

## MODULE 3

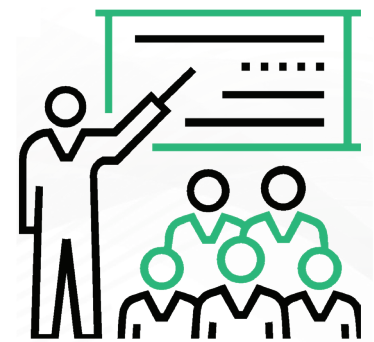
# Version Control System vs Distributed Version Control System

You will learn about the 'Version Control System vs Distributed Version Control System' in this module.

### Module Objectives

At the end of this module, you will be able to:

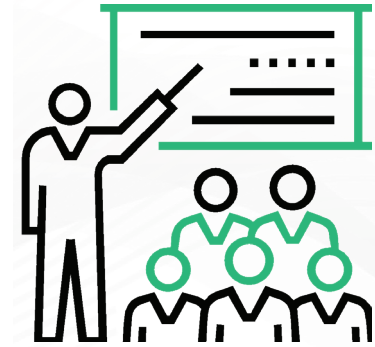
- Explain how local version control works
- Discuss the basics of Centralized Version Control Systems (CVCS)
- Describe the basics of Distributed Version Control Systems (DVCS)
- Enlist the advantages of DVCS, such as:
  - Private Workspace
  - Easier merging
  - Easy to scale horizontally
- List the disadvantages of DVCS
- Explain how CVCS and DVCS compare with each other
- Describe the working of the multiple repositories model
- Learn how to reset the local environment
- Identify how revert operation is used to undo the changes



## Module Topics

Let us take a quick look at the topics that we will cover in this module:

1. Local repository
2. Advantages of distributed version control system
3. The multiple repositories model
4. Completely resetting local environment
5. Revert - canceling out changes

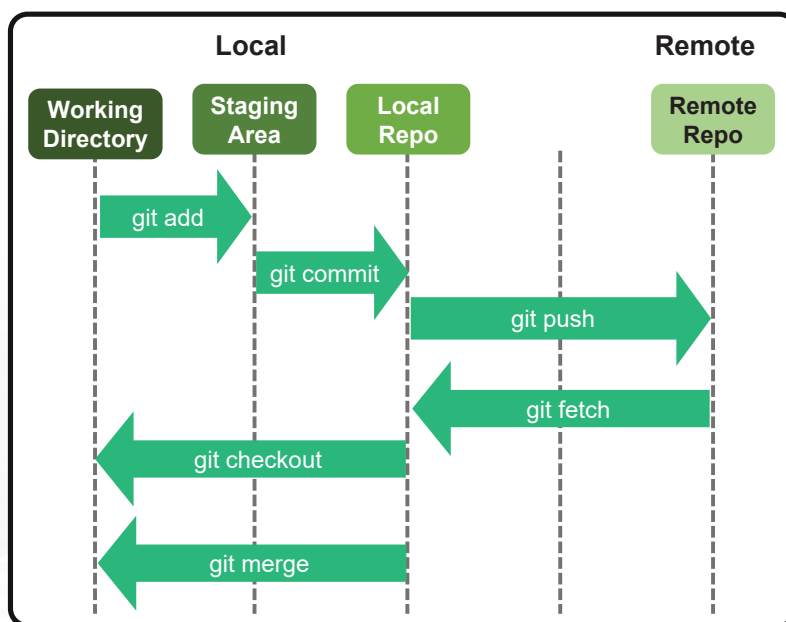


### 1.1 Local Repository

Following are the key details of the local repository:

- Local version control keeps track of the files within the local system.
- Local version control systems save a series of patches, but collaboration or branching is almost impossible with local repositories.
- A local repository, from a DVCS viewpoint is a collection of files which originate from a certain version of the repository.
- The collection of files is called the working tree or the checkout.

The picture explains how the changes are made to the local repository and pushed to the remote repository.



A local version control, as the name suggests, is a local system that keeps track of the files within the local system. We've seen about the local version control system in module 1.

We'll now look at local repositories from a DVCS viewpoint.

There are two kinds of repositories in a DVCS.

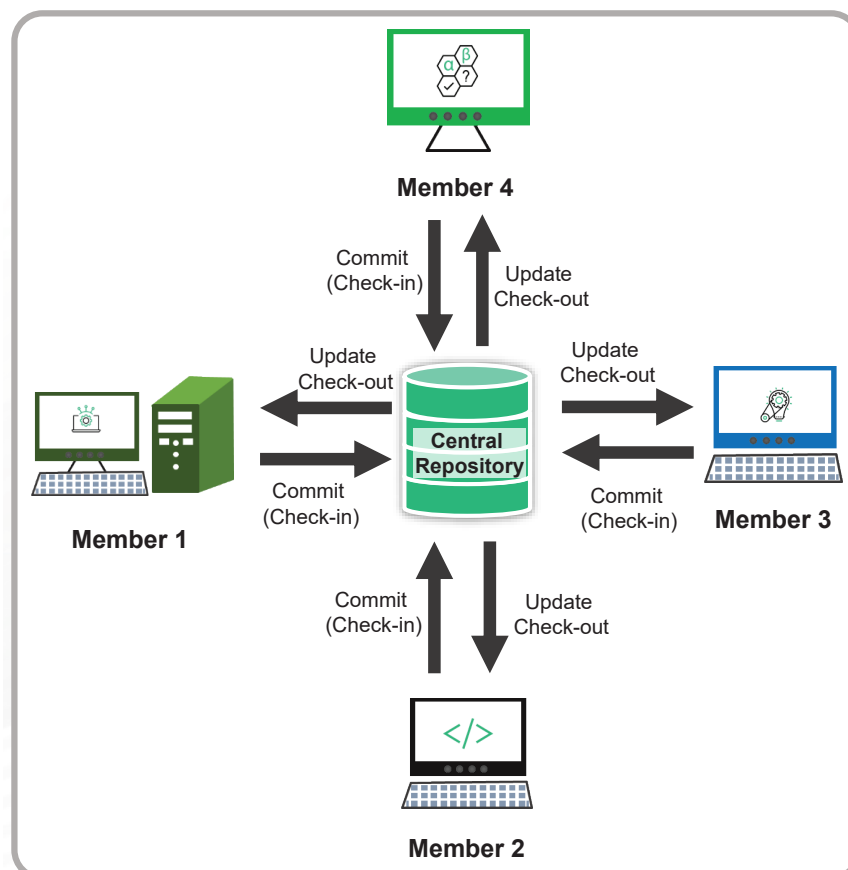
- Local repository—a copy of a central repository, that is available on the local computer.
- Remote repository—the repository available in the central server

The primary advantage of a DVCS like Git is that the local repository is the mirror of the central remote repository. It is the local repository that the developers make the necessary changes. The local repository is called the working copy or the checkout. The picture above shows the typical workflow in Git, a DVCS. The workflow consists of the steps or operations that we learnt in module 2. From the working directory, changes are added to the staging area and committed to the local repository. From the local repository, changes are then pushed to the remote repository.

## 1.2 Centralized Version Control System (CVCS)

Following are the key details of the Centralized Version Control System (CVCS):

- Centralized version control systems keep a single copy of the project on a central server and the developers commit their changes to this central copy of the project.
- Updates to the code is recorded in the central system; when developers pull the code, changes are automatically applied to the files that were changed and developers pull this updated version.
- CVCSs eliminate the need to keep multiple copies of their files on their hard drives.
- The version control system will communicate with the central system and retrieve any version on the go.
- Some examples of CVCS are CVS, Subversion (SVN), Perforce, etc.



We saw a brief about the types of version control systems in module 1. Let's see a quick recap of centralized version control systems.


From the figure above, you can understand how a CVCS works. Centralized Version Control Systems (CVCS), as the name suggests, keep a single copy of the project in a central server. This means, the history of changes is kept on a single central server, from which everyone requests the latest version of the source code and the changes are also pushed to the same copy in the central server.

The typical workflow in a CVCS is as follows:

- Pull down any changes other people have made from the central server.
- Make your changes, and make sure they work properly.
- Commit your changes to the central server, so other programmers can see them.

## What did You Grasp?

---



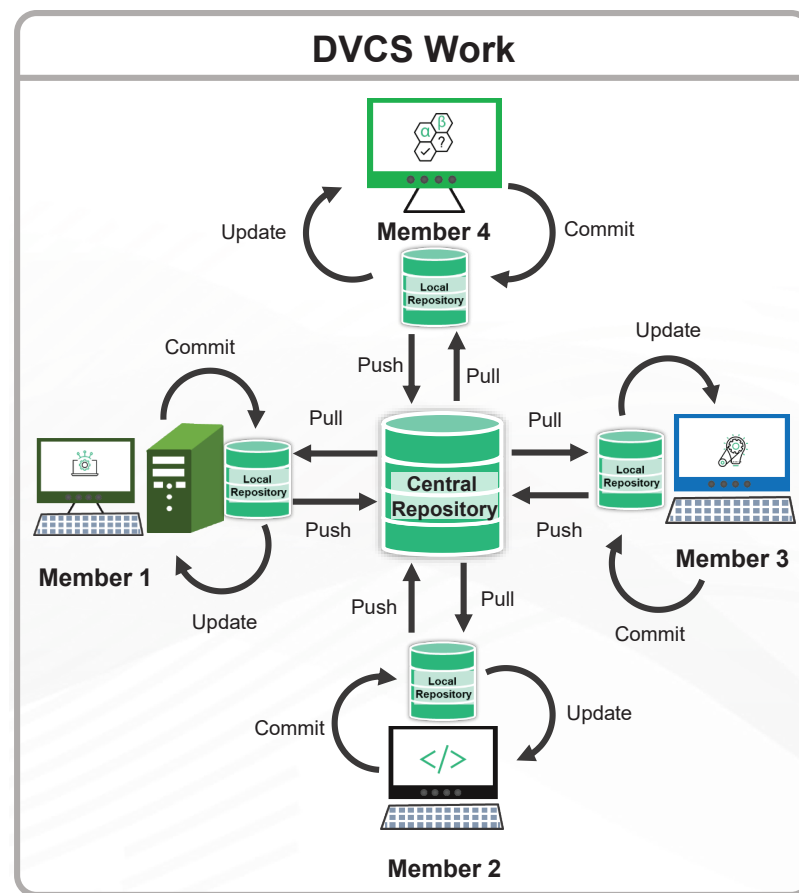
1. Which of the following commands is used to move the working copy to the staging area?  
A) `git add`  
B) `git commit`  
C) `git push`  
D) `git fetch`
2. State True or False  
There are multiple central repositories in a CVCS.  
A) True  
B) False

## 2.1 Distributed Version Control System (DVCS)

---

Following are the key details of the Distributed version control system (DVCS):

- A centralized version control system (CVCS) has a single repository to be accessed by the users.
- Distributed version control system (DVCS) replicates the repository onto each user's machine, i.e., each user has a self-contained first-class repository.
- There is no need for a privileged master repository, though teams have it by convention, for doing continuous integration.



Like CVCS, we have seen an overview of DVCS in module 1 and some examples of it in module 2. We'll now see a quick recap of DVCS and its workflow.

DVCS has emerged from the thought process that every developer has their own local repository, apart from the central repository. This way, they don't check out a snapshot of the source code, but they fully mirror the central repository. This means, DVCSs do not rely completely on the central server for all the versions of the source code, every developer has a clone of the source file that is available in the central repository, and the complete history of the project is available on their own hard drive. This clone has all the metadata of the original source file.

A developer, by means of 'pulling', gets the new changes from the central repository to the local repository. Developer's changes are then applied to the code in the local repository and then 'pushed' back to the central repository.

## What did You Grasp?



1. State True or False

The working copy contains only the partial code of the central repository.

- A) True
- B) False

## 2.2 Advantages of Distributed Version Control System

Following are the advantages of DVCS:

- Other than push and pull, all actions can be performed very quickly, since it is the hard drive, and not the remote server that is accessed every time.
- Changesets can be committed to the local repository first and then a group of these changesets can be pushed to the central repository in a single shot.
- Only the pushing and pulling activities need internet connectivity; everything else can be managed locally.
- Every developer has a complete copy of the entire repository and the impact any change can be checked locally before the code is pushed to the central repository.
- DVCS is built to handle changes efficiently, since every change has a Global Unique Identifier (GUID) that makes it easy to track.
- Tasks like branching and merging can be done with ease, since every developer has their own branch and every shared change is like reverse integration
- DVCS is very easy to manage compared to CVCS.

The advantages of DVCS are listed in the slide above. Your facilitator will shed more light on the points given above. The upcoming slides will explain the very important advantages of Distributed Version Control Systems.

## 2.3 Private Workspace

---

Following are the key details of a private workspace:

- DVCS offers each developer a private copy of the complete repository.
- By giving developers a private copy of the entire repository, the DVCS opens up much more flexibility for the kind of things they can do in their private workspace.
- With DVCS, a developer can do frequent commits as often as they want to.
- This option proves advantageous for a solitary developer, who never has to worry about coordinating with others and managing the maintenance overhead.
- With private workspace, one can commit incomplete functionality regularly to the local repository to check point without affecting other users.

Almost all version control tools offer a private workspace. In CVCS, developers get a working copy of the files, which acts as the private space. With DVCS developers get the complete repository as a private copy, which is the most important point to note about DVCS.

This private workspace provides an added advantage in the sense that the developers never have to think about coordinating with others during the development. When there are multiple developers in a team, the situation becomes complex. Normally, version control systems take this responsibility of managing the complexities. With the private space in DVCS, a developer gets a feel that he/she is working alone on the project, for at least a while. Developers have the flexibility to do anything within their private workspace, without affecting the workflow of other developers.

## 2.4 Easier Merging

---

Following are the key details of merging:

- Branching is generally an easy thing to do, but merging is not.
- People using a CVCS tend to avoid branching because most of those centralized tools aren't very good at merging.
- Merging in a DVCS is less error-prone, since they keep the developer's changes distinct from the intended merge in order to get the changes committed.
- DVCS deals with whole-tree branches, not directory branches. The path names in the tree are independent of the branch. This improves interoperability with other tools.

Branching is easy as compared to merging. Branching is like two people going off in their own directions and not collaborating.

People using a CVCS tend to avoid branching because most of those centralized tools aren't very good at merging. When they switch to a DVCS,

they tend to bring that attitude with them, even though it's not really necessary anymore. Decentralized tools are much better at merging.

The reasons are as follows:

- They're built on a Directed Acyclic Graph (DAGs). Merge algorithms need good information about history and common ancestors. A DAG is a better way to represent that kind of information than the techniques used by most centralized tools.
- They keep the developer's intended changes distinct from the merge she had to do in order to get those changes committed. This approach is less error-prone at commit time, since the developer's changes are already cleanly tucked away in an immutable changeset. The only thing that needs to be done is the merge itself, so it gets all the attention it needs. Later, when tracking down a problem, it is easy to figure out if the problem happened during the intended changes or the merge, since those two things are distinct in the history.
- They deal with whole-tree branches, not directory branches. The path names in the tree are independent of the branch. This improves interoperability with other tooling.

## 2.5 Easy to Scale Horizontally

---

### Easy to Scale Horizontally

- A DVCS has much more modest hardware requirements for a central server.
- Users don't interact with the server unless they need to push or pull.
- All the heavy lifting happens on the client side so the server hardware can be very simple indeed.
- With a DVCS, it is also possible to scale the central server by turning it into a server farm.
- Instead of one large server machine, you can add capacity by adding more small server machines, using scripts to keep them all in sync with each other.

With a CVCS, the server holding the central repository needs to be powerful enough to serve the needs of the entire team. For a team of 10 people, this is not an issue. For larger teams, the hardware limitations of the server can be a performance bottleneck. Some systems expect the server to do a lot of work. It can be challenging and expensive to set up a server to support thousands of users.



A DVCS has much more modest hardware requirements for a central server. Users don't interact with the server unless they need to push or pull. All the heavy lifting happens on the client side so the server hardware can be very simple indeed. With a DVCS, it is also possible to scale the central server by turning it into a server farm. Instead of one large server machine, you can add capacity by adding more small server machines, using scripts to keep them all in sync with each other.

## 2.6 Disadvantages of Distributed Version Control System

Following are the disadvantages of DVCS:

### Disadvantages

- With many projects, large binary files that are difficult to compress, will occupy more space.
- Projects with a long history, i.e., a large number of changesets may take a lot of time and occupy more disk space.
- With DVCS, a backup is still needed, since the latest updated version may not be available to all the developers.
- Though DVCS doesn't prevent having a central server, not having a central server might cause confusions in identifying the right recent version.
- Though every repo has its own revision numbers, releases have to be tagged with appropriate names to avoid confusions.

There is almost no disadvantage with a DVCS, and that is the reason for the popularity of DVCS like Git. The above slide lists a very few disadvantages of DVCS, which may not happen in all circumstances.

## What did You Grasp?



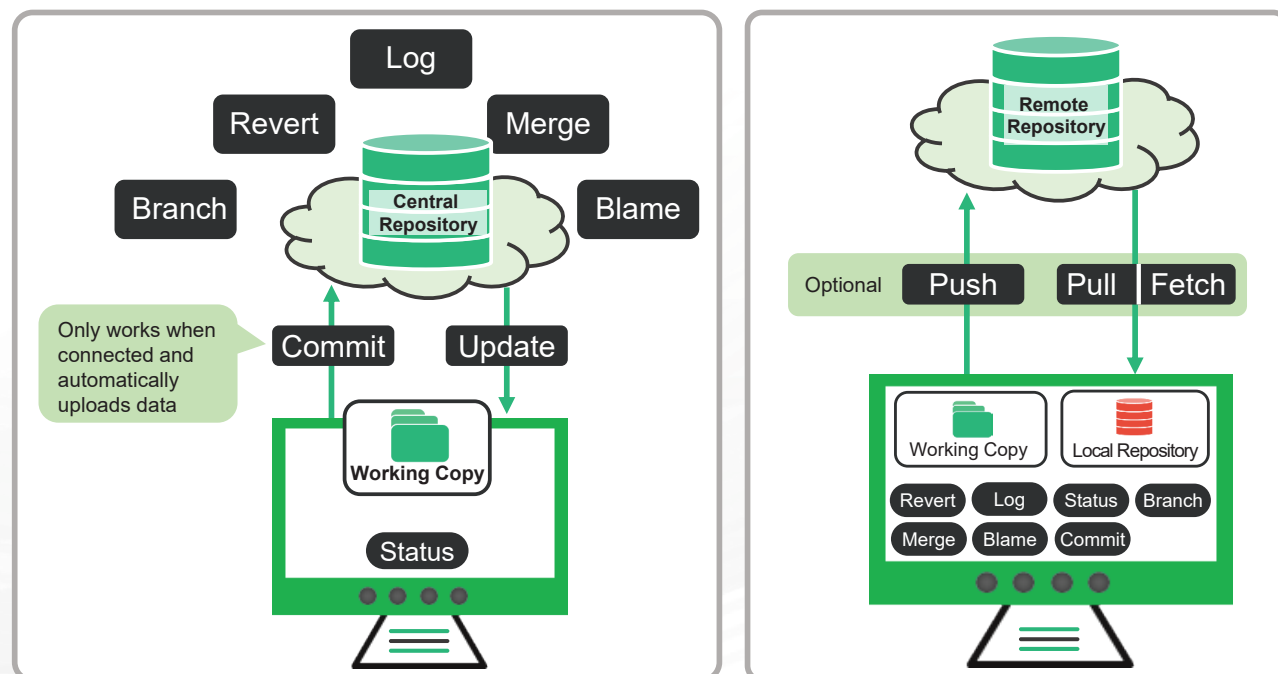
1. Which of the following statements is true?
  - A) In a DVCS, the central server is accessed every time an update is done to the code
  - B) Push and pull activities in a DVCS require internet connectivity
  - C) Merging in a DVCS is hard compared to CVCS
  - D) The idea of private workspace is unique to DVCS
2. State True or False.
 

Having a central server is still possible with DVCS.

  - A) True
  - B) False

### 3.1 Centralized vs Distributed Version Control Systems

The following picture highlights the difference between centralized and distributed version control systems.



source: Scriptcrunch

So far, the important concepts of CVCS and DVCS were explained along with the advantages and shortcomings of DVCS. The figure above is intended to give a comparative picture of the two version control systems, on the basis of how they work. The upcoming section will list down the comparing and contrasting features of the two systems.


### 3.2 Comparison of CVCS and DVCS

CVCS	DVCS
CVCS focuses on synchronizing, tracking, and backing up files.	DVCS focuses on sharing changes; every change has a guid or unique id.
CVCS works based on a client-server relationship, with the source repository located on one single server, providing access to developers across the globe.	Every developer has one local copy of the source code repository, in addition to the central source code repository.
Recording/downloading and applying a change are separate steps in a centralized system, they happen together.	Distributed systems have no forced structure. You can create "centrally administered" locations or keep everyone as peers.
CVCS relies on internet connectivity for access to the server.	DVCS enables working offline. Apart from push and pull actions, everything is done locally.

CVCS	DVCS
CVCS requires contacting the server for every command and is relatively slow.	DVCS takes time only during the initial checkout as it mirrors the central repository on the local environment. Subsequent updates and commits are done in the local repository and so the process is faster.
CVCS allows checking out of a few lines of code, if the work has to be done in only a few modules.	DVCS doesn't allow partial checkout and projects with a long history will take a long time and occupy more disk space.
CVCS is comparatively easy to understand for beginners.	DVCS has some complex processes for beginners.

Go through the table above and understand the differences between CVCS and DVCS.

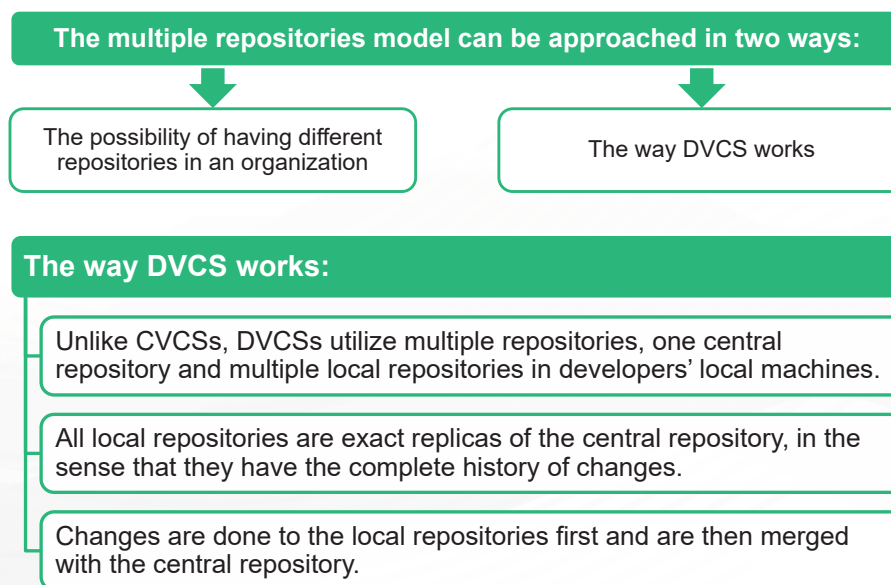
## What did You Grasp?



1. Which of the following statements is incorrect?

- A) DVCS works on a client-server relationship
- B) CVCS requires the Internet connection for access to the central server
- C) With CVCS it is possible to checkout portions of code
- D) Initial checkout is a time consuming process in DVCS

## 4.1 Multiple Repositories Model



With DVCS, the multiple repositories model can be approached in two ways.

- By understanding how a DVCS works
- The flexibility offered by DVCS of having different repositories for different services within the same organization

Let's look at the first approach.

We are familiar that in a DVCS there are multiple repositories, one central repository and multiple local repositories. Although it is widely claimed that with DVCS, there is no single authoritative central repository, there is always an option to label any repository, that which has all the revision history updated, as the central repository.

From the central repository, any number of local repositories can be cloned and kept in the developers' local computers. These local repositories will be the exact replicas of the central repository, with the entire revision history. Developers make changes to this local repository and once the changes are done, there is a room to test if the changes impact rest of the code. Once tested, changes from the local repository are merged with the central repository.

One of the major advantages of this is there is no single point of failure at any point in time. Even if the central repository is broken, developers can continue to commit changes to the local repository and push them to the central repository when it is fixed. Since the local repository is the mirror of the central repository, any of them can be made as the central repository, even if any irreparable damage happens to the central repo.

Let's now look at how DVCS like Git allows having multiple repositories within the same organization.

## 4.2 Multiple Repositories for Different Services

### Multiple Repositories for Different Services

- In mono repositories, the source code is placed in a single repository in an organization and developers have access to all code in a single shot.
- The Multiple repository concept refers to organizing the projects into their own separate repositories.
- DVCS like Git allows us to have multiple repositories, by which giving access to subsets of repositories on a need basis, becomes possible.
- Especially, products with multiple services can be handled efficiently by having separate repositories for each of them.
- While setting up continuous deployment projects, it is easier to let the repositories have their own processes for getting deployed.


The second approach to this multiple repository model is about organizing the project into multiple repositories. The multiple repository concept refer to organizing the project into separate repositories. With DVCS like Git, there is always an option to have different repositories for different services in the same project.

Especially, with developers work from different locations across the globe, having multiple repos gives us the freedom to share only those repositories that are required for a developer at that point in time.

Setting up continuous deployment projects become easier to handle, since each of the repositories will have their own processes for getting deployed.

Having multiple repositories might have its own disadvantages, especially if the number of repositories are more. Managing the repositories will become a tedious task and leveraging the full advantage of DVCS might become a question. With multiple repositories, it might be difficult for developers to git clone a whole lot of different repositories at a time.

## What did You Grasp?



1. State True or False

With DVCS like Git, there is no room for having different repositories for different services.

A) True  
B) False

## 5.1 Resetting the Local Environment

- Reset is one of the operations used for undoing the changes.
- We can understand the reset operation using Git. The Git reset operation has three basic forms of invocation that correspond to three commands, such as follows:



- These arguments correspond to three state management systems of Git, namely the Commit Tree (HEAD), the Staging Index and the Working Directory.
- The reset command moves both the HEAD and the branches to a specific commit.

Reset is one of the operations performed to undo the changes done to the local repository. We can understand this with the help of reset option in git.

There are three basic forms or modes of invoking the reset operation that is done using three arguments:

- `--soft` - Does not affect the index file or the working tree at all. This command resets the head to `<commit>`, in a way that the other modes perform). This leaves all your changed files “Changes to be committed”, as `git status` would put it.
- `--mixed` - This mode resets the index, but not the working tree (the files that undergo change are preserved but not marked for commit). It also reports what has not been updated. This is the default action. If `-N` is specified, removed paths are marked as intent-to-add.
- `--hard` - This mode resets both the index and working tree. Changes made to the tracked files in the working tree since `<commit>` are discarded if this mode is selected. This option also wipes out the uncommitted changes. This option has to be used with caution as the lost data cannot be recovered.

These arguments again correspond to three state management systems in Git, namely the Commit Tree (HEAD), the Staging Index and Working Directory.

The reset operation behaves in a way similar to the checkout operation. The command `git reset`, moves both the HEAD and branch refs to a specific commit. The command `git reset` will modify the state of the three trees as specified above. The command line arguments `--soft`, `--mixed`, and `--hard` direct how to modify the Staging Index, and Working Directory trees.

There is a risk of losing work with the reset operation, and revert has been always a safe option to undo any commit. Git reset will not delete a commit, but commits will become orphaned, i.e., there will not be any direct path from a ref to access them.

## 5.2 Revert - Canceling out the Changes

### Revert - Canceling out the Changes

- Revert is also one of the options for undoing the changes, but works in a different way compared to the traditional undo operation.
- The revert option doesn't remove the commit from the project history, but it checks how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.
- The revert operation prevents Git from losing the history, which is very important to maintain the integrity of the version history.
- Revert option can be used when an inverse of a commit from the project has to be applied.


Like reset, revert is also one of the operations for undoing the changes. Unless there is a dire need, it's always safe to use the revert option instead of reset.

Revert doesn't remove the entire commit from the project's history. The `git revert` command is used for undoing changes to a commit history. Other 'undo' commands like, `git checkout` and `git reset`,

move the HEAD and branch ref pointers to a specific commit. Git revert also takes a particular commit, however, git revert does not move ref pointers to this commit. A revert operation will take the specified commit, inverse the changes from that commit, and create a new 'revert commit'. The ref pointers are then updated to point at the new revert commit making it the tip of the branch.

Git revert is a safer alternative to git reset in regards to losing work.

## What did You Grasp?



1. Which of the following commands is used to move the HEAD and the branch refs to a specific commit?
  - A) `git reset`
  - B) `git move`
  - C) `git revert`
  - D) `git clean`

## In a nutshell, we learnt:



1. How local version control works
2. A brief on centralized version control systems (CVCS)
3. A brief on distributed version control systems (DVCS)
4. Advantages of DVCS
  - Private Workspace
  - Easier merging
  - Easy to scale horizontally
5. Disadvantages of DVCS
6. Comparison of CVCS and DVCS
7. Multiple repositories model
8. Resetting the local environment
9. Reverting - undoing the changes

