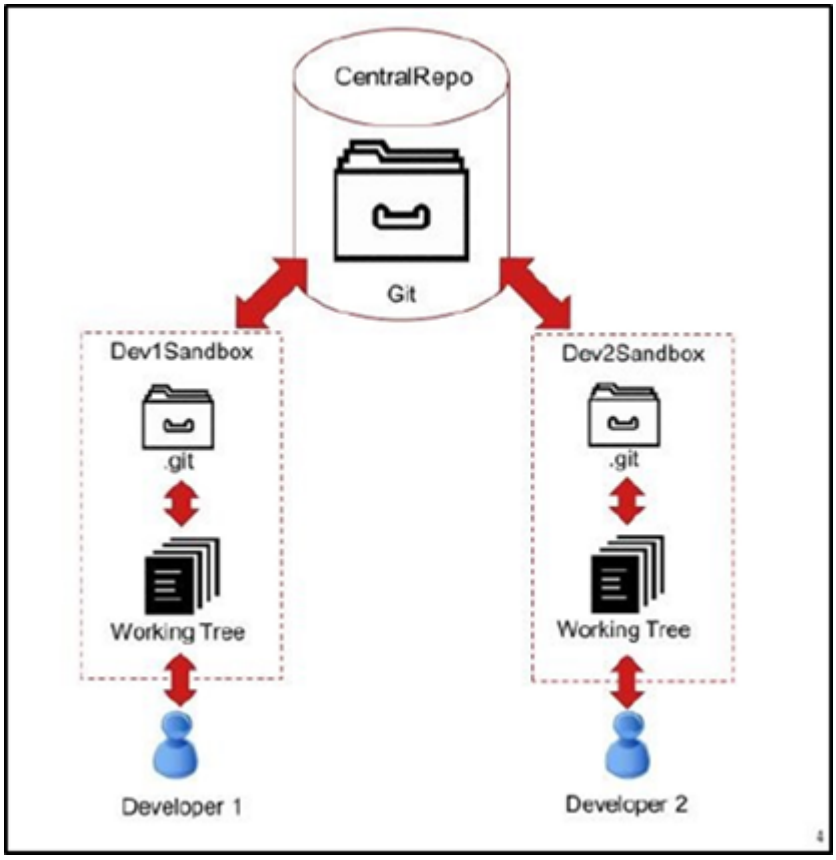
# Appendix IV: Using a Shared Repository and Merging Conflicts

## Use Shared Repository

Multiple developers use a Central (also called Remote) Repository, allowing each authorized developer to push and fetch changes from a single source. Although each developer maintains a local copy of the Repository on their machine, the Remote Repository collects the changes from everyone on the team. Each developer is responsible for updating or fetching changes from the Remote Repository regularly.




## Connect to Shared Repository

Adding a Remote Reference allows developers to establish a connection between their Local Repository and a Remote Repository. It enables developers to push their changes to the Remote Repository, fetch updates made by others, and synchronize their work with the rest of the team.

## Push Changes to Shared Repository

Pushing changes to a Remote Repository allows the developer to send the local **Commits** and updates to the Remote Repository, making them accessible to others working on the project.

## Fetch Changes from Shared Repository

Fetching changes from the Shared Repository ensures that the developers have the most recent code updates, allowing them to incorporate the changes into the Local **Branch** and maintain a synchronized codebase.

## Managing File Conflicts

A merge conflict occurs when two or more developers make conflicting changes to the same part of a file. For example, if Developer 1 modifies a function while Developer 2 modifies the same function, the Version Control System may be unable to determine which changes should take precedence automatically.

## Resolving Merged Conflicts

When working with Git, conflicts can be encountered when merging two **Branches**. This occurs when Git cannot automatically merge changes made to the same lines of code in both **Branches**. The following steps outline the process to resolve merge conflicts:

1. Open the file(s) that have conflicts.
2. Look for the conflict markers in the file(s), which looks like below screenshot:

```
<<<<<<< HEAD
code from the current branch
=======
code from the branch being merged
>>>>>>> branch-name
```

3. Edit the code to reflect the changes to be kept.

4. Remove the conflict markers from the file(s).
5. Save the changes to the file(s).
6. Add the modified file(s) to the staging area.
7. Commit the changes.

8. Push the changes to the Remote Repository.
   If using a Merge tool like VS Code or SourceTree, it should have a graphical interface to help resolve conflicts more easily. Launch the tool and follow its instructions to resolve conflicts.

   It's important to note that resolving merge conflicts can be a complex and time-consuming process, especially if there are many conflicts to resolve. It's always a good idea to carefully review and test changes after resolving conflicts to ensure they work as intended.

## Sharing Changes with the Remote Repository

Once the changes are made to the Local Repository and any Merge conflicts resolved, developers need to share those changes with the Remote Repository. Here are a few Developer responsibilities:

- **Commit Changes:** It is essential to commit changes locally. Make frequent well-documented **Commits**.
- **Fetch the Latest Changes:** Fetching the latest changes from the Remote Repository is good practice. This ensures that users have the most up-to-date version of the codebase and reduces the chances of conflicts.
- **Rebase:** Use Rebase to update Local Repository with Remote Repository changes.
- **Push Changes:** Push Local Repository changes to the Remote Repository frequently.
- **Verify Changes:** After pushing the changes, it is essential to verify that they have been successfully shared with the Remote Repository.

## Using Stash

Stash allows developers to store the modifications, including both staged and un-staged changes, in a safe place to switch to a clean working directory without losing their work. It provides temporary storage for their changes, enabling them to seamlessly move between tasks or **Branches**. Stashing is particularly useful when developers are not yet ready to commit their changes or when they want to work on a different task without the interference of their current modifications.

The following points will explain the process of using stash effectively:

- **Stashing Changes**: A common use case for stashing changes is when users make changes to the working directory that are not yet committed but need to fetch changes from another developer in the Remote Repository.
- **Viewing the Stash**: Users can view the contents of the stash at any time by using the git stash list command. Users can also view the stash graphically by reviewing the Revision Graph diagram.
- **Applying the Stash Changes**: When ready, users can restore the stash into the current staged snapshot and then commit the changes including the stashed changes.