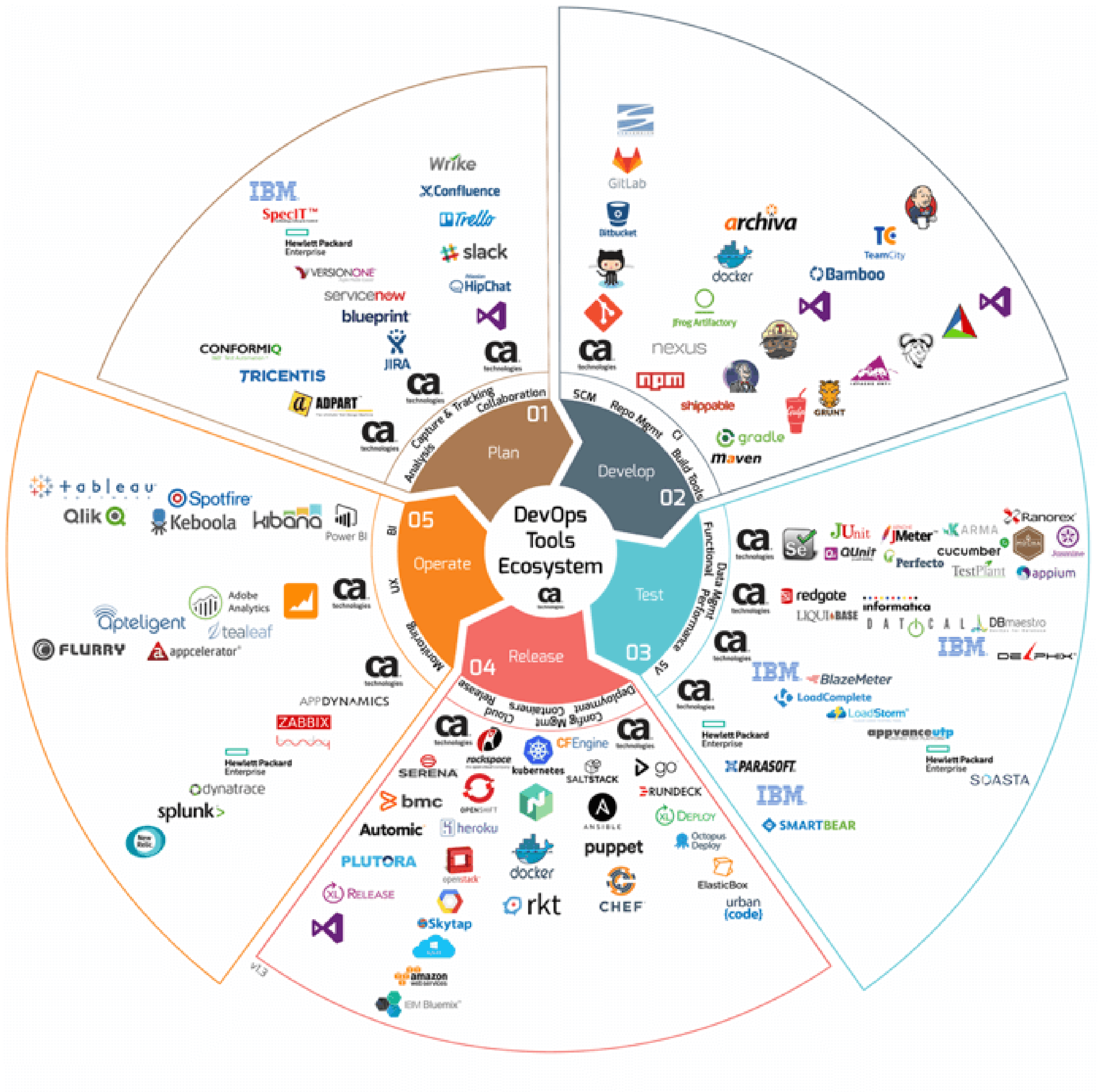
APEX INSTITUTE of TECHNOLOGY
Discover - Learn - Empower

NAAC GRADE A+ ACCREDITED UNIVERSITY

Pre-Earned Credit Program

Introduction to DevOps

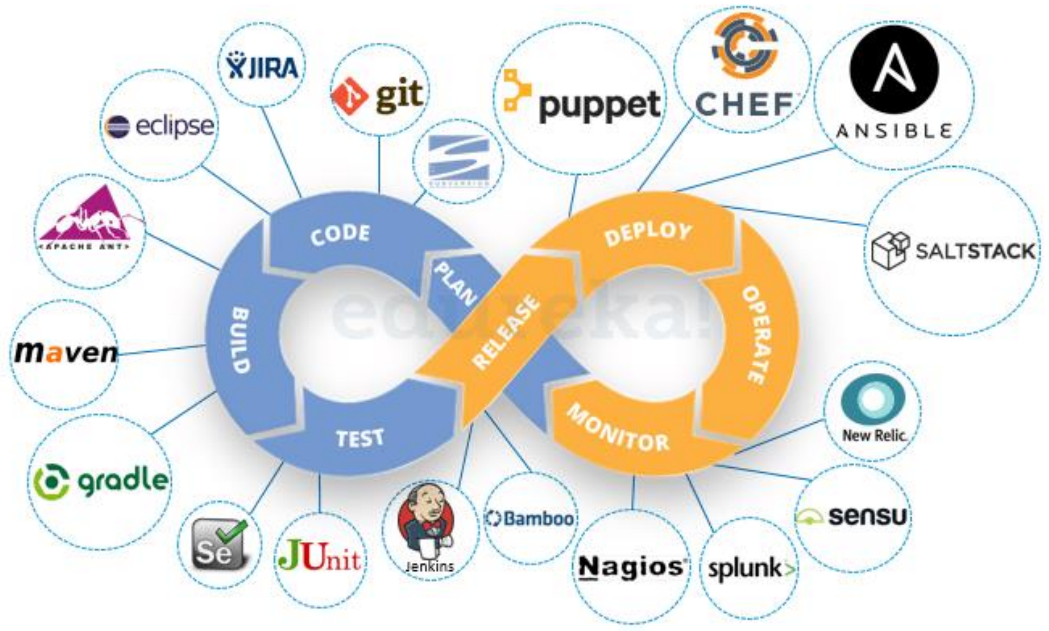


Contents

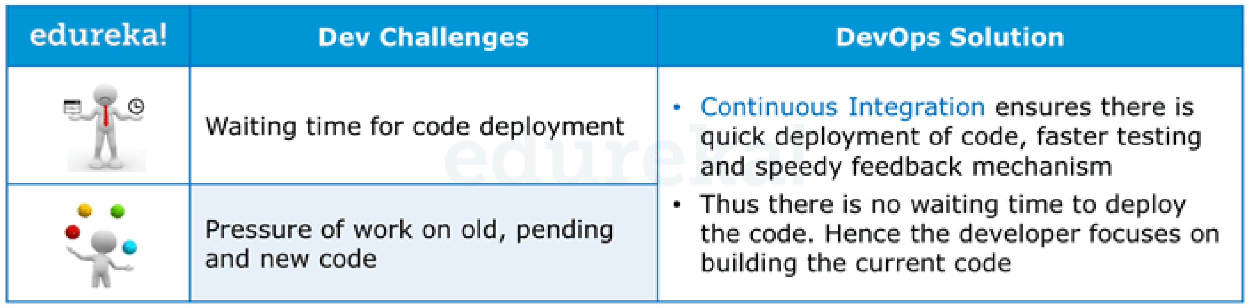
What is DevOps?

According to the DevOps culture, a single group of Engineers (developers, system admins, QA's. Testers etc. turned into DevOps Engineers) has end to end responsibility of the Application (Software) right from gathering the requirement to development, to testing, to infrastructure deployment, to application deployment and finally monitoring & gathering feedback from the end users, then again implementing the changes.

This is a never ending cycle and the logo of DevOps makes perfect sense to me. Just look at the above diagram – What could have been a better symbol than infinity to symbolize DevOps?



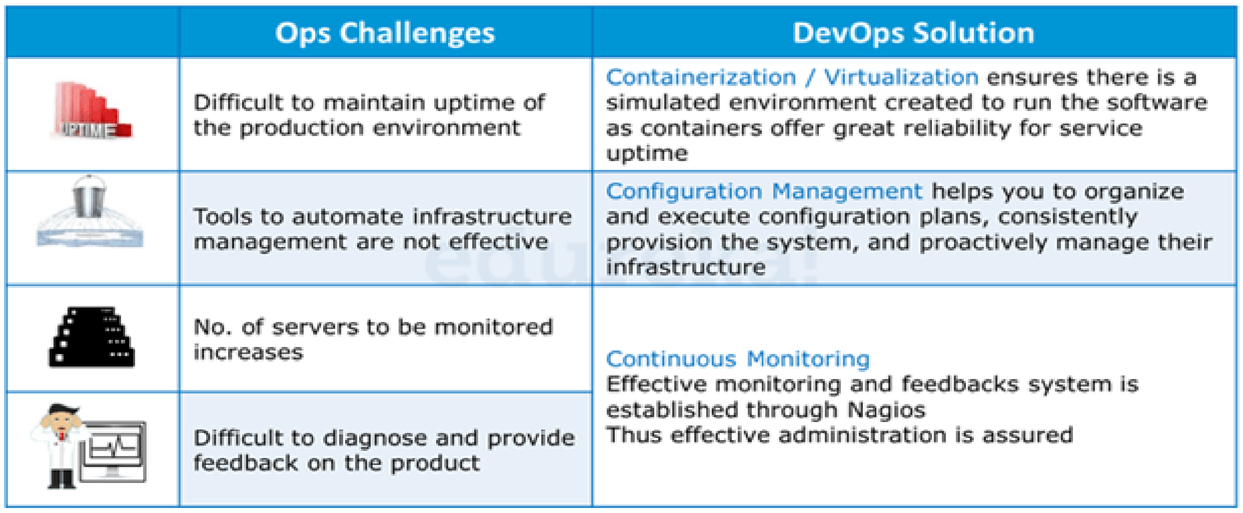
Now let us see how DevOps takes care of the challenges faced by Development and Operations. Below table describes how DevOps addresses Dev Challenges.



edureka!	Dev Challenges	DevOps Solution
	Waiting time for code deployment	• Continuous Integration ensures there is quick deployment of code, faster testing and speedy feedback mechanism • Thus there is no waiting time to deploy the code. Hence the developer focuses on building the current code
	Pressure of work on old, pending and new code	

DevOps Tutorial Table 1 – Above table states how DevOps solves Dev Challenges

Going further, below table describes how DevOps addresses Ops Challenges.



	Ops Challenges	DevOps Solution
	Difficult to maintain uptime of the production environment	Containerization / Virtualization ensures there is a simulated environment created to run the software as containers offer great reliability for service uptime
	Tools to automate infrastructure management are not effective	Configuration Management helps you to organize and execute configuration plans, consistently provision the system, and proactively manage their infrastructure
	No. of servers to be monitored increases	Continuous Monitoring Effective monitoring and feedbacks system is established through Nagios Thus effective administration is assured
	Difficult to diagnose and provide feedback on the product	

DevOps Tutorial Table 2 – Above table states how DevOps solves Ops Challenges

However, to implement DevOps. To expedite and actualize DevOps process apart from culturally accepting it, one also needs various DevOps tools like Puppet, Jenkins, GIT, Chef, Docker, Selenium, AWS etc. to achieve automation at various stages which helps in achieving Continuous Development, Continuous Integration, Continuous Testing, Continuous Deployment, Continuous Monitoring to deliver a quality software to the customer at a very fast pace.

So since what is DevOps, let us have a look at the history of DevOps.

History of DevOps

Before DevOps, we had to know about the Software Development Lifecycle (SDLC), its Life Cycle, Usability of SDLC, **Stages of the SDLC, Traditional vs modern SDLC methodologies,** systematic approach to developing software two approaches for software development namely the Waterfall and the Agile.

The Software Development Lifecycle (SDLC): An Introduction

The Software Development Lifecycle (SDLC) describes the systematic approach to developing software.

Software development lifecycle

The SDLC helps to ensure high quality software is built and released to end-users quickly and at an optimized cost. How you determine the quality of your software might vary, but general measurements include:

- The robustness of the software functionality
- Overall performance
- Security

- Ultimately, the user experience

Regardless of which software development you subscribe to—Agile, Waterfall, or other variations—this lifecycle can apply.

Who uses the SDLC

Not so long ago, Watt S. Humphrey, known as the father of quality in software, remarked:

"Every business is a software business."

More recently, Microsoft CEO Satya Nadella repeated the quote: "Every company is a software company".

Of course, we can point to many specific technology companies who develop software. If there's an app, someone developed it.

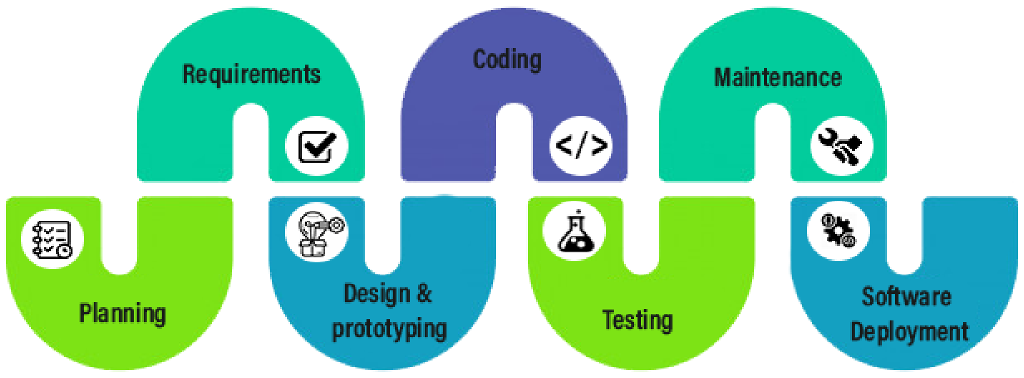
But business organizations that aren't "in software" rely on on software and technology to do business (which is to say, all of them). These organizations will need to adapt at least some off-the-shelf solutions, likely to tweak software to align and optimize with their unique business operations.

That's why people beyond developers or engineers should understand the SDLC approach: many stakeholders might be involved in various stages. Plus, cross-functional teams might adopt the SDLC to collaborate on Agile- and DevOps-based projects. Following modern SDLC practices and frameworks can significantly improve the software development process.

Stages of the SDLC

The SDLC follows a series of phases involved in software development. Depending on the SDLC framework, these phases may be adopted sequentially or in parallel. (More on this below.)

The SDLC workflows may involve repeated transitions or iterations across the phases before reaching the final phase.



Phase 1: Requirement Analysis

The initial stage of the SDLC involves stakeholders from tech, business, and leadership segments of the organization. In this initial state, you'll:

- Analyze and translate business questions into engineering problems by considering a variety of factors: cost, performance, functionality, and risk.
- Evaluate he broad scope of the project and then identify available resources.
- Consider project opportunities and risks across the technical and business aspect for every decision choice in each SDLC phase.

This stage may continue for a prolonged period and includes provision for strategic changes as the SDLC evolves.

Phase 2: Feasibility Study

During this stage, evaluate the requirements for feasibility. Not every single requirement will be feasible for your current scope. The goal of this stage is to quantify the opportunities and risk of addressing the agreed requirements with the variety of resources and strategies you have available.

The feasibility study evaluates the following key aspects, among others:

- **Economic:** Is it financially viable to invest in the project based on the available resources?
- **Legal:** What is the scope of regulations and the organization's capacity to guarantee compliance?
- **Operational:** Can we satisfy the requirements within scope definition according to the proposed operational framework and workflows?
- **Technical:** What is the availability of technology and HR resources to support the SLDC process?
- **Schedule:** Can we finish the project in time?

Executive decision makers should answer and document these questions and study them carefully—before proceeding with the software design and implementation process.

Phase 3: Architectural Design

Next, the appropriate technical and business stakeholders document, review, and evaluate the design specifications and choices against the risk, opportunities, practical modalities, and constraints.

In this phase, you'll have technical documentation that specifies:

- Systems architecture
- Configurations
- Data structure
- Resource procurement model

Desired output can include prototypes, pseudocode, minimal viable products (MVPs) and/or architecture reports and diagrams that include the necessary technology details:

- High-level design details include the desired functionality of software and system modules.
- Low-level design details can include the functional logic, interface details, dependency issues, and errors.

Phase 4: Software Development

Implementation follows the design phase. Several independent teams and individuals collaborate on feature development and coding activities. Frequently, individual developers will build their own codebase within the development environment, then merge it with the collaborating teams in a common build environment.

While the requirements analysis and design choices are already defined, feedback from the development teams is reviewed for potential change in direction of the design strategies.

This is the longest process in the SDLC pipeline and it assists subsequent phases of software testing and deployment.

Phase 5: Testing

In this phase, you'll use testing to:

- Investigate the performance of the software
- Discover and identify potential issues to fix or address

Testing teams develop a test plan based on the predefined software requirements. The testing plan should:

- Identify the resources available for testing
- Provide instructions and assignments for testers

- Select types of tests to be conducted
- Determine what to report to technical executives and decision makers

Testers often work collectively with development teams and rework the codebase to improve test results.

It is very common for teams to repeat the development and testing phases several times, before moving onto the final stages of deploying and and releasing the software.

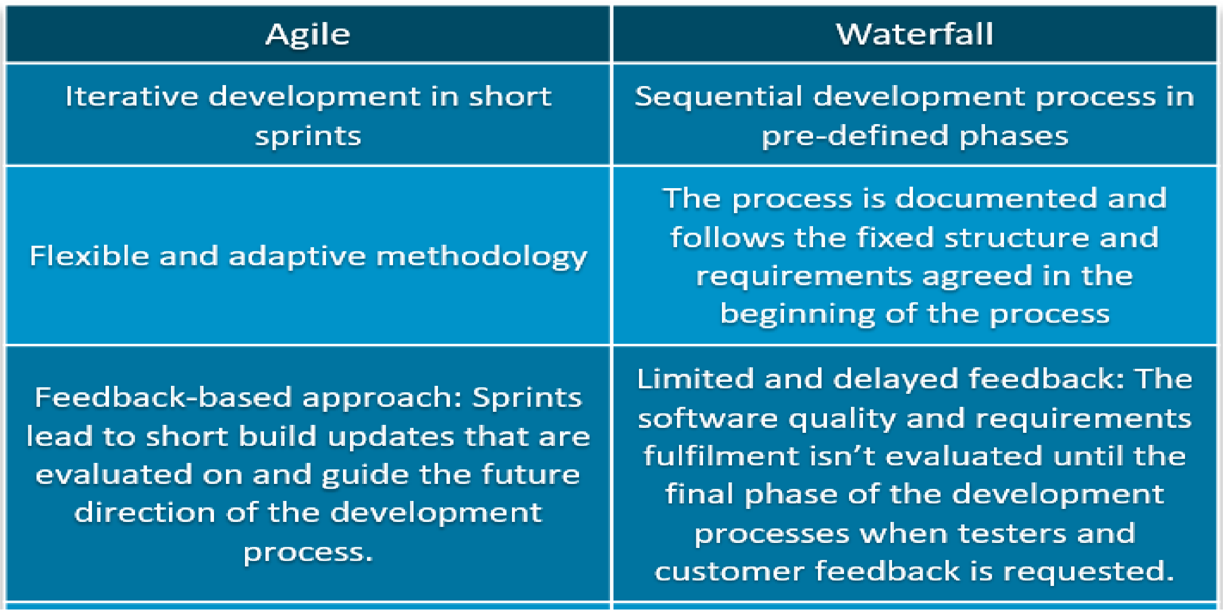
Phase 6: Deployment

You’ve reached the final phase of the SDLC pipeline when your finished product has passed the necessary tests. Now, make it available for release to end users in the real environment. Several procedures and preparation activities are involved before a software product can be shipped, including:

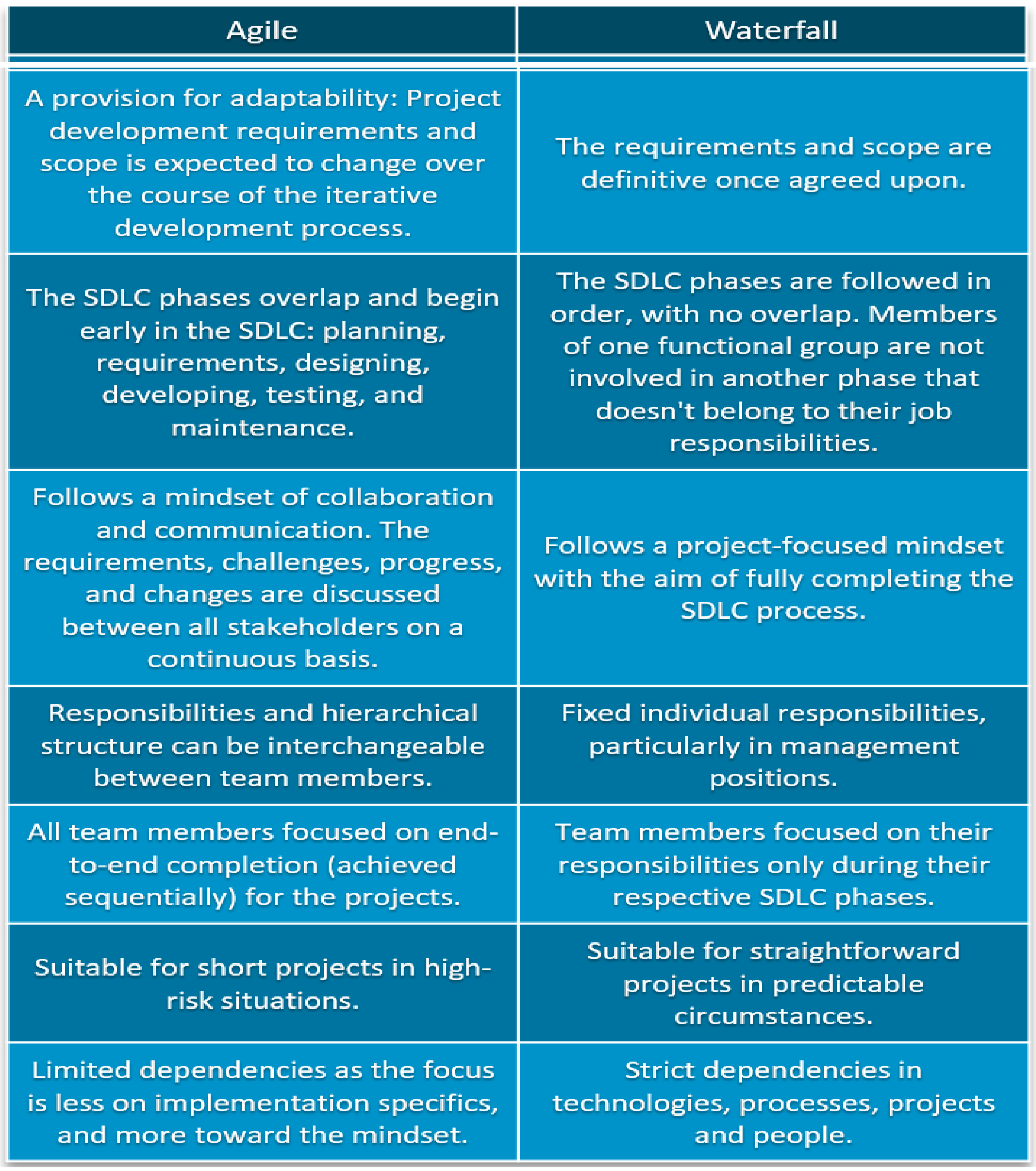
- Documentation
- Transferring ownership and licensing,
- Deploying and installing the product on customer systems

Traditional vs modern SDLC methodologies

With traditional SDLC methodologies, such as Waterfall, these phases are performed independently in series by disparate teams. Under the Agile methodology, these phases are performed in short, iterative, incremental sprints.



Agile	Waterfall
Iterative development in short sprints	Sequential development process in pre-defined phases
Flexible and adaptive methodology	The process is documented and follows the fixed structure and requirements agreed in the beginning of the process
Feedback-based approach: Sprints lead to short build updates that are evaluated on and guide the future direction of the development process.	Limited and delayed feedback: The software quality and requirements fulfilment isn’t evaluated until the final phase of the development processes when testers and customer feedback is requested.



Agile	Waterfall
A provision for adaptability: Project development requirements and scope is expected to change over the course of the iterative development process.	The requirements and scope are definitive once agreed upon.
The SDLC phases overlap and begin early in the SDLC: planning, requirements, designing, developing, testing, and maintenance.	The SDLC phases are followed in order, with no overlap. Members of one functional group are not involved in another phase that doesn't belong to their job responsibilities.
Follows a mindset of collaboration and communication. The requirements, challenges, progress, and changes are discussed between all stakeholders on a continuous basis.	Follows a project-focused mindset with the aim of fully completing the SDLC process.
Responsibilities and hierarchical structure can be interchangeable between team members.	Fixed individual responsibilities, particularly in management positions.
All team members focused on end-to-end completion (achieved sequentially) for the projects.	Team members focused on their responsibilities only during their respective SDLC phases.
Suitable for short projects in high-risk situations.	Suitable for straightforward projects in predictable circumstances.
Limited dependencies as the focus is less on implementation specifics, and more toward the mindset.	Strict dependencies in technologies, processes, projects and people.

(*Agile vs Waterfall SDLC methodologies*)

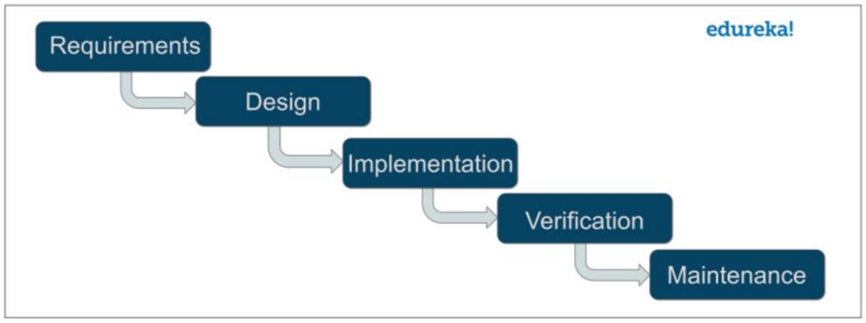
An SDLC pipeline and framework can be as varied as the number of organizations adopting them—virtually every company tries to adopt a strategy that works best for their organization.

In today's era of software development, however, these stages are not always followed sequentially. Modern SDLC frameworks such as DevOps and Agile encourage cross-functional organizations to share responsibilities across these phases conducted in parallel.

For instance, the DevOps SDLC framework encourages Devs, Ops, and QA personnel to work together for continuous development, testing and deployment activities. Additionally, the testing procedure is shifted left and early in the SDLC pipeline so that software defects are identified before it's too late to fix them.

Waterfall Model

- The waterfall model is a software development model that is pretty straight forward and linear. This model follows a top-down approach.
- This model has various starting with **Requirements gathering and analysis**. This is the phase where you get the requirements from the client for developing an application. After this, you try to analyze these requirements.



- The next phase is the **Design** phase where you prepare a blueprint of the software. Here, you think about how the software is actually going to look like.
- Once the design is ready, you move further with the **Implementation** phase where you begin with the coding for the application. The team of developers works together on various components of the application.
- Once you complete the application development, you test it in the **Verification** phase. There are various tests conducted on the application such as unit testing, integration testing, performance testing, etc.
- After all the tests on the application are completed, it is deployed onto the production servers.
- At last, comes the **Maintenance** phase. In this phase, the application is monitored for performance. Any issues related to the performance of the application are resolved in this phase.

Advantages of the Waterfall Model:

- Simple to understand and use
- Allows for easy testing and analysis
- Saves a significant amount of time and money

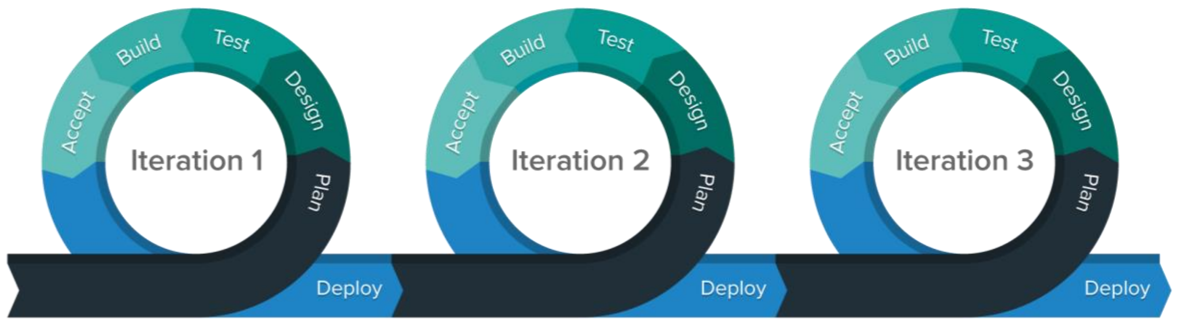
- Good for small projects if all requirements are clearly defined
- Allows for departmentalization & managerial control

Disadvantages of Waterfall Model:

- Risky and uncertain
- Lack of visibility of the current progress
- Not suitable when the requirements keep changing
- Difficult to make changes to the product when it is in the testing phase
- The end product is available only at the end of the cycle
- Not suitable for large and complex projects

Agile Methodology

Agile Methodology is an iterative based software development approach where the software project is broken down into various iterations or sprints. Each iteration has phases like the waterfall model such as Requirements Gathering, Design, Development, Testing, and Maintenance. The duration of each iteration is generally 2-8 weeks.



Agile Process

- In Agile, a company releases the application with some high priority features in the first iteration.
- After its release, the end-users or the customers give you feedback about the performance of the application.
- Then you make the necessary changes into the application along with some new features and the application is again released which is the second iteration.
- You repeat this entire procedure until you achieve the desired software quality.

Advantages of Agile Model

- It adaptively responds to requirement changes favourably
- Fixing errors early in the development process makes this process more cost-effective
- Improves the quality of the product and makes it highly error-free
- Allows for direct communication between people involved in software project
- Highly suitable for large & long-term projects

- Minimum resource requirements & very easy to manage

Disadvantages of Agile Model

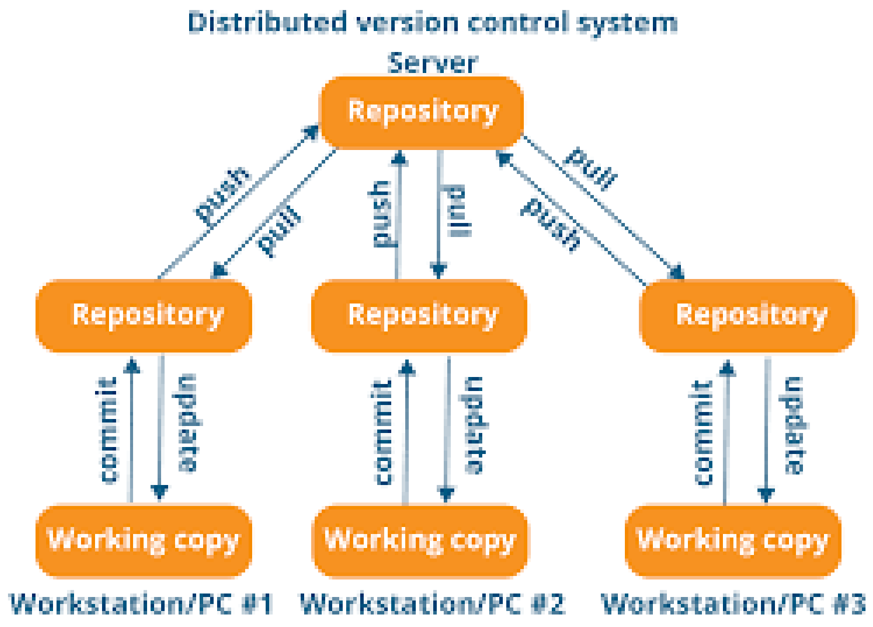
- Highly dependent on clear customer requirements
- Quite Difficult to predict time and effort for larger projects
- Not suitable for complex projects
- Lacks documentation efficiency
- Increased maintainability risks

As mentioned earlier, the various stages such as continuous development, continuous integration, continuous testing, continuous deployment, and continuous monitoring constitute the DevOps Life cycle. Now let us have a look at each of the stages of DevOps life cycle one by one.

Stage – 1: Continuous Development

Tools Used: Git, SVN, Mercurial, CVS

Process Flow:



- This is the phase that involves ‘planning‘ and ‘coding‘ of the software. You decide the project vision 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 a number of tools for maintaining the code.

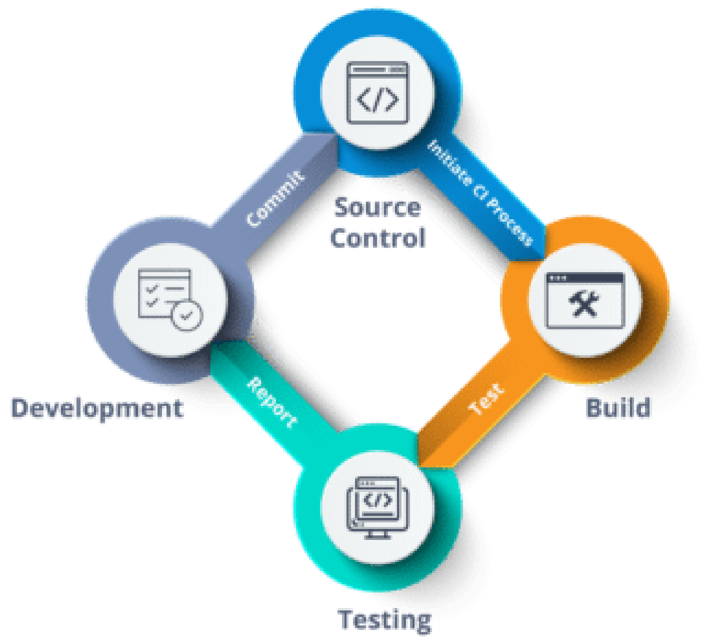
- The code can be in any language, but you maintain it by using Version Control tools. This process of maintaining the code is known as Source Code Management.
- After the code is developed, then you move to the Continuous Integration phase.

Stage – 2: Continuous Integration

Tools: Jenkins, TeamCity, Travis

Process Flow:

- This stage is the core of the entire DevOps life cycle. It is a practice in which the developers require to commit changes to the source code more frequently. This may be either on a daily or weekly basis.
- You then build every commit and this allows early detection of problems if they are present. Building code not only involves compilation but it also includes code review, unit testing, integration testing, and packaging.

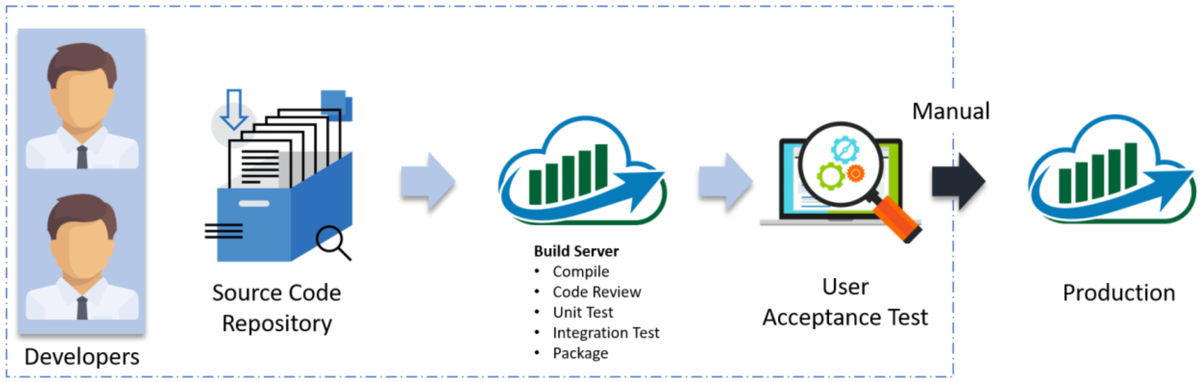


- The code supporting new functionality is ***continuously integrated*** with the existing code. Since there is a continuous development of software, you need to integrate the updated code continuously as well as smoothly with the systems to reflect changes to the end-users.
- In this stage, you use the tools for building/ packaging the code into an executable file so that you can forward it to the next phases.

Stage – 3: Continuous Testing

Tools: Jenkins, Selenium TestNG, JUnit

Process Flow:



- This is the stage where you test the developed software continuously for bugs using automation testing tools. These tools allow QAs to test multiple code-bases thoroughly in parallel to ensure that there are no flaws in the functionality. In this phase, you can use Docker Containers for simulating the test environment.

- ***Selenium*** is used for automation testing, and the reports are generated by ***TestNG***. You can automate this entire testing phase with the help of a Continuous Integration tool called Jenkins.

- Suppose you have written a selenium code in Java to test your application. Now you can build this code using ant or maven. Once you build the code, you then test it for User Acceptance Testing (UAT). This entire process can be automated using ***Jenkins***.

Stage – 4: Continuous Deployment

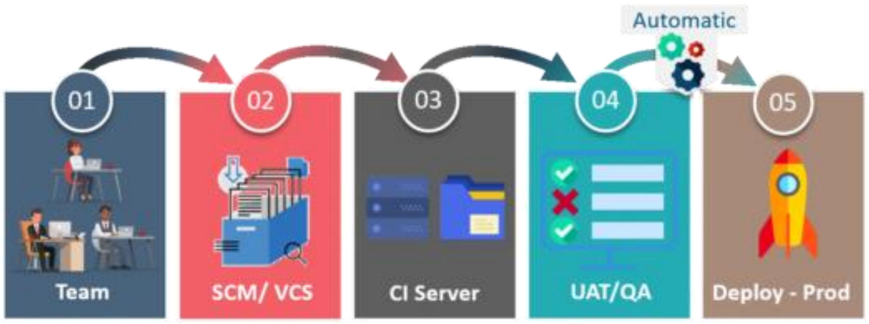
Tools Used:

Configuration Management – Chef, Puppet, Ansible

Containerization – Docker, Vagrant

Process Flow:

- This is the stage where you deploy the code on the production servers. It is also important to ensure that you correctly deploy the code on all the servers. Before moving on, let us try to understand a few things about Configuration management and ***Containerization tools***. These set of tools here help in achieving Continuous Deployment (CD).



- ***Configuration Management*** is the act of establishing and maintaining consistency in an application's functional requirements and performance. Let me put this in easier words, it is the act of releasing deployments to servers, scheduling updates on all servers and most importantly keeping the configurations consistent across all the servers.

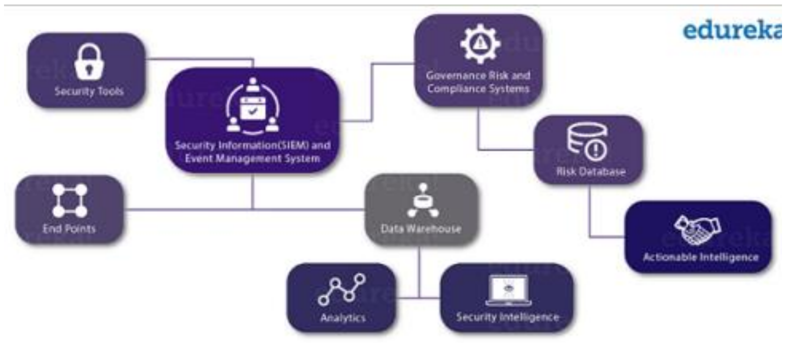
- Containerization tools also play an equally crucial role in the deployment stage. The containerization tools help produce consistency across Development, Test, Staging as well as Production environments. Besides this, they also help in scaling-up and scaling-down of instances swiftly.

Stage – 5: Continuous Monitoring

Tools Used: Splunk, ELK Stack, Nagios, New Relic

Process Flow:

- This is a very critical stage of the DevOps life cycle where you continuously monitor the performance of your application. Here you record vital information about the use of the software. You then process this information to check the proper functionality of the application. You resolve system errors such as low memory, server not reachable, etc in this phase.



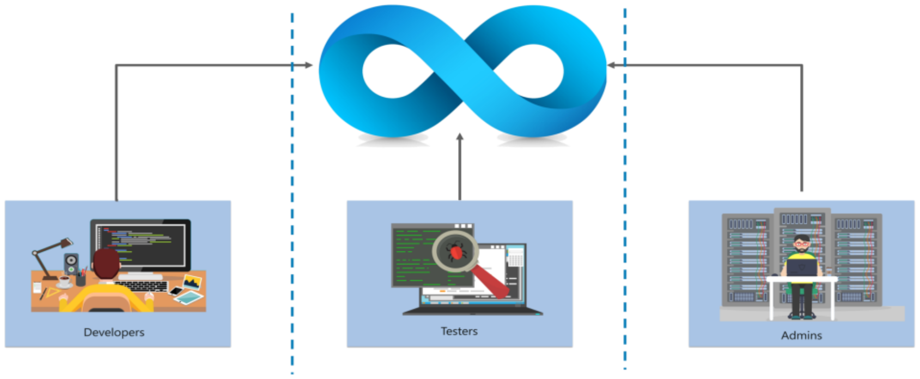
- This practice involves the participation of the Operations team who will monitor the user activity for bugs or any improper behavior of the system. The Continuous Monitoring tools help you monitor the application's performance and the servers closely and also enable you to check the health of the system proactively.

Lastly, we will discuss who exactly is a DevOps Engineer.

Who is a DevOps Engineer?

DevOps Engineer is somebody who understands the Software Development Lifecycle and has the outright understanding of various automation tools for developing digital pipelines (CI/ CD pipelines).

DevOps Engineer works with developers and the IT staff to oversee the code releases. They are either developers who get interested in deployment and network operations or sysadmins who have a passion for scripting and coding and move into the development side where they can improve the planning of test and deployment.



Git – Commands And Operations In Git

The motive of Git is to manage a project or a set of files as they change over time. Git stores this information in a data structure called a Git repository. The repository is the core of Git.

How do I see my GIT repository?

To be very clear, a Git repository is the directory where all of your project files and the related metadata resides.

Git records the current state of the project by creating a tree graph from the index. It is usually in the form of a Directed Acyclic Graph (DAG).

Before you go ahead, check out this video on Git tutorial to have better in-sight.

- Create a "repository" (project) with a git hosting tool (like Bitbucket)
- Copy (or clone) the repository to your local machine
- Add a file to your local repo and "commit" (save) the changes
- "Push" your changes to your master branch

- Make a change to your file with a git hosting tool and commit
- "Pull" the changes to your local machine
- Create a "branch" (version), make a change, commit the change
- Open a "pull request".
- "Merge" your branch to the master branch

Difference between Git Shell and Git Bash?

Git Bash and Git Shell are two different command line programs which allow you to interact with the underlying Git program. Bash is a Linux-based command line while Shell is a native Windows command line.

A few Operations & Commands

Some of the basic operations in Git are:

1. Initialize
2. Add
3. Commit
4. Pull
5. Push

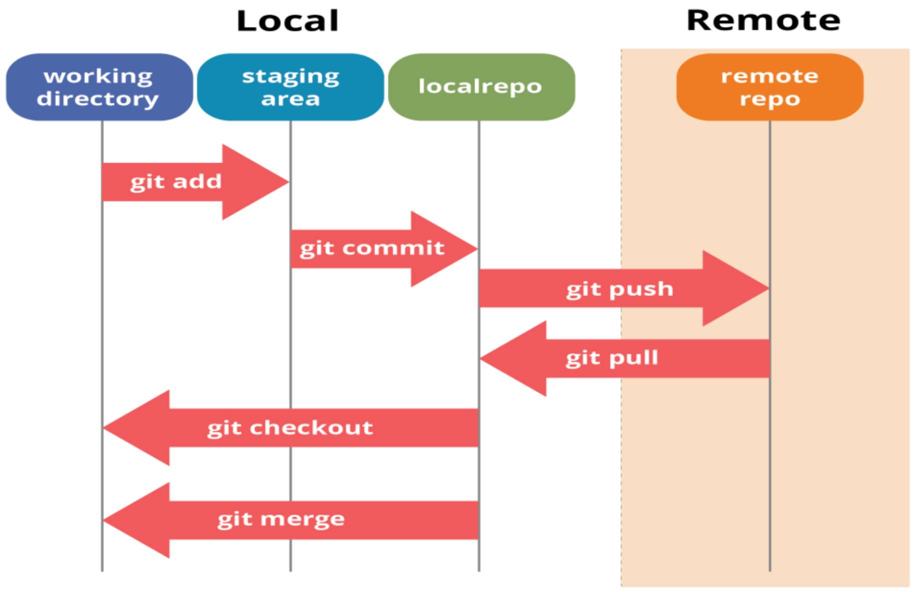
Some advanced Git operations are:

1. Branching
2. Merging
3. Rebasing

Let me first give you a brief idea about how these operations work with the Git repositories. Take a look at the architecture of Git below:

If you understand the above diagram well and good, but if you don't, you need not worry, I will be explaining these operations in this Git Tutorial one by one. Let us begin with the basic operations.

You need to install Git on your system first. If you need help with the installation, ***click here***.

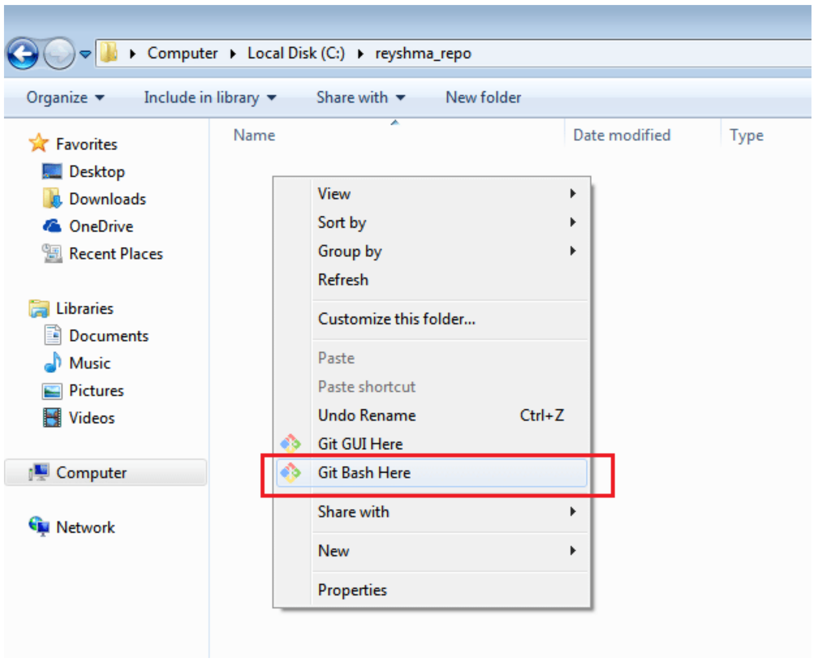


What is Git Bash used for?

This Git Bash Tutorial focuses on the commands and operations that can be used on Git Bash.

How do I navigate Git Bash?

After installing Git in your Windows system, just open your folder/directory where you want to store all your project files; right click and select '***Git Bash here***'.

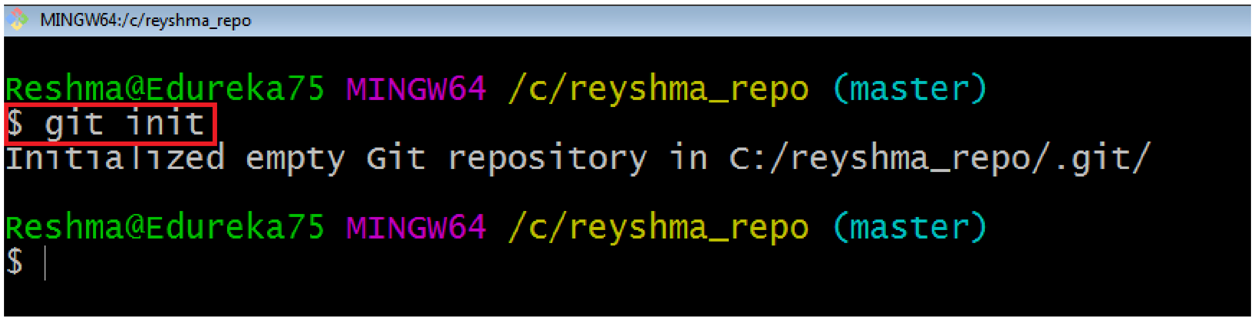


This will open up Git Bash terminal where you can enter commands to perform various Git operations.

Now, the next task is to initialize your repository.

Initialize

In order to do that, we use the command **git init.** Please refer to the below screenshot.



```
MINGW64:/c/reyshma_repo

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git init
Initialized empty Git repository in C:/reyshma_repo/.git/

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

git init creates an empty Git repository or re-initializes an existing one. It basically creates a **.git** directory with sub directories and template files. Running a **git init** in an existing repository will not overwrite things that are already there. It rather picks up the newly added templates.

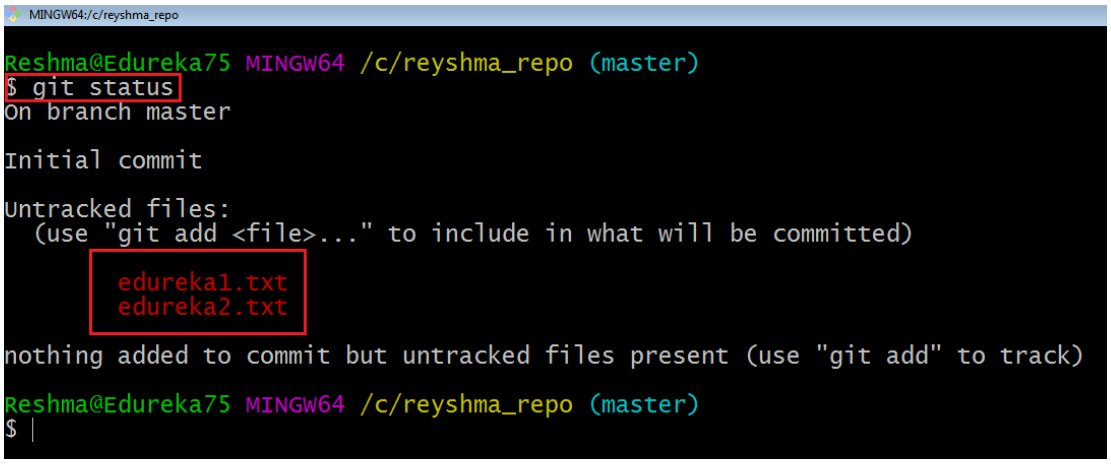
Now that my repository is initialized, let me create some files in the directory/repository. For e.g. I have created two text files namely *edureka1.txt* and *edureka2.txt*.

Let's see if these files are in my index or not using the command **git status**. The index holds a snapshot of the content of the working tree/directory, and this snapshot is taken as the contents for the next change to be made in the local repository.

Git status

The **git status** command lists all the modified files which are ready to be added to the local repository.

Let us type in the command to see what happens:



```
MINGW64:/c/reyshma_repo

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        edureka1.txt
        edureka2.txt

nothing added to commit but untracked files present (use "git add" to track)

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

This shows that I have two files which are not added to the index yet. This means I cannot commit changes with these files unless I have added them explicitly in the index.

Add

This command updates the index using the current content found in the working tree and then prepares the content in the staging area for the next commit.

Thus, after making changes to the working tree, and before running the **commit** command, you must use the **add** command to add any new or modified files to the index. For that, use the commands below:

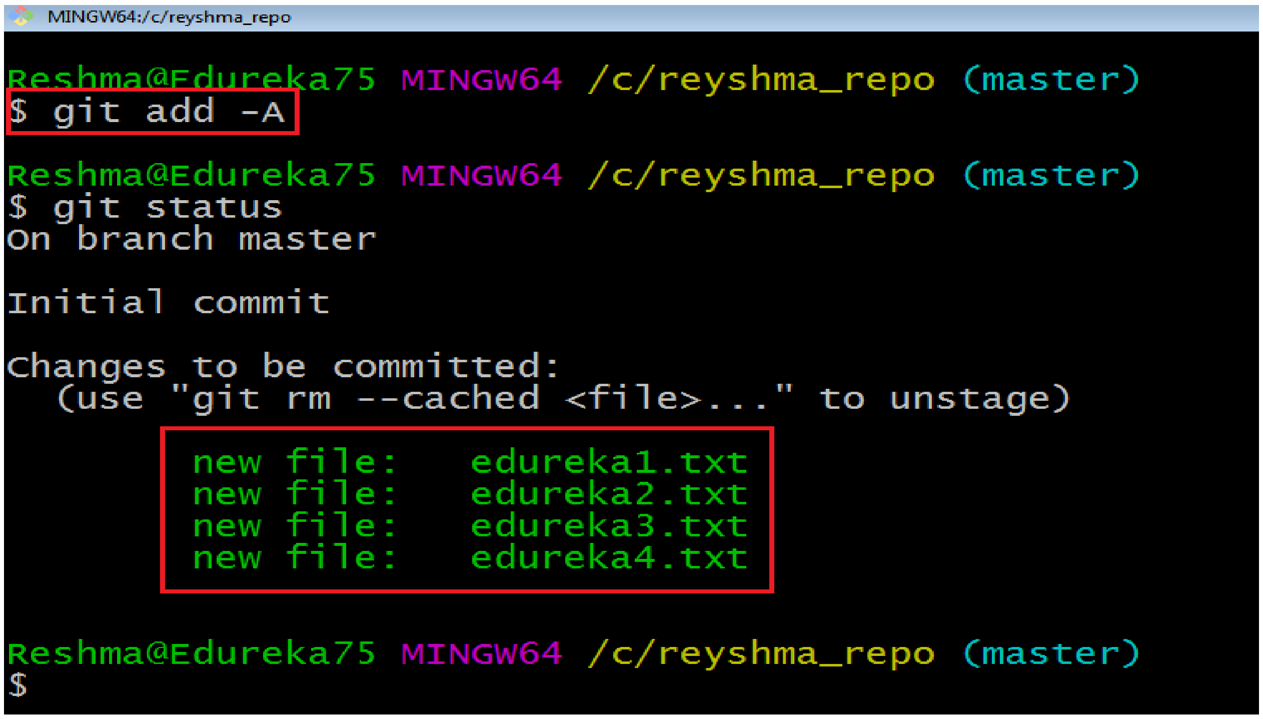
git add <directory>

or

git add <file>

Let me demonstrate the **git add** for you so that you can understand it better.

I have created two more files *edureka3.txt* and *edureka4.txt*. Let us add the files using the command **git add -A**. This command will add all the files to the index which are in the directory but not updated in the index yet.

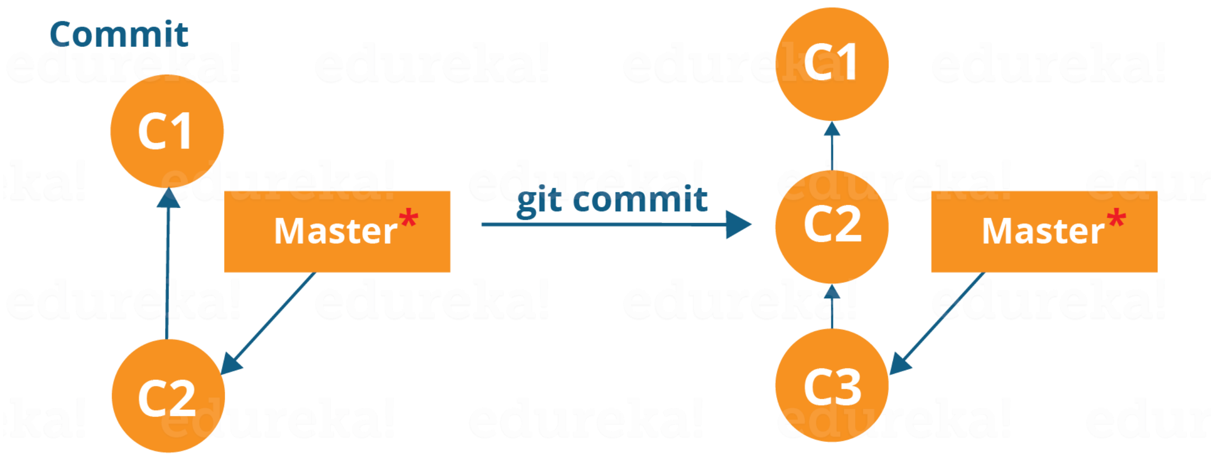


```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git add -A

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   edureka1.txt
        new file:   edureka2.txt
        new file:   edureka3.txt
        new file:   edureka4.txt


Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$
```

Now that the new files are added to the index, you are ready to commit them.

Commit

It refers to recording snapshots of the repository at a given time. Committed snapshots will never change unless done explicitly. Let me explain how commit works with the diagram below:



Here, C1 is the initial commit, i.e. the snapshot of the first change from which another snapshot is created with changes named C2. Note that the master points to the latest commit.

Now, when I commit again, another snapshot C3 is created and now the master points to C3 instead of C2.

Git aims to keep commits as lightweight as possible. So, it doesn't blindly copy the entire directory every time you commit; it includes commit as a set of changes, or "delta" from one version of the repository to the other. In easy words, it only copies the changes made in the repository.

You can commit by using the command below:

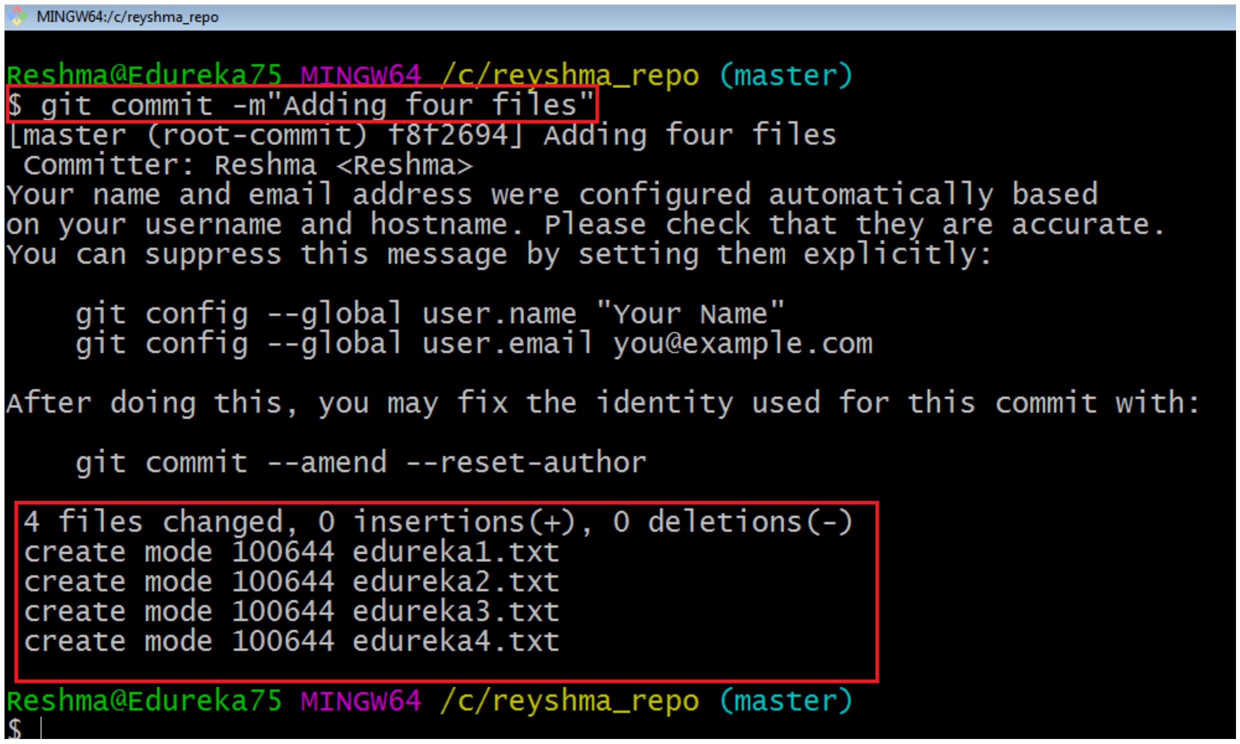
git commit

This will commit the staged snapshot and will launch a text editor prompting you for a commit message.

Or you can use:

git commit -m "<message>"

Let's try it out.


```
MINGW64:/c/reyshma_repo

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git commit -m"Adding four files"
[master (root-commit) f8f2694] Adding four files
 Committer: Reshma <Reshma>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 4 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 edureka1.txt
 create mode 100644 edureka2.txt
 create mode 100644 edureka3.txt
 create mode 100644 edureka4.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

As you can see above, the **git commit** command has committed the changes in the four files in the local repository.

Now, if you want to commit a snapshot of all the changes in the working directory at once, you can use the command below:

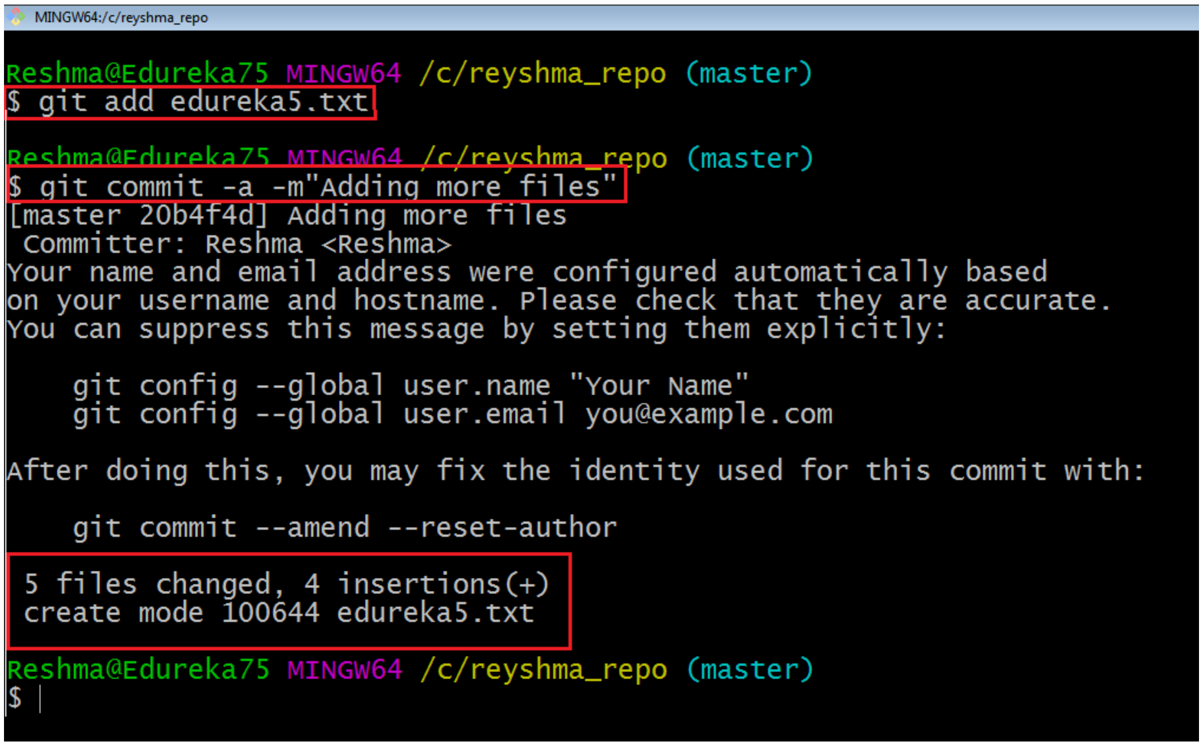
git commit -a

I have created two more text files in my working directory viz. *edureka5.txt* and *edureka6.txt* but they are not added to the index yet.

I am adding edureka5.txt using the command:

git add edureka5.txt

I have added *edureka5.txt* to the index explicitly but not *edureka6.txt* and made changes in the previous files. I want to commit all changes in the directory at once. Refer to the below snapshot.



```
MINGW64:/c/reyshma_repo

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git add edureka5.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git commit -a -m"Adding more files"
[master 20b4f4d] Adding more files
 Committer: Reshma <Reshma>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 5 files changed, 4 insertions(+)
 create mode 100644 edureka5.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

This command will commit a snapshot of all changes in the working directory but only includes modifications to tracked files i.e. the files that have been added with **git add** at some point in their history. Hence, *edureka6.txt* was not committed because it was not added to the index yet. But changes in all previous files present in the repository were committed, i.e. *edureka1.txt*, *edureka2.txt*, *edureka3.txt*, *edureka4.txt* and *edureka5.txt*.
Now I have made my desired commits in my local repository.

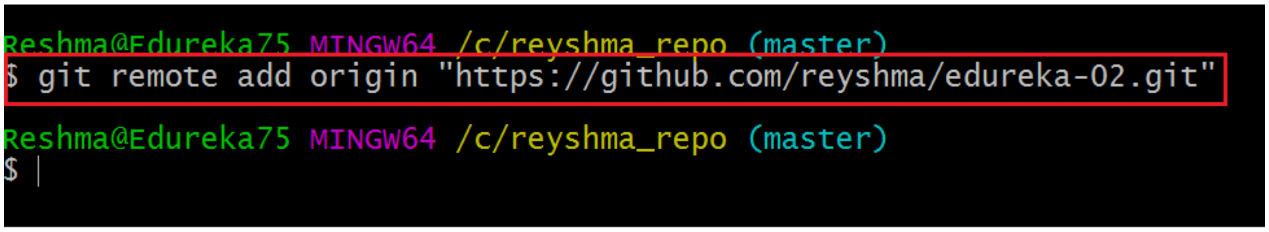
Note that before you affect changes to the central repository you should always pull changes from the central repository to your local repository to get updated with the work of all the collaborators that have been contributing in the central repository. For that we will use the **pull** command.

Pull

The **git pull** command fetches changes from a remote repository to a local repository. It merges upstream changes in your local repository, which is a common task in Git based collaborations.

But first, you need to set your central repository as origin using the command:

git remote add origin <link of your central repository>

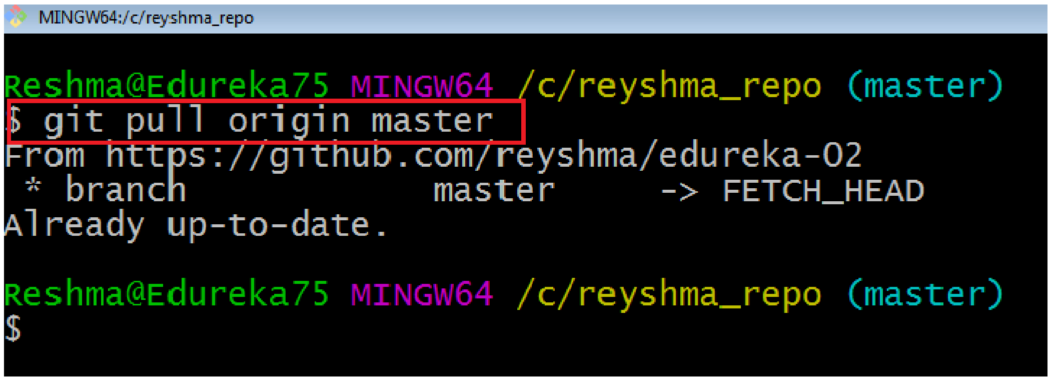


```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git remote add origin "https://github.com/reyshma/edureka-02.git"

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

Now that my origin is set, let us extract files from the origin using pull. For that use the command:

git pull origin master

This command will copy all the files from the master branch of remote repository to your local repository.



```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git pull origin master
From https://github.com/reyshma/edureka-02
 * branch            master     -> FETCH_HEAD
Already up-to-date.

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$
```

Since my local repository was already updated with files from master branch, hence the message is Already up-to-date. Refer to the screen shot above.

Note: *One can also try pulling files from a different branch using the following command:*

git pull origin <branch-name>

Your local Git repository is now updated with all the recent changes. It is time you make changes in the central repository by using the **push** command.

Push

This command transfers commits from your local repository to your remote repository. It is the opposite of pull operation.

Pulling imports commits to local repositories whereas pushing exports commits to the remote repositories.

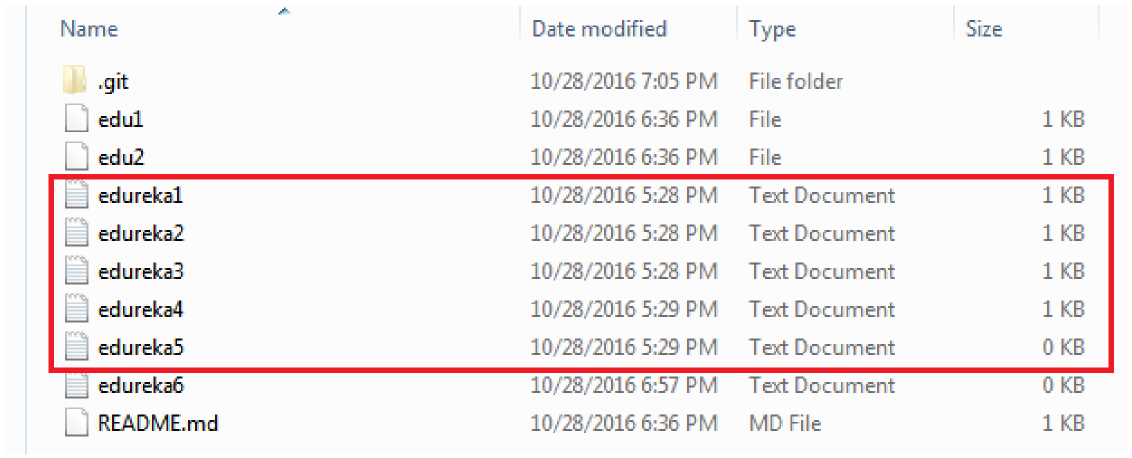
The use of **git push** is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you can then push them to the central repository by using the following command:

git push <remote>

Note: *This remote refers to the remote repository which had been set before using the pull command.*

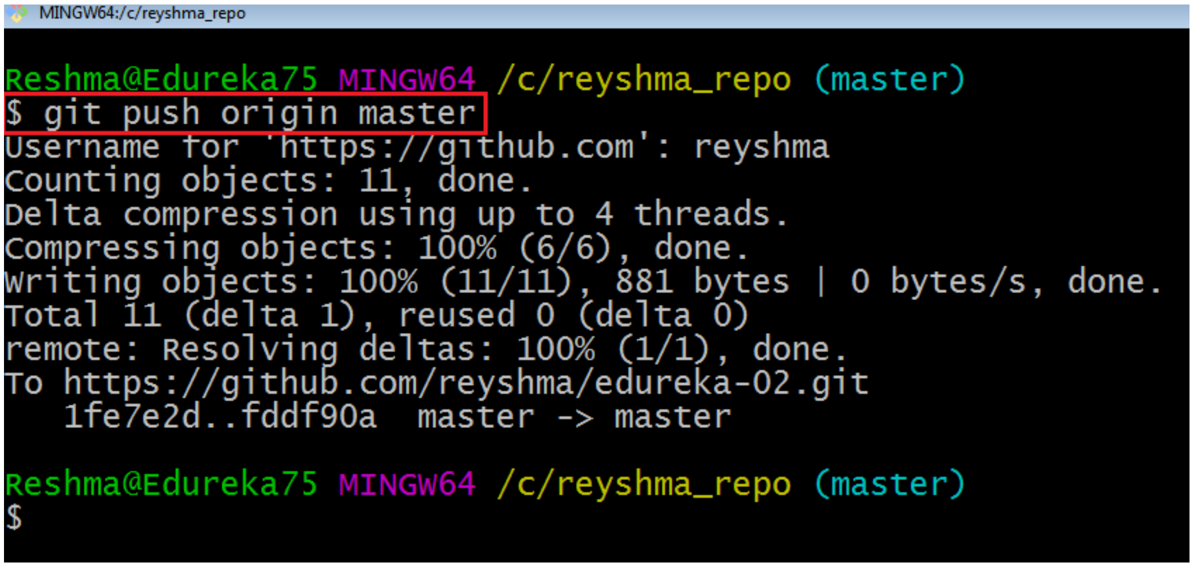
This pushes the changes from the local repository to the remote repository along with all the necessary commits and internal objects. This creates a local branch in the destination repository.

Let me demonstrate it for you.



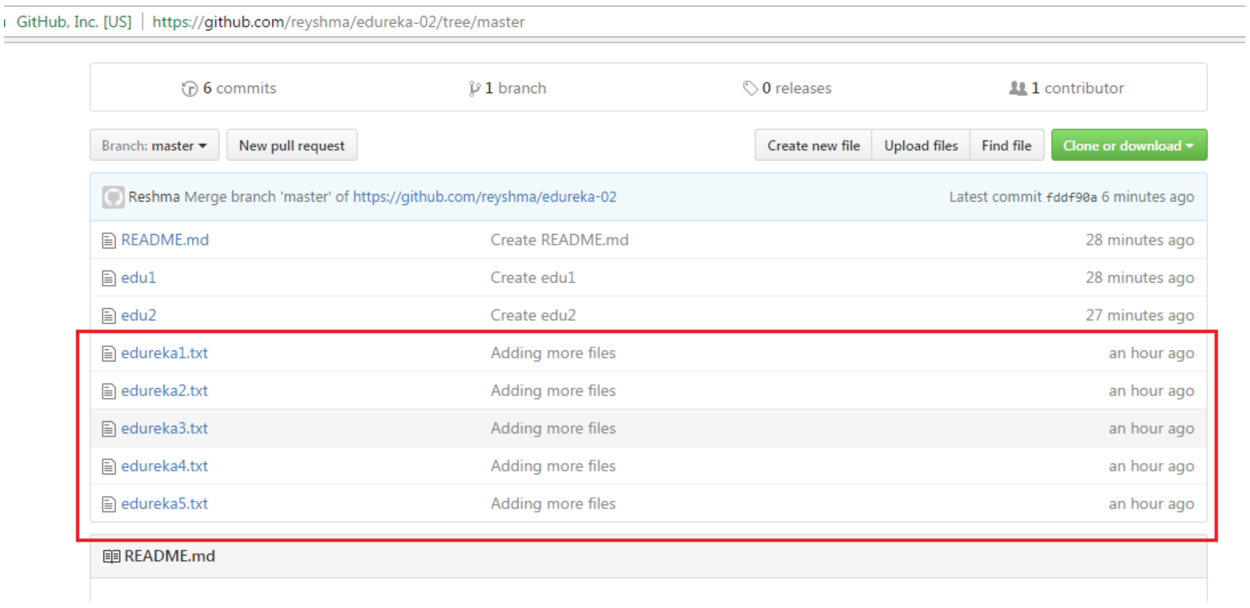
Name	Date modified	Type	Size
.git	10/28/2016 7:05 PM	File folder	
edu1	10/28/2016 6:36 PM	File	1 KB
edu2	10/28/2016 6:36 PM	File	1 KB
edureka1	10/28/2016 5:28 PM	Text Document	1 KB
edureka2	10/28/2016 5:28 PM	Text Document	1 KB
edureka3	10/28/2016 5:28 PM	Text Document	1 KB
edureka4	10/28/2016 5:29 PM	Text Document	1 KB
edureka5	10/28/2016 5:29 PM	Text Document	0 KB
edureka6	10/28/2016 6:57 PM	Text Document	0 KB
README.md	10/28/2016 6:36 PM	MD File	1 KB

The above files are the files which we have already committed previously in the commit section and they are all "*push-ready*". I will use the command **git push origin master** to reflect these files in the master branch of my central repository.



```
MINGW64:/c/reyshma_repo

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git push origin master
Username for 'https://github.com': reyshma
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (11/11), 881 bytes | 0 bytes/s, done.
Total 11 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/reyshma/edureka-02.git
   1fe7e2d..fddf90a  master -> master

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$
```

Let us now check if the changes took place in my central repository.


GitHub, Inc. [US] | https://github.com/reyshma/edureka-02/tree/master

6 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

Reshma Merge branch 'master' of https://github.com/reyshma/edureka-02 — Latest commit fddf90a 6 minutes ago

README.md	Create README.md	28 minutes ago
edu1	Create edu1	28 minutes ago
edu2	Create edu2	27 minutes ago
edureka1.txt	Adding more files	an hour ago
edureka2.txt	Adding more files	an hour ago
edureka3.txt	Adding more files	an hour ago
edureka4.txt	Adding more files	an hour ago
edureka5.txt	Adding more files	an hour ago

README.md

Yes, it did. :-)

To prevent overwriting, Git does not allow push when it results in a non-fast forward merge in the destination repository.

Note: *A non-fast forward merge means an upstream merge i.e. merging with ancestor or parent branches from a child branch.*

To enable such merge, use the command below:

git push <remote> –force

The above command forces the push operation even if it results in a non-fast forward merge.

At this point of this Git Tutorial, I hope you have understood the basic commands of Git. Now, let's take a step further to learn branching and merging in Git.

Branching

Branches in Git are nothing but pointers to a specific commit. Git generally prefers to keep its branches as lightweight as possible.

There are basically two types of branches viz. ***local branches*** and ***remote tracking branches***.

A local branch is just another path of your working tree. On the other hand, remote tracking branches have special purposes. Some of them are:

- They link your work from the local repository to the work on central repository.
- They automatically detect which remote branches to get changes from, when you use **git pull**.

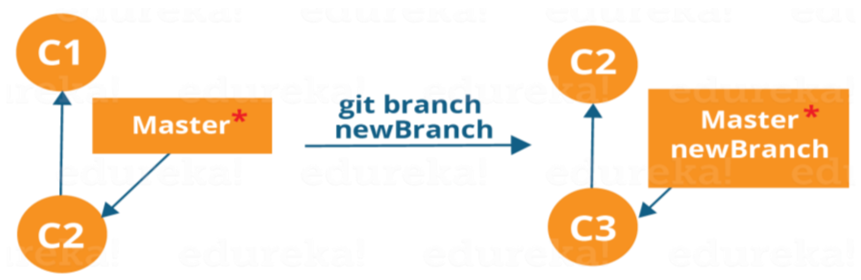
You can check what your current branch is by using the command:

git branch

The one mantra that you should always be chanting while branching is "branch early, and branch often"

To create a new branch, we use the following command:

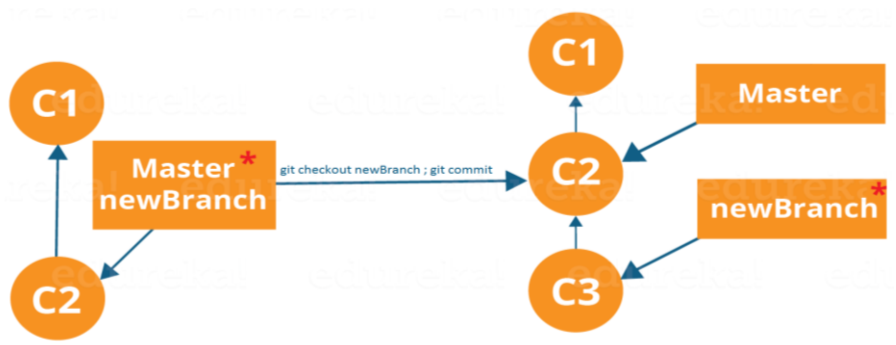
git branch <branch-name>



The diagram above shows the workflow when a new branch is created. When we create a new branch it originates from the master branch itself.

Since there is no storage/memory overhead with making many branches, it is easier to logically divide up your work rather than have big chunky branches.

Now, let us see how to commit using branches.

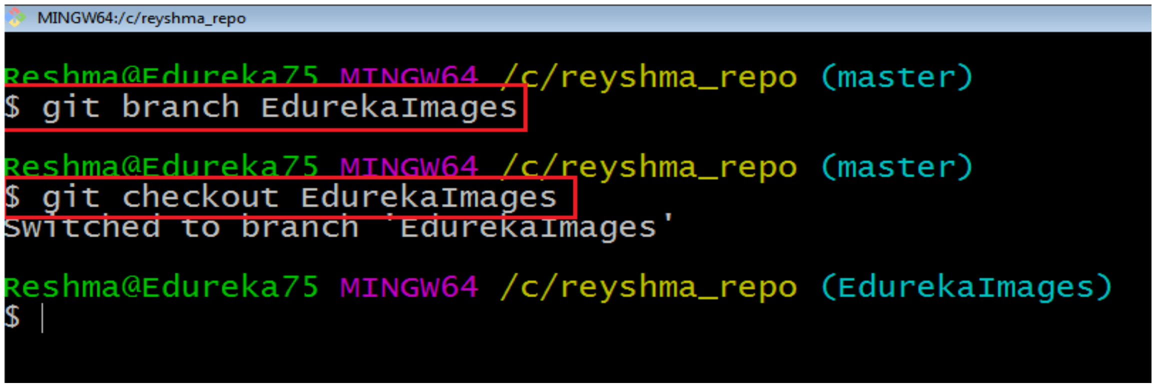


Branching includes the work of a particular commit along with all parent commits. As you can see in the diagram above, the new Branch has detached itself from the master and hence will create a different path.

Use the command below:

git checkout <branch_name> and then

git commit



```
MINGW64:/c/reyshma_repo

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git branch EdurekaImages

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git checkout EdurekaImages
Switched to branch 'EdurekaImages'

Reshma@Edureka75 MINGW64 /c/reyshma_repo (EdurekaImages)
$ |
```

Here, I have created a new branch named "EdurekaImages" and switched on to the new branch using the command **git checkout**.

One shortcut to the above commands is:

git checkout -b[branch_name]

This command will create a new branch and checkout the new branch at the same time.

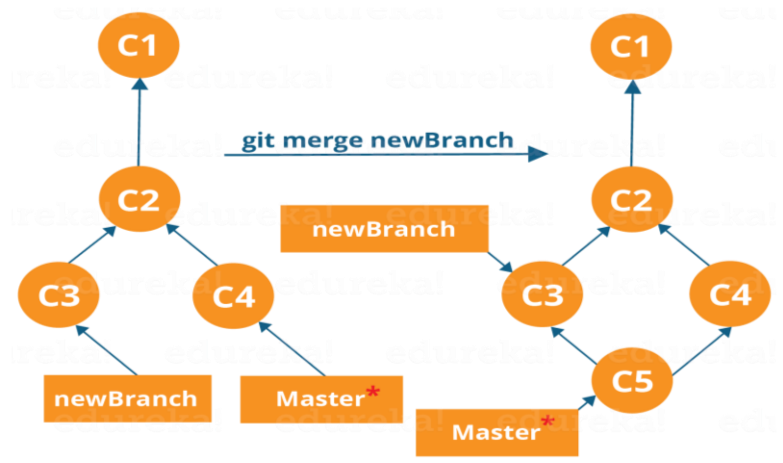
Now while we are in the branch EdurekaImages, add and commit the text file *edureka6.txt* using the following commands:

git add edureka6.txt

git commit –m "adding edureka6.txt"

Merging

Merging is the way to combine the work of different branches together. This will allow us to branch off, develop a new feature, and then combine it back in.



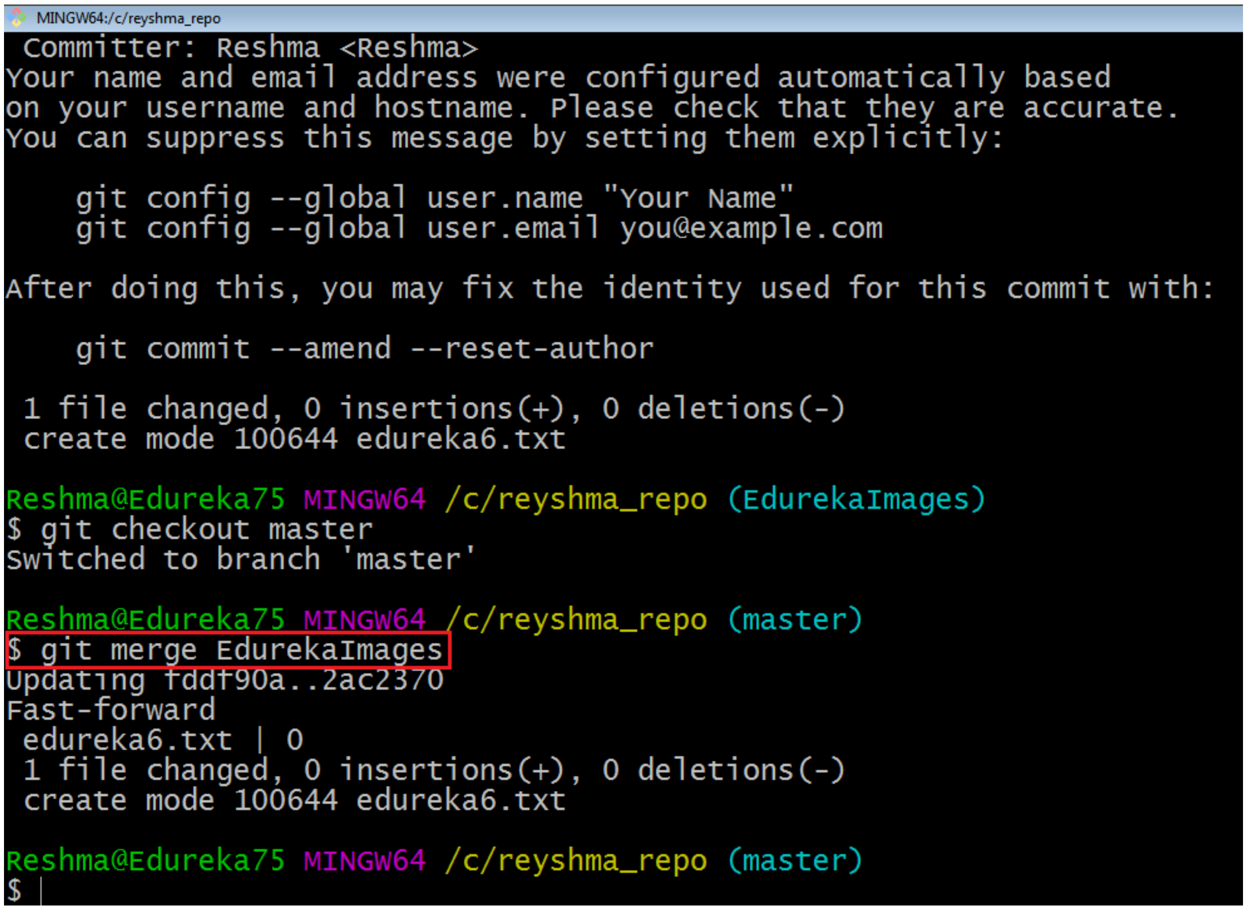
The diagram above shows us two different branches-> newBranch and master. Now, when we merge the work of newBranch into master, it creates a new commit which contains all the work of master and newBranch.

Now let us merge the two branches with the command below:

git merge <branch_name>

It is important to know that the branch name in the above command should be the branch you want to merge into the branch you are currently checking out. So, make sure that you are checked out in the destination branch.

Now, let us merge all of the work of the branch EdurekaImages into the master branch. For that I will first checkout the master branch with the command **git checkout master** and merge EdurekaImages with the command **git merge EdurekaImages**



```
MINGW64:/c/reyshma_repo
 Committer: Reshma <Reshma>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly:

    git config --global user.name "Your Name"
    git config --global user.email you@example.com

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 edureka6.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (EdurekaImages)
$ git checkout master
Switched to branch 'master'

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git merge EdurekaImages
Updating fddf90a..2ac2370
Fast-forward
 edureka6.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 edureka6.txt

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

As you can see above, all the data from the branch name are merged to the master branch. Now, the text file *edureka6.txt* has been added to the master branch.

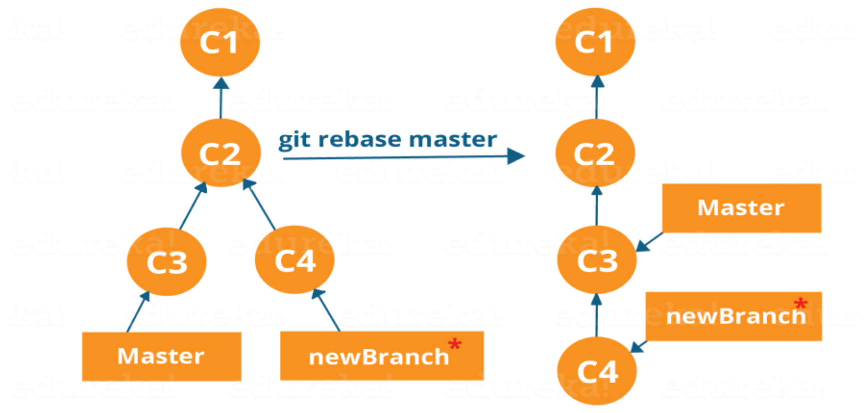
Merging in Git creates a special commit that has two unique parents.

Rebasing

This is also a way of combining the work between different branches. Rebasing takes a set of commits, copies them and stores them outside your repository.

The advantage of rebasing is that it can be used to make linear sequence of commits. The commit log or history of the repository stays clean if rebasing is done.

Let us see how it happens.

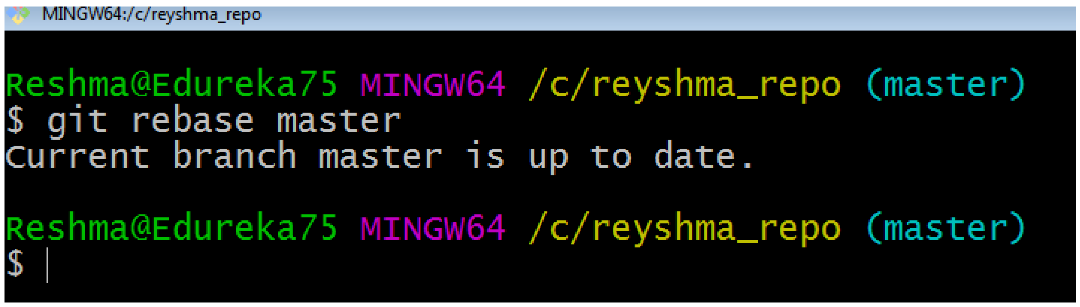


Now, our work from newBranch is placed right after master and we have a nice linear sequence of commits.

Note: *Rebasing also prevents upstream merges, meaning you cannot place master right after newBranch.*

Now, to rebase master, type the command below in your Git Bash:

git rebase master



```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git rebase master
Current branch master is up to date.

Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ |
```

This command will move all our work from current branch to the master. They look like as if they are developed sequentially, but they are developed parallelly.

Git Tutorial – Tips And Tricks

Now that you have gone through all the operations in this Git Tutorial, here are some tips and tricks you ought to know. :-)

- **Archive your repository**

Use the following command-

git archive master –format=zip –output= ../name-of-file.zip

It stores all files and data in a zip file rather than the **.git** directory.

Note that this creates only a single snapshot omitting version control completely. This comes in handy when you want to send the files to a client for review who doesn't have Git installed in their computer.

- **Bundle your repository**

It turns a repository into a single file.

Use the following command-

git bundle create ../repo.bundler master

This pushes the master branch to a remote branch, only contained in a file instead of a repository.

An alternate way to do it is:

cd..

git clone repo.bundle repo-copy -b master

cd repo-copy

git log

cd.. /my-git-repo

- **Stash uncommitted changes**

When we want to undo adding a feature or any kind of added data temporarily, we can "stash" them temporarily.

Use the command below:

git status

git stash

git status

And when you want to re-apply the changes you "stash"ed ,use the command below:

git stash apply

I hope you have enjoyed this Git Bash Tutorial and learned the commands and operations in Git.

Certifications

1. The Linux Basics Course - https://kodekloud.com/p/the-linux-basics-course
2. Docker for Beginners: Full Free Course! - https://www.youtube.com/watch?v=zJ6WbK9zFpI
3. Devops-pre-requisite-course - https://kodekloud.com/p/devops-pre-requisite-course

Future

1. Docker for the Absolute Beginner - Hands On - https://kodekloud.com/courses/enrolled/296044

Learn through Videos

1. Introduction to SDLC - https://www.youtube.com/watch?v=i-QyW8D3ei0
2. SDLC Life Cycle for Beginners | Software Development Life Cycle with Real life example - https://www.youtube.com/watch?v=kSU2MPeptpM
3. DevOps Tools Full Course in 11 Hours | DevOps Tools Tutorial | DevOps Training | Edureka - https://www.youtube.com/watch?v=S_0q75eD8Yc
4. Docker for Beginners: Full Free Course! - https://www.youtube.com/watch?v=zJ6WbK9zFpI
5. Learn the Basics of Git in Under 10 Minutes - https://www.freecodecamp.org/news/learn-the-basics-of-git-in-under-10-minutes-da548267cc91/
6. GIT Documentation - https://git-scm.com/docs/gittutorial
7. Git --fast-version-control (Book) - https://git-scm.com/book/en/v2
8. Git Videos - https://git-scm.com/videos
9. Git External Links - https://git-scm.com/doc/ext
10. Git Downloads - https://git-scm.com/downloads
11. Git Community - https://git-scm.com