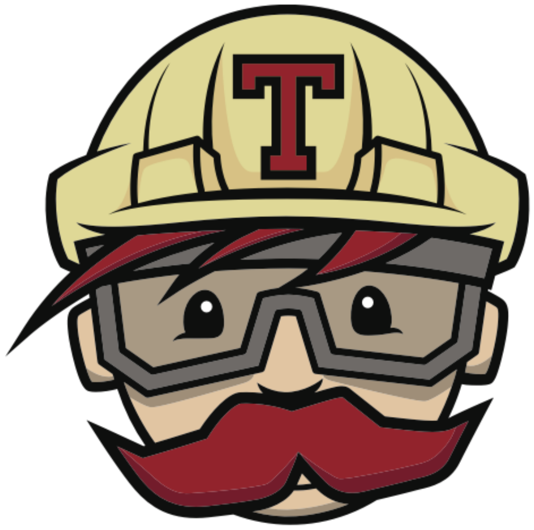
Lecture 3 - Version Control For DevOps



Leveraging VCS'S For Developer Operations

Now that we have some files set for version control tracking and committed for a push to a remote repository, it's time to set up that remote repo. You'll find quite commonly in small to large organizations that it's the Operations team who's responsible for managing repository access and permissions. Let's dig a bit deeper into the day-to-day operations a DevOps engineer would be responsible for when it comes to managing a version control system.

Fundamental VCS Operations

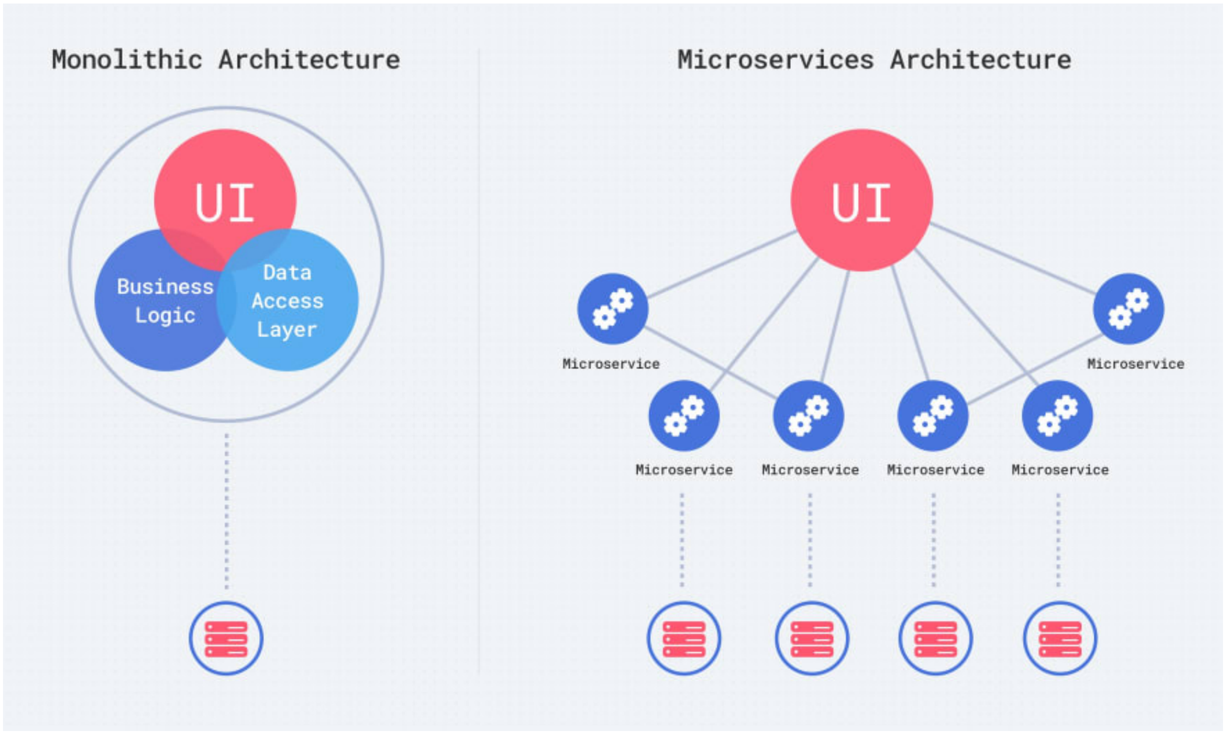
As previously discussed, one advantage of version control is the ability to retain a long-term history of changes made on a project. This would include any change a developer might make to an applications source code like creating, deleting, edits etc.

Another piece of an artifact change should include the author, date and any notes from the change as well. Over time, especially working on projects of significant size and trajectory, these artifacts add up and with that comes the expense of storage. A good DevOps engineer is always thinking about ways to lower expense for a business through the tools they have at their disposal.

A Few Reasons Why VCS is critical for DevOps

- ✓ The avoidance of dependency issues for containerized applications

Microservices have essentially become the default for the development of new applications, and more and more teams are containerizing monolithic applications as well. **Application containerization is an OS-level virtualization method which is commonly used to tightly package, deploy and run distributed applications**. It's a perfect solution for a microservice architecture. It allows an isolated application or services to run on a single host and access the same OS kernel. If you think about **what a microservice is, it's a collection of loosely coupled services that are typically considered to be lightweight solutions to complex problems.**



Looking at a microservice versus a monolithic application, it quickly becomes apparent what this strategy brings to a production app. A microservice is typically designed to fail safely – meaning that if something were to go wrong, it should have a minimized impact that is armed and prepared for failure. Whether this comes in the form of monitoring or the ability to spawn a new container, it is better controlled should something go awry. With a monolithic approach, it's quite likely that a bug or issue can have a much more severe degree of impact when affected.

To provide an example, let's look at a hypothetical situation involving a web application that supports real-time communication with WebSocket's. With a microservice architecture, you'd commonly find some sort of messaging service that is self-contained. If there is an issue with the messaging microservice, the deployment process to provide a fix ideally should be quick and easy and only affect an isolated part of the application. Should an issue arise in a monolithic application, it might not be as apparent where the issue is coming from. Not only that, but deploying a fix would be a much more time-consuming operation. We'd have to re-deploy an application in it's entirety which might accrue downtime. A properly architected microservice should have little to no impact on the overall application.

There are some new problems that a microservice architecture does introduce such as the orchestration needed when deploying the service and the impact it may have on other services currently running. **Luckily, there are orchestration tools to solve these problems, like Google's Kubernetes or Docker Swarm.** These tools are here for a DevOps engineer to leverage in order to better manage infrastructure with a lot of moving parts. Microservice architecture and the DevOps tools available to provide ease when working with these patterns are out of the scope for this course but it's important to understand the concept of containerization.

When it comes to scalability & reliability, Microservices are the obvious winner. For scalability, mono-apps are inferior due the inability to adjust to the demands of a web apps traffic. **A monolithic application uses vertical scaling, which means that it solves traffic demands by adding more power in the form of CPU and RAM.** When working with cloud infrastructure this is an easy enough problem to solve although not ideal due to the associated costs of increasing the power of the server where the application is hosted.

A properly architected microservice application uses horizontal scaling. It can adjust by adding more machines to a pool of resources. When coupled with containerization and an orchestration tool like Kubernetes, it has the concepts of pods. Should one pod go down, a new one is spawned. Should more resources be needed, a new pod is brought into the resource pool.

With respect to reliability, as previously mentioned if one microservice breaks in a properly architected environment, it should in theory only impact one part of the application. If, for example, you're building a banking app and the microservice responsible for money withdrawal is down, it poses less impact than an entire monolithic application being forced to stop.

If your task was to perform a cost analysis on a microservice versus monolithic architecture, it is a bit tricky to say which one would be the obvious winner. Sometimes a monolith architecture is cheaper, in other instances not so much. **Microservices offer the ability to auto-scale** which grants the stakeholder with the benefit of only paying for the needed resources on demand. It really boils down to the size of the application when performing the cost analysis. If the application is small enough a monolith app could cost as little as $5 - $20 a month for hosting. When the application is larger in size, hosting cost may a higher business expense versus a microservice architecture which hosts the application over multiple small, cheap hosts.

When it comes to dependency management, whether microservices or a mon-app, dependencies have always been a big concern. Dependency hell, affectionately called JAR hell in Java, is common in applications across multiple languages. Another advantage made possible by microservices, is something known as dependency isolation. In the field, I've seen over 30 microservices working together to provide not only high-availability for uptime but also the ability to choose the right tool for the job with little impact or concern on some of the other services running. The linear path of building new features and fixing bugs is bound to clash with the parallel development of another feature by another team.

Working in an isolated environment that containerization brings, makes it much easier to manage dependencies in small units rather than having to deal with one giant dependency graph for a single application.

For DevOps, finding the balance between moving quickly and maintaining application reliability is crucial. It's a necessity to keep software in a highly transparent environment which cultivates traceability and the visualization of changes made to the code. Understanding how these changes affect the code base resulting in performance improvements or degradations is managed with much greater ease when a VCS is properly rolled. With each change to the source code has the potential of impacting performance. Should there be a noticeable degradation, it is critical to have the capability of accessing different code iterations committed to the remote repository with ease and little to no impact on application performance.

- ✓ Higher DevOps performance is a byproduct of good version control

In 2014 there was an **annual study known as the "State of DevOps"** which discovered that "version control was consistently one of the highest predictors of performance". The data collected was able to show that top performing engineering teams were able to achieve a higher throughput of deployment frequency at 8x the average and 8000x faster deployment lead times.

Version control certainly has not lost its significance in relation to performance since the study was completed. Last year, it was again tied to success in Continuous Delivery workflows, which naturally resulted in higher IT performance for an organization.

The correlation between version control and high performance is partially tied to the ability to more easily view and understand how changes to one part of the code base results in the causation of problems across an application. To be more accurate, it more so has to do with how **VCS enables coding practices like Continuous Integration which results in Continuous Delivery/Deployment.**

- ✓ **Supports building more reliable applications**

The same study from 2014 found that apps built with comprehensive VCS strategies were also more reliable. The 2014 study which can be found in the supplementary section of this week's lab, also reported properly defined VCS's produced a 50% drop in failure rates when a change was introduced to an applications code base.

In my first role as a JR. Developer, I was told that bugs can typically be found in the most recent change of an applications source. To this day, those words still hold truth and there is data that shows my mentor was indeed correct. It's not hard to draw the connections between a high performing DevOps team and reliable application. The role of a DevOps engineer regardless of the organization is to enable successful advancements. With advancing comes change, therefore significant focus should be put on the mitigation of risk associated with change.

Releasing small changes more frequently (CI) may seem counterintuitive at first but it accelerates workflow while tracking changes. CI provides the possibility for everyone to be looking at the same thing. In the event of a failure, it really makes troubleshooting as a team much easier as everyone should always have a fresh copy of source changes.

Version control is often looked at as such a basic concept that some may implement it naturally with their own personal files by saving multiple versions with different file names. It's importance and the significance of VCS for DevOps can't be overlooked. As the "State of DevOps" shows, VCS is directly correlated to IT performance which has a clear impact on company revenue and success.

Through team collaboration and enabling retroactive forensics of a codebase, VCS helps with the creation of more reliable source code, especially when dealing with 100's of files and potentially 1000's of changes.

Version Control in The Era of DevOps

Some of the most important features a VCS enables:

- Improved collaboration
- Improved CI
- Better support for distributed teams working in a single code base
- Streamlined workflows through the enablement of CI/CD

To this day, there is currently not a better way in the industry to automate the integration of new features/requests for a product than VCS transactions. Previously, users had to manually communicate with a member of the operations team to have something deployed to a QA server. VCS coupled with a well-defined pipeline allows for the automation of builds and deployments.

VCS's allow for DevOps to support teams of any size but what makes the process so effective? The "State of DevOps" study has shown year-after-year that a properly defined VCS allows for DevOps settings to be made in an environment, which is much easier than managing configuration settings in the code. If something is misconfigured, it can result in a service not functioning correctly. When migrations fail, most likely it's due to environmental misconfigurations, not problems in the code.

With that being said, it brings the countermeasure of **all production configurations being checked into version control**. This ultimately **serves as a single source of truth that can be deployed for all environments** such as development, staging or a QA server.

Stability is another benefit that version control brings to an organization. For Continuous Delivery (CD) to exist, developers create production-like environments on-demand via an automated build process which is kicked off typically by a code merge into an applications master branch. Developer and Operation teams can count on version control to ensure changes to the code and environment are continuously being integrated and deployed. Version control for DevOps is ultimately translatable to an increase in IT and organizational performance.

With the incorporated techniques of automating a build process through VCS's allows for something known Infrastructure as Code (IaC). IaC enables automation, testability, quality assurance and a higher rate of predictability when making deployments. A version control process through the lens of a DevOps engineer is heavily focused on automating out mundane time-consuming practices, such as manual deployments.

DevOps engineers build something known as triggers into the code so that changes committed to a repository kick off things like automated tests, code quality analysis's and deployments to development or staging environments.

Since Git is the most commonly used version control system, it's what we'll be using for our lab work. As previously mentioned though, most of the concepts are transferrable from VCS-to-VCS. Git allows for changes to code to be committed on your local machine and then pushed to a repository server. For our repository server, we'll be using GitHub as it offers the features we need for free and ideally, as a developer you should have some sort of online GitHub presence. It can serve as a portfolio of sorts when engaging with prospective employers.

In our lab work this week, we'll be creating a repository on GitHub and linking our public SSH key to it for the ability to push code changes remotely. We'll also take a small tour of some of the other areas we'll be working with when it comes to automating our very own build process.