UNIT-3

Introduction to project management: The need for source code control, The history of source code management, Roles and code, source code management system and migrations, Shared authentication, Hosted Git servers, Different Git server implementations, Docker intermission, Gerrit,The pull request model, GitLab.

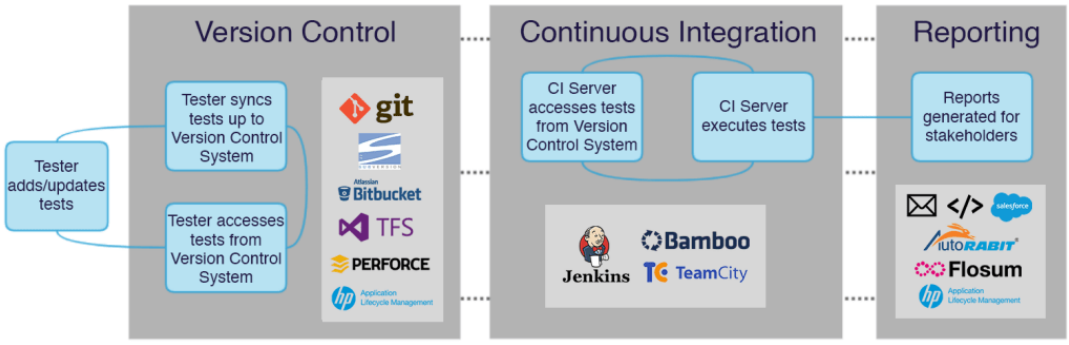
Introduction to project management:

DevOps project management is a union between the two disciplines where project management is tailored to and supports the DevOps model. In a DevOps project management approach, project managers serve as coordinators between multiple contributors and as trackers of timelines and dependencies.

However, project managers also need to be closely aligned with the DevOps team and bring a deep understanding of the development process and skills needed to create the end product. There are several best practices that project managers should follow to achieve this integration with the DevOps pipeline.

The need for source code control:

In today's fast-paced software development environment, DevOps has emerged as a leading approach that enables faster and more reliable software delivery. DevOps brings together development and operations teams, automating the entire software development lifecycle, from code development to deployment and operations. However, for DevOps to be successful, it requires a robust source code control (SCC) mechanism.



Source code control, also known as version control, is the process of managing and tracking changes to the source code. SCC provides a centralized repository for storing and managing the source code, allowing developers to track changes, collaborate, and maintain version control. SCC is necessary in DevOps for several reasons:

Collaborative Development: In DevOps, developers often work on the same codebase simultaneously. SCC allows multiple developers to work on the same codebase, manage conflicts, and merge changes without disrupting each other's work. SCC provides a way to coordinate the efforts of multiple developers working on the same code, ensuring that everyone is working with the most up-to-date version of the code.

Version Control: SCC enables developers to maintain different versions of the same codebase. This is important when testing new features or bug fixes without affecting the existing codebase. Version control also provides an audit trail of changes, making it easier to identify when and

where issues occur. With SCC, developers can easily revert to a previous version of the codebase if a problem is encountered.

Continuous Integration and Delivery (CI/CD): SCC integrates with CI/CD tools to automate the build, testing, and deployment process. This allows developers to make changes to the codebase without disrupting the production environment. SCC provides the infrastructure necessary to ensure that code is continuously integrated, tested, and deployed, ensuring faster time-to-market and a better quality product.

Backup and Disaster Recovery: SCC provides a backup of the entire codebase, allowing developers to recover quickly in the event of a disaster. This ensures that critical code is never lost, and teams can continue working without significant downtime. SCC also enables developers to restore the codebase to a previous version in the event of a problem, ensuring that issues can be resolved quickly.

The history of source code management:

Source code management (SCM) is the process of tracking changes to source code files over time, enabling multiple developers to work on the same codebase without interfering with each other's work. SCM allows developers to collaborate more effectively, and it is essential for managing large software projects.

The history of SCM can be traced back to the early days of software development, when developers used manual version control systems such as making backups of code files or maintaining different versions of code in separate folders. In the 1970s, the concept of version control was formalized with the introduction of the Revision Control System (RCS), which allowed developers to manage changes to their codebase and revert to earlier versions of the code if necessary.

In the 1980s, the Concurrent Versioning System (CVS) was developed, which introduced the concept of branches, allowing multiple developers to work on different versions of the same codebase in parallel. CVS was widely adopted and remained a popular choice for version control until the mid-2000s.

In the late 1990s and early 2000s, the open-source community developed a number of new version control systems, including Subversion (SVN) and Git. SVN improved upon CVS with features such as atomic commits, which ensured that either all changes were made or none were made, and versioned directories, which allowed developers to easily move files around within the codebase.

Git, which was created in 2005 by Linus Torvalds, the creator of the Linux operating system, revolutionized SCM with its distributed model. In a distributed SCM, each developer has a complete copy of the codebase, enabling them to work offline and commit changes to their local repository before pushing them to a central repository. This makes it much easier to manage complex software projects with many contributors, and Git quickly became the most popular version control system in use today.

In terms of roles, SCM is typically the responsibility of a dedicated team or individual within a software development organization. This person, known as the SCM manager or version control administrator, is responsible for setting up and maintaining the version control system, defining

the workflows and processes for managing code changes, and enforcing policies such as code review and quality assurance.

Developers, on the other hand, are responsible for working within the version control system, committing changes to the repository, resolving conflicts, and adhering to the workflows and processes established by the SCM manager. Code is the central component of SCM, and the version control system is responsible for managing changes to code files and ensuring that the correct version of the code is deployed to production environments.

Roles and code

Version control is a revision control system designed to track changes to code over time. The term is often used interchangeably with source code management.

Version control is managed with version control software that tracks every code change in a special type of database. When developers save changes made to a file, the system retains — rather than overwrites — all the changes from every previous version of that file.

Maintaining a versioned record of all code changes in this way offers several benefits:

It protects the source code: Version control helps software teams manage source code and prevents it from being accidentally lost, deleted or otherwise compromised by human error.

Developers can compare or restore earlier versions of code: With a complete history of a project's code, developers can compare the current codebase to previous versions to help fix mistakes. They can also roll back to an earlier version if their changes introduce a bug, for example, or if multiple changes conflict.

It allows developers to work independently: Developers can work individually on their own code changes in their own workspace, then merge all their changes together and be alerted to any conflicts that need to be addressed.

As part of the branch strategy, there are two types of version control systems: centralized and distributed.

Centralized version control systems maintain one copy of a project on a centralized server to which all code changes are committed. Developers can check out the specific files they need to work on, but never have a full copy of the project locally.

Centralized Version Control is a version control system using server/client model and server contains all the history of source code.

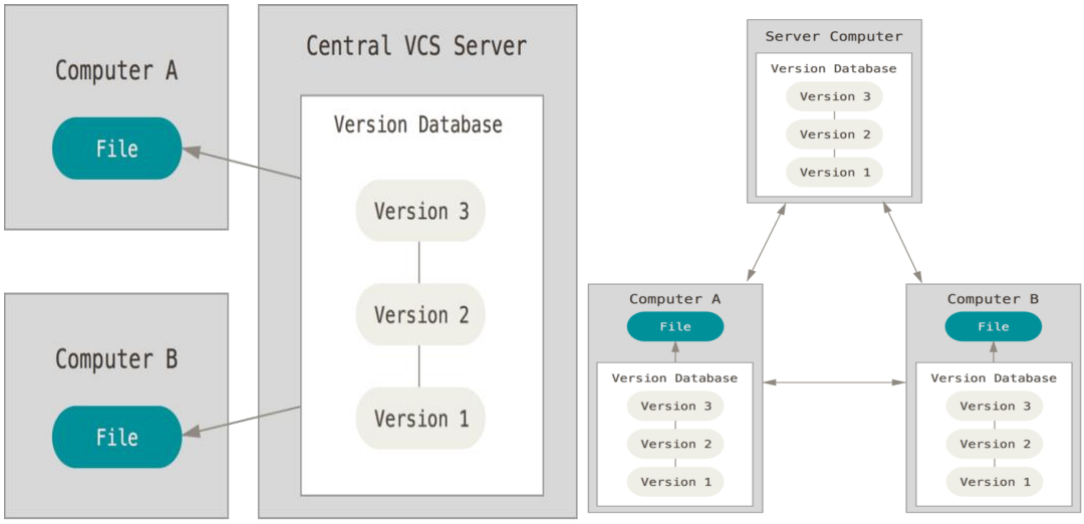
Distributed version control systems allow developers to clone the repository to their own local server or machine so they have the complete history of the project. They then clone the code they need to work on from this local copy of the master repository. When they're done with their changes, they commit them to their local source code repository, then "push" this local repository code to the master repository.

Distributed Version Control is a version control where each client can have same copy of source code as server has and both server and client maintain history of source code.

Following are the important difference between Centralized Version Control and Distributed Version Control.

Sr. No.	Key	Centralized Version Control	Distributed Version Control
1	Working	In CVS, a client need to get local copy of source from server, do the changes and commit those changes to centeral source on server.	In DVS, each client can have a local branch as well and have a complete history on it. Client need to push the changes to branch which will then be pushed to server repository.
2	Learning Curve	CVS systems are easy to learn and set up.	DVS systems are difficult for beginners. Multiple commands needs to be remembered.
3	Branches	Working on branches in difficult in CVS. Developer often faces merge conflicts.	Working on branches in easier in DVS. Developer faces lesser conflicts.
4	Offline Access	CVS system do not provide offline access.	DVD systems are workable offline as a client copies the entire repository on their local machine.
5	Speed	CVS is slower as every command need to communicate with server.	DVS is faster as mostly user deals with local copy without hitting server everytime.

6	Backup	If CVS Server is down, developers cannot work.	If DVS server is down, developer can work using their local copies.



CVCS **DVCS**

source code management system migrations

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 Microsoft **Visual SourceSafe** (**VSS**) and ClearCase.

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

Choosing a branching strategy

When working with code that deploys to servers, it is important to agree on a branching strategy across the organization.

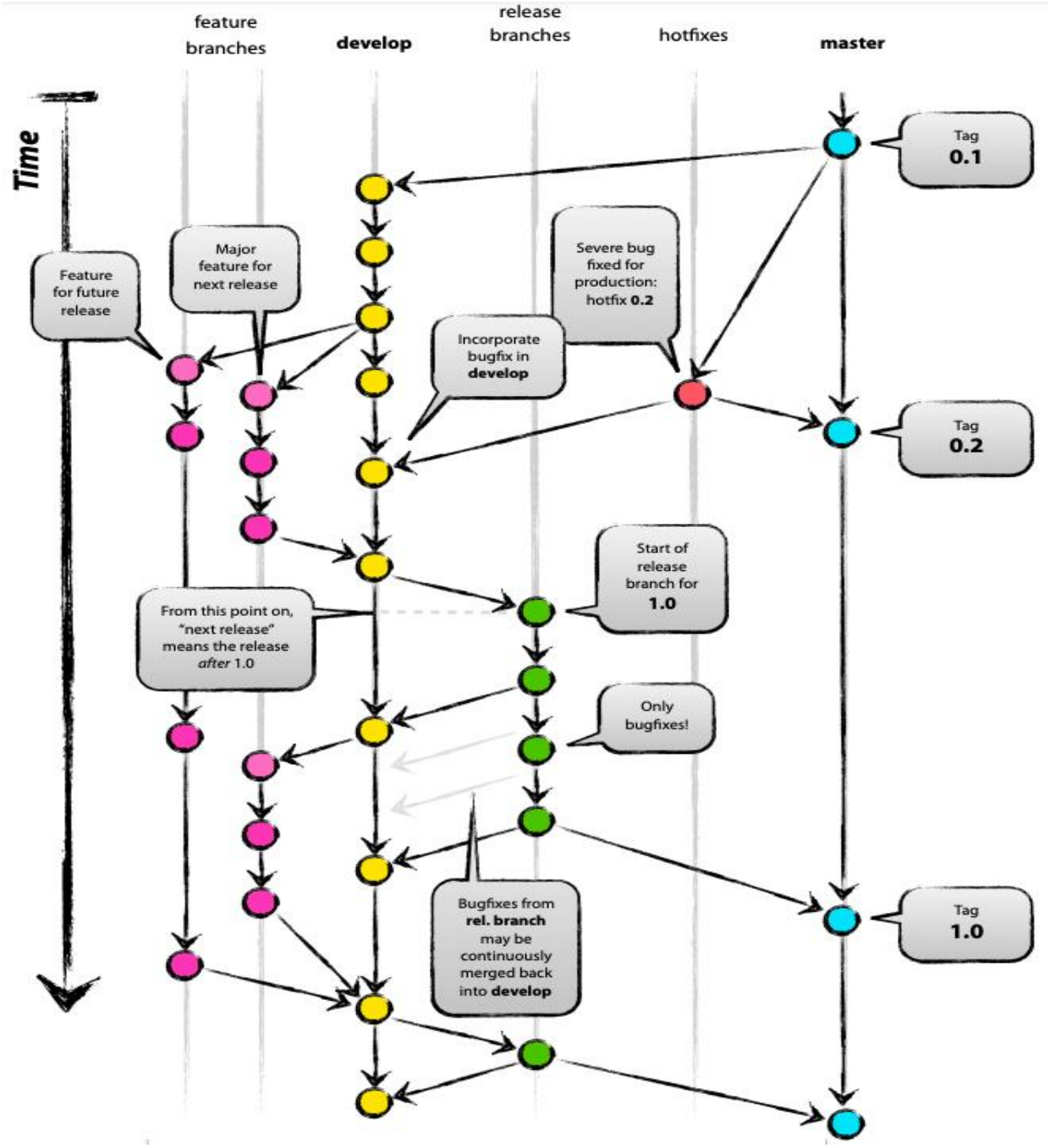
A branching strategy is a convention, or a set of rules, that describes when branches are created, how they are to be named, what use branches should have, and so on.

Branching strategies are important when working together with other people and are, to a degree, less important when you are working on your own, but they still have a purpose.

Most source code management systems do not prescribe a particular branching strategy and neither does Git. The SCM simply gives you the base mechanics to perform branching.

With Git and other distributed VCS, it is usually pretty cheap to work locally with feature branches. A feature, or topic, branch is a branching strategy that is used to keep track of ongoing development regarding a particular feature, bug, and so on. This way, all changes in the code regarding the feature can be handled together.

There are many well-known branching strategies. Vincent Driessen formalized a branching strategy called **Git flow**, which has many good features. For some, Git flow is too complex, and, in those cases, it can be scaled down. There are many such scaled-down models available. This is what Git flow looks like:



Git flow looks complex, so let's have a brief look at what the branches are for:

- The **master** branch only contains finished work. All commits are tagged, since they represent releases. All releases happen from the **master** branch.
- The **develop** branch is where work on the next release happens. When work is finished here, the **develop** branch is merged with the **master** branch.

- We use separate **feature branches** for all new features. **Feature branches** are merged to the **develop** branch.
- When a devastating bug is revealed in production, a **hotfix** branch is made where a bug fix is created. The **hotfix** branch is then merged to the **master** branch, and a new release for production is made.

Git flow is a centralized pattern and, as such, it's reminiscent of the workflows used with Subversion, CVS, and so on. The main difference is that using Git has some technical and efficiency-related advantages.

Another strategy, called the **forking pattern**, where every developer has a central repository, is rarely used in practice within organizations, except when, for instance, external parties such as subcontractors are being employed.

Choosing a client

One of the nice aspects of Git is that it doesn't mandate the use of a particular client. There are several to choose from, and they are all compatible with each other. Most of the clients use one of several core Git implementations, which is good for stability and quality.

Most current development environments have good support for using Git.

In this sense, choosing a client is not something we actually need to do. Most clients work well enough, and the choice can be left to the preferences of those using the client. Many developers use the client integrated in their development environments, or the command-line Git client. When working with operations tasks, the command-line Git client is often preferred because it is convenient to use when working remotely through an SSH shell.

The one exception where we actually have to make a choice is when we assist people in the organization who are new to source code management in general, and Git in particular.

In these cases, it is useful to maintain instructions on how to install and use a simple Git client and configure it for our organization's servers.

A simple instruction manual in the organization's wiki normally solves this issue nicely.

Setting up a basic Git server

It's pretty easy to set up a basic Git server. While this is rarely enough in a large organization, it is a good exercise before moving on to more advanced solutions.

Let's first of all specify an overview of the steps we will take and the bits and pieces we will need to complete them:

1. **A client machine, with two user accounts, which has the Git and SSH packages installed:** The SSH protocol features prominently as a base transport for other protocols, which also is the case for Git. You need your SSH public keys handy. If you don't have the keys for some reason, use ssh-keygen to create them.

We need two users, because we will simulate two users talking to a central server. Make keys for both test users.

2. **A server where the SSH daemon is running:** This can be the same machine as the one where you simulate the two different client users, or it can be another machine.

3. **A Git server user:** We need a separate Git user who will handle the Git server functionality. Now, you need to append the public keys for your two users to the authorized_keys file, which is located in the respective user's ssh directory. Copy the keys to this account.

Are you starting to get the feeling that all of this is a lot of hassle? This is why we will have a look at solutions that simplify this process later.

4. **A bare Git repository:** Bare Git repositories are a Git peculiarity. They are just Git repositories, except there is no working copy, so they save a bit of space. Here's how to create one:

```
cd /opt/git
mkdir project.git
cd project.git
git init --bare
```

5. Now, try cloning, making changes, and pushing to the server.

Let's review the solution:

- **This solution doesn't really scale very well**: If you have just a couple of people requiring access, the overhead of creating new projects and adding new keys isn't too bad. It's just a onetime cost. If you have a lot of people in your organization who need access, this solution requires too much work.
- **Solving security issues involves even more hassle**: While I am of the perhaps somewhat controversial opinion that too much work is spent within organizations limiting employee access to systems, there is no denying that you need this ability in your setup.

In this solution, you would need to set up separate Git server accounts for each role, and that would be a lot of duplication of effort. Git doesn't have fine-grained user access control out of the box.

Shared authentication

Connecting using token-based authentication is useful not only to keep from having to enter your user name and password in a third-party tool, but you can also set specific privileges per token. And, if needed, tokens can be easily revoked, which makes them a more flexible option if the sole purpose is for connecting a tool.

There are three parts to setting up a connection using a token:

- Generate a token in the tool (Jira, GitHub, or Jenkins)
- Set up basic authentication credentials in DevOps
- Use DevOps to connect to the tool

Hosted Git servers

Many organizations can't use services hosted within another organization's walls at all. These might be government organizations or organizations dealing with money, such as banking,

insurance, and gaming organizations. The causes might be legal or, simply nervousness about letting critical code leave the organization's doors, so to speak.

If you have no such qualms, it is quite reasonable to use a hosted service, such as GitHub or GitLab, that offers private accounts.

Using GitHub or GitLab is, at any rate, a convenient way to get to learn to use Git and explore its possibilities. It is also quite easy to install a GitLab instance on your organization's premises.

Both vendors are easy to evaluate, given that they offer free accounts where you can get to know the services and what they offer. See if you really need all the services or if you can make do with something simpler.

Some of the features offered by both GitLab and GitHub over plain Git are as follows:

- Web interfaces
- A documentation facility with an in-built wiki
- Issue trackers
- Commit visualization
- Branch visualization
- The pull request workflow

While these are all useful features, it's not always the case that you can use the facilities provided. For instance, you might already have a wiki, a documentation system, an issue tracker, and so on that you need to integrate with.

The most important features we are looking for, then, are those most closely related to managing and visualizing code.

Large binary files

GitHub and GitLab are pretty similar but do have some differences. One of them springs from the fact that source code systems such as Git traditionally didn't cater much for the storage of large binary files. There have always been other ways, such as storing file paths to a file server in plain text files.

But what if you actually have files that are, in a sense, equivalent to source files, except that they are binary, and you still want to version them? Such file types might include image files, video files, and audio files. Modern websites make increasing use of media files, and this area has typically been the domain of content management systems (CMSes). CMSes, however nice they might be, have disadvantages compared to DevOps flows, so the allure of storing media files in your ordinary source handling system is strong. Disadvantages of CMSes include the fact that

they often have quirky or nonexistent scripting capabilities. Automation, another word that should have really been included in the DevOps portmanteau, is therefore often difficult with a CMS.

You can, of course, just check in your binary to Git, and it will be handled like any other file. What happens then is that Git operations involving server operations suddenly become sluggish. And then, some of the main advantages of Git–efficiency and speed–vanish out the window.

Solutions to this problem have evolved over time, and there are currently two contenders:

- Git Large File Storage (LFS), supported by GitHub and GitLab
- git-annex, supported by GitLab, but only in the Enterprise Edition

git-annex is the more flexible; Git LFS is easier to get started with. Both solutions are open source and are implemented as extensions to Git via its plugin mechanism.

There are several other systems, which indicates that this is an unresolved pain point in the current state of Git. git-annex has a comparison between the different versions at **http://git-annex.branchable.com/not/**.

If you need to perform version control of your media files, you should start by exploring Git LFS. It is easy to get started with.

It should also be noted that the primary benefit of this type of solution is the ability to version media files together with the corresponding code. When working with code, you can examine the differences between versions of code conveniently. Examining differences between media files is harder and less useful.

In a nutshell, git-annex uses a tried and tested solution to data logical problems: adding a layer of indirection. It does this by storing symlinks to files in the repository. The binary files are then stored in the filesystem and fetched by local workspaces using other means, such as rsync. This involves more work to set up the solution, of course. Git LFS uses another approach, which is easier to set up, but offers less flexibility.

Trying out different Git server implementations

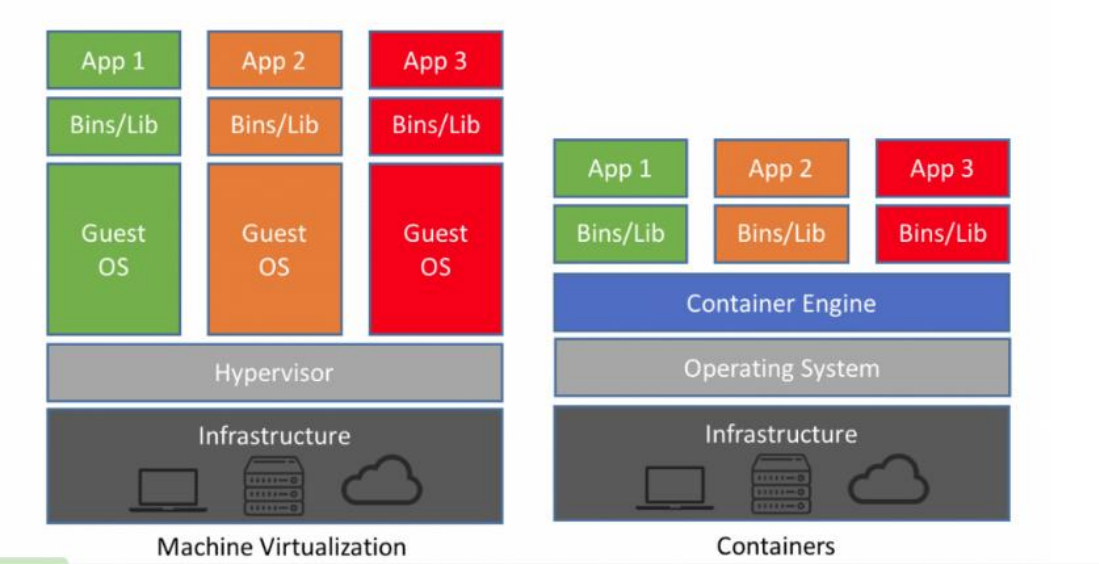
The distributed nature of Git makes it possible to try out different Git implementations for various purposes. The client-side setup will be similar regardless of how the server is set up.

You can also have several solutions running in parallel. The client side is not unduly complicated by this, since Git is designed to handle several remotes if need be.

Docker intermission

Docker is a software development platform for virtualization with multiple Operating systems running on the same host. It helps to separate infrastructure and applications in order to deliver software quickly. Unlike Hypervisors, which are used for creating VM (Virtual machines), virtualization in Docker is performed on system-level, also called Docker containers.

As you can see the difference in the image below, Docker containers run on top of the host's Operation system. This helps you to improves efficiency and security. Moreover, we can run more containers on the same infrastructure than we can run Virtual machines because containers use fewer resources.



Virtualization in Docker vs Hypervisor

Unlike the VMs which can communicate with the hardware of the host (ex: Ethernet adapter to create more virtual adapters) Docker containers run in an isolated environment on top of the host's OS. Even if your host runs Windows OS, you can have Linux images running in containers with the help of Hyper-V, which automatically creates small VM to virtualize the system's base image, in this case, Linux.

To get started with Docker, follow these steps:

1. To begin with, install Docker according to the particular instructions for your operating system. For Red Hat derivatives, it's a simple dnf install docker command. Then, the docker service needs to be running:

```
systemctl enable dockersystemctl start docker
```

2. We need another tool, Docker Compose. If you don't have it available in your package repositories for your distribution, follow the instructions at https://docs.docker.com/compose/install/.

Gerrit

A basic Git server is good enough for many purposes. Sometimes, you need precise control over the workflow, though.

One concrete example is merging changes into configuration code for critical parts of the infrastructure. While my opinion is that it's central to DevOps to not place unnecessary red tape around infrastructure code, there is no denying that it's sometimes necessary. If nothing else, developers might feel nervous committing changes to the infrastructure and would like for someone more experienced to review the code changes.

Gerrit is a Git-based code review tool that can offer a solution in these situations. In brief, Gerrit allows you to set up rules to allow developers to review and approve changes made to a code base by other developers. These might be senior developers reviewing changes made by inexperienced developers, or the more common case, which is simply that more eyeballs on the code is good for quality in general.

Gerrit is Java-based and uses a Java-based Git implementation under the hood.

Gerrit can be downloaded as a Java WAR file and has an integrated setup method. It needs a relational database as a dependency, but you can opt to use an integrated Java-based H2 database that is good enough for evaluating Gerrit.

An even simpler method is using Docker to try out Gerrit. There are several Gerrit images on the Docker registry hub to choose from. The following one was selected for this evaluation exercise: https://hub.docker.com/r/openfrontier/gerrit/.

To run a Gerrit instance with Docker, follow these steps:

1. Initialize and start Gerrit:

```
docker run -d -p 8080:8080 -p 29418:29418 openfrontier/gerrit
```

2. Open your browser to http://<docker host url>:8080.

Now, we can try out the code review feature we would like to have.

Installing the git-review package

Install git-review on your local installation:

```
sudo dnf install git-review
```

This will install a helper application for Git to communicate with Gerrit. It adds a new command, git-review, that is used instead of git push to push changes to the Gerrit Git server.

The value of history revisionism

When we work with code together with other people in a team, the code's history becomes more important than when we work on our own. The history of changes to files becomes a way to communicate. This is especially important when working with code review and code review tools such as Gerrit.

The code changes also need to be easy to understand. Therefore, it is useful, although perhaps counter-intuitive, to edit the history of the changes in order to make the resulting history clearer.

As an example, consider a case where you made a number of changes and later changed your mind and removed them. It is not useful information for someone else that you made a set of edits and then removed them. Another case is when you have a set of commits that are easier to understand if they are a single commit. Adding commits together in this way is called squashing in the Git documentation.

Another case that complicates history is when you merge from the upstream central repository several times, and merge commits are added to the history. In this case, we want to simplify the changes by first removing our local changes, then fetching and applying changes from the upstream repository, and then, finally, reapplying our local changes. This process is called rebasing.

Both squashing and rebasing apply to Gerrit.

The changes should be clean, preferably one commit. This isn't something that is particular to Gerrit; it's easier for a reviewer to understand your changes if they are packaged nicely. The review will be based on this commit:

1. To begin with, we need to have the latest changes from the Git/Gerrit server side. We rebase our changes on top of the server-side changes:

```
git pull --rebase origin master
```

1. Then, we polish our local commits by squashing them:

```
git rebase -i origin/master
```

Now, let's have a look at the Gerrit web interface:

We can now approve the change, and it is merged to the master branch.

There is much to explore in Gerrit, but these are the basic principles and should be enough to base an evaluation on.

Are the results worth the hassle, though? These are my observations:

· Gerrit allows fine-grained access to sensitive code bases. Changes can go in after being reviewed by authorized personnel.
This is the primary benefit of Gerrit. If you just want to have mandatory code reviews for

unclear reasons, don't. The benefit has to be clear for everyone involved. It's better to agree on other more informal methods of code review than an authoritarian system.

- If the alternative to Gerrit is to not allow access to code bases at all, even read-only access, then implement Gerrit.
- Some parts of an organization might be too nervous to allow access to things such as infrastructure configuration. This is usually for the wrong reasons. The problem you usually face isn't people taking an interest in your code; it's the opposite.
- Sometimes, sensitive passwords are checked in to code, and this is taken as a reason to disallow access to the source. Well, if it hurts, don't do it. Solve the problem that leads to there being passwords in the repositories instead.

The pull request model

There is another solution to the problem of creating workflows around code reviews: the pull request model, which has been made popular by GitHub.

In this model, pushing to repositories can be disallowed except for the repository owners. Other developers are allowed to fork the repository, though, and make changes in their fork. When they are done making changes, they can submit a pull request. The repository owners can then review the request and opt to pull the changes into the master repository.

This model has the advantage of being easy to understand, and many developers have experience in it from the many open source projects on GitHub.

Setting up a system capable of handling a pull request model locally will require something like GitHub or GitLab, which we will look at next.

GitLab

GitLab supports many convenient features on top of Git. It's a large and complex software system based on Ruby. As such, it can be difficult to install, what with getting all the dependencies right and so on.

There is a nice Docker Compose file for GitLab available at https://registry.hub.docker.com/u/sameersbn/gitlab/. If you followed the instructions for Docker shown previously, including the installation of docker-compose, it's now pretty simple to start a local GitLab instance:

```
mkdir gitlab
```

```
cd gitlab
```

```
wget https://raw.githubusercontent.com/sameersbn/docker-gitlab/master/docker-compose.yml
docker-compose up
```

The docker-compose command will read the .yml file and start all the required services in a default demonstration configuration.

If you read the start up log in the console window, you will notice that three separate application containers have been started: gitlab postgresql1, gitlab redis1, and gitlab gitlab1.

The gitlab container includes the Ruby-based web application and Git backend functionality. Redis is a distributed key-value store, and PostgreSQL is a relational database.

If you are used to setting up complicated server functionality, you will appreciate that we have saved a great deal of time with docker-compose.

The docker-compose.yml file sets up data volumes at /srv/docker/gitlab.

To log in to the web user interface, use the administrator password given with the installation instructions for the GitLab Docker image. They have been replicated here, but beware that they might change as the Docker image author sees fit:

- Username: root
- Password: 5iveL!fe

Here is a screenshot of the GitLab web user interface login screen:

Try importing a project to your GitLab server from, for instance, GitHub, or a local private project.

Have a look at how GitLab visualizes the commit history and branches.

While investigating GitLab, you will perhaps come to agree that it offers a great deal of interesting functionality.

When evaluating features, it's important to keep in mind whether it's likely that they will be used after all. What core problem would GitLab, or similar software, solve for you?

It turns out that the primary value added by GitLab, as exemplified by the following two features, is the elimination of bottlenecks in DevOps workflows:

- The management of user SSH keys
- The creation of new repositories

These features are usually deemed to be the most useful.

Visualization features are also useful, but the client-side visualization available with Git clients is more useful to developers.