January 27, 2022

What is DevOps? The History, Definition, Best Practices, Use Cases + Examples of DevOps

GLOSSARY | DEVOPS

We've seen a huge explosion of interest in DevOps over the last few years. But for people who are new to these ideas, it's not always obvious what DevOps actually entails and what the benefits are, particularly in larger environments.

Additionally, given that DevOps started off as a grassroots movement and continues to be heavily influenced by practitioners, it may be even less clear to senior managers what it's all about.

So, what is DevOps all about? And what do you need to know to succeed? Find out in this blog.

Table of Contents

Back to top

What is DevOps?

> The word "DevOps" is a combination of the words "development" and "operations." **DevOps is a set of practices and tools that helps organizations deliver software faster.**

DevOps looks a bit different for every organization that does it, which makes it hard to strictly define DevOps. But in a way, that's a good thing: Without a hard-and-fast definition of DevOps, the DevOps movement can grow and respond to the changing landscape of infrastructure options and best practices instead of being locked into a prescribed template.

Back to top

DevOps Examples

> DevOps involves a development team using continuous integration and continuous delivery (CI/CD) practices to develop, release, and monitor software. DevOps teams **develop code, automate testing, deploy to production, and iterate for improvements**.

Definitions and theory are useful for guidance, but the best way to understand DevOps is to get an idea of what it looks like in the real world.

Here's an example of a DevOps process in action:

- **Planning:** The dev team works with stakeholders to get information and plan new software features or updates.
- **Development:** Developers write code for those new features and updates using a version control system and committing code changes to a shared repository.
- **CI/CD:** A CI/CD tool detects changes in the repository, tests it to make sure it doesn't conflict with existing code, and delivers it to a staging environment.
- **Staging:** Human testers and other stakeholders can get their hands on the application in a staging environment before it goes to production. In this step, they might do usability testing, functional testing, and other forms of QA.
- **Monitoring:** Automated tools monitor the performance of the application as users interact with it. It can relay issues and other relevant information to the development team to fix.
- **Deployment:** The team makes the latest version of the application available to users.
- **Iteration:** The DevOps team solicits feedback from users and stakeholders about the application. They can use that information – what works and what doesn't – to plan new features or updates. Then the DevOps cycle starts again.

Back to top

How DevOps is Defined + Key Ideas in How DevOps Came to Be

Before we delve further, let's set the stage by looking at why DevOps matters at all, and some of the landscape changes that have enabled DevOps.

A Context for DevOps

Today, every organization depends on software. Retail, logistics, government, scientific research, tech, education, financial services — every sector needs some sort of software to meet customer and user needs.

The expectations people have of software have changed dramatically over the last decade: They expect reliable and convenient services that are regularly improved. The complexity of our computing infrastructure increases continually, as does the pressure to deliver more software, more frequently, and at higher standards of quality.

The normal way of delivering software in organizations has been incredibly dysfunctional, because incentives just haven't been aligned. Too often, developers are incentivized solely to deliver new features; their responsibility ends as soon as the software is handed to the operations teams to deploy. Operations teams have been incentivized to keep infrastructure as stable as possible; their responsibility for software delivery starts only once they've been handed the software to deploy.

And of course, operations normally has plenty of responsibilities in addition to deploying software, including managing costs, user accounts and overall capacity, plus ensuring overall security.

The incentives of these two groups are fundamentally opposed. We can't fix that situation with technological practices alone.

If we all recognize how broken this situation is, why did it take this long for us to work out a collection of practices to fix it? We see two main trends that led us to DevOps.

APIs

In recent years, we started getting better and easier APIs around infrastructure management. Being able to invoke APIs to do work like cloud provisioning, along with the rise of infrastructure-as-code software like Puppet, meant that we could start actually treating our infrastructure like software.

This in turn meant we could take advantage of everything we've learnt in the software engineering field over the last couple of decades — for example, the value of version control, branching strategies, and code review.

Increasing prevalence of simpler and easier-to-use APIs (such as the widespread adoption of **RESTful APIs**) made it easier for non-developers to use them, which resulted in a wider group of people within the operations field being able to do development as opposed to scripting. This pushed more operations people into learning basic software engineering practices.

Agile

Second, we had a general recognition that agile software development was a better way of working, resulting in higher-quality software that could be delivered more quickly.

The Puppet and wider DevOps communities started generating more reusable content they could share, which naturally started exposing more and more sysadmins to current thinking around software development practices.

The growing popularity of agile methodologies resulted in more releases, putting even more pressure on operations teams and making it more urgent to improve how they managed infrastructure.

As The Agile Admin put it:

> "DevOps is also characterized by operations staff making use of many of the same techniques as developers for their systems work."

If you work in operations, "doing DevOps" and making use of these techniques doesn't mean you need to pick up *all* the programming skills of a senior software developer. We've always done development in operations, whether it be shell script snippets, useful shell aliases or batch files. It just wasn't expected that we absorb the principles of software engineering and delivery, and we didn't think about it as development.

You don't need to become a professional full-time software developer to "do DevOps" as an operations person – you just need to understand basic software practices like version control, peer review, releases and testing, and have sufficient facility with high-level programming languages and frameworks to get the job done.

It's critical to note that simply implementing these software practices inside an ops silo isn't sufficient. The problem isn't just a technical tooling or practice issue. Cultural and process changes around shared responsibility are also necessary.

DevOps and CAMS

In 2010, John Willis and Damon Edwards coined the term CAMS: Culture, Automation, Measurement, and Sharing. The CAMS model is a set of principles and values used in DevOps.

The CAMS model is a great place to learn what DevOps is and settle on a definition of DevOps. It's also a good framework for understanding DevOps from the perspective of a practitioner or team leader.

Culture

The culture of DevOps is about:

- Communication and responsibility sharing
- Accepting failure
- Cross-functional alignment
- Empathy

DevOps is about much more than simply applying agile principles to infrastructure management. One important reason for this is the strong cultural and organizational dimension to resolving the conflict between incentives for development and operations teams.

DevOps places a strong focus on cross-functional teams working together across the divide. Plus, there's an understanding that failures must be evaluated objectively, with no-blame postmortems.

Perhaps most important of all, DevOps depends on the understanding of collective shared responsibility. Especially as organizations grow, you must nurture this notion of shared responsibility as an organic part of the culture, because with growth comes the tendency to separate different areas of responsibility into organizational silos.

And silos lead, in turn, to different ways of working, different incentives, and even different subcultures. So preventing and working against this tendency requires active, conscious effort.

Automation

It's pretty much impossible to envisage any kind of DevOps approach that doesn't involve a high degree of automation, and that doesn't heavily rely upon representing your infrastructure in a code-like manner.

The ability to use software engineering processes to treat your infrastructure like software is, of course, critical for delivering reliable infrastructure quickly. At the same time, it gives you common practices and tooling between application development and infrastructure deployment.

As the line between app dev and infrastructure deployment becomes more blurred, your ability to deliver software quickly, and with fewer errors, improves dramatically.

Automation and self-service infrastructure allow you to minimize cycle time, as individuals can not only rely upon predictable automated outcomes, but also, with the addition of self-service, those individuals and their teams can run at their own cadence and avoid external bottlenecks. This allows for faster experimentation and more agility to handle changes in direction, as well as increased reliability and quality.

Discover how Puppet can help you automate and manage cron jobs and regularly scheduled tasks >>

Measurement

Automated processes enable far more reliable measurement. Measurement of the whole system in turn enables identification of bottlenecks, and once you know where the bottlenecks are, you can work on removing or mitigating them.

The importance of measurement comes as no surprise to anyone in IT or business in general. As the saying goes, you can't improve what you don't measure, and this is, if anything, even more true when you're talking about using DevOps to improve the whole software delivery lifecycle.

Additionally, the existence of well-understood metrics that are generated by automated systems can go a long way toward reducing friction across teams when you're investigating issues. These metrics are objective, and having them available can take a lot of the emotional heat out of issue discussions.

Sharing

Sharing code, tooling and processes has a number of advantages. For one thing, it's far more efficient for different tech teams to use the same tools; you save time on handoffs and integrations, and also actual cash.

There's a more subtle saving, too. Tools shape people's thinking, and so when people use very different tools, they often think very differently. A shared toolset can go a long way towards helping people understand each other more quickly, and this empathy helps across all the interactions people have with each other within a company.

Sharing actual code is also well understood by most people in software. There's not much point in recreating something when you can simply reuse existing code that's known to work well.

Puppet modules are a great example of this (though not the only example). Eighty percent of what a sysadmin does every day is the same or similar to what other sysadmins do for 80 percent of their day, so why not share the code that automates it? Especially when the data that's specific to each workplace has been abstracted away from the operations code, as it is in Puppet.

Going back to the bigger picture, sharing between teams is a key tenet of DevOps. That sharing takes place in many formats and locations — for example, in retrospectives, in lunch-and-learns, in experiments and their outcomes. And of course, a lot of sharing takes place in the open source world, where Puppet and so many other DevOps technologies have their roots.

The CAMS definition is great, but we find it doesn't always resonate with senior managers and executives who manage multiple teams and departments.

Gene Kim's The Three Ways of DevOps

> The Three Ways of DevOps are core principles of DevOps patterns originated by author Gene Kim. They include Systems Thinking and Flow, Amplify Feedback Loops, and Continual Experimentation and Learning.

In 2012, CTO, researcher, and author Gene Kim defined The Three Ways and although we've seen some tweaks over time, the core ideas are very sound and do tend to resonate well with senior managers.

The First Way: Systems Thinking and Flow

This is about considering the performance of the whole system, not just an individual component or department. Don't just optimize locally, but look for global throughput and flow.

It can be difficult inside a large enterprise environment to actually gather enough information to do this accurately, but if all of your teams are automating processes, measuring them, and sharing their status and results, then it becomes more workable.

The Second Way: Amplify Feedback Loops

Shorten your feedback loops, and amplify them, so people know early what the issues are, and they can be resolved quickly. This allows the whole system to be improved.

Because the environment in which we work is continually changing — and doing so faster and faster — fast feedback loops are necessary to keep the whole system continuously improving in response to that continual change.

The Third Way: Continual Experimentation and Learning

How do you adapt to continual change? By continually experimenting and learning from the results of your experiments.

Experimentation is enabled by a number of things, including automation (to let it happen fast) and measurement (to understand results). But there's a lot more to achieving continual experimentation than just tools and processes. You must create and nurture a culture that supports, encourages, and rewards experimentation.

As the DevOps Report has shown, middle management has a powerful role to play in creating and sustaining a learning-friendly culture. It's middle management that takes the strategy from upper management and translates it into tactics to be carried out by practitioners, and it's middle managers who are most critical in enabling (or fighting against!) desired cultural changes.

This doesn't mean upper management is off the hook. They also have an important role in nurturing experimentation and learning. Upper management needs to support the process of learning from experiments and also needs to model acceptance of failures so that the middle management layer follows suit. After all, experiments that fail are actually a success if they disprove a hypothesis, helping the organization to make important strategic decisions.

When upper management accepts the importance of learning and its role in business growth, then the budget gets allocated to research, learning, and staff development. The rewards system gets adjusted to incentivize creativity and risk-taking.

And in this kind of environment, people feel safe to try things out, disagree, advance hypotheses, and develop new ways of measuring and learning. In fact, people become empowered, more engaged, and more deeply invested in the success of the entire organization.

Back to top

DevOps: The Key to Organizational Success

We hope we've been able to give you an idea here of why DevOps is rapidly becoming the go-to philosophy of intelligent, forward-looking executives. For the latest on DevOps, check out the State of DevOps report.

GET THE DEVOPS REPORT

This blog was originally published on January 30, 2017, and has since been updated for accuracy and relevance.

Learn More About DevOps

- Read our guide to getting started with DevOps.
- Watch our webinar on DevSecOps: Integrating Security into the Enterprise Software Delivery Lifecycle.
- Wondering about infrastructure as code, but afraid to ask? We've got you covered: What is infrastructure as code?
- Explore these DevOps examples of automation in practice.
- How DevOps tools can support your DevOps initiatives.

Back to top