Unit 3

Introduction to project management

The need for source code control:

Source code control (also known as version control) is an essential part of DevOps practices. Here are a few reasons why:

Collaboration: Source code control allows multiple team members to work on the same codebase simultaneously and track each other's changes.

Traceability: Source code control systems provide a complete history of changes to the code, enabling teams to trace bugs, understand why specific changes were made, and roll back to previous versions if necessary.

Branching and merging: Teams can create separate branches for different features or bug fixes, then merge the changes back into the main codebase. This helps to ensure that different parts of the code can be developed independently, without interfering with each other.

Continuous integration and delivery: Source code control systems are integral to continuous integration and delivery (CI/CD) pipelines, where changes to the code are automatically built, tested, and deployed to production.

In summary, source code control is a critical component of DevOps practices, as it enables teams to collaborate, manage changes to code, and automate the delivery of software.

History of source code management

The history of source code management (SCM) in DevOps dates back to the early days of software development. Early SCM systems were simple and focused on tracking changes to source code over time.

In the late 1990s and early 2000s, the open-source movement and the rise of the internet led to a proliferation of new SCM tools, including CVS (Concurrent Versions System), Subversion, and Git. These systems made it easier for developers to collaborate on projects, manage multiple versions of code, and automate the build, test, and deployment process.

As DevOps emerged as a software development methodology in the mid-2000s, SCM became an integral part of the DevOps toolchain. DevOps teams adopted Git as their SCM tool of choice, leveraging its distributed nature, branch and merge capabilities, and integration with CI/CD pipelines.

Today, Git is the most widely used SCM system in the world, and is a critical component of DevOps practices. With the rise of cloud-based platforms, modern SCM systems also offer

features like collaboration, code reviews, and integrated issue tracking.

Roles and code in Devops

In DevOps, roles and code play a critical role in the development, delivery, and operation of software.

Roles:

- Development team: responsible for writing and testing code.
- Operations team: responsible for the deployment and maintenance of the code in production.
- DevOps team: responsible for bridging the gap between development and operations, ensuring that code is delivered quickly and reliably to production.

Code:

- Code is the backbone of DevOps and represents the software that is being developed, tested, deployed, and maintained.
- Code is managed using source code control systems like Git, which provide a way to track changes to the code over time, collaborate on the code with other team members, and automate the build, test, and deployment process.
- Code is continuously integrated and tested, ensuring that any changes to the code do not cause unintended consequences in the production environment.

In conclusion, both roles and code play a critical role in DevOps. Teams work together to ensure that code is developed, tested, and delivered quickly and reliably to production, while operations teams maintain the code in production and respond to any issues that arise.

Overall, SCM has been an important part of the evolution of DevOps, enabling teams to collaborate, manage code changes, and automate the software delivery process.

Source code management system and migrations

- A source code management (SCM) system is a software application that provides version control for source code. It tracks changes made to the code over time, enabling teams to revert to previous versions if necessary, and helps ensure that code can be collaborated on by multiple team members.
- SCM systems typically provide features such as version tracking, branching and merging, change history, and rollback capabilities. Some popular SCM systems include Git, Subversion, Mercurial, and Microsoft Team Foundation Server.
- Source code management (SCM) systems are often used to manage code migrations, which are the process of moving code from one environment to another. This is typically done as part of a software development project, where code is moved from a development environment to a testing environment and finally to a production

environment.

SCM systems provide a number of benefits for managing code migrations, including:

1. **Version control**
2. **Branching and merging**
3. **Rollback**
4. **Collaboration**
5. **Automation**

1) Version control: SCM systems keep a record of all changes to the code, enabling teams to track the code as it moves through different environments.

Purpose of Version Control:

- Multiple people can work simultaneously on a single project. Everyone works on and edits their own copy of the files and it is up to them when they wish to share the changes made by them with the rest of the team.
- It also enables one person to use multiple computers to work on a project, so it is valuable even if you are working by yourself.
- It integrates the work that is done simultaneously by different members of the team. In some rare cases, when conflicting edits are made by two people to the same line of a file, then human assistance is requested by the version control system in deciding what should be done.
- Version control provides access to the historical versions of a project. This is insurance against computer crashes or data loss. If any mistake is made, you can easily roll back to a previous version. It is also possible to undo specific edits that too without losing the work done in the meanwhile. It can be easily known when, why, and by whom any part of a file was edited.

Benefits of the version control system:

- Enhances the project development speed by providing efficient collaboration,
- Leverages the productivity, expedites product delivery, and skills of the employees through better communication and assistance,
- Reduce possibilities of errors and conflicts meanwhile project development through traceability to every small change,
- Employees or contributors of the project can contribute from anywhere irrespective of the different geographical locations through this VCS,
- For each different contributor to the project, a different working copy is maintained and not merged to the main file unless the working copy is validated. The most popular example is Git, Helix core, Microsoft TFS,
- Helps in recovery in case of any disaster or contingent situation,
- Informs us about Who, What, When, Why changes have been made.

Types of Version Control Systems:

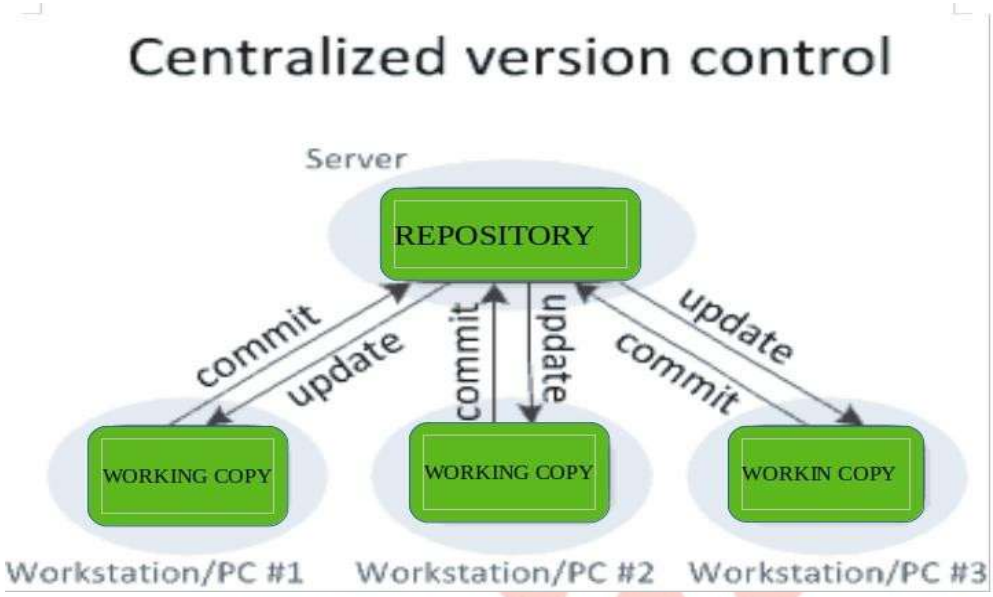
- Local Version Control Systems
- Centralized Version Control Systems
- Distributed Version Control Systems

Local Version Control Systems: It is one of the simplest forms and has a database that kept all the changes to files under revision control. RCS is one of the most common VCS tools. It keeps patch sets (differences between files) in a special format on disk. By adding up all the patches it can then re-create what any file looked like at any point in time.

Centralized Version Control Systems: Centralized version control systems contain just one repository globally and every user need to commit for reflecting one's changes in the repository. It is possible for others to see your changes by updating.

Two things are required to make your changes visible to others which are:

- You commit
- They update



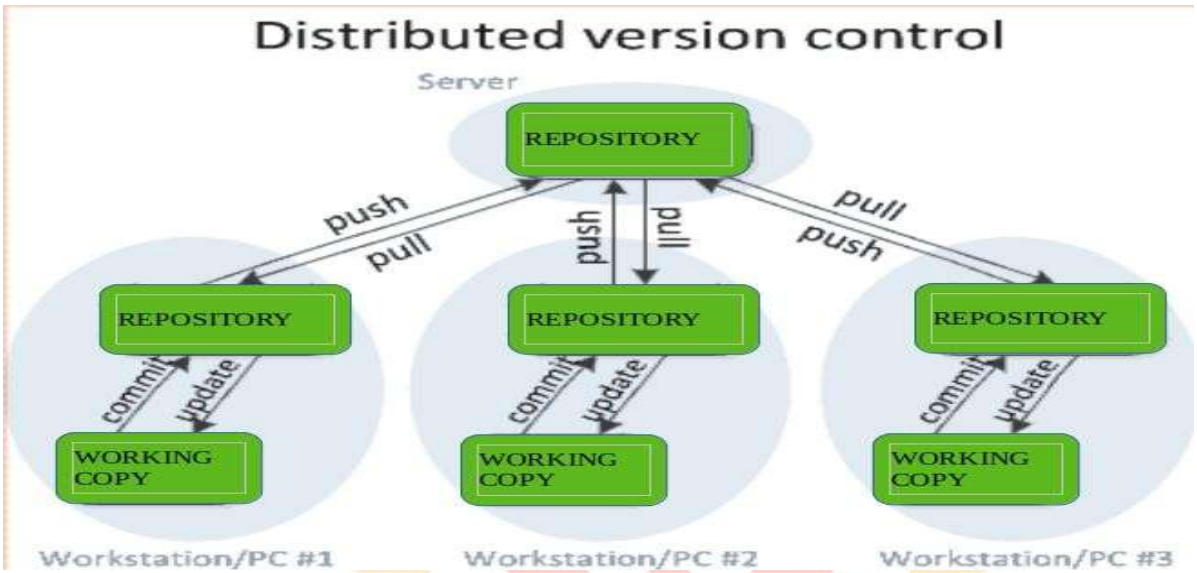
The benefit of CVCS (Centralized Version Control Systems) makes collaboration amongst developers along with providing an insight to a certain extent on what everyone else is doing on the project. It allows administrators to fine-grained control over who can do what.
It has some downsides as well which led to the development of DVS. The most obvious is the single point of failure that the centralized repository represents if it goes down during that period collaboration and saving versioned changes is not possible. What if the hard disk of the central database becomes corrupted, and proper backups haven't been kept? You lose absolutely everything.

Distributed Version Control Systems:

Distributed version control systems contain multiple repositories. Each user has their own repository and working copy. Just committing your changes will not give others access to your changes. This is because commit will reflect those changes in your local repository and you need to push them in order to make them visible on the central repository. Similarly, When you update, you do not get others' changes unless you have first pulled those changes into your repository.
To make your changes visible to others, 4 things are required:

- You commit
- You push
- They pull
- They update

The most popular distributed version control systems are Git, and Mercurial. They help us overcome the problem of single point of failure.



2)Branching and merging: Teams can create separate branches of code for different environments, making it easier to manage the migration process.

Branching and merging are key concepts in Git-based version control systems, and are widely used in DevOps to manage the development of software.

Branching in Git allows developers to create a separate line of development for a new feature or bug fix. This allows developers to make changes to the code without affecting the main branch, and to collaborate with others on the same feature or bug fix.

Merging in Git is the process of integrating changes made in one branch into another branch. In DevOps, merging is often used to integrate changes made in a feature branch into the main branch, incorporating the changes into the codebase.

Branching and merging provide several benefits in DevOps:

Improved collaboration: By allowing multiple developers to work on the same codebase at the same time, branching and merging facilitate collaboration and coordination among team members.

Improved code quality: By isolating changes made in a feature branch, branching and merging make it easier to thoroughly review and test changes before they are integrated into the main codebase, reducing the risk of introducing bugs or other issues.

Increased transparency: By tracking all changes made to the codebase, branching and merging provide a clear audit trail of how code has evolved over time.

Overall, branching and merging are essential tools in the DevOps toolkit, helping to improve collaboration, code quality, and transparency in the software development process.

Rollback: In the event of a problem during a migration, teams can quickly revert to a previous version of the code.

Rollback in DevOps refers to the process of reverting a change or returning to a previous version of a system, application, or infrastructure component. Rollback is an important capability in DevOps, as it provides a way to quickly and efficiently revert changes that have unintended consequences or cause problems in production.

There are several approaches to rollback in DevOps, including:

Version control: By using a version control system, such as Git, DevOps teams can revert to a previous version of the code by checking out an earlier commit.

Infrastructure as code: By using infrastructure as code tools, such as Terraform or Ansible, DevOps teams can roll back changes to their infrastructure by re-applying an earlier version of the code.

Continuous delivery pipelines: DevOps teams can use continuous delivery pipelines to automate the rollback process, by automatically reverting changes to a previous version of the code or infrastructure if tests fail or other problems are detected.

Snapshots: DevOps teams can use snapshots to quickly restore an earlier version of a system or infrastructure component.

Overall, rollback is an important capability in DevOps, providing a way to quickly revert changes that have unintended consequences or cause problems in production. By using a combination of version control, infrastructure as code, continuous delivery pipelines, and snapshots, DevOps teams can ensure that their systems and applications can be quickly and easily rolled back to a previous version if needed.

Collaboration: SCM systems enable teams to collaborate on code migrations, with team members working on different aspects of the migration process simultaneously.

Collaboration is a key aspect of DevOps, as it helps to bring together development, operations, and other teams to work together towards a common goal of delivering high-quality software quickly and efficiently.

In DevOps, collaboration is facilitated by a range of tools and practices, including:

Version control systems: By using a version control system, such as Git, teams can collaborate on code development, track changes to source code, and merge code changes from multiple contributors.

Continuous integration and continuous deployment (CI/CD): By automating the build, test, and deployment of code, CI/CD pipelines help to streamline the development process and reduce the risk of introducing bugs or other issues into the codebase.

Code review: By using code review tools, such as pull requests, teams can collaborate on code development, share feedback, and ensure that changes are thoroughly reviewed and tested before they are integrated into the codebase.

Issue tracking: By using issue tracking tools, such as JIRA or GitHub Issues, teams can collaborate on resolving bugs, tracking progress, and managing the development of new features.

Communication tools: By using communication tools, such as Slack or Microsoft Teams, teams can collaborate and coordinate their work, share information, and resolve problems quickly and efficiently.

Overall, collaboration is a critical component of DevOps, helping teams to work together effectively and efficiently to deliver high-quality software. By using a range of tools and practices to facilitate collaboration, DevOps teams can improve the transparency, speed, and quality of their software development processes.

Automation: Many SCM systems integrate with continuous integration and delivery (CI/CD) pipelines, enabling teams to automate the migration process.

In conclusion, SCM systems play a critical role in managing code migrations. They provide a way to track code changes, collaborate on migrations, and automate the migration process, enabling teams to deliver code quickly and reliably to production.

Shared authentication

Shared authentication in DevOps refers to the practice of using a common identity management system to control access to the various tools, resources, and systems used in software development and operations. This helps to simplify the process of managing users and permissions and ensures that everyone has the necessary access to perform their jobs. Examples of shared authentication systems include Active Directory, LDAP, and SAML-based identity providers.

Hosted Git servers

Hosted Git servers are online platforms that provide Git repository hosting services for software development teams. They are widely used in DevOps to centralize version control of source code, track changes, and collaborate on code development. Some popular hosted Git servers include GitHub, GitLab, and Bitbucket. These platforms offer features such as pull requests, code reviews, issue tracking, and continuous integration/continuous deployment (CI/CD) pipelines. By using a hosted Git server, DevOps teams can streamline their development processes and collaborate more efficiently on code projects.

Different Git server implementations

There are several different Git server implementations that organizations can use to host their Git repositories. Some of the most popular include:

GitHub: One of the largest Git repository hosting services, GitHub is widely used by developers for version control, collaboration, and code sharing.

GitLab: An open-source Git repository management platform that provides version control, issue tracking, code review, and more.

Bitbucket: A web-based Git repository hosting service that provides version control, issue tracking, and project management tools.

Gitea: An open-source Git server that is designed to be lightweight, fast, and easy to use.

Gogs: Another open-source Git server, Gogs is designed for small teams and organizations and provides a simple, user-friendly interface.

GitBucket: A Git server written in Scala that provides a wide range of features, including issue tracking, pull requests, and code reviews.

Organizations can choose the Git server implementation that best fits their needs, taking into account factors such as cost, scalability, and security requirements.

Docker intermission

Docker is an open-source project with a friendly-whale logo that facilitates the deployment of applications in software containers. It is a set of PaaS products that deliver containers (software packages) using OS-level virtualization. It embodies resource isolation features of the Linux kernel but offers a friendly API.

In simple words, Docker is a tool or platform design to simplify the process of creating, deploying, and packaging and shipping out applications along with its parts such as libraries and other dependencies. Its primary purpose is to automate the application deployment process and operating-system-level virtualization on Linux. It allows multiple containers to run on the same hardware and provides high productivity, along with maintaining isolated applications and facilitating seamless configuration.

Docker benefits include:

- High ROI and cost savings
- Productivity and standardization
- Maintenance and compatibility
- Rapid deployment

- Faster configurations
- Seamless portability
- Continuous testing and deployment
- Isolation, segregation, and security

Docker vs. Virtual Machines

Virtual Machine is an application environment that imitates dedicated hardware by providing an emulation of the computer system. Docker and Vmboth have their set of benefits and uses, but when it comes to running applications in multiple environments, both can be utilized. So which one wins? Let's get into a quick Docker vs. VM comparison.

OS Support: VM requires a lot of memory when installed in an OS, whereas Docker containers occupy less space.

Performance: Running several VMs can affect the performance, whereas, Docker containers are stored in a single Docker engine; thus, they provide better performance.
Boot-up time: VMs have a longer booting time compared to Docker. Efficiency: VMs have lower efficiency than Docker.

Scaling: VMs are difficult to scale up, whereas Docker is easy to scale up.

Space allocation: You cannot share data volumes with VMs, but you can share and reuse them among various Docker containers.

Portability: With VMs, you can face compatibility issues while porting across different platforms; Docker is easily portable.
Clearly, Docker is a hands-down winner.

Gerrit

Gerrit is a web-based code review tool which is integrated with Git and built on top of Git version control system (helps developers to work together and maintain the history of their work). It allows to merge changes to Git repository when you are done with the code reviews.

Gerrit was developed by *Shawn Pearce* at Google which is written in Java, Servlet, GWT (Google Web Toolkit). The stable release of Gerrit is 2.12.2 and published on March 11, 2016 licensed under *Apache License v2*.

Why Use Gerrit?

Following are certain reasons, why you should use Gerrit.

- You can easily find the error in the source code using Gerrit.
- You can work with Gerrit, if you have regular Git client; no need to install any Gerrit client.
- Gerrit can be used as an intermediate between developers and git repositories.

Features of Gerrit

- Gerrit is a free and an open source Git version control system.
- The user interface of Gerrit is formed on *Google Web Toolkit*.
- It is a lightweight framework for reviewing every commit.
- Gerrit acts as a repository, which allows pushing the code and creates the review for your commit.

Advantages of Gerrit

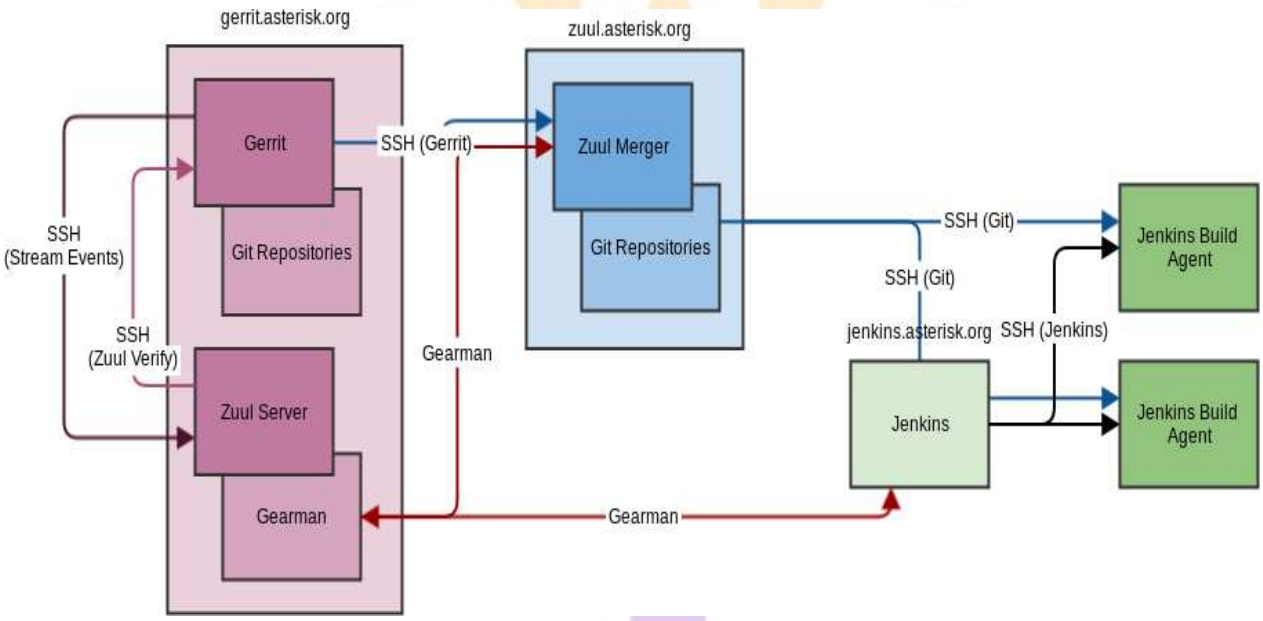
- Gerrit provides access control for Git repositories and web frontend for code review.
- You can push the code without using additional command line tools.
- Gerrit can allow or decline the permission on the repository level and down to the branch level.
- Gerrit is supported by Eclipse.

Disadvantages of Gerrit

- Reviewing, verifying and resubmitting the code commits slows down the time to market.
- Gerrit can work only with Git.
- Gerrit is slow and it's not possible to change the sort order in which changes are listed.
- You need administrator rights to add repository on Gerrit.

What is Gerrit?

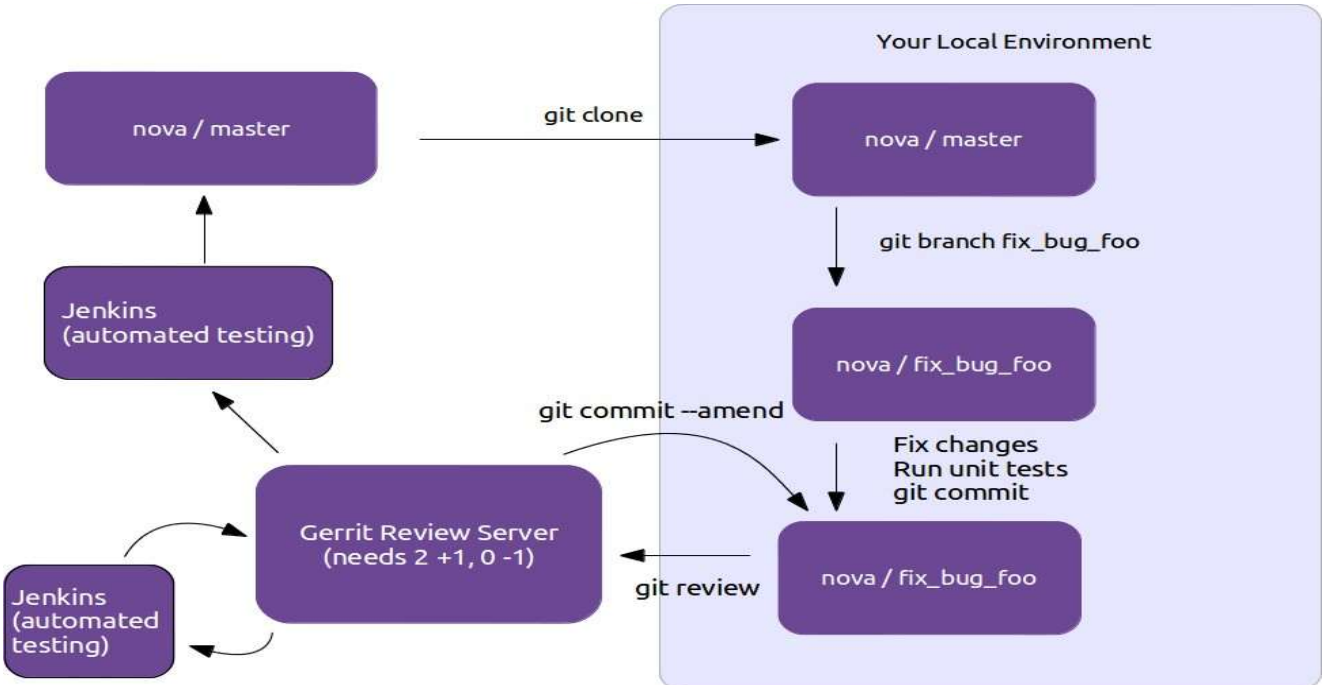
Gerrit is an exceptionally extensible and configurable apparatus for online code survey and storehouse the executives for projects utilizing the Git rendition control framework. Gerrit is similarly helpful where all clients are believed committers, for example, might be the situation with shut source business advancement.



It is used to store the merged code base and the changes under review that have not being merged yet. Gerrit has the limitation of a single repository per project.

Gerrit is first and foremost an arranging region where changes can be looked at prior to turning into a piece of the code base. It is likewise an empowering agent for this survey cycle, catching notes and remarks about the progressions to empower conversation of the change. This is especially valuable with conveyed groups where this discussion can't occur eye to eye.

How Gerrit Works Architecture?



Use case of Gerrit

- Knowledge exchange:
  - The code review process allows newcomers to see the code of other more experienced developers.
  - Developers can get feedback on their suggested changes.
  - Experienced developers can help to evaluate the impact on the whole code.
  - Shared code ownership: by reviewing code of other developers the whole team gets a solid knowledge of the complete code base.

The pull request model

Pull request is a feature of Git-based version control systems that allows developers to propose changes to a Git repository and request feedback or approval f rom other team members. It is widely used in DevOps to facilitate collaboration and code review in the software development process.

In the pull request model, a developer creates a new branch in a Git repository, makes changes to the code, and then opens a pull request to merge the changes into the main branch. Other team members can then review the changes, provide feedback, and approve or reject the request.

Pull Requests are a mechanism popularized by github, used to help facilitate merging of work, particularly in the context of open-source projects. A contributor works on their contribution in a fork (clone) of the central repository. Once their contribution is finished they create a pull request to notify the owner of the central repository that their work is ready to be merged into the mainline. Tooling supports and encourages code review of the contribution before accepting the request. Pull requests have become widely used in software development, but critics are concerned by the addition of integration friction which can prevent continuous integration.

Pull requests essentially provide convenient tooling for a development workflow that existed in many open-source projects, particularly those using a distributed source-control system (such as git). This workflow begins with a contributor creating a new logical branch, either by starting a new branch in the central repository, cloning into a personal repository, or both. The contributor then works on that branch, typically in the style of a Feature Branch, pulling any updates from Mainline into their branch. When they are done they communicate with the maintainer of the central repository indicating that they are done, together with a reference to their commits. This reference could be the URL of a branch that needs to be integrated, or a set of patches in an email.

Once the maintainer gets the message, she can then examine the commits to decide if they are ready to go into mainline. If not, she can then suggest changes to the contributor, who then has opportunity to adjust their submission. Once all is ok, the maintainer can then merge, either with a regular merge/rebase or applying the patches from the final email.

Github's pull request mechanism makes this flow much easier. It keeps track of the clones through its fork mechanism, and automatically creates a message thread to discuss the pull request, together with behavior to handle the various steps in the review workflow. These conveniences were a major part of what made github successful and led to "pull request" becoming a fundamental part of the developer's lexicon.

So that's how pull requests work, but should we use them, and if so how? To answer that question, I like to step back from the mechanism and think about how it works in the context of a source code management workflow. To help me think about that, I wrote down a series of patterns for managing source code branching. I find understanding these (specifically the Base and Integration patterns) clarifies the role of pull requests.

In terms of these patterns, pull requests are a mechanism designed to implement a combination of Feature Branching and Pre-Integration Reviews. Thus to assess the usefulness of pull requests we first need to consider how applicable those patterns are to our situation. Like most patterns,

they are sometimes valuable, and sometimes a pain in the neck - we have to examine them based on our specific context. Feature Branching is a good way of packaging together a logical contribution so that it can be assessed, accepted, or deferred as a single unit. This makes a lot of sense when contributors are not trusted to commit directly to mainline. But Feature Branching comes at a cost, which is that it usually limits the frequency of integration, leading to complicated merges and deterring refactoring. Pre-Integration Reviews provide a clear place to do code review at the cost of a significant increase in integration friction. [1]

That's a drastic summary of the situation (I need a lot more words to explain this further in the feature branching article), but it boils down to the fact that the value of these patterns, and thus the value of pull requests, rest mostly on the social structure of the team. Some teams work better with pull requests, some teams would find pull requests a severe drag on the effectiveness. I suspect that since pull requests are so popular, a lot of teams are using them by default when they would do better without them.

While pull requests are built for Feature Branches, teams can use them within a Continuous Integration environment. To do this they need to ensure that pull requests are small enough, and the team responsive enough, to follow the CI rule of thumb that everybody does Mainline Integration at least daily. (And I should remind everyone that Mainline Integration is more than just merging the current mainline into the feature branch). Using the ship/show/ask classification can be an effective way to integrate pull requests into a more CI-friendly workflow.
The wide usage of pull requests has encouraged a wider use of code review, since pull requests provide a clear point for Pre-Integration Review, together with tooling that encourages it. Code review is a Good Thing, but we must remember that a pull request isn't the only mechanism we can use for it. Many teams find great value in the continuous review afforded by Pair Programming. To avoid reducing integration frquency we can carry out post-integration code review in several ways. A formal process can record a review for each commit, or a tech lead can examine risky commits every couple of days. Perhaps the most powerful form of code review is one that's frequently ignored. A team that takes the attitude that the codebase is a fluid system, one that can be steadily refined with repeated iteration carries out Refinement Code
Review every time a developer looks at existing code. I often hear people say that pull requests are necessary because without them you can't do code reviews - that's rubbish. Pre-integration code review is just one way to do code reviews, and for many teams it isn't the best choice.

The pull request model provides several benefits in DevOps:

Improved code quality: Pull requests encourage collaboration and code review, helping to catch potential bugs and issues before they make it into the main codebase.

Increased transparency: Pull requests provide a clear audit trail of all changes made to the code, making it easier to understand how code has evolved over time.

Better collaboration: Pull requests allow developers to share their work and get feedback from others, improving collaboration and communication within the development team.

Overall, the pull request model is an important tool in the DevOps toolkit, helping to improve the quality, transparency, and collaboration of software development processes.

GitLab

GitLab is an open-source Git repository management platform that provides a wide range of features for software development teams. It is commonly used in DevOps for version control, issue tracking, code review, and continuous integration/continuous deployment (CI/CD) pipelines.

GitLab provides a centralized platform for teams to manage their Git repositories, track changes to source code, and collaborate on code development. It offers a range of tools to support code review and collaboration, including pull requests, code comments, and merge request approvals.

In addition, GitLab provides a CI/CD pipeline tool that allows teams to automate the process of building, testing, and deploying code. This helps to streamline the development process and reduce the risk of introducing bugs or other issues into the codebase.

Overall, GitLab is a comprehensive Git repository management platform that provides a wide range of tools and features for software development teams. By using GitLab, DevOps teams can improve the efficiency, transparency, and collaboration of their software development processes.

What is Git?

Git is a distributed version control system, which means that a local clone of the project is a complete version control repository. These fully functional local repositories make it easy to work offline or remotely. Developers commit their work locally, and then sync their copy of the repository with the copy on the server. This paradigm differs from centralized version control where clients must synchronize code with a server before creating new versions of code.

Git's flexibility and popularity make it a great choice for any team. Many developers and college graduates already know how to use Git. Git's user community has created resources to train developers and Git's popularity make it easy to get help when needed. Nearly every development environment has Git support and Git command line tools implemented on every major operating system.

Git basics

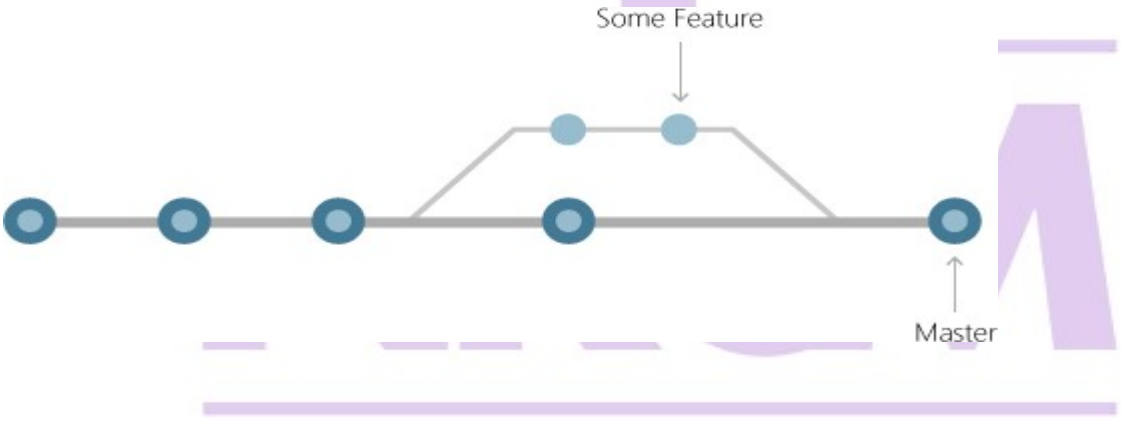
Every time work is saved, Git creates a commit. A commit is a snapshot of all files at a point in time. If a file hasn't changed from one commit to the next, Git uses the previously stored file.

This design differs from other systems that store an initial version of a file and keep a record of deltas over time.

Commits create links to other commits, forming a graph of the development history. It's possible to revert code to a previous commit, inspect how files changed from one commit to the next, and review information such as where and when changes were made. Commits are identified in Git by a unique cryptographic hash of the contents of the commit. Because everything is hashed, it's impossible to make changes, lose information, or corrupt files without Git detecting it.

Branches

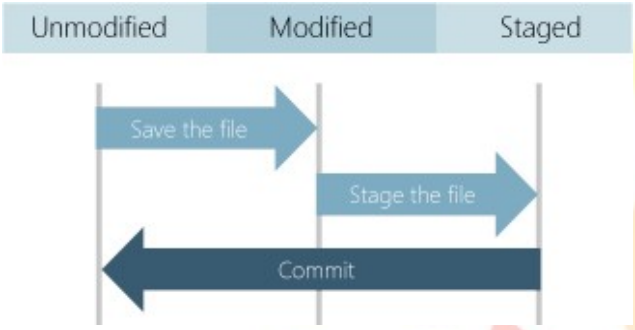
Each developer saves changes to their own local code repository. As a result, there can be many different changes based off the same commit. Git provides tools for isolating changes and later merging them back together. Branches, which are lightweight pointers to work in progress, manage this separation. Once work created in a branch is finished, it can be merged back into the team's main (or trunk) branch.



Files and commits

Files in Git are in one of three states: modified, staged, or committed. When a file is first modified, the changes exist only in the working directory. They aren't yet part of a commit or the development history. The developer must *stage* the changed files to be included in the commit. The staging area contains all changes to include in the next commit. Once the developer is happy

with the staged files, the files are packaged as a *commit* with a message describing what changed. This commit becomes part of the development history.



Staging lets developers pick which file changes to save in a commit in order to break down large changes into a series of smaller commits. By reducing the scope of commits, it's easier to review the commit history to find specific file changes.

Benefits of Git

The benefits of Git are many.

Simultaneous development

Everyone has their own local copy of code and can work simultaneously on their own branches. Git works offline since almost every operation is local.

Faster releases

Branches allow for flexible and simultaneous development. The main branch contains stable, high-quality code from which you release. Feature branches contain work in progress, which are merged into the main branch upon completion. By separating the release branch from development in progress, it's easier to manage stable code and ship updates more quickly.

Built-in integration

Due to its popularity, Git integrates into most tools and products. Every major IDE has built-in Git support, and many tools support continuous integration, continuous deployment, automated testing, work item tracking, metrics, and reporting feature integration with Git. This integration simplifies the day-to-day workflow.

Strong community support

Git is open-source and has become the de facto standard for version control. There is no shortage of tools and resources available for teams to leverage. The volume of community support for Git compared to other version control systems makes it easy to get help when needed.

Git works with any team

Using Git with a source code management tool increases a team's productivity by encouraging collaboration, enforcing policies, automating processes, and improving visibility and traceability of work. The team can settle on individual tools for version control, work item tracking, and continuous integration and deployment. Or, they can choose a solution like GitHub or Azure DevOps that supports all of these tasks in one place.

Pull requests

Use pull requests to discuss code changes with the team before merging them into the main branch. The discussions in pull requests are invaluable to ensuring code quality and increase knowledge across your team. Platforms like GitHub and Azure DevOps offer a rich pull request experience where developers can browse file changes, leave comments, inspect commits, view builds, and vote to approve the code.

Branch policies

Teams can configure GitHub and Azure DevOps to enforce consistent workflows and process across the team. They can set up branch policies to ensure that pull requests meet requirements before completion. Branch policies protect important branches by preventing direct pushes, requiring reviewers, and ensuring clean builds.