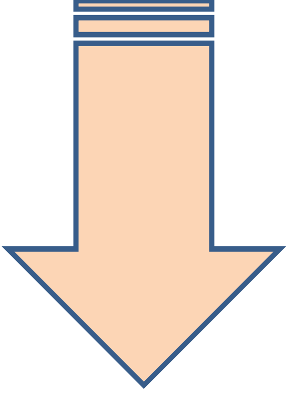
DEVOPS

DevOps



DevOps

UNIT-1:

- **Introduction to Devops:** definition of Devops, Devops Stake holders, Devops goals, Devops life cycle, **Devops stages:** version control, continuous integration, continuous delivery, continuous deployment, continuous monitoring.
- **Git basics**, Git features, installing Git, Git essentials, common commands in Git, Working with remote repositories.

UNIT-2:

Continuous integration using Jenkins: Introduction-Understanding continuous integration, introduction about Jenkins, Build Cycle, Jenkins Architecture, installation, Jenkin Management, Adding a slave node to Jenkins, Building Delivery Pipeline, Pipeline as a Code.

UNIT-3:

Continuous Deployment: Containerization with Docker, Containerization using Kubernetes

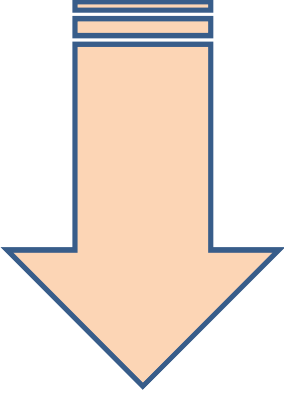
UNIT-4:

Configuration Management & Continuous Monitoring: Configuration Management with Puppet, Configuration Management with Ansible, Continuous Monitoring with Nagios.

UNIT-1:

Introduction to Devops: definition of Devops, Devops Stake holders, Devops goals, Devops life cycle, **Devops stages:** version control, continuous integration, continuous deliver, continuous deployment, continuous monitoring.

Git basics, Git features, installing Git, Git essentials, common commands in Git, Working with remote repositories.



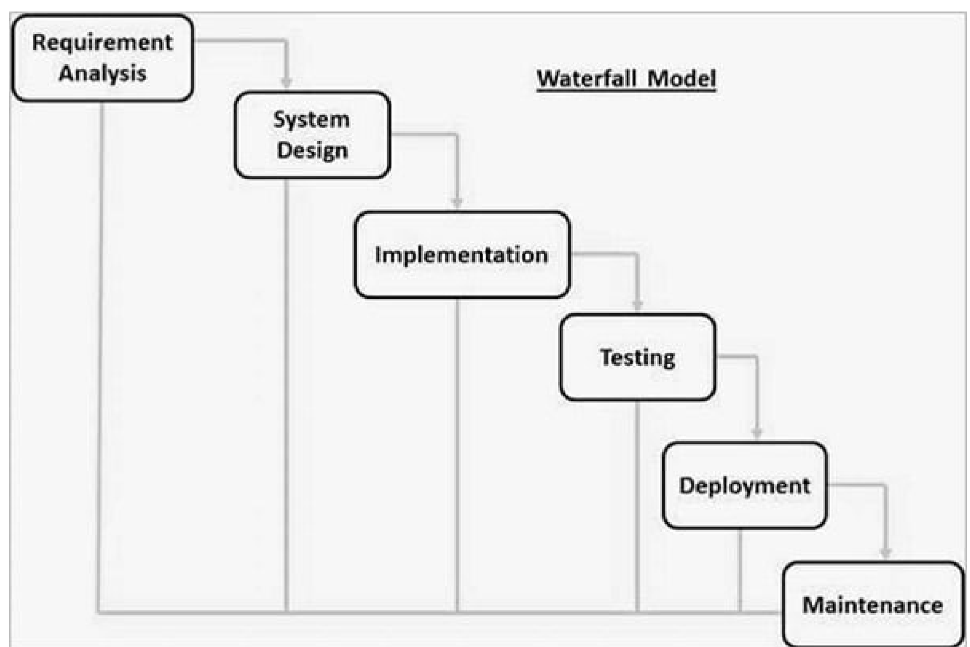
DevOps

Introduction to Devops:

SDLC models: Waterfall Model, Incremental Model, RAD Model, Iterative Model, Spiral Model, Prototype model, Agile Model.

Waterfall model:

- The Waterfall Model was the first Process Model to be introduced. It was widely used as it was easy to understand & implement .
- WaterFall Model is also known as **Linear-Sequential Life Cycle Model** because work is done linearly - the next stage begins after completion of the previous stage . Output of the previous stage is the input of the next stage .
- The name **"WaterFall Model"** is because the process looks like flowing steadily downwards - as shown below just like a waterfall .



DevOps

Waterfall model:

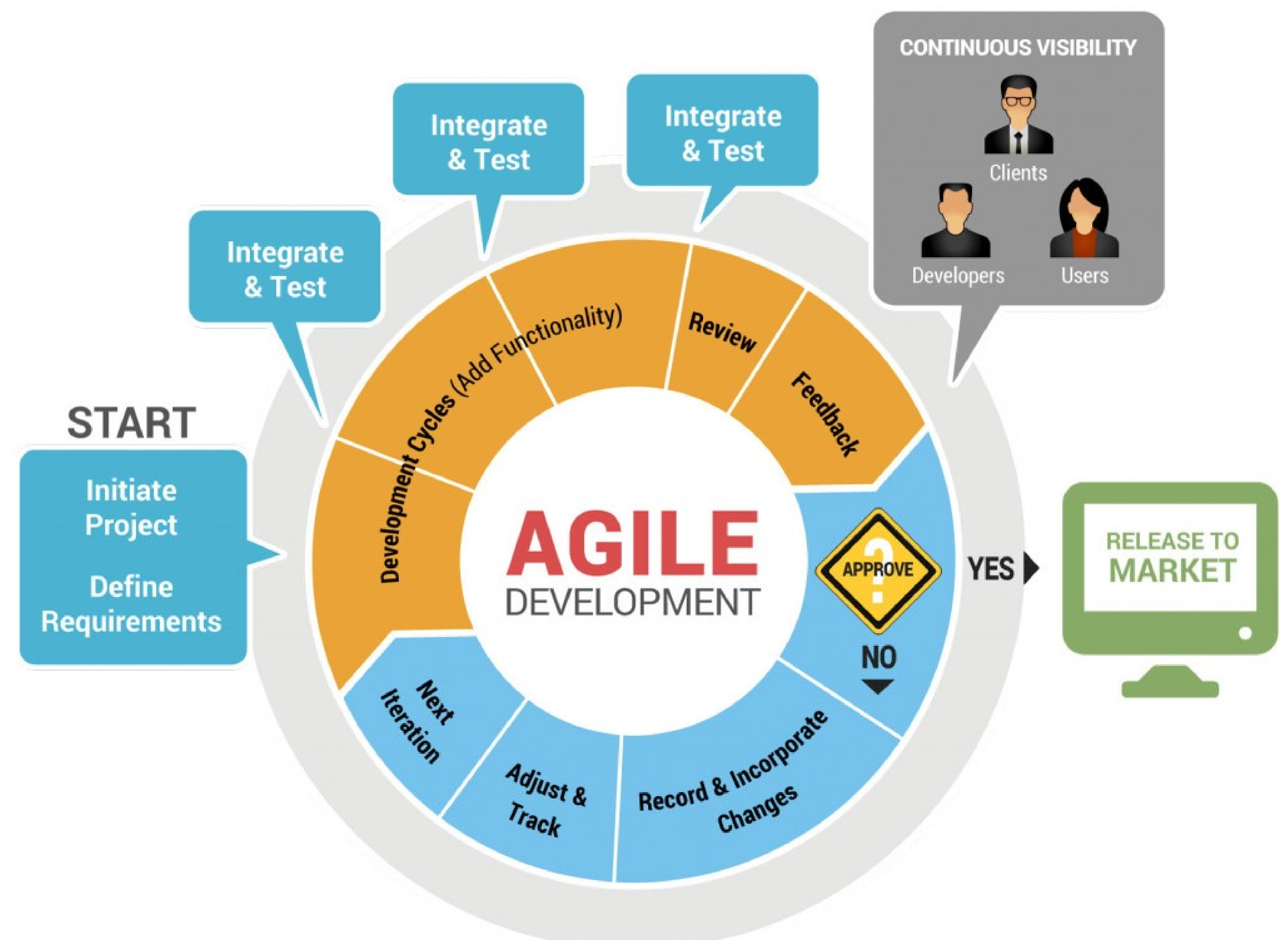
- The Phases are:-
- **Requirement Analysis:** In this phase all the requirements about the project are gathered . For eg features in projects , etc . All these requirements are well documented in the SRS (Software Requirement Specifications) document.
- **System Design:** Systems Design is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. The SRS document is then used to design the system.
- **Implementation:** After the System design phase is completely over . Implementation phase starts . Here each unit is individually developed.
- **Testing & Integration:** Each individual unit is tested for potential bugs or errors . After testing is successful , individual units are integrated together.
- **Deployment:** Once the integration is done, the software is deployed to production servers.
- **Maintenance:** Project is continuously being monitored for any user inconvenience or bugs . Time to time **new patches** are released as a result of any bug fixes.

DevOps

Agile Methodology:

- In practical terms, agile software development methodologies are all about delivering small pieces of working software quickly to improve customer satisfaction.
- These methodologies use adaptive approaches and teamwork to focus on continuous improvement.
- Instead of developing software sequentially from one phase to the next, which is how the waterfall method ensures product quality, an agile method can promote development and testing as concurrent and continuous processes. Put another way, waterfall development holds that an entire phase should be completed before moving on to the next, whereas agile supports multiple sequences happening at the same time.

Agile Model



Agile - Advantages

- **Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.** Customer satisfaction and quality deliverables are the focus.

- **Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.** Don't fight change, instead learn to take advantage of it.

- **Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.** Continually provide results throughout a project, not just at its peeks.

- **Business people and developers must work together daily throughout the project.** Collaboration is key.

- **Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.** Bring talented and hardworking members to the team and get out of their way.

- **The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.** Eliminate as many opportunities for miscommunication as possible.

- **Working software is the primary measure of progress.** It doesn't need to be perfect, it needs to work.

Agile - Advantages

- **Agile processes promote sustainable(stable) development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.** Slow and steady wins the race.
- **Continuous attention to technical excellence and good design enhances agility.** Don't forget to pay attention to the small stuff.
- **Simplicity—the art of maximizing the amount of work not done—is essential.** Trim the fat.
- **The best architectures, requirements, and designs emerge from self-organizing teams.** Related to Principle 5, you'll get the best work from your team if you let them figure out their own roles.
- **At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.** Elicit(extract) and provide feedback, absorb the feedback, and adjust where needed.

Agile - disadvantages

- **Less predictable.** The flexibility at the core of the Agile method also means a much lower degree of predictability**. It can be much more difficult to accurately estimate the time necessary or quantify the resources** and efforts required to complete a project. Many teams fear this uncertainty, and that fear can lead to frustration and **poor decision-making**.
- **More time and commitment.** Communication and collaboration is great, but that constant interaction takes more time and energy for everyone involved.
- **Greater demands on developers and clients.** Commitment from everyone involved is required for Agile Methodology to be effective. Anyone who isn't on board can negatively impact the quality of a project.
- **Lack of necessary documentation.** Because tasks are often completed just in time for development under the Agile Method, **documentation tends to be less thorough**, which can lead to misunderstanding and difficulties down the road.
- **Projects easily fall off track.** The less-structured nature of Agile Methodology means projects can easily go astray or run beyond the original scope of the project.

Agile - Limitations

- **LIMITED SUPPORT FOR A REMOTE AGILE TEAM**

  The different geographical location can become one of the crucial limitations of agile methods, as it can cause many problems in so many different ways.

  I) Time zone differences, communication gap, maintainig documents from distant places
- **LIMITED SUPPORT FOR DEVELOPMENT INVOLVING LARGE TEAMS**
- Larger agile teams have sub-teams of developers who may be working from different areas. Large teams focus on large projects and commonly needed to solve complex problems. Only the senior programmers and developers are capable, of taking the kind of decisions required during the development process. Since the agile approach prone on the universality and cross-functionality of agile teams but the reality is quite the opposite.
- **LIMITED SUPPORT FOR DEVELOPING SAFETY-CRITICAL SOFTWARE**

Agile - Limitations

- **THE RISK INVOLVED IN FOLLOWING CUSTOMERS' REQUIREMENTS**

  If the customer is not sure about what final outcome that they demand, then the project can easily get taken off from the track. Selecting the most complex or difficult component may introduce the danger of failing to create the system the customer needs.

- **AGILE DEVELOPMENT AS A MICROMANAGED APPROACH**

  The pressure is persistently on for every week deliveries hence quality suffers, all they care about is the timeline. The loss of control at the management level leads to micro-management. Agile development is a true challenge. Especially in case, you attempt to communicate with those who are far from development methods whatsoever.

- **AGILE CULTURE IS A LONG TERM APPROACH IN REAL TIME AGILE DEVELOPMENT IMPLEMENTATION**

  The agile does not work in a hierarchy-driven organizational setup. The agile school of thought circles around the smart moves of the digital chessboard. It is the collection of trackers, pointers, calculators, planners, testers, and designing tools. Which divide the task into the possibly smallest cycles and assign it to different team members.

Agile - Limitations

- **AGILE METHODOLOGY CAN BE TRICKIER TO IMPLEMENT**

  Agile will not work in a scenario where a flaw is not an option. Today, agile is more than an agile manifesto; individual and interactions, working software, customer collaboration and responding to change. Agile is not just a team mechanism or computer game but it is also like thinking agile, doing agile and be agile. Today, agile is about business agility, communicating agility and evolving with agility.

- **MANAGEMENT FAILURES OF AGILE SOFTWARE DEVELOPMENT**

  Agile team organism has the ability to glue together the versatile members in one team. Thus it can uplift the burden of the heaviest and the speediest mechanism of any organization. To merge the Excellency and proficiency, of all-around members into one team is an imperative need of the super evolving era of digital transformation.

DevOps

- ✓ The meaning of Agile is swift or versatile."**Agile process model**" refers to a software development approach based on iterative development.
- ✓ Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.
- Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.
-

DevOps

- ✓ The meaning of Agile is swift or versatile."**Agile process model**" refers to a software development approach based on iterative development.
- ✓ Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.
- Each iteration is considered as a short time "frame" in the Agile process model, which typically lasts from one to four weeks. The division of the entire project into smaller parts helps to minimize the project risk and to reduce the overall project delivery time requirements. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.
-

DevOps

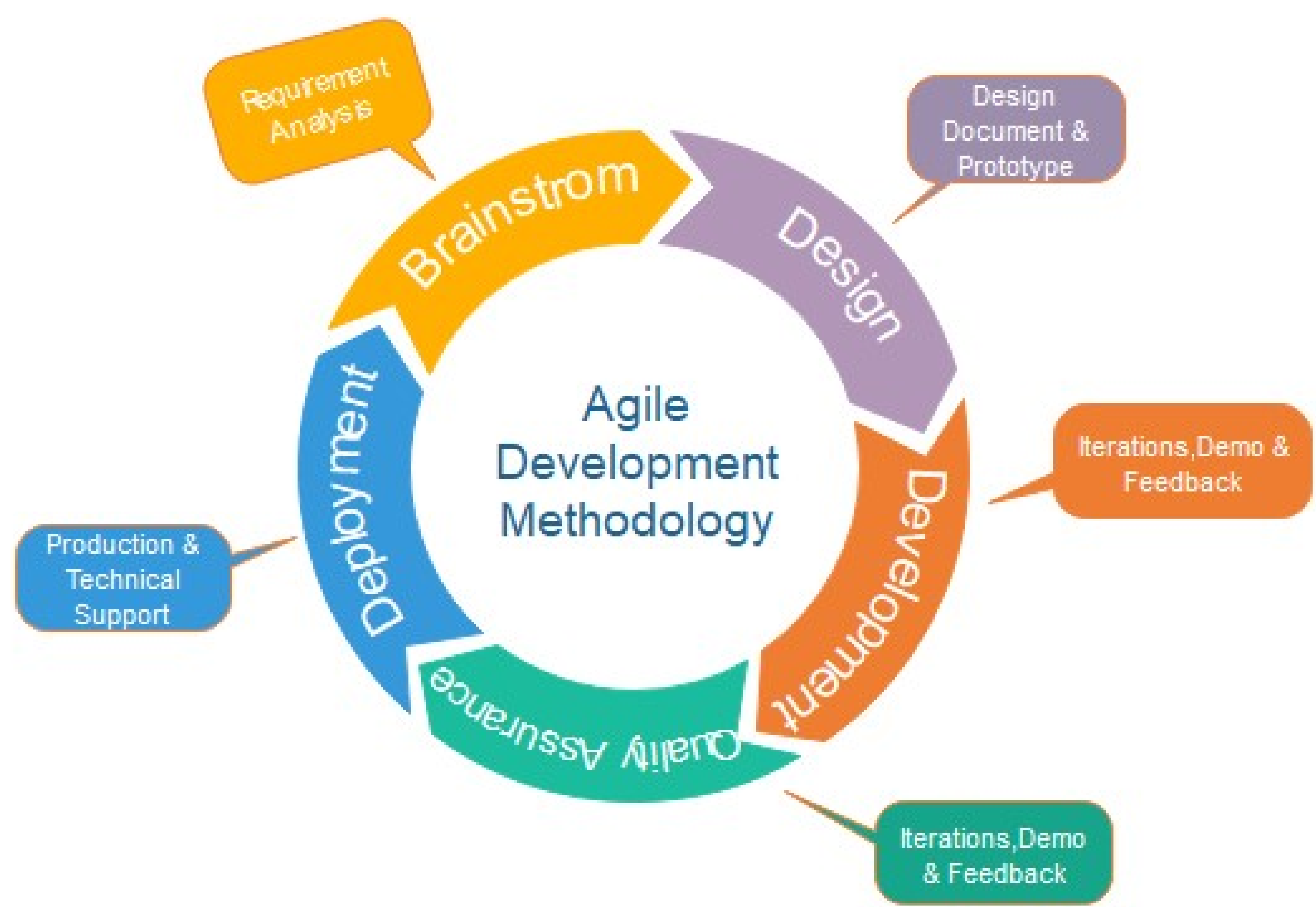


Fig. Agile Model

Agile

When to Use:

- When frequent changes are required.
- When a highly qualified and experienced team is available.
- When a customer is ready to have a meeting with a software team all the time.
- When project size is small.

Adv:

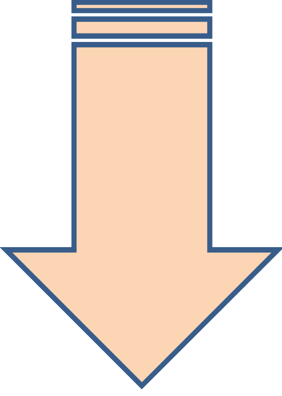
- Frequent Delivery
- Face-to-Face Communication with clients.
- Efficient design and fulfils the business requirement.
- Anytime changes are acceptable.
- It reduces total development time.

DisAdv:

- Due to the **shortage of formal documents**, it creates confusion and crucial decisions taken throughout various phases can be misinterpreted at any time by different team members.
- Due to the **lack of proper documentation**, once the project completes and the developers allotted to another project, maintenance of the finished project can become a difficulty.

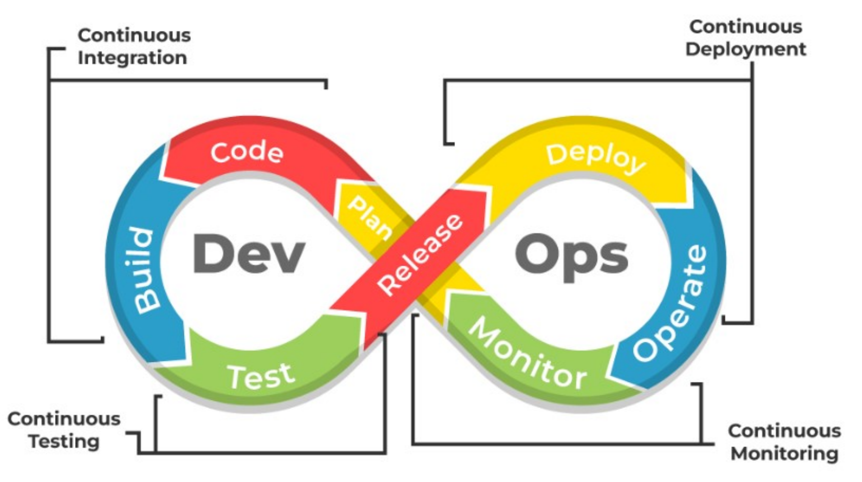
UNIT-1 contd...:

Introduction to Devops: definition of Devops, Devops Stake holders, Devops goals, Devops life cycle, **Devops stages:** version control, continuous integration, continuous deliver, continuous deployment, continuous monitoring.



DevOps

- ✓ definition of Devops
- ✓ Devops Stake holders
- ✓ Devops goals
- ✓ Devops life cycle



definition of Devops:

- The DevOps is the combination of two words, one is **Development** and other is **Operations**. It is a culture to promote the development and operation process collectively.
- This allows a single team to handle the entire application lifecycle, from development to **testing, deployment**, and **operations**.
- DevOps helps you to reduce the disconnection between software developers, quality assurance (QA) engineers, and system administrators.
- DevOps helps to increase organization speed to deliver applications and services. It also allows organizations to serve their customers better and compete more strongly in the market.
- **DevOps can also be defined as a sequence of development and IT operations with better communication and collaboration.**
- DevOps has become one of the most valuable business disciplines for enterprises or organizations. With the help of DevOps, **quality**, and **speed** of the application delivery has improved to a great extent.

Why DevOps?

- The operation and development team worked in complete isolation.
- After the design-build, the testing and deployment are performed respectively. That's why they consumed more time than actual build cycles.
- Without the use of DevOps, the team members are spending a large amount of time on designing, testing, and deploying instead of building the project.
- Manual code deployment leads to human errors in production.
- Coding and operation teams have their separate timelines and are not in synch, causing further delays.

DevOps – Advantages & Disadvantages

Advantages:

- DevOps is an excellent approach for quick development and deployment of applications.
- It responds faster to the market changes to improve business growth.
- DevOps escalate business profit by decreasing software delivery time and transportation costs.
- DevOps clears the descriptive process, which gives clarity on product development and delivery.
- It improves customer experience and satisfaction.
- DevOps simplifies collaboration and places all tools in the cloud for customers to access.
- DevOps means collective responsibility, which leads to better team engagement and productivity.

Disadvantages:

- DevOps professional or expert's developers are less available.
- Developing with DevOps is so expensive.
- Adopting new DevOps technology into the industries is hard to manage in short time.
- Lack of DevOps knowledge can be a problem in the continuous integration of automation projects.

DevOps – Stakeholders

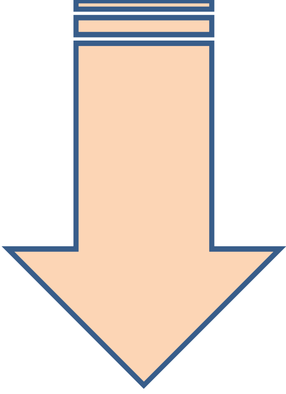
Dev Includes all people involved in developing software products and services including but not exclusive to:

- Architects,
- business representatives,
- customers,
- product owners,
- project managers,
- quality assurance (QA),
- testers and analysts,
- Suppliers etc...

Ops **Includes** all people involved in delivering and managing software products and services including but not exclusive to:

- Information security professionals,
- systems engineers,
- system administrators,
- IT operations engineers,
- release engineers,
- database administrators (DBAs),
- network engineers,
- support professionals,
- third party vendors and suppliers…

Devops goals, Devops life cycle, **Devops stages:** version control, continuous integration, continuous deliver, continuous deployment, continuous monitoring.



DevOps – Goals

1. Ensures effective collaboration between teams
2. Creates scalable infrastructure platforms
3. Builds on-demand release capabilities
4. Provides faster feedback

1. Ensures effective collaboration between teams

- Effective collaboration in any process relies on shared ownership.
- During the development process, all those involved should embrace the fact that everyone is equally responsible for the entire development process. Whether it is development, testing, or deployment, each team member should be involved.
- They should understand that they have an equal stake in the final outcome.
- In the DevOps paradigm, passing of work from one team to another is completely defined and broken down. This accelerates the entire process of development since collaboration between all the teams involved is streamlined.

2. Creates scalable infrastructure platforms

- The primary focus of DevOps is to create a sustainable infrastructure for applications that make them highly scalable.
- According to the demands of the modern-day business world, scalable apps have become an absolute necessity. In an ideal situation, the process of scaling should be reliable and fully automated.
- As a result, the app will have the ability to adapt to any situation when a marketing effort goes viral. With the app being scalable, it can adjust itself to large traffic volumes and provide an immaculate(clean) user experience.

DevOps – Goals

3. Builds on-demand release capabilities

- Companies must focus on keeping their software in a ‘releasable’ state.
- Continuous delivery will allow the software to add new features and go live at any stage.
- DevOps aims to automate the process of release management because it has several advantages.
- Automated release management is predictable, fast, and very consistent.
- Through automation, companies can **release new versions** as per their requirements.
- Automated release management also has complete and thorough **audit trials**, as these are essential for compliance purposes.

4. Provides faster feedback

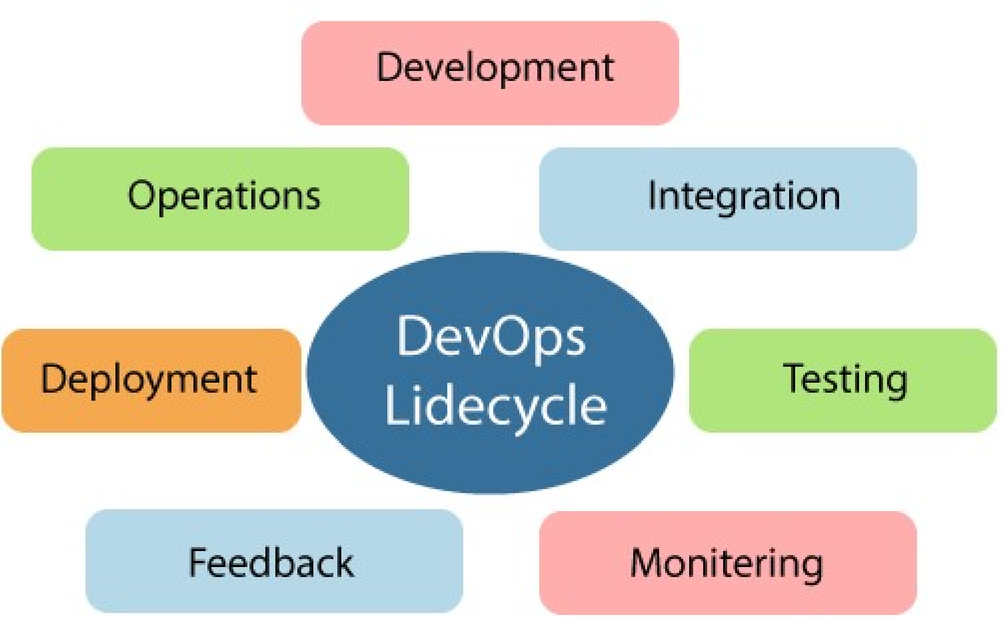
- Automating monotonous tasks such as testing and reporting will accelerate the process of rapid feedback.
- Since the development team will know what has to change, it can roll out the updated version faster. In addition, the team can better understand the impact of the changes that it has done in the software lifecycle.
- A concrete understanding of changes will assist team members in working efficiently in cool manner.
- With rapid feedback, the operations team and developers can make better decisions collectively and enhance the app’s performance.

DevOps – Life cycle

https://www.javatpoint.com/devops-lifecycle

- **DevOps** defines an agile relationship between operations and Development. It is a process that is practiced by the development team and operational engineers together from beginning to the final stage of the product.
- **The DevOps lifecycle includes seven phases**

DevOps Lifecycle:



DevOps – lifecycle

7 stages in DevOps lifecycle:

1. Continuous Development
2. Continuous Integration
3. Continuous Testing
4. Continuous Monitoring
5. Continuous Feedback
6. Continuous Deployment
7. Continuous Operations

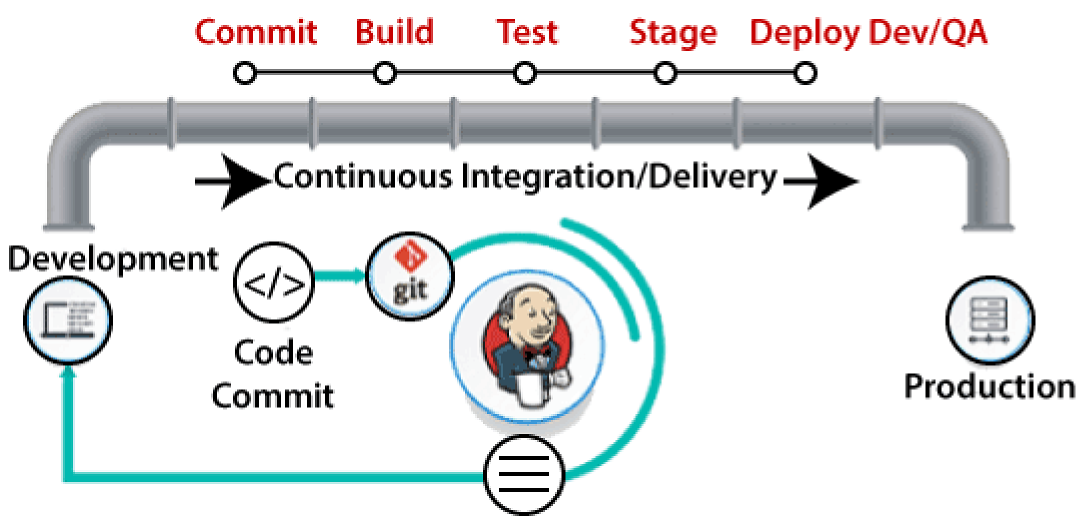
1. Continuous Development:

- This phase involves the planning and coding of the software.
- The vision of the project is decided during the planning phase. And the developers begin developing the code for the application.
- There are no DevOps tools that are required for planning, but there are several tools for maintaining the code.

DevOps – lifecycle

2. Continuous Integration

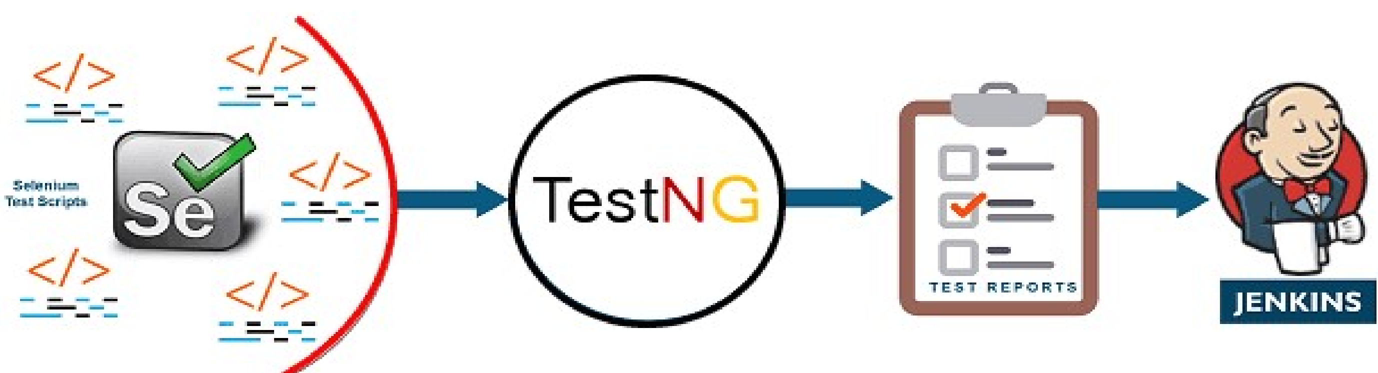
- This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. This may be on a daily or weekly basis. Then every commit is built, and this allows early detection of problems if they are present. Building code is not only involved compilation, but it also includes **unit testing, integration testing, code review**, and **packaging**.
- The code supporting new functionality is continuously integrated with the existing code. Therefore, there is continuous development of software. The updated code needs to be integrated continuously and smoothly with the systems to reflect changes to the end-users.
- **Jenkins** is a popular tool used in this phase. Whenever there is a change in the Git repository, then Jenkins fetches the updated code and prepares a build of that code, which is an executable file in the form of war or jar. Then this build is forwarded to the test server or the production server.



DevOps – lifecycle

3. Continuous Testing

- This phase, where the developed software is continuously testing for bugs. For constant testing, automation testing tools such as **TestNG, JUnit, Selenium**, etc are used. These tools allow QAs to test multiple code-bases thoroughly in parallel to ensure that there is no flaw in the functionality. In this phase, **Docker** Containers can be used for simulating the test environment.



- **Selenium** does the automation testing, and TestNG generates the reports. This entire testing phase can automate with the help of a Continuous Integration tool called **Jenkins**.
- Automation testing saves a lot of time and effort for executing the tests instead of doing this manually. Apart from that, report generation is a big plus. The task of evaluating the test cases that failed in a test suite gets simpler. Also, we can schedule the execution of the test cases at predefined times. After testing, the code is continuously integrated with the existing code.

DevOps – lifecycle

4. Continuous Monitoring

- Monitoring is a phase that involves all the operational factors of the entire DevOps process, where important information about the use of the software is recorded and carefully processed to find out trends and identify problem areas. Usually, the monitoring is integrated within the operational capabilities of the software application.
- It may occur in the form of documentation files or maybe produce large-scale data about the application parameters when it is in a continuous use position. The system errors such as server not reachable, low memory, etc are resolved in this phase. It maintains the security and availability of the service.

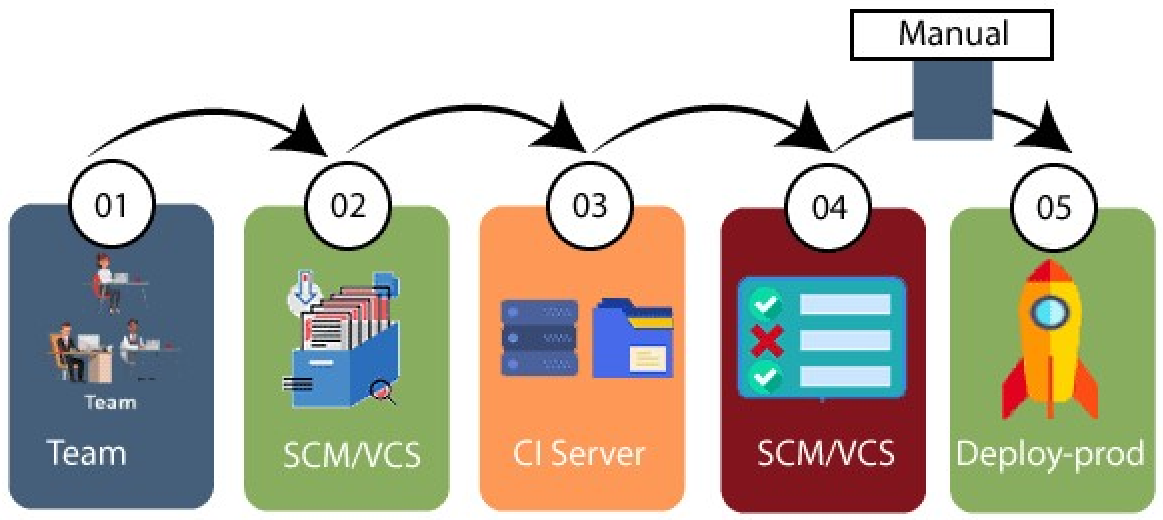
5. Continuous Feedback

- The application development is consistently improved by analyzing the results from the operations of the software. This is carried out by placing the critical phase of constant feedback between the operations and the development of the next version of the current software application.
- The continuity is the essential factor in the DevOps as it removes the unnecessary steps which are required to take a software application from development, using it to find out its issues and then producing a better version.

DevOps – lifecycle

6. Continuous Deployment

- In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers.



- The new code is deployed continuously, and configuration management tools play an essential role in executing tasks frequently and quickly. Here are some popular tools which are used in this phase, such as **Chef, Puppet, Ansible**, and **SaltStack**.

DevOps – lifecycle

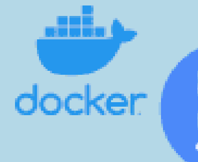
- Containerization tools are also playing an essential role in the deployment phase. **Vagrant** and **Docker** are popular tools that are used for this purpose. These tools help to produce consistency across development, staging, testing, and production environment. They also help in scaling up and scaling down instances softly.
- Containerization tools help to maintain consistency across the environments where the application is tested, developed, and deployed. There is no chance of errors or failure in the production environment as they package and replicate the same dependencies and packages used in the testing, development, and staging environment. It makes the application easy to run on different computers.

7. Continuous Operations:

- All DevOps operations are based on the continuity with complete automation of the release process and allow the organization to accelerate the overall time to market continuingly.

DevOps – Tools

DevOps Tools



DevOps – Tools

https://www.atlassian.com/devops/devops-tools

DevOps life cycle stages:

- Plan
- Build
- Continuous integration and deployment
- Monitor
- Operate
- Plan

Build

DevOps – Tools

Production-identical environments for development:

Infrastructure as code:

GitHub

Continuous integration and delivery

DevOps – Tools

Continuous Integration:

Test:

Jira Software

Deployment:

DevOps – Tools

Operate

pingdom
Nagios

sumo logic

Continuous Feedback

GetFeedback

Jira Service Management

pendo

- Application and server performance monitoring:
-

Jira Service Management   Jira Software   Opsgenie

Statuspage

DevOps – Tools

Jenkins: https://www.jenkins.io/download/

- Jenkins a DevOps tool for monitoring execution of repeated tasks. It is one of the best software deploy tools which helps to integrate project changes more easily by quickly finding issues.
- It supports **continuous integration and continuous delivery**
- Jenkins is an open source automation tool written in Java programming language **that allows continuous integration..**
- Jenkins **builds** and **tests** our software projects which continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.

Features:

- It increases the scale of automation
- Jenkins requires little maintenance and has built-in GUI tool for easy updates.
- It offers 400 plugins to support building and testing virtually any project.
- It is Java-based program ready to run with Operating systems like Windows, Mac OS X, and UNIX
- It supports continuous integration and continuous delivery
- It can easily set up and configured via web interface
- It can distribute tasks across multiple machines thereby increasing concurrency.

https://www.jenkins.io/download/

DevOps – Puppet

Puppet: Puppet is a **configuration management tool** developed by Puppet Labs in order to automate infrastructure management and configuration. Puppet is a very powerful tool which helps in the concept of Infrastructure as code.

- Puppet follows client server model.

Puppet Enterprise is a DevOps tool. It is one of the popular DevOps tools that allows managing **entire infrastructure as code** without expanding the size of the team.

Features:

- Puppet enterprise tool eliminates manual work for software delivery process. It helps developer to deliver great software rapidly
- Model and manage entire environment
- Intelligent orchestration and visual workflows
- Real-time context-aware reporting
- Define and continually enforce infrastructure
- It inspects and reports on packages running across infrastructure
- Desired state conflict detection and remediation

https://puppet.com/try-puppet/puppet-enterprise/

DevOps – Tools

Docker: Docker is a container management service.

Docker is a DevOps technology suite. It allows DevOps teams to build, ship, and run distributed applications. This tool allows users to assemble apps from components and work collaboratively.

- The whole idea of Docker is for developers to easily develop applications, ship them into containers which can then be deployed anywhere.

https://www.docker.com/products/docker-hub

DevOps – Tools

Selenium:

- Selenium is an open-source tool that automates web browsers. It provides a single interface that lets you write test scripts in programming languages like Ruby, Java, NodeJS, PHP, Perl, Python, and C#, among others.

- Selenium is a free (open source) **automated testing suite for web applications** across different browsers and platforms. It is quite similar to HP Quick Test Pro (QTP now UFT) only that Selenium focuses on automating web-based applications. Testing done using Selenium tool is usually referred as Selenium Testing.

- Selenium IDE Tutorial for Beginners (guru99.com)

DevOps – Tools

Ansible:

Ansible is a leading DevOps tool. It is a simple way to automate IT for automating entire application lifecycle. It is one of the best automation tools for DevOps which makes it easier for DevOps teams to scale automation and speed up productivity.

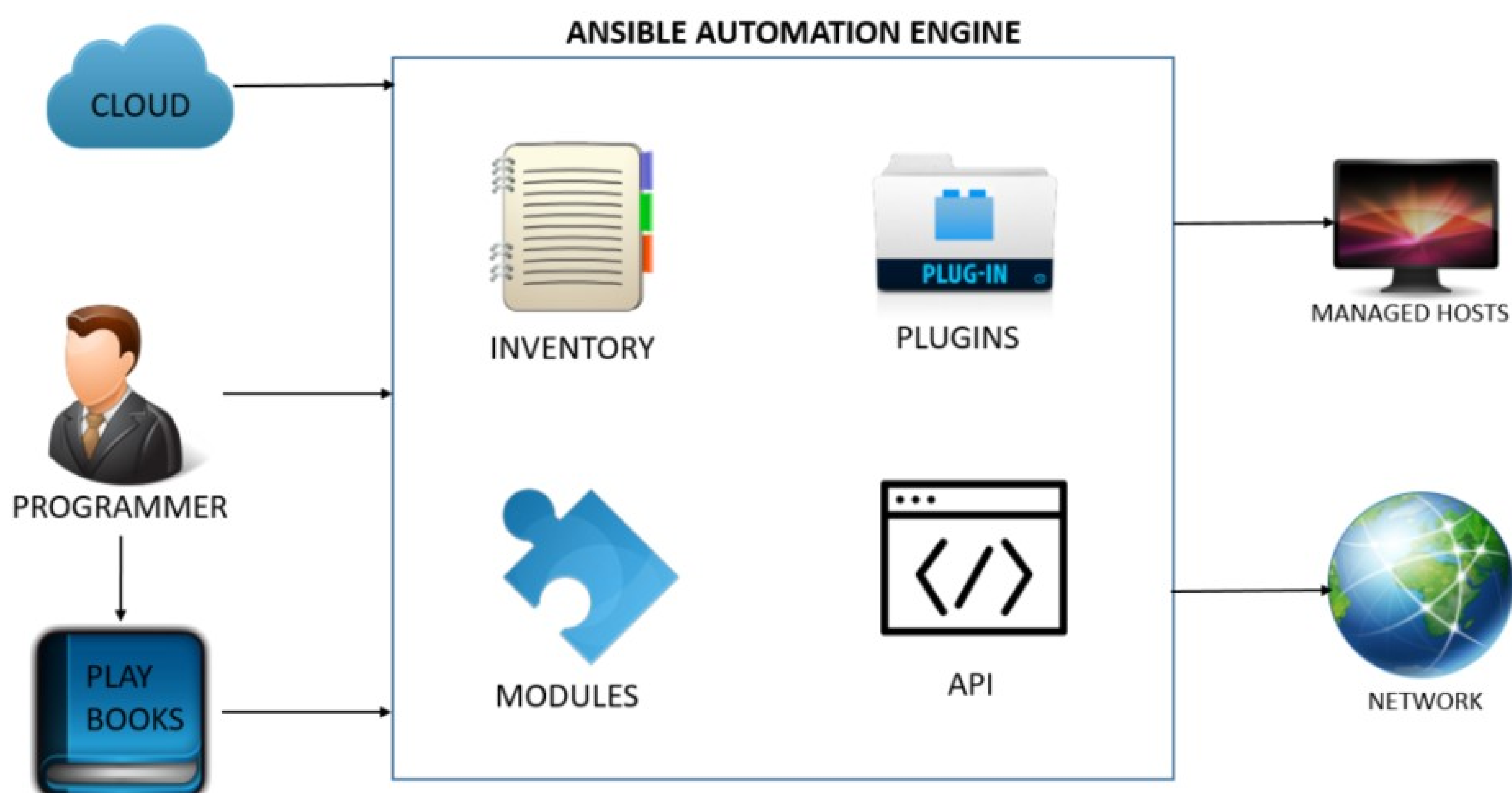
- Ansible is a simple but powerful configuration management and orchestration tool.
- It is fundamentally intended for IT professionals, who use it for configuration management, cloud provisioning, application deployment, intra-service orchestration, updates on workstations and servers, and nearly for anything a systems administrator does on a day-to-day basis.
- often need maintenance, updates, scaling-up activities, for system admins to keep up-to-date of everything manually is a burden and a daunting task. The automation simplifies complex tasks using tools like Ansible,

Features:

- It is easy to use open source deploy apps
- It helps to avoid complexity in the software development process
- IT automation eliminates repetitive tasks that allow teams to do more strategic work
- It is an ideal tool to manage complex deployments and speed up development process

https://www.redhat.com/en/technologies/management/ansible/try-it

DevOps - ANSIBLE tool



DevOps – Tools

Gradle:

- Gradle is a build automation tool known for its flexibility to build software.
- A build automation tool is used **to automate the creation of applications**. The building process includes compiling, linking, and packaging the code. The process becomes more consistent with the help of build automation tools.
- Gradle is the official build tool for Android. Other IDEs are Eclipse, IntelliJIDEA, visual studio 2019 and XCode

DevOps – Tools

Chef:

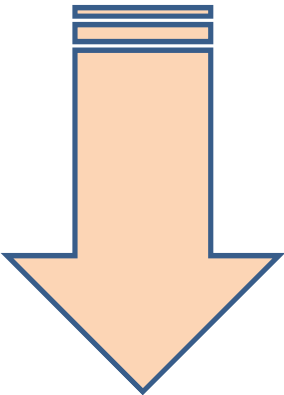
- Today, in an Organization the system admins or ***DevOps Engineer*** spends more time in deploying new services and application, installing and updating network packages and making machine server ready for deployment. This causes tedious human efforts and requires huge human resources.To solve this problem, configuration management was introduced. By using configuration management tools like Chef, Puppet you can deploy, repair and update the entire application infrastructure with automation.
- Chef is a useful DevOps tool for achieving speed, scale, and consistency. It is a Cloud based system. It is one of the best DevOps automation tools that can be used to ease out complex tasks and perform automation.
- Chef is a powerful automation tool that can deploy, repair and update and also manage server and application to any environment.

Features:

- Accelerate cloud adoption
- Effectively manage data centers
- It can manage multiple cloud environments
- It maintains high availability

https://downloads.chef.io/

Git, JUnit, Maven, **Devops stages:** Version Control, continuous integration, continuous deliver, continuous deployment, continuous monitoring.



DevOps – Tools - Git

Git:

- Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.
- Git relies on the basis of distributed development of software where more than one developer may have access to the source code of a specific application and can modify changes to it which may be seen by other developers.
- It allows the user to have "versions" of a project, which show the changes that were made to the code over time, and allows the user to back track if necessary and undo those changes.

DevOps – Tools - JUnit

Junit:

- JUnit is an open source framework designed for the purpose of writing and running tests in the Java programming language
- JUnit is a Java Unit Testing framework that's one of the best test methods for regression testing. An open-source framework, it is used **to write and run repeatable automated tests**

DevOps – Tools - Maven

Maven:

- Maven is a popular **open-source build tool** developed by the Apache Group to build, publish, and deploy several projects at once for better project management.
- Maven is written in Java and is used to build projects written in C#, Scala, Ruby, etc. Based on the Project Object Model (POM), this tool has made the lives of Java developers easier while developing reports, checks build and testing automation setups.

DevOps – Stages

1. Version Control
2. Continuous integration
3. Continuous delivery
4. Continuous deployment
5. Continuous monitoring

DevOps – Stages

Version Control:

- Version control, also known as source control, is the practice of tracking and managing changes to software code.
- VCS are sometimes known as SCM (Source Code Management) tools or RCS (Revision Control System)
- Git is free and open source.
- Version control systems are software tools that help software teams manage changes to source code over time.
- Version control enables teams to deal with conflicts that result from having multiple people working on the same file or project at the same time, and provides a safe way to make changes and roll them back if necessary.
- Using version control early in your team's or product's lifecycle will facilitate adoption of good habits.
- Version control systems help software teams work faster and smarter. They are especially useful for DevOps teams since they help them to reduce development time and increase successful deployments.
- Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

DevOps – Stages

Version Control:

- For an organization, code is the most valuable asset whose value must be protected.
- Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.
- Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.
- Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent concurrent work from conflicting. Changes made in one part of the software can be incompatible with those made by another developer working at the same time. This problem should be discovered and solved in an orderly manner without blocking the work of the rest of the team. Further, in all software development, any change can introduce new bugs on its own and new software can't be trusted until it's tested. So testing and development proceed together until a new version is ready.

DevOps – Stages

Version Control:

- Good version control software supports a developer's preferred workflow without imposing one particular way of working. Ideally it also works on any platform, rather than dictate what operating system or tool chain developers must use. Great version control systems facilitate a smooth and continuous flow of changes to the code rather than the frustrating and clumsy mechanism of file locking - giving the green light to one developer at the expense of blocking the progress of others.
- Software teams that do not use any form of version control often run into problems like not knowing which changes that have been made are available to users or the creation of incompatible changes between two unrelated pieces of work that must then be painstakingly untangled and reworked. If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version. Perhaps you've commented out code blocks because you want to disable certain functionality without deleting the code, fearing that there may be a use for it later. Version control is a way out of these problems.

DevOps – Stages

Version Control:

- Version control software is an essential part of the every-day of the modern software team's professional practices. Individual software developers who are accustomed to working with a capable version control system in their teams typically recognize the incredible value version control also gives them even on small solo projects. Once accustomed to the powerful benefits of version control systems, many developers wouldn't consider working without it even for non-software projects.

DevOps – VCS (Version Control Systems)

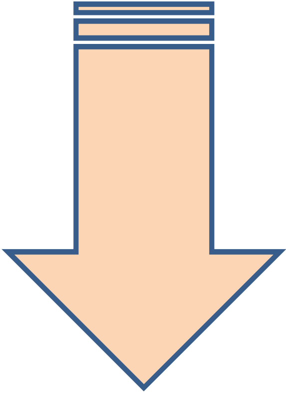
Benefits of VCS:

1. Version control also helps developers move faster and allows software teams to preserve efficiency and agility as the team scales to include more developers.
2. A complete long-term change history of every file. This means every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their contents.

- Having the complete history enables going back to previous versions to help in root cause analysis for bugs and it is crucial when needing to fix problems in older versions of software. If the software is being actively worked on, almost everything can be considered an "older version" of the software.
  Creating a "branch" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict.
- Being able to trace each change made to the software and connect it to project management and bug tracking software such as Jira, and being able to annotate each change with a message describing the purpose and intent of the change can help not only with root cause analysis and other forensics.

Devops stages: Version Control, **continuous integration, continuous delivery, continuous deployment, continuous monitoring.**

Continuous Integration in DevOps | How it is Performed with Advantages (educba.com)



DevOps – Continuous Integration

1. **Continuous integration:**

Definition:

- *Continuous integration refers to the build and unit testing stages of the software release process. Every revision that is committed triggers an automated build and test.*

Continuous Integration in DevOps is the process of automating the build and deploy phase through certain tools and best practices.

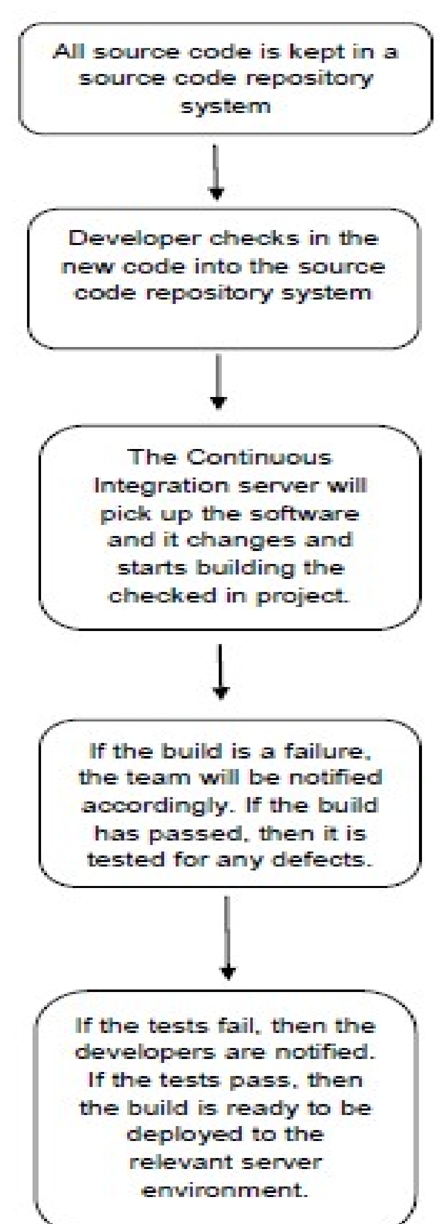
- Continuous integration has become a very integral part of any software development process. The continuous Integration process helps to answer the following questions for the software development team .
- Continuous integration is a DevOps software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration most often refers to the build or integration stage of the software release process and entails both an automation component (e.g. a CI or build service) and a cultural component (e.g. learning to integrate frequently). The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

Addresses:

- Do all the software components work together as they should?
- Is the code too complex for integration purposes?
- Does the code adhere to the established coding standards?
- How much code is covered by automated tests?
- Were all the tests successful after the latest change?

DevOps – Continuous Integration

continuous integration process working:



DevOps – Continuous Integration

continuous integration process working: Steps:

1. First, a developer commits the code to the version control repository. Meanwhile, the Continuous Integration server on the integration build machine polls source code repository for changes (e.g., every few minutes).
2. Soon after a commit occurs, the Continuous Integration server detects that changes have occurred in the version control repository, so the Continuous Integration server retrieves the latest copy of the code from the repository and then executes a build script, which integrates the software.
3. The Continuous Integration server generates feedback by e-mailing build results to the specified project members.
4. Unit tests are then carried out if the build of that project passes. If the tests are successful, the code is ready to be deployed to either the staging or production server.
5. The Continuous Integration server continues to poll for changes in the version control repository and the whole process repeats.

DevOps – Continuous Integration

continuous integration process working: Steps:

1. First, a developer commits the code to the version control repository. Meanwhile, the Continuous Integration server on the integration build machine polls source code repository for changes (e.g., every few minutes).
2. Soon after a commit occurs, the Continuous Integration server detects that changes have occurred in the version control repository, so the Continuous Integration server retrieves the latest copy of the code from the repository and then executes a build script, which integrates the software.
3. The Continuous Integration server generates feedback by e-mailing build results to the specified project members.
4. Unit tests are then carried out if the build of that project passes. If the tests are successful, the code is ready to be deployed to either the staging or production server.
5. The Continuous Integration server continues to poll for changes in the version control repository and the whole process repeats.

DevOps – Continuous Integration

continuous integration process working: Steps:

1. First, a developer commits the code to the version control repository. Meanwhile, the Continuous Integration server on the integration build machine polls source code repository for changes (e.g., every few minutes).
2. Soon after a commit occurs, the Continuous Integration server detects that changes have occurred in the version control repository, so the Continuous Integration server retrieves the latest copy of the code from the repository and then executes a build script, which integrates the software.
3. The Continuous Integration server generates feedback by e-mailing build results to the specified project members.
4. Unit tests are then carried out if the build of that project passes. If the tests are successful, the code is ready to be deployed to either the staging or production server.
5. The Continuous Integration server continues to poll for changes in the version control repository and the whole process repeats.

DevOps – Continuous Integration

- Continuous Integration in DevOps is the process of automating the build and deploy phase through certain tools and best practices. Continuous Integration (CI) applies to all types of software projects such as developing websites, Mobile Applications, and Microservices based APIs. There are three major categories of tools associated with CI: Versioning tool, Build tool, and repositories for centralized artifacts. CI pipeline helps the developer to easily commit the code and helps for quality development.
-

V1.1
CONTINUOUS INTEGRATION
APPROVE DEPLOY
CONTINUOUS DELIVERY
AUTOMATIC DEPLOY
CONTINUOUS DEPLOYMENT
AUTOMATED
AUTOMATED
SOURCE CONTROL
COMMIT CHANGES
BUILD
RUN BUILD AND UNIT TESTS
STAGING
DEPLOY TO TEST ENVIRONMENT
RUN INTEGRATION TESTS, LOAD TESTS, AND OTHER TESTS
PRODUCTION
DEPLOY TO PRODUCTION ENVIRONMENT

DevOps – Continuous Integration

Advantages:

1. Improve Developer Productivity:

- Continuous integration helps your team be more productive by freeing developers from manual tasks and encouraging behaviors that help reduce the number of errors and bugs released to customers.

2. Find and Address Bugs Quicker

ith more frequent testing, your team can discover and address bugs earlier before they grow into larger problems later.

. Deliver Updates Faster

- Continuous integration helps your team deliver updates to their customers faster and more frequently.

CI Tools:

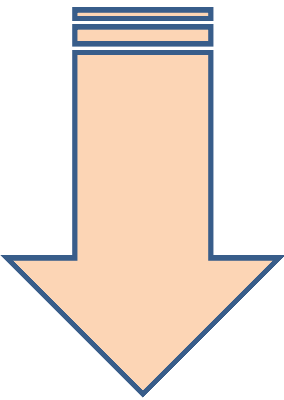
DevOps – Continuous Integration

Continuous Integration

List Of The Top Continuous Integration Tools

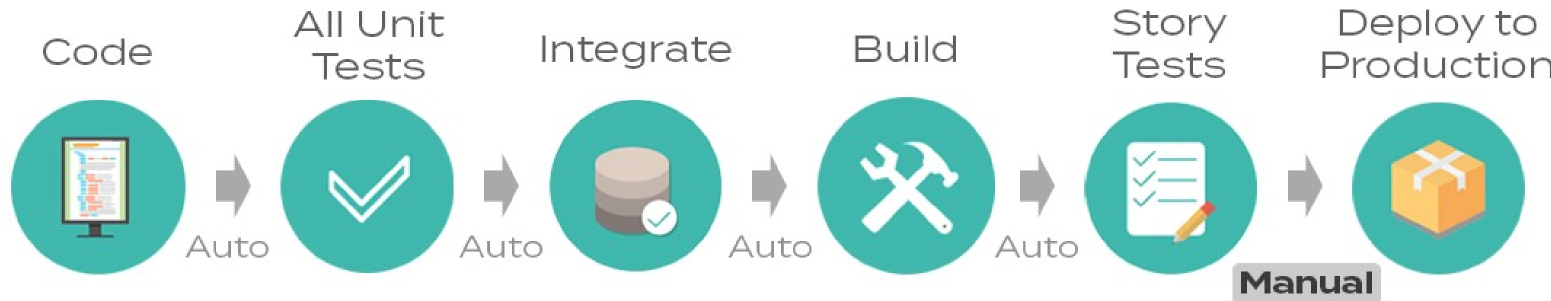
- #1) Buddy
- #2) Jenkins
- #3) Buildbot
- #4) ThoughtWorks
- #5) UrbanCode
- #6) Perforce Helix
- #7) Bamboo
- #8) TeamCity
- #9) CircleCI
- #10) Codeship
- #11) CruiseControl
- #12) Go/GoCD
- #13) Travis
- #14) Integrity
- #15) Strider or Strider CD

Devops stages : continuous delivery, continuous deployment, continuous monitoring.
https://www.flagship.io/glossary/continuous-deployment/
https://devopedia.org/continuous-delivery
https://www.headspin.io/blog/what-is-continuous-monitoring-in-devops



DevOps – Continuous Delivery

- Continuous Delivery (CD) goes one step further from Continuous Integration (CI). It ensures that every code change is tested and ready for the production environment, after a successful build. CI ensures every code is committed to the main code repository whereas CD ensures the system is in an executable state at all times, after changes to code.
- The automation of methods to safely deliver changes into production is known as Continuous Delivery.



- Continuous Delivery can work only if Continuous Integration is in place. It involves running extensive regression, UI, and performance tests to ensure that the code is production-ready. It's applicable to both bug fixes and new feature releases
-

DevOps – benefits of Continuous Delivery

- **Smaller maintenance and support teams:** Apart from developers, product teams generally maintain several parallel maintenance teams. In the traditional model, each team consisting of build/integration/test members is dedicated to a release version. However, in the automated CD process, these tasks happen without human intervention without compromising product quality.
- **Reducing risk of failure for customer releases:** CD aims to make production changes safe and routine. Deploying to production becomes a boring non-event that can be performed as often as needed. Since the magnitude of code change in each release is very small, risk of failure is also minimised.
- **Enforces process discipline in the team:** Since code is always meant to be production ready, team members are constantly tuning their code, refactoring, reviewing, automating repetitive build tasks, reworking test scripts and getting feature-related documentation done without any delays.
-

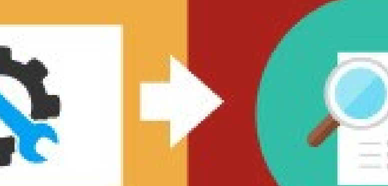
DevOps – Continuous Delivery process

Source

Checkout Code

Developers are committing code to code repositories and which will be triggered by AWS Source stage.

Build



Build Code

Once the code is checkout, code binaries will be build at this stage automatically and send them to S3 bucket

Test/QA

Unit-Test/QA

Unit Tests and QA test scripts can be integrated automatically and send results to the relevant teams.

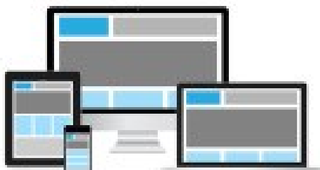
Staging

75%
30%
90%
60%

Staging Test

At this stage, release can be checked in test Stress environment , with endurance testing, load testing.

Deploy



Prod-Deploy

As this final stage, release can be deployed automatically in production server after successful verification.

DevOps – CHALLENGES of Continuous Delivery

- **Open-ended research projects:** Since such projects generally deal with new, experimental technologies that may not have a customer identified yet, production code is not meant for customer deployment. Team might still implement Continuous Integration, but may not have a system testing environment yet. CD can be done later in such cases.
- **Large legacy projects with low test automation:** Projects with high level of manual interventions (approval hierarchies, security checklists) built into their system and processes will find it difficult to migrate to the CD paradigm.
- **System architecture limitations:** Large monolithic architectures are difficult to automate, scale or debug. In contrast, products with modular and scalable architecture are conducive to CD. Impacted functions are easy to localise and manage every time there's a code update. Microservices, SOA, web applications are good examples of CD success stories.
- **Open-source collaborative development:** With several peer developers working in parallel, open-source platforms require an equally robust build configuration and test automation team. Otherwise, it becomes difficult to synchronise their work, making implementation of CD difficult.
-

DevOps – Continuous Deployment

- Continuous Delivery/Deployment (CD) is **a process by which an application is delivered to various environments, such as testing or production, once someone (usually a product owner/manager) decides that it is ready to go**.

- **Continuous Delivery** is the automation of steps to safely get changes into production. Where **Continuous Deployment focuses on the actual deployment, Continuous Delivery focuses on the release and release strategy**

- **Continuous deployment** (CD, or CDE) is a strategy or methodology for software releases where any new code update or change that makes it through the rigorous automated test process is deployed directly into the live production environment where it will be visible to customers.

DevOps – Continuous Deployment benefits

1. Maintain Capability for Quick New Releases

The most important feature of continuous deployment is that is enables developer teams to get their new releases into the production environment as quickly as possible. Most software companies can no longer rely on development methodologies that were common when developers released software updates once per year. Some companies are rolling out up to **10 deployments per day** with continuous deployment.

2. Enable a More Rapid Feedback Loop with Customers

More frequent updates to your application means a shorter feedback loop with the customer. Using state-of-the-art monitoring tools, developer teams can assess the impact of a new change on user behavior or engagement and make adjustments accordingly. The ability to rapidly release changes is an asset when customer behavior indicates the need for a quick pivot or change in strategy.

DevOps – Continuous Deployment benefits

3. Reducing Manual Processes with Automation

- Continuous deployment is defined by its use of automation in the application deployment process. In fact, continuous deployment wants developers to automate the entire software development process to the greatest extent possible, especially when it comes to release testing. Automation doesn't just help developers push out new releases faster, they actually spend less time on manual processes and get more work done.

- Continuous deployment also takes away release day pressure so developers' sole focus is on building the software, which automatically goes live in a matter of minutes.
- It also heightens productivity and allows developers to respond to rapidly shifting market demands.
- Continuous deployment, however, requires a high degree of monitoring and maintenance to ensure it continues to run smoothly as well as a great capacity to respond to changes quickly.

DevOps – Continuous Deployment vs Continuous Release - differences

difference between Continuous Delivery and Continuous Deployment:

A) Continuous Delivery is a manual task, while Continuous Deployment is an automated task

B) Continuous Delivery has a manual release to a production decision, while Continuous Deployment has releases automatically pushed to production.

C) Continuous Delivery includes all steps of the software development life cycle, Continuous deployment may skip a few steps such as validation and testing.

D) Continuous Delivery means complete delivery of the application to the customer, Continuous Deployment includes the only deployment of the application in a customer environment.

DevOps – Continuous Deployment vs Continuous Release

- a major difference between the two. **With continuous delivery, someone will need to approve the release of the feature to production but with continuous deployment, this process happens automatically upon passing automated testing.**
- Continuous deployment is basically when teams rely on a fully-automated pipeline. This practice fully eliminates any manual steps and automates the entire process. Therefore, continuous deployment ensures that code is continuously being pushed into production.
- Real-time monitoring, however, would still be needed to track and address any issues that arise during the automated tests and to ensure that the builds passed these tests.

DevOps – CI/CD and CD Process

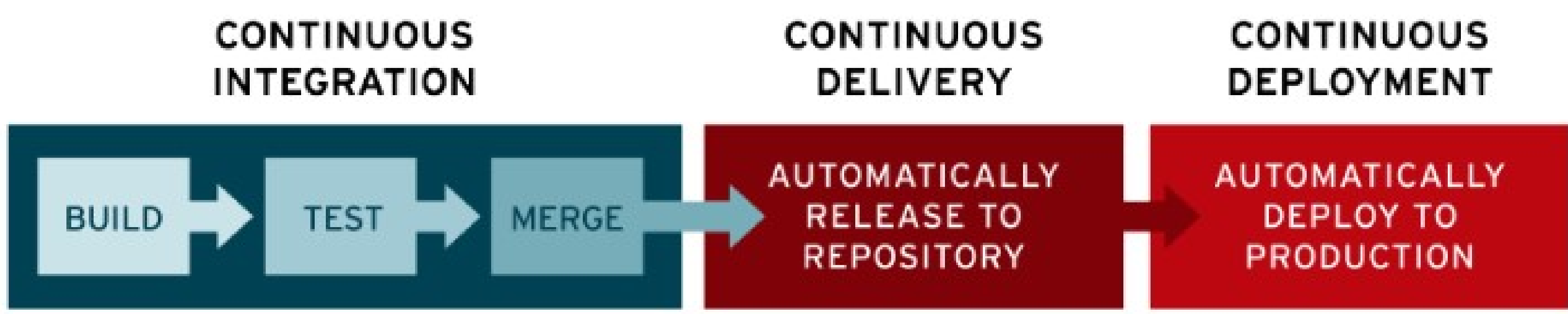
- Continuous deployment is part of a continuous process. The first step starts with continuous integration when developers merge their changes into the trunk or mainline on a frequent basis. This helps teams evade what is known as 'merge hell', which happens when developers attempt to merge several separate branches to the shared trunk on a less regular basis.
- **Continuous deployment goes one step further and combines continuous integration and continuous delivery to make software automatically available to users without any human intervention.**
- Hence, continuous deployment requires the complete automation of all deployments and so refers to the process of automatically releasing developers' changes from the repository to production, bypassing the need for developer approval for each release. Consequently, this process relies heavily on well-designed test automation.
- Then comes continuous delivery when code is deployed to a testing or production environment. Here, developers' changes are uploaded to a repository, where they are then deployed to a production environment. With continuous delivery, you can decide to release daily, weekly or monthly but it is usually recommended to release as often as possible in small batches to be able to easily and quickly fix any issue that arises.

DevOps – CI/CD

- **Continuous integration (CI):** integrate changes to a shared trunk several times a day.
- **Continuous delivery (CD):** continuous integration then deploy all code changes to production environment; deployment is manual.
- **Continuous deployment (CD):** step further from continuous delivery; automated deployment to production without any need for developer approval.

- CI/CD, visualized as a pipeline, automates the software delivery process by building code, running tests and deploying this code to a live production environment.

DevOps – ci/cd pipeline



DevOps – Continuous Monitoring

- **Continuous monitoring** in DevOps is the process of identifying threats to the security and compliance rules of a software development cycle and architecture. Also known as continuous control monitoring or CCM.
- This is one of the most crucial steps in a DevOps lifecycle and will help to achieve true efficiency and scalability.
- This is an automated procedure that can be extended to detect similar inconsistencies in IT infrastructures.
- Continuous monitoring helps business and technical teams determine and interpret analytics to solve crucial issues, as mentioned above, instantaneously.
- Continuous monitoring or CM is a step towards the end of the DevOps process. The software is usually sent for production before continuous monitoring is conducted.
- CM informs all relevant teams about the errors encountered during the production period. Once detected, these flaws are then looked into by the people concerned.
- DevOps tools for continuous monitoring include **Prometheus, Monit, Datadog, and Nagios.**
- Continuous control monitoring goes a long way to help enterprises acquire data from various ecosystems, which can then be used to take more robust security measures like **threat assessment, quick response to breaches, root cause analysis, and cyber forensics.**
- continuous monitoring keeps a tab and reports on the overall well-being of the DevOps setup
- https://www.browserstack.com/guide/continuous-monitoring-in-devops .

DevOps – Continuous Monitoring - Benefits

- **Better security:** Continuous monitoring can be utilized to automate many security measures. CM analyzes data across the entire ecosystem, giving backend teams a broad spectrum of visibility throughout the environment. This helps identify inconsistencies and trigger events that can lead to security failures.
- System admins can identify security threats and respond to them much faster due to continuous monitoring.
- **Quicker feedback and real-time reports** help security teams avert breach attempts and lessen the aftermath should an attack occur.

- **Performance errors are detected earlier:** Continuous monitoring is flexible in its entry into the software development cycle. **Although CM is traditionally introduced during the production phase, initiating it in staging and testing environments can help determine performance inconsistencies much ahead of time.** The production cycle can thus deal only with more stable releases.

- **System downtime is significantly reduced:** System downtimes are infamous for causing significant disruptions in business operations, where recurring incidents can lead to a loss in revenue.
- Continuous monitoring can assist technical teams to keep a vigilant eye on the **database, network, and applications**, helping to resolve any issues before they give rise to system downtime. Past issues are also evaluated to avoid them in the future and to build more enhanced software solutions.

DevOps – Continuous Monitoring - Benefits

- **Facilitates better performance for the business:** Since continuous monitoring works to alleviate system downtimes, it improves **user experience** **and business credibility**.
- continuous monitoring is used extensively to track user feedback that comes in handy when evaluating new updates and changes to the system.

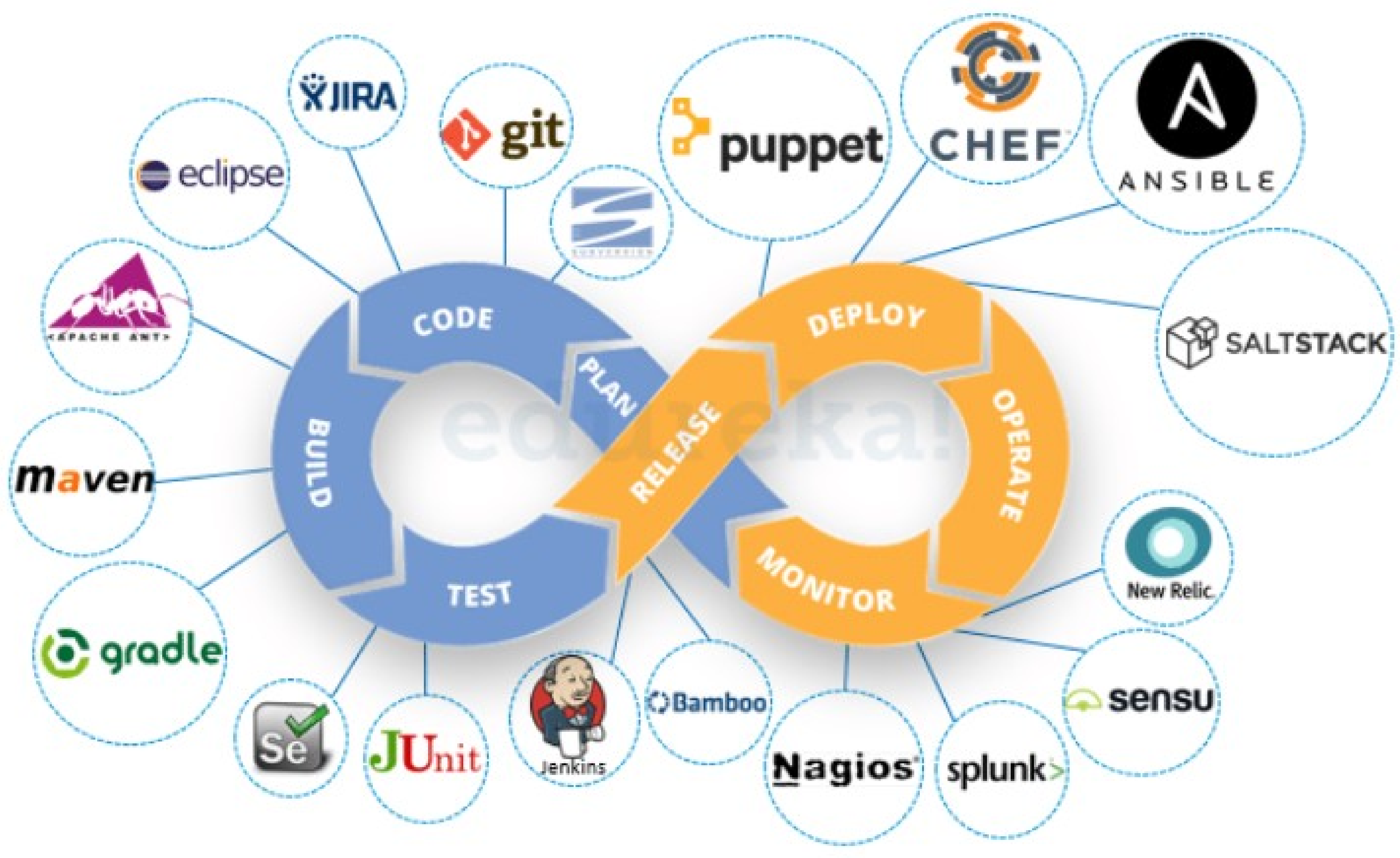
DevOps – Continuous Monitoring - Types

- **Infrastructure Monitoring**: Monitors and manages the IT infrastructure required to deliver products and services. This includes data centers, networks, hardware, software, servers, storage, and the like. Infrastructure Monitoring collates and examines data from the IT ecosystem to improve product performance as far as possible.
- **Tool must monitor:** Server Availability, CPU & Disk Usage, Server & System Uptime, Response Time to Errors, Storage, Database Health, Storage, Security, User permissions, Network switches, Process level usage, Relevant performance trends
- **Tools:** Nagios, Zabbix, ManageEngine OpManager, Solarwinds,Prometheus etc
- **Application Monitoring**: Monitors the performance of released software based on metrics like uptime, transaction time and volume, system responses, API responses, and general stability of the back-end and front-end.
- **Tool must monitor:** availability, error rate, throughput, user response time, pages with low load speed, third-party resource speed, browser speed, end-user transactions, SLA status.

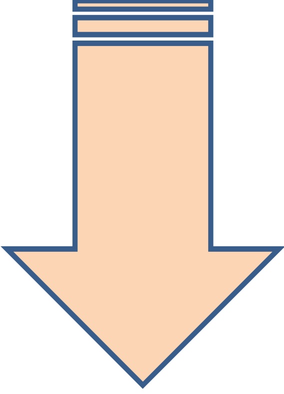
  Tools: Appdynamics, Dynatrace, Datadog, Uptime Robot,Uptrends, Splunk etc
- **Network Monitoring:** Monitors and tracks network activity, including the status and functioning of firewalls, routers, switches, servers, Virtual Machines, etc. Network Monitoring detects possible and present issues and alerts the relevant personnel. Its primary goal is to prevent network downtime and crashes.

  Tool: Cacti, ntop,nmap, Spiceworks, Wireshark,Traceroute, bandwidth Monitor etc.
- **Tool must monitor:** Latency, Multiple port level metrics, Server bandwidth, CPU use of hosts, Network packets flow .

DevOps tools



Version control with Git: Git basics, Git features, installing Git, Git essentials, common commands in Git, Working with remote repositories.



Git

- https://www.javatpoint.com/git
- https://www.javatpoint.com/github
- https://www.geeksforgeeks.org/version-control-systems/

Git

- **Git** is an **open-source distributed version control system**.
- It was created by **Linus Torvalds in 2005**.
- It is used for:
  - Tracking code changes
  - Tracking who made changes
  - Coding collaboration
- It is developed to manage projects with high speed and efficiency.
- The version control system allows us to monitor and work together with our team members at the same workspace.
- Git can be used with Windows, Linux, Mac
- It is designed to handle minor to major projects with high speed and efficiency. It is developed to co-ordinate the work among the developers. The version control allows us to track and work together with our team members at the same workspace.
- Git is foundation of many services like **GitHub** and **GitLab**, but we can use Git without using any other Git services.
- Git can be used **privately** and **publicly**. it is easy to learn, and has fast performance.
- It is superior to other **SCM (Source Code Management)** tools like Subversion, **CVS (Concurrent Versions System**), Perforce, and ClearCase.
- Other distributed version control system is **Mercurial**
- **VCS examples are Git, Helix core, Microsoft TFS,**

Git - features

Features:

- ✓ **Open Source :**
- Git is an **open-source tool**. It is released under the **GPL** (General Public License) license.
- ✓ **Scalability**
- Git is **scalable**, which means when the number of users increases, the Git can easily handle such situations.
- ✓ **Distributed**
- One of Git's great features is that it is **distributed**. Distributed means that instead of switching the project to another machine, we can create a "clone" of the entire repository. Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project. We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can **push these changes to a remote repository.**
- ✓ **Security:** Git is secure. It uses the **SHA1 (Secure Hash Function)** to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout. It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.
- ✓ **Speed:** Git is very **fast**, so it can complete all the tasks in a while. Most of the git operations are done on the local repository, so it provides a **huge speed**. Also, a centralized version control system continually communicates with a server somewhere.
- ✓ **core part of Git** is **written in C**

Git - features

Features:

- ✓ **Branching and Merging:**

  Branching and merging are the **great feature**s of Git, which makes it different from the other SCM tools. Git allows the **creation of multiple branches** without affecting each other. We can perform tasks like **creation**, **deletion**, and **merging** on branches, and these tasks take a few seconds only.

- We can **create a separate branch** for a new module of the project, commit and delete it whenever we want.
- We can have a **production branch**, which always has what goes into production and can be merged for testing in the test branch.
- We can create a **demo branch** for the experiment and check if it is working. We can also remove it if needed.
- The core benefit of branching is if we want to push something to a remote repository, we do not have to push all of our branches. We can select a few of our branches, or all of them together.

- ✓ **Data Assurance**: The Git data model ensures the cryptographic integrity of every unit of our project. It provides a unique commit ID to every commit through a SHA algorithm. We can retrieve and update the commit by commit ID. Most of the centralized version control systems do not provide such integrity by default
- ✓ **Staging Area:** The **Staging area** is also a **unique functionality** of Git. It can be considered as a **preview of our next commit**, moreover, an **intermediate area** where commits can be formatted and reviewed before completion. When you make a commit, Git takes changes that are in the staging area and make them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the changes.

Git - features

Features:

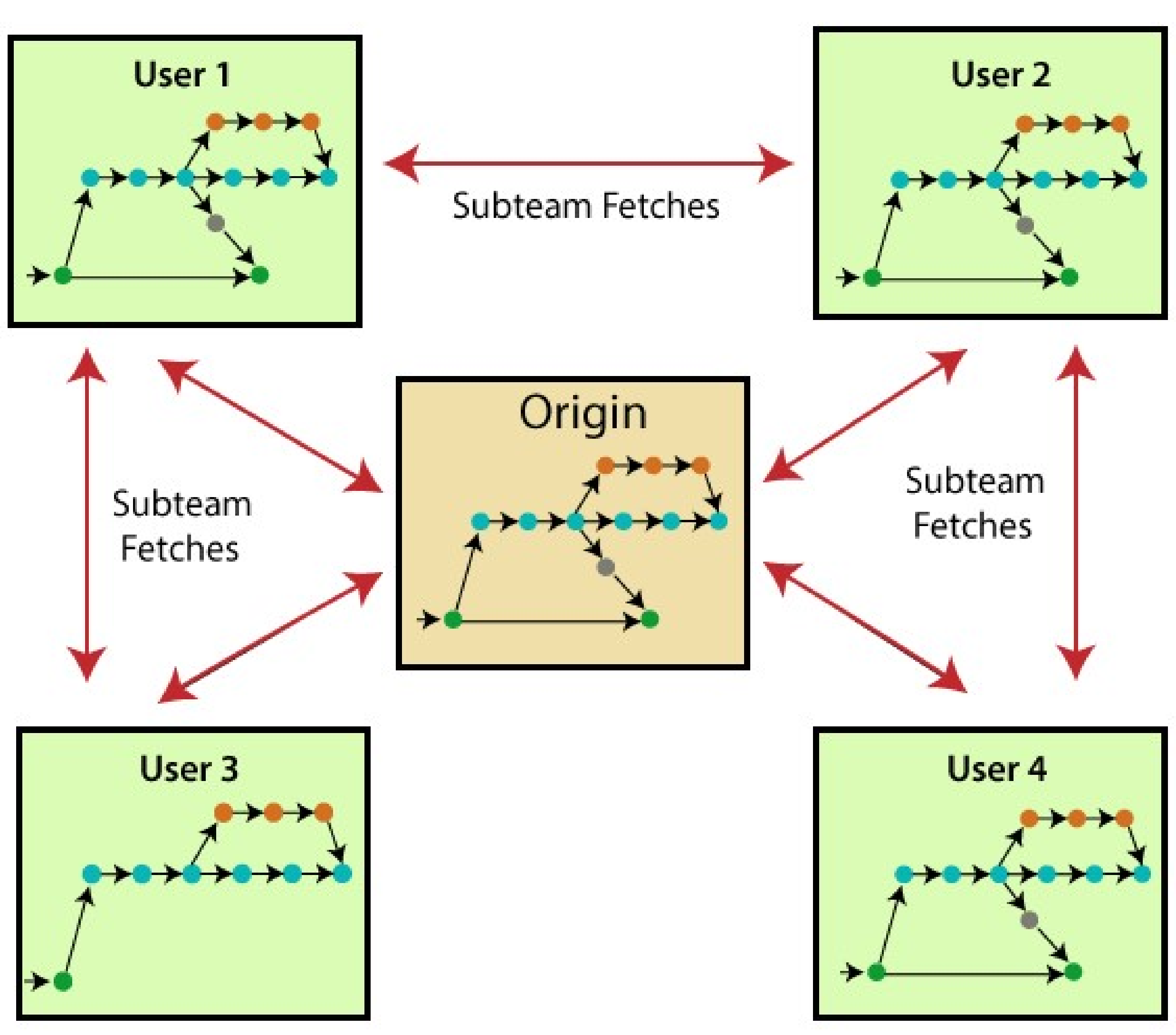
- ✓ **Maintain the clean history:** Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the master branch and puts our code on top of that. Thus, it maintains a clean history of the project.

Benefits of Git:

- A version control application allows us to **keep track** of all the changes that we make in the files of our project. Every time we make changes in files of an existing project, we can push those changes to a repository. Other developers are allowed to pull your changes from the repository and continue to work with the updates that you added to the project files.

Git

Distributed System:

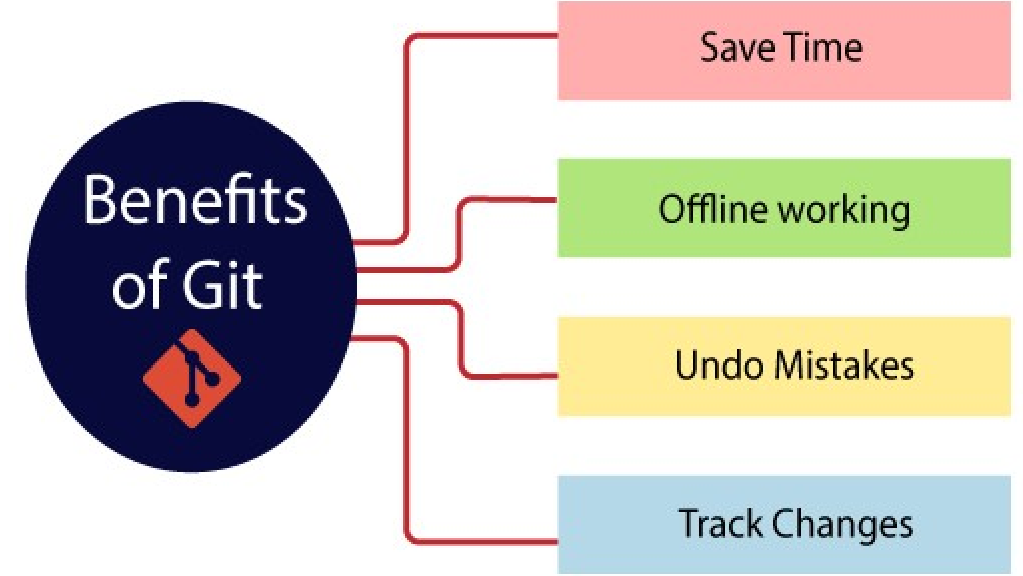


Git - Benefits

Benefits of Git:

- A version control application allows us to **keep track** of all the changes that we make in the files of our project. Every time we make changes in files of an existing project, we can push those changes to a repository. Other developers are allowed to pull your changes from the repository and continue to work with the updates that you added to the project files.

1. Save Time
2. Offline Working
3. Undo mistakes
4. Track Changes

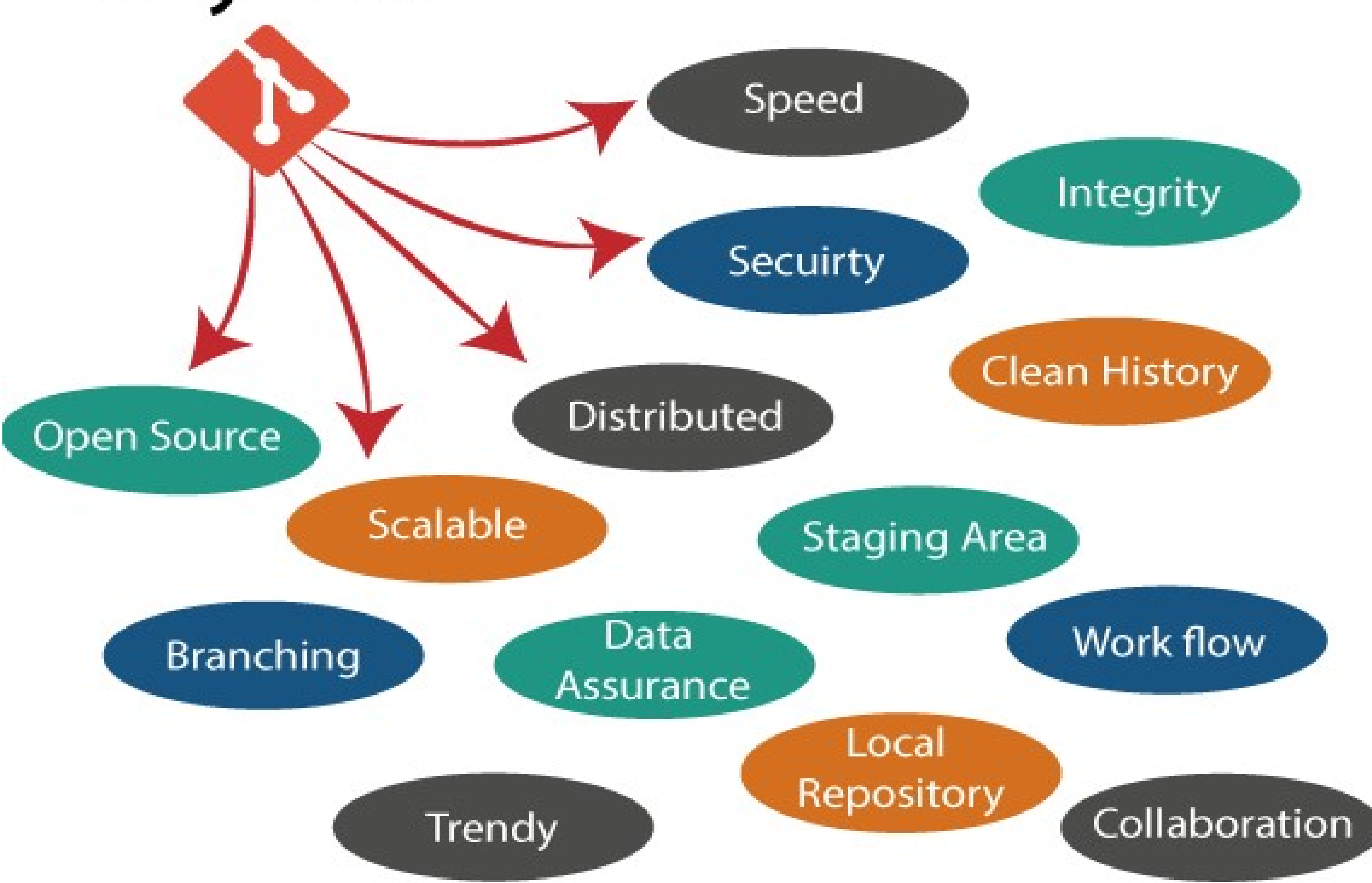


Git - Benefits

Benefits of Git:

- **Saves Time**
  Git is lightning fast technology. Each command takes only a few seconds to execute so we can save a lot of time as compared to login to a GitHub account and find out its features.
- **Offline Working**
  One of the most important benefits of Git is that it supports **offline working**. If we are facing internet connectivity issues, it will not affect our work. In Git, we can do almost everything locally. Comparatively, other CVS(Concurrent Version System) like SVN (Sub version) is limited and prefer the connection with the central repository.
- **Undo Mistakes**
  One additional benefit of Git is we can **Undo** mistakes. Sometimes the undo can be a good option for us. Git provides the undo option for almost everything.
- **Track the Changes**
  Git facilitates with some exciting features such as **Diff, Log,** and **Status**, which allows us to track changes so we can **check the status, compare** our files or branches.

Why Git



Why Git contd..

- **Git Integrity**
  Git is **developed to ensure** the **security** and **integrity** of content being version controlled. It uses checksum during transit or tampering with the file system to confirm that information is not lost. Internally it creates a checksum value from the contents of the file and then verifies it when transmitting or storing data.
- **Trendy Version Control System**
  Git is the **most widely used version control system**. It has **maximum projects** among all the version control systems. Due to its **amazing workflow** and features, it is a preferred choice of developers.
- **Everything is Local**
  Almost All operations of Git can be performed locally; this is a significant reason for the use of Git. We will not have to ensure internet connectivity.
- **Collaborate to Public Projects**
  There are many public projects available on the GitHub. We can collaborate on those projects and show our creativity to the world. Many developers are collaborating on public projects. The collaboration allows us to stand with experienced developers and learn a lot from them; thus, it takes our programming skills to the next level.
- **Impress Recruiters**
  We can impress recruiters by mentioning the Git and GitHub on our resume. Send your GitHub profile link to the HR of the organization you want to join. Show your skills and influence them through your work. It increases the chances of getting hired.

VCS Types

Types of Version Control Systems:

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

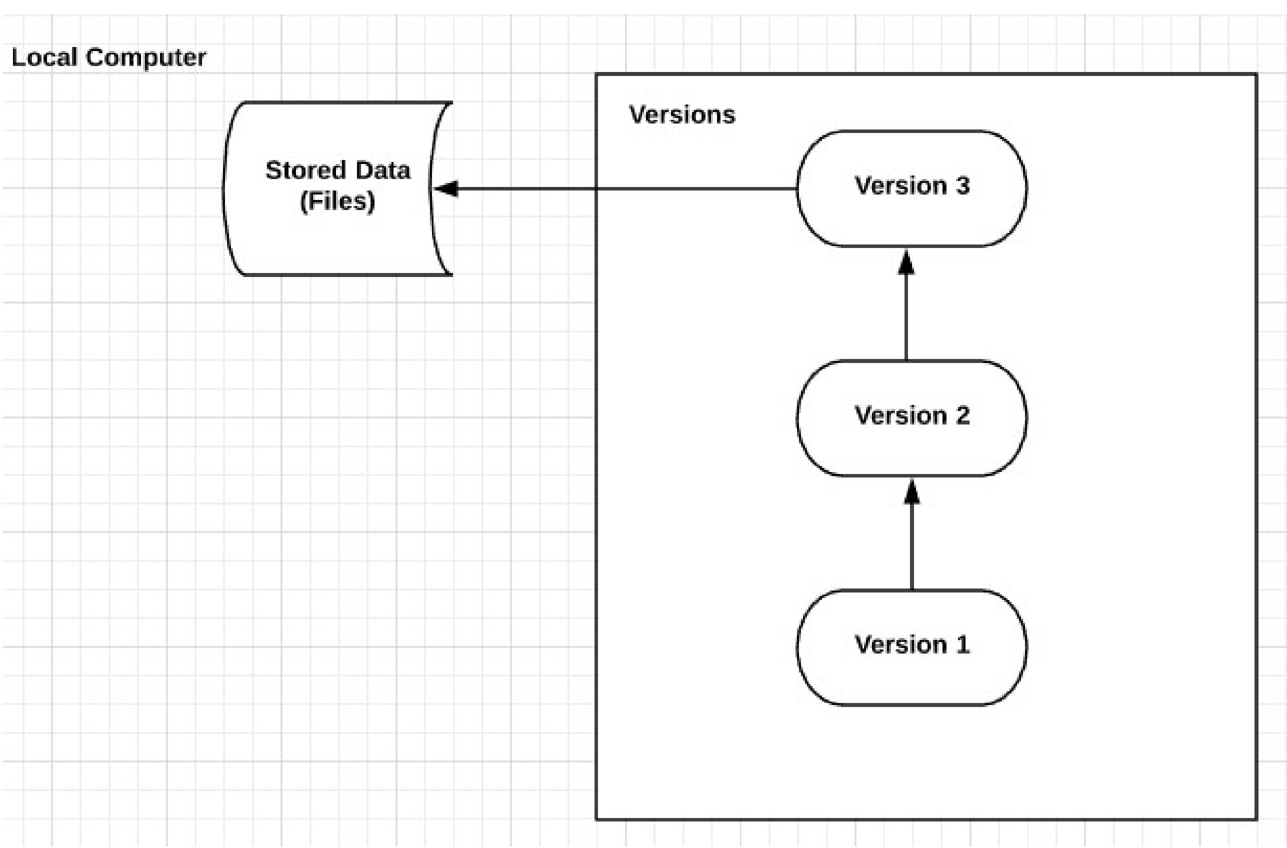
Local Version Control Systems:

- **Local version control system maintains track of files within the local system.**
- This approach is very common and simple. This type is also error prone which means the chances of accidentally writing to the wrong file is higher.
- 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.

VCS Types

Types of Version Control Systems:

- Local Version Control Systems



VCS Types

Centralized Version Control Systems:

- With centralized version control systems, you have a single "central" copy of your project on a server and commit your changes to this central copy. You pull the files that you need, but **you never have a full copy of your project locally.**
- 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.
  - when working in larger teams, the central server and repository can become a bottleneck.
- Examples are **Subversion (SVN) and Perforce, Tortoise SVN, Concurrent Version Systems (CVS)**

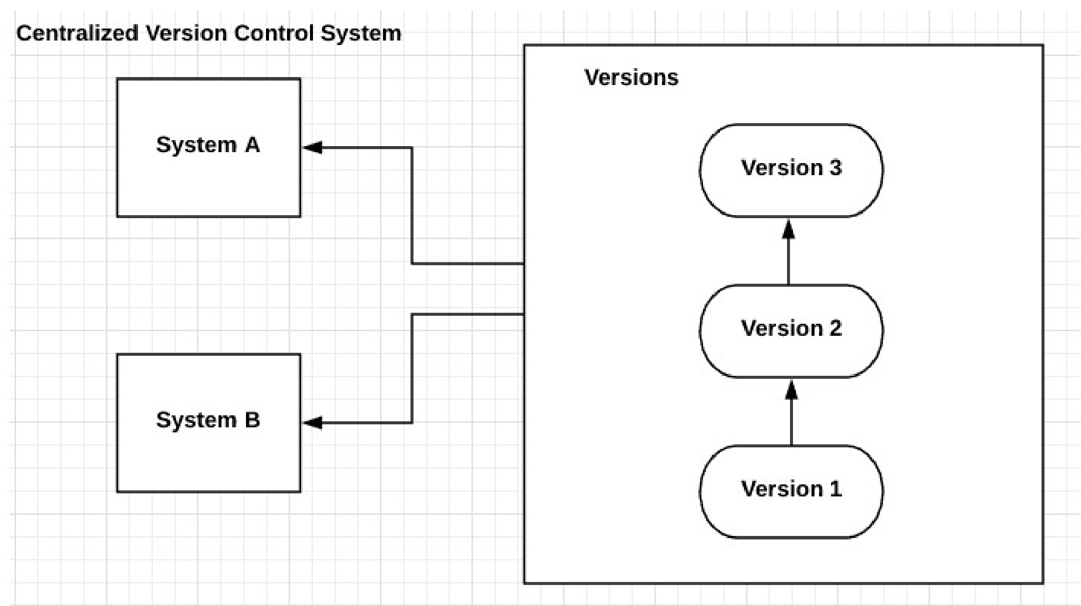
VCS Types – Centralized

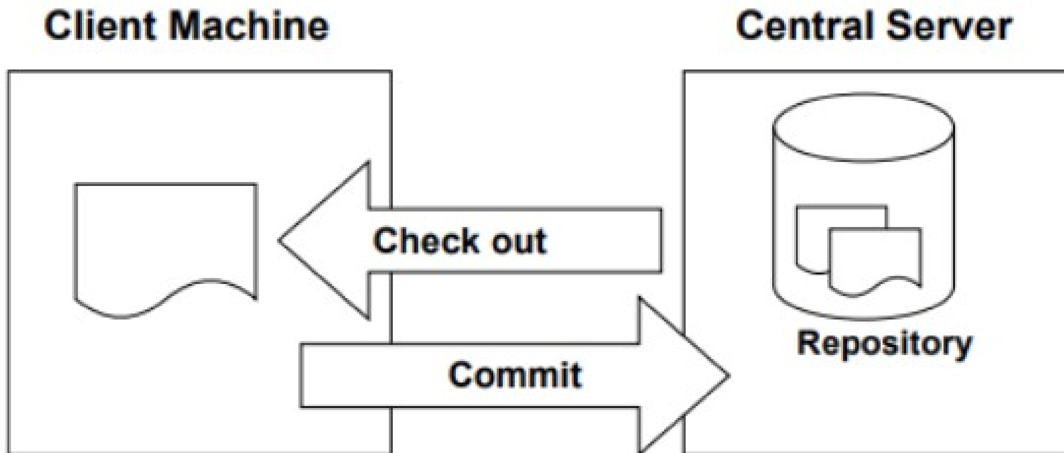
- In centralized source control**, there is a server and a client.** The server is the master repository that contains all of the versions of the code.
- To work on any project, firstly user or client needs to get the code from the master repository or server. So the client communicates with the server and **pulls all the code or current version of the code from the server to their local machine.** In other terms we can say, you need to take an update from the master repository and then you get the local copy of the code in your system.
- So once you get the latest version of the code, you start making your own changes in the code and after that, you simply **need to commit those changes straight forward into the master repository.** Committing a change simply means merging your own code into the master repository or making a new version of the source code. So everything is centralized in this model.
- **There will be just one repository and that will contain all the history or version of the code and different branches of the code.**

Working:

- So the basic workflow involves in the centralized source control is getting the latest version of the code from a central repository that will contain other people's code as well, making your own changes in the code, and then committing or merging those changes into the central repository.

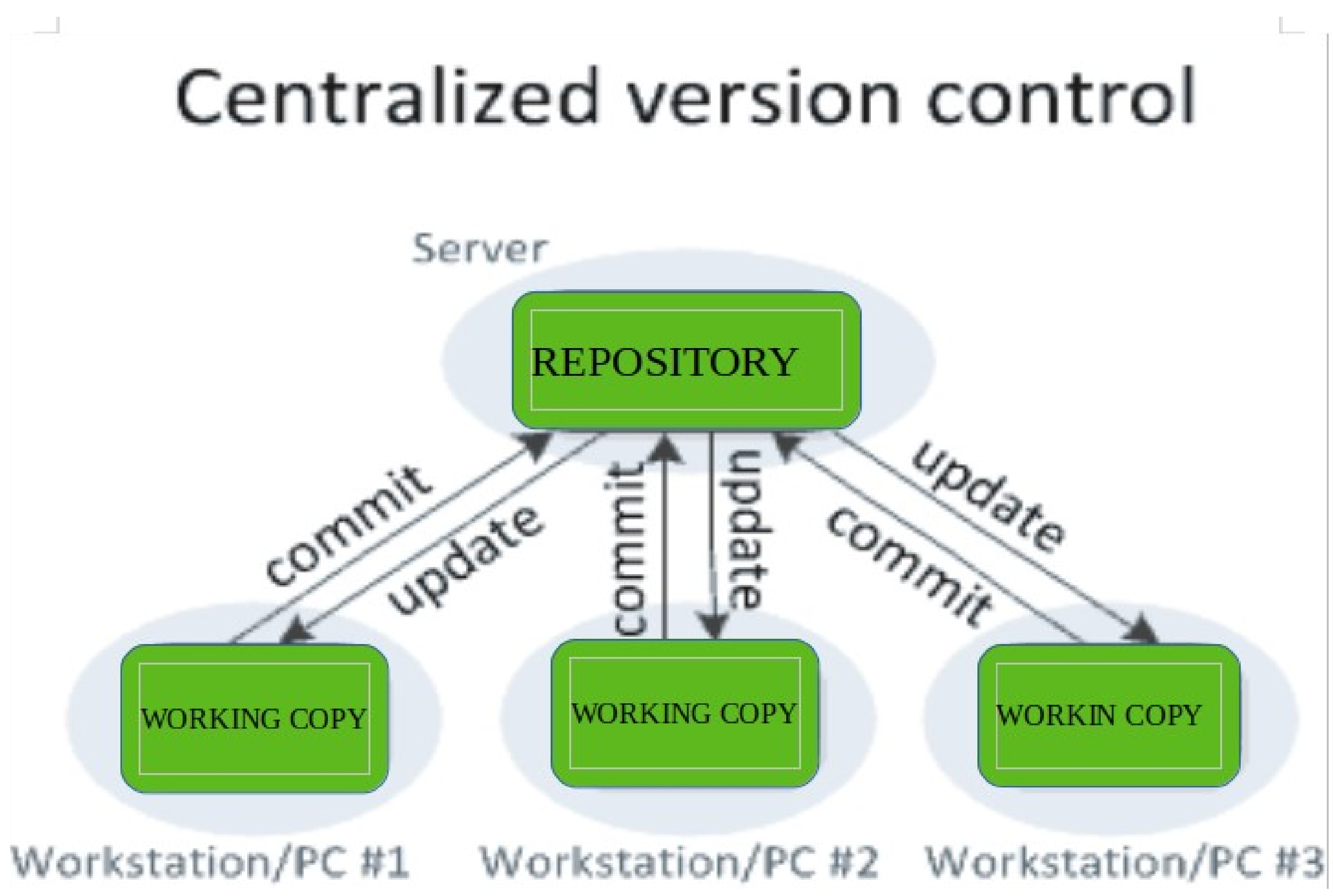
VCS Types – Centralized





VCS Types

Centralized Version Control Systems:



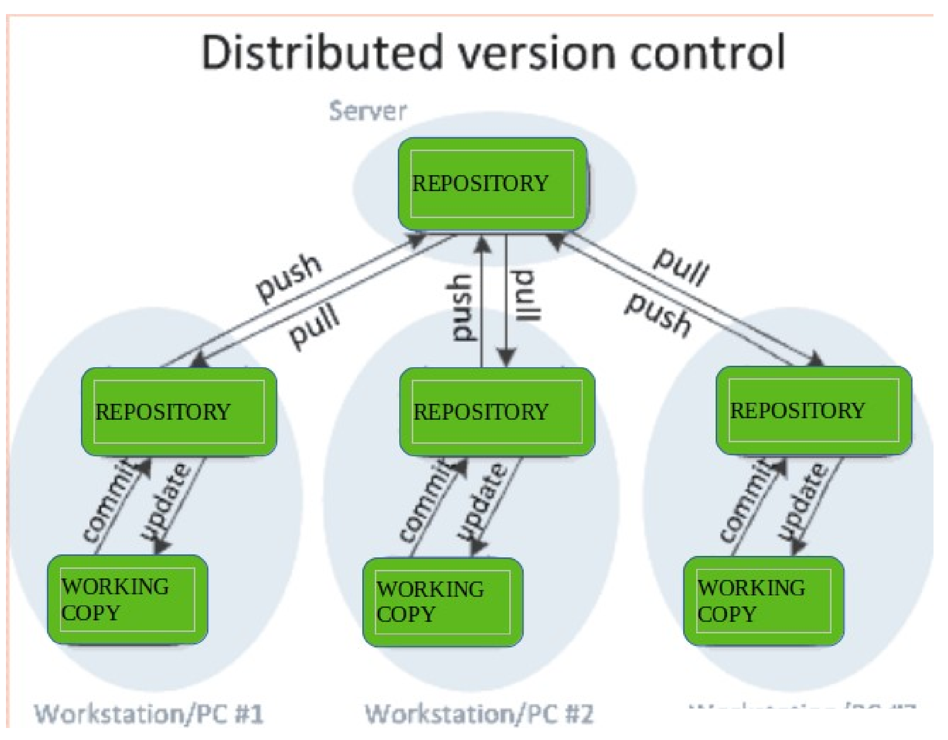
VCS Types - Distributed Version Control Systems

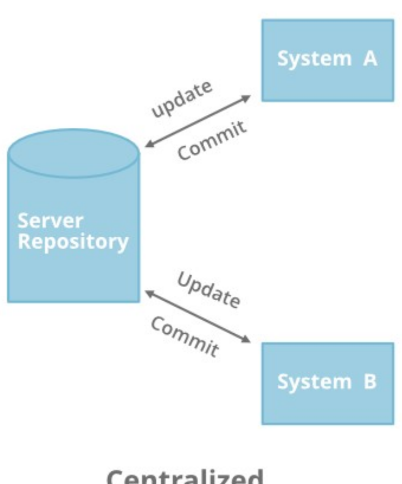
- A distributed version control system (DVCS) is a type of version control where the complete codebase — including its full version history — is mirrored on every developer's computer.
- Distributed version control systems contain multiple repositories. Each user has their own repository and working 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.
- **The clients completely clone the repository including its full history.**
- To make your changes visible to others, 4 things are required:
  1. You commit
  2. You push
  3. They pull
  4. They update
- **The most popular distributed version control systems are Git, and Mercurial**.
- Git is open source software distributed under the terms of the GNU (General Public License).

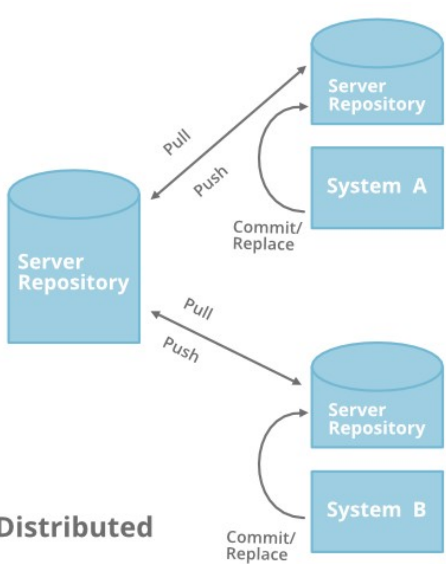
Advantages:

- Branching and merging can happen automatically and quickly
- Developers have the ability to work offline
- Multiple copies of the software eliminate reliance on a single backup

VCS Types - Distributed Version Control Systems



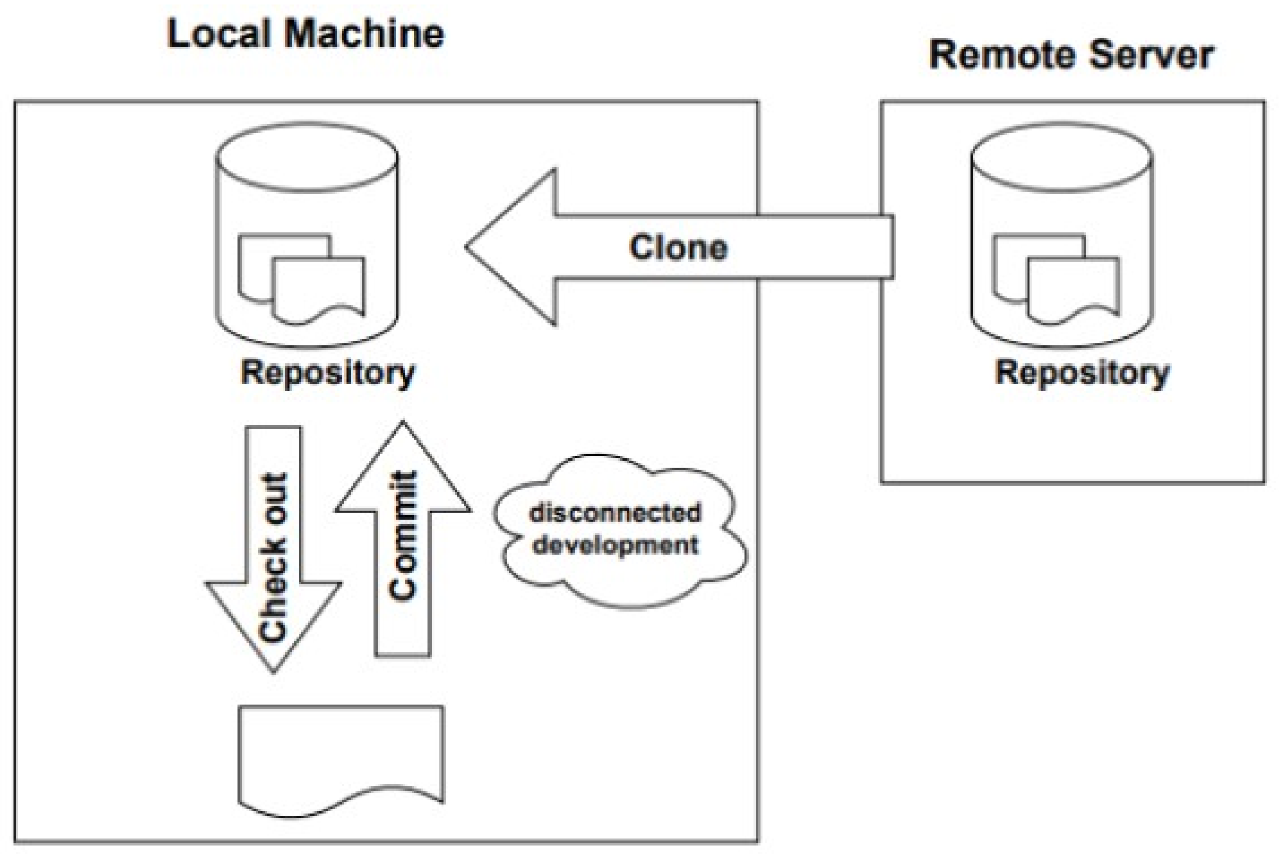




VCS Types - Distributed Version Control Systems

- In distributed version control most of the mechanism or model applies the same as centralized. The only major difference you will find here is**, instead of one single repository which is the server, here every single developer or client has their own server and they will have a copy of the entire history or version of the code and all of its branches in their local server or machine.**
- Basically, every client or **user can work locally and disconnected** which is more convenient than centralized source control and that's why it is called distributed.
- You don't need to rely on the central server, you can clone the entire history or copy of the code to your hard drive. So when you start working on a project, you clone the code from the master repository in your own hard drive, then you get the code from your own repository to make changes and after doing changes, you commit your changes to your local repository and at this point, your local repository will have '*change sets*' but it is still disconnected with the master repository (master repository will have different '***sets of changes***' from each and every individual developer's repository), so to communicate with it, you issue a request to the master repository and push your local repository code to the master repository. Getting the new change from a repository is called "**pulling**" and merging your local repository's 'set of changes' is called "**pushing**".
- It doesn't follow the way of communicating or merging the code straight forward to the master repository after making changes.
- **Firstly you commit all the changes in your own server or repository and then the 'set of changes' will merge to the master repository.**

VCS Types - Distributed Version Control Systems



VCS Types – centralized vs distributed vcs

- **Centralized version control is easier to learn than distributed**. If you are a beginner you'll have to remember all the commands for all the operations in DVCS and working on DVCS might be confusing initially. **CVCS is easy to learn and easy to set up**.
- **DVCS** has the biggest advantage in that **it allows you to work offline and gives flexibility**. You have the entire history of the code in your own hard drive, so all the changes you will be making in your own server or to your own repository which doesn't require an internet connection, but this is not in the case of CVCS.
- **DVCS is faster than CVCS** because you don't need to communicate with the remote server for each and every command. **You do everything locally which gives you the benefit to work faster than CVCS.**
- **Working on branches is easy in DVCS.** Every developer has an entire history of the code in DVCS, so developers can share their changes before merging all the 'sets of changes to the remote server. In CVCS it's difficult and time-consuming to work on branches because it requires to communicate with the server directly.
- If the project has a long history or the project contain large binary files, in that case**,** **downloading the entire project in DVCS can take more time and space than usual**, whereas in CVCS you just need to get few lines of code because you don't need to save the entire history or complete project in your own server so there is no requirement for additional space.

VCS Types – centralized vs distributed vcs

- **If the main server goes down or it crashes in DVCS, you can still get the backup** or entire history of the code from your local repository or server where the full revision of the code is already saved. This is not in the case of CVCS, there is just a single remote server that has entire code history.
- **Merge conflicts with other developer's code are less in DVCS**. Because every developer work on their own piece of code. **Merge conflicts are more in CVCS** in comparison to DVCS.
- **In DVCS, sometimes developers take the advantage of having the entire history of the code and they may work for too long in isolation** which is not a good thing. This is not in the case of CVCS.

VCS Types – centralized vs distributed vcs

Sl. No.	Centralized VCS	Distributed VCS
1	Centralized VCS is the simplest form of version control in which the central repository of the server provides the latest code to the clients.	Distributed vcs is a form of vcs where the complete codebase (including the history) is mirrored on every developer's computer.
2	There are no local repositories	There are local repositories
3	Works comparatively slower	Works faster
4	Always require internet connectivity	Developers can work with a local repository without an internet connection
5	Considers the entire columns for compression.	Considers columns as well as partial columns.
6	Focuses on synchronizing, tracking and backing up files	Focuses on sharing changes
7	A failure in the central server terminates all the versions	A failure in the main server does not affect the development

VCS Types – centralized vs distributed vcs

Definition

- Centralized version control is the simplest form of version control in which the central repository of the server provides the latest code to the client machines. Distributed version control, on the other hand, is a form of version control where the complete codebase (including its full history) is mirrored on every developer's computer. Thus, this is the main difference between centralized and distributed version control.

Local Repositories

- In centralized version control, there are no local repositories; however, in distributed version control, there are local repositories. Hence, this is also an important difference between centralized and distributed version control.

Speed

- Furthermore, distributed version control works faster than centralized version control.

Internet Connectivity

- Moreover, centralized version controlling always require internet connectivity while developers in distributed version control can work with a local repository without an internet connection.

Main Focus

- Also, one other difference between centralized and distributed version control is their **focus**. Centralized version control focuses on synchronizing, tracking, and backing up files, while distributed version control focuses on sharing changes.

Failures

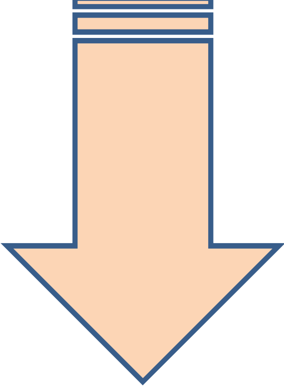
- In centralized version control, a failure in the central server terminates all the versions, while in distributed version control, a failure in the main server does not affect the development. Thus, this is another important difference between centralized and distributed version control

VCS Types – centralized vs distributed vcs

- https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems/?ref=lbp
- https://www.w3schools.com/git/
- https://www.tutorialspoint.com/git/git_basic_concepts.htm
-

Git

Git basics, Git features, installing Git, Git essentials, common commands in Git, Working with remote repositories.



Git basics

- https://git-scm.com/
- Git is a version control system that manages and keeps track of your code. GitHub, on the other hand, is a service that let you host, share, and manage your code files on the internet.
- Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.
- Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.
- Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.
- **It is used for:**
  - Tracking code changes
  - Tracking who made changes
  - Coding collaboration

What does Git do?

- Manage projects with Repositories
- Clone a project to work on a **local copy**
- Control and track changes with **Staging and Committing**
- **Branch** and Merge to allow for work on different parts and versions of a project
- **Pull** the latest version of the project to a local copy
- **Push** local updates to the main project

Git basics

Why Git?

- Over 70% of developers use Git!
- Developers can work together from anywhere in the world.
- Developers can see the full history of the project.
- Developers can revert to earlier versions of a project.
- We can use **‘git bash’** command shell in Windows to use Git.

Git Bash commands:

To check the installation version of the Git:

$ git --version

git version 2.37.1.windows.1

Git Installation

- to install it. You can get it a number of ways; the two major ones are **to install it from source** or **to install an existing package for your platform.**
- To install Git, you need to have the following libraries that Git depends on: **curl, zlib, openssl, expat, and libiconv.**
- For example, if you're on a system that has **yum** (such as Fedora) or **apt-get** (such as a Debian based system), you can use one of these commands to install all of the dependencies:
- **yum install curl-devel expat-devel gettext-devel **
- openssl-devel zlib-devel
- $ **apt-get install libcurl4-gnutls-dev libexpat1-dev gettext **

  libz-dev libssl-dev

To Install on Windows:

- Installing Git on Windows is very easy. The msysGit project has one of the easier installation procedures. Simply download the installer exe file from the GitHub page, and run it:

  http:**//msysgit.github.io**
- After it's installed, you have both a command-line version (including an **SSH client** that will come in handy later) and the **standard GUI.**

GitHub.com

- **GitHub.com** makes storing Git repositories and sharing code simple. Once published to GitHub, the process of sharing Git repositories, facilitating code review, and contributing content changes becomes simpler than ever.

Git commands

- Git config command
- Git init command
- Git clone command
- Git add command
- Git commit command
- Git status command
- **Git push Command**
- **Git pull command**
- **Git Branch Command**
- **Git Merge Command**
- **Git remote command**
- Git log command

Git basics

Git Configuration:

Step 1:Create folder for Git:

$mkdir myproject

A folder will be created in c:/users/bh/ with myproject name

$cd myproject

Step 2: Initialize git: initialize the project directory as a Git repository.

$git init

```
git init
git add .
git commit –m "The first commit"
```

Step 3: Username password setting: (User Identity setting):

git config --global user.name "w3schools-test"

git config --global user.email test@w3schools.com

Ex: git config user.name "kbhaskar"

git config user.email kbhaskar511@gmail.com

git config –global user.name

→ displays username

--global specifies username and password is for all repositories

Git basics

$git status:

To check the status of git repo (gives tracked and untracked files details)

- The ***git status*** command is used to display the state of the working directory and the staging area. It allows you to see which changes have been staged, which haven't, and which files arenot being tracked by Git.
- **It does not show you any information about the committed project history.** For this, you need to use the git log. It also lists the files that you've changed and those you still need to add or commit.

Output:

```
On branch master
No commits yet
Untracked files:
    (use "git add <file>..." to include in what will be committed)
    sample.html
nothing added to commit but untracked files present (use "git add" to track)
```

git Repo folder files status:

- Files in your Git repository folder can be in one of 2 states:
    1. **Tracked -** files that Git knows about and are added to the repository
    2. **Untracked -** files that are in your working directory, but not added to the repository

✓ When you first add files to an empty repository, they are all untracked. To get Git to track them, you need to stage them, or add them to the staging environment.

Git basics

Remote Origin:

In Git, a remote origin refers to the remote repository where your local repository's code will be hosted. This remote repository could be on platforms like GitHub, GitLab, Bitbucket, or a private server. Adding a remote origin establishes a connection between your local repository and the remote repository, provides the exchange of code changes.

git remote add origin <HTTPS_URL>

git remote add origin https://github.com/kbhaskarcs/meet2022_new.git

git remote set-url origin git@ https://github.com/kbhaskarcs/meet2022_new.git

git remote -v

show the remote URLs associated with your repository.

git push -u origin main

pushes the code to GitHub. The -u flag creates a tracking reference for the branch, and origin main puts the code in the main branch.

Git basics

Git add command:

- This command is used to add one or more files to staging (Index) area.
- To add one file

  $ git add Filename
- To add more than one file

  $ git add*

Git init command:

The git init command **creates a new Git repository**. It can be used to convert an existing, unversioned project to a Git repository or initialize a new, empty repository.

Git basics

Commit changes: To Commit all the files that have been added, along with a commit message:

git commit -m "Initial commit"

- This creates a new commit with the given message. A commit is like a save or snapshot of your entire project.
- **Git commit -m <msg>**

  Use the given <msg> as the commit message. If multiple -m options are given, their values are concatenated as separate paragraphs.

Ex1: $ git commit -m "first commit“ → immediately creates a commit with a passed commit message

```
bh@mypc MINGW64 ~/repo_ex1 (master)
$ git commit -a -m  "first commit"
[master (root-commit) de08805] first commit
 1 file changed, 12 insertions(+)
 create mode 100644 sample.html
```

Ex2:

git commit –a → Tell the command to automatically stage files that have been modified and deleted, but new files you have not told Git about are not affected.

Git basics

git clone:

- This command is used to make a copy of a repository from an existing URL. If I want a local copy of my repository from GitHub, this command allows creating a local copy of that repository on your local directory from the repository URL.

- The **git clone** command is used to copy an existing Git repository from a server to the local machine.

Syntax: $git clone <URL>

- **Ex:** $ git clone https://github.com/kbhaskararao/DevOps.git

  Cloning into 'DevOps'...

  remote: Enumerating objects: 16, done.

  remote: Counting objects: 100% (16/16), done.

  remote: Compressing objects: 100% (13/13), done.

  remote: Total 16 (delta 1), reused 0 (delta 0), pack-reused 0

  Receiving objects: 100% (16/16), 16.18 MiB | 3.48 MiB/s, done.

  Resolving deltas: 100% (1/1), done.

Ex:

- **cd <path where you'd like the clone to create a directory>**
- git clone https://github.com/username/projectname.git
- This creates a directory called projectname on the local machine, containing all the files in the remote Git repository. This includes **source files for the project**, as well as a .git sub-directory which contains the entire **history and configuration for the project.**
- To declare that identity for *all repositories, use* ***git config –global***

To get command help:

git config -help → *gives help regarding 'git config' command.*

Ex 2:

$git clone

Git basics

git checkout master

git add .

git commit -m "my changes"

git push origin master

Where:

origin is the default name (alias) of Your remote repository and

master is the remote branch You want to push Your changes to.

git remote –v → check the remote origin

git remote remove origin →

git remote add origin git@github.com:kbhaskararao/DevOps.git

- When developers want to take their local Git repository and share it with another developer or push the code into a cloud-based distributed version control service, such as GitHub or GitLab, they can use the git remote add origin command.

Git reflog (reference logs) command:

bh@mypc MINGW64 ~/DevOps (master)

$ git reflog

0d46ec1 (**HEAD -> master) HEAD@{0}: commit: first changes**

9a857aa HEAD@{1}: initial pull

bh@mypc MINGW64 ~/DevOps (master)

- $

Git basics

Commiting changes to Remote Repo:

- To share your code you create a repository on a remote server to which you will copy your local repository.

On the remote server:

- **git init --bare /path/to/repo.git**
-

On the local machine:

- **git remote add origin ssh://username@server:/path/to/repo.git**
- (Note that ssh: is just one possible way of accessing the remote repository.)

Copying local repository to the remote:

On the remote server:

git init --bare /path/to/repo.git

On the local machine:

git remote add origin ssh://username@server:/path/to/repo.git

Ex:

git remote add origin http://**cameronmcnz:T55tutorial@**github.com/cameronmcnz/my-github-repo.git

git push --set-upstream origin master

Setting your user name and email:

- You need to **set who you are *before* creating any commit. That will allow commits to have**
- **git config --global user.name "Your Name"**
- **git config --global user.email mail@example.com**

Git basics

Viewing the history:

Git is able to show the history of the repository.

$git log

$git log --all --graph --decorate –oneline → gives oneline output

Setting editor:

$git config --global core.editor notepad++

Cheking the settings:

$git config –list

$ Git config -global color.ui true

Color.ui:

$ Git config -global color.ui true

The default value of color.ui is set as auto, which will apply colors to the immediate terminal output stream. You can set the color value as true, false, auto, and always.

Git basics

Git configuration levels:

- The git config command can accept arguments to specify the configuration level. The following configuration levels are available in the Git config.
  - local
  - global
  - system
- **--local**
- It is the default level in Git. Git config will write to a local level if no configuration option is given. Local configuration values are stored in **.git/config** directory as a file.
- **--global**
- The global level configuration is **user-specific configuration**. User-specific means, it is applied to an individual operating system user. Global configuration values are stored in a user's home directory. **~ /.gitconfig** on UNIX systems and **C:\Users\\.gitconfig** on windows as a file format.
- **--system**
- The system-level configuration is applied across an entire system. The entire system means **all users on an operating system and all repositories**. The system-level configuration file stores in a **gitconfig** file off the system directory. **$(prefix)/etc/gitconfig** on UNIX systems and **C:\ProgramData\Git\config** on Windows.

Git basics

There are 3 levels of git config; project, global and system.

project: Project configs are only available for the **current project** and stored in .git/config in the project's directory.

global: Global configs are available **for all projects for the current user** and stored in ~/.gitconfig.

system: System configs are available **for all the users/projects** and stored in /etc/gitconfig.

Create a project specific config, you have to execute this under the project's directory.

$ git config user.name "John Doe"

Create a global config

$ git config --global user.name "John Doe"

Create a system config

$ git config --system user.name "John Doe"

Git Tools

- Git Bash
- Git GUI
- Git CMD
- Gitk

GitBash:

- Git Bash is an application for the Windows environment. It is used as Git command line for windows. Git Bash provides an emulation layer for a Git command-line experience.
- Bash is a standard default shell on Linux and macOS. A shell is a terminal application which is used to create an interface with an operating system through commands.
- By default, Git Windows package contains the Git Bash tool. We can access it by right-click on a folder in Windows Explorer.
- Git Bash also includes the full set of Git core commands like **git clone, git commit, git checkout, git push**, and more.

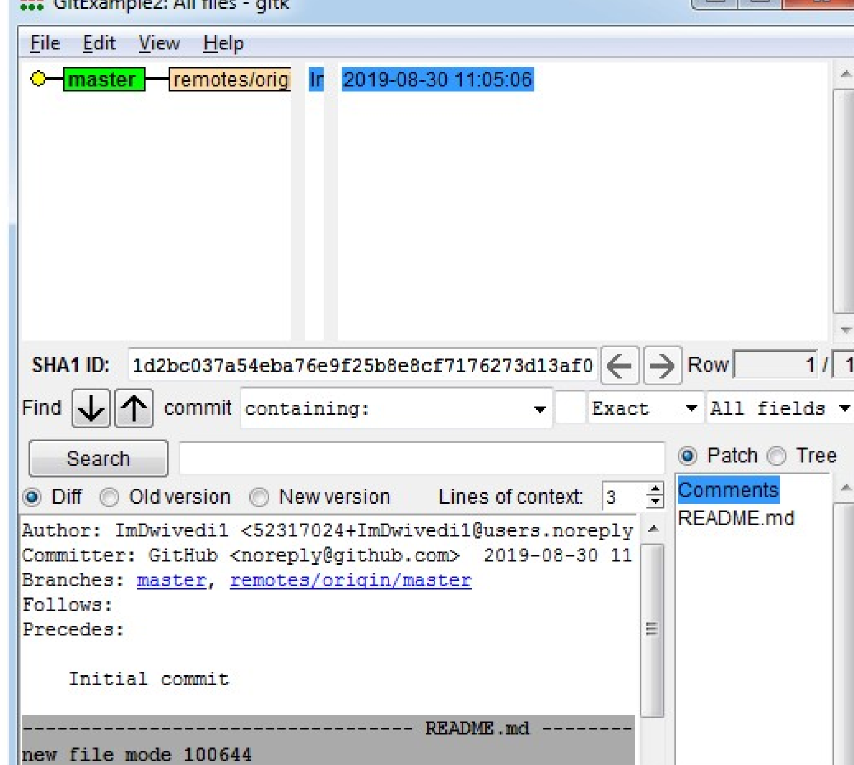
Git Tools

Git GUI:

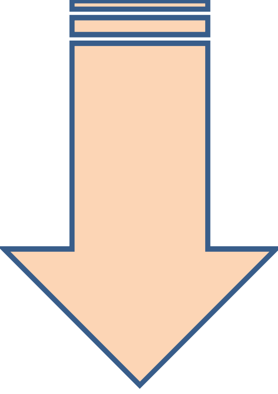
- Git GUI is a powerful alternative to Git BASH. It offers a graphical version of the Git command line function, as well as comprehensive visual diff tools. We can access it by simply right click on a folder or location in windows explorer.
- **$ git gui → for running Git GUI tool**

Gitk:

- **gitk is a graphical history viewer tool**. It's a robust GUI shell over **git log** and **git grep**. This tool is used to find something that happened in the past or visualize your project's history.
- This command invokes the gitk graphical interface and displays the project history. The Gitk interface looks like this:



Github Desktop



Github Desktop

- GitHub Desktop is an open source tool that enables you to be more productive. GitHub Desktop encourages you and your team to collaborate using best practices with Git and GitHub.
- GitHub Desktop extends and simplifies your Git and GitHub workflow using a visual interface.
- With GitHub Desktop, you can interact with GitHub using a GUI instead of the command line or a web browser. You can use GitHub Desktop to complete most Git commands from your desktop, such as pushing to, pulling from, and cloning remote repositories, attributing commits, and creating pull requests, with visual confirmation of changes.
- You can use GitHub Desktop to complete most Git commands from your desktop with visual confirmation of changes. You can push to, pull from, and clone remote repositories with GitHub Desktop, and use collaborative tools such as attributing commits and creating pull requests.
- Getting started with GitHub Desktop - GitHub Docs

Just a **few of the many things you can do with GitHub Desktop are:**

- Add changes to your commit interactively
- Quickly add co-authors to your commit
- Checkout branches with pull requests and view CI (Continuous Integration) statuses
- Compare changed images

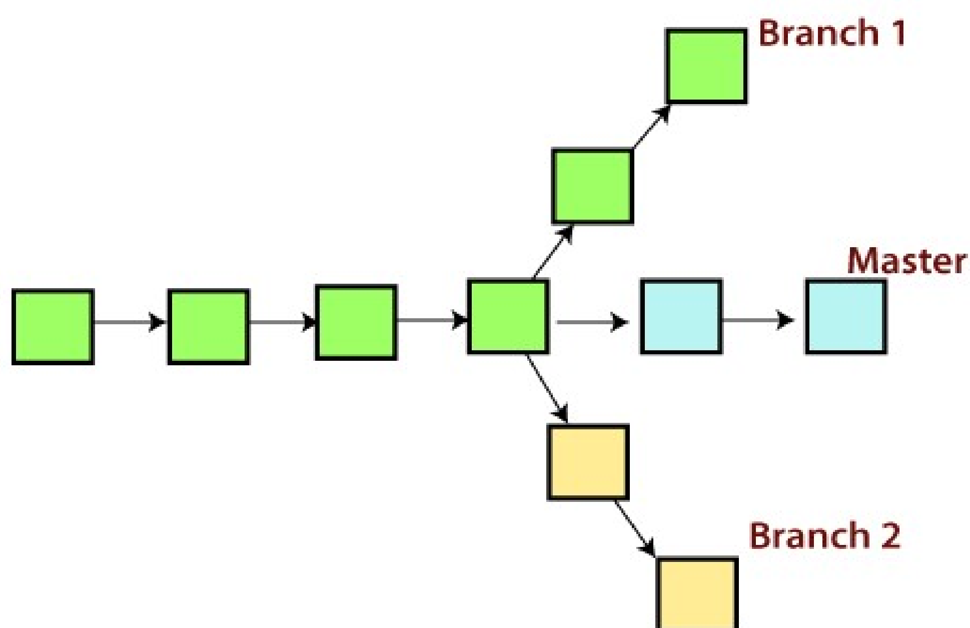
Github Desktop

Branching & Merging

- You can use branches to safely experiment with changes to your project. Branches isolate your development work from other branches in the repository. For example, you could use a branch to develop a new feature or fix a bug.
- You always create a branch from an existing branch. Typically, you might create a branch from the default branch of your repository. You can then work on this new branch in isolation from changes that other people are making to the repository.
- You can also create a branch starting from a previous commit in a branch's history. This can be helpful if you need to return to an earlier view of the repository to investigate a bug, or to create a hot fix on top of your latest release.
- Once you're satisfied with your work, you can create a pull request to merge your changes in the current branch into another branch.
- You can always create a branch in GitHub Desktop if you have read access to a repository, but you can only push the branch to GitHub if you have write access to the repository.
- Repository administrators can enable protections on a branch. If you're working on a branch that's protected, you won't be able to delete or force push to the branch. Repository administrators can enable other protected branch settings to enforce specific workflows before a branch can be merged.

Git basics - Git Branch

- **Git Branch**
- A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems.
- A Git project can have more than one branch.
- These branches are a pointer to a snapshot of your changes.
- When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes.
- So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.



Github Desktop

- ✓ **Creating a Branch**
- ✓ **Creatinga Branch from a previous commit**
- ✓ **Publishing a Branch**
- ✓ **Switching between Branches**
- ✓ **Deleting a Branch**

To create a new branch in GitHub Desktop,

Steps:

1. first click the middle header that says **Current Branch**.
2. Then, click the button that says **New Branch**.
3. Enter a name for your branch in the dialog box that appears and click **Create Branch**.

Creatinga Branch from a previous commit

Steps:

1. In the left sidebar, click **History**.
2. Right-click on the commit you would like to create a new branch from and select **Create Branch from Commit**.

Deleting a Branch:

Steps:

1. In the repository bar, click **Current Branch**, then click the branch that you want to delete.
2. In your menu bar, click **Branch**, then click **Delete...**. You can also press Ctrl+Shift+D.

Github Desktop

- **Publishing a Branch**
- If you create a branch on GitHub, you'll need to publish the branch to make it available for collaboration on GitHub.

Steps:

1. In the repository bar, click **Current Branch**, then click the branch that you want to publish.
2. Click **Publish branch**.

Switching between Branches:

- You can view and make commits to any of your repository's branches. If you have uncommitted, saved changes, you'll need to decide what to do with your changes before you can switch branches. You can commit your changes on the current branch, stash your changes to temporarily save them on the current branch, or bring the changes to your new branch.

Steps:

1. In the repository bar, click **Current Branch**, then click the branch that you want to switch to.
2. If you have saved, uncommitted changes, in the "Switch Branch" window, select **Leave my changes on CURRENT-BRANCH** or **Bring my changes to NEW-BRANCH**, then click **Switch Branch**.

Github Desktop

Managing Commits in Github Desktop:

1. **Reverting a commit**
2. **Cherry-picking a commit**
3. **Reordering Commits**
4. **Stashing Commits**
5. **Amending a Commit**
6. **Merging tags**
7. **Checking out a Commit**

Reverting a commit:

- You can use GitHub Desktop to revert a specific commit to remove its changes from your branch.
- In the left sidebar, click **History**. → Right-click the commit you want to revert and click **Revert Changes in Commit**.

Github Desktop

Cherry-picking a commit:

- You can use GitHub Desktop to pick a specific commit on one branch and copy the commit to another branch.

Steps:

1. In GitHub Desktop, click **Current Branch**.
2. In the list of branches, click the branch that has the commit that you want to cherry-pick.
3. In the left sidebar, click **History**.
4. Select the commit you would like to cherry-pick.
5. Right-click the selected commit and click **Cherry pick commit**, then select the branch that you want to copy the commit to. You can also drag the commit that you want to cherry-pick from the "History" tab to the **Current Branch** dropdown menu, then drop the commit on the branch that you want to copy the commit to.

Reordering Commits

You can use GitHub Desktop to reorder commits in your branch's history.

Steps:

1. In GitHub Desktop, click Current Branch.
2. In the list of branches, click the branch with the commits that you want to reorder.
3. In the left sidebar, click History.
4. Drag the commit that you want to reorder and drop it between two adjoining commits.

Github Desktop

Cherry-picking a commit:

- You can use GitHub Desktop to pick a specific commit on one branch and copy the commit to another branch.

Steps:

1. In GitHub Desktop, click **Current Branch**.
2. In the list of branches, click the branch that has the commit that you want to cherry-pick.
3. In the left sidebar, click **History**.
4. Select the commit you would like to cherry-pick.
5. Right-click the selected commit and click **Cherry pick commit**, then select the branch that you want to copy the commit to. You can also drag the commit that you want to cherry-pick from the "History" tab to the **Current Branch** dropdown menu, then drop the commit on the branch that you want to copy the commit to.

Reordering Commits

You can use GitHub Desktop to reorder commits in your branch's history.

Steps:

1. In GitHub Desktop, click Current Branch.
2. In the list of branches, click the branch with the commits that you want to reorder.
3. In the left sidebar, click History.
4. Drag the commit that you want to reorder and drop it between two adjoining commits.

Github Desktop

Stashing a Commit:

If you have saved changes that you are not ready to commit yet, you can stash the changes for later. When you stash changes, the changes are temporarily removed from the files and you can choose to restore or discard the changes later.

1. Right click the **changed files** header
2. Click **Stash All Changes**..

Github Desktop

Stashing a Commit:

Git commands

- Git Config command
- Git init command
- Git clone command
- Git add command
- Git commit command
- Git status command
- Git push Command
- Git pull command
- Git Branch Command
- Git Merge Command
- Git log command
- Git remote command

Git basics

- git Push
- git Pull
- git Branch
- git Merge
- git Remote
- git checkout
- git diff
- git reset
- git rm
- git log
- git show
- git tag

Git basics

Step 1:Create folder for Git:

$mkdir myproject

A folder will be created in c:/users/bh/ with myproject name

$cd myproject

Step 2: Initialize git: initialize the project directory as a Git repository.

$git init

git init

git add .

git commit –m “The first commit”

Step 3: Username password setting: (User Identity setting):

git config --global user.name "w3schools-test"

git config --global user.email test@w3schools.com

Ex: git config user.name "kbhaskar"

git config user.email kbhaskar511@gmail.com

git config –global user.name

→ displays username

--global specifies username and password is for all repositories

Git basics

Step 4:

git remote remove origin →

git remote add origin git@github.com:kbhaskararao/DevOps.git

- When developers want to take their local Git repository and share it with another developer or push the code into a cloud-based distributed version control service, such as GitHub or GitLab, they can use the git remote add origin command.

Step 5:

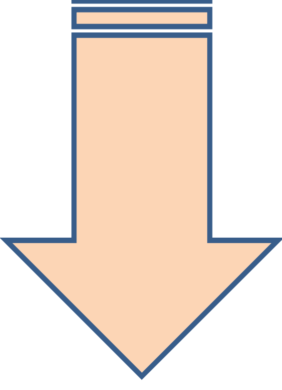
git push origin master

Where:

origin is the default name (alias) of Your remote repository and

master is the remote branch You want to push Your changes to.

git clone



git clone

1. Create a folder in local file system.

2. Goto the location of the folder from Git Bash

3. Login to the github and identify the url of repo

4. Use ‘git clone’ to create a copy of the repo

Git clone:

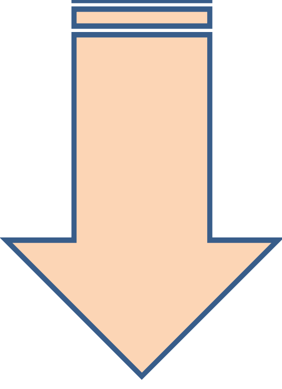
- The git clone command is used to create a copy of a specific repository or branch within a repository.
- When you clone a repo you get a copy of the entire history of the repo. The command used for cloning any repository is:
- **git clone <repository-link>**
- When you clone a repo you get a copy of the entire history of the repo.

Cloning a branch:

Git clone –b <branch-name> <repo-url>

git clone -b master https://github.com/kbhaskarcs/interior-designs.git

git push

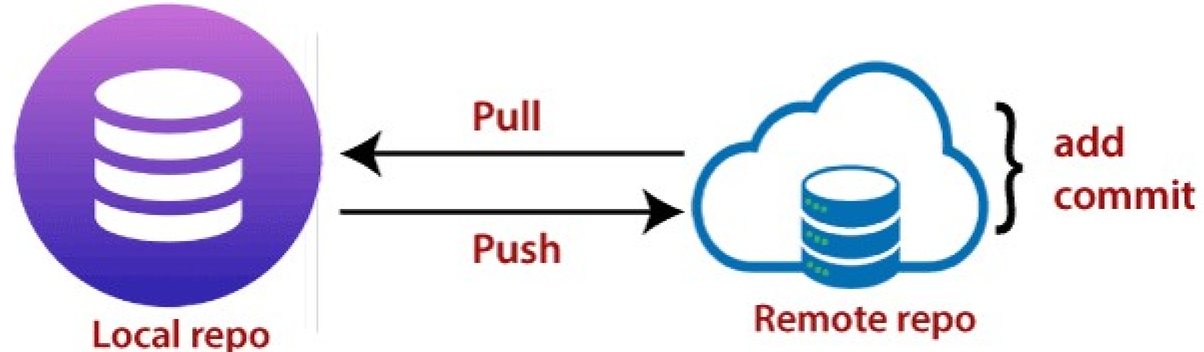


Git basics – git push

git push:

The push term refers to uploading local repository content to a remote repository. Pushing is an act of **transfer commits from your local repository to a remote repository**. Pushing is capable of overwriting changes; caution should be taken when pushing.

- The git push command is used to transfer or push the commit, which is made on a local branch in your computer to a remote repository like GitHub. The command used for pushing to GitHub
- we can say the push updates the remote refs with local refs.
- If we do not specify the location of a repository, then it will push to default location at **origin master**.
- The "git push" command is used to push into the repository. The push command can be considered as a tool to transfer commits between local and remote repositories. The basic syntax is given below:
- **$ git push <option> [<Remote URL><branch name><refspec>...]**



Git basics – git push

$ git push <option> [<Remote URL><branch name><refspec>...]

<option> can be:

- **<repository>:** The repository is the destination of a push operation. It can be either a URL or the name of a remote repository.
- **<refspec>:** It specifies the destination ref to update source object.
- **--all:** The word "all" stands for all branches. It pushes all branches.
- **--prune:** It removes the remote branches that do not have a local counterpart. Means, if you have a remote branch say **demo**, if this branch does not exist locally, then it will be removed.
- **--mirror:** It is used to mirror the repository to the remote. Updated or Newly created local refs will be pushed to the remote end. It can be force updated on the remote end. The deleted refs will be removed from the remote end.
- **--dry-run:** Dry run tests the commands. It does all this except originally update the repository.
- **--tags:** It pushes all local tags.
- **--delete:** It deletes the specified branch.

git push origin master: → pushes the local content on to the master branch of the remote location.

- ***Git push origin master*** is a special command-line utility that specifies the remote branch and directory. When you have multiple branches and directory, then this command assists you in determining your main branch and repository.

Git basics – git push

- Generally, the term **origin** stands for the remote repository, and master is considered as the main branch. So, the entire statement "**git push origin master**" pushed the local content on the master branch of the remote location.
- **Syntax:**
- $ git push origin master

git force push: The git force push allows you to push local repository to remote without dealing with conflicts.

$ git push <remote><branch> -f (OR)

$ git push <remote><branch> -force

$git push origin master –f (OR)

$git push <remote> -f

- ➢ When the remote and the branch both are omitted, the default behavior is determined by **push.default** setting of git config.

  $ git push -f

$ git push -v → pushes objects verbosely (gives details of objects pushed)

(OR)

$ git push --verbose

Git basics

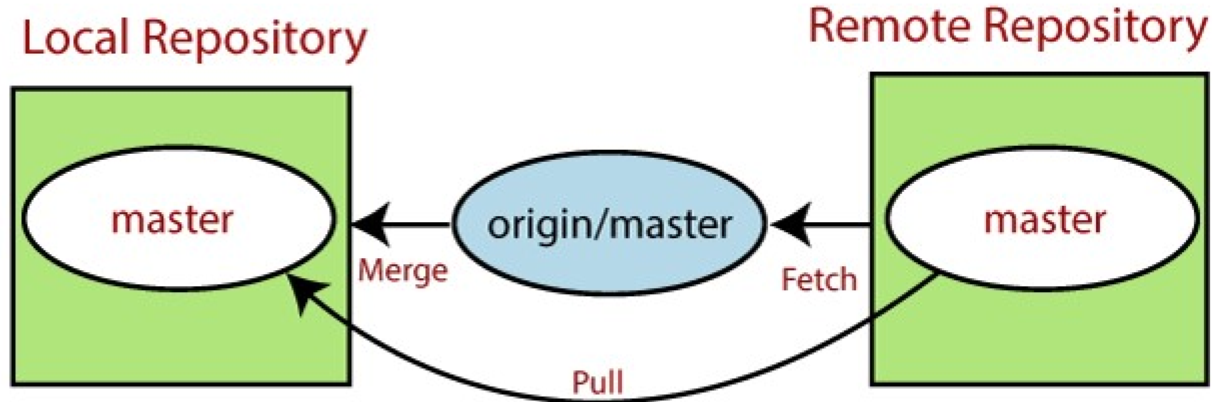
Delete a Remote Branch:

- We can delete a remote branch using git push. It allows removing a remote branch from the command line. To delete a remote branch, perform below command:

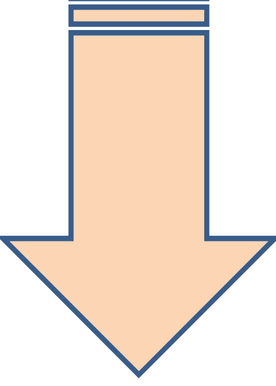
  $ git push origin **-delete** edited → deletes a remote branch named 'edited'

Git Pull / Pull Request:

- The term pull is used to receive data from GitHub. It fetches and merges changes from the remote server to your working directory. The **git pull command** is used to pull a repository.

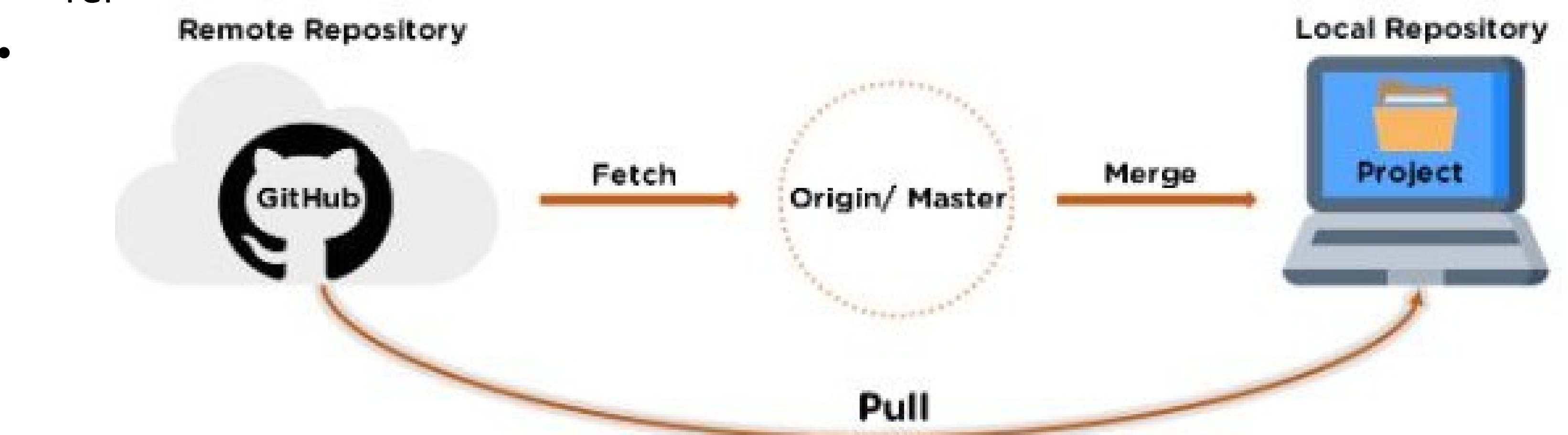


git pull

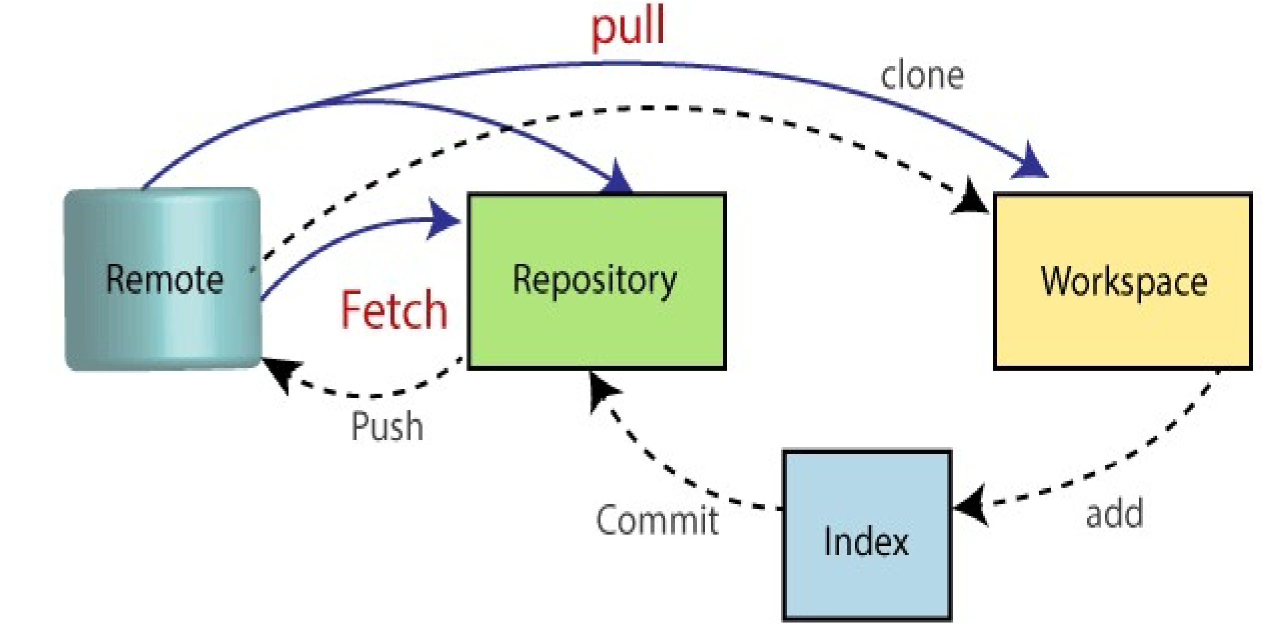


Git basics – git pull

- The **git pull** command is used to fetch and merge code changes from the remote repository to the local repository.
- git pull is a Git command used to update the local version of a repository from a remote.
- **pull** is a combination of **fetch and merge.** It is used to pull all changes from a remote repository into the branch you are working on. **In the first stage**, Git fetch is executed that downloads content from the required remote repository. **Then,** the Git merge command combines multiple sequences of commits into a single branch.
- It updates the local branches with the remote-tracking branches.
- Remote tracking branches are branches that have been set up to push and pull from the remote repository.
- **Use pull to update our local Git.** This is how you keep your local Git up to date from a rer
-



Git basics – git pull

- The below figure demonstrates how pull acts between different locations and how it is similar or dissimilar to other related commands.
- 

pull
Change local repo
Git status
Git add .
Git

Git basics – git pull

1. git pull https://github.com/kbhaskararao/DevOps.git

Git pull example:

bh@mypc MINGW64 ~/DevOps (master)

2. $ git status

On branch master

Changes not staged for commit:

 (use "git add <file>..." to update what will be committed)

 (use "git restore <file>..." to discard changes in working directory)

   modified: index.html

no changes added to commit (use "git add" and/or "git commit -a")

bh@mypc MINGW64 ~/DevOps (master)

3. $ git add .

bh@mypc MINGW64 ~/DevOps (master)

4. $ git commit -m "first changes"

[master 0d46ec1] first changes

 1 file changed, 2 insertions(+), 1 deletion(-)

bh@mypc MINGW64 ~/DevOps (master)

5. Git status

•

Git basics – git pull

The syntax of the git pull command is given below:

Syntax:

- $ git pull **<option>** [**<repository** URL**><refspec>**...]

Here:

<option>: Options are the commands; these commands are used as an additional option in a particular command. Options can be **-q** (quiet), **-v** (verbose), **-e**(edit) and more.

<repository URL>: Repository URL is your remote repository's URL where you have stored your original repositories like GitHub or any other git service. This URL looks like:

- https://github.com/ImDwivedi1/GitExample2.git

<Refspec>: A ref is referred to commit, for example, head (branches), tags, and remote branches. You can check head, tags, and remote repository in **.git/ref** directory on your local repository. **Refspec** specifies and updates the refs.

Pulling remote repo:

$ git pull → pull a remote repository

Pulling from a specific branch: git pull <Remote-Name> <Branch-Name>

Git Pull Remote Branch:

- Git allows fetching a particular branch. Fetching a remote branch is a similar process, as mentioned above, in **git pull command**. The only difference is we have to copy the URL of the particular branch we want to pull.

Syntax:

- $ git pull **<remote** branch URL**>**

Git basics – git pull

$ git fetch -all → Use the git fetch command to download the latest updates from the remote without merging or rebasing.

- Use the **git reset** command to reset the master branch with updates that you fetched from remote. The hard option is used to forcefully change all the files in the local repository with a remote repository.

Ex:

$ git reset -hard **<remote>/<branch_name>**

$ git reset-hard master

Git Pull Origin Master:

- There is another way to pull the repository. We can pull the repository by using the **git pull** command. The syntax is given below:
- $ git pull **<options><remote>/<branchname>**
- $ git pull origin master

$ git remote -v → To check the remote location of the repository, use the below command

Git basics – pull command

- pull is a combination of 2 different commands:
  - fetch
  - merge

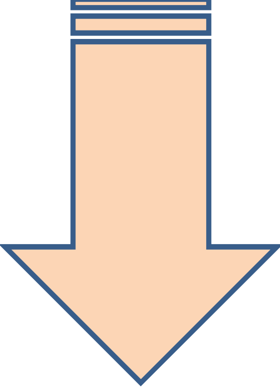
Example:

```
bh@mypc MINGW64 ~ (master)
$ git pull https://github.com/kbhaskararao/DevOps.git
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 16 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (16/16), 16.18 MiB | 1.25 MiB/s, done.
From https://github.com/kbhaskararao/DevOps
 * branch            HEAD       -> FETCH_HEAD
```

Git - username, mailID configuration

- $ **git config --global user.name** "John Doe"
- $ **git config --global user.email** johndoe@example.com

git branch



Git basics – git branch

Branching:

- ✓ Git branching allows developers to diverge from the production version of code to fix a bug or add a feature.
- ✓ Developers create branches to work with a copy of the code without modifying the existing version.
- ✓ You create branches to isolate your code changes, which you test before merging to the main branch (more on this later).
- ✓ There is nothing special about the main branch. It is the first branch made when you initialize a Git repository using the git init command.

4 ways of creating Branches:

1. Using ***git branch*** command

- ✓ To create a branch, use the **git branch** command followed by the name of the branch. After making the branch, use **git branch** gain to view available branches.

```
git branch <branch name>
git branch
```

Git - Branching

- **Branching** is probably the most powerful feature of Git. A branch represents an independent development path. The branches coexist in the same repository, but each one has its own history.

Creating a branch with Git is so simple:

git branch <branch-name>

For example:

git branch second-branch

- Git creates a pointer with that branch name (second-branch) that points to the commit where the branch has been created.

To work with the branch: use checkout command

git checkout second-branch

To return to master:

git checkout master

Merging branches:

It is combining the branches.

git checkout master

git merge second-branch

Listing branches: git branch → lists local repo branches

git branch –r → lists branches in remote repository

git branch –a → lists all local and remote branches of the repository

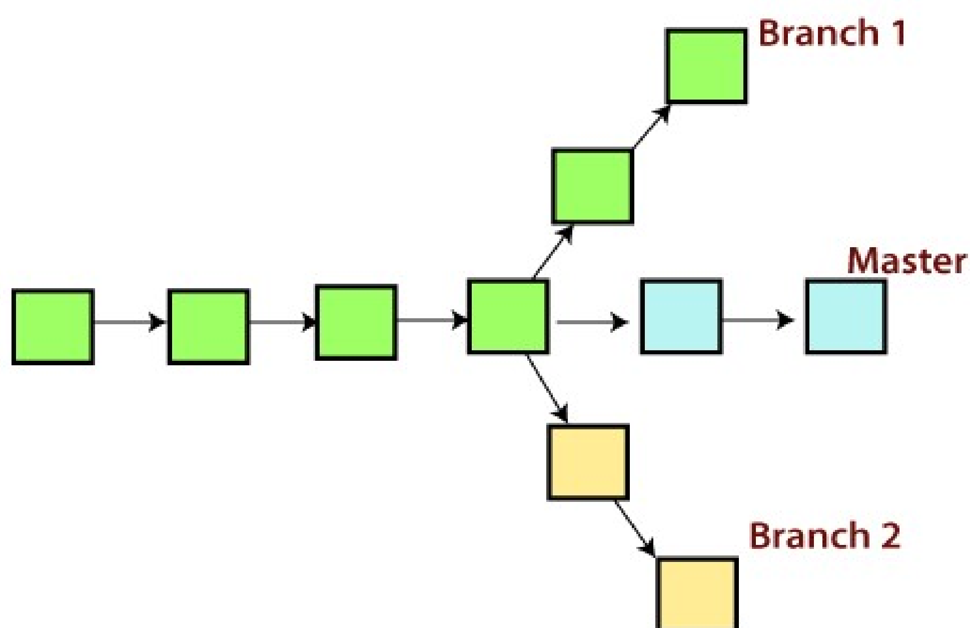
git branch –vv → lists detailed info of the local branches

git branch –vvr → lists detailed info of the remote branches

git branch –vvra → lists detailed info of the local remote branches

Git basics - Git Branch

- **Git Branch**
- A branch is a version of the repository that diverges from the main working project. It is a feature available in most modern version control systems.
- A Git project can have more than one branch.
- These branches are a pointer to a snapshot of your changes.
- When you want to add a new feature or fix a bug, you spawn a new branch to summarize your changes.
- So, it is complex to merge the unstable code with the main code base and also facilitates you to clean up your future history before merging with the main branch.



Git basics – git branch

Git Master Branch:

- The master branch is a **default branch in Git**.
- It is instantiated when first commit made on the project.
- When you make the first commit, you're given a master branch to the starting commit point. When you start making a commit, then master branch pointer automatically moves forward. A repository can have only one master branch.
- Master branch is the branch in which all the changes eventually get merged back. It can be called as an official working version of your project.

5 Operations on branches:

- **Create**
- **List**
- **Delete**
- **Switch**
- **Rename**

Git basics – git branch

3 Operations on branches:

- **Create**
- **List**
- **Delete**

Create Branch:

- You can create a new branch with the help of the **git branch** command. This command will be used as:

Syntax:

- $ git branch **<branch** name**>**
- $git branch B1 → create the **branch B1** locally in Git directory.

List Branch:

- You can List all of the available branches in your repository by using the following command.
- Either we can use **git branch --list** or **git branch** command to list the available branches in the repository.

Syntax:

$ git branch --list (OR)

$ git branch

➢ Here, both commands are listing the available branches in the repository. The symbol * is representing currently active branch.

git show-branch → gives detailed info on the branches.

Git basics – git branch

Delete Branch

- You can delete the specified branch. It is a safe operation. In this command, Git prevents you from deleting the branch if it has unmerged changes. Below is the command to do this.

Syntax:

$ git branch -d **<branch** name**>**

Ex:

$git branch –d B1 → delete the existing branch B1 from the repository.

The '**git branch D**' command is used to delete the specified branch.

Syntax:

$ git branch -D **<branch** name**>**

Delete a Remote Branch:

- You can delete a remote branch from Git desktop application. Below command is used to delete a remote branch:

Syntax:

- $ git push origin -delete **<branch** name**>**

Git basics

Switch Branch:

- Git allows you to switch between the branches without making a commit. You can switch between two branches with the **git checkout** command. To switch between the branches, below command is used:
- $ git checkout**<branch** name**>**

Switch from master Branch

- You can switch from master to any other branch available on your repository without making any commit.
- **Syntax:**
- $ git checkout **<branch** name**>**

Switch to master branch

- You can switch to the master branch from any other branch with the help of below command.
- **Syntax:**
- $ git branch -m master

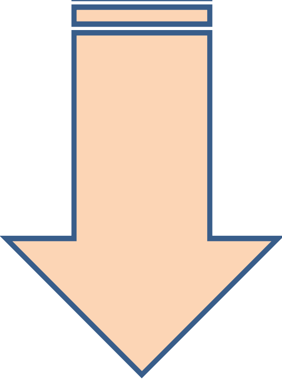
Rename Branch

- We can rename the branch with the help of the **git branch** command. To rename a branch, use the below command:

Syntax:

- $ git branch -m **<old** branch name**><new** branch name**>**

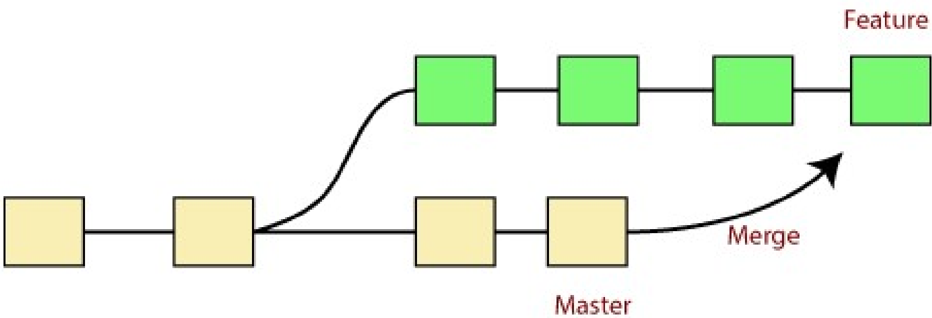
git merge



Git basics – git merge

Git Merge and Merge Conflict:

- Git merge will help to combine the changes from two or more branches into a single branch. Developers will work on different branches to improve code or to develop the code after completion we can merge them into a single version of the code.
- **Git allows you to merge the other branch with the currently active branch**. You can merge two branches with the help of **git merge** command.
- In Git, the merging is a procedure to connect the forked history.
- It joins two or more development history together.
- The git merge command facilitates you to take the data created by git branch and integrate them into a single branch.
- Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.



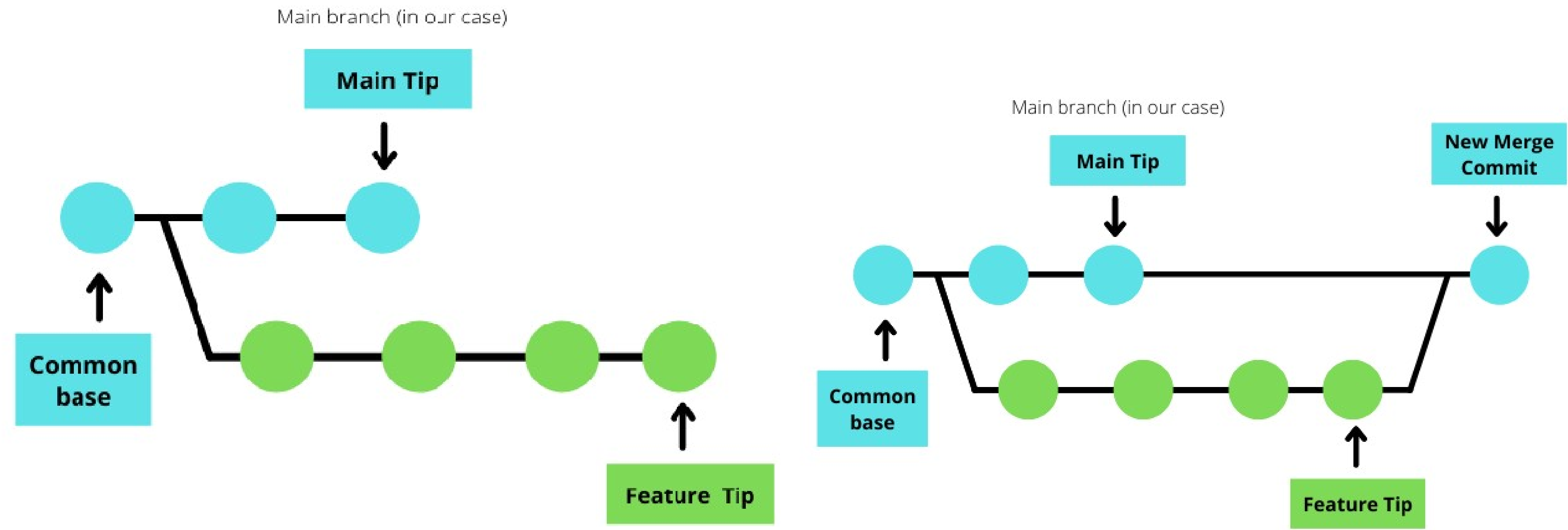
Syntax:

- $ git merge **<branch** name>   branchName → name of the branch to be merged.

Ex: git merge B1   → merges B1 with the master branch

git merge - working

- The concept of git merging is basically to merge multiple sequences of commits, stored in multiple branches in a unified history, or to be simple you can say in a single branch.
- What happens is when we try to merge two branches, git takes two commit pointers and starts searching for a common base commit in those two specified bit branches. When git finds the common base commit it simply creates a “merge commit” automatically and merges each queued merge commit sequence. There is a proper merging algorithm in git, with the help of which git performs all of these operations and presents conflicts if there are any.

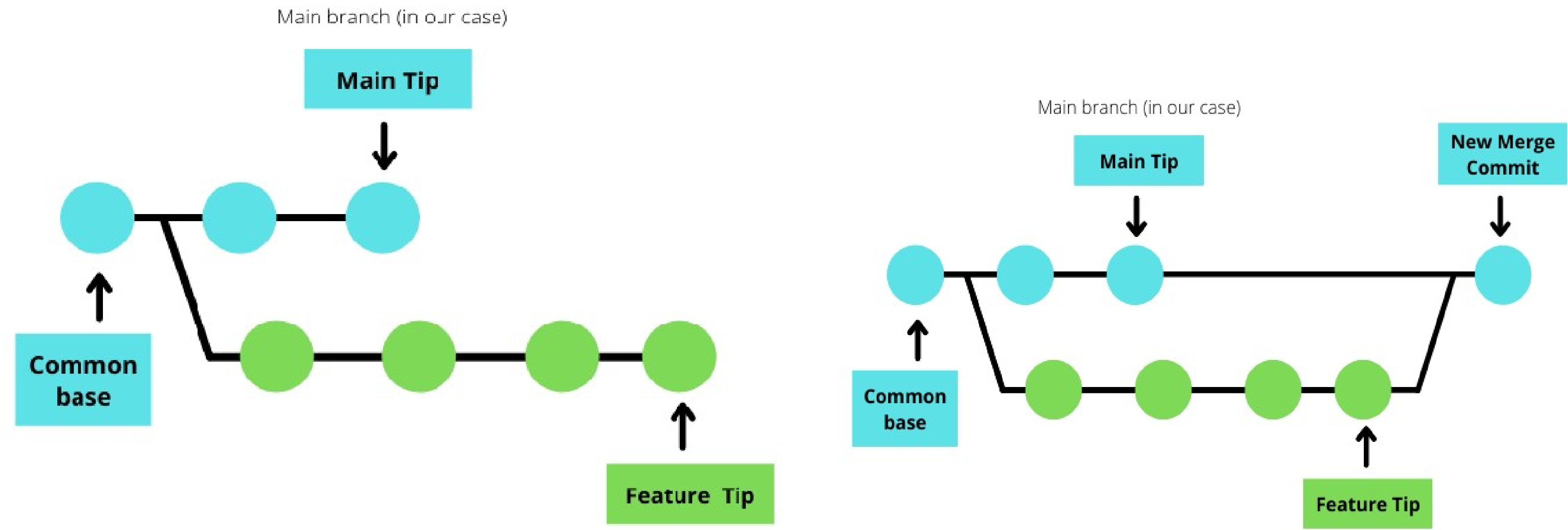


git merge

Merging types:

1. Fast-forward merging.
2. Three-way merging.

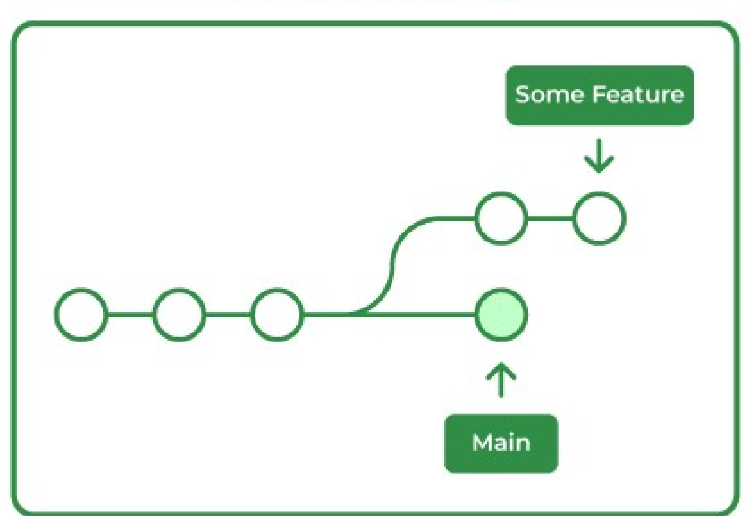
1. Fast-forward merging: Fast forward merge happens when the tip of the current branch (“dev” in our case) is a direct ancestor of the target branch (“main” in our case). Here instead of actually merging the two branches git simply moves the current branch tip up to the target branch tip. Fast-forward merge is not possible if the branches have diverged.

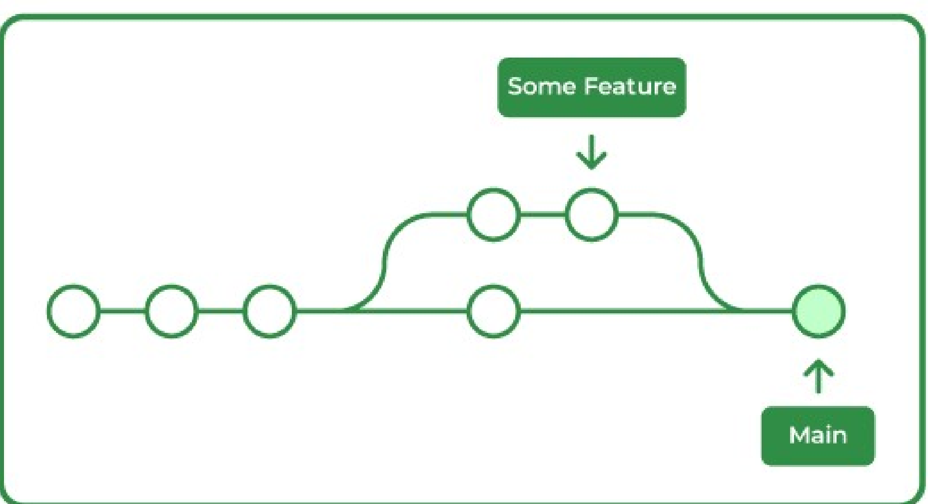


git merge

2. Three-way merging.

✓ When the base branch has changed since the branch was first created, this kind of merging takes place. Git in this situation generates a fresh merging commit that incorporates the modifications from both branches. Git compares the modifications made to both branches with those made to the base branch using a three-way merge process. Following that, it integrates both sets of changes into a single new commit.





To merge 'emergency-fix' branch with master:
git checkout master
git merge emergency-fix

git merge - steps

Steps To Merge a Branch:

Step 1: Creating a new branch.

- Create a new branch from the remote repository branch which you want to merge. If errors are faced while merging we can go back to the previous version immediately.

Step 2: Make sure always latest changes are pulled.

- Always make sure before merging the latest changes that the latest changes are pulled from both branches like the master and the branch you want to merge.

Step 3: Resolving the merge conflicts.

- While merging the branches it is possible that some merge conflicts will be raised then git will prompt you to resolve the merge conflicts. If any merge conflicts are not raised then git will automatically merge the branches.

Step 4: Merged code needs to be tested.

- It is essential to test the merged code and we have to make sure that the code doesn’t have any bugs and it is working properly. To test the code we do it automatically or manually.

Step 5: Commit the merged code.

- Once completing of merging the code if you are satisfied with the work, Know it’s time to commit the new changes of code into the new branch.

Step 6: Push the merged branch.

- Lastly, make the new branch accessible to other team members by pushing it to the repository.

Git basics – git merge

Merge Scenario:

Scenario 1: To merge the specified commit to currently active branch.

Scenario 2: To merge commits into the master branch.

Scenario 3: Git merge branch.

Git basics – git merge

Merge Scenario:

Scenario 1: To merge the specified commit to currently active branch.

Use the below command to merge the specified commit to currently active branch.

$ git merge **<commit ID>**

- The above command will merge the specified commit to the currently active branch. You can also merge the specified commit to a specified branch by passing in the branch name in <commit> .

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git add newfile1.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git commit -m "edited newfile1.txt"
[test d2bb07d] edited newfile1.txt
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git log
commit d2bb07dc9352e194b13075dcfd28e4de802c070b (HEAD -> test)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 11:27:44 2019 +0530

    edited newfile1.txt

commit 2852e020909dfe705707695fd6d715cd723f9540 (test2, master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 10:29:07 2019 +0530

    newfile1 added
```

Git basics – git merge

Scenario 2: To merge commits into the master branch.

- To merge a specified commit into master, first discover its commit id. Use the log command to find the particular commit id.

  $git log

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git log
commit 2852e020909dfe705707695fd6d715cd723f9540 (HEAD -> test)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 10:29:07 2019 +0530

    newfile1 added

commit 4a6693a71151323623c04dd75cb0d44c1c4dbadf (origin/master, origin/HEAD, mas
ter)
Merge: 30193f3 78c5fbd
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Mon Sep 9 15:24:13 2019 +0530

    Merge pull request #1 from ImDwivedi1/branch2

    Create merge the branch
```

➢ Now, Switch to branch 'master' to perform merging operation on a commit.

$ git checkout master

➢ Use the git merge command along with master branch name. The syntax for this is as follows:

$ git merge master

Git basics – git merge

Now, Switch to branch 'master' to perform merging operation on a commit. Use the git merge command along with master branch name. The syntax for this is as follows:
$ git merge master

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git merge 2852e020909dfe705707695fd6d715cd723f9540
Updating 4a6693a..2852e02
Fast-forward
 newfile.txt  | 1 +
 newfile1.txt | 1 +
 2 files changed, 2 insertions(+)
 create mode 100644 newfile.txt
 create mode 100644 newfile1.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ |
```

Git basics – git merge

Scenario 3: Git merge branch.

- Git allows merging the whole branch in another branch. Suppose you have made many changes on a branch and want to merge all of that at a time. Git allows you to do so. See the below example:
- **Step 1:**
- git add newfile1.txt
- In the given output, I have made changes in newfile1 on the test branch. Now, I have committed this change in the test branch
- **Step 2:**
- git commit –m “edit newfile1”
- **Step 3:**

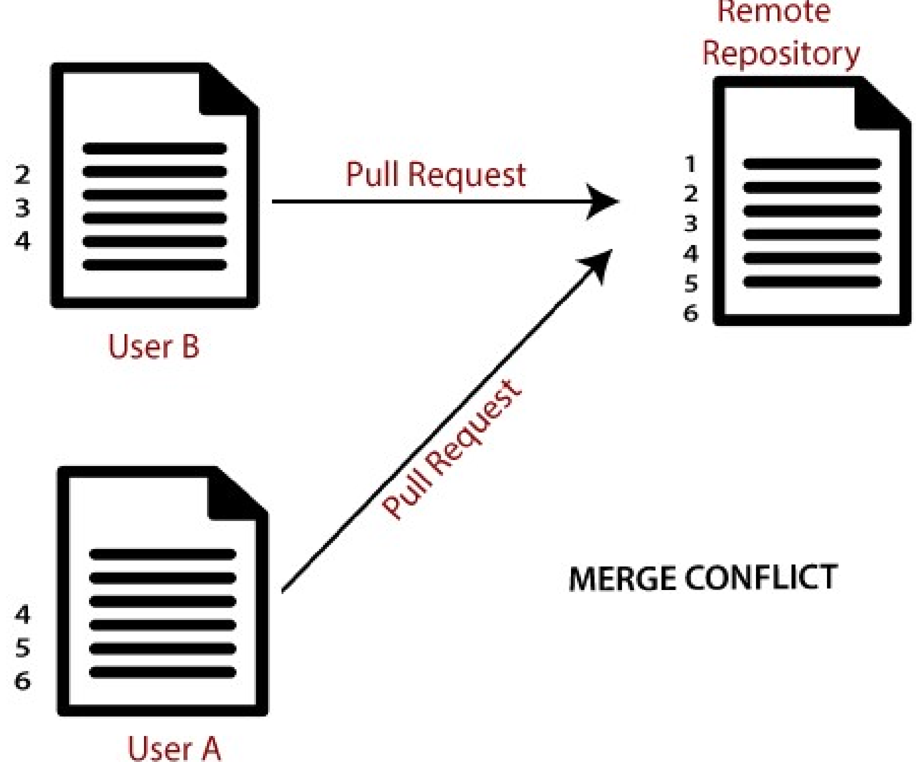
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout master
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git merge test2
Updating 2852e02..a3644e1
Fast-forward
 newfile1.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$
```

Git basics – git merge

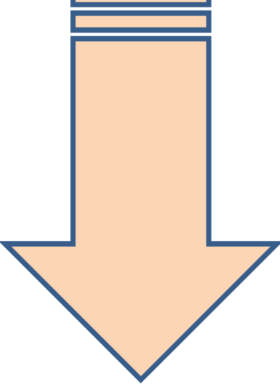
git Merge Conflict:

Merge Conflict definition:

- When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called **merge conflict**.
- If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.



git diff



Git basics – git diff

- Git diff is a command-line utility.
- When it is executed, it runs a diff function on Git data sources. These data sources can be **files, branches, commits**, and more.
- It is used to show changes between commits, commit, and working tree, etc.
- It compares the different versions of data sources.
- However, we can also track the changes with the help of git log command with option -p. The **git log** command will also work as a **git diff** command.

diff scenarios:

Scenario 1: Track the changes that have not been staged.

Scenario 2: Track the changes that have staged but not committed.

Scenario 3: Track the changes after committing a file.

Scenario 4: Track the changes between two commits.

Git basics – git diff

Scenario 1: Track the changes that have not been staged.

- The usual use of git diff command that we can track the changes that have not been staged.
- Suppose we have edited the newfile1.txt file. Now, we want to track what changes are not staged yet. Then we can do so from the git diff command. Consider the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git diff
diff --git a/newfile1.txt b/newfile1.txt
index ade63b7..41a6a9c 100644
--- a/newfile1.txt
+++ b/newfile1.txt
@@ -3,3 +3,4 @@ i am on test2 branch.
 git commit1
 git commit2
 git merge demo
+changes are made to understand the git diff command.
```

Git basics

Scenario 2: Track the changes that have staged but not committed.

- The git diff command allows us to track the changes that are staged but not committed. We can track the changes in the staging area. To check the already staged changes, use the --staged option along with git diff command.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git status
On branch test2
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory
)
        modified:   newfile1.txt

no changes added to commit (use "git add" and/or "git commit -a")

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git add newfile1.txt
```

- Now, the file is added to the staging area, but it is not committed yet.
- So, we can track the changes in the staging area also. To check the staged changes, run the git diff command along with **--staged** option. It will be used as:
-

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git diff --staged
diff --git a/newfile1.txt b/newfile1.txt
index ade63b7..41a6a9c 100644
--- a/newfile1.txt
+++ b/newfile1.txt
@@ -3,3 +3,4 @@ i am on test2 branch.
 git commit1
 git commit2
 git merge demo
+changes are made to understand the git diff command.
```

Git basics

Scenario 3: Track the changes after committing a file.

- Suppose we have committed a file for the repository and made some additional changes after the commit. So we can track the file on this stage also.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git commit -m "newfile1 is updated"
[test2 e553fc0] newfile1 is updated
 1 file changed, 1 insertion(+)
```

- Now, we have changed the newfile.txt file again as "Changes are made after committing the file." To track the changes of this file, run the git diff command with **HEAD** argument. It will run as follows:
-

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git diff HEAD
diff --git a/newfile1.txt b/newfile1.txt
index 41a6a9c..e14624d 100644
--- a/newfile1.txt
+++ b/newfile1.txt
@@ -4,3 +4,4 @@ git commit1
 git commit2
 git merge demo
 changes are made to understand the git diff command.
+changes are made after committing the file.
```

Git basics

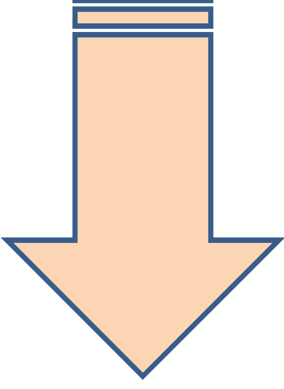
Scenario 4: Track the changes between two commits.

- We can track the changes between two different commits. Git allows us to track changes between two commits, whether it is the latest commit or the old commit. But the required thing for this is that we must have a list of commits so that we can compare. The usual command to list the commits in the git log command. To display the recent commits, we can run the command as: $ git log

$ git log -p --follow -- filename →display all the commits of a specified file.

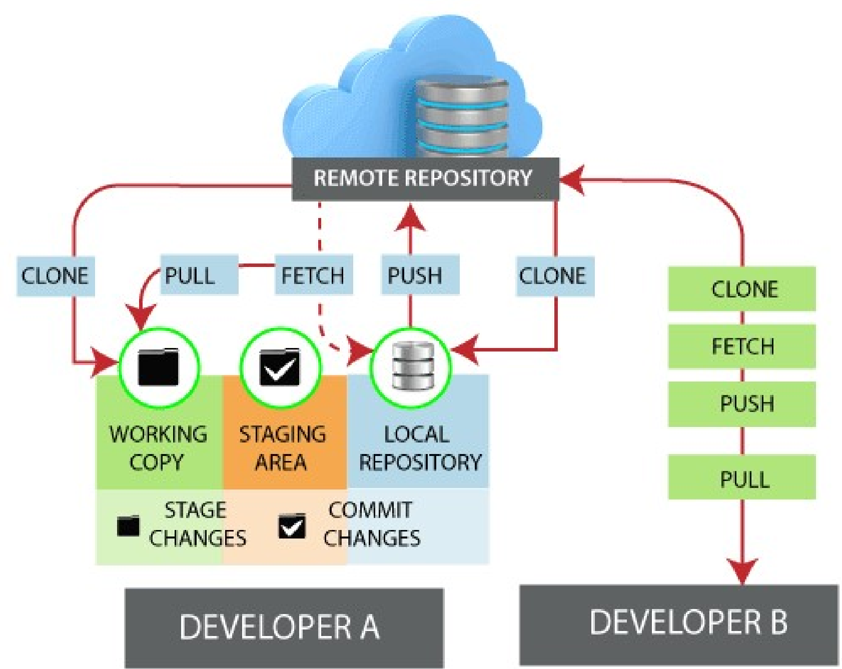
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git log -p --follow -- newfile1.txt
commit e553fc08cb41fd493409a90e12064c7a3cb2da7e (HEAD -> test2)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Nov 6 18:16:04 2019 +0530

    newfile1 is updated

diff --git a/newfile1.txt b/newfile1.txt
index adc63b7..41a6a9c 100644
--- a/newfile1.txt
+++ b/newfile1.txt
@@ -3,3 +3,4 @@ i am on test2 branch.
 git commit1
 git commit2
 git merge demo
+changes are made to understand the git diff command.

commit f1ddc7c9c765bd688c2c5503b2c88cb1dc835891
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sat Sep 28 12:31:30 2019 +0530

    new comit on test2 branch

diff --git a/newfile1.txt b/newfile1.txt
:...skipping...
commit e553fc08cb41fd493409a90e12064c7a3cb2da7e (HEAD -> test2)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Nov 6 18:16:04 2019 +0530

    newfile1 is updated

diff --git a/newfile1.txt b/newfile1.txt
index adc63b7..41a6a9c 100644
--- a/newfile1.txt
```

git Remote
git checkout
git reset
git rm
git log
git show
git tag



Git basics – git remote

- In Git, the term remote is concerned with the remote repository.
- It is a shared repository that all team members use to exchange their changes. A remote repository is stored on a code hosting service like an internal server, GitHub, Subversion, and more.
- The developers can perform many operations with the remote server. These operations can be a clone, fetch, push, pull, and more.
- To check the configuration of the remote server, run the **git remote** command.
- The git remote command allows accessing the connection between remote and local.
- If you want to see the original existence of your cloned repository, use the git remote command.



Git basics

Syntax:

$ git remote

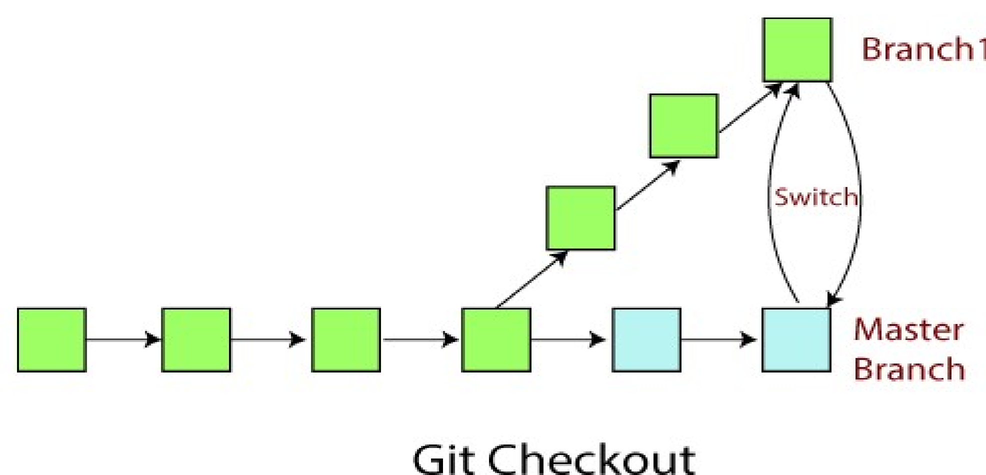
$ git remote -v

- Git remote supports a specific option -v to show the URLs that Git has stored as a short name. These short names are used during the reading and write operation.

Git checkout

Git checkout:

- In Git, the term checkout is used for the act of switching between different versions of a target entity. The **git checkout** command is used to switch between branches in a repository



Git Checkout

• The git checkout command operates upon three different entities which are **files, commits, and branches**. Sometimes this command can be dangerous because there is no undo option available on this command.

• It checks the branches and updates the files in the working directory to match the version already available in that branch, and it forwards the updates to Git to save all new commit in that branch.

•We can perform many operations by git checkout command like the switch to a specific branch, create a new branch, checkout a remote branch, and more.

Git checkout

- The git checkout command operates upon three different entities which are files, commits, and branches.

Checkout types:

1. Checkout Branch
2. Create and Switch Branch
3. Checkout Remote Branch

1. Checkout Branch

:$ git branch

$ git checkout <branchname>

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git branch
  TestBranch
* master

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout testbranch
Switched to branch 'testbranch'
```

Git checkout

2. Create and Switch Branch:

- The git checkout commands let you create and switch to a new branch.
- You can not only create a new branch but also switch i t simultaneously by a single command. The git checkout -b option is a convenience flag that performs run git branch <new-branch>operation before running git checkout <>.
- $ git checkout -b <branchname> new-branch

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout -b branch3
Switched to a new branch 'branch3'
A       design.css
```

- branch3 is created and switched from the master branch.

Git remote

3. Checkout Remote Branch

- Git allows you to check out a remote branch by git checkout command.
- It is a way for a programmer to access the work of a colleague or collaborator for review and collaboration.
- Each remote repository contains its own set of branches. So, to check out a remote branch, you have first to fetch the contents of the branch.

$ git fetch --all

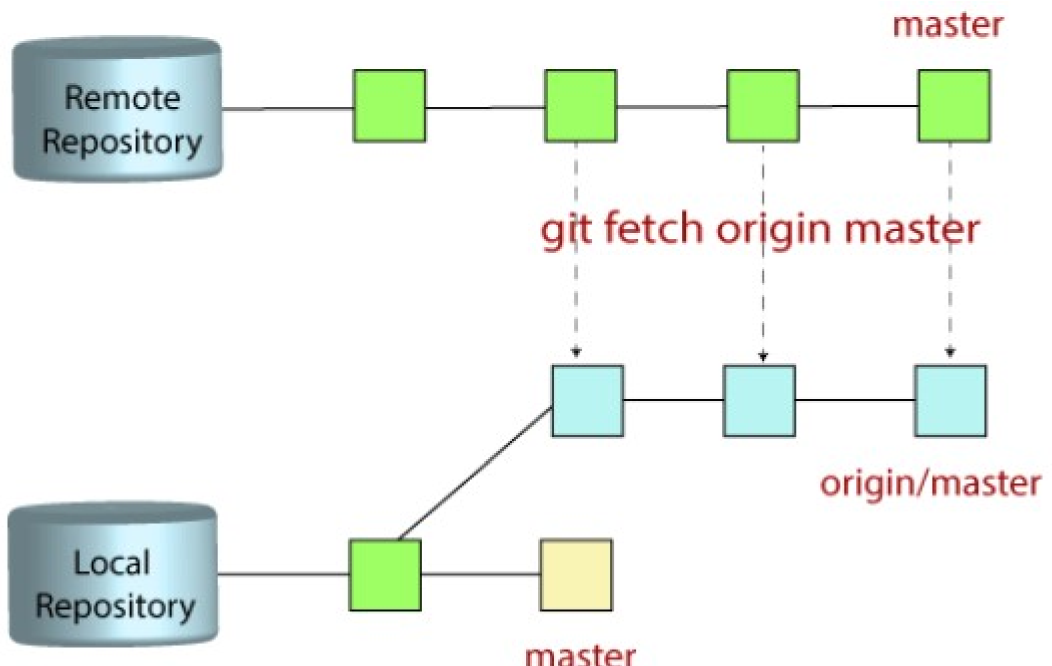
$ git checkout <remotebranch>

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git fetch --all
Fetching origin
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/ImDwivedi1/GitExample2
 * [new branch]      edited     -> origin/edited

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git checkout edited
Switched to a new branch 'edited'
Branch 'edited' set up to track remote branch 'edited' from 'origin'.

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (edited)
$ |
```

- In the above output, first, the **fetch** command is executed to fetch the remote data; after that, the **checkout** command is executed to check out a remote branch.
- Edited is my remote branch. Here, we have switched to edited branch from master branch by git command line.
-

git fetch

- The "**git fetch**" **command** is used to pull the updates from remote-tracking branches.
- we can get the updates that have been pushed to our remote branches to our local machines.
- As we know, a branch is a variation of our repositories main code, so the remote-tracking branches are branches that have been set up to pull and push from remote repository.



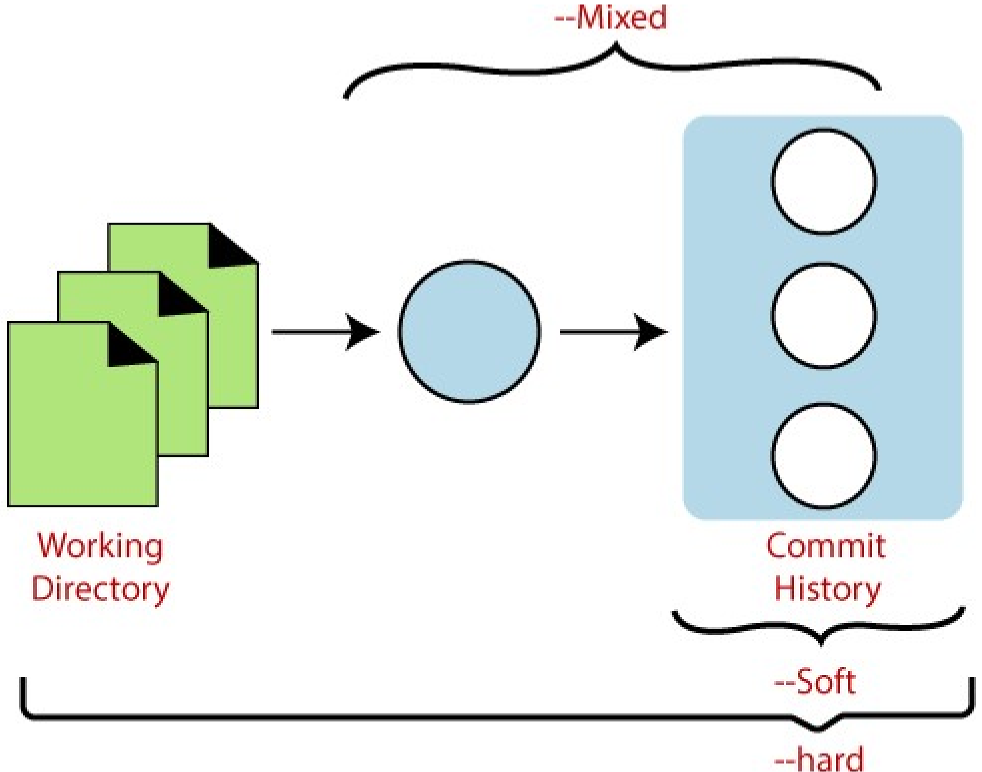
Git basics

- **git reset**
- **git rm**
- **git log**
- **git show**
- **git tag**

git reset:

- The term reset stands for undoing changes. The git reset command is used to reset the changes. The git reset command has three core forms of invocation. These forms are as follows.
  - **Soft**
  - **Mixed**
  - **Hard**
- If we say in terms of Git, then Git is a tool that resets the current state of HEAD to a specified state.
- It is a sophisticated and versatile tool for undoing changes. It acts as a **time machine for Git**. You can jump up and forth between the various commits.
- Each of these reset variations affects specific trees that git uses to handle your file in its content.
- Additionally, **git reset can operate on whole commits objects or at an individual file level**.
- Each of these reset variations affects specific trees that git uses to handle your file and its contents.

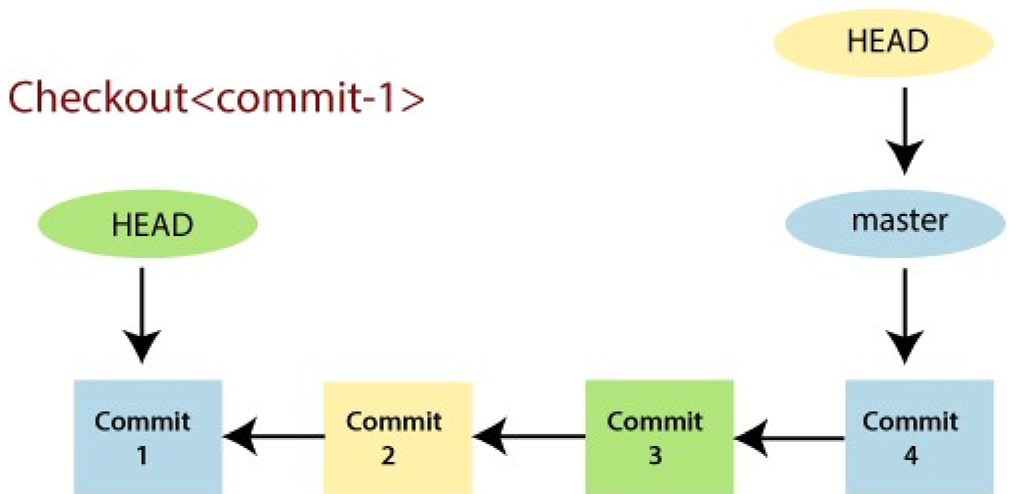
Git basics



The working directory lets you change the file, and you can stage into the index. The staging area enables you to select what you want to put into your next commit. A commit object is a cryptographically hashed version of the content. It has some Metadata and points which are used to switch on the previous commits.

Git Head

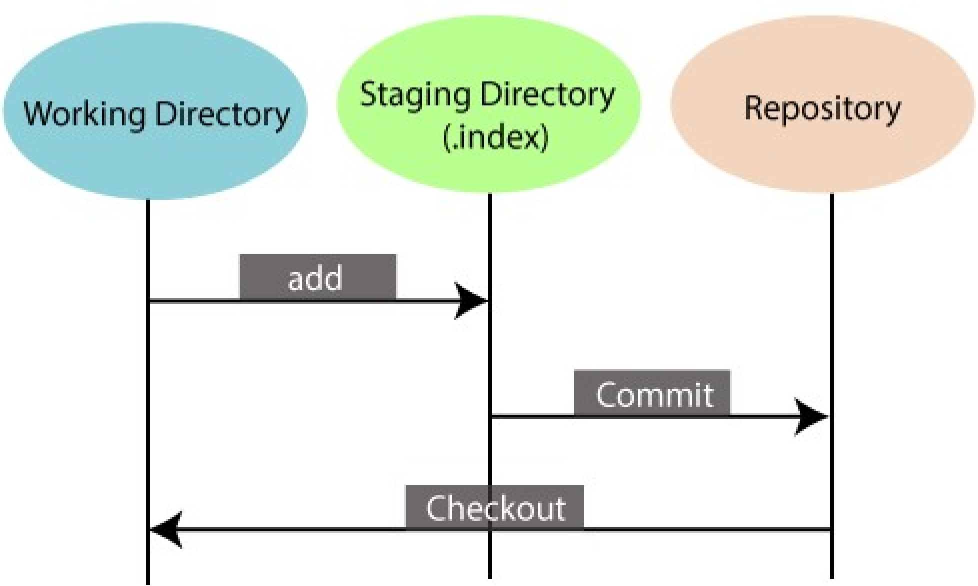
- The **HEAD** points out the last commit in the current checkout branch. It is like a pointer to any reference. The HEAD can be understood as the "**current branch**." When you switch branches with 'checkout,' the HEAD is transferred to the new branch.



The above fig shows the HEAD referencing commit-1 because of a 'checkout' was done at commit-1. When you make a new commit, it shifts to the newer commit. The git head command is used to view the status of Head with different arguments. It stores the status of Head in **.git\refs\heads** directory.

Git Index

- The Git index is a staging area between the working directory and repository. It is used to build up a set of changes that you want to commit together. To better understand the Git index, then first understand the working directory and repository.



There are three places in Git where file changes can reside, and these are working directory, staging area, and the repository. To better understand the Git index first, let's take a quick view of these places.

Git Index

Working directory:

- When you worked on your project and made some changes, you are dealing with your project's working directory. This project directory is available on your computer's filesystem. All the changes you make will remain in the working directory until you add them to the staging area.

Staging area:

- The staging area can be described as a preview of your next commit. When you create a git commit, Git takes changes that are in the staging area and make them as a new commit. You are allowed to add and remove changes from the staging area. The staging area can be considered as a real area where git stores the changes.

Repository:

- In Git, Repository is like a data structure used by GIt to store metadata for a set of files and directories. It contains the collection of the files as well as the history of changes made to those files. Repositories in Git is considered as your project folder. A repository has all the project-related data. Distinct projects have distinct repositories.

Git Reset Hard

- It will first move the Head and update the index with the contents of the commits. It is the most direct, unsafe, and frequently used option. The --hard option changes the Commit History, and ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory need to reset to match that of the specified commit. Any previously pending commits to the Staging Index and the Working Directory gets reset to match Commit Tree. It means any awaiting work will be lost.
- Let's understand the --hard option with an example. Suppose I have added a new file to my existing repository. To add a new file to the repository, run the below command:
- $ git add <file name>
- To check the status of the repository, run the below command:
- $ git status
- To check the status of the Head and previous commits, run the below command:
- $ git log

Git Reset Hard

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git add newfile2.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git status
On branch test2
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   newfile2.txt


HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git log
commit 34c25eb6f855476cb7999e3bec3eeb8e57727162 (HEAD -> test2)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Mon Oct 21 11:35:16 2019 +0530

    Revert "CSS file "

    This reverts commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3.

commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3 (test)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Tue Oct 1 12:30:40 2019 +0530

    CSS file

    See the proposed CSS file.
```

In the above output, I have added a file named **newfile2.txt**. I have checked the status of the repository. We can see that the current head position yet not changed because I have not committed the changes. Now, I am going to perform the **reset --hard** option. The git reset hard command will be performed as:

$ git reset --hard

Git Reset Hard

- Consider the below output:
-

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git reset --hard
HEAD is now at 34c25eb Revert "CSS file "
```

- As you can see in the above output, the -hard option is operated on the available repository. This option will reset the changes and match the position of the Head before the last changes. It will remove the available changes from the staging area. Consider the below output:
-

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git status
On branch test2
nothing to commit, working tree clean
```

- The above output is displaying the status of the repository after the hard reset. We can see there is nothing to commit in my repository because all the changes removed by the reset hard option to match the status of the current Head with the previous one. So the file **newfile2.txt** has been removed from the repository.
- There is a safer way to reset the changes with the help of git stash.
- Generally, the reset hard mode performs below operations:
- It will move the HEAD pointer.
- It will update the staging Area with the content that the HEAD is pointing.
- It will update the working directory to match the Staging Area.
-

Git Rm

- In Git, the term rm stands for remove. It is used to remove individual files or a collection of files.
- The key function of git rm is to remove tracked files from the Git index. Additionally, it can be used to remove files from both the working directory and staging index.
- The files being removed must be ideal for the branch to remove.
- No updates to their contents can be staged in the index. Otherwise, the removing process can be complex, and sometimes it will not happen. But it can be done forcefully by **-f** option.
- If we want to remove the file from our repository. Then it can be done by the git rm command.
- $ git rm <file Name>
- The above command will remove the file from the Git and repository. The git rm command removes the file not only from the repository but also from the staging area.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git rm newfile.txt
rm 'newfile.txt'

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    newfile.txt
```

Git log

- Git log is a utility tool to review and read a history of everything that happens to a repository.
- Generally, the git log is a record of commits. A git log contains the following data:

  A commit hash, which is a 40 character checksum data generated by SHA (Secure Hash Algorithm) algorithm. It is a unique number.

  Commit Author metadata: The information of authors such as author name and email.

  Commit Date metadata: It's a date timestamp for the time of the commit.

  Commit title/message: It is the overview of the commit given in the commit message.

Basic Git log:

- Git log command is one of the most usual commands of git. It is the most useful command for Git. Every time you need to check the history, you have to use the git log command. The basic git log command will display the most recent commits and the status of the head. It will use as:
- **$ git log** → The above command will display the last commits.

Git log

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --
d, tag: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530

    Update design2.css

commit 0a1a475d0b15ecec744567c910eb0d8731ae1af3 (test)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Tue Oct 1 12:30:40 2019 +0530

    CSS file

    See the proposed CSS file.

commit f1ddc7c9e765bd688e2c5503b2c88cb1dc835891
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sat Sep 28 12:31:30 2019 +0530
```

Git log

- We can perform some action like scrolling, jumping, move, and quit on the command line. To scroll on the command line press k for moving up, j for moving down, the spacebar for scrolling down by a full page to scroll up by a page and q to quit from the command line.

Git log online

Git Log Oneline

The oneline option is used to display the output as one commit per line. It also shows the output in brief like the first seven characters of the commit SHA and the commit message.

It will be used as follows:

$ git log --oneline

It displays:

one commit per line

the first seven characters of the SHA

the commit message

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --oneline
0d3835a (HEAD -> master) newfile2 Re-added
56afce0 (tag: -d, tag: --delete, tag: --d, tag: projectv1.1, origin/master, test
ing) Added an empty newfile2
0d5191f added a new image to prject
828b962 (tag: olderversion) Update design2.css
0a1a475 (test) CSS file
f1ddc7c new comit on test2 branch
7fe5e7a  new commit in master branch
dfb5364 commit2
4fddabb commit1
a3644e1 edit newfile1
d2bb07d edited newfile1.txt
2852e02 newfile1 added
4a6693a Merge pull request #1 from ImDwivedi1/branch2
30193f3 new files via upload
78c5fbd Create merge the branch
1d2bc03 Initial commit
```

Git log stat

Git Log Stat:

- The log command displays the files that have been modified. It also shows the number of lines and a summary line of the total records that have been updated.
- Generally, we can say that the stat option is used to display
  - the modified files,
  - The number of lines that have been added or removed
  - A summary line of the total number of records changed
  - The lines that have been added or removed.

Usage: $ git log --stat

Git Log Graph

- Git log command allows viewing your git log as a graph. To list the commits in the form of a graph, run the git log command with --graph option. It will run as follows:
- $ git log --graph
- To make the output more specific, you can combine this command with --oneline option. It will operate as follows:
- $ git log --graph --oneline

Git log stat

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --stat
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

 newfile.txt  | 2 --
 newfile2.txt | 0
 2 files changed, 2 deletions(-)

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --d, ta
g: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

 newfile2.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530

    added a new image to prject

 abc.jpg | Bin 0 -> 777835 bytes
 1 file changed, 0 insertions(+), 0 deletions(-)

commit 828b9628a873091ee26ba53c0fcfc0f2a943c544 (tag: olderversion)
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date:   Thu Oct 3 11:17:25 2019 +0530

    Update design2.css
```

Git log p or patch

Git log P or Patch:

- The git log patch command displays the files that have been modified. It also shows the location of the added, removed, and updated lines.
- $git log –patch
- Generally, we can say that the --patch flag is used to display:
  - Modified files
  - The location of the lines that you added or removed
  - Specific changes that have been made.

Git log --patch

-

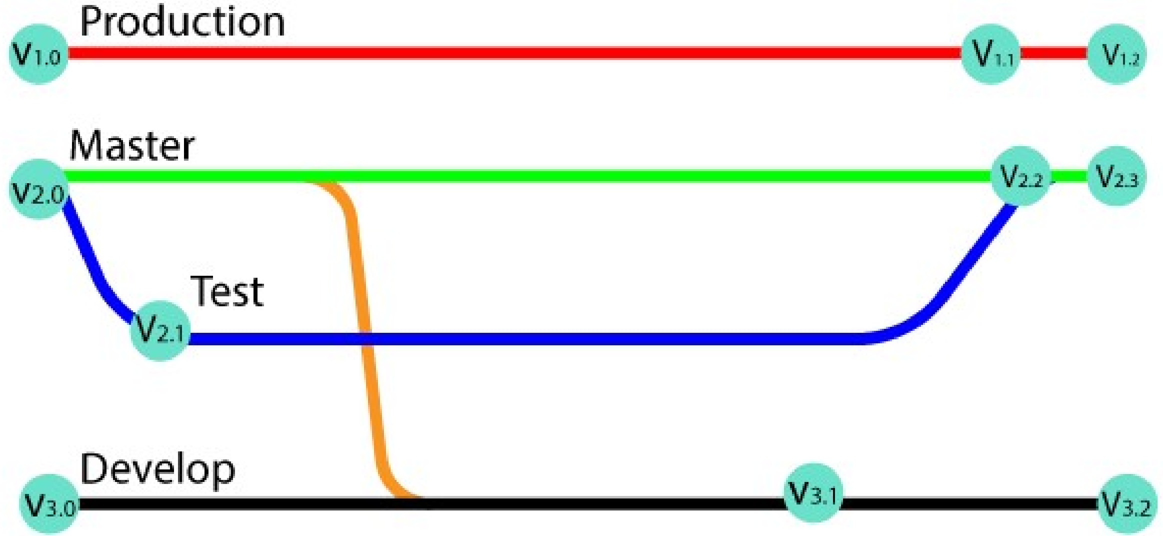
```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (master)
$ git log --patch
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Fri Nov 8 15:49:51 2019 +0530

    newfile2 Re-added

diff --git a/newfile.txt b/newfile.txt
deleted file mode 100644
index d411be5..0000000
--- a/newfile.txt
+++ /dev/null
@@ -1,2 +0,0 @@
-new file to check git Head
-NEW COMMIT IN MASTER BRANCH.
diff --git a/newfile2.txt b/newfile2.txt
deleted file mode 100644
index e69de29..0000000

commit 56afce0ea387ab840819686ec9682bb07d72add6 (tag: -d, tag: --delete, tag: --
d, tag: projectv1.1, origin/master, testing)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Oct 9 12:27:43 2019 +0530

    Added an empty newfile2

diff --git a/newfile2.txt b/newfile2.txt
new file mode 100644
index 0000000..e69de29

commit 0d5191fe05e4377abef613d2758ee0dbab7e8d95
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Sun Oct 6 17:37:09 2019 +0530
:
commit 0d3835a746b82a4dc7ca97bcfbebd4e39b26a680 (HEAD -> master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
```

Git show

- Shows one or more objects (blobs, trees, tags and commits).
- For **commits** it shows the log message and textual diff. It also presents the merge commit in a special format as produced by **git diff-tree --cc**.
- For **tags**, it shows the tag message and the referenced objects.
- For **trees**, it shows the names (equivalent to **git ls-tree** with --name-only).
- For **plain blobs**, it shows the plain contents.

Git tag

- Tags make a point as a specific point in Git history. Tags are used to mark a commit stage as relevant. We can tag a commit for future reference.



$ git tag **<tag** name**>**

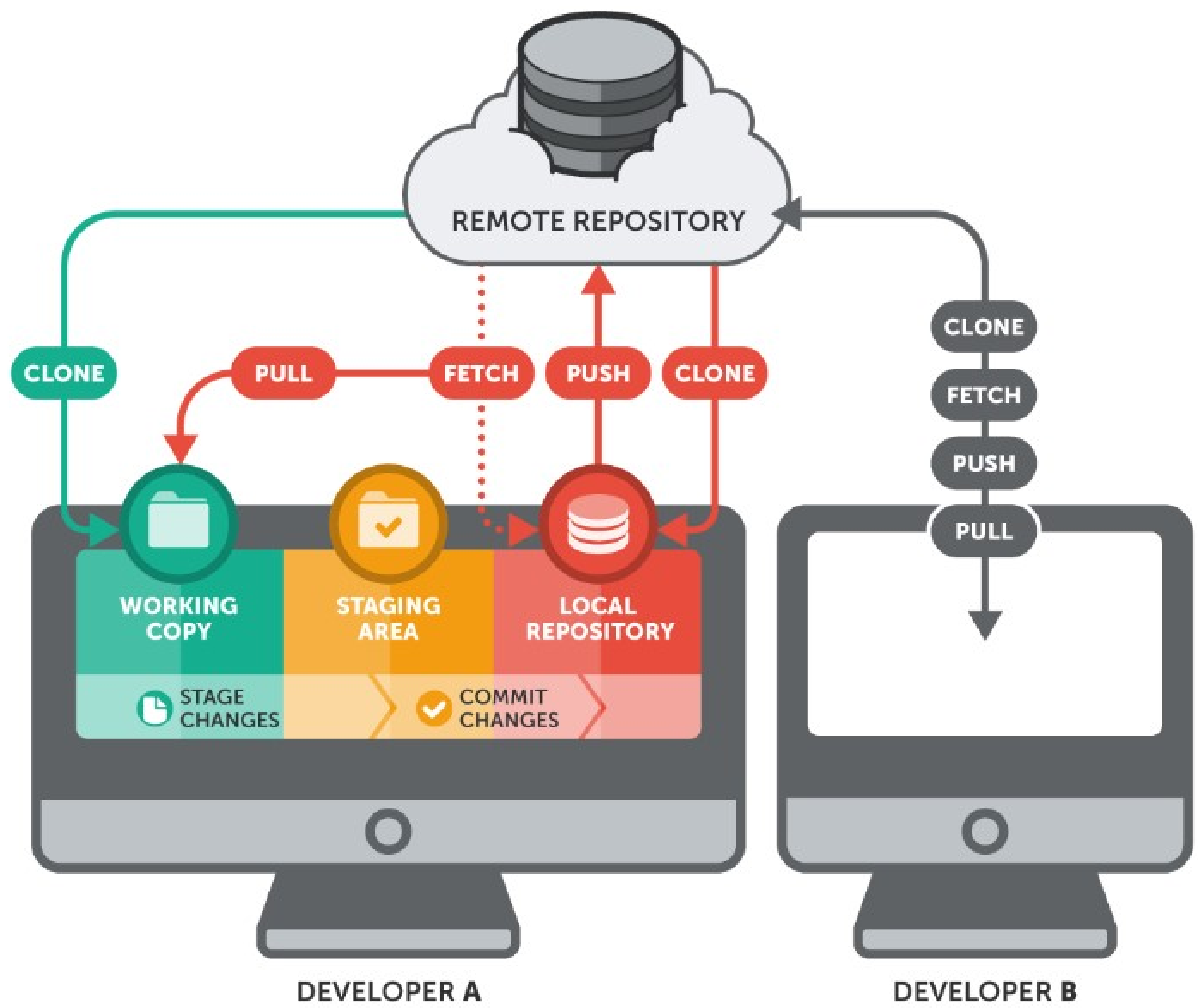
- command will mark the current status of the project.

Git List Tag:

- $git tag

Working with Remote Repository

-



Working with Remote Repository

- **A remote git repository** is nothing but a space in a remote computer which allow us to manage our project.
- **About 90% of version control related work happens in the local repository:** staging, committing, viewing the status or the log/history, etc. Moreover, if you're the only person working on your project, chances are you'll never need to set up a remote repository.

- Only when it comes to sharing data with your teammates, a remote repo comes into play. Think of it like a "file server" that you use to exchange data with your colleagues.

Working with Remote Repository

- **Distinguish between Local and Remote Repo**

Parameters:

- **Location**
- **Features**
- **Usage**
- **Creation**
- **Local / Remote Workflow**

- **Location:** Local repositories reside on the computers of team members. In contrast, remote repositories are hosted on a **server** that is accessible for all team members - most likely on the internet or on a local network.
- **Features:**
- Technically, a remote repository doesn't differ from a local one: it contains branches, commits, and tags just like a local repository.
  - However, a local repository has a working copy associated with it: a directory where some version of your project's files is checked out for you to work with.
- A remote repository doesn't have such a working directory: it only consists of the bare ".git" repository folder.

Working with Remote Repository

- **Usage**
- Actual work on your project happens only in your local repository: all modifications have to be made & committed locally.
- Then, those changes can be uploaded to a remote repository in order to share them with your team.
- Remote repositories are only thought as a means for sharing and exchanging code between developers - not for actually working on files.

- **Creation**
- You have two options to get a **local repository** onto your machine:
  - you can either create a new, empty one or
  - clone it from an existing remote repository.
- Creating a **remote repository** can also be done in two ways:
  - if you already have a local repository that you want to base it on, you can clone this local one with the "--bare" option (creates a repo with only .git folder files).

  - **$ git clone --bare my_project my_project.git**
  - In case you want to create a blank remote repository, use "git init", also with the "--bare" option.

- **Local / Remote Workflow**
- In Git, there are only a mere handful of commands that interact with a remote repository.

- Majority of work happens in the local repository. Until this point (except when we called "git clone"), we've worked exclusively with our local Git repository and never left our local computer. We were not dependent on any internet or network connection but instead worked completely offline.

Working with Remote Repository

- **Creating remote repo using GitHub:**

1. Log on to your github account. Click on + icon located on top right corner to create a new remote git repo
2. Click on New Repository menu item. You will now see a screen shown below where you need to put repository name, description (optional),
3. **select public or private and**
4. **hit Create Repository button**
5. **make our local git repo to be aware of our newly created remote git repo.** So that you can push your local git changes to remote git repo.. For this, goto local repo from git bash. And from 'git bash'command prompt issue the following command:

   git remote add origin git@github.com:kbhaskararao/CloudComp-R18-CR-2022-23.git

6. Generate ssh key for allowing write access to repo.

```
$ ssh-keygen -t ed25519 -C "kbhskarararo"
```

Name:

Paste the ssh key in: useracctname ➔ settings →SSH / GPG keys

7. Push local changes to Git

    git push -u origin master

Pushes your local repo contents to remote repo

Working with Remote Repository

Create remote_repo:

 # New local repository

git init

git add .

git commit -m "Initial commit"

 # New remote repository

git remote add origin git@github.com:kbhaskararao/new_repo #ssh

 # Now push

git push -u origin master

git remote add origin git@github.com:kbhaskararao/CC-CR-R18-2022-23

git remote add origin https://github.com/kbhaskararao/CC-CR-R18-2022-23.git

Working with Remote Repository

- git clone
- git remote add
- git push
- git fetch
- git pull

git clone: git creates a new directory to put the content of the repository.

git@github.com:kbhaskararao/DevOps.git → **url of remote repo**

git clone <remote_repo_url>

Git clone git@github.com:kbhaskararao/DevOps.git

git remote add:

if you want to set up a remote server for a repository you have on your local machine, you will use the remote add command.

Use **git remote add <name> <url>** to setup a remote server for the repository created

name → is the name reference for the server, which is typically "origin". The URL value can be grabbed off the GitHub page where you set up the empty repository.

Working with Remote Repository

-

```
MINGW64:/c/git-projects/foo-bar

Cameron@ThinkPad MINGW64 /c/git-projects/foo-bar (master)
$ git remote add origin https://github.com/cameronDz/foo-bar.git

Cameron@ThinkPad MINGW64 /c/git-projects/foo-bar (master)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 214 bytes | 214.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/cameronDz/foo-bar.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.

Cameron@ThinkPad MINGW64 /c/git-projects/foo-bar (master)
$
```

Working with Remote Repository

- **Git push:**
- **git push --set-upstream origin master**
- When that command was run in the example, it was setting the current local branch, which was "master", to a branch on the remote server that we set up, "origin", to a branch we called "master".
- **Git push –delete origin old-branch** → "old-branch" branch is deleted from the server that is called "origin".
- **Git fetch:**
- The fetch command is used to update any references to remote branches or tags.
- This means that you will bring those changes down from the remote server, and the local repository will be aware of the changes that have occurred, but the changes will not be made to the local repository branches.
- **Git pull:**
- The pull command is used to take any changes that have occurred on the remote repository, and move them into you local repository.
- When you run "git fetch", the reference of the changes on the remote come down to your local repository.
- When you run "git pull", you actually put the changes into your repository.

Working with Remote Repository

Creating remote repo:

-

DevOps – CI/CD Pipeline

- https://www.guru99.com/ci-cd-pipeline.html
- https://www.edureka.co/blog/ci-cd-pipeline/

CI/CD Pipeline:

- A CI/CD pipeline automates the process of software delivery.
- It builds code, runs tests, and helps you to safely deploy a new version of the software.
- CI/CD pipeline reduces manual errors, provides feedback to developers, and allows fast product iterations.
- CI/CD pipeline introduces automation and continuous monitoring throughout the lifecycle of a software product.
- It involves from the integration and testing phase to delivery and deployment. These connected practices are referred as **CI/CD pipeline.**

DevOps – CI/CD Pipeline

- **What is CI/CD?**

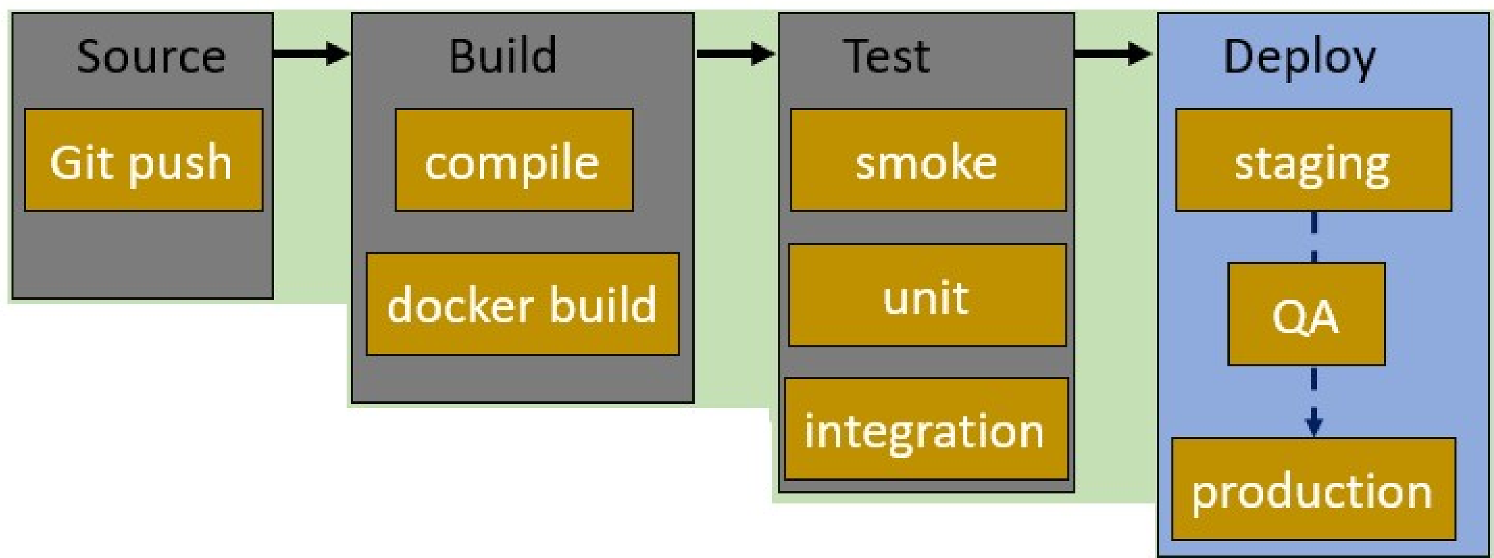
(Continuous Integration / Continuous Delivery //Continuous Deployment):

- **Continuous integration** is a software development method where members of the team can integrate their work **at least once a day**. In this method, every integration is checked by an automated build to search the error.
- **Continuous delivery** is a software engineering method in which a team develops software products in a short cycle. It ensures that software can be easily released at any time. (manual)
- **Continuous deployment** (automated) is a software engineering process in which product functionalities are delivered using **automatic deployment**. It helps testers to validate whether the codebase changes are correct, and it is stable or not.

DevOps – CI/CD Pipeline - stages

- A CI/CD pipeline is a runnable specification of the steps that any developer should perform to deliver a new version of any software. Failure in each and every stage triggers a notification via email, Slack, or other communication platforms. It enables responsible developers to know about the important issues.

Stages of CI/CD pipeline:



DevOps – CI/CD Pipeline - Stages

Source Stage:

- In the source stage, CI/CD pipeline is triggered by a code repository. Any change in the program triggers a notification to the CI/CD tool that runs an equivalent pipeline. Other common triggers include user-initiated workflows, automated schedules, and the results of other pipelines.

Build Stage:

- This is the second stage of the CI/CD Pipeline in which you ***merge the source code and its dependencies***. It is done mainly to build a runnable instance of software that you can potentially ship to the end-user.
- Programs that are written in languages like C++, Java, C, or Go language should be compiled. On the other hand, JavaScript, Python, and Ruby programs can work without the build stage.
- Failure to pass the build stage means there is a fundamental project misconfiguration, so it is better that you address such issue immediately.

Test Stage:

- Test Stage includes the execution of automated tests to validate the correctness of code and the behaviour of the software. This stage prevents easily reproducible bugs from reaching the clients. It is the responsibility of developers to write automated tests.

Deploy Stage:

- This is the last stage where your product goes live. Once the build has successfully passed through all the required test scenarios, it is ready to deploy to live server.

CI/CD Pipeline - Example

- **Source Code Control:** Host code on GitHub as a private repository. This will help you to integrate your application with major services and software.
- **Continuous integration:** Use continuous integration and delivery platform CircleCI and commit every code. When the changes notify, this tool will pull the code available in GitHub and process to build and run the test.
- **Deploy code to UAT:** Configure CircleCI to deploy your code to AWS UAT server.
- **Deploy to production:** You have to reuse continuous integration steps for deploying code to UAT.

CI/CD Pipeline - Advantages

- Builds and testing can be easily performed manually.
- It can improve the consistency and quality of code.
- Improves flexibility and has the ability to ship new functionalities.
- CI/CD pipeline can streamline communication.
- It can automate the process of software delivery.
- Helps you to achieve faster customer feedback.
- CI/CD pipeline helps you to increase your product visibility.
- It enables you to remove manual errors.
- Reduces costs and labour.
- CI/CD pipelines can make the software development lifecycle faster.
- It has automated pipeline deployment.
- A CD pipeline gives a rapid feedback loop starting from developer to client.
- Improves communications between organization employees.
- It enables developers to know which changes in the build can turn to the brokerage and to avoid them in the future.
- The automated tests, along with few manual test runs, help to fix any issues that may arise.

CI/CD Tools

Jenkins: Jenkins is an **open-source Continuous Integration server** that helps to achieve the Continuous Integration process (and not only) in an automated fashion.

- Jenkins is free and is entirely written in Java. Jenkins is a widely used application around the world that has around 300k installations and growing day by day.

Features:

- Jenkin will build and test code many times during the day.
- Automated build and test process, saving timing, and reducing defects.
- The code is deployed after every successful build and test.
- The development cycle is fast.

Bamboo: Bamboo is a continuous integration build server that performs – automatic build, test, and releases in a single place. It works seamlessly with JIRA software and Bitbucket.

Features:

- Run parallel batch tests
- Setting up Bamboo is pretty simple
- Per-environment permissions feature allows developers and QA to deploy to their environments
- Built-in Git branching and workflows. It automatically merges the branches.

CI/CD Tools

- CircleCi is a flexible CI tool that runs in any environment like a cross-platform mobile app, Python API server, or Docker cluster. This tool reduces bugs and improves the quality of the application.
- **Features:**
- Allows to select Build Environment
- Supports many languages including C++, JavaScript, NET, PHP, Python, and Ruby
- Support for Docker lets you configure a customized environment.
- Automatically cancel any queued or running builds when a newer build is triggered.

CI/CD Pipeline

Why Does the CI/CD Pipeline Matter for IT Leaders?

- CI/CD pipeline can improve reliability.
- It makes IT team more attractive to developers.
- CI/CD pipeline helps IT leaders, to pull code from version control and execute software build.
- Helps to move code to target computing environment.
- Enables project leaders to easily manage environment variables and configure for the target environment.
- Project managers can publish push application components to services like web services, database services, API services, etc.
- Providing log data and alerts on the delivery state.
- It enables programmers to verify code changes before they move forward, reducing the chances of defects ending up in production.

CI/CD Pipeline

Summary:

- A CI/CD pipeline automates the process of software delivery.
- CI/CD pipeline introduces automation and continuous monitoring throughout the lifecycle of a software product.
- Continuous integration is a software development method where members of the team can integrate their work at least once a day.
- Continuous delivery is a software engineering method in which a team develops software products in a short cycle.
- Continuous deployment is a software engineering process in which product functionalities are delivered using automatic deployment.
- There are four stages of a CI/CD pipeline 1) Source Stage, 2) Build Stage, 3) Test Stage, 4) Deploy Stage.
- Important CI/CD tools are Jenkins, Bambo, and Circle CI.
- CI/CD pipeline can improve reliability.
- CI/CD pipeline makes IT team more attractive to developers.
- Cycle time is the time taken to go from the build stage to production.
- Development frequency allows you to analyse bottlenecks you find during automation.
- Change Lead Time measures the start time of the development phase to deployment.
- Change Failure Rate focuses on the number of times development get succeeds vs. the number of times it fails.
- MTTR (Mean Time to Recovery) is the amount of time required by your team to recover from failure.
- MTTF (Mean Time to Failure) measures the amount of time between fixes and outages.