What's DevOps?

IT has a cool new name

Justin
Jan 5, 2021

53

The TL;DR

DevOps is a process (e.g. a bunch of key practices) that helps teams take software they've built and **make sure it works well at scale**.

- Building software is only part of the battle - you need to **distribute it to your customers**, and it needs to work for them, all the time
- Mass adoption of the cloud has moved software to the web and scaled it by a lot, which makes actually **running that software kinda hard**
- DevOps can be a team, but it's mostly a process that centers around 4 big things: **pre-release building**, **CI/CD**, **infrastructure**, and **monitoring**

Entire public companies like JFrog and Datadog exist within the DevOps workflow, so it's worth getting to know.

How we got here

To understand why DevOps as a concept has gotten really popular recently, you need to understand software delivery and how that's changed over time.

→ **What it means to deliver software**

When a team of engineers builds an application, it's just a bunch of code. And that code needs to run somewhere for users to access it and get value out of it. During development, developers tend to run that code on their personal computers, but that's obviously not scalable - so when they're ready to share the software with the world, they'll **put that code on big powerful servers**, and let people access it via a domain name like https://technically.dev.

Sometimes, the codebase might be split between those servers and your laptop. When you run Apple's Mail app, the actual app (the **frontend**) is running on your computer, but the **backend** (storing the mail, sending emails, etc.) is happening on the servers of whoever is providing your email address (Google, if you're using a Gmail address).

So in general, delivering software means taking the application you've built and figuring out how to distribute it widely to whoever wants to use it.

→ How things have changed, a lot

Just 10-15 short years ago, a lot of delivering software meant literally delivering software - Microsoft Office used to **ship to you on a CD** that you'd install directly to your computer. And it wasn't web-based, so you didn't need internet access to use Excel or Powerpoint. The public cloud (AWS and co.) didn't exist, so if you needed to run software on a server or two, you'd need to literally build that infrastructure yourself, which used to cost millions of dollars up front. So naturally, software delivery was bespoke and on the slower side.

But then people started consuming software via the internet, and public clouds like AWS made it cheap and easy to use a server without having to build a data center. That **fundamentally changed 3 things**:

1. *The scale of software increased* - software is generally used by a lot more people than in the past. You could realistically need to support millions of users for your application
2. *Infrastructure got more complicated* - we're moving towards increasingly specialized managed infrastructure for different parts of the stack. Generally, you don't just throw your code onto a box and forget about it anymore
3. *Teams started releasing a lot more often* - changes in philosophy mean teams are now shipping code changes to users as often as multiple times a day, which means many many more opportunities to break things

With these fundamental shifts happening, teams needed to start building processes for managing this stuff, and making sure their apps didn't constantly break and disappoint their users. And that's basically what DevOps is - making sure your app works at scale.

Pillars of DevOps

Different articles across the web will describe DevOps as a philosophy, a process, a set of practices, or even a team - the easiest way to understand it for me personally has been as a set of responsibilities and practices that *someone* (ideally everyone) at your company should be focusing on.

→ Pre-release building

Making sure your software can get delivered at scale starts with building it in a fundamentally sound and scalable way. There are some basic principles to follow here, like separating frontend and backend logic, trying to keep your app not huge, optimizing database queries, and things like that.

→ CI/CD and testing

Once software is ready to ship, it needs to get rigorously tested to make sure you haven't missed any edge cases and that you can have reasonable confidence that it won't break for your users.

> 🧠 **Jog your memory** 🧠
>
> **CI/CD** (continuous integration and delivery) is the philosophy of getting your tests integrated with your code automatically, and shipping code very frequently. We wrote about it more in depth here.
>
> 🧠 **Jog your memory** 🧠

Let's say we're building a special new onboarding flow for our users that helps personalize their experience by asking them what their company role is (e.g. data scientist, engineer, product manager). An automated testing suite might run through this flow as a new user *and* an existing user, and test to make sure that the flow only triggers for new users.

Testing generally happens via some CI pipeline, and that generally exists on a separate server. Teams can set up their own CI server, or use something like CircleCI or Jenkins.

→ Infrastructure

Most modern startups will be running their code on servers from AWS, Azure, or GCP. There's a good chance you'll be using 10+ different services from your cloud provider, and you'll need to figure out how to network those services together, integrate data, and a lot of other fun sounding stuff that engineers definitely like doing.

> 🚨 **Confusion Alert** 🚨
>
> If the point of things like AWS is to *simplify* infrastructure for developers, why does it seem like a full time job to just even *understand* their basic product suite? It's an interesting question, and one that DigitalOcean has tried to build a business on (we're simple!). Just note that AWS offers literal certifications for using its own products.
>
> 🚨 **Confusion Alert** 🚨

Luckily, ~~the AWS interface is fantastic~~ the AWS interface is so bad that people are literally building entire companies to fix it.

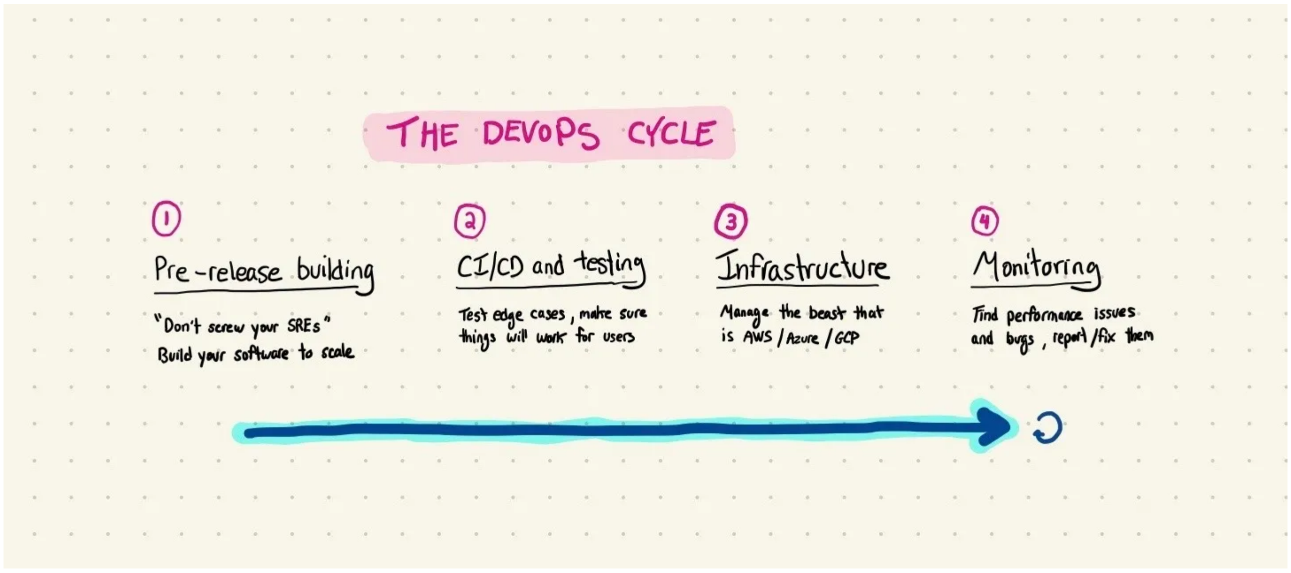
→ **Monitoring**

Once your software is up and running on your cloud provider of choice, you need to monitor it to make sure things are working smoothly. That monitoring generally splits into two categories:

1. *Server performance monitoring*: how much CPU, RAM, Disk, and IOPS is my server (or Docker container) using
2. *Application performance monitoring*: how quickly are my pages loading, what's the latency on my API requests

Teams can build custom stuff for this, but nowadays tools like Datadog and Grafana are pretty popular across the board, which you'll often hear referred to as **observability**.

Monitoring will usually uncover problems, which will be caused by bugs, which will get reported and fixed, and then require a new version to get released. And so the cycle begins again.

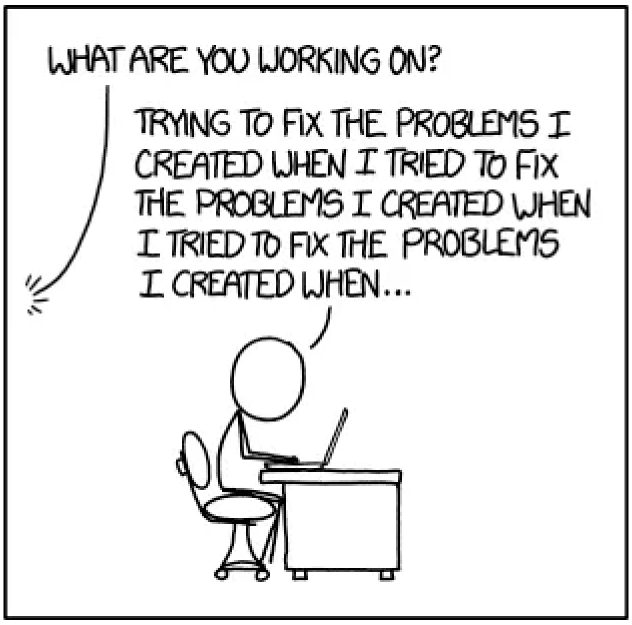

THE DEVOPS CYCLE
1
Pre-release building
"Don't screw your SREs" Build your software to scale
2
CI/CD and testing
Test edge cases, make sure things will work for users
3
Infrastructure
Manage the beast that is AWS / Azure / GCP
4
Monitoring
Find performance issues and bugs, report / fix them

Team roles and responsibilities

As companies get larger, they will often start hiring entire engineers (and then teams) to focus on application delivery. But at the earlier stages, DevOps is the responsibility of the whole engineering team. A couple of useful things you might want to know about how teams of developers do this stuff:

→ On-call rotations

Software has a lot of bugs, and fixing them will never fully solve the problem.


WHAT ARE YOU WORKING ON?
TRYING TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN I TRIED TO FIX THE PROBLEMS I CREATED WHEN...

If every engineer on your team needed to stop what they're working on to fix bugs, things would never get done. So engineering teams will generally designate one (or a few) engineers to be “on-call” - which means for a rotating period of a week or two, their job is to respond to operational issues and fix critical bugs.

→ **Alerts and paging**

Software like Datadog integrates with your communication platforms (email, Slack), so you can get notified when your servers are down, or your users API is seeing huge latency spikes. Engineering teams will often use something like PagerDuty for more pressing issues, which integrates directly with your codebase (or something like Datadog) and can alert on-call engineers directly.

If an engineer friend has ever said “I’m getting paged” and pulled out their laptop at a bar, this is why. And pour one out for them.

→ **The SRE**

The concept of the SRE has emerged from Google recently - it stands for **Site Reliability Engineer,** and it basically refers to someone whose full time job is making sure software is running smoothly (not actually building that software themselves). Here’s a quote from the aforementioned site:

> *Our job is a combination not found elsewhere in the industry. Like traditional operations groups, we keep important, revenue-critical systems up and running despite hurricanes, bandwidth outages, and configuration errors.*

This is ridiculous, as it is found all over the industry. But that’s none of my business.

Terms and concepts covered

`CI/CD`

`Observability`

`On-call`

`SRE`

Further reading

- Strategically, there's an open question as to where CI/CD tools fit in the stack. Companies like Gitlab think things should be integrated into your remote repository provider, while CircleCI thinks we should maintain a separate server, and the recent trend towards build hooks seems to indicate things should live nowhere (?)
- If you search for "DevOps" you'll be inundated with content marketing - the space is quite hot right now, and a massive piece of the software market (whatever that means) is indeed happening post-building

Comments

Write a comment...