**GIT**

Git is a free, open source distributed version control system tool designed to handle everything from small to very large projects with speed and efficiency. It was created by Linus Torvalds in 2005 to develop Linux Kernel. Git has the functionality, performance, security and flexibility that most teams and individual developers need.

**What is Git – Why Git** **Came Into Existence?**

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:**

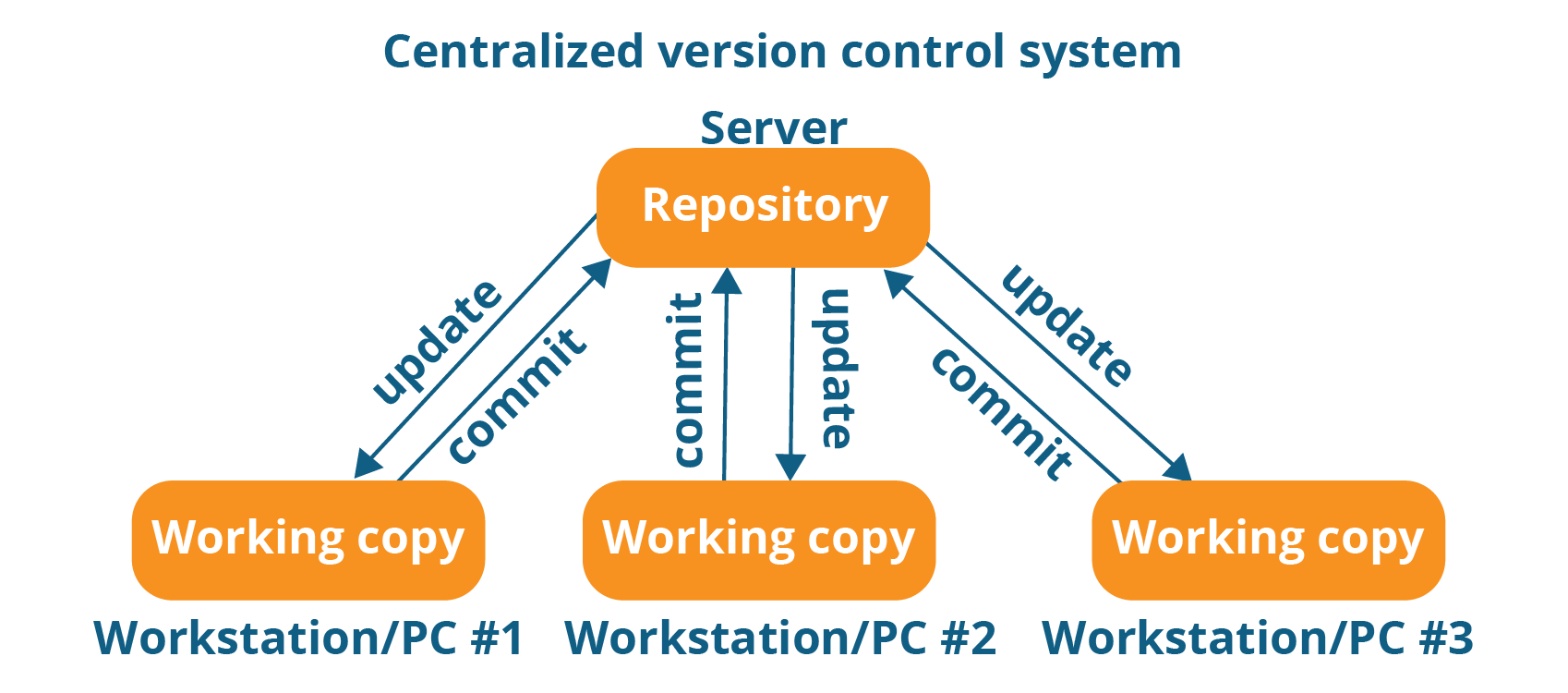
Centralized Version Control System (CVCS)

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.

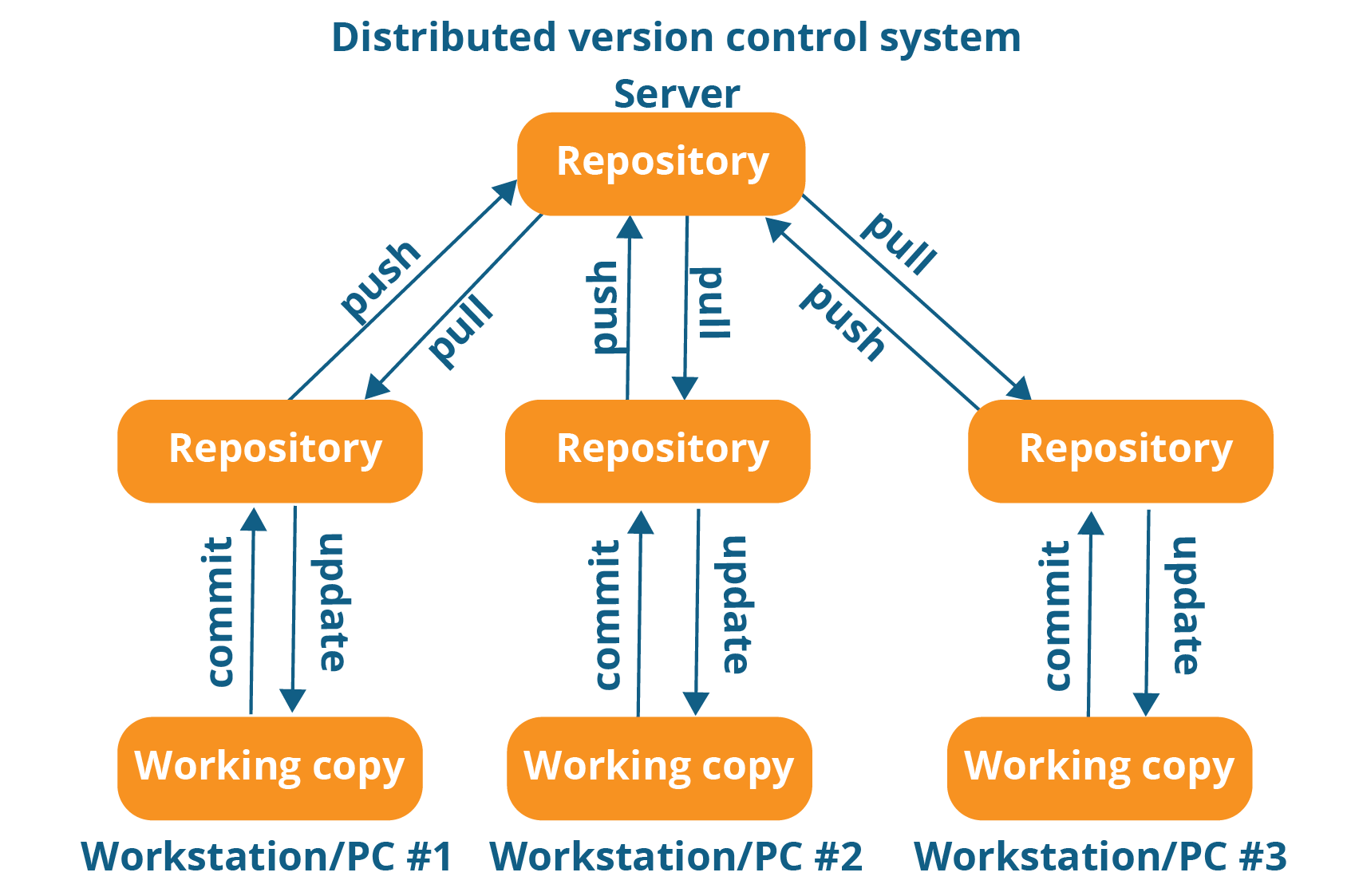
This is when Distributed VCS comes to the rescue.

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.**

Git is a Distributed Version Control tool that supports distributed non-linear workflows by providing data assurance for developing quality software.

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. 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. 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.(not covered)**

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.

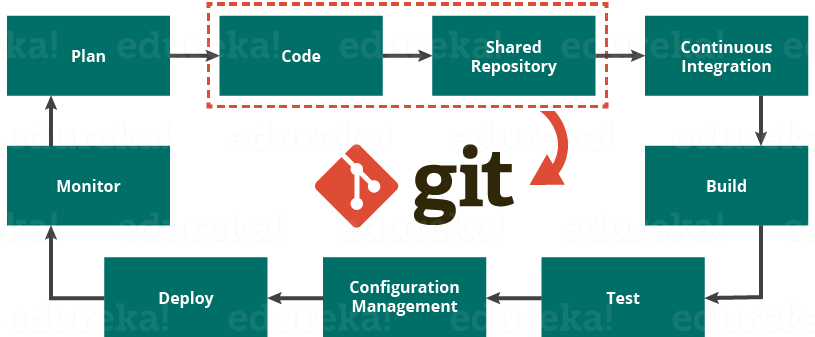
Compatibility with existing systems or protocol Repositories can be published via http, ftp or a Git protocol over either a plain socket, or ssh. Git also has a Concurrent Version Systems (CVS) server emulation, which enables the use of existing CVS clients and IDE plugins to access Git repositories.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

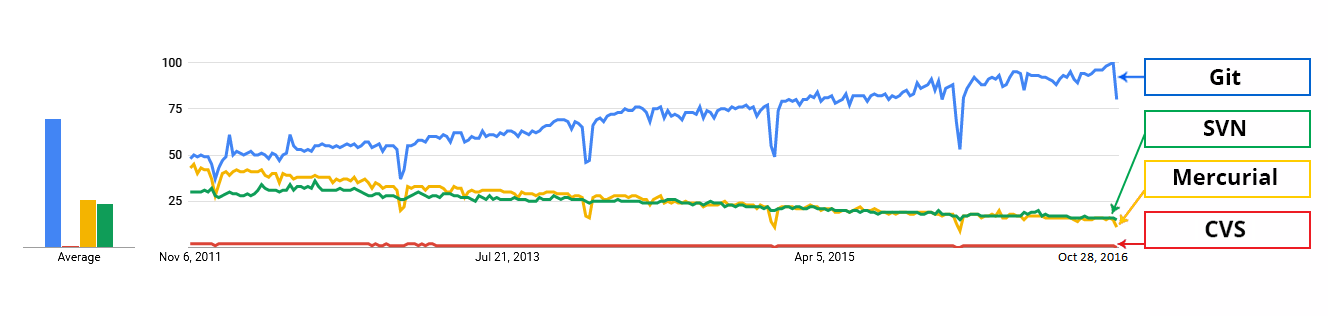


The diagram above shows the entire life cycle of Devops starting from planning the project to its deployment and monitoring. Git plays a vital role when it comes to managing the code that the collaborators contribute to the shared repository. This code is then extracted for performing continuous integration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

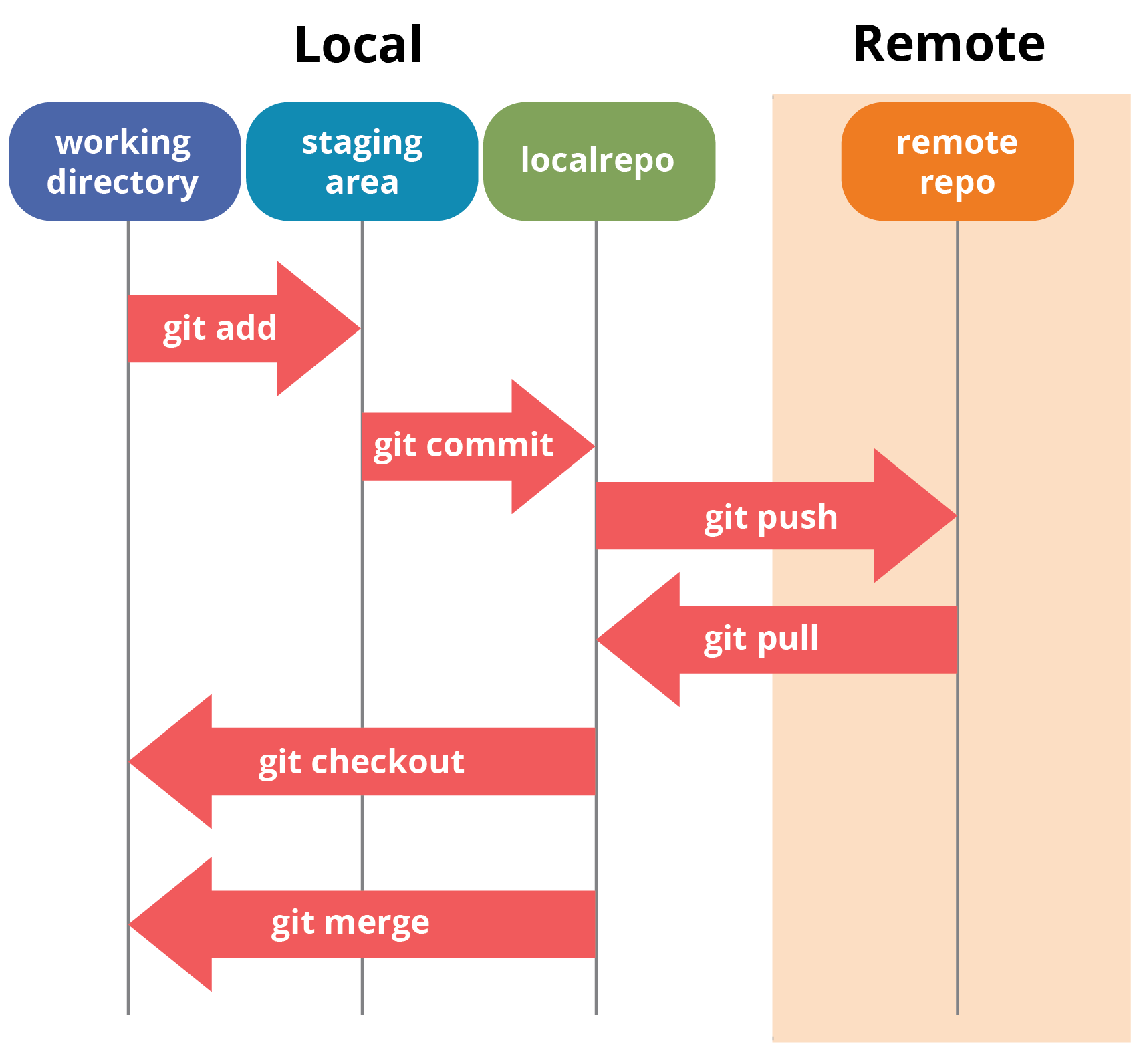
Some of the basic operations in Git are:

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

Some advanced Git operations are:

\*Branching \* Merging \*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:



**Initialize**

In order to do that, we use the command **git init.**

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.

**Git status**

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

This means I cannot commit changes with these files unless I have added them explicitly in the index.

**Add**

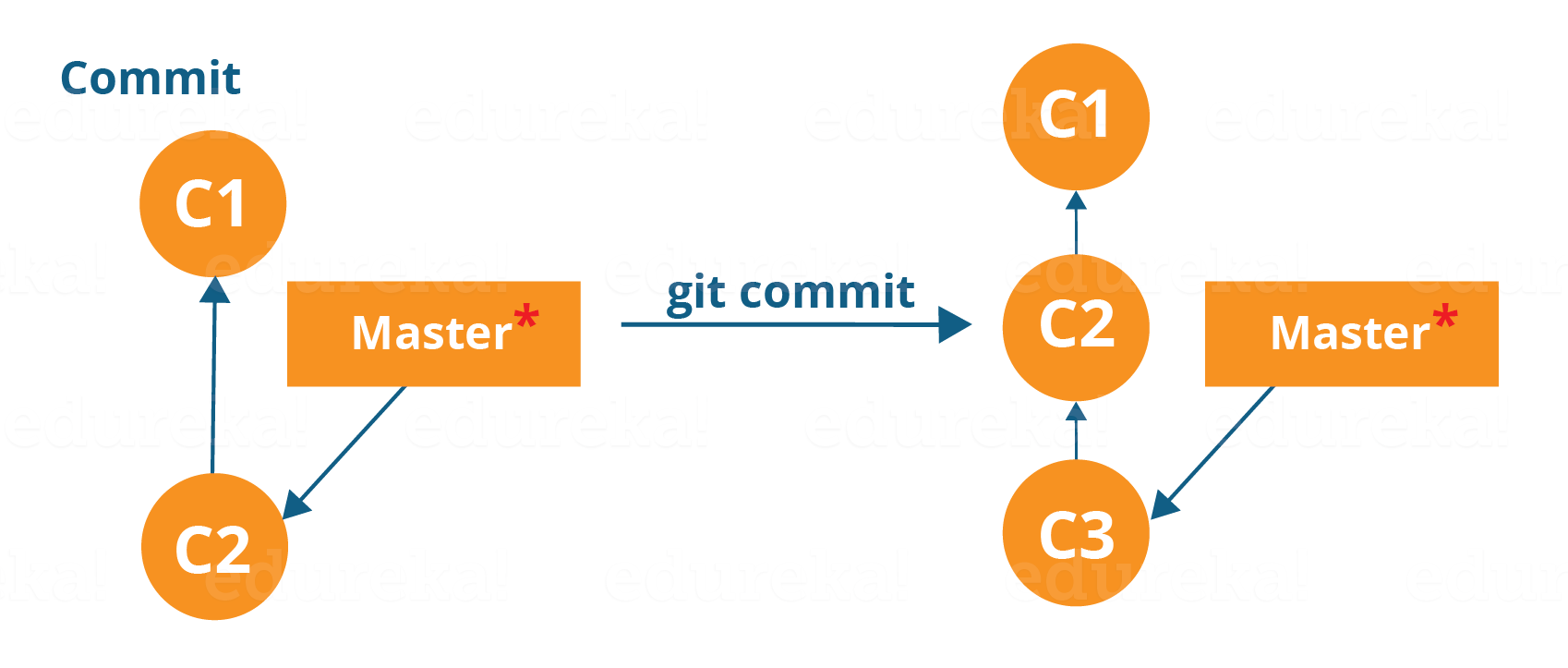
This command updates the index using the current content found in the working tree and then prepares the content in the staging area for the next commit.

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.

**Command for adding directory to the staging area:**

**git add . or git add <directory> or git add file**

**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>”**

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**

**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>**

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.

**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.

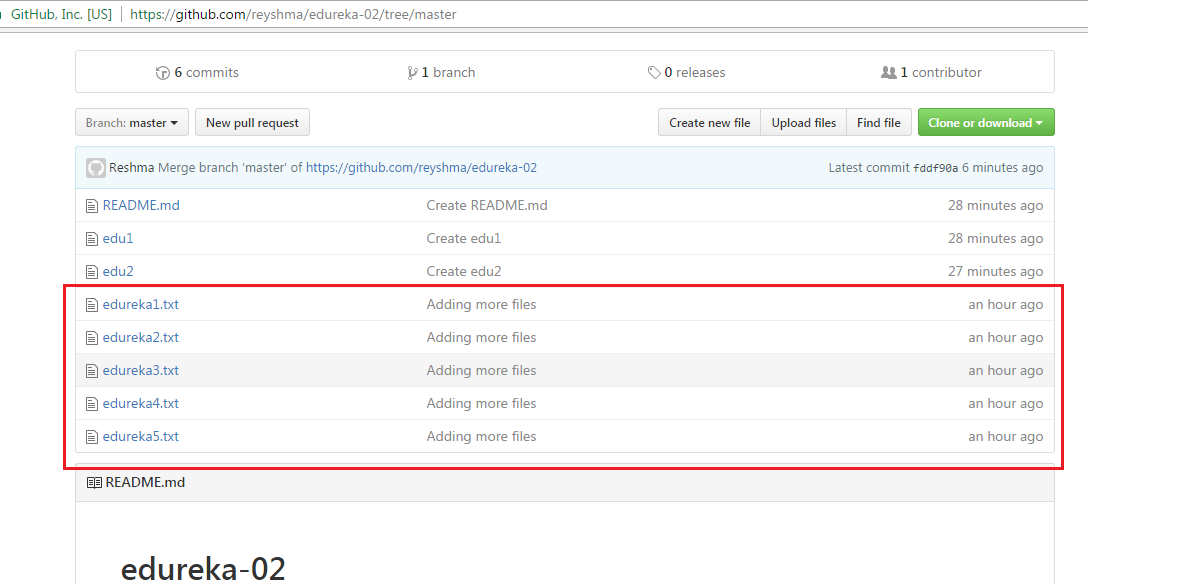
Pulling imports commits to local repositories whereas pushing exports commits to the remote repositories .

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.

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



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.

**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.

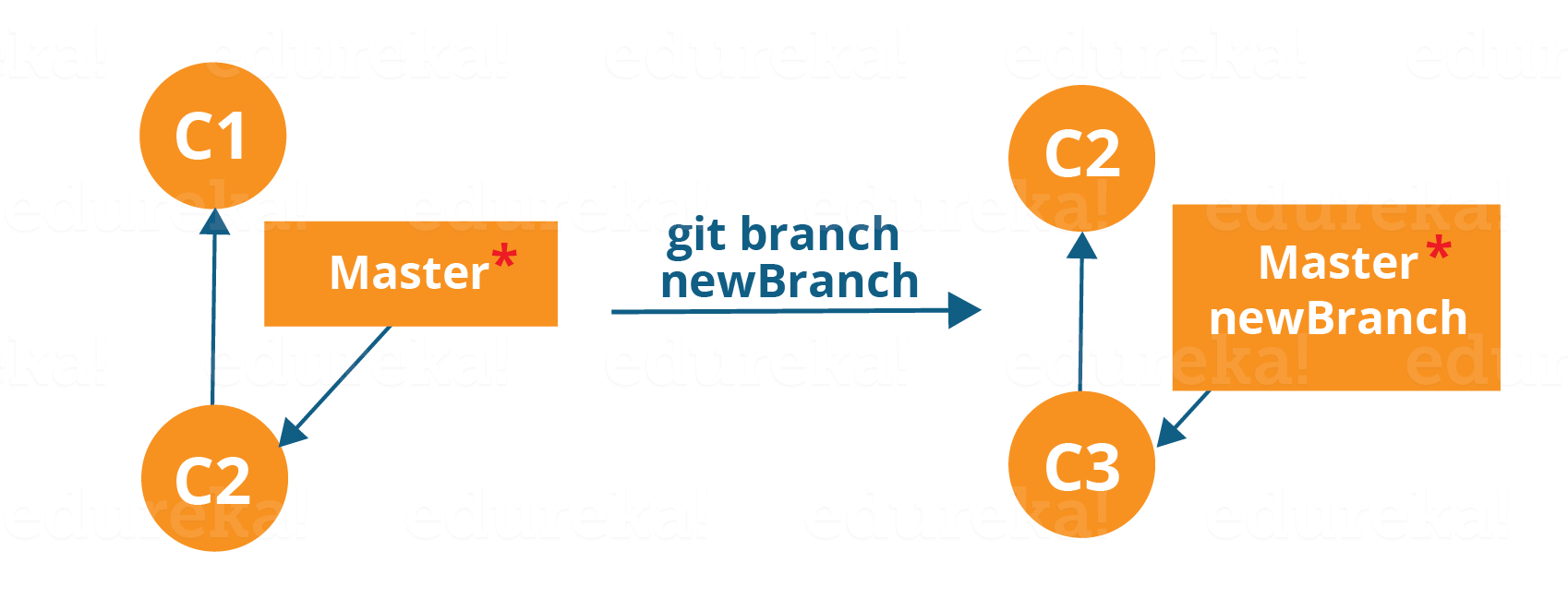
You can check what your current branch is by using the command:

**git branch**

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

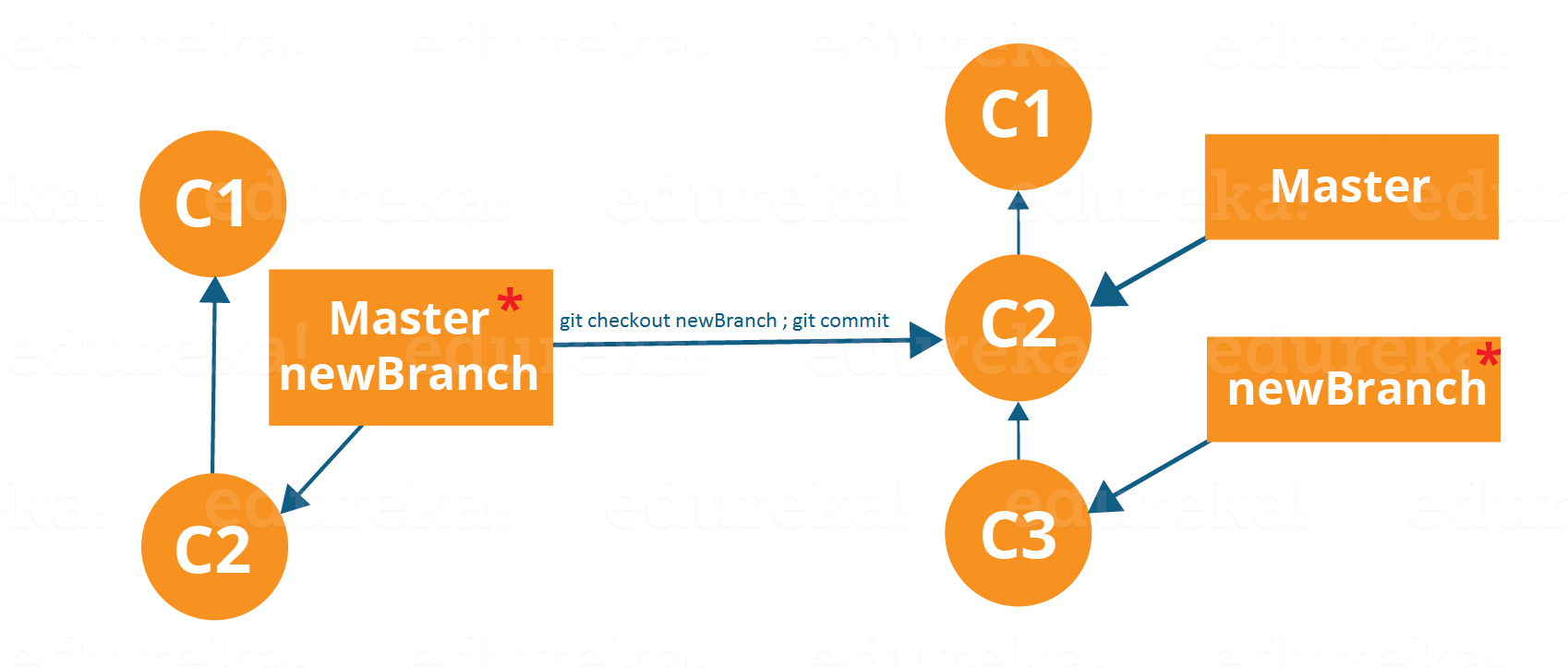
To create a new branch we use the following command:

**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**

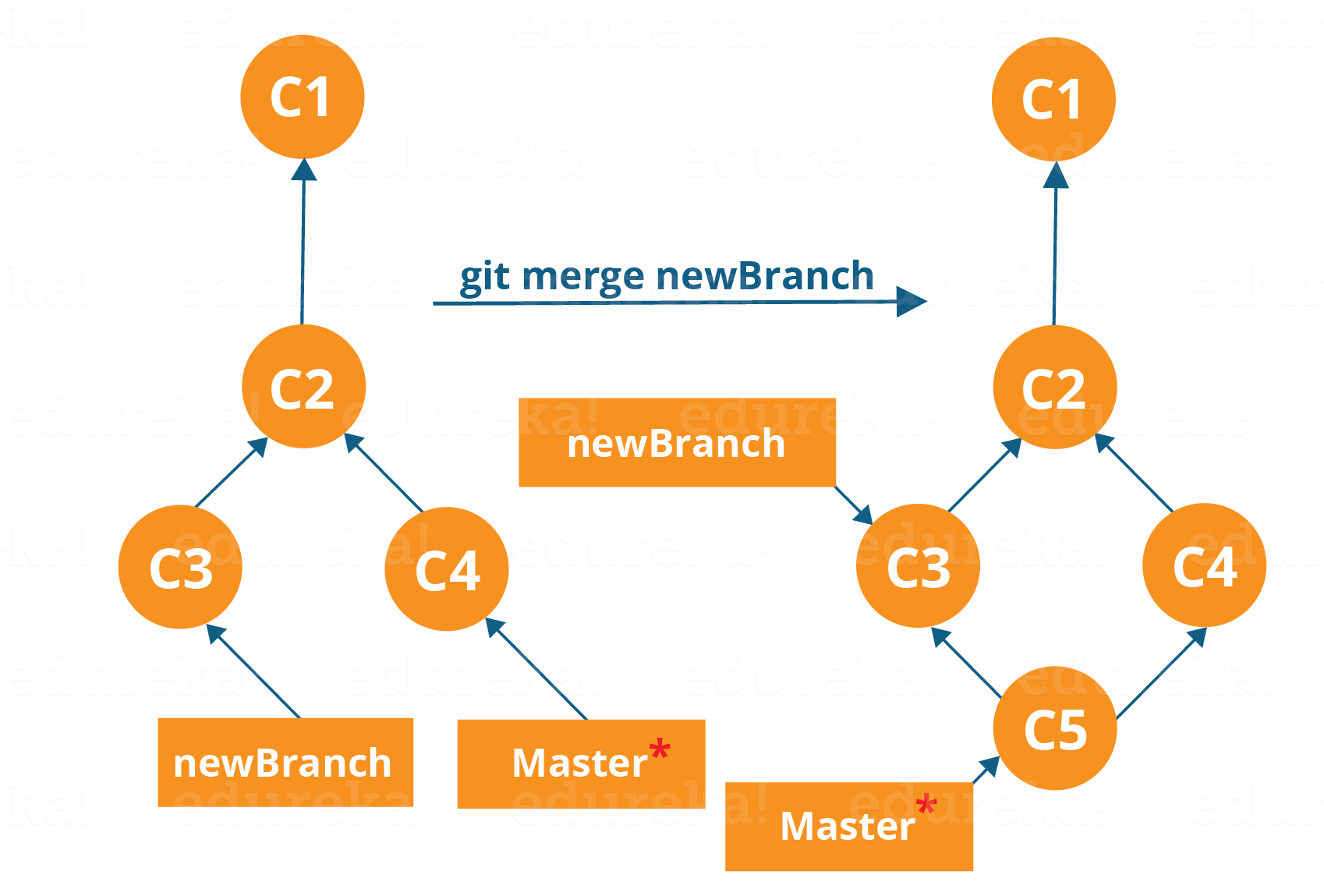
**One shortcut to the above commands is:**

**git checkout -b[ branch\_name]**

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

**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.

Eg: 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**

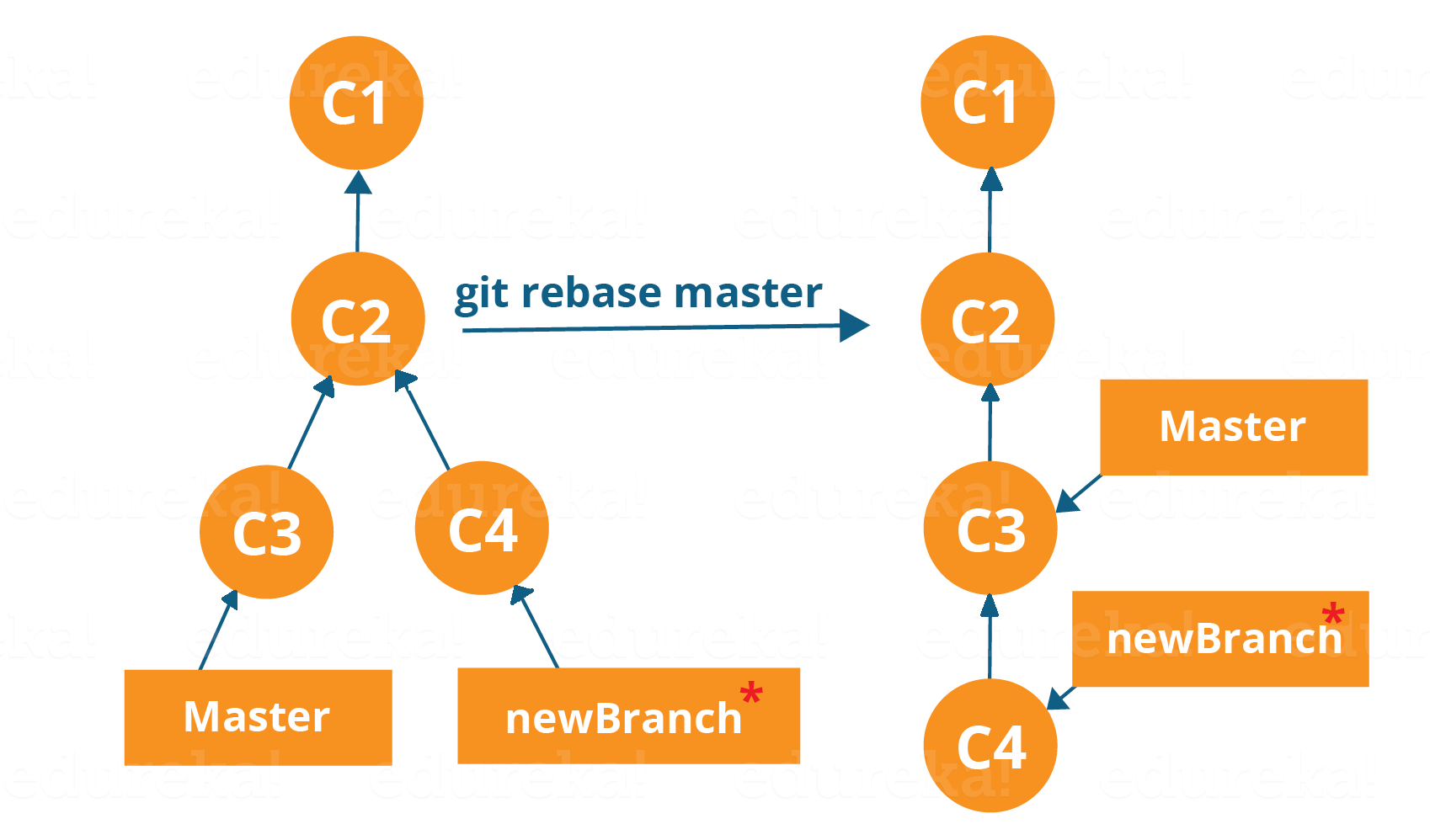
all the data from the branch name are merged to the master branch. Merging in Git creates a special commit that has two unique parents.

**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.

**Command: git rebase 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**

**Some other useful commands:**

**Moving head from one file to another file:**

Advantages: for un commit a file and move to head

Command: **git reset –hard (1e681ad) àhash code**

To see dangling commit: **git fsck - - lost-found**

It display dangling commit

**Without staging how can directly commit:**

**# git commit –a –m “file is added”**

This works only when u modified or changed a file