always contains the most recent "public release" of any book.

What is DevOps?

"DevOps," as a term, doesn't have a really concrete definition. It's a philosophy, a way of working, and it means different things to different people. For the most part, the DevOps community generally (if reluctantly) accepts the definition from the DevOps article on Wikipedia, which states in part:

> ...a software development method that stresses communication, collaboration, integration, automation, and measurement of cooperation between software developers and other information-technology (IT) professionals.

I don't personally feel that the definition goes far enough; DevOps is far more that a "software development method." However, let's play along for a moment, and recognize that *software rules the world*. The President of the United States didn't stand up and say, "everyone should learn to subnet," he said, "everyone should learn to code." The massive global Internet, an impressive set of network infrastructure engineering as has ever been seen, is for the most part a dumb pipe used to deliver software. Software's what it's all about. But software doesn't get anywhere, or do anything, without infrastructure. It's the two working together that make technology useful, and that's why DevOps strings together both "Development" *and* "Operations." So for this book, I'd like to take the liberty of slightly re-defining DevOps as:

> ...an approach to technology management that stresses communication, collaboration, integration, automation, and measurement of cooperation between software developers and IT operational ("ops") personnel for the purpose of creating and delivering software applications to their users.

Understanding that DevOps is *a big thing* is important, because it's actually so big that it's almost impossible to look at all at once. It involves numerous techniques, multiple roles within an organization (that's why "cooperation" is in the description, along with "collaboration"), and lots of intersecting technologies. *This* book isn't going to try and look at all that.

Some Background

The term "DevOps" was likely coined by Patrick Dubois , inspired by a Velocity 2009 presentation by John Allspaw. Philosophically, it's inspired in large part by the "Lean Manufacturing" teachings of luminaries like W. E. Deming, Taiichi Ono, Eli Goldratt, and others. That's important, because those gents based their thinkings on the premise that

most workers want to do a good job. A common thread throughout Lean Manufacturing - and in fact a specific point of Deming's approach - was to end reliance on "QA" as a means of achieving quality. Yes, you put measures in place to help people prevent their own silly mistakes, but you don't put "gates" in place that assume your workers are malicious or incompetent. That one principle often becomes the biggest hurdle in adopting Lean Manufacturing, DevOps, or anything else that derives from that principle.

DevOps, for Ops

Instead, this book will look at the microcosm of DevOps related more specifically to *Ops*. In any organization attempting to implement a DevOps approach to life, the operational side of the house needs to deliver certain capabilities. In many cases, the operational side of the organization needs to deliver a level of automation and a kind of self-service that allows the *Dev* side to conduct there business without as much operational intervention. Operations, in that regard, is all about providing Development with safe, manageable, monitor-able ways of getting software to the end user, without it always being a major Operational project. *Exactly* how you proceed, and what capabilities you provide, will vary greatly depending on your organization.

In providing those capabilities, Operations will itself have to embark on a certain amount of software development, to create units of automation that make the operations side of the organization run more independently. Those software development efforts can themselves be conducted in a very DevOps-centric fashion, and this book will also focus heavily on that activity.

It's a Philosophy

DevOps is a lot like accounting, in that it's a set of abstract principles, approaches, and patterns. In accounting, you've got Generally Accepted Accounting Practices, or GAAP. They're not rules, per se, but they're *so* generally accepted that they do carry the weight of law in a lot of ways. DevOps is, or may become, like that, in that it can embody a set of practices and approaches that are generally recognized as the best way to go. Accounting also has tools that help you implement its practices. QuickBooks, for example, is a software package that embodies and enforces a lot of accounting practices, making it easier to put those into effect in your organization. Similarly, the DevOps world has a number of tools - many still nascent, but DevOps itself is pretty new - that help you implement DevOps practices and approaches. As the DevOps world tries things, learns from them, and fine-tunes its approaches, you'll find more and more tools being created to help make those

approaches easier and more consistent to apply in the real world. In this book, we'll focus far more on practices and patterns than on tools, so that we can stay higher-level and not force you to commit to a particular technology stack.

Unlike accounting, and as I've already mentioned, DevOps is *really* new. And, unlike accounting, DevOps lives in field that is itself constantly evolving and changing. So don't expect a lot of concrete, "here's what you must do" rules and regulations. Instead, the practice of DevOps is currently 80% theory, 10% what people have experienced so far, and 10% pure guesswork. There are a lot of companies experimenting with DevOps approaches, so as an industry we're still figuring it out. Much of this book will focus on what's been done successfully elsewhere, and look more concretely at what Operations delivers in those situations.

It's an Approach

And understand above all that DevOps *is* a technology management approach. It suggests ways of managing projects, ways of managing software development, and ways of managing operations. That said, without management buy-in in your organization, you can't do DevOps. So if you're thinking, "well, my organization will never get behind this idea of developers being able to push code into production," then you might as well stop reading right now, unless you're just interested for curiosity's sake. This book, at least, isn't going to make the case for DevOps in a big way - that's been done elsewhere. This book kind of assumes that you've already accepted the benefits of DevOps, and that you're interested in digging a little deeper into what that means for a traditional IT operations team.

There's No Such Thing as a DevOps Team

And let's be very, very clear: you cannot have a "DevOps Team" in your organization. That's nonsensical. DevOps is a *management approach* that encompasses software development, managers, and operations in a single body. Everyone works together to smooth the creation and deployment of applications. It's possible that only one of many internal projects will be conducted in a DevOps fashion - but given the kinds of changes Ops will need to make in order to facilitate the DevOps approach, it's going to be tricky to "do" DevOps in a piecemeal fashion. Just be aware of that - DevOps is about changing the way you do business, and if you haven't bought off on that idea, then it's always going to feel a little scary and awkward.

You *can* have specific teams or projects within your organization acting in a DevOps manner, provided that team is sufficiently cross-functional to provide all the disciplines of Dev, Test, Ops, and so on that are needed. So the whole IT estate doesn't need to "go

DevOps," but an individual project might. That said, having *just one project* run in a DevOps fashion can be sticky, because at some point it's going to run up against your "normal" IT operations, and the two might not get along.

So, you *very well might* have teams that behave according to DevOps principles, and you may call it a "DevOps team" if you have only one. But it's wrong to think that DevOps is implemented by some team dedicated to DevOps implementations. It's not "the team that handles DevOps for us," although it might be "a team that behaves according to DevOps." That's a super-fine line, perhaps, but it's an important distinction.

What DevOps *Isn't*

Given that DevOps is a philosophy... a management approach... and the combination of multiple IT disciplines... it might be easier to quickly look at some of what it *isn't*.

- DevOps is not Agile. That said, your teams might indeed use Agile as a development methodology within an overall DevOps-style approach. Agile is certainly DevOps-compatible, and, like DevOps, values short, continual improvement.
- DevOps is not Continuous Integration. That said, CI is often a part of DevOps-style behavior. The two can be really closely related, in fact - so closely that it's hard to tell the difference. I suppose you could argue that it's difficult to practice the DevOps philosophy without using CI as an enabling implementation, but you can definitely have CI without behaving like a DevOps organization, so the two aren't exactly the same thing.
- DevOps isn't "the developers running Operations." If anything, it's Operations automating things to the point where Operations runs itself in response to authorized actions taken by other roles, including developers.
- DevOps isn't a software development methodology. See the first bullet, above. DevOps is what happens while software development is happening, and largely what happens when software development (or a cycle of it), is *done*. You still need to manage your software development - you just need to use a methodology that's DevOps-compatible.
- DevOps is not automation. However, you can't have DevOps without automation. Automation is perhaps the biggest thing that Operations brings to the DevOps table, in fact.

Further, it actually seems to be an unstated goal of many DevOps champions to *avoid* the creation of any kind of trademarked, rigid, rulebook of "how to do DevOps," a la ITIL or TQM or something. *This* book certainly doesn't attempt to provide "rules;" the goal here is to provide some understanding of what DevOps' broad goals are.

What Does DevOps Look Like?

If we're going to concentrate on IT Operations' role in a DevOps organization, it's useful to think about what a DevOps project actually looks like. What, exactly, is IT Operations providing? What capabilities does the organization need? So let's take a super high-level look at a DevOps-style project, and what it involves. We'll dig deeper into various pieces of this in the remainder of this book.

HOWEVER, I want to emphasize that you can't achieve DevOps entirely within the Operations team. DevOps is about thinking of your entire *system* (a very Deming phrase), from the people who write the code to the people who use the code, and everything in between. The Operations team has a *contribution*, as do many other teams and roles.

There are a *lot* of people talking about DevOps these days, and so there are a *lot* of different opinions on how a "DevOps project" should work. In looking for a concise, high-level explanation, I was most taken by a description of how Spotify organizes their IT efforts. While a lot of that description focuses on how the software developers are organized, the interesting bit for me was that their IT Operations' team *main* job was to create units of automation so that the developers could deploy code to test, QA, and production on their own. Operations, in other words, facilitated a safe and managed connection between developers and application (service) users. Ops more or less arranged things so that Ops itself "got out of the way," within a managed and controlled framework of activity.

This is the heart of DevOps, and if it makes *your* heart skip a beat, then you have to remember that DevOps is a very different philosophy than what you've done before. In the past, QA and Operations were usually separate teams within IT. Code went from developers to QA and back again, until QA passed it, and then Operations worked on deploying the code. The intent of having these "gates" between roles was to make sure nobody did anything they weren't supposed to, like deploy unapproved code to production. This created several distinct problems:

- Developers became lazy. They knew QA was checking their work, and so they concentrated less on producing quality code. QA, in turn, had to take *their* job more seriously, and so organizations started investing heavily in QA automation. As a result, the organization spent a ton of time and money enabling developers to do their jobs less well. This was good for nobody. Nobody's saying that testing isn't important, just that the dev-versus-QA approach hasn't been massively beneficial or efficient.
- The organization developed a natural us-versus-them attitude, which is probably how *your* organization behaves right now. At the very least, it's no fun. After all, we're all *supposed* to have the same goal - delivering software and services to users - so we're

supposed to be in it together. In the worst cases, the inter-departmental rivalry becomes truly toxic, making for an unpleasant and unproductive workplace.
- Operations made mistakes simply because *they didn't write the code*, and developers had little incentive to write code that was easy to deploy, manage, or monitor. Developers threw the code "over the wall" and Operations just had to deal with it - increasing the tension between the departments.

All of this conspired to create something that is essentially the antithesis of DevOps. Software releases are slower, because of the implacable march of code from development through to QA, through to production. Operations basically lives in fear of new code, because they know little about it, and it wasn't necessarily designed with ease-of-operating in mind. Slower releases meant more pressure to pack more features into those releases, so each release became a "win," which simply made the process even worse.

DevOps, by contrast, envisions application and service delivery than constantly pushes small, incremental updates to users, with a minimum of operational overhead. Smaller releases are easier to code and test, and with the right approach, safer to push into production on an ongoing basis. But in order for all that to happen, everyone has to work together. The hard line between developer and operations has to become fuzzy.

In a DevOps environment, things work differently. Here's a super-simplified look:

1. Developers code, and check their code into a repository.
2. At some point, the repository's current code is pulled and built into an application.
3. Tests - usually automated, and created by developers - are run, including individual models, integration tests, and even user acceptance tests.
4. If the tests succeed, the build is deployed automatically into production (or at least into some deployment cycle).
5. User feedback is collected, feeding the next iteration of the cycle. Return to step 1.

Parts of this can be extremely automated, and parts - like user acceptance - may still be done manually by human beings. The point is that you create as few barriers as possible between coder and user. That *does not mean* there are no *checkpoints* along the way - that's what testing is all about, after all - but you don't put one part of the IT team in charge of "stopping" another part "from doing something stupid." DevOps, as a philosophy, implicitly means that you trust your team. If you don't trust someone on your team, you have an HR problem, and you should educate them so that you *do* trust them, or fire them and replace them with someone you *do* trust. If your company "would never let a developer's code get into production without thirty other people approving it first," then *you can't do DevOps.* That's what I was writing earlier about management buy-in being the first step.

The idea behind DevOps is, as I've noted, to smooth the path between coder and user, so that small, incremental application updates can be pushed more or less all the time. As user feedback is received, coders respond and updates are pushed.

Incidentally, here's a really great explanation of what DevOps is - and what it isn't. It's a long article, but it's worth reading, and you'll notice how much management buy-in is needed for all of those things to work.

So, *for the purposes of this book*, we need to look at some of the things needed to make step 4 happen, and a little bit about what's needed in step 3 as well. Again, we're going to focus mainly on processes and practices; you'll definitely need some technology to *implement* those in real life, but the exact technologies you choose will depend on your specific environment, so we'll keep this a little more abstract for right now.

Operational Capabilities of a DevOps Environment

So what are some of the broad capabilities you need to implement in a DevOps environment?

Automated Environment Creation

First, and possibly foremost, you need the ability to automatically and consistently spin up environments. That's a huge deal, and it isn't easy.

- Automatically: This means enabling a variety of *authorized roles* within your organization to start environments on-demand, without involving any human beings. This might be a developer spinning up a development or test environment, which might be something they need to do several times a day. It might also be an automated process spinning up an environment in which to run acceptance tests.
- Consistently: The environments that are spun up must accurately reflect the final production environment. There are two ways to do that:
    - Come up with a method of creating environments, and use that to also create the production environment as well as whatever other environments are needed. That way, you know they all match.
    - Come up with a method of *modeling* the production environment, and then applying that model to whatever other environments you spin up.

Emerging configuration management technologies - such as Microsoft's Desired State Configuration, or products like Chef, Salt, Puppet, and Ansible - are examples of tools that help implement some of these capabilities. When you can write some kind of configuration document that describes the environment, and then have a tool that can implement that document wherever and whenever you want, then you're getting close to the necessary capability. Containerization is another enabling technology that can help in this space, since it helps abstract a number of environmental variables, reducing variation and complexity.

It's easy to understand why this is such an important capability, though. If you can guarantee that everyplace an application might run - development, test, or production - is exactly the same, all the time, then you're much less likely to run into problems moving the code from environment to environment. And, by giving other roles - like developers - the ability to spin up these accurate environments on demand, you help facilitate more real-world testing, and eliminate more problems during the development phase.

I don't want to downplay the difficulty involved in actually creating this capability, nor do I want to dismiss the management concerns. Environments take resources to run, and so organizations can be justifiably concerned about having developers spin up virtual machines willy-nilly. But *we're not talking about unmanaged capability.* That's something that kills me every time I get into a discussion about DevOps with certain kinds of organizations. "Well, once we give developers permission to spin up whatever VMs they want, that's the end of the world!" and they throw up their hands in defeat. But that's not what we're talking about.

The reason DevOps has "Ops" at the end of it is because Operations *doesn't go away*. Developers don't "take over." Our job is to provide developers with a *managed* set of capabilities. So yes, a developer working on a project should be able to spin up a virtual environment without anyone else's intervention, and they should be able to recycle - that is, delete and re-create - that environment anytime they want. That doesn't mean they get to change the environment's specification on their own, nor does it mean they get free reign of the virtualization infrastructure.

Let me offer you a really simplistic, yet incredibly real-world, example of this. Amazon's Elastic Beanstalk service is designed to spin up new environments - that is, virtual machines - more or less on-demand in response to customer load. Each new virtual machine starts as an identical copy of a base operating system image, and each new virtual machine can load content - like a web site - from a GitHUb repository. So right there, you've created some of the automation and consistency you need. With a button push, or in reaction to user load, you can automate the creation of new environments, and because they all come from known, standard sources, they'll be consistent.

It's extremely likely that developers will need environmental changes beyond what's in the OS base image, and so developers can specify additional items. They can set environment variables, specify packages to be downloaded and installed, and so on. In the past, a developer would have tinkered with their development environment until everything worked, and then hopefully communicated the results of that tinkering to someone in Operations. Ops would then, hopefully, faithfully re-create what the developer did. But did you get the right versions of the packages? Did you set all the environment variables?

In Elastic Beanstalk, though, developers don't just "tweak" the environment. That's because every time a virtual machine shuts down, it vanishes. Any tinkering that was done is gone. On next startup, it reverts back to that base OS image. So, as part of the project's source in GitHub, developers can specify a configuration file that explicitly *lists* all the extra packages, environment settings, or whatever. Because that configuration information is part of the GitHub source, *every new VM* created by Elastic Beanstalk will be created with those exact same settings, every time.

This is a very DevOps approach, and in this case, Amazon has taken on the role of "Ops." If a developer wants to make an environmental change, they modify the project's source, and then tell Amazon to recycle the environment. Everything shuts down, and a whole new, fresh environment spins up. It's completely documented, so if it works the way the dev wants, then it'll be perfect when it's used for test, production, or anything else. And, in a typical cloud-centric way, Ops - that is, Amazon - doesn't have to be manually involved in any way. They've created automation interfaces that let any authorized user spin up whatever they want.

As a sidebar, this DevOps idea is a kind of follow-on to the concept of "private cloud." Private cloud simply means running your private IT resources in a way similar to public cloud providers - meaning *automation* on the Operations side. You come up with a way of specifying who can do what, and then you let them do it on their own. With a public cloud provider, permissions more or less consist of "whatever you want to pay for," but in a private cloud situation, permissions can be much more granular or even completely different. Nobody's suggesting that you build your own AWS or Azure; that's not what private cloud means. But you'll find that the private cloud capabilities are the very ones that you need to provide, as an Operations person, to enable a DevOps approach within your organization.

Development and Test Infrastructure

As I described in the previous chapter, traditional IT management places some pretty firm "gates" between development, test, and especially operations - with "operations" being more or less synonymous with "production." In DevOps, we break that relationship and eliminate the gates. Operations is responsible for *infrastructure*, whether that infrastructure supports developers, testing efforts, or production users. And those different phases of the application lifecycle get much more tightly integrated. Some of the high-level things you'll need include:

- Source code repositories. Git is a common example these days, as is Microsoft's Team Foundation Server and others. What's important is that your developers' tools be tightly integrated with whatever you've chosen. Ideally, these repositories should have, or be capable of integrating with, some pretty deep coding of their own. For example, the repository should be able to run pre-defined tests on code before it allows check-ins, and might perform an automated build-and-test routine each time code is checked in.
- Dashboards. Developers and testers need access to the operational capabilities you've provided them, such as the ability to recycle a virtual development environment. *Ideally*, you can integrate this as part of their main tool surface, such as an integrated development environment. Being able to click one button to "compile that, spin up the dev environment, load the compiled code, and run the app" is pretty powerful. In cases where that level of integration isn't possible, then you'll need to provide some other

interface for making some of those activities easy to perform.
- Testing tools. A certain amount of testing needs to be automated, so that developers can get immediate feedback, and so that tests can be run as consistently as possible.

That last capability is perhaps one of the most complex. In one ideal approach (although certainly not the only one, and even this will be a simplified example), the workflow might be something like this:

1. Developer writes code.
2. Developer runs code in a "private" development environment, performing unit tests.
3. Developer repeats steps 1-2 until they're satisfied with the code, and then checks it into a repository.
4. Repository runs certain quality checks - which might simply enforce things like coding conventions - before allowing check-in.
5. If check-in succeeds, repository kicks off an automated build of the code. This is deployed to a newly-created test environment.
6. Automated testing tools run a number of acceptance tests on the code. This might involve providing specific inputs to the application and then looking for specific outputs, "hacking" data into a database to test application response, or so on. Creating these tests is really a coding effort in and of itself, and it might be completed by the developer working on the code, or by a dedicated test coder.
7. Test results are stored - often in a part of the source code repository.
8. If tests were successful, then the build is staged for deployment. Deployment might happen during a scheduled window following that build.

You can see that the human labor here is almost all on developers, which is one reason people refer to DevOps as a "software development methodology." But the Ops piece provides all the infrastructure and automation from step 4 on, enabling a successful build to move directly to production.

Obviously, different organizations will have different takes on this. Some might mandate user acceptance testing as an additional manual step, although Ops could help automate that. For example, after step 7 above, you might automate the creation of a user acceptance testing environment, deploy the code to that environment, and then notify someone that it's ready for testing. Their acceptance might trigger the stage-for-production step, or their rejection might feed back to the developer to begin again at step 1.

The point is that *Operations* needs to provide the automation so that this sequence runs with as little *unnecessary* manual intervention as possible. Certainly, *Ops* should never be acting as a gatekeeper. We're not code testers. If the code passed whatever quality checkpoints have been defined, then the code's ready to deploy, and we should handle as much of that automatically as possible. Even the deployment - once approved, and on whatever schedule we've defined - should happen automatically.

You can see that DevOps, as an abstract philosophy, actually requires a lot of concrete tooling. And you can perhaps see that, because organizations will all have different particulars about how they want to manage the process, it would be difficult for commercial vendors to produce that tooling. There's not really a "one size fits all" approach for DevOps, which means Operations will end up creating a lot of it's *own* tooling. That's where *platform* technologies come into play. They can provide a set of building blocks that make it easier to create those custom DevOps tools you'll need.

End-User Experience Monitoring

This is perhaps the most important part of a DevOps organization, and it's the easiest to overlook.

As an IT Ops person, you're probably already pretty familiar with monitoring, and make no mistake: it's just as important under DevOps as it was before DevOps. Monitoring not only to notify someone when something goes wrong, but also monitoring to help profile applications (and their supporting services and infrastructure), so you can proactively address problems before they become severe.

But IT Ops' definition of "monitoring" often isn't as inclusive as it should be. We tend to only monitoring *the things that are directly under our control.* We monitor network usage, processor load, and disk space. We monitor network latency, service response times, and server health. We monitor these things *because we can affect these things*.

One of the biggest collaborations a DevOps organization can have, however, is monitoring *the end user experience*. It's something we, as IT people, can't directly touch, but if the whole point of IT is to deliver apps and services to users (and yes, that *is* the whole point), then the end-user experience of those apps and services is quite literally *the only metric that matters*. Why do we measure network latency? Because it contributes to the user experience. Why do we measure service response time? User experience. We attempt to *indirectly* measure the end-user experience, because we've often no way of *directly* measuring it.

DevOps' philosophy of developers and operations collaborating comes to a pinnacle with end-user experience monitoring. Developers should build applications with the ability to track the end-user experience. For example, when some common operation is about to begin, the application should track the start time, and then track the end time. Any major steps in between should receive a timestamp, too, and that information should be logged someplace. In Operations, we need to provide a place for that log - that *performance artifact* - to live, and we need to provide a way for developers to access it. We need to baseline what "normal"

performance looks like, and monitor to track for declines in that baseline. Operations may be responsible for the monitoring itself, but developers, in their code, can give us the instrumentation to monitor what matters most.

If end-user experience numbers begin to decline - say, the time it takes to perform a common query and display the results starts to get longer and longer - then we can dig into more detailed instrumentation and see if we can find the cause. Is it network latency? Server response time? Any other correlations that might point to a cause? But by directly measuring *what our users experience*, we have an unassailable top-level metric that represents the most real-world thing we can possibly have on the radar.

I'm making a big deal of end-user experience monitoring not only because it's important and useful, but also because it's one of the easiest-to-grasp examples of what DevOps is all about. Developers have traditionally *cared* about users' experience (in theory), but they're extremely *disconnected* from it. Operations is very connected to what users experience (we get the Help Desk calls, after all), but we're relatively powerless to put our fingers directly on it. Through the collaboration that drives DevOps philosophy, though, developers and operations personnel can come together to do their collective job better.

IT Ops Skills in a DevOps Environment

So let's say you've decided, at least in theory, to help take your organization onto a DevOps standing. You've read about some of the high-level capabilities that you, as an Operations person, need to provide to the organization.

How do you do it?

In a word, "glue."

I'll say it again: DevOps is a *philosophy*. Accounting remains a good example. The accounting industry agrees, more or less, on what constitutes good accounting, and that's where GAAP comes from. Similarly, the DevOps industry is slowly coalescing a feeling of what "good" DevOps "feels like."

But every organization does it their own way. Look at how any one organization handles their accounting, in detail, and you'll see plenty of differences between other organizations. Perhaps auditors work a little differently, or perhaps a different job role is responsible for different accounting duties. Some companies need fairly simplistic accounting, whereas others need incredibly complex accounting that demands hundreds of people working around the clock. Although they're all operating on the same *principles*, their implementations vary widely.

So it is with DevOps.

In a small organization, accounting may be simple enough that off-the-shelf tools, like Quickbooks, are sufficient. In that size of an organization, DevOps might not even be a thing, because a company of that size might simply not be doing any "dev" to begin with. In a massive, multi-departmental enterprise, accounting might involve "off-the-shelf" tools that requires months and months of customization and tweaking. Similarly, DevOps in that same organization might involve customized tooling that uses generic building blocks... and a lot of custom glue.

Providing the operational infrastructure for a DevOps organization can be hacking at its finest. Yes, you'll find plenty of off-the-shelf technologies and products... but many of them will only get you to a certain point in your organization's goals. After that, it'll be a bit of customizing, a bit of "gluing" different tools together, and a bit of hacking around the rough edges. That will probably *always* be the case, just as it's still the case that large new deployments of accounting tools *always* take months and months. Nothing off-the-shelf can possibly fit every organization's needs, so you'll simply have to be prepared to do some customizing. Some hacking. Some gluing.

With that in mind, what are the right skills to have?

- The ability to learn quickly. You'll have to master new products and technologies on the fly.
- Creativity. You'll need to think of clever solutions to work around stumbling blocks. Don't expect everything to "just work" - it won't.
- Deep understanding of your platform(s). Whether you're working on Microsoft Windows, a Linux distribution, or some other platform, you need to know *deeply* how it works, because you're going to be interacting with it in the sub-basement level.
- Scripting. You're going to need to be *fluent* in the leading systems programming ("scripting") language(s) used on your platform, because that's the "glue" you'll use to stick different technologies together into a cohesive, custom solution.

This DevOps stuff is not for beginners, nor is it for the faint of heart. This is, in my own personal belief, why companies create job titles like "DevOps Engineer." Most of the DevOps community quite justifiably freaks out about job titles like that, because they're often a demonstration that *someone in the organization doesn't get it*. DevOps isn't a job role. However, in an organization practicing DevOps, there certainly are some skills that will come in handy, especially on the Operations side. Someone possessing those skills might justifiably be called a "DevOps Engineer," which is perhaps less cumbersome than "IT person who knows enough to make all these bits stick together so we can get the DevOps-enabling capabilities we need." That'd be a big business card. "DevOps Engineer" is probably also a title less demeaning to one's co-workers than "Cleverest IT Person We've Got," which is also usually the case.

IT Ops folks working to provide DevOps-compatible capabilities are often the more experienced, cleverer folks on the team. They often have the broadest experience and knowledge, and they're often the ones most eager to tackle a challenge.

By the way, notice how I phrased that. "...working to provide DevOps-compatible capabilities..." was a very deliberate phrase. A DevOps-practicing organization does need specific capabilities, and the Operations side provides some of them, in close collaboration with the Dev side. That doesn't mean you have a "DevOps Department," because that misses the point. "DevOps Engineer" as a job title is only legit if it means "Engineer Who Helps Provide Our DevOps-Related Capabilities." DevOps isn't something you *do*; it's something you *believe*, which in turn drives you to do things. If you believe in DevOps, your organization needs to behave in a certain way, and it needs certain tools to support those behaviors.

There are new Development skills that need to be brought into a DevOps environment, too. Developers have to focus more on building code that *can* be deployed, monitored, and managed in a DevOps-centric way. For example, in most Windows-centric environments, developers would often use the tools bundled into Visual Studio to create Windows Installer

packages for applications. Those packages weren't always easy to deploy in an automated fashion, may have required (or thought they required) Administrator privileges, and other elements that simply made deploying the code difficult and even dangerous. To "do" DevOps, that has to change. Operations needs to give Development the ability to seamlessly slide code into production - but Development needs to write code that supports that model. The burden is on both groups, as a combined team, not just on Operations to make things simpler.

Plan for Failure

> "Wait a damn minute," I can hear you saying, "sliding new code into production is what causes all the problems!"

Agreed. *Any kind of change* has the potential to create problems. The point of DevOps - and most particularly the Operations role in DevOps - is to *create an environment where you can fail quickly, and fix just as quickly* (thanks to Chris Hunt for that). If DevOps means constantly pushing out small bits of code, then you have to be prepared to - in Facebook's language - "move fast and break things." Eventually some release is going to be problematic, and so the role of Operations is *not to slow things down to avoid the problem* but rather to *hit the problem hard and fast*. Virtualization, as one example, gives us the ability to rapidly "roll back" entire operating environments to a "known good state," making the prospect of failure a little less frightening. *Plan for failure*, rather than trying to avoid failure entirely.

Ask yourself if you're the type of person who routinely plans for failure. For example, on every airline flight I take, I have a set of spare clothes in my carry-on, even if that's just my computer bag. I have a small stick of deodorant, because that's an item not included in airlines' amenity packs. I *assume* there will be a failure in the trip, and I have simple plans in place to mitigate that failure. *Few* people take these simple steps, though, and so when failure eventually does happen, they become angry, stressed, and uncomfortable - even when the causes of failure are completely outside human control, like weather. I plan longer layovers than most people - usually 2 hours domestically - and am often able to avoid a trip failure because of that extra margin.

In a DevOps environment, you have to accept that failure will occur. Your effort should go less into preventing that failure - especially through time-consuming "gates" that put a wall between coders and users - and instead put effort into being able to iterate and recovery quickly. In a true DevOps team, a buggy release doesn't mean you roll back to the last one - it means you release another one really quickly. That's moving forward, not rolling back, and having the capability to do that is the main hallmark of a DevOps-ready organization.

Operations as Development

There's an interesting piece of fallout to having an Operations team get more DevOps-supportive, and it's that the Operations team becomes a kind of special-purpose Development team. This fallout, in fact, creates one of the biggest misunderstandings about DevOps: the believe that DevOps means "Operations turning into coders."

DevOps does not *mean* Operations turning into coders. It *means* Operations working to smooth the path between coder and user. It *turns out* that the most common way for Operations to do that is by providing automation, and the most common way to provide automation usually involves some coding. So DevOps *usually results in* Operations turning into coders, at least to some degree.

Most operating systems that Operations will deal with have some systems programming-level language that's designed to facilitate operational automation. In Linux, for example, Perl and Python are extremely common scripting languages. In Microsoft Windows, Windows PowerShell has taken on that role. So this isn't programming a la C++, C#, or another "deep" programming language; it's "scripting," usually in a higher-level language that's more purpose-built for the task of operational automation. As I noted in the previous chapter, the main skill that Operations needs to bring to the DevOps picnic is skill in an environment-appropriate scripting language.

But once Operations begins producing units of automation - that is, *code* - Operations itself needs to start acting like a DevOps shop. Those units of automation are the application that the coder (Ops) delivers to the user (in this case, other roles in the IT team). So Operations needs the tools and management approaches that let them quickly iterate their code, test it, and deliver it to production. As users (in this case, that's probably developers) define new needs (such as the ability to deploy code to end-users), Operations must deliver.

This entire concept is often one of the biggest obstacles to organization-wide DevOps mentality, especially in shops that are heavily built on Microsoft Windows. The hurdle happens because Windows administrators, in general, haven't had decades of investment in coding and automation, in large part because the OS only started offering the capability in 2006, and didn't offer a significant capability until 2012. Administrators in the space simply haven't had the tools, and so they haven't learned the techniques. Change is always scary for some people (and for some organizations), and so the change of switching from GUI-based administration (which doesn't support DevOps) to code-based administration (which does), can be scary.

Many administrators - again, the Windows space perhaps has this the most - are accustomed to getting fully-fledged tools for administering their environments. They'll maybe complain that the tools don't work quite the way they want them to, but they're close enough. Moving into a DevOps-centric world, though, simply introduces too many variables. What kind of code are you delivering? What methodology do your developers use? What are the production concerns around stability and availability? How much room is there for error? What sort of maintenance windows are available? How do you communicate with the user base? The sheer number of variables means that pretty much every organization is unique, which means off-the-shelf tools simply can't be produced that are "close enough." As a result, DevOps almost demands that Operations build its own tools and processes, usually by "gluing" together off-the-shelf *platform technologies*. That's what I covered in the previous chapter, albeit from a slightly different perspective.

That "gluing" process is where Operations strays into its own development. You might be using Microsoft System Center Virtual Machine Manager to manage your virtual infrastructure - but you'll be writing some code to make it do what you want in accordance with your particular processes. You might use Chef to handle declarative configuration of virtual machine environments - but you'll be writing some code to tell Chef exactly what it is you want, and to manage custom elements that exist only in your environment.

Another result of this DevOps approach is that, once you get really good at it, you start to treat your infrastructure as code, and you start approaching infrastructure management in a more agile (if not Agile) manner. Virtualization in particular has made this tremendously easy, because we can tear down and re-create entire massive environments with the push of a button. Don't like the current environment configuration? No problem - modify the declarative configuration document and recycle the environment. Not happy with the result? Repeat. Reconfiguring the environment can (and should) be as easy as modifying a code-like structure, just as modifying an application is as easy as changing the code. In other words, once you're using code, or something like it, to describe how your environment should look, then you're basically treating the infrastructure *as code*. Development methodologies like Agile and Lean start to become an option for managing the infrastructure... and suddenly, you're looking a lot more DevOps-ish.

Mind-melding completely with all of these concepts - infrastructure as code, Operations as glue-coders - really opens up some possibilities. You're no longer constrained to this "big vendor" approach, where you have to find one vendor stack that meets all of your needs (which was never really practical anyway). Instead, you become *comfortable* dragging in components from multiple vendors as needed, because you grow confident in your ability to glue them all together into the Fraken-structure you *need*.

DevOps Doesn't Exclude Anyone

Because... well, because it's *called* Dev Ops, there's often this feeling that the philosophy excludes security... or network infrastructure... or graphics designers... or someone.

It doesn't.

If DevOps *means* any one thing, it means *everyone collaborating.* But what it doesn't mean is someone having a veto. For example, IT Security can't swoop in and say, "there's no way we can automate the deployment of that app Because Security." That's not collaboration, it's obstructionism. What they *can* say is, "in order to automate the deployment of that particular app, we need to make sure *x* and *y* are happening." Security, Development, and Operations can work together to *automate* those requirements, helping ensure they're carried out consistently every single time. Security *wins* because their concerns get met *as a part of the process*. Development wins because they get better insight into Security concerns. Operations wins because they get to stop being the middleman who has to reconcile everyone's crap.

But this is why DevOps *cannot* work without management buy-in from a very high level. Like, the CEO and CIO (or CTO). The bits of your company that have traditionally worked in their own little fiefdoms need to give up their royal styles and work *together*. "No" is never the answer; it's "here's how." That, as I'm sure you can imagine, can be massively difficult, politically, in some organizations. And *that* is where people fail at DevOps.

A DevOps Reading List

Many thanks to Chris Hunt (@cdhunt on Twitter) for providing this suggested reading list.

The Phoenix Project by Gene Kim
The Goal by Eliyahu M. Goldratt
It's Not Luck by Eliyahu M. Goldratt
The Checklist Manifesto by Atul Gawande
Thinking in Systems: A Primer by Donella H. Meadows
Lean Enterprise: How High Performance Organizations Innovate at Scale by Jez Humble
Becoming a Technical Leader: An Organic Problem-Solving Approach by Gerald M. Weinberg
Theories of Work: How We Design and Manage Work by David Joyce
Quiet: The Power of Introverts in a World That Can't Stop Talking by Susan Cain
Continuous Delivery by Jez Humble and David Farley
Test Driven Development by Kent Beck