Module 3

Effect of DevOps on Software Architecture and Code Control

Software Architecture

- How DevOps affects the architecture of our applications.
- Software architecture:
  - Functional requirement
    - Particular behaviours
  - Non-functional requirements
    - Different characteristics of the software rather than the requirements on particular behaviours.

- Example:
  - Functional requirement our system should be able to deal with credit card transactions.
  - Non-functional requirement could be that the system should be able to manage several such credit cards transactions per second.

Software Architecture

Here are two of the non-functional requirements that DevOps and Continuous Delivery place on software architecture:

- We need to be able to deploy small changes often
- We need to be able to have great confidence in the quality of our changes

- The normal case should be that we are able to deploy small changes all the way from developers' machines to production in a small amount of time.
- Rolling back a change because of unexpected problems caused by it should be a rare occurrence.

The monolithic scenario

One way to understand the issues that a problematic architecture can cause for Continuous Delivery is to consider a counterexample for a while.

Let's suppose we have a large web application with many different functions. We also have a static website inside the application. The entire web application is deployed as a single Java EE application archive. So, when we need to fix a spelling mistake in the static website, we need to rebuild the entire web application archive and deploy it again.

The Monolithic Scenario

As DevOps engineers, this could be an actual situation that we might be asked to solve.

Let's break down the consequences of this tangling of concerns. What happens when we want to correct a spelling mistake? Let's take a look:

1. We know which spelling mistake we want to correct, but in which code base do we need to do it? Since we have a monolith, we need to make a branch in our code base's revision control system. This new branch corresponds to the code that we have running in production.
2. Make the branch and correct the spelling mistake.
3. Build a new artifact with the correction. Give it a new version.
4. Deploy the new artifact to production.

The Monolithic Scenario

Okay, this doesn't seem altogether too bad at first glance. But consider the following too:

- We made a change in the monolith that our entire business critical system comprises. If something breaks while we are deploying the new version, we lose revenue by the minute. Are we really sure that the change affects nothing else?
- It turns out that we didn't really just limit the change to correcting a spelling mistake. We also changed a number of version strings when we produced the new artifact. But changing a version string should be safe too, right? Are we sure?

The point here is that we have already spent considerable mental energy in making sure that the change is really safe. The system is so complex that it becomes difficult to think about the effects of changes, even though they might be trivial.

Now, a change is usually more complex than a simple spelling correction. Thus, we need to exercise all aspects of the deployment chain, including manual verification, for all changes to a monolith.

Architecture rules of thumb

There are a number of architecture rules that might help us understand how to deal with the previous undesirable situation.

The separation of concerns

The fundamental principle is simply that we should consider different aspects of a system separately.

The principle of cohesion

In computer science, cohesion refers to the degree to which the elements of a software module belong together. Cohesion can be used as a measure of how strongly related the functions in a module are. It is desirable to have strong cohesion in a module. We can see that strong cohesion is another aspect of the principle of the separation of concerns.

Architecture rules of thumb

Coupling

Coupling refers to the degree of dependency between two modules. We always want low coupling between modules.

Again, we can see coupling as another aspect of the principle of the separation of concerns. **Systems with high cohesion and low coupling would automatically have separation of concerns, and vice versa.**

Architecture rules of thumb

Back to the monolithic scenario

In the previous scenario with the spelling correction, it is clear that we failed with respect to the separation of concerns. We didn't have any modularization at all, at least from a deployment point of view. The system appears to have the undesirable features of low cohesion and high coupling.

- If we had a set of separate deployment modules instead, our spelling correction would most likely have affected only a single module. It would have been more apparent that deploying the change was safe.
- How this should be accomplished in practice varies, of course. In this particular example, the spelling corrections probably belong to a frontend web component. At the very least, this frontend component can be deployed separately from the backend components and have their own life cycle.

Architecture rules of thumb

A practical example

we work for an organization called Matangle. This organization is a software as a service (SaaS) provider that sells access to educational games for schoolchildren. As with any such provider, we will, with all likelihood, have a database containing customer information. It is this database that we will start out with. The organization's other systems will be fleshed out as we go along, but this initial system serves well for our purposes.

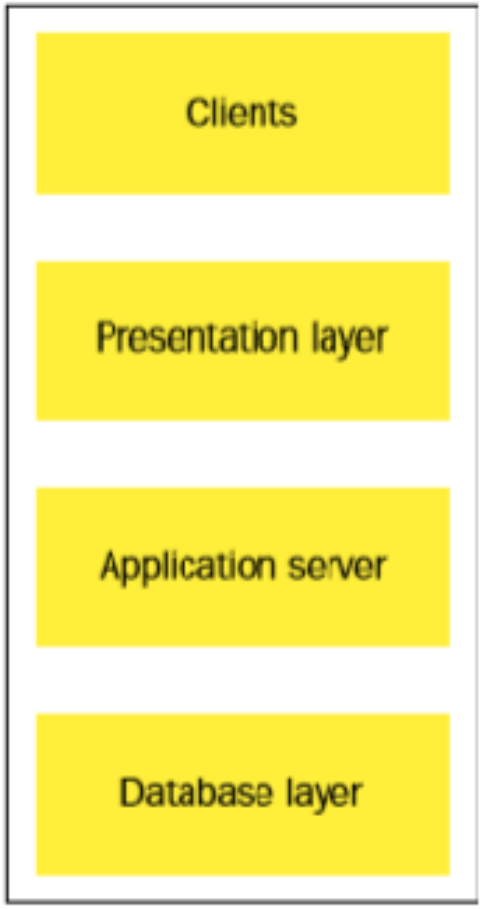
References

- What is SEPARATION OF CONCERNS? What does SEPARATION OF CONCERNS mean?
- https://www.youtube.com/watch?v=DZaO3c30MEU
- Software Engineering Separation Of Concerns For Beginners
- https://www.youtube.com/watch?v=Fv9_udQJTzM

Three-tier Systems

The Matangle customer database is a very typical three-tier, CRUD (create, read, update, and delete) type of system. This software architecture pattern has been in use for decades and continues to be popular.

Three-tier systems



In this figure, we can see the separation of concerns idea in action:

https://www.ibm.com/cloud/blog/four-architecture-choices-for-application-development

Three-tier systems

The presentation tier

The presentation tier will be a web frontend implemented using the React web framework. It will be deployed as a set of JavaScript and static HTML files.

The logic tier

The logic tier is a backend implemented using the Clojure language on the Java platform. The Java platform is very common in large organizations, while smaller organizations might prefer other platforms based on Ruby or Python.

The data tier

In our case, the database is implemented with the PostgreSQL database system. PostgreSQL is a relational database management system. PostgreSQL is a robust system.

Three-tier systems

From a DevOps point of view, the three-tier pattern looks compelling, at least superficially. It should be possible to deploy changes to each of the three layers separately, which would make it simple to propagate small changes through the servers.

- In practice, though, the data tier and logic tier are often tightly coupled.
- The same might also be true for the presentation tier and logic tier. To avoid this, care must be taken to keep the interfaces between the tiers lean.

If we don't take care while designing our system, we can still wind up with an undesirable monolithic system.

Three-tier Systems

Handling database migrations

Handling changes in a relational database requires special consideration. A relational database stores both data and the structure of the data.

- Upgrading a database schema offers other challenges then the ones present when upgrading program binaries.
- Usually, when we upgrade an application binary, we stop the application, upgrade it, and then start it again. We don't really bother about the application state. That is handled outside of the application.
- When upgrading databases, we do need to consider state, because a database contains comparatively little logic and structure, but contains much state. In order to describe a database structure change, we issue a command to change the structure.

Three-tier systems

Handling database migrations

The database structures before and after a change is applied should be seen as different versions of the database. How do we keep track of database versions?

- Liquibase is a database migration system that, at its core, uses a tried and tested method.
- When Liquibase is being asked to upgrade a database, it looks at the metadata table and determines which changesets to run in order to make the database up-to-date with the latest version.
- The Clojure ecosystem also has at least one similar database migration system of its own, called Migratus.
- Flyway is another example for the Java platform.

Three-tier Systems

A binary file is **a file whose content must be interpreted by a program or a hardware processor that understands in advance exactly how it is formatted**. ... Programmers often talk about an executable program as a "binary" and will ask another programmer to "send me the binaries." (A synonym for this usage is object code .)

Three-tier Systems

- Generally, database migration systems employ some variant of the following method:
  - Add a table to the database where a database version is stored.
  - Keep track of database change commands and bunch them together in versioned changesets. In the case of Liquibase, these changes are stored in XML files. Flyway employs a slightly different method where the changesets are handled as separate SQL files or occasionally as separate Java classes for more complex transitions.
  - When Liquibase is being asked to upgrade a database, it looks at the metadata table and determines which changesets to run in order to make the database up-to-date with the latest version.

Three-tier Systems

Rolling upgrades

Another thing to consider when doing database migrations is what can be referred to as rolling upgrades. These kinds of deployments are common when you don't want your end user to experience any downtime, or at least very little downtime.

- Here is an example of a rolling upgrade for our organization's customer database. When we start, we have a running system with one database and two servers. We have a load balancer in front of the two servers. We are going to roll out a change to the database schema, which also affects the servers.

Three-tier Systems

Rolling upgrades

We are going to split the customer name field in the database into two separate fields, first name and surname. This is an incompatible change. How do we minimize downtime? Let's look at the solution:

1. We start out by doing a database migration that creates the two new name fields and then fills these new fields by taking the old name field and splitting the field into two halves by finding a space character in the name. This was the initial chosen encoding for names, which wasn't stellar. This is why we want to change it. This change is so far backward compatible, because we didn't remove the name field; we just created two new fields that are, so far, unused.
2. Next, we change the load balancer configuration so that the second of our two servers is no longer accessible from the outside world. The first server chugs along happily, because the old name field is still accessible to the old server code.

Three-tier Systems

Rolling upgrades

3. Now we are free to upgrade server two, since nobody uses it. After the upgrade, we start it, and it is also happy because it uses the two new database fields.

4. At this point, we can again switch the load balancer configuration such that server one is not available, and server two is brought online instead. We do the same kind of upgrade on server one while it is offline. We start it and now make both servers accessible again by reinstating our original load balancer configuration. Now, the change is deployed almost completely. The only thing remaining is removing the old name field, since no server code uses it anymore.

Microservices

Microservices is a recent term used to describe systems where the logic tier of the three-tier pattern is composed of several smaller services that communicate with language-agnostic protocols. Typically, the language-agnostic protocol is HTTP based, commonly JSON REST, but this is not mandatory. There are several possibilities for the protocol layer.

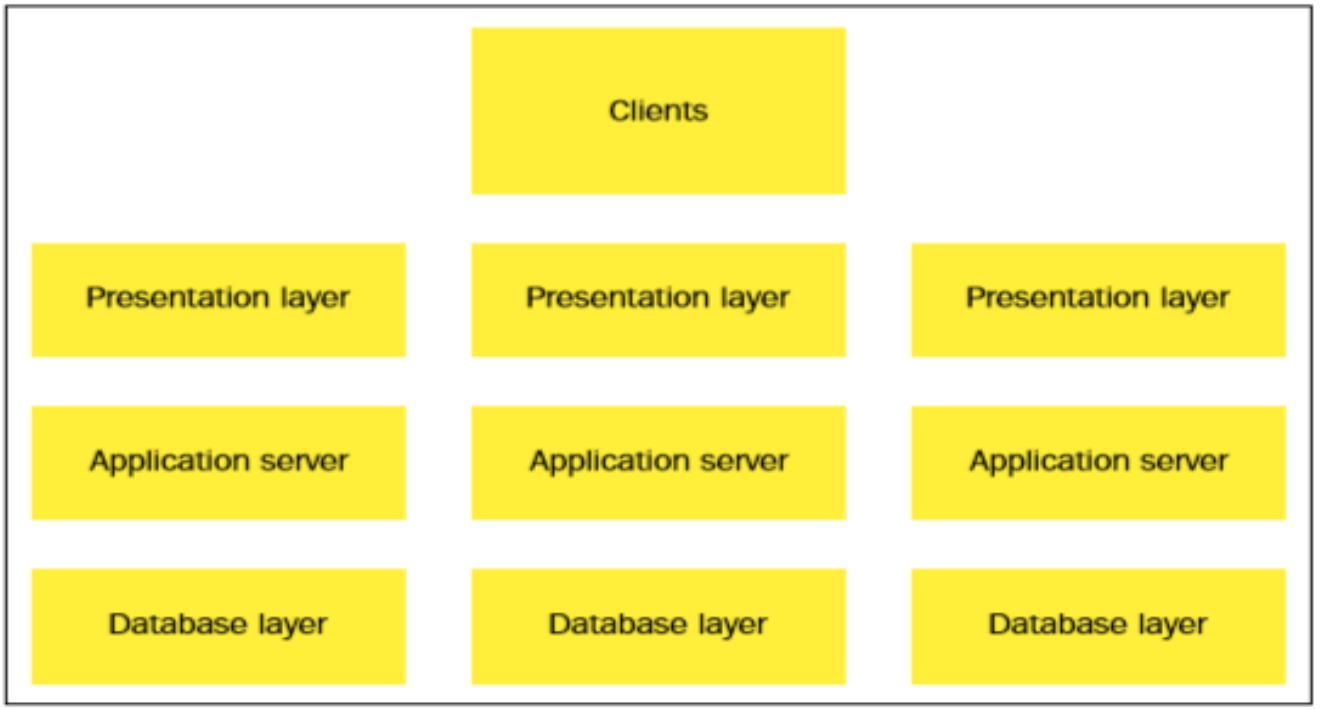
This architectural design pattern lends itself well to a Continuous Delivery approach since, as we have seen, it's easier to deploy a set of smaller standalone services than a monolith.

Microservices

Microservices is a recent term used to describe systems where the logic tier of the three-tier pattern is composed of several smaller services that communicate with language-agnostic protocols. Typically, the language-agnostic protocol is HTTP based, commonly JSON REST, but this is not mandatory. There are several possibilities for the protocol layer.

This architectural design pattern lends itself well to a Continuous Delivery approach since, as we have seen, it's easier to deploy a set of smaller standalone services than a monolith.

An illustration of what a microservices deployment might look like:



What are Microservices | Microservices Architecture Training | Microservices Tutorial | Edureka https://www.youtube.com/watch?v=gfWr2_H39N0

Microservices and the data tier

One way of viewing microservices is that each microservice is potentially a separate three-tier system. We don't normally implement each tier for each microservice though.

- With this in mind, we see that each microservice can implement its own data layer. The benefit would be a potential increase of separation between services.
- It is more common in my experience, though, to put all of the organization's data into a single database or at least a single database type. This is more common, but not necessarily better.
- There are pros and cons to both scenarios. It is easier to deploy changes when the systems are clearly separate from each other. On the other hand, data modeling is easier when everything is stored in the same database.

Interlude – Conway's Law

In 1968, Melvin Conway introduced the idea that the structure of an organization that designs software winds up copied in the organization of the software. This is called Conway's Law. The three-tier pattern, for instance, mirrors the way many organizations' IT departments are structured:

- The database administrator team, or DBA team for short
- The backend developer team
- The frontend developer team
- The operations team

Well, that makes four teams, but we can see the resemblance clearly between the architectural pattern and the organization

Interlude – Conway's Law

The primary goal of DevOps is to bring different roles together, preferably in crossfunctional teams. If Conway's Law holds true, the organization of such teams would be mirrored in their designs. The microservice pattern happens to mirror a cross-functional team quite closely.

DevOps, architecture, and resilience

We have seen that the microservice architecture has many desirable properties from a DevOps point of view.

An important goal of DevOps is to place new features in the hands of our user faster. This is a consequence of the greater amount of modularization that microservices provide. Microservices do offer challenges of their own:

- We want to be able to deploy new code quickly, but we also want our software to be reliable.
- Microservices have more integration points between systems and suffer from a higher possibility of failure than monolithic systems.

DevOps, architecture, and resilience

- Automated testing is very important with DevOps so that the changes we deploy are of good quality and can be relied upon. This is, however, not a solution to the problem of services that suddenly stop working for other reasons. Since we have more running services with the microservice pattern, it is statistically more likely for a service to fail.
- We can partially mitigate this problem by making an effort to **monitor the services and take appropriate action when something fails**. This should preferably be automated.

DevOps, architecture, and resilience

In our customer database example, we can employ the following strategy:

- We use two application servers that both run our application
- The application offers a special monitoring interface via JsonRest
- A monitoring daemon periodically polls this monitoring interface
- If a server stops working, the load balancer is reconfigured such that the offending server is taken out of the server pool

This is obviously a simple example, but it serves to illustrate the challenges we face when designing resilient systems that comprise many moving parts and how they might affect architectural decisions.

DevOps, architecture, and resilience

Why do we offer our own application-specific monitoring interface though?

- Since the purpose of monitoring is to give us a thorough understanding of the current health of the system, we normally monitor many aspects of the application stack.
- Server CPU isn't overloaded, that there is sufficient disk and memory space available, and that the base application server is running. This might not be sufficient to determine whether a service is running properly, though.
- For example service might for some reason have a broken database access configuration. A service-specific health probing interface would attempt to contact the database and return the status of the connection in the return structure.
- Best if your organization can agree on a common format for the probing return structure. The structure will also depend on the type of monitoring software used.

Module 3.3 and 3.4

SCM

The need for source code control

Terence McKenna, an American author, once said that everything is code. While one might not agree with McKenna about whether everything in the universe can be represented as code, for DevOps purposes, indeed nearly everything can be expressed in codified form, including the following:

- The applications that we build
- The infrastructure that hosts our applications
- The documentation that documents our products

Even the hardware that runs our applications can be expressed in the form of software.

Given the importance of code, it is only natural that the location that we place code, the source code repository, is central to our organization. Nearly everything we produce travels through the code repository at some point in its life cycle.

Roles and code

From a DevOps point of view source code management tool- Natural meeting point.

Many different roles have a use for source code management.

Developers live and breathe source code management. It's their bread and butter.

Operations personnel also favor managing the descriptions of infrastructure in the form of code, scripts, and other artifacts. Such infrastructural descriptors include network topology, versions of software that should be installed on particular servers, and so on.

Quality assurance personnel can store their automated tests in codified form in the source code repository. This is true for software testing frameworks such as Selenium and Junit etc.

There is a problem with the documentation of the manual steps needed to perform various tasks. While many organizations employ a wiki solution such as the wiki engine powering Wikipedia, there is still a lot of documentation floating around in the Word format on shared drives and in e-mails.

This makes documentation hard to find and use for some roles and easy for others.

From a DevOps viewpoint, this is regrettable, and an effort should be made so that all roles can have good and useful access to the documentation in the organization.

It is possible to store all documentation in the wiki format in the central source code repository, depending on the wiki engine used.

Which source code management system?

There are many source code management (SCM) systems.

Currently, there is a dominant system, however, and that system is Git.

The primary benefit of Git versus older systems is that it is a distributed version control system (DVCS). There are many other distributed version control systems, but Git is the most pervasive one.

Distributed version control systems have several advantages, including, but not limited to, the following:

- It is possible to use a DVCS efficiently even when not connected to a network. You can take your work with you when traveling on a train or an intercontinental flight.
- Since you don't need to connect to a server for every operation, a DVCS can be faster than the alternatives in most scenarios.
- You can work privately until you feel your work is ready enough to be shared.
- It is possible to work with several remote logins simultaneously, which avoids a single point of failure.

Other distributed version control systems apart from Git include the following:

- Bazaar
- Mercurial

Source code management system migrations

Source code management systems transitions from one type of system to another.

Sometimes, much time is spent on keeping all the history intact while performing a migration. For some systems, this effort is well spent, such as for venerable free or open source projects.

For many organizations, keeping the history is not worth the significant expenditure in time and effort. If an older version is needed at some point, the old source code management system can be kept online and referenced. This includes migrations from Visual SourceSafe and ClearCase.

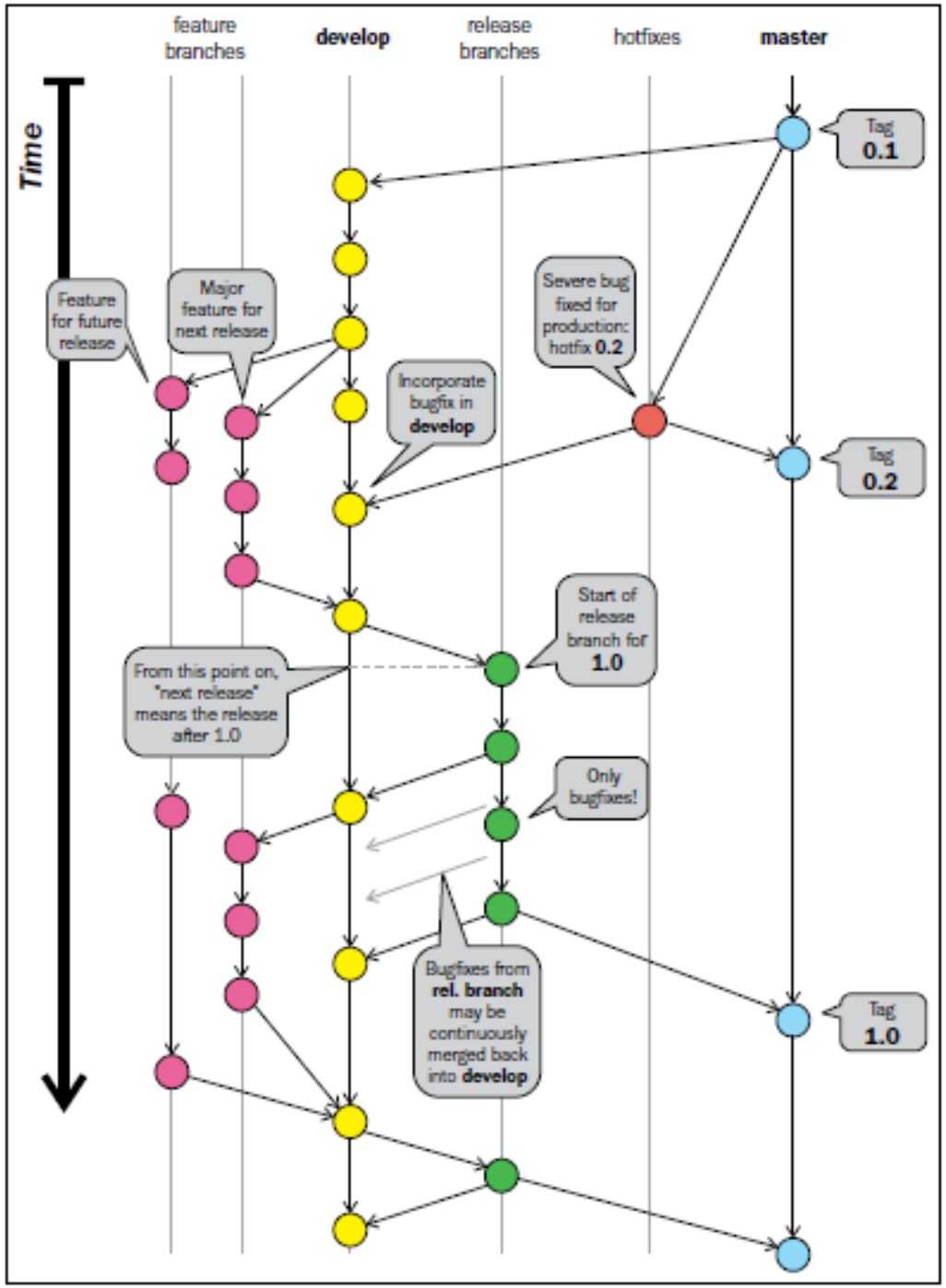
Some migrations are trivial though, such as moving from Subversion to Git. In these cases, historic accuracy need not be sacrificed.

The Top Five Main Git Branching Strategies

Strategy	Overview
GitFlow	Created by Vincent Driessen, GitFlow works with two main branches – master and develop – over the lifetime of the project. It also uses three supporting branches: feature-*, hotfix-*, and release-*. It's the most complex model.
GitHub Flow	A simpler strategy promoted by Scott Chacon of GitHub mandates keeping a continuously deployable main branch. A feature branch is created to work on any feature or bug fix. Each feature branch must be finished and fully tested before being merged with the main branch.
Trunk-Based Development	Very similar to GitHub Flow except that Trunk-Based Development suggests deployment after production code is merged to the main branch to minimize chances for regression.
GitLab Flow	Created by GitLab, this strategy is like an extension of GitHub Flow with master and feature branches. However, it adds environment and release branches to better support SaaS and mobile projects.

- https://gitential.com/git-branching-strategies-for-your-team-how-to-choose-the-best/

- GitFlow


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

How Does Branching Strategy Relate to DevOps?

- DevOps focuses on efficiency – making things easier and faster without compromising quality. Where branching strategy is concerned, DevOps wants to know how pull and merge requests are handled, the amount of overhead involved in managing the branches.
- "It's very important to find a branching strategy that fits the way the team works. If branching is not done right, it introduces big burdens and slows the team down. It also can impact the quality of your product… The best guidance is to start small and add complexity only if you need it."
- https://writeabout.net/2018/05/04/git-branching-guidance-for-devops-teams/
- Learn Git from Scratch - How to Create Branches and push to Github https://www.youtube.com/watch?v=pDmYNK68IEc