This tutorial contains following topic:

- **DevOps and SDLC model tutorial**
- **GIT configuration and Commands**

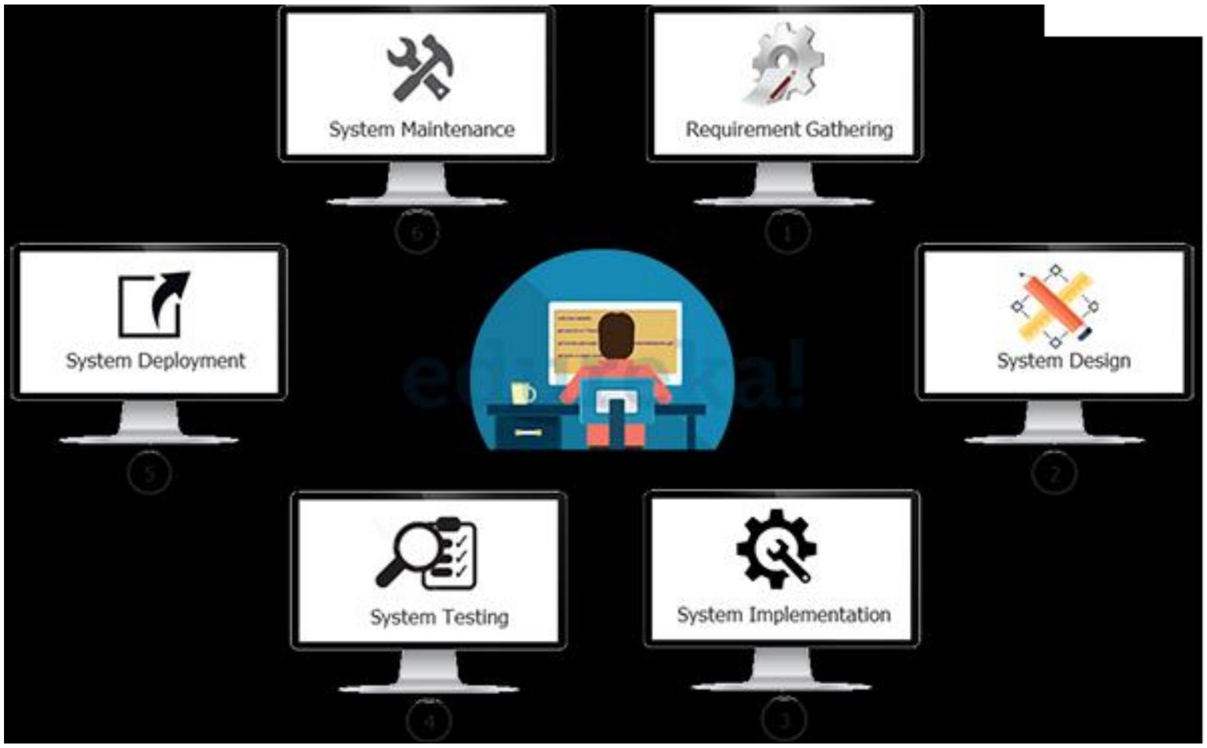
The DevOps Tutorial

In this DevOps Tutorial blog I will take you through the following things, which will be the base of the upcoming blogs:

- What led DevOps to come into existence
- Introduction of DevOps

Waterfall Model

Let's consider developing software in a traditional way using a Waterfall Model.



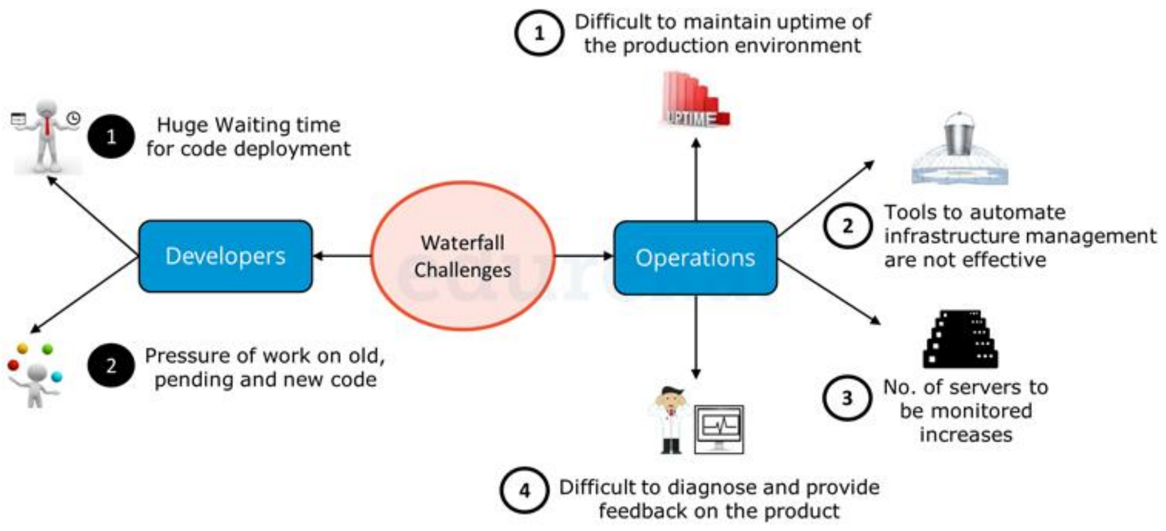
In the above diagram you will see the phases it will involve:

- In phase 1 – Complete Requirement is gathered and SRS is developed
- In phase 2 – This System is Planned and Designed using the SRS
- In phase 3 – Implementation of the System takes place

- In phase 4 – System is tested and its quality is assured
- In phase 5 – System is deployed to the end users
- In phase 6 – Regular Maintenance of the system is done

Waterfall Model Challenges

The Water-fall model worked fine and served well for many years however it had some challenges. In the following diagram the challenges of Waterfall Model are highlighted.


1 Difficult to maintain uptime of the production environment
UPTIME
1 Huge Waiting time for code deployment
Developers
Waterfall Challenges
Operations
2 Tools to automate infrastructure management are not effective
2 Pressure of work on old, pending and new code
3 No. of servers to be monitored increases
4 Difficult to diagnose and provide feedback on the product

In the above diagram you can see that both Development and Operations had challenges in the Waterfall Model. From Developers point of view there were majorly two challenges:

1 After Development, the code deployment time was huge.

2 Pressure of work on old, pending and new code was high because development and deployment time was high.

On the other hand, Operations was also not completely satisfied. There were four major challenges they faced as per the above diagram:

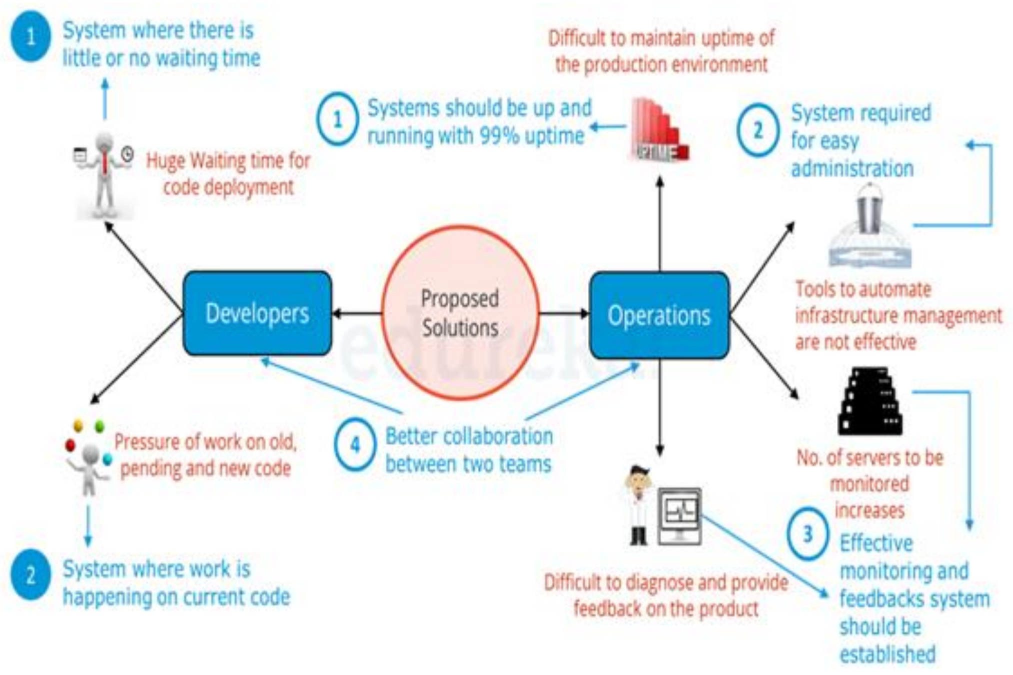
1 It was difficult to maintain ~100% uptime of the production environment.

2 Infrastructure Automation tools were not very affective.

3 Number of severs to be monitored keeps on increasing with time and hence the complexity.

4 It was very difficult to provide feedback and diagnose issue in the product.

In the following diagram proposed solution to the challenges of Waterfall Model are highlighted.


1 System where there is little or no waiting time
Huge Waiting time for code deployment
Difficult to maintain uptime of the production environment
1 Systems should be up and running with 99% uptime
2 System required for easy administration
Developers
Proposed Solutions
Operations
Tools to automate infrastructure management are not effective
Pressure of work on old, pending and new code
4 Better collaboration between two teams
2 System where work is happening on current code
No. of servers to be monitored increases
3 Effective monitoring and feedbacks system should be established
Difficult to diagnose and provide feedback on the product

In the above diagram, Probable Solutions for the issues faced by Developers and Operations are highlighted in blue. This sets the guidelines for an Ideal Software Development strategy.

From Developers point of view:

1 A system which enables code deployment without any delay or wait time.

2 A system where work happens on the current code itself i.e. development sprints are short and well planned.

From Operations point of view:

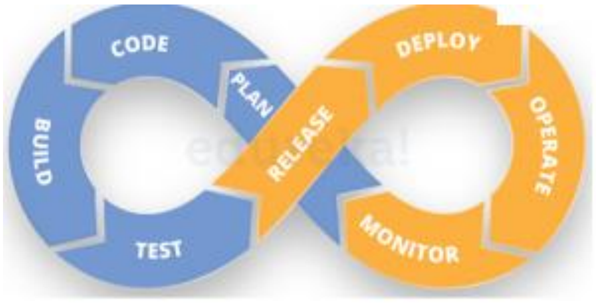
1 System should have at-least 99% uptime.

2 Tools & systems are there in place for easy administration.

3 Effective monitoring and feedbacks system should be there.

4 Better Collaboration between Development & Operations and is common requirement for Developers and Operations team.

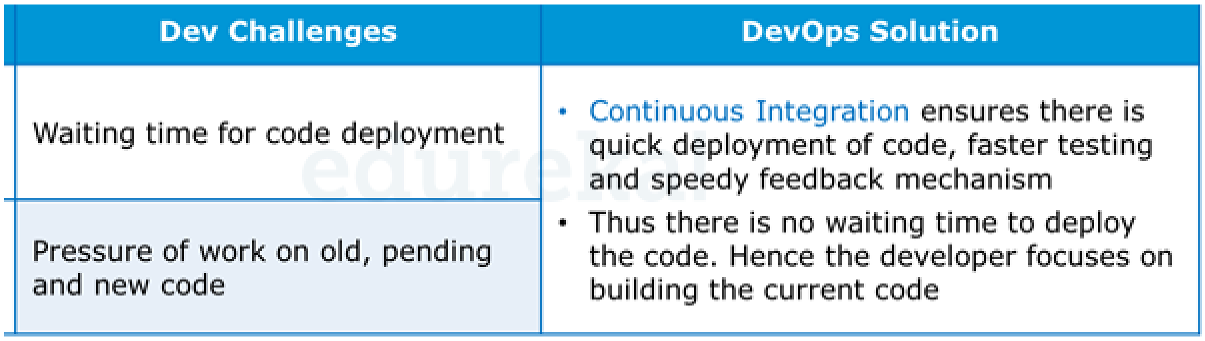
DevOps integrates developers and operations team to improve collaboration and productivity.



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?

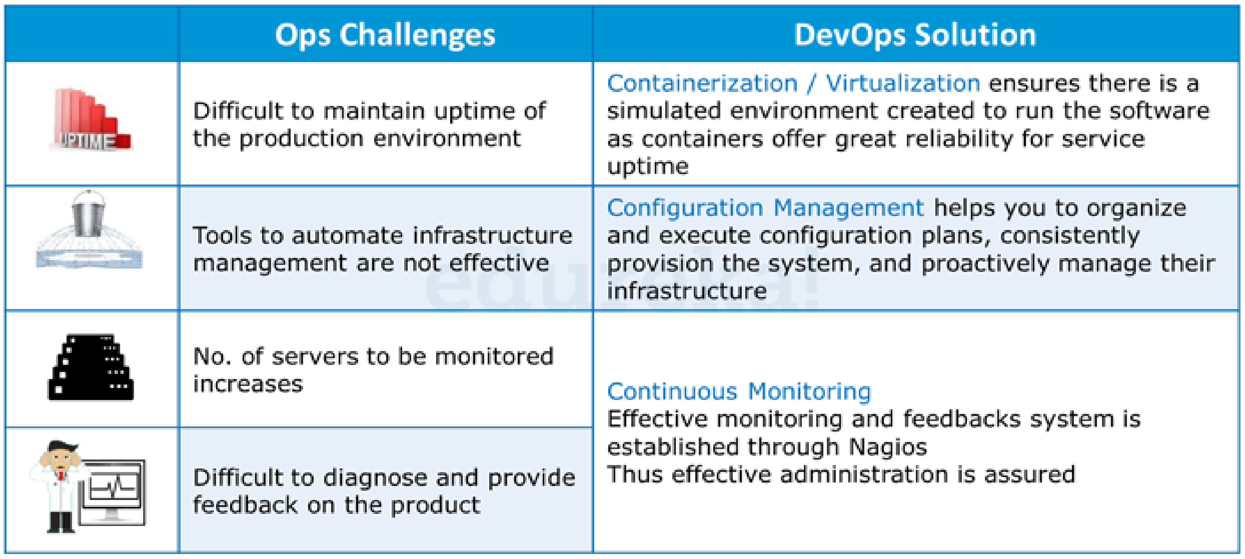
Now let us see how DevOps takes care of the challenges faced by Development and Operations. Below table describes how DevOps addresses Dev Challenges.



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

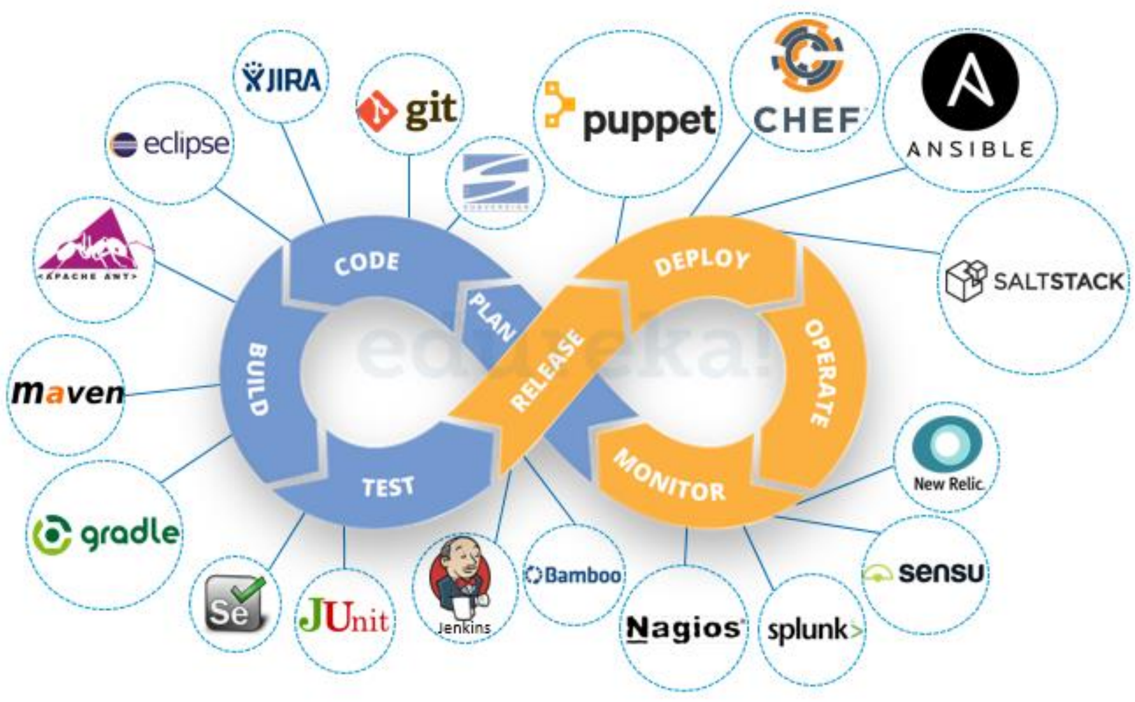
Going further, below table describes how DevOps addresses Ops Challenges.



	Ops Challenges	DevOps Solution
UPTIME	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, you would still be wondering, how 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.



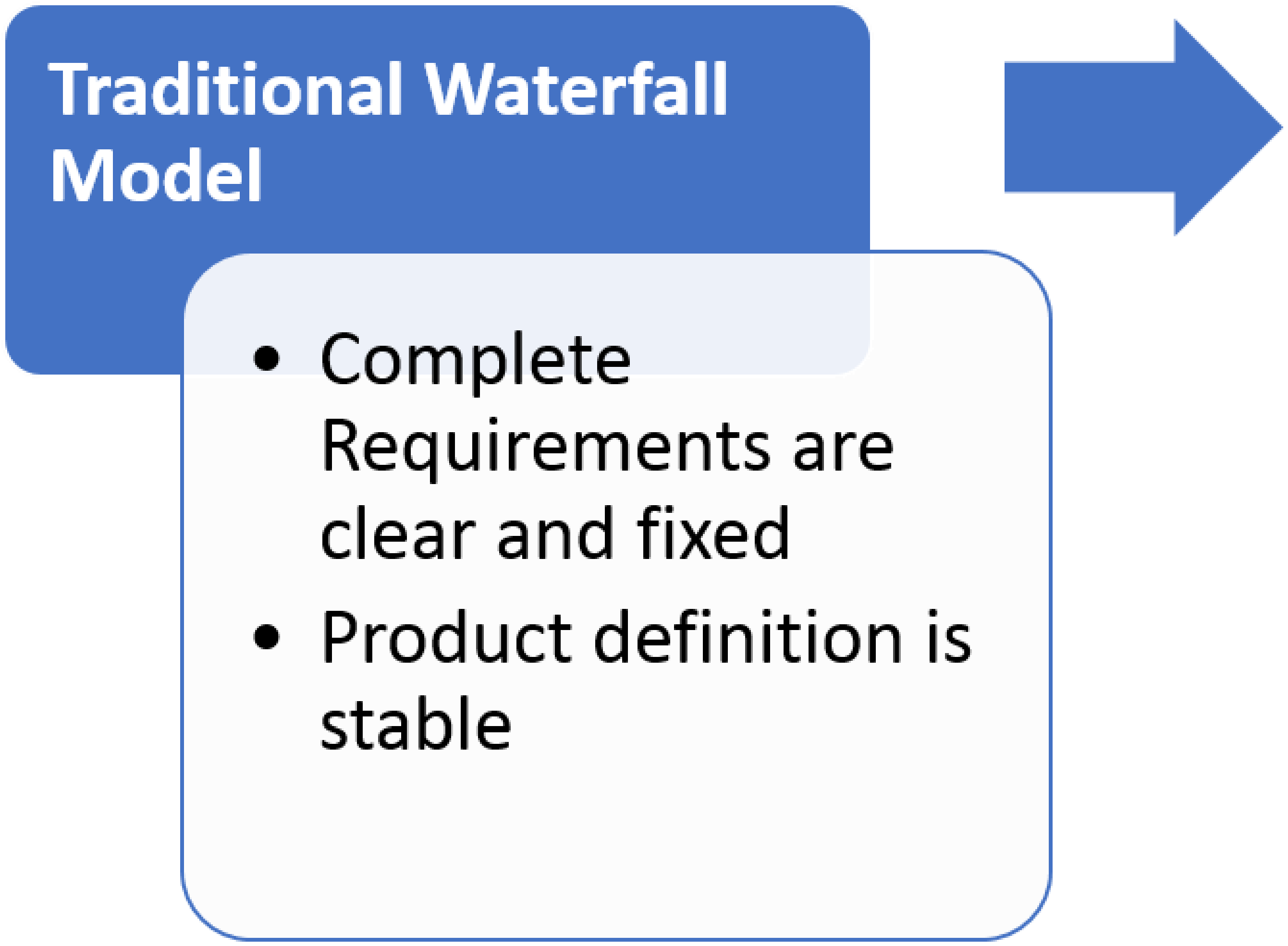
These tools has been categorized into various stages of DevOps. Hence it is important that I first tell you about DevOps stages and then talk more about DevOps Tools.

DevOps Lifecycle can be broadly broken down into the below DevOps Stages:

- Continuous Development
- Continuous Integration
- Continuous Testing
- Continuous Monitoring
- Virtualization and Containerization

Evolution of Software Development

DevOps evolved from existing software development strategies/methodologies over the years in response to business needs. Let us briefly look at how these models evolved and in which scenarios they would work best.


Traditional Waterfall Model
Complete Requirements are clear and fixed
Product definition is stable

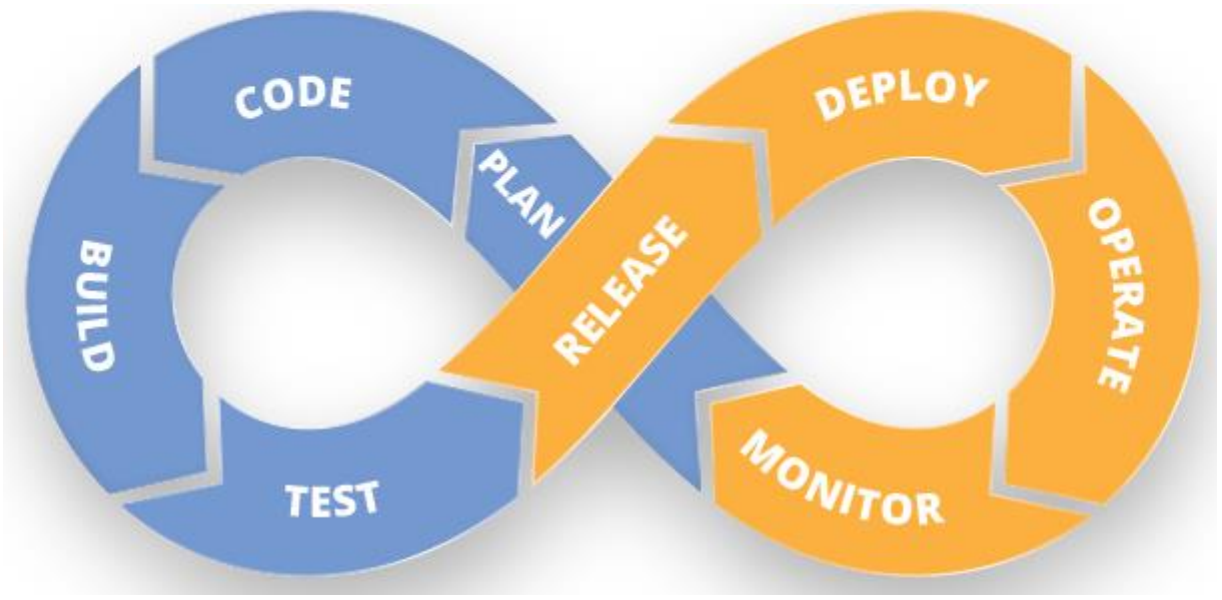
The slow and cumbersome Waterfall model evolved into Agile which saw development teams working on the software in short sprints lasting not more than two weeks. Having such a short release cycle helped the development team work on client feedback and incorporate it along with bug fixes in the next release. While this Agile SCRUM approach brought agility to development, it was lost on Operations which did not come up to speed with Agile practices. Lack of collaboration between Developers and Operations Engineers still slowed down the development process and releases. DevOps methodology was born out of this need for better collaboration and faster delivery. DevOps enables continuous software delivery with less complex problems to fix and faster resolution of problems.

Now that we have understood the evolution of DevOps, let us look at what is DevOps in detail.

What is DevOps ?

DevOps is a Software Development approach which involves Continuous Development, Continuous Testing, Continuous Integration, Continuous Deployment and Continuous Monitoring of the software throughout its development life cycle. These activities are possible only in DevOps, not Agile or waterfall, and this is why Facebook and other top companies have chosen DevOps as the way forward for their business goals. DevOps is the preferred approach to develop high quality software in shorter development cycles which results in greater customer satisfaction. Check out the below video on What is DevOps before you go ahead.

Your understanding of what is DevOps is incomplete without learning about its life cycle. Let us now look at the DevOps life cycle and explore how they are related to the Software Development stages depicted in the diagram below.



Continuous Development:

This is the stage in the DevOps life cycle where the Software is developed continuously. Unlike the Waterfall model the software deliverables are broken down into multiple sprints of short development cycles, developed and then delivered in a very short time. This stage involves the Coding and Building phases and makes use of tools such as Git and SVN for maintaining the different versions of the code, and tools like Ant, Maven, Gradle for building / packaging the code into an executable file that can be forwarded to the QAs for testing.

Continuous Testing:

It is the stage where the developed software is continuously tested for bugs. For Continuous testing testing automation tools like Selenium, JUnit etc are used. These tools enables the QA's for testing multiple code-bases thoroughly in parallel to ensure that there are no flaws in the functionality. In this phase use of Docker containers for simulating testing environment on the fly, is also a preferred choice. Once the code is tested, it is continuously integrated with the existing code.

Continuous Integration:

This is the stage where the code supporting new functionality is integrated with the existing code. Since there is continuous development of software, the updated code needs to be integrated continuously as well as smoothly with the systems to reflect changes to the end users. The changed code, should also ensure that there are no errors in the runtime environment, allowing us to test the changes and check how it reacts with other changes. Jenkins is a very popular tool used for Continuous Integration. Using Jenkins one can pull the latest code revision from GIT repository and produce a build which can finally be deployed to test or production server. It can be set to trigger a new build automatically as soon as there is change in the GIT repository or can be triggered manually on click of a button.

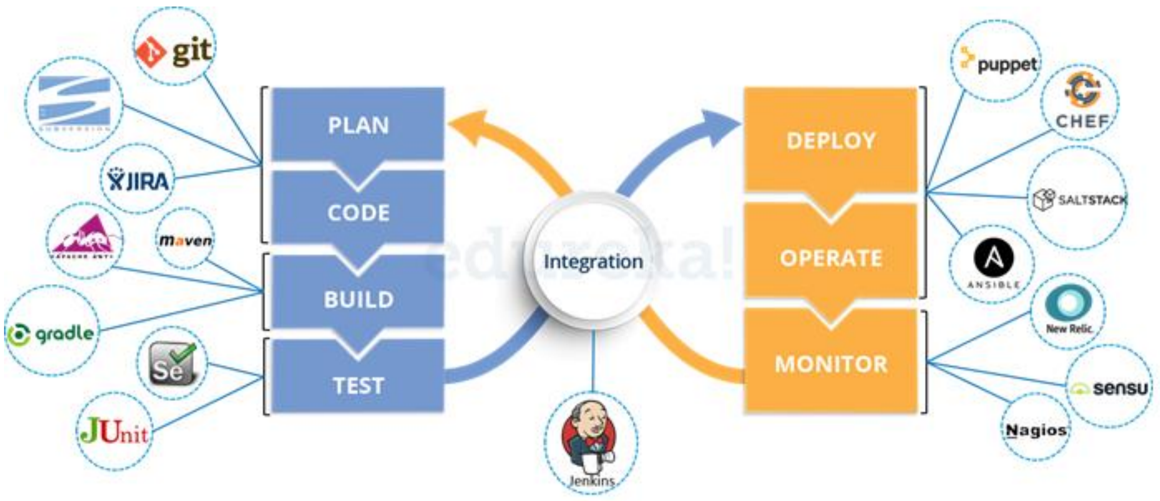
Continuous Deployment:

It is the stage where the code is deployed to the production environment. Here we ensure that the code is correctly deployed on all the servers. If there is any addition of functionality or a new feature is introduced then one should be ready to welcome greater website traffic. So it is also the responsibility of the SysAdmin to scale up the servers to host more users. Since the new code is deployed on a continuous basis, automation tools play an important role for executing tasks quickly and frequently. Puppet, Chef, SaltStack and Ansible are some popular tools that are used in this stage.

Continuous Monitoring:

This is a very crucial stage in the DevOps life cycle which is aimed at improving the quality of the software by monitoring its performance. This practice involves the participation of the Operations team who will monitor the user activity for bugs / any improper behavior of the system. This can also be achieved by making use of dedicated monitoring tools which will continuously monitor the application performance and highlight issues. Some popular tools used are Nagios, NewRelic and Sensu. These tools help you monitor the application and the servers closely to check the health of the system proactively. They can also improve productivity and increase the reliability of the systems, reducing IT support costs. Any major issues found could be reported to the Development team so that it can be fixed in the continuous development phase.

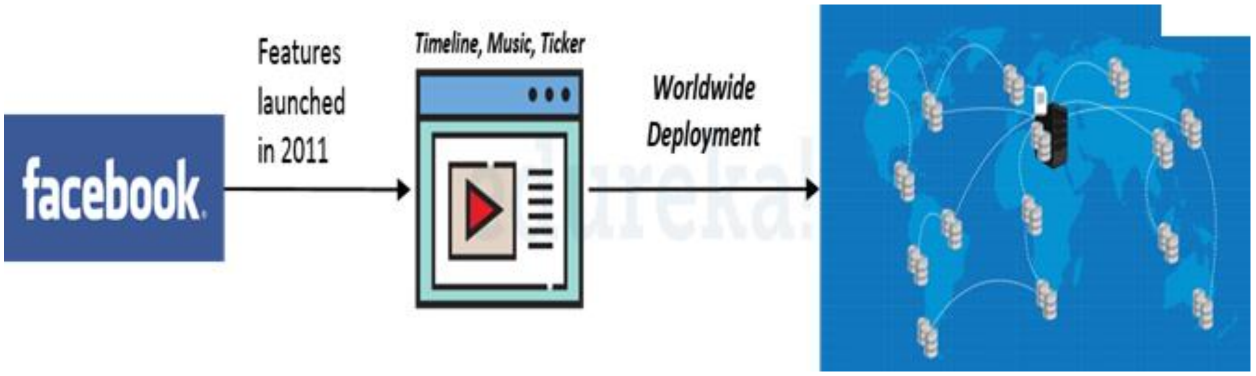
These DevOps stages are carried out on loop continuously until the desired product quality is achieved. The diagram given below will show you which tools can be used in which stage of the DevOps life cycle.



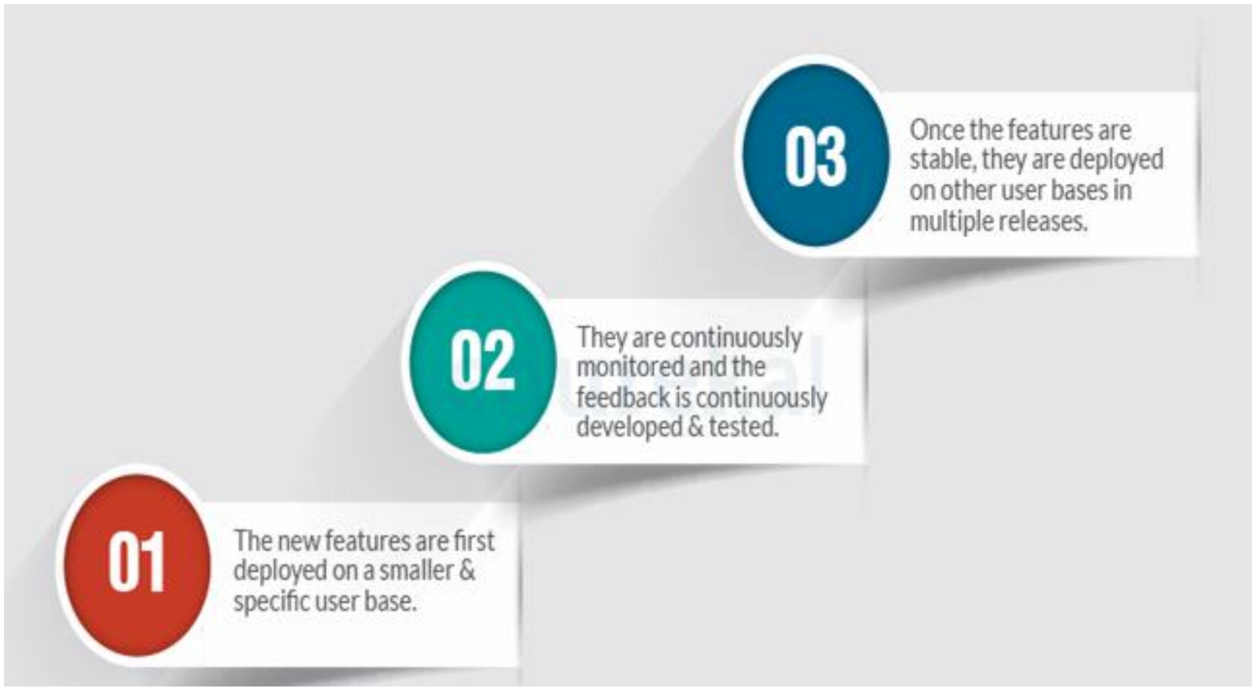
Now that we have established the significance of DevOps and learnt about its different stages along with the DevOps tools involved, let us now look at a Facebook case study and understand why they moved from Agile to DevOps. We will do this by taking up the use case of Facebook's 2011 roll-out of new features that resulted in them reassessing their product delivery and taking on the DevOps approach. But before go through the below video on DevOps Tools.

DevOps Case Study: Facebook

In 2011, Facebook rolled out a slew of new features – timeline, ticker and music functionalities – to its 500 million users spread across the globe. The huge traffic that was generated on Facebook following the release led to a server meltdown. The features that were rolled out garnered mixed response from users which led to inconclusive results of the effectiveness of the new features, leaving them with no actionable insights.

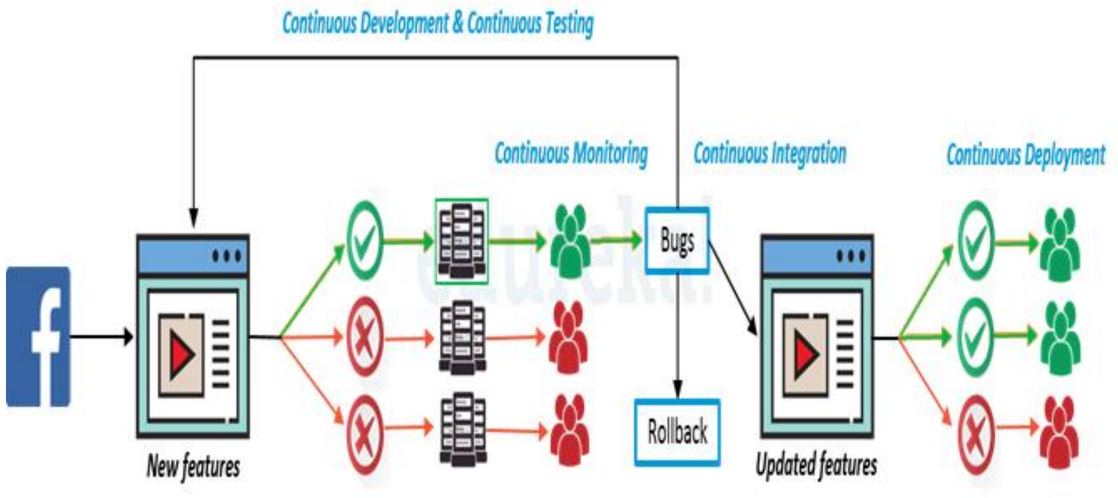


This led to an evaluation and reassessment of strategies, resulting in Facebook coming up with the Dark Launching Technique. Using the DevOps principles, Facebook created the following methodology for the launch of its new releases.


03
Once the features are stable, they are deployed on other user bases in multiple releases.
02
They are continuously monitored and the feedback is continuously developed & tested.
01
The new features are first deployed on a smaller & specific user base.

Facebook Dark Launching Technique

Dark launching is the process of gradually rolling out production-ready features to a select set of users before a full release. This allows development teams to get user feedback early on, test bugs, and also stress test infrastructure performance. A direct result of continuous delivery, this method of release helps in faster, more iterative releases that ensure that application performance does not get affected and that the release is well received by customers.



In the Dark Launching technique, features are released to a small user base through a dedicated deployment pipeline. In the below given diagram of Facebook Dark Launch, you can see that that only one deployment pipeline is turned on to deploy the new features to a select set of users. The remaining hundreds of pipelines are all turned off at this point. The specific user base on which the features have been deployed are continuously monitored to collect feedback and identify bugs. These bugs and feedback will be incorporated in development, tested and deployed on the same user base until the features becomes stable. Once stability is achieved, the features will be gradually deployed on other user bases by turning on other deployment pipelines.

Facebook does this by wrapping code in a feature flag or feature toggle which is used to control who gets to see the new feature and when. This exposes pain points and areas of the application's infrastructure that needs attention prior to the full-fledged launch while still simulating the full effect of launching the code to users. Once the features are stable, they are deployed to the rest of the users over multiple releases.

This way Facebook has a controlled or stable mechanism for developing new functionality to its massive user base. On the contrary if the feature does not get a good response they have an option to rollback on their deployments altogether. This also helps them to prepare their servers for deployment as they can predict the user activity on their website and they can scale up their servers accordingly. The diagram given above depicts how a dark launch takes place at Facebook.

Facebook, Amazon, Netflix and Google, along with many leading tech giants, use dark launches to gradually release and test new features to a small set of their users before releasing to everyone.

GIT

What is Git – Why Git Came Into Existence?

We all know "Necessity is the mother of all inventions". And similarly Git was also invented to fulfill certain necessities that the developers faced before Git. So, let us take a step back to learn all about Version Control Systems (VCS) and how Git came into existence.

Version Control is the management of changes to documents, computer programs, large websites and other collection of information.

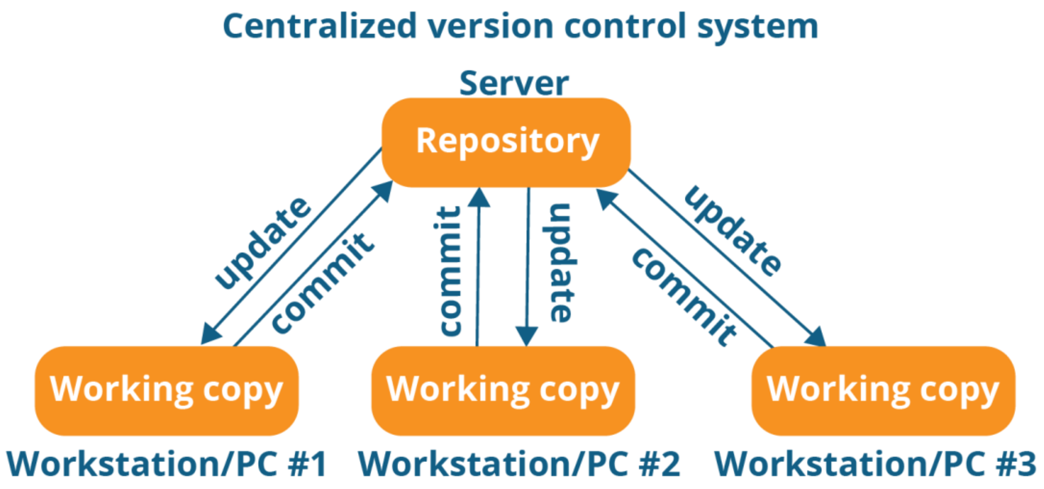
There are two types of VCS:

- Centralized Version Control System (CVCS)
- Distributed Version Control System (DVCS)

Centralized VCS

Centralized version control system (CVCS) uses a central server to store all files and enables team collaboration. It works on a single repository to which users can directly access a central server.

Please refer to the diagram below to get a better idea of CVCS:



The repository in the above diagram indicates a central server that could be local or remote which is directly connected to each of the programmer's workstation.

Every programmer can extract or **update** their workstations with the data present in the repository or can make changes to the data or **commit** in the repository. Every operation is performed directly on the repository.

Even though it seems pretty convenient to maintain a single repository, it has some major drawbacks. Some of them are:

- It is not locally available; meaning you always need to be connected to a network to perform any action.
- Since everything is centralized, in any case of the central server getting crashed or corrupted will result in losing the entire data of the project.

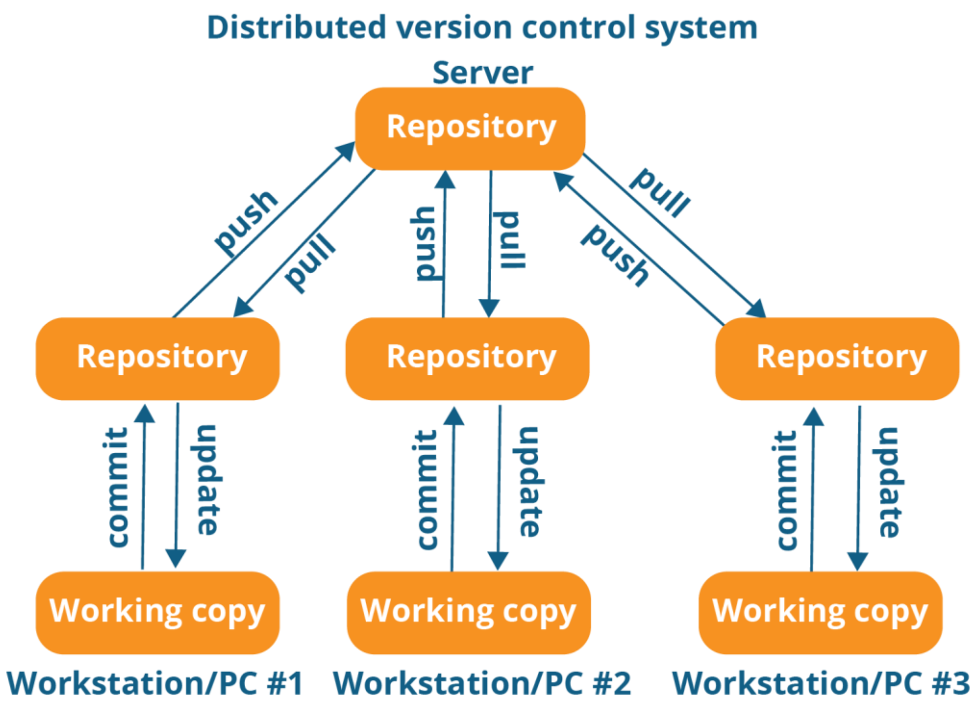
This is when Distributed VCS comes to the rescue.

Distributed VCS

These systems do not necessarily rely on a central server to store all the versions of a project file.

In Distributed VCS, every contributor has a local copy or "clone" of the main repository i.e. everyone maintains a local repository of their own which contains all the files and metadata present in the main repository.

You will understand it better by referring to the diagram below:



As you can see in the above diagram, every programmer maintains a local repository on its own, which is actually the copy or clone of the central repository on their hard drive. They can commit and update their local repository without any interference.

They can update their local repositories with new data from the central server by an operation called "**pull**" and affect changes to the main repository by an operation called "**push**" from their local repository.

The act of cloning an entire repository into your workstation to get a local repository gives you the following advantages:

- All operations (except push & pull) are very fast because the tool only needs to access the hard drive, not a remote server. Hence, you do not always need an internet connection.
- Committing new change-sets can be done locally without manipulating the data on the main repository. Once you have a group of change-sets ready, you can push them all at once.
- Since every contributor has a full copy of the project repository, they can share changes with one another if they want to get some feedback before affecting changes in the main repository.
- If the central server gets crashed at any point of time, the lost data can be easily recovered from any one of the contributor's local repositories.

After knowing Distributed VCS, its time we take a dive into what is Git.

What Is Git?

Git is a Distributed Version Control tool that supports distributed non-linear workflows by providing data assurance for developing quality software. Before you go ahead, check out this video on GIT which will give you better in-sight.

Git provides with all the Distributed VCS facilities to the user that was mentioned earlier. Git repositories are very easy to find and access. You will know how flexible and compatible Git is with your system when you go through the features mentioned below:

What is Git – Features Of Git

Free and open source:
Git is released under GPL's (General Public License) open source license. You don't need to purchase Git. It is absolutely free. And since it is open source, you can modify the source code as per your requirement.

Speed:

Since you do not have to connect to any network for performing all operations, it completes all the tasks really fast. Performance tests done by Mozilla showed it was an order of magnitude faster than other version control systems. Fetching version history from a locally stored repository can be one hundred times faster than fetching it from the remote server. The core part of Git is written in C, which avoids runtime overheads associated with other high level languages.

Scalable:

Git is very scalable. So, if in future , the number of collaborators increase Git can easily handle this change. Though Git represents an entire repository, the data stored on the client's side is very small as Git compresses all the huge data through a lossless compression technique.

Reliable:

Since every contributor has its own local repository, on the events of a system crash, the lost data can be recovered from any of the local repositories. You will always have a backup of all your files.

Secure:

Git uses the ***SHA1*** (Secure Hash Function) to name and identify objects within its repository. Every file and commit is check-summed and retrieved by its checksum at the time of checkout. The Git history is stored in such a way that the ID of a particular version (a *commit* in Git terms) depends upon the complete development history leading up to that commit. Once it is published, it is not possible to change the old versions without it being noticed.

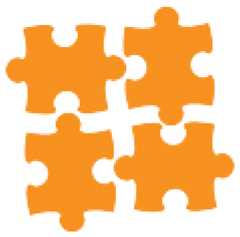
Economical:

In case of CVCS, the central server needs to be powerful enough to serve requests of the entire team. For smaller teams, it is not an issue, but as the team size grows, the hardware limitations of the server can be a performance bottleneck. In case of DVCS, developers don't interact with the server unless they need to push or pull changes. All the heavy lifting happens on the client side, so the server hardware can be very simple indeed.

Supports non-linear development:
Git supports rapid branching and merging, and includes specific tools for visualizing and navigating a non-linear development history. A core assumption in Git is that a change will be merged more often than it is written, as it is passed around various reviewers. Branches in Git are very lightweight. A branch in Git is only a reference to a single commit. With its parental commits, the full branch structure can be constructed.

Easy Branching:
Branch management with Git is very simple. It takes only few seconds to create, delete, and merge branches. Feature branches provide an isolated environment for every change to your codebase. When a developer wants to start working on something, no matter how big or small, they create a new branch. This ensures that the master branch always contains production-quality code.

Distributed development:
Git gives each developer a local copy of the entire development history, and changes are copied from one such repository to another. These changes are imported as additional development branches, and can be merged in the same way as a locally developed branch.



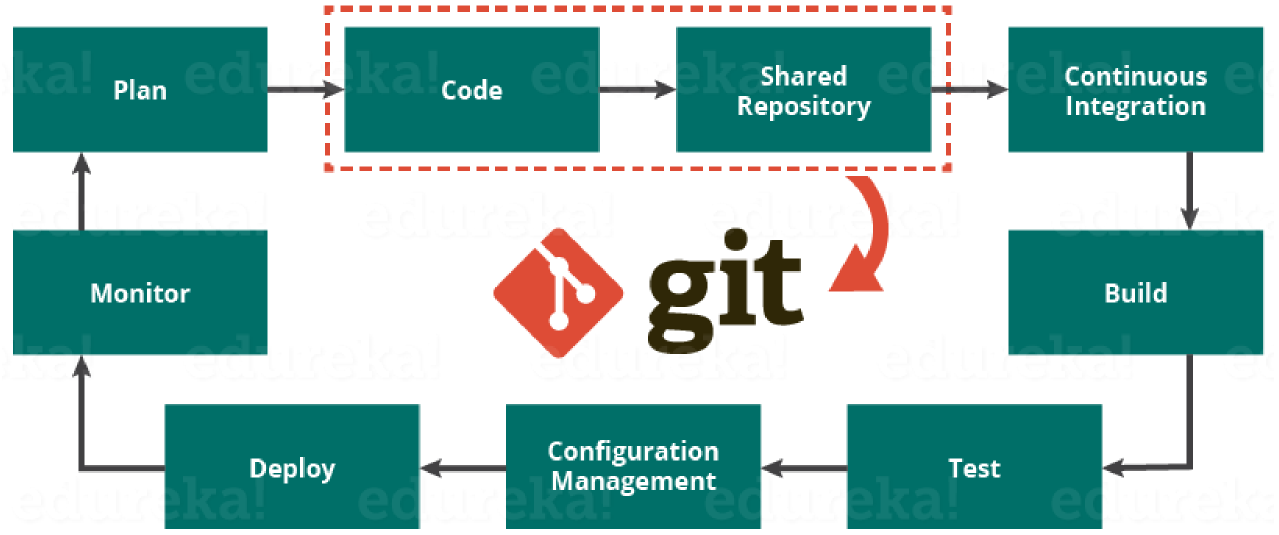
Compatibility with existing systems or protocol
Repositories can be published via http, ftp or a Git protocol over either a plain socket, or ssh. Git also has a Concurrent Version Systems (CVS) server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories. Apache SubVersion (SVN) and SVK repositories can be used directly with Git-SVN.

What is Git – Role Of Git In DevOps?

Now that you know what is Git, you should know Git is an integral part of DevOps.

DevOps is the practice of bringing agility to the process of development and operations. It's an entirely new ideology that has swept IT organizations worldwide, boosting project life-cycles and in turn increasing profits. DevOps promotes communication between development engineers and operations, participating together in the entire service life-cycle, from design through the development process to production support.

The diagram below depicts the Devops life cycle and displays how Git fits in Devops.

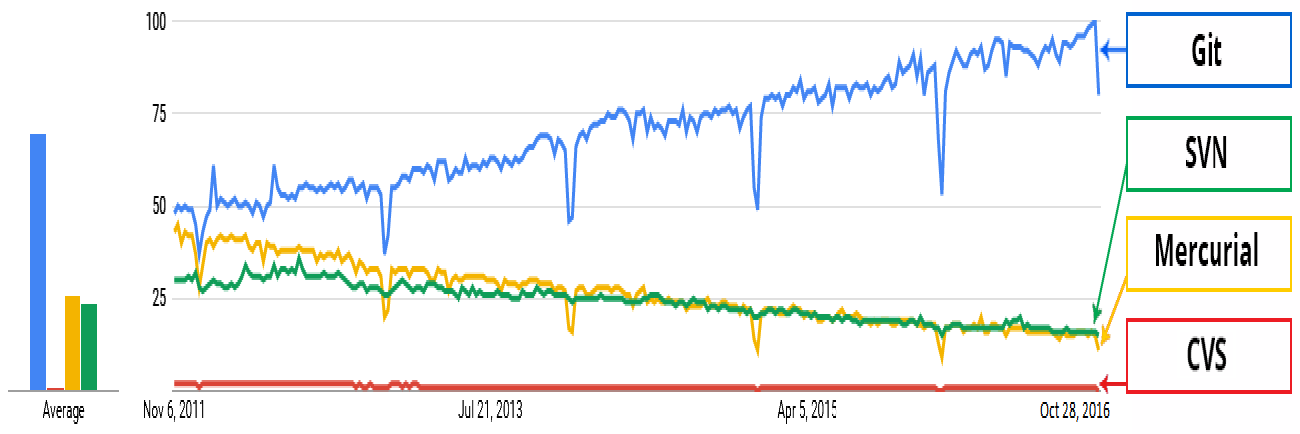


The diagram above shows the entire life cycle of Devops starting from planning the project to its deployment and monitoring. Git plays a vital role when it comes to managing the code that the collaborators contribute to the shared repository. This code is then extracted for performing continuous integration to create a build and test it on the test server and eventually deploy it on the production.

Tools like Git enable communication between the development and the operations team. When you are developing a large project with a huge number of collaborators, it is very important to have communication between the collaborators while making changes in the project. Commit messages in Git play a very important role in communicating among the team. The bits and pieces that we all deploy lies in the Version Control system like Git. To succeed in DevOps, you need to have all of the communication in Version Control. Hence, Git plays a vital role in succeeding at DevOps.

Companies Using Git

Git has earned way more popularity compared to other version control tools available in the market like Apache Subversion(SVN), Concurrent Version Systems(CVS), Mercurial etc. You can compare the interest of Git by time with other version control tools with the graph collected from *Google Trends* below:



In large companies, products are generally developed by developers located all around the world. To enable communication among them, Git is the solution.

Some companies that use Git for version control are: Facebook, Yahoo, Zynga, Quora, Twitter, eBay, Salesforce, Microsoft and many more.

Lately, all of Microsoft's new development work has been in Git features. Microsoft is migrating .NET and many of its open source projects on GitHub which are managed by Git.

One of such projects is the LightGBM. It is a fast, distributed, high performance gradient boosting framework based on decision tree algorithms which is used for ranking, classification and many other machine learning tasks.

Here, Git plays an important role in managing this distributed version of LightGBM by providing speed and accuracy.

Git Tutorial – Operations & Commands

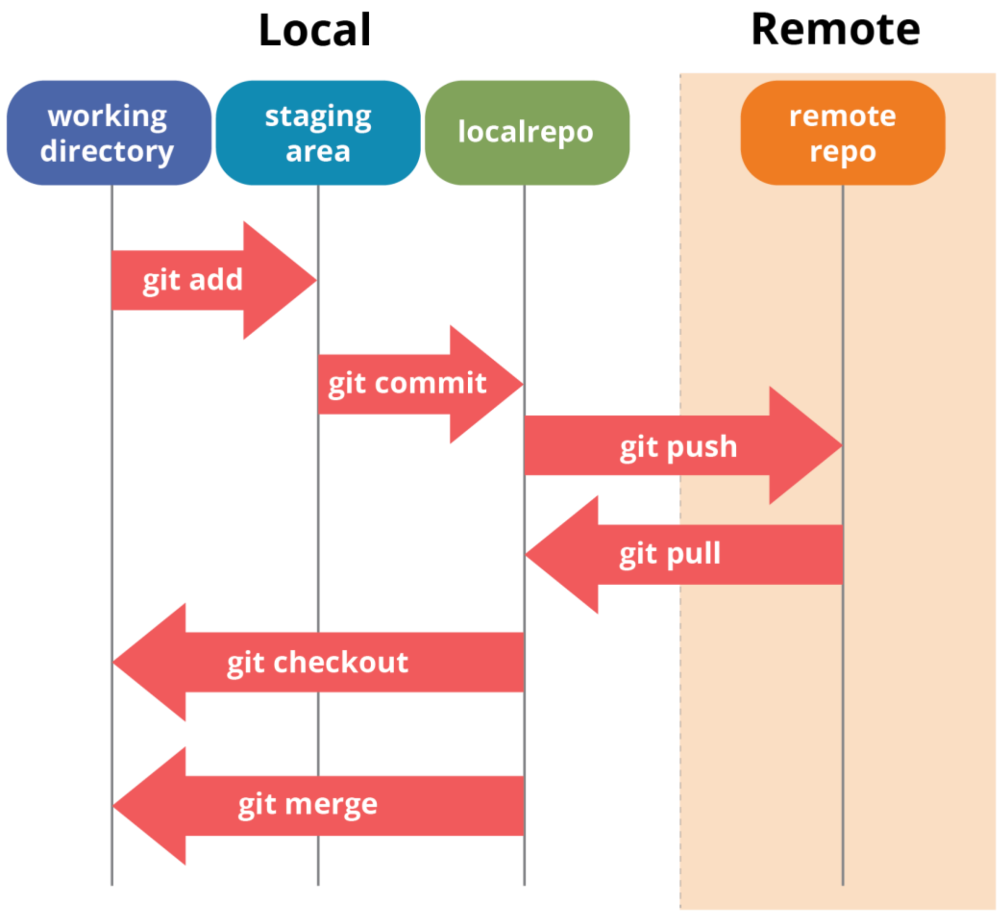
Some of the basic operations in Git are:

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:

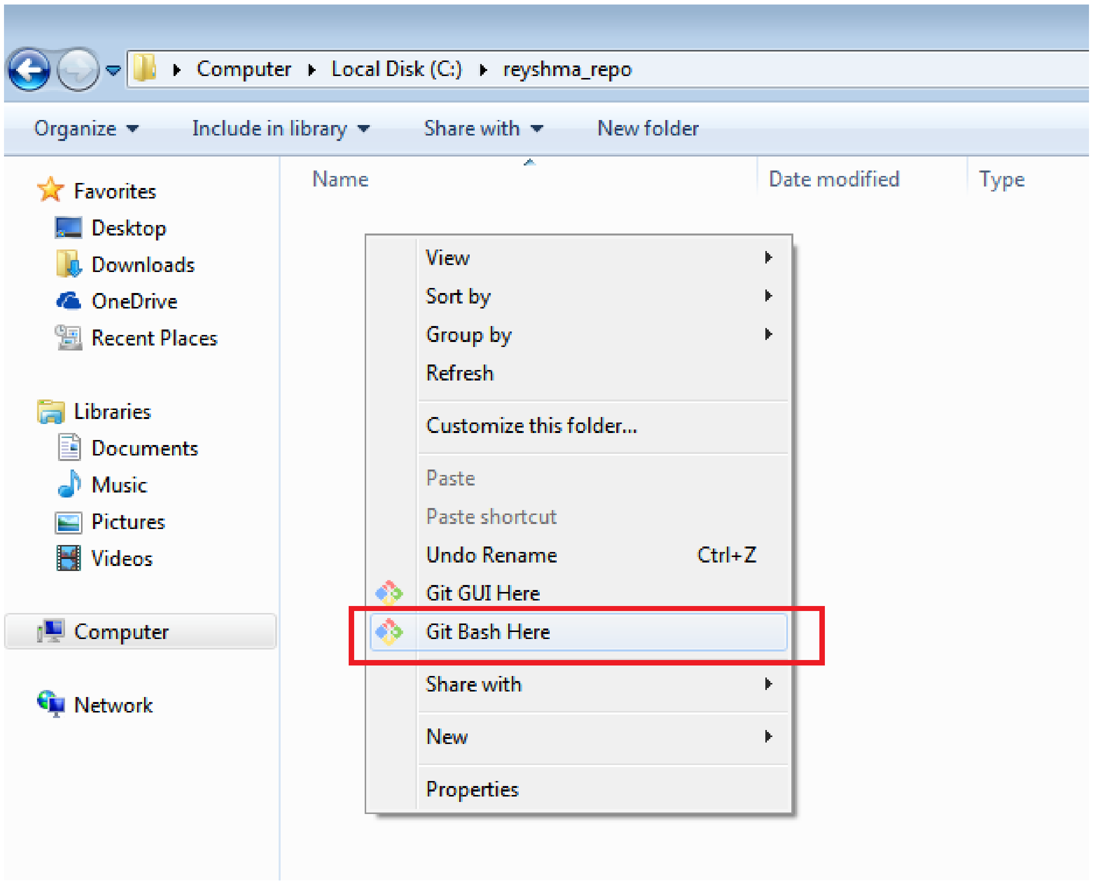


If you understand the above diagram well and good, but if you don't, you need not worry, I will be explaining these operations in this Git Tutorial one by one. Let us begin with the basic operations.

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

In this Git Tutorial, I will show you the commands and the operations using Git Bash. Git Bash is a text-only command line interface for using Git on Windows which provides features to run automated scripts.

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***'.

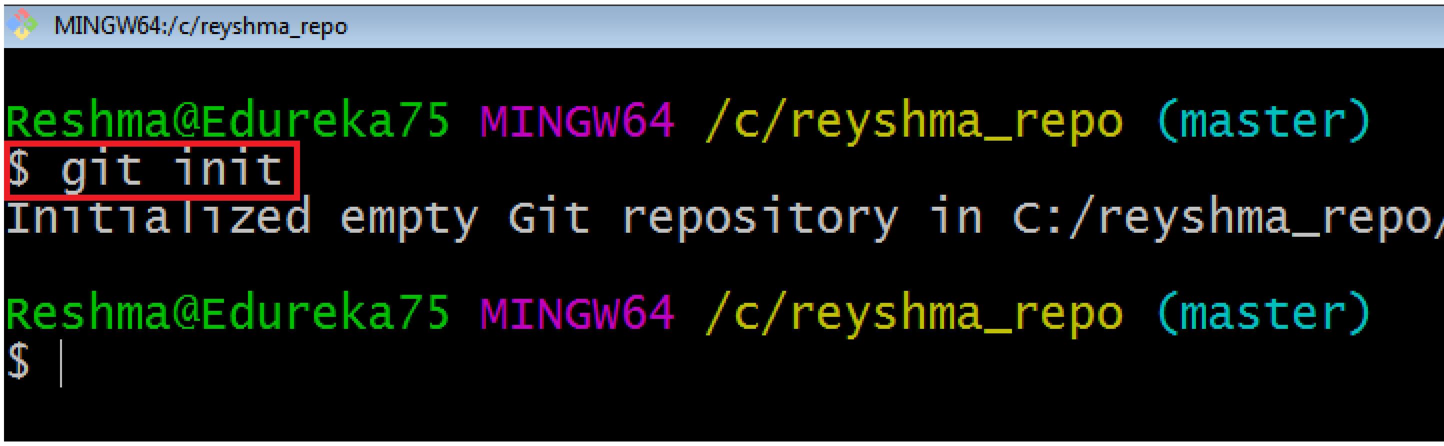


This will open up Git Bash terminal where you can enter commands to perform various Git operations.

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/

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.

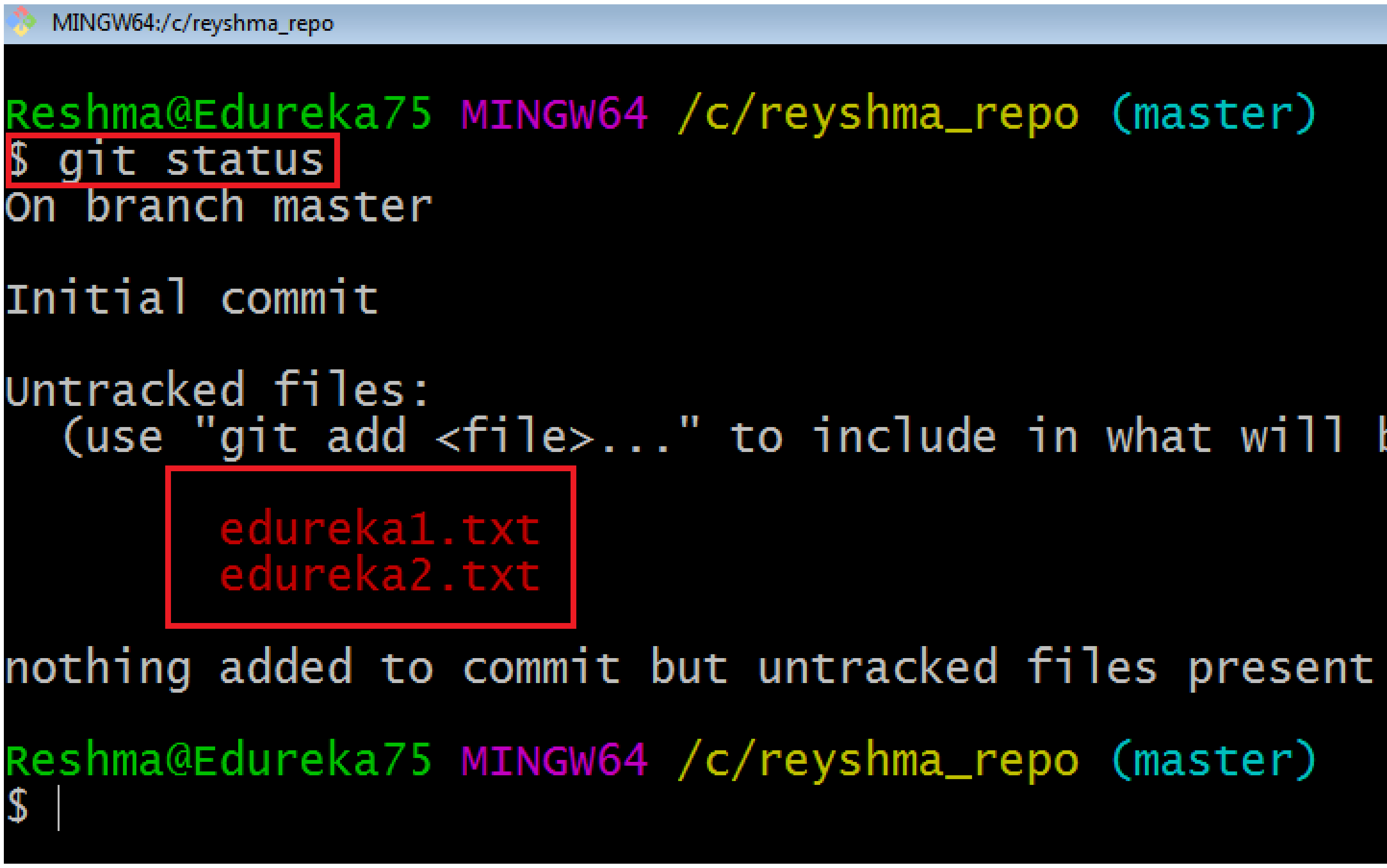
Now that my repository is initialized, let me create some files in the directory/repository. For e.g. I have created two text files namely *edureka1.txt* and *edureka2.txt*.

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:



```
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

        edureka1.txt
        edureka2.txt

nothing added to commit but untracked files present

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:

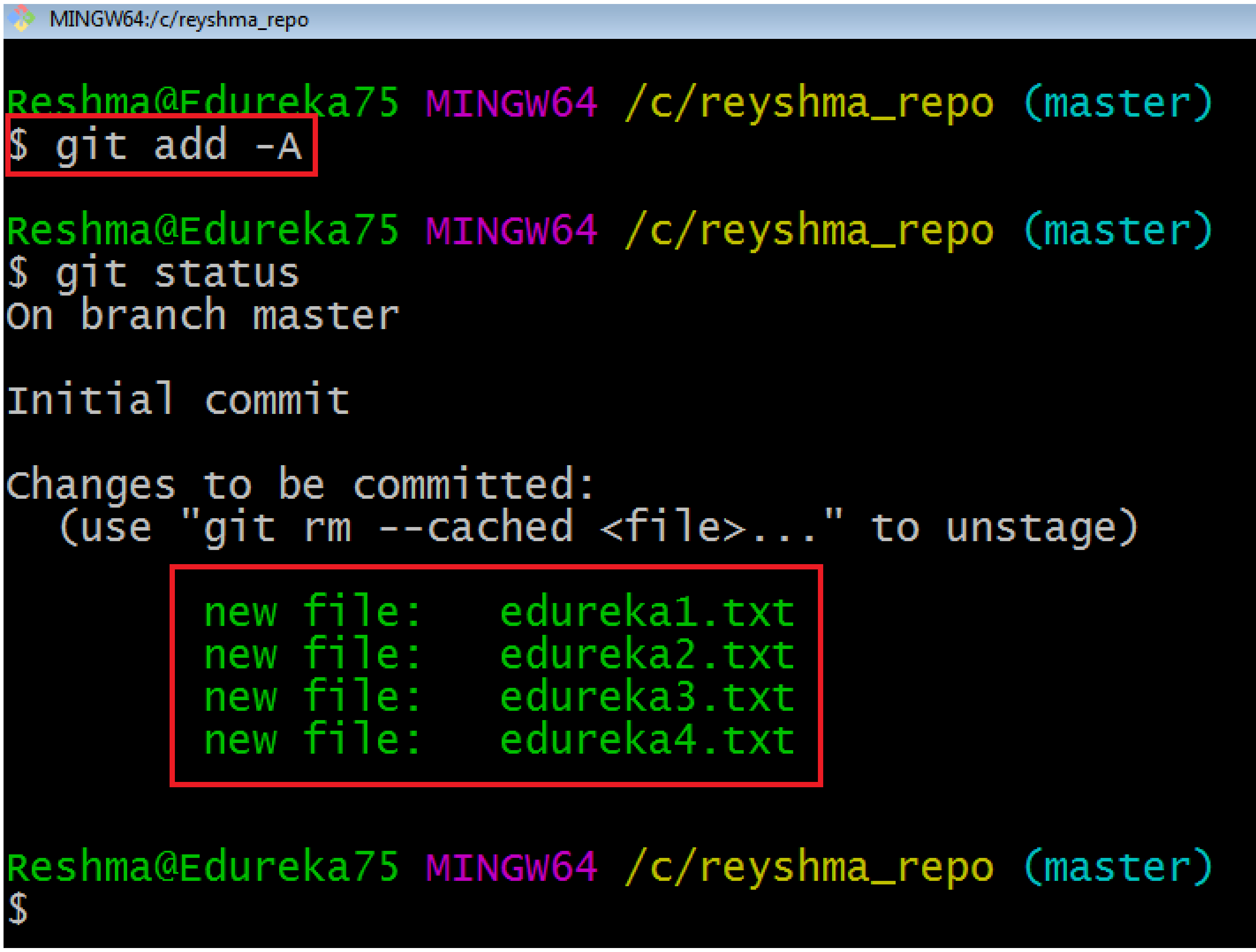
git add <directory>

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.

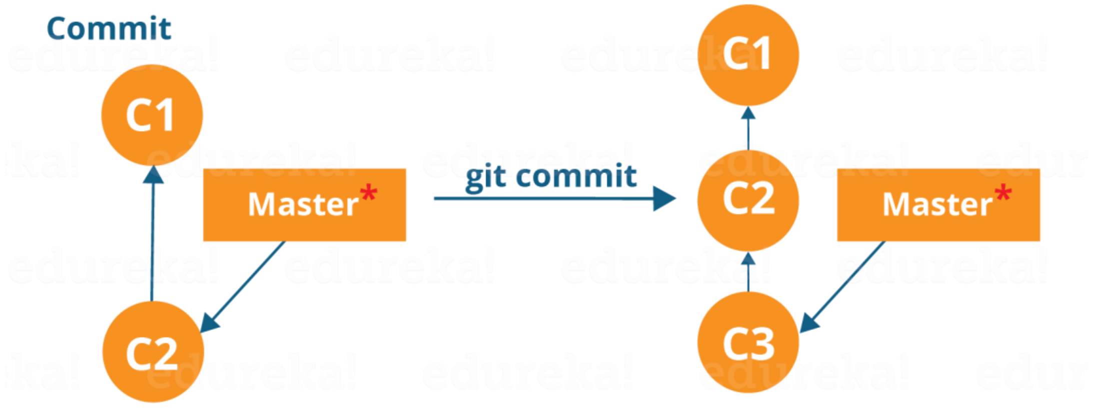

MINGW64:/c/reyshma_repo

```
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:

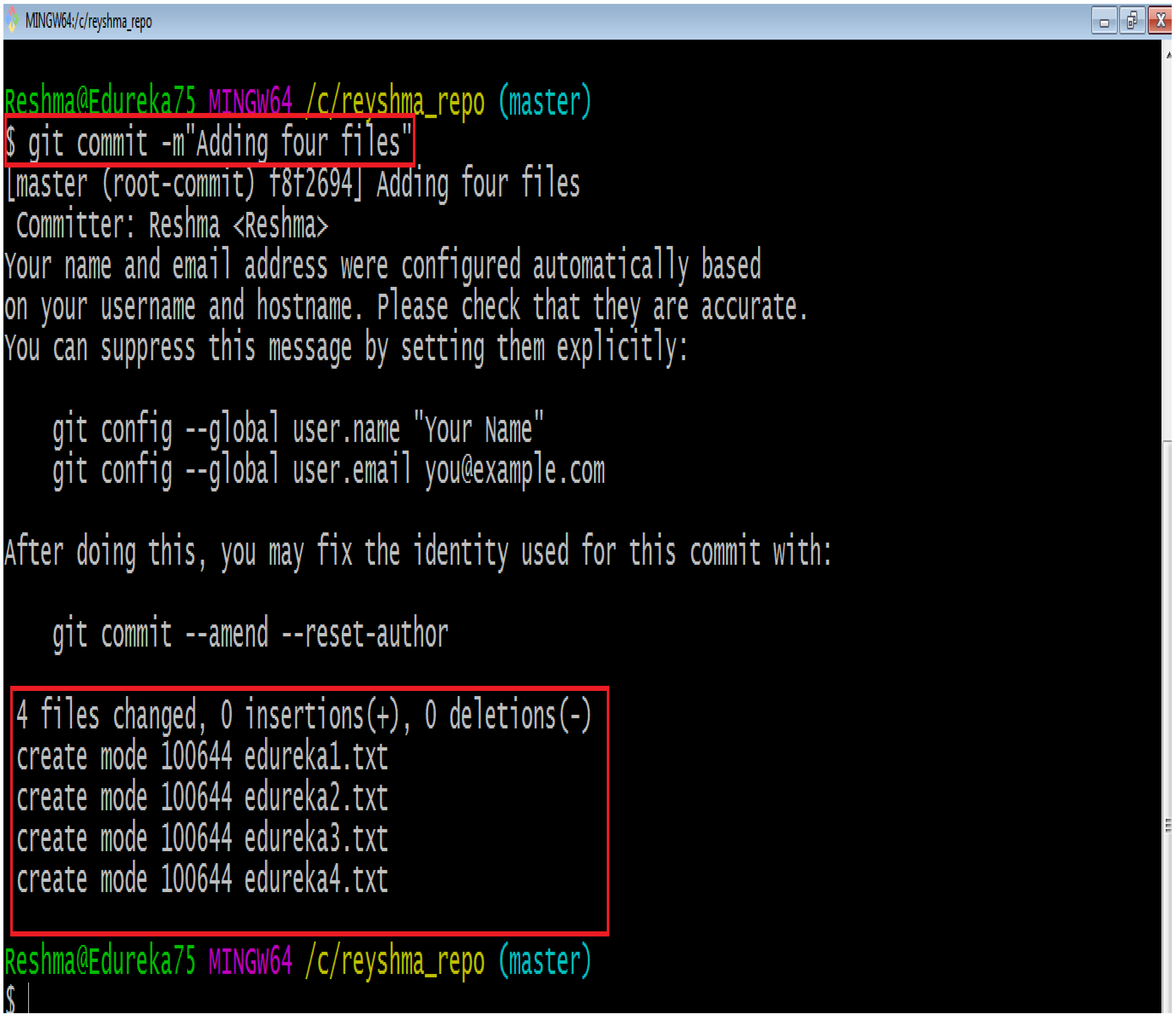
git commit

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.



```
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:

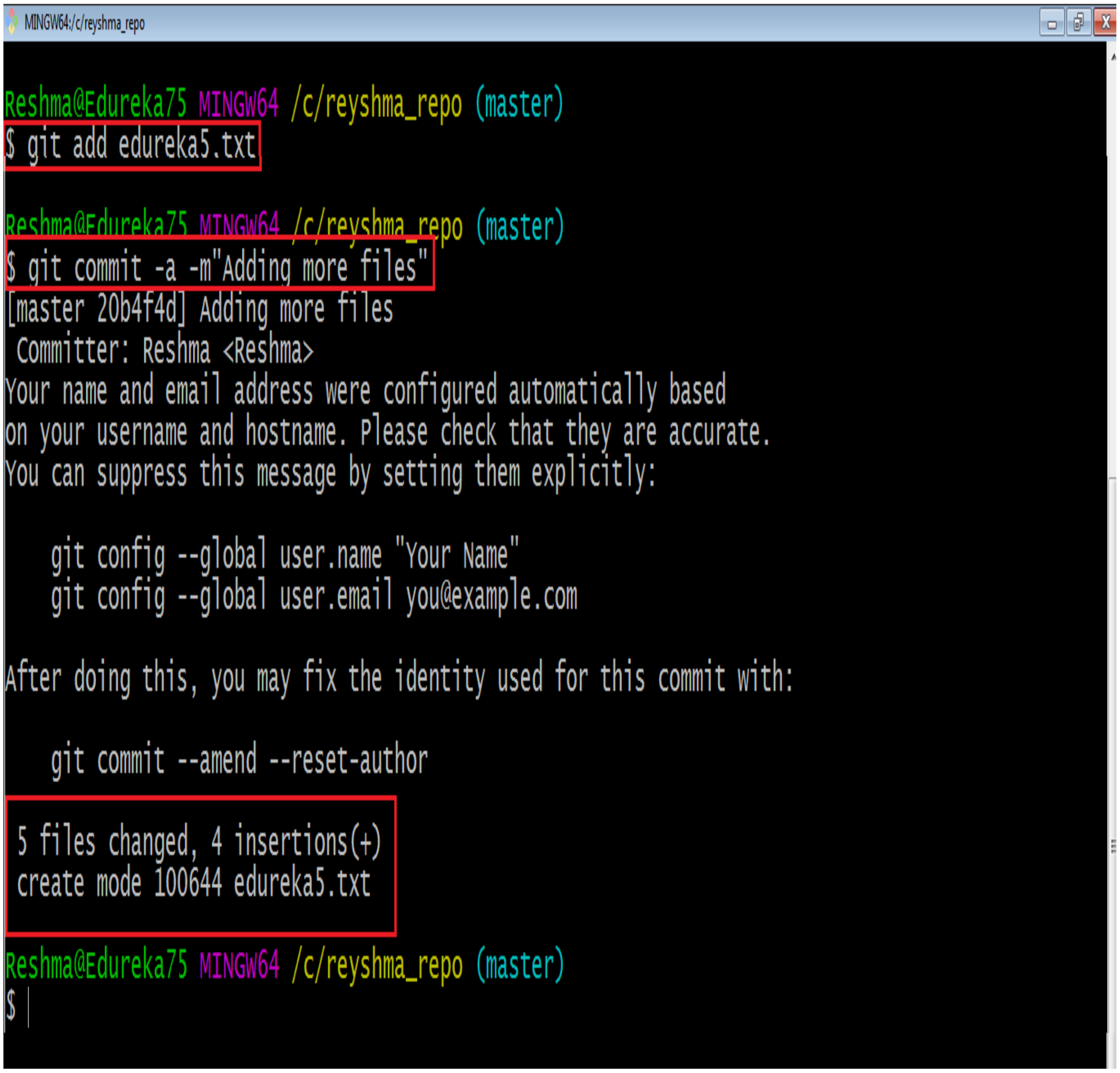
git commit -a

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.



```
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.

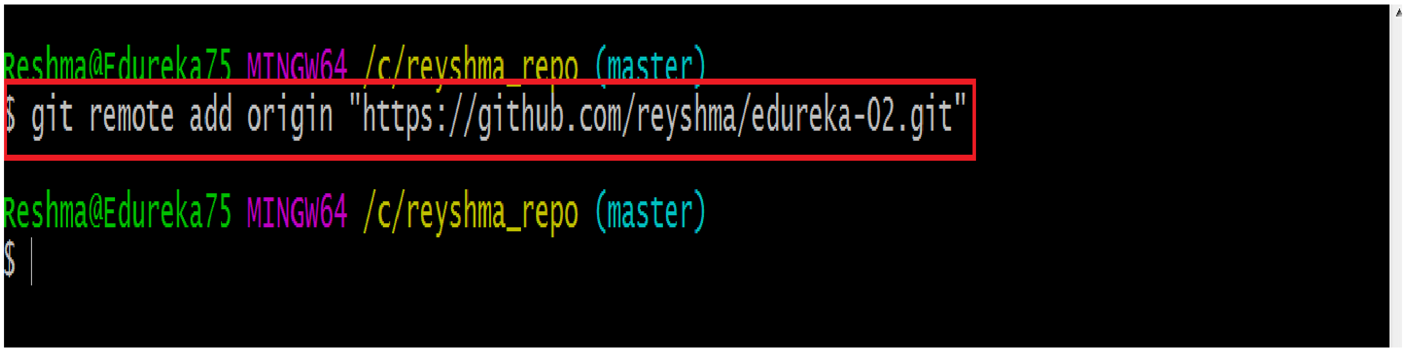
Note that before you affect changes to the central repository you should always pull changes from the central repository to your local repository to get updated with the work of all the collaborators that have been contributing in the central repository. For that we will use the **pull** command.

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>

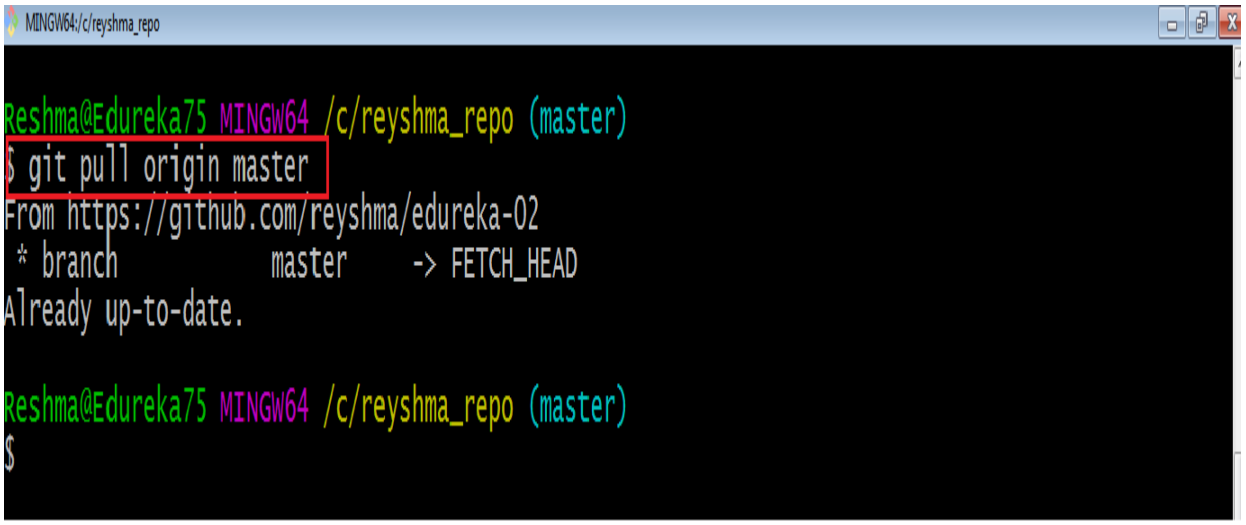


```
Reshma@Edureka75 MINGW64 /c/reyshma_repo (master)
$ git remote add origin "https://github.com/reyshma/edureka-02.git"

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 .

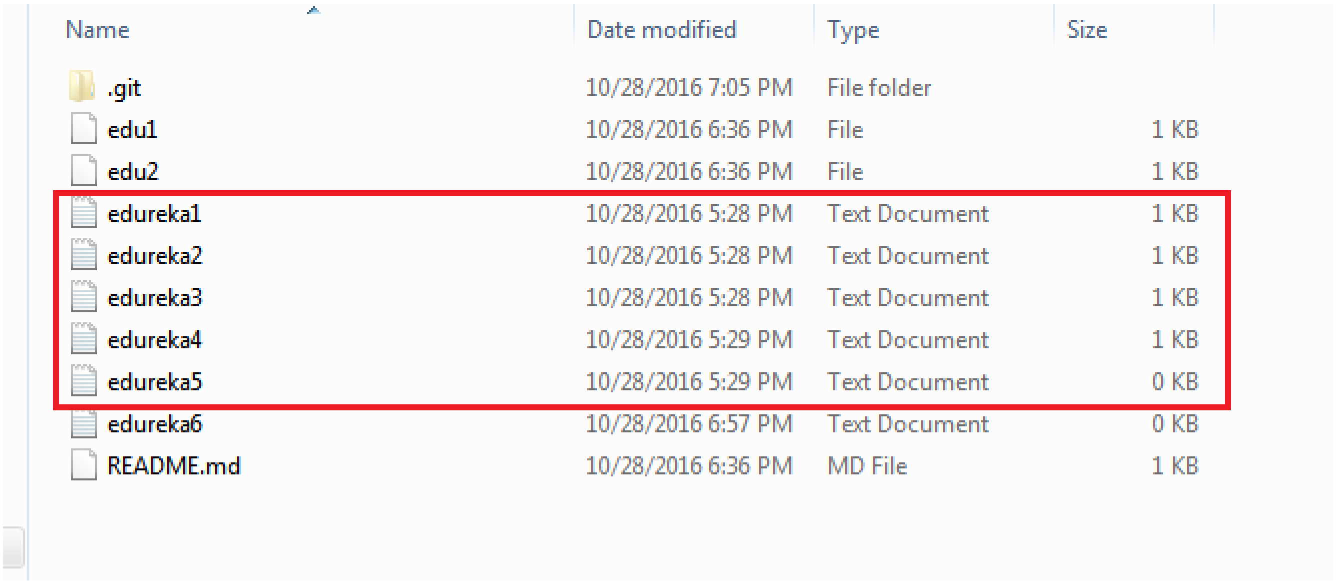
The use of **git push** is to publish your local changes to a central repository. After you've accumulated several local commits and are ready to share them with the rest of the team, you can then push them to the central repository by using the following command:

git push <remote>

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

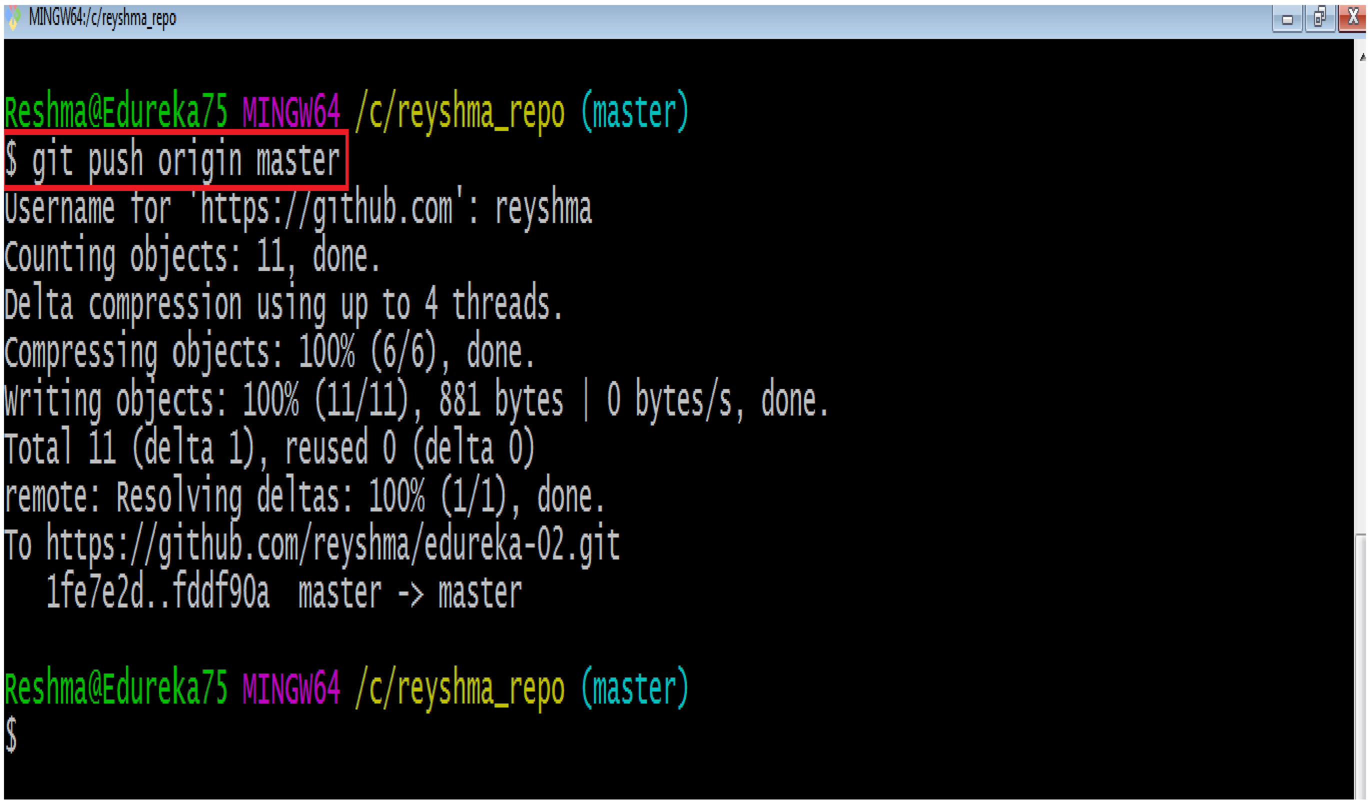
This pushes the changes from the local repository to the remote repository along with all the necessary commits and internal objects. This creates a local branch in the destination repository.

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.



```
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)
$
```

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**.

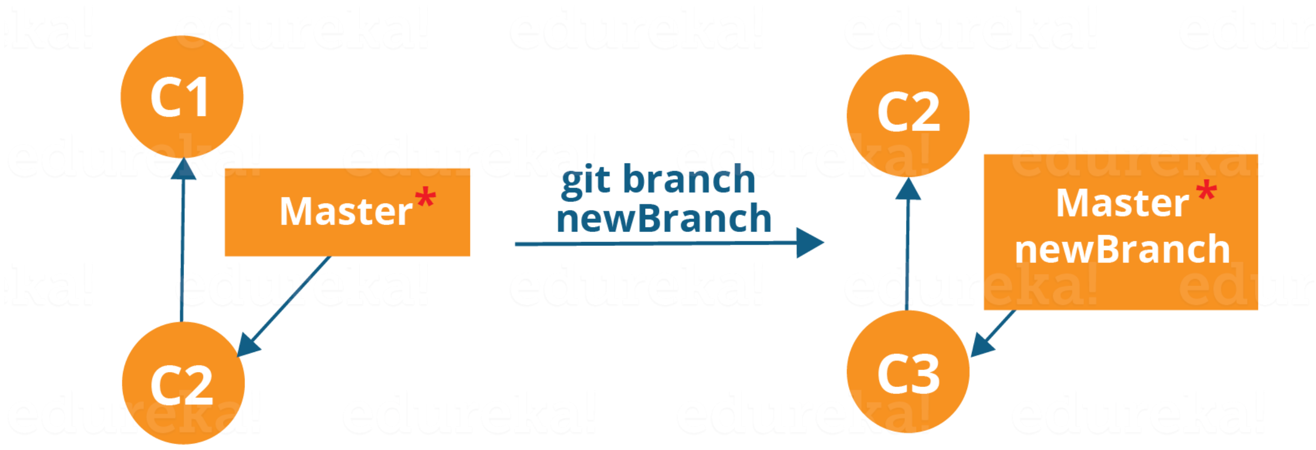
You can check what your current branch is by using the command:

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:

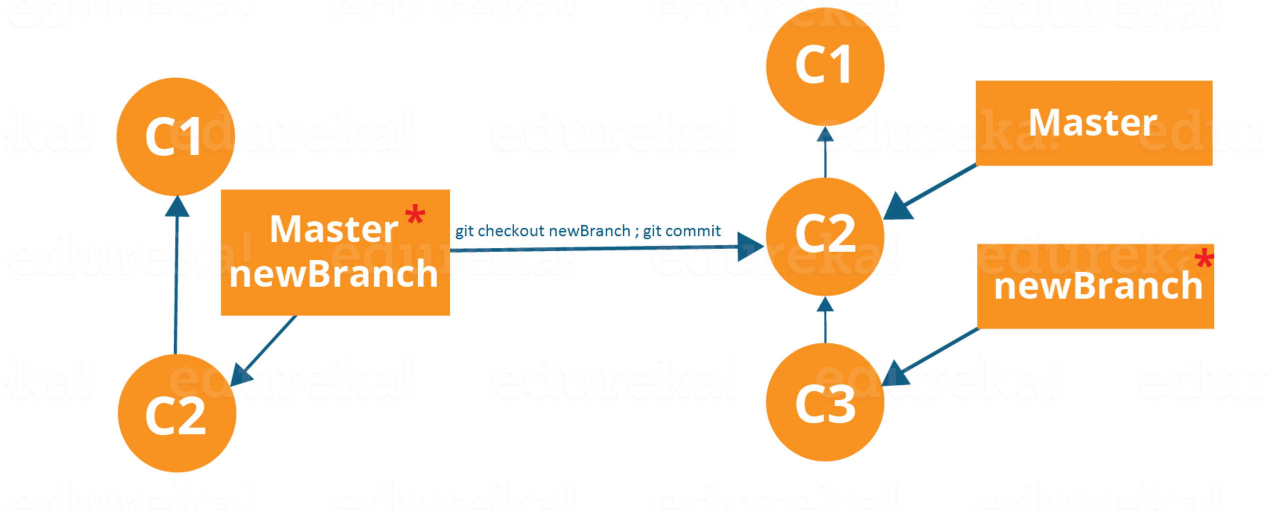
git branch <branch-name>



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.

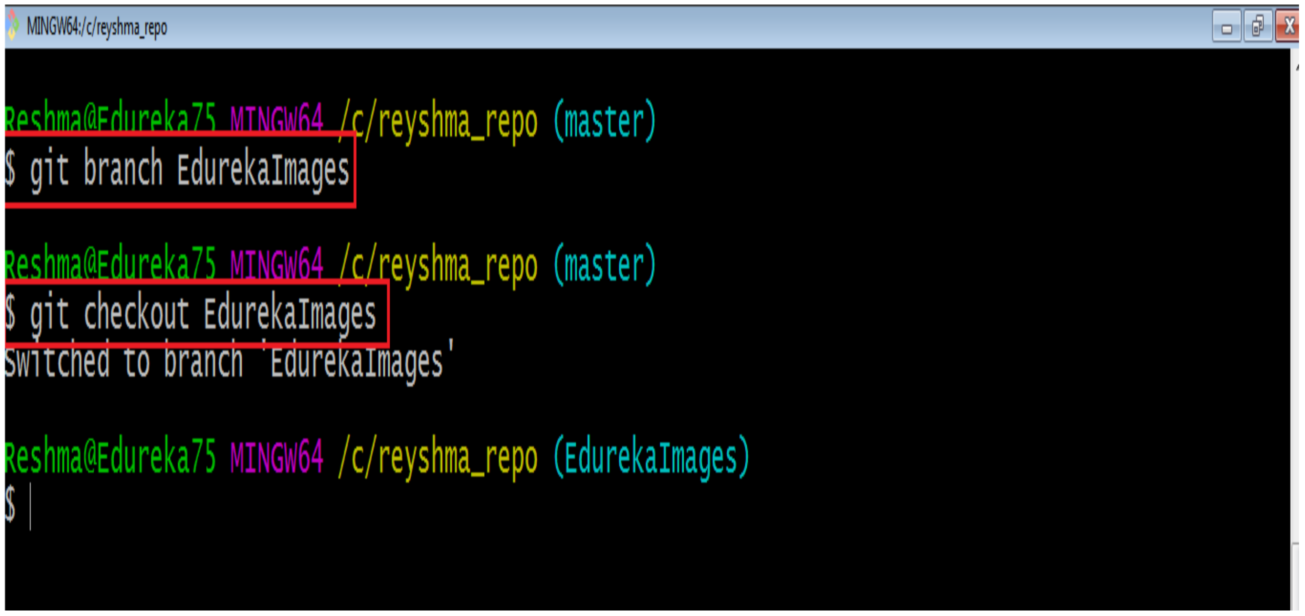


Branching includes the work of a particular commit along with all parent commits. As you can see in the diagram above, the newBranch has detached itself from the master and hence will create a different path.

Use the command below:

git checkout <branch_name> and then

git commit



```
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.

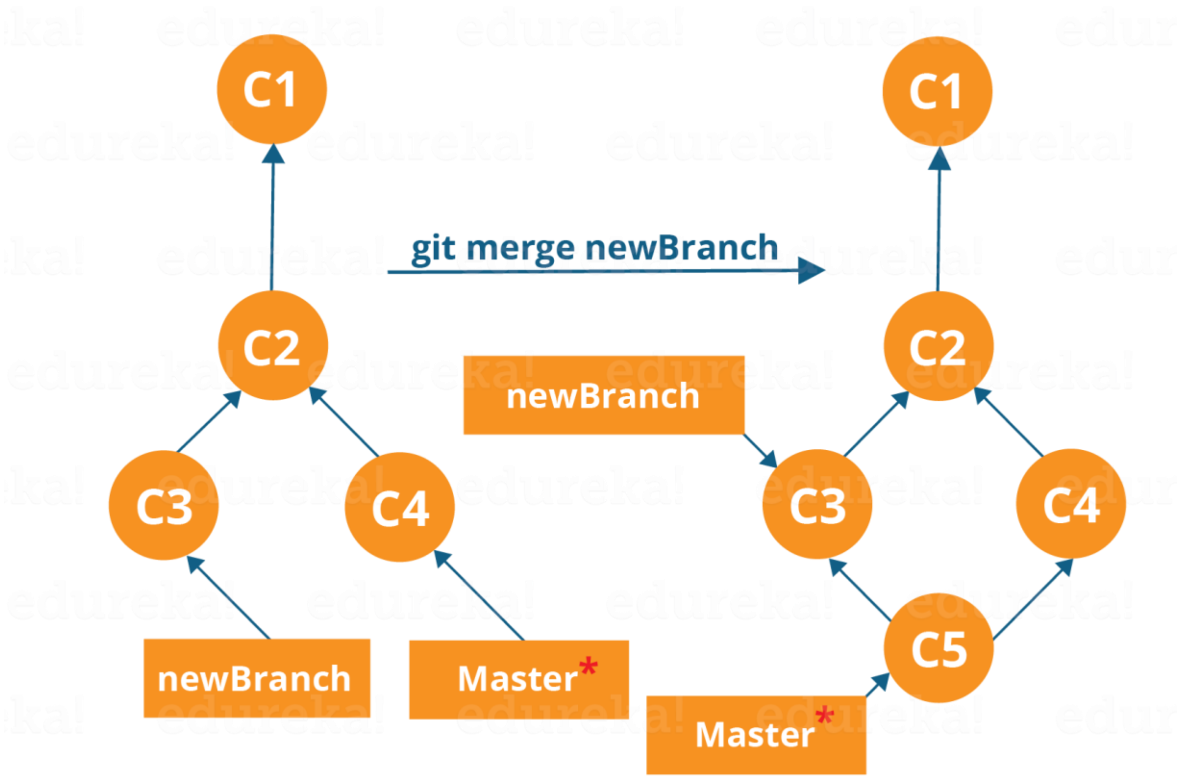
Now while we are in the branch EdurekaImages, add and commit the text file *edureka6.txt* using the following commands:

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.



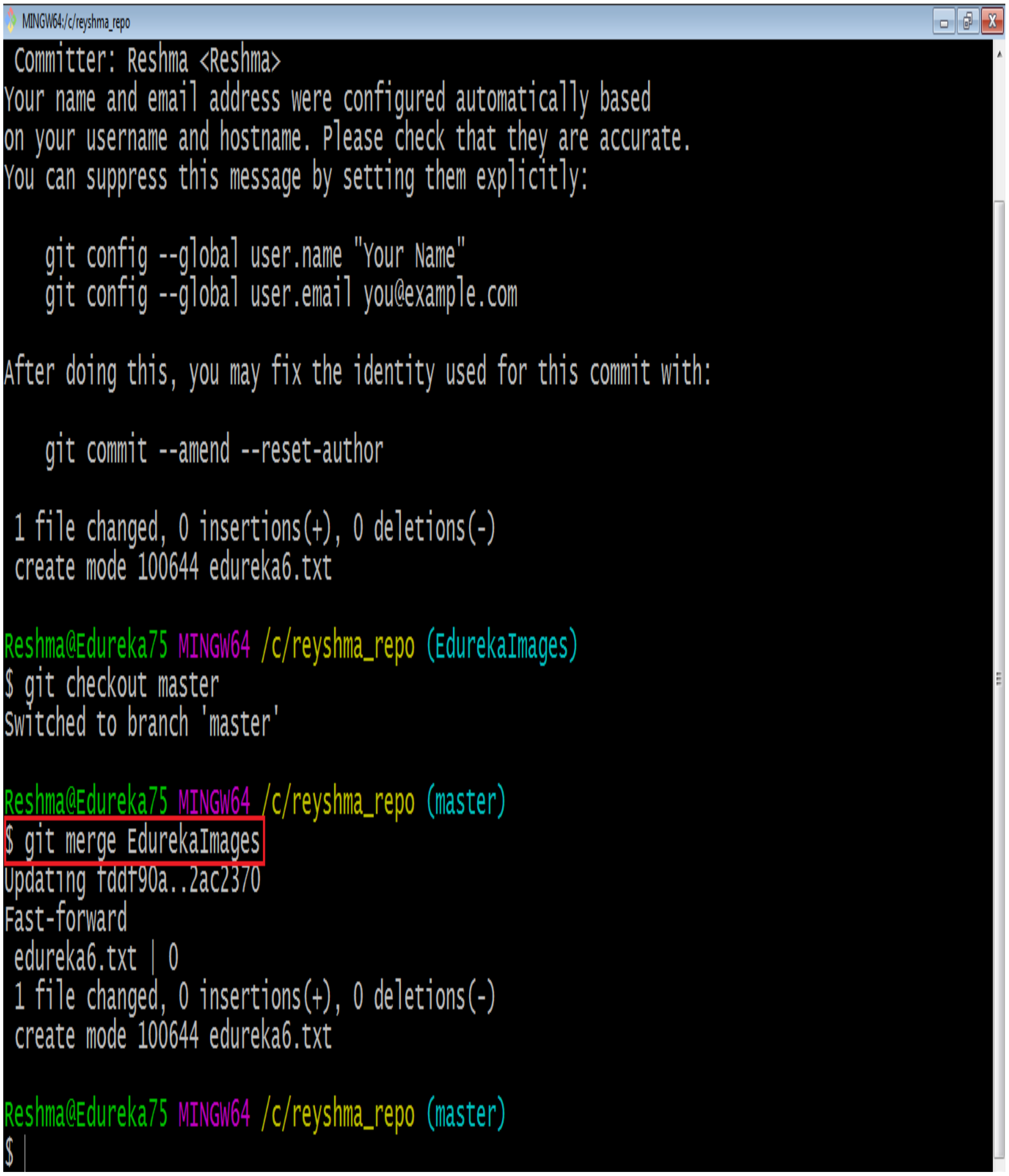
The diagram above shows us two different branches-> newBranch and master. Now, when we merge the work of newBranch into master, it creates a new commit which contains all the work of master and newBranch.

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**



```
 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.

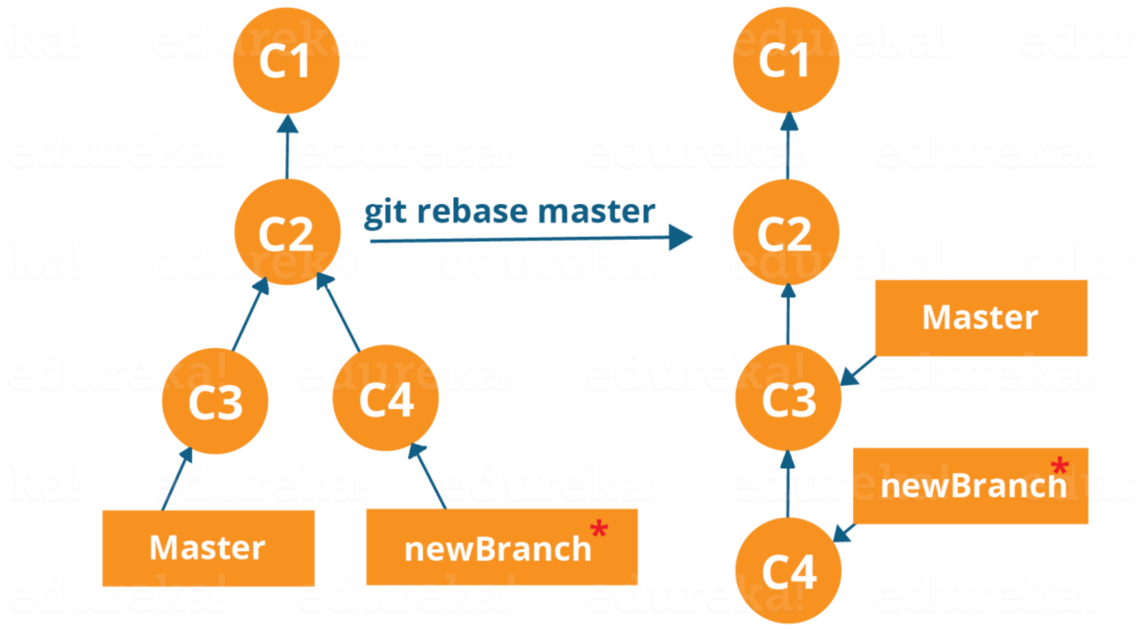
Merging in Git creates a special commit that has two unique parents.

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.

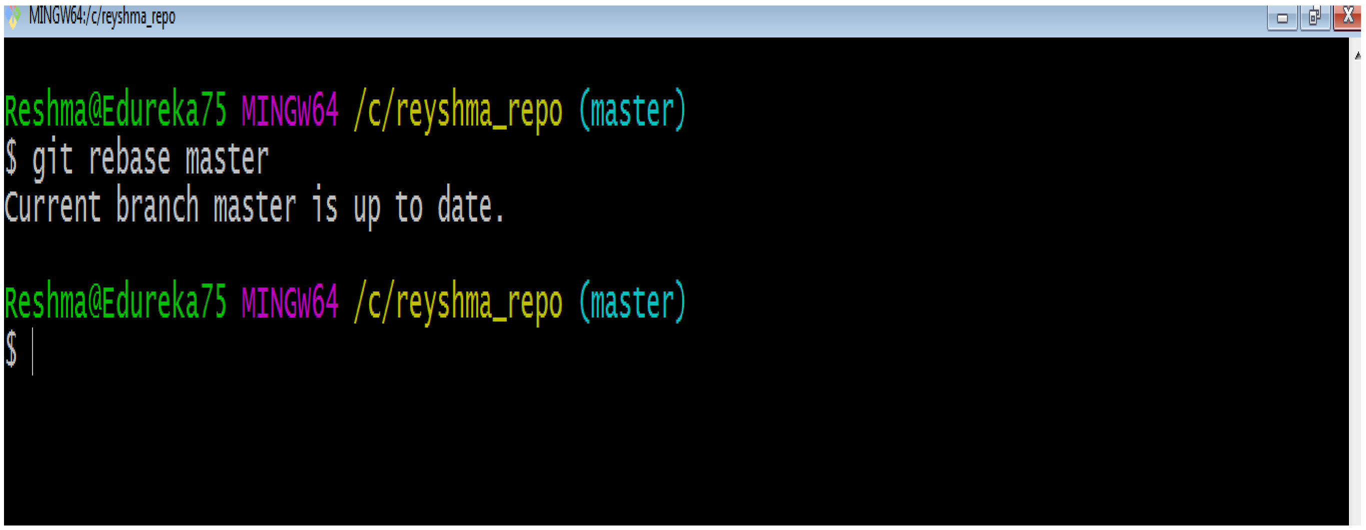


Now, our work from newBranch is placed right after master and we have a nice linear sequence of commits.

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
```