DevOps as a Culture

Contents

Overview

The term **DevOps** is a difficult one to define as different organisations have varying approaches to implementing DevOps. It is best described as a *cultural* approach to software development project structure with a particular philosophy designed to achieve the following:

- Increased collaboration
- Reduction in silos
- Shared responsibility
- Autonomous teams
- Increase in quality
- Valuing feedback
- Increase in automation

This module explores the drawbacks of traditional software development approaches and how the DevOps methodology is used to overcome them.

How Things Used to be Done

Traditionally, software companies are structured as separate, stratified teams for **development**, **quality assurance** (sometimes known as testers), **security**, and **operations**.

These teams tend to have varying and sometimes conflicting goals and there is often poor communication between them, regularly resulting in work that is out of sync with other parts of the organisation. These isolated teams are referred to as *silos*.

As a result, this structure regularly results in *slower releases*, *wasted time* (and therefore money), and *blame cultures*, where production problems become the fault of another team.

How DevOps Changes Things Up

DevOps is based off of the agile project management, an approach to projects that promotes:

- **Individuals and interactions** over processes and tools

- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

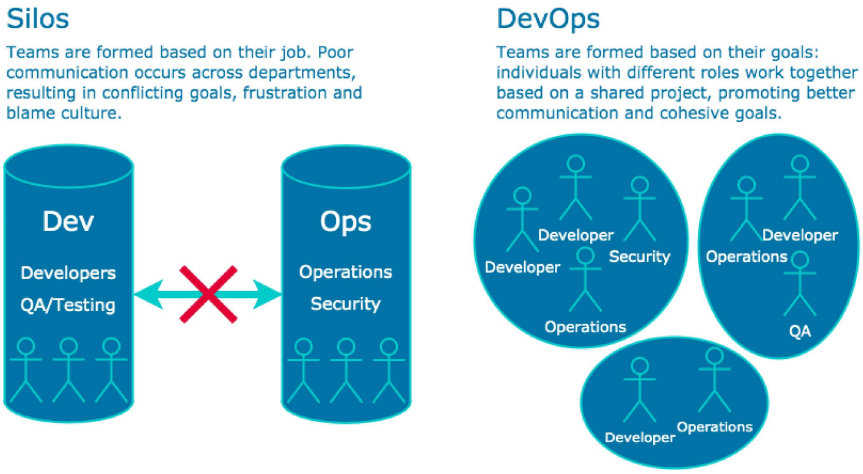
You can read more about Agile here, but in short it's meant to encourage flexible teamwork with the ability to **fail (and recover) fast** and **celebrate achievements** to promote a productive work culture and bridge the gap between developers and customers.

Agile development teams measure their progress by the amount of working software and have product owners, Devs, Testers, and UX people working together with the same goals in mind.

While Agile focuses on bridging the gap between *developers* and *customers*, DevOps is focused on bridging the gap between **developers** and **operations** teams.

Within software companies, there has historically been friction between the **developers** – who are responsible for generating code and creating new software features – and **operations** teams – who are responsible for *IT infrastructure*, *security*, and *application deployment*.

Devs would regularly generate code that broke the applications they were working on, which Ops would throw back to Devs in frustration and often without sufficient details of the problem. This ultimately results in *slower release times*, team-members being *unable to focus on their primary responsibilities*, and *general frustration* within the organisation.



Adoption of the DevOps methodology requires the *dismantlement of silos*. Devs and Ops are encouraged to break down their silos and start *collaborating*, as well as *share the responsibility* for maintaining the system on which the software is running, and prepare the software to run on the system with a better *feedback and delivery automation*.

It is the responsibility of a DevOps engineer to create infrastructure that encourages this collaboration through *automated, continual testing and integration of new code* so that Devs can focus on developing and Ops can focus on maintaining their IT infrastructure.

Existing software organisations may find the transition to a DevOps methodology to be jarring. As silos become dismantled and teams reorganised, individuals may find their responsibilities – once explicitly defined by their job description – to be less specific due to the emphasis on shared responsibility, resulting in feeling lost in the new paradigm.

However, with the right support during the adoption of DevOps, individuals should adapt reasonably quickly and start reaping the benefits of increased collaboration within their organisation and increased productivity from tedious tasks being fully automated.

There are two key elements to the DevOps approach that ensure that the methodology works: **automation** and **measurement**.

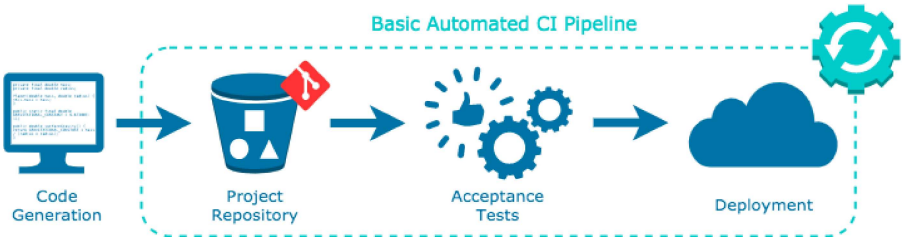
Automation

A DevOps culture encourages **automation** in as many areas of the production pipeline as possible. As a rule of thumb: if a machine *could* do it, a machine *should* be doing it.

Working manually allows the potential for human error and slower development/deployment times, as people are not best suited to performing repetitive tasks as we tend to get bored and distracted.

Machines, on the other hand, never tire nor become bored with their work. Automation provides a level of consistency, predictability, scalability, and quality for this reason.

Eliminating the human error in a production pipeline allows teams to pinpoint constraints in the production workflow itself so that they can be fixed and streamlined to provide maximum output.



Automation allows for faster production times from development to delivery and allows developers to focus the majority of their time generating new features rather than busywork. The result is an efficient pipeline and happier developers, as they get to focus on the creative aspect of their work.

Some automation techniques include:

Continuous Integration (CI)

- When code is committed to a repository, it is automatically built and subjected to acceptance tests.
- Test failures result in the code being prevented from integrating with the repository. Developers are immediately notified of a test failure so they can fix issues as quickly as possible.

Continuous Deployment/Delivery (CD)

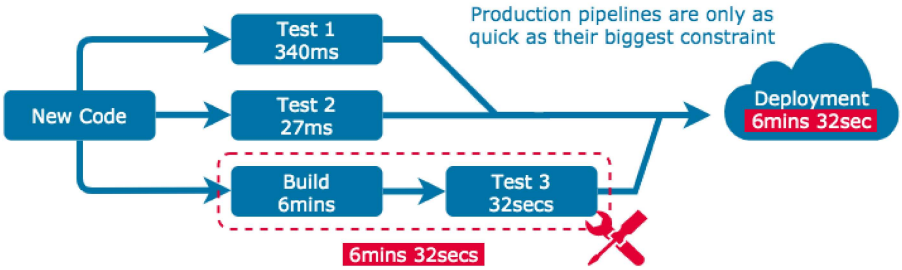
- As new code passes acceptance tests, it is automatically integrated into a deployment environment.
- Being able to choose a version to deploy with one push a button requires a fair amount of automation.

Infrastructure as Code (IaC)

- To deploy and redeploy production environments (i.e. where your application lives at each stage in the production pipeline) quickly and easily, IaC is used to specify the configuration of a computer environment with easy-to-write/read config files.
- Having environment infrastructure declared in code allows for infrastructure to be created or modified using version control.
- IaC allows for simple replication of environments so they stay consistent across the pipeline. This is important for testing environments where you want to replicate the deployment environment as closely as possible.

Measurement

Measurement is central to ensuring that a production pipeline is working efficiently. After all, how can you know something is working better if you can't measure it?



Accurate and precise measurements allow us to *pinpoint constraints* in the pipeline and fix or improve them faster.

Measurements are also important from a cultural standpoint as they can inform teams when they're working more productively and what can be done to improve.

The types of metrics we work to measure include:

Frequency of deployments

- DevOps pipelines encourage *frequent, smaller updates* to software, so charting the frequency of deployments is a good indicator of the effectiveness of a pipeline.
- Upon adoption of the DevOps methodology, deployment frequency should tend upwards until it reaches a natural plateau, though fluctuation is normal.

Mean time to recovery (MTTR)

- This refers to the *average time it takes to solve problems* that impact the end-user. Common problems include outages, security issues, and severe bugs.
- This is a more worthwhile metric than charting the frequency of failures as DevOps is less interested in minimising problems than the speed at which they are solved. Failure is inevitable, what matters is how well we respond to it.

Mean time to discovery (MTTD)

- This refers to *how quickly problems are discovered*. The faster problems are identified, the faster they can be fixed. This metric is measured from the point of integration into production to the point the problem is identified. Naturally, faster MTTDs are more desirable.
- This metric should also indicate whether discovery is made by the customer or the automated systems, with the latter being more desirable.

System availability

- In almost all cases, we want our systems to be *available at all times* to customers. Knowing the availability of our systems allows us to pinpoint which parts of our infrastructure need attention.

Service performance

- Measuring this metric allows us to see whether our services are running within the desired thresholds.
- Metrics may include response times per request, CPU load, or how long it takes for a website to load.

Tutorial

This is no tutorial for this module.

Exercises

There are no exercises for this module.