Transcript

Module 2: Introduction to Git

Video 1: Module Overview

This module introduces to you the concept of version control and how to work with a distributed version control system such as Git. Version control is an essential knowledge and skill for software engineering and AI solutioning. It enables you to develop large-scale software applications and complex AI models in a team-based setting, and to enhance the quality of the development process and output. Version control also lies at the heart of modern DevOps, DataOps and MLOps practices. In particular, the ability to perform continuous integration and continuous delivery/continuous deployment requires that the code base is integrated in a central location and thoroughly tested.

In the subsequent videos, I will explain the core concepts of distributed version control and Git as well as walk you through the complete lifecycle of creating a repository and tracking changes made to the repository. Sit tight and fasten your seat belt!

Video 2: Overview of Version Control System

Welcome to the introductory video on version control system or VCS. Version control refers to the systematic approach of recording changes to a set of files over time. In this course, our main focus is on files containing source code written to develop software applications and create machine learning models. A version control system is a piece of software that is used to perform version control such that we can revert selected files to their original state and to compare changes made to the files over time. Version control can be performed manually by copying files into backup folders. But, using a VCS is certainly more productive and effective.

Version control is extremely useful and important in modern AI solutioning for two main reasons. First, AI projects typically involve a large number of source files. Tracking changes made to these files becomes increasingly difficult. Second, AI projects usually involve a team of multiple developers due to the scale and complexity of real-world AI solutions. Coordinating and integrating changes made to different files across multiple developers is a nontrivial task. Let’s find out more about VCS.

Local VCS is the most basic type of VCS involving the use of a simple database that keeps track of all changes made to files under version control. The database and files repository reside on the laptop of a developer. Although Local VCS is better than manual version control, it does not allow collaboration with other developers Local VCS also suffers from the problem of single point of failure. If the database and file repository are damaged, the developer will lose everything.

Centralised VCS improves upon Local VCS by moving the database to a separate server computer. This approach allows multiple developers to check out files from the centralised server onto their own computers for development. For a long time, Centralised VCS became the standard for version control. Other advantages of Centralised VCS include providing developers with visibility on everybody’s development progress and simplifying administration as compared to multiple databases on the computers of every developer. However, if the server is down, nobody can check out or commit files. Centralised VCS also suffers from the problem of single point of failure. If the database and file repository are corrupted, it is not possible to restore them.

Distributed VCS mitigates the disadvantages of Centralised VCS with a more elaborate architecture. The notion of a server computer hosting the database and file repository remains. But when developers check out the latest version of files, they will also copy a full mirror of the database and file repository onto their respective computer. This mirror or local repository includes the full history of the server's repository, which is known as the remote repository. Thus, a developer can continue with development even if the server is offline. More importantly, if the server is corrupted, any of the local repositories can be used to restore the server's remote repository. This approach effectively mitigates the single point of failure problem and enables developers to collaborate.

Distributed VCS has replaced Centralised VCS as the gold standard for version control. Today, Distributed VCS tools such as Git and Mercurial are widely used for AI solutioning and software engineering projects. I will introduce you to Git in the next video.

Video 3: Introduction to Git and GitHub

This video introduces you to Git, one of the most popular Distributed Version Control System or VCS with a global market share of almost 90%. Git originated from the Linux development community to manage the codebase of this large-scale open-source operating system. Just imagine for once, if Git can be used to track and distribute changes to Linux with over 30 million lines of code, there is no other project that cannot be efficiently managed by Git.

Git is designed with several goals in mind. First, let's talk about speed, Git is designed to be amazingly fast in terms of tracking changes and storing data. Next, Git features a simple and efficient storage mechanism that avoids redundant data. Third, Git provides excellent support for branching to enable non-linear development. It is possible to have thousands of parallel branches in a large project. Last but not least, it is fully distributed and scale up well to handle large projects with many developers. Let's move on to find out more about the working mechanisms of Git.

I will begin by explaining the storage mechanism of Git by comparing with other traditional VCS. A traditional VCS stores data as a set of files and changes made to each file over time. This approach results in a list of file-based changes or patch sets, and is known as delta-based version control. This approach suffers from several weaknesses. To open a file just for reading, it is necessary to take the base version of the file and apply all the patch sets incrementally. For example, to open File C, the VCS needs to apply patch sets 1, 2 and 3 to the base file. This is a time-consuming task. Moreover, if a particular patch set is corrupted, it may not be possible to reconstruct the affected file fully.

In comparison, Git stores each file as a series of snapshots instead of differences. Each time the state of a project is saved or committed, Git takes a snapshot of what the complete files look like at that moment and stores a reference to that snapshot. To be efficient, Git does not store a file again if it has not been changed since the last commit. Rather, Git stores a link to a previous version of the file it has already stored. In this regard, Git handles data more like a stream of snapshots and resembles a miniature file system. For example, to open Version 5 of File C for reading or editing, Git simply loads snapshot C3. If the developer needs to know the changes made between Version 5 and Version 4 of File C, Git will work backward to determine the differences between snapshot C3 and C2. Viewing the changes is an exception rather than the norm. Thus, Git usually operates very fast when loading files for a typical coding session.

Git also exhibits several other exemplary characteristics beyond the storage mechanism. For instances, nearly every operation in Git is local requiring only local files and resources. No information from another computer is needed. This is mainly because each developer's local repository contains a complete development history of the project. Thus, the developer can continue working even if offline. Interaction with the server or remote repository is required only when sharing changes with, or fetching changes made by, other developers.

Git also check-sum all files using a cryptographic hashing algorithm before storing them. It is thus impossible to change the content of any file or directory without Git's knowledge. Thus, Git provides a very level of data integrity.

Finally, nearly all actions performed in Git only adds data to its repository. Thus, most actions can be undone and the chance of losing data is extremely low.

With these understandings of Git and its working mechanisms, you are ready to find out more about the structure of a Git repository in the next video.

Video 4: Overview of a Git Repository

This video provides you with an overview of a Git repository. A Git repository is essentially a virtual storage of your project consisting of three main sections. Git has three main states that files can reside in. The three main sections of a Git repository correspond to files in each of these three states.

Files in a Git repository can be broadly divided into tracked files and untracked files. Tracked files are those already included in previous commits or added to the Git repository, i.e., files that Git are aware of. Untracked files are those that have not yet been committed or added to the Git repository. Untracked files are usually newly created files.

Tracked files can reside in one of three states. Modified files are files that have been changed but not yet committed. Staged files are modified files that have been marked to go into the next commit snapshot. Committed files are those that have been safely stored into the local repository.

The working directory of a Git repository contains modified files. Of course, the working directory also contains unmodified files that have been checked out but not yet changed. The staging area contains staged files. Think of the staging area as a special file in the repository that stores information about other files that will go into the next commit. The .git directory contains committed files. This is also where Git stores the metadata and object database for the project.

Files are moved across the three sections of the repository as their respective state changes in a typical coding session. Committed files in the .git directory are moved to the working directory after being checked out. These files are unmodified initially but gradually become modified as the developer edits them. The developer then gradually stages the modified files for the next commit. As and when the developer has finalised the changes to the files and reached a stable point, the developer will proceed to commit the files. This action will move a copy or snapshot of each staged file into the .git directory. It is important to note that new files or modified files that have not been staged will not go into the next commit.

Thus, the basic Git workflow entails modifying files in the working directory, selectively staging those changes to be part of the next commit and performing a commit to store the staged files safely into the .git directory. This workflow repeats itself as the developer continues to make more changes to files in the project.

In summary, a particular version of a file is considered committed if it is stored in the .git directory. If it was changed since it was checked out but has not been staged, it is modified. If it has been modified and was added to the staging area, it is staged.

Let's move on to find out how to obtain a Git repository. There are two ways to do so. The first approach is to create a new repository using the git init command. The git init command takes a directory or folder on your computer and turns it into a new Git repository. This is essentially a local repository. The second approach is to make a copy of a remote repository residing on a server computer using the git clone command. This will also create a local repository on your computer. We will discuss more about the cloning of a remote repository in a future video.

You now have sufficient knowledge of Git to learn how to track changes in a Git repository in the next video.

Video 5: Tracking Changes to a Git Repository

Recall that we can obtain a Git repository by creating a new one using the git init command or making a copy of an existing remote repository using the git clone command. Whichever is the case, at the end of the respective process, a Git repository will be created on your computer with a checkout or working copy of all its files. From this point onwards, you can create new files, make changes to existing files or delete unwanted files.

A newly created file is untracked initially and must be explicitly staged for commit using the git add command. This changes the state of the new file to staged.

An existing file is tracked and unmodified initially. Changing its content will cause the file to become modified. The file can then be staged for inclusion in the next commit. There are two important points to keep in mind when working with existing files. First, an existing file that has not been explicitly staged can still be included in the next commit if the commit operation is configured to automatically staged existing files that have been modified and deleted. Second, if further changes are made to a staged file, only the first set of changes will be committed unless the file is staged again.

Once the project reaches a desired state, the git commit command can be used to commit the project. This command offers several useful options. The -a or -all options can be used to automatically staged existing files that have been modified or deleted. The -m option can be used to specify a commit message to remind yourself of the important changes that have been made. When you perform a commit, git will checksums each affected files and sub-directories and store them into the .git directory. A commit object containing the metadata and a pointer to the root project tree will also be created. This enables the snapshot to be re-created when needed in the future. A commit object is typically named using its hashsum.

The first commit in the repository is known as the initial commit object and does not point to anything. Each successive new commit object will always point to the commit object that comes immediately before it. The linking of commit objects in this fashion allows Git to trace the change history of files. Other than routine edits to files, let's find out more about other changes that you can make to files in a Git repository.

To delete a file from your project, you need to remove the file from the Git repository completely. This entails removing the file from both the staging area and working directory such that it is not even shown as an untracked file subsequently. The git rm command is used to perform these two tasks.

To move a file, the git mv command is used to specify the name of the target file and the destination folder path. This moves the file to the destination folder, keeping the same name. However, you should note that Git does not explicitly track file movement. Rather, the existing target file is treated as deleted and the file in the destination folder is treated as a new untracked file. Thus, the git mv command is equivalent to the three commands of the operating system's move command, the git rm command and the git add command. The git add command explicitly stages the new file in the destination folder for the next commit.

In Git, renaming a file is equivalent to moving an existing file to the same folder albeit with a new name. Thus, the same git mv command can be used. In this case, the git mv command will simply specify the name of the target file and the new name.

In many development scenarios, there will be different categories of files that Git should ignore. Ignored files are files that Git should not automatically add or even show as being untracked. Examples of files that should be ignored include automatically generated files such as log files or files produced by build systems such as binary executables. To ignore files, a special file listing patterns that match folder paths and/or filenames to be ignored by Git can be created. This special file is named as .gitignore. The rules for the patterns that can be put in the .gitignore file can be specified using the standard glob patterns, which are simplified regular expressions used in Unix shells. Glob patterns can be applied recursively through the entire working directory.

Let's take a look at an example of a .gitignore file for a simple Java project. This .gitignore file will ignore compiled class files and built Java archive files as well as any log files.

In the last segment of this video, we will briefly discuss how to undo changes that you have made to your Git repository. Changes made to a Git repository can be undo at any stage. However, it is necessary to exercise some caution since it is not always possible to undo some of these undos. In such rare cases, changes may be lost if an undo is performed wrongly.

First, let's find out how to undo a commit. This often happens when you want to redo the last commit to include additional files or to revise the commit message. Thus, undoing a commit essentially replaces the last commit with a new, improved commit. To undo and redo a commit, you can use the git commit command with the --amend option. This command takes the staging area and uses it for the new commit and also update the commit message.

Next, to unstage a previously staged file, the git reset HEAD command can be used in conjunction with the name of the required file.

Finally, you can unmodify a modified file by reverting the file back to what it looked like in the last commit. This is useful when changes to a particular file are not longer required. The git checkout -- command can be used in conjunction with the name of the required file. This command will instruct Git to copy the original file in the last commit to overwrite the current modified file. Take note that the changes in the modified but not committed file will be lost permanently. In other words, this undo cannot be undone.

Now that you have a good understanding of tracking changes to a Git repository, let's move on to find out how to perform branching with Git.

Video 6: Branching in Git

Recall that in Git, as you go about making changes to the repository, new commit objects will be created. Other than the initial commit, each successive new commit object will point to the commit object that comes immediately before it. The linking of commit objects in this fashion creates a linear pattern known as the main line of development.

In version control, branching refers to diverging from the main line of development to perform other tasks such as creating a new use case or fixing an error or bug, without affecting the main line of development. This ensures that the code in the main line of development is always tested to be error-free and ready for deployment to a production environment.

Let's examine a simple example of branching. In this example, there is one new branch created for a new use case development and another new branch created to fix an error. As you will observe, the changes made to each branch remain isolated from the main line of development until they have been tested to be error-free. Many branches can be created as required to cater to project development requirements.

But of course, at some point in time, the branches will have to be merged or integrated back into the main line of development such that new use cases and error fixes can be deployed to the production environment. Returning to the previous example, commit c8 will contain the new use case and error fix ready for production deployment.

The practice of frequent branching and merging generally leads to better quality code and a more dependable software application being delivered to end users in a timely fashion.

Nearly every VCS provides some form of branching support. But in some VCS tools, branching can be a slow and expensive process that requires the creation of a new copy of the source code directory. This process can take a long time for large projects and incur significant storage space.

In comparison, branching in Git is a very lightweight and fast process that does not require a new copy of the source code directory to be created. In Git, a branch is simply a lightweight pointer that points to a commit object in the project's development history. Switching back and forth between branches is also fast as changes to files in the working directory is kept to a minimum. Git thus encourages development workflows that branch and merge frequently, leading to high-quality code and software application.

Within a particular branch, the branch pointer always move forward to point to the latest commit as each new successive commit object is created. In Git, a newly created repository contains a default branch typically named as main. The main branch pointer initially points to commit c0 and gradually move forward to point to c1 and eventually c2.

Since a Git repository can contain multiple branches, how does Git know which branch you are currently working on? Well, Git use a special HEAD pointer to point to the branch that you are currently working on. Thus, the HEAD pointer also moves forward together with the branch pointer as new commit objects are created.

Returning to the earlier example, you can observe that after all the merging, the HEAD pointer will be pointing to the main branch. Moreover, each of the branch pointer points to the latest commit of its respective branch.

In the next video, we will explore how to perform branching and merging in a Git repository.

Video 7: Basic Branching and Merging Operations

Recall from the earlier video that a Git repository contains a default branch typically named as main. As new commits are being made to the project, the branch pointer for main will gradually move forward to point to the latest commit object. This ensures that you always have access to the latest version of the project files.

To create a new branch, the git branch command can be used in conjunction with the name of the new branch. By default, the new branch pointer will point to the same commit object as the main branch, or whichever branch you are currently working on. It is important to note that creating a new branch does not switch to the new branch. You will remain in the same branch that you are currently on. In other words, the HEAD pointer does not point to the new branch and the working directory remains unchanged.

Let's examine an example of creating a new feature1 branch after making three commits in the Git directory. Observe that the main branch pointer and feature1 branch pointer are both pointing to commit c2. More importantly, the HEAD pointer remains pointing to main.

The git checkout command can be used to switch to a different branch from the one that you are currently working on. This will move the HEAD pointer to point to the branch that you are switching to. The files in the working directory will be replaced accordingly.

Return to the earlier example, the git checkout feature1 command will move the HEAD pointer from main to point to feature1. However, the files in the working directory remain unchanged since main and feature1 are both pointing to the same commit c2.

What happens if you perform a new commit after switching branch? Well, take note that the pointer that moves forward to point to this new commit will be the pointer of the branch that you haves switched into. The pointer of the original branch that you have switched out from will remain unchanged.

In our example, the feature1 branch will move forward to point to commit c3 while the main branch remains unchanged and continues to point to commit c2. The HEAD pointer still points to the feature1 branch and hence to commit c3. On a relative basis, we say that main is one commit behind feature1 and feature1 is one commit ahead of main.

Suppose you would now like to switch back to the main branch from the feature1 branch. This will move the HEAD pointer to point to main. The files in the working directory will also need to be replaced with those in commit c2 since feature1 is pointing to c3.

Performing a new commit now will create c4 and moves the main pointer forward to point to c4. Since main and feature1 are now pointing to different commits, on a relative basis, you will observe a divergence in the development history between these two branches. The git log command can be used to view the history of your commits, e.g., where your branch pointers are, and how the development history has diverged.

Suppose you create another new branch error1, switch to the new branch to fix a bug and perform a new commit. The commit tree will now show the error1 branch pointing to c5 and the HEAD pointer pointing to error1.

We are now ready to merge the new use case in feature1 branch and error fix in error1 branch back into the main branch for deployment to production.

To perform merging, you need to switch to the branch that you are merging into, in this case main. Then use the git merge command in conjunction with the name of the branch that you are merging in from. We will examine two different types of merging and their respective impact on the Git repository.

First, let's merge error1 into main using the commands git checkout main and git merge error1. Observe that commit c4 of the main branch that you are merging into is directly in front of commit c5 of the error1 branch that you are merging in from. In other words, commit c4 and c5 are adjacent to each other and lie along the same linear path. In this case, Git will perform a fast-forward merge by advancing the main pointer to point to commit c5. There is no need to replace any file in the working directory.

After merging the error1 branch into the main branch, you can delete the error1 branch with the git branch using the -d option.

Next, let's merge feature1 into main using the same git merge command. Note that you are already on the main branch after merging in the error1 branch earlier. Observe that commit c5 of the main branch that you are merging into does not lie on a linear path relative to commit c3 of the feature1 branch that you are merging in from. In fact, there is a divergence between main and feature1.

In this case, Git will perform a three-way merge by using the two commits at the respective tip of the branch you are merging into and the branch you are merging in from, i.e., c5 and c3, together with their common ancestor, i.e., c2.

After a three-way merge, a merge commit is created to close the divergence between the development history of the two branches. This is depicted as c6 in the commit tree of our example.

In most cases, a three-way merge will proceed smoothly and complete without any problem. But in the case of a merge conflict, you will need to resolve the problem manually. Merge conflict will be discussed in the next video.

To help you identify branches that have not yet been merged into the main branch, you can use the git branch command with the -no-merged option.

Video 8: Merge Conflict Resolution

Recall that there are two types of merge in Git. A fast-forward merge involves two commits that lie on the same linear path. Thus, a fast-forward merge should not involve any conflict. In contrast, a three-way merge involves two commits on a divergence and their common ancestor. In some cases, a three-way merge might involve different changes made at the same part of the same file across the two branches that you are merging together. This leads to a merge conflict.

Whenever a merge conflict occurs, Git will not be able to merge the changes cleanly. More specifically, Git will not be able to choose the overlapping changes on your behalf. Rather, it will pause the process for you to resolve the conflict manually. This process is known as merge conflict resolution.

You can use the git status command to view files that contain merge conflicts. Git adds standard conflict-resolution markers to files that have conflicts. You need to open each file manually and resolve those conflicts by removing the conflict-resolution markers.

There are two general approaches to resolve a merge conflict. First, you can choose the changes in either one side or the other. Second, you can merge the content from both sides yourself.

Let's look at a simple example of a merge conflict involving a Python source file. The change above ======= is from the HEAD, i.e., the branch you are currently on and the branch you are merging into. The change below ======= is from the branch that you are merging in from, in this case feature1.

To keep the change in the HEAD branch, you should remove the lines in red.

To keep the change in the feature1 branch, you should remove the lines in red.

To keep both changes, you should only remove the conflict-resolution markers in red and then manually reconcile the changes.

If you find that manually resolving merge conflicts is tedious, you can always use a graphical merge to assist you.

After resolving the conflicts in each file, you need to stage the file with the git add command to mark the conflicts as resolved. Once all conflict files have been resolved, you can then proceed to finalise the merge commit using the git commit command.

To minimise the chance of having merge conflict, some good practices include modularising your code into separate source files and assigning the responsibility of changing each file to a specific team member.

Now that you have an excellent understanding of working with a local Git repository, let's move on to find out how to work with a remote repository that is running on a server computer.

Video 9: Working with a Remote Repository in GitHub

One of the greatest strengths of Git is its fully distributed nature that enables a large team of developers to collaborate remotely. To do so, you need to add at least one remote repository to your Git repository.

If you are creating a new Git repository using the git init command, you will need to explicitly add a new remote repository using the git remote add command in conjunction with the short name and the URL of the remote repository. For example, if you are using GitHub, the URL will resemble this format https://github.com/<USER_NAME>/<REPO_NAME>.git. By convention, the first remote repository adopts the short name of origin.

If you have cloned your local repository from a remote server, the git clone command will implicitly add the origin remote repository for you. You can use the git remote command to show all configured remote servers.

The typical workflow for collaborating in a team and sharing changes via a remote repository entails two simple steps. First, you will upload your changes to the remote repository. Thereafter, your teammates will be able to download the changes that you have made. For a new repository, you need to upload the entire repository to the origin remote server for the first time before your teammates can clone the repository. Thereafter, the workflow for sharing changes can continue in a cyclic fashion. Of course, your teammates will be able to share their changes with you and other teammates too.

To upload or push changes, you use the git push command. By default, the git push command will push the current branch to the origin remote server. You can use the --all option to push all branches. Suppose there is only one main branch in the local repository with the initial commit c0, this branch will be pushed to the remote repository. After your teammates have cloned from the remote repository, they will be able to see the main branch pointing to commit c0.

Subsequently, suppose you have made another commit, i.e., c1, and pushed c1 to origin. Your teammates have two approaches of downloading the changes. The first approach is to pull the changes into their repositories. Pulling will download and integrate any changes directly into the local repository. The second approach is to fetch the changes into their repositories. Fetching will download any changes but does not integrate them into the local repository. Rather, your teammates will be able to view your changes and decide what to do. In this case, your teammates can merge the changes into their local branch or checkout the newly fetched branch. Fetching is more useful once your project has many branches and you and your teammate might prefer fetching new branches instead of pulling them.

Let's look at the difference between pulling and fetching using only the main branch for illustration. Observe that in pulling, commit c1 is directly merged into your teammates' local repository.

But in fetching, commit c1 is marked as the remote branch origin/main.

The process for your teammates to share their changes with you or other teammates is exactly the same. In other words, your teammates will push their changes to the remote repository and everyone else, including you, will pull or fetch the changes as required.

Video 10: Module Summary

We have come to the end of this introductory module on Git. You now understand how to effectively work with a Git repository to manage changes made to your project's source code. In particular, you know the essential concepts of distributed VCS and Git, how to obtain a Git repository, track changes, perform branching and merging, and collaborate with other teammates using a remote repository. Distributed VCS and Git can be readily applied across any programming languages and development stacks to improve the productivity and quality of your project. Keep practising these techniques to perfect them!