About Version Control

What is “version control”, and why should you care?

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book, you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert selected files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

### Local Version Control Systems

Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.



Figure 1. Local version control

One of the most popular VCS tools was a system called RCS (revision control system), which is still distributed with many computers today. [RCS](https://www.gnu.org/software/rcs/) works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

### Centralized Version Control Systems

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems (such as CVS, Subversion, and Perforce) have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.



Figure 2. Centralized version control

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what, and it’s far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on. If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything — the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCSs suffer from this same problem — whenever you have the entire history of the project in a single place, you risk losing everything.

### Distributed Version Control Systems

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don’t just check out the latest snapshot of the files; rather, they fully mirror the repository, including its full history. Thus, if any server dies, and these systems were collaborating via that server, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

Apache Subversion, Azure DevOps server



Figure 3. Distributed version control

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren’t possible in centralized systems, such as hierarchical models.

## A Short History of Git

As with many great things in life, Git began with a bit of creative destruction and fiery controversy.

The Linux kernel is an open source software project of fairly large scope. During the early years of the Linux kernel maintenance (1991–2002), changes to the software were passed around as patches and archived files. In 2002, the Linux kernel project began using a proprietary DVCS called BitKeeper.

In 2005, the relationship between the community that developed the Linux kernel and the commercial company that developed BitKeeper broke down, and the tool’s free-of-charge status was revoked. This prompted the Linux development community (and in particular Linus Torvalds, the creator of Linux) to develop their own tool based on some of the lessons they learned while using BitKeeper. Some of the goals of the new system were as follows:

* Speed
* Simple design
* Strong support for non-linear development (thousands of parallel branches)
* Fully distributed
* Able to handle large projects like the Linux kernel efficiently (speed and data size)

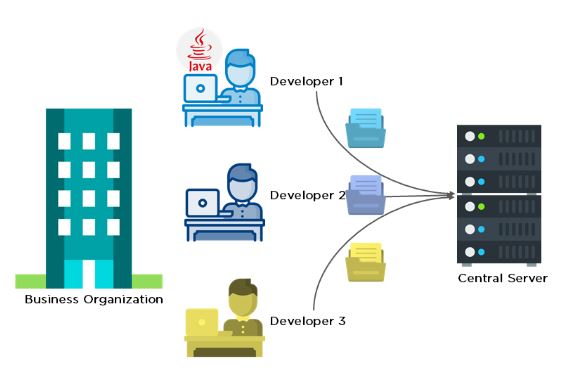
Since its birth in 2005, Git has evolved and matured to be easy to use and yet retain these initial qualities. It’s amazingly fast, it’s very efficient with large projects, and it has an incredible branching system for non-linear development

What is GIT

[Git](https://www.simplilearn.com/tutorials/git-tutorial/git-tutorial-for-beginner) is a [DevOps tool](https://www.simplilearn.com/tutorials/devops-tutorial/devops-tools) used for source code management. It is a free and open-source version control system used to handle small to very large projects efficiently. Git is used to tracking changes in the source code, enabling multiple developers to work together on non-linear development. Linus Torvalds created Git in 2005 for the development of the [Linux](https://www.simplilearn.com/linux-programming-for-beginners-article) kernel.

Before diving deep, let’s explain a scenario before Git:

* Developers used to submit their codes to the central server without having copies of their own
* Any changes made to the source code were unknown to the other developers
* There was no communication between any of the developers



Now let’s look at the scenario after Git:

* Every developer has an entire copy of the code on their local systems
* Any changes made to the source code can be tracked by others
* There is regular communication between the developers



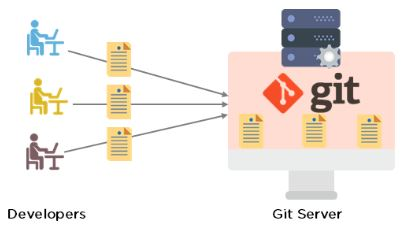
## **What is Git?**

Git is a version control system used for tracking changes in computer files. It is generally used for source code management in software development.

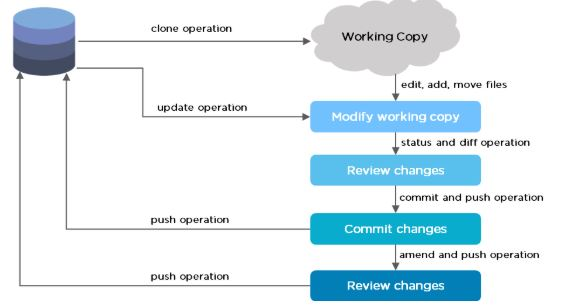
* Git is used to tracking changes in the source code
* The distributed version control tool is used for source code management
* It allows multiple developers to work together
* It supports non-linear development through its thousands of parallel branches

## **Features of Git**

* Tracks history
* Free and open source
* Supports non-linear development
* Creates backups
* Scalable
* Supports collaboration
* Branching is easier
* Distributed development

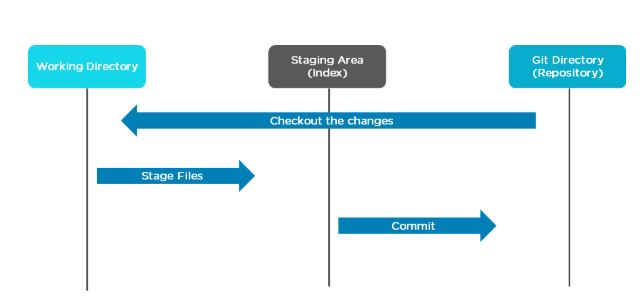


**Git Workflow**

****

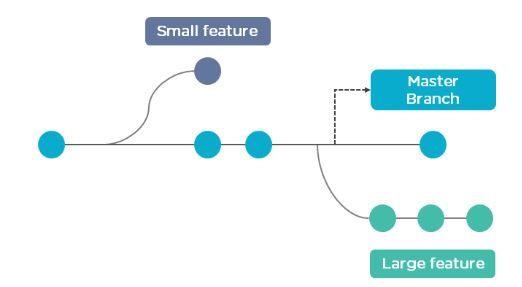
The Git workflow is divided into three states:

* Working directory - Modify files in your working directory
* Staging area (Index) - Stage the files and add snapshots of them to your staging area
* Git directory (Repository) - Perform a commit that stores the snapshots permanently to your Git directory. Checkout any existing version, make changes, stage them and commit.



## **Branch in Git**

Branch in Git is used to keep your changes until they are ready. You can do your work on a branch while the main branch (master) remains stable. After you are done with your work, you can merge it with the main office.



The above diagram shows there is a master branch. There are two separate branches called “small feature” and “large feature.” Once you are finished working with the two separate branches, you can merge them and create a master branch.