

TECHNOLOGY



Coding Bootcamp

TECHNOLOGY



Git

Branching



Learning Objectives

By the end of this lesson, you will be able to:

- 🕒 Design a branching strategy for a software development project, detailing the use of feature, develop, and main branches
- 🕒 Apply basic Git branching commands to create, manage, rename, and delete branches in a repository
- 🕒 Execute branch merging in Git to integrate changes from different branches into a single branch
- 🕒 Evaluate different Git branching strategies for their effectiveness in managing code changes and supporting collaboration



Learning Objectives

By the end of this lesson, you will be able to:

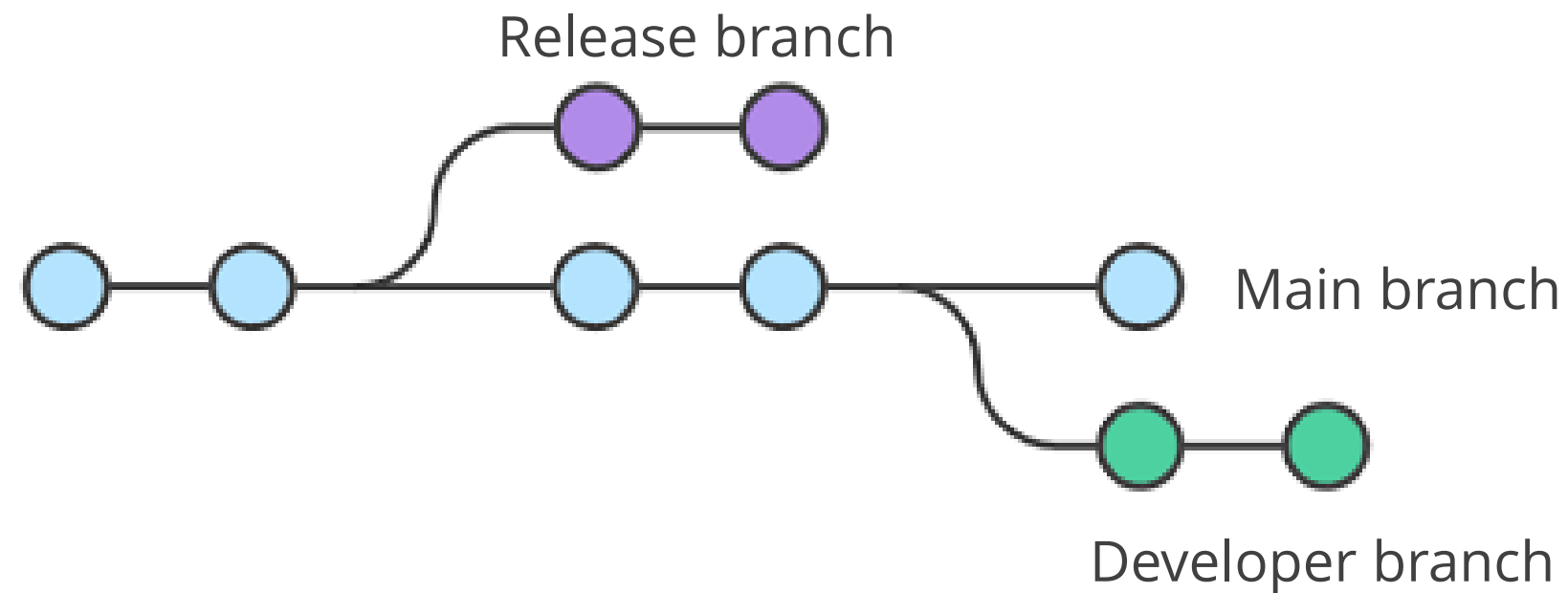
- 🕒 Evaluate different Git branching strategies for their effectiveness in managing code changes and supporting collaboration
- 🕒 Apply techniques to resolve merge conflicts in Git for maintaining a clean codebase
- 🕒 Execute Git rebase commands for integrating changes from one branch to another while maintaining a linear commit history



Branching in Git

Branching in Git

A Git branch is a lightweight movable pointer to the commits. It allows developers to create isolated copies of the codebase at specific points of the development line within the project.



It allows developers to work on a feature or bug fix without affecting the main codebase.

Branching in Git

The primary purpose of branching in Git is to create isolated environments for development within the project. Additionally, here are some key reasons to use branches:



Experimentation and feature development: It provides developers with separate space to test and experiment with new software features.



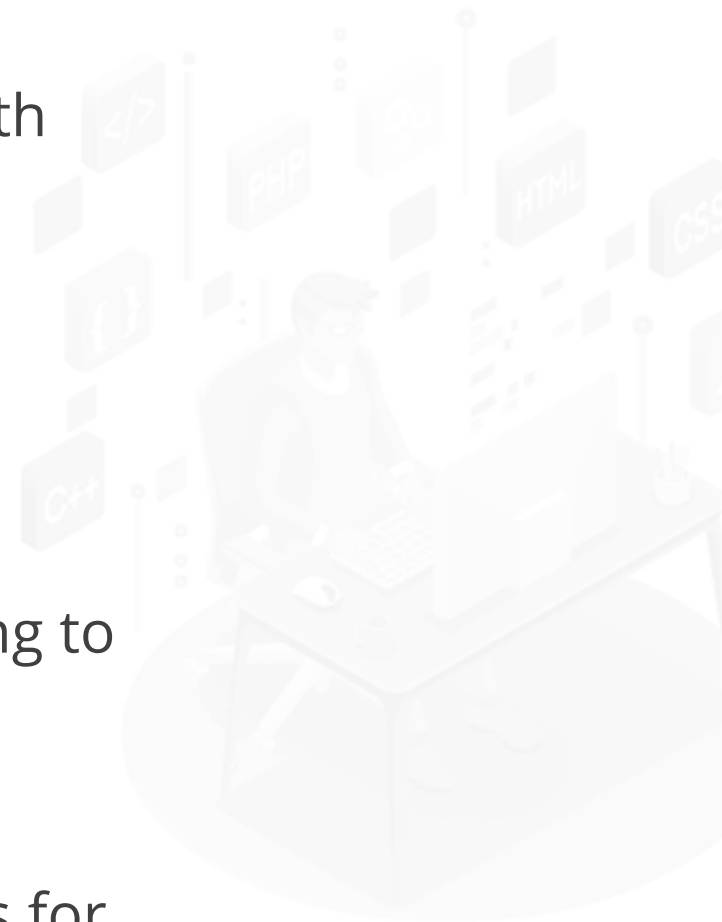
Collaboration and parallel development: It allows developers to work simultaneously without conflicts or overwriting.



Version control and code history: It allows tracking of changes, reverting to previous versions, and understanding the evolution of the codebase.

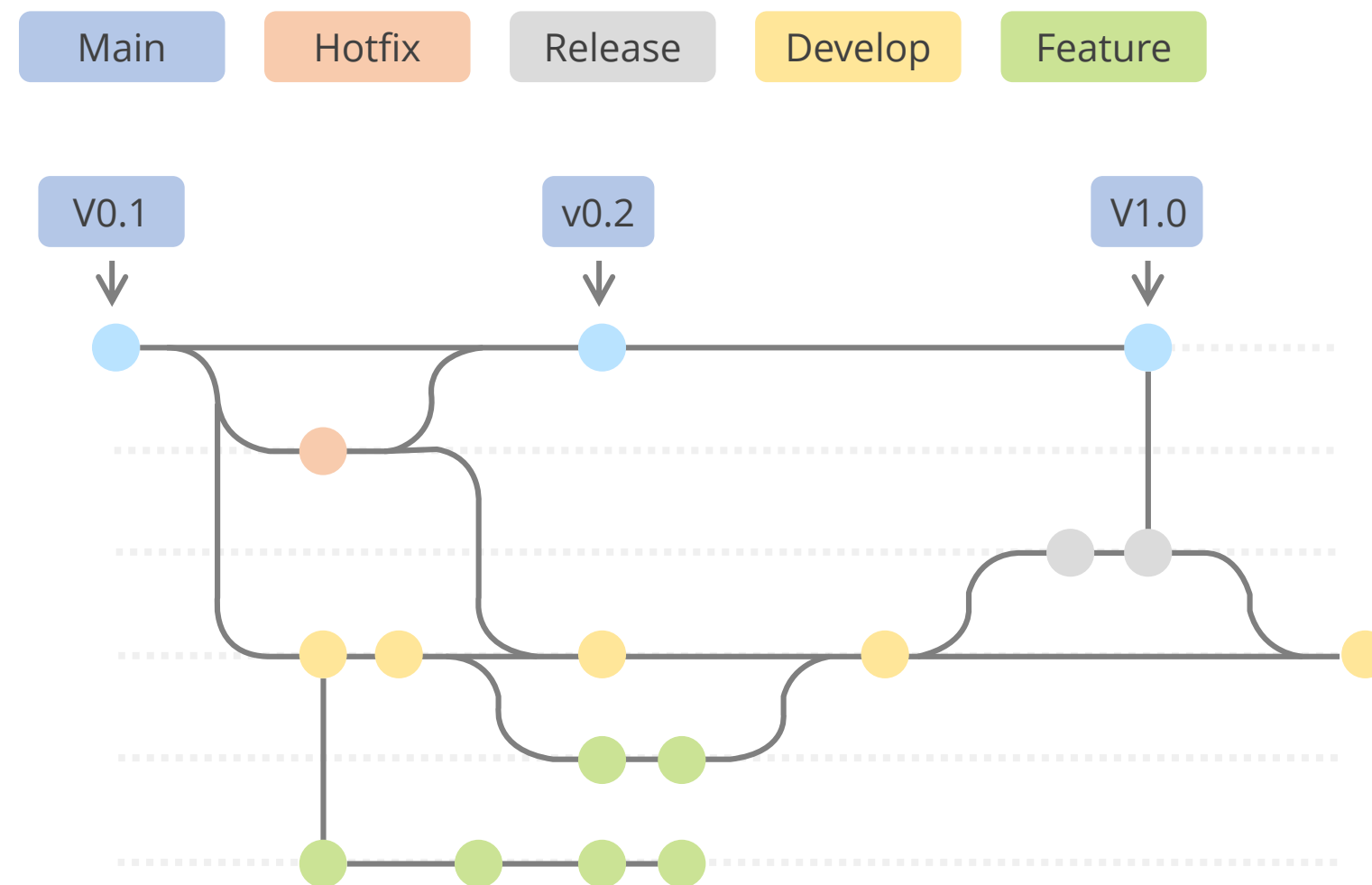


Deployment strategies: It can be used to create separate environments for deployment purposes.



Git Branching Model

A Git branching model or Git flow is a set of guidelines or conventions that define how to create, manage, and use branches in a Git workflow.



It primarily has **main**, **develop**, **feature**, **hotfix**, and **release** branches.



Git Branching Model

The following are the main branches used in a Git branching model:

Branch	Action	Purpose
Main	Mainline for stable releases	The deployment of the final code to the pre-production and production environment
Develop branch	Feature development	Developers create feature branches and submit pull requests to merge the features
Feature branch	Feature development	Developers work on features in these branches

Git Branching Model

The following are the main branches used in a Git branching model:

Branch	Action	Purpose
Release branch	Release preparation	The release branch from the develop branch is used for staging deployment and release preparation.
Bugfix branch (if needed)	Bug fixing	If bugs are found in staging, a bugfix branch is created from the release branch to fix the bugs.
Hotfix branch (if needed)	Hotfix bugs	The issues found in pre-production or production are fixed in the hotfix branch.

Basic Git Branching Commands

Basic Git Branching Commands

The **git branch** command, when used without any options or additional arguments, lists all the local branches in the current repository, highlighting the branch you are currently on with an asterisk (*).

Syntax:

```
git branch
```

Example:

```
$ git branch  
* main  
feature-branch  
bugfix-branch
```

In this example, **main** is the current branch, and the other local branches are **feature-branch** and **bugfix-branch**.

Basic Git Branching Commands

The **git branch -a** command lists all branches in the repository, including both local and remote branches. Remote branches are usually prefixed with **remotes/** followed by the remote repository name.

Syntax:

```
git branch -a
```

Example:

```
$ git branch -a
* main
  feature-branch
  bugfix-branch
  remotes/origin/HEAD ->
  origin/main
  remotes/origin/feature-branch
  remotes/origin/main
```

In this example, **main**, **feature-branch**, and **bugfix-branch** are local branches, while **remotes/origin/feature-branch** and **remotes/origin/main** are remote branches.

Basic Git Branching Commands

The **git branch <branch_name>** command is used to create a new branch within the Git repository.

Syntax:

```
git branch <branch_name>
```

Example:

```
git branch feature-branch
```

In this example, a new branch named **feature-branch** is created in the current Git repository.

Basic Git Branching Commands

The following command is used to delete a branch from the repository:

Syntax:

```
git branch -d <branch-name>
```

The **-d** option is used to delete a branch that has been fully merged into its upstream branch or the current **HEAD**.



Basic Git Branching Commands

Example:

```
git branch -d test_branch
```

If the changes are not merged, Git will throw the following error while deleting:

```
error: The branch 'test_branch' is not fully merged with 'master.'
```

```
If you are sure to delete it, run 'git branch -D test_branch.'
```


Basic Git Branching Commands

The following command is used to delete a branch permanently:

Syntax:

```
git branch -D <branch_name>
```

The following command is used to delete a remote branch:

Syntax:

```
git push <remote_name> --delete <branch_name>
```

Or

```
git push <remote_name> :refs/heads/<branch_name>
```



Basic Git Branching Commands

The **git checkout** command is versatile and serves several purposes in Git, including switching between branches, checking out specific commits, and restoring files.

Syntax:

```
git checkout <branch_name>
```

Example:

```
git checkout feature-branch
```

In this example, the command switches the working directory to the branch named **feature-branch**.

Basic Git Branching Commands

The following command is used to create a new branch and immediately switch to it:

Syntax:

```
git checkout -b <branch_name>
```

Example:

```
git checkout -b feature-branch
```

In this example, the command creates a new branch called **feature-branch** and switches to it immediately.

Working with Branches



Duration: 10 Min.

Problem Statement:

You have been assigned a task to create, manage, rename, and delete branches in GitHub using Git commands, enhancing your ability to organize and collaborate on code efficiently.

Outcome:

By completing this task, you will be able to create, manage, rename, and delete branches in GitHub using Git commands, enhancing your ability to organize and collaborate on code efficiently.

Note: Refer to the demo document for detailed steps:
01_Working_with_Branches

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create branches using GitHub
2. Work with branches using Git commands
3. Rename and delete branches

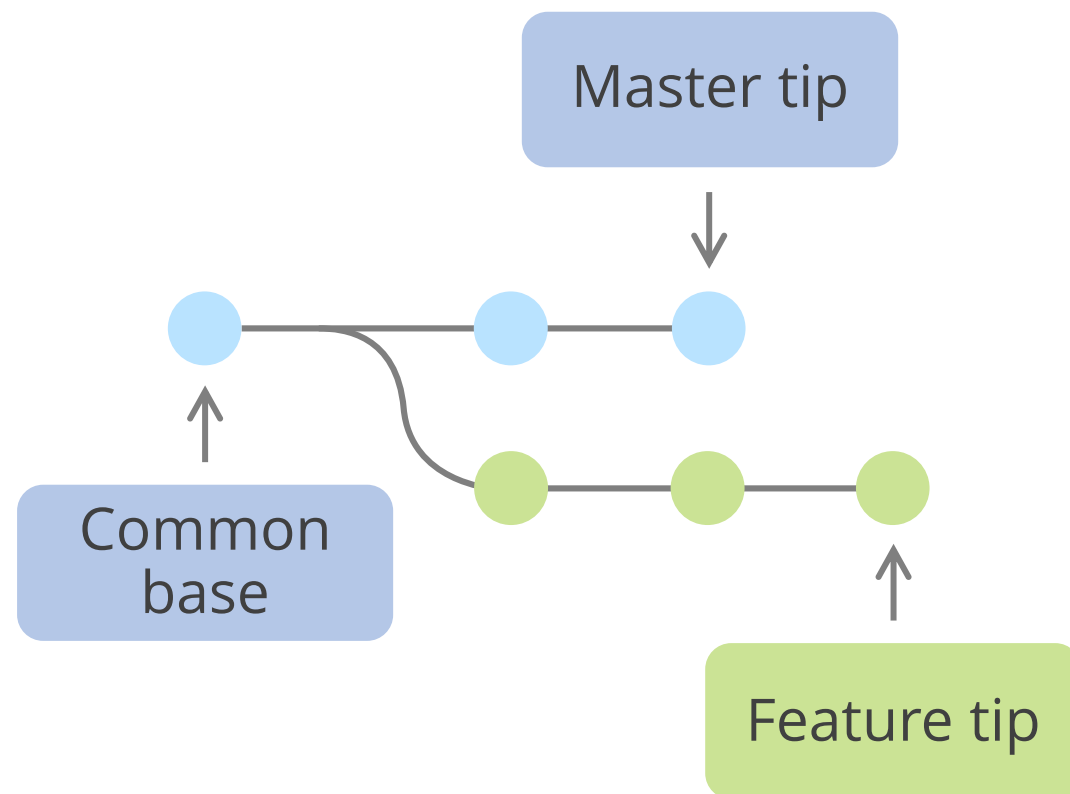


Merging Branches in Git

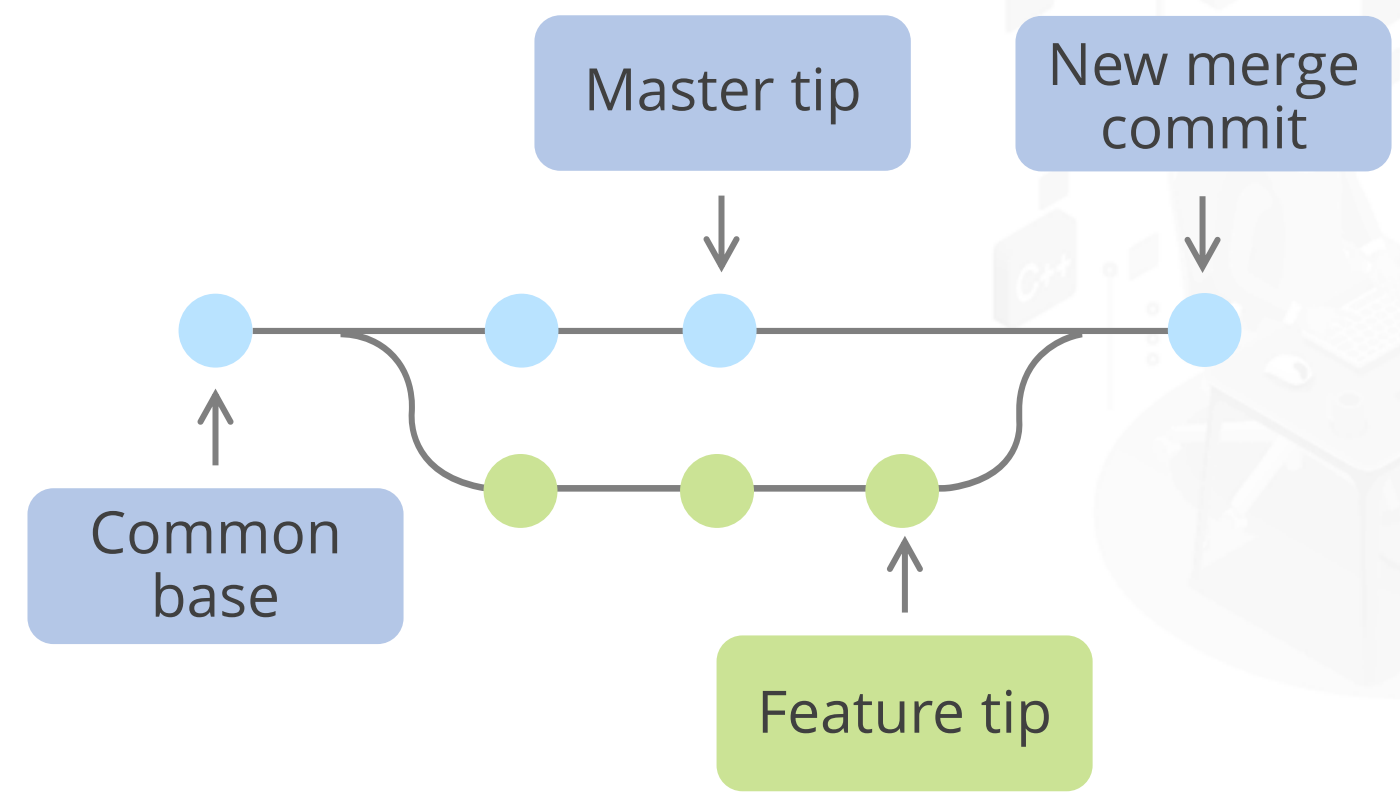
Merging Branches in Git

Merging branches in Git is a technique for integrating changes from different development lines into a single branch.

The Git merge combines multiple sequences of commits into one unified history. In most of the cases, git merge is used to combine two branches.



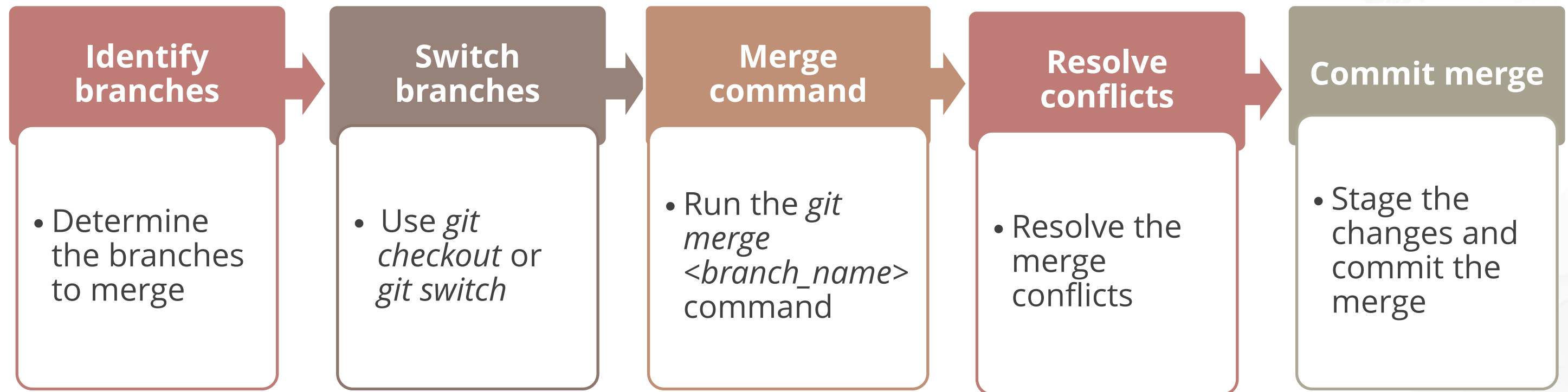
Before merging



After merging

Merging Branches in Git



The following shows the process of merging branches in a Git repository:



Basic Merge

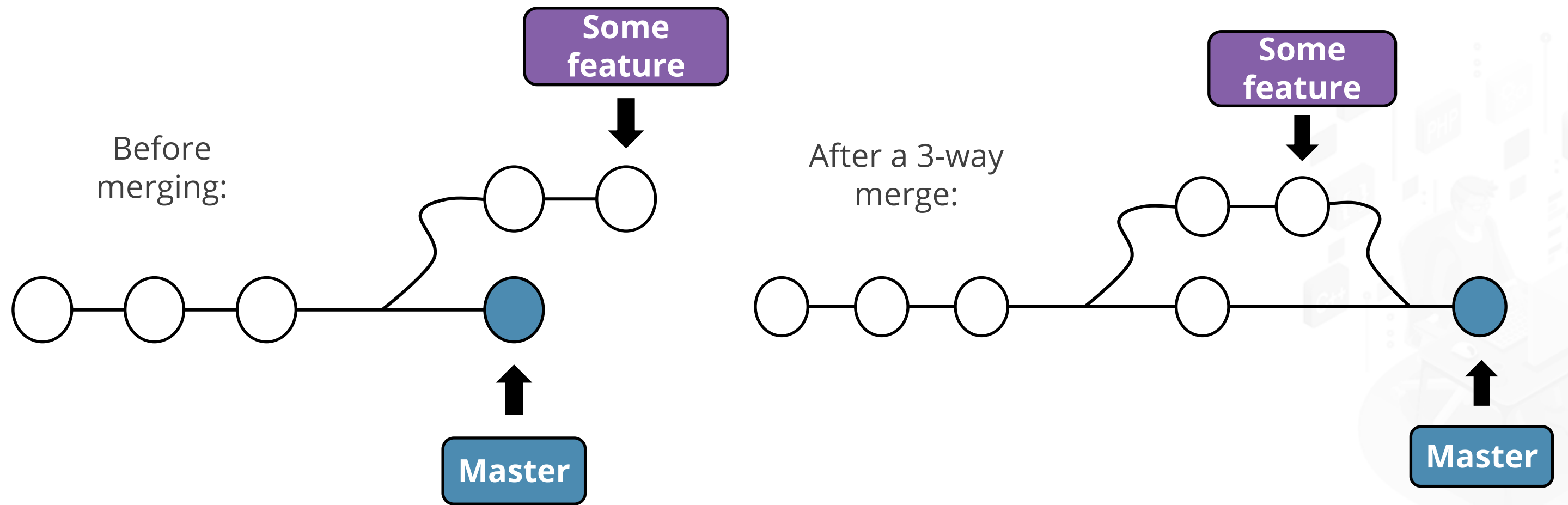
A basic merge in Git is the process of integrating changes from one branch into another.

The following happens during a basic merge operation:

-  **Three-way merge:** Git performs a three-way merge using the two branch tips and their common ancestor.
-  **Merge commit:** If there are changes in both branches, Git will create a new merge commit.



Basic Merge



Basic Merge

Following are the steps for basic Git merge:

1. Checkout the target branch

```
git checkout main
```

2. Merge the source branch

```
git merge feature-branch
```

These commands merge the changes from the **feature-branch** into the **main** branch.



Fast-Forward Merge

A fast-forward merge occurs when the current branch is an ancestor of the branch being merged. This means that there are no divergent changes, and Git can simply move the branch pointer forward.

The following happens during a fast-forward merge operation:



Pointer update: Instead of creating a new merge commit, Git just moves the pointer HEAD.

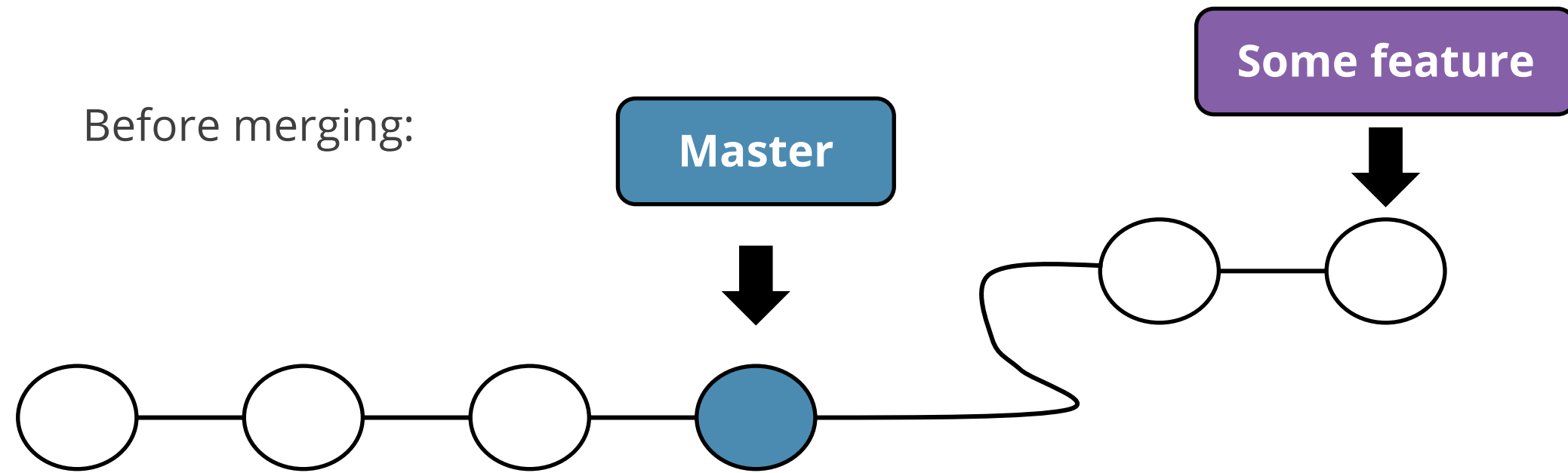


No merge commit: Git does not perform any new commit as the history is linear.

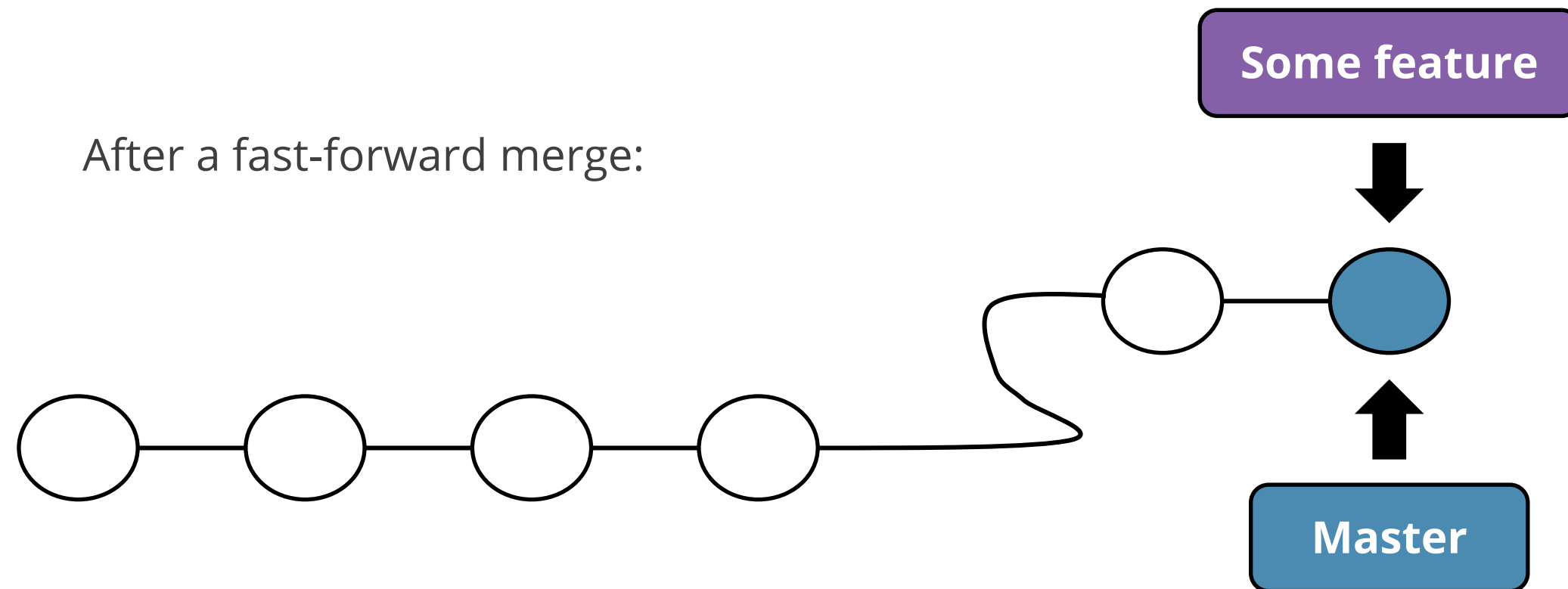


Fast-Forward Merge

Before merging:



After a fast-forward merge:



Fast-Forward Merge

Following are the steps for fast-forward Git merge:

1. Checkout the target branch

```
git checkout main
```

2. Merge the source branch

```
git merge feature-branch
```

If the **main** branch is an ancestor of the **feature-branch**, Git will perform a fast-forward merge.



Merging Branches in Git



Duration: 10 Min.

Problem Statement:

You have been assigned a task to merge branches in Git to integrate changes from one branch into another while maintaining a cohesive codebase and version history.

Outcome:

By completing this task, you will be able to merge branches in Git to integrate changes from one branch into another while maintaining a cohesive codebase and version history by creating and cloning a GitHub repository, managing branches, committing changes, and merging branches.

Note: Refer to the demo document for detailed steps:
[02_Merging_Branches_in_Git](#)

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new GitHub repository
2. Clone the GitHub repository
3. List all the branches in your repository
4. Create and switch to the new branch
5. Create a file and commit the changes
6. Check the status of the new branch
7. Switch back to the main branch
8. Merge the branches



Merge Conflicts in Git

Merging branches in Git integrates changes into the codebase, but sometimes conflicts arise when different branches modify the same lines of code.

Conflicts occur when Git encounters changes made to the same lines of code in separate branches while merging.

Reasons for merge conflicts

Git cannot automatically decide which version to keep, so it flags the conflict and requires manual intervention.

Merge Conflicts in Git

The following are the ways to resolve merge conflicts:

Resolve conflicts manually

Identifying the conflicted file manually and performing the required changes

Use a merge tool

Using popular merge tools, such as **kdifff3**, **meld**, and **Beyond Compare**, to resolve conflicts visually

Accept all the changes

Accepting all changes from one branch, either the current branch or the incoming branch

Abort merge

Aborting the merge process and returning to the state before the merge attempt

Resolving Merge Conflicts in Git



Duration: 10 Min.

Problem Statement:

You have been assigned a task to demonstrate how to resolve merge conflicts in Git to maintain a clean codebase and project continuity when conflicting changes occur.

Outcome:

By completing this task, you will be able to demonstrate how to resolve merge conflicts in Git to maintain a clean codebase and project continuity when conflicting changes occur by creating a new branch, making changes, handling conflicting changes, and resolving merge conflicts.

Note: Refer to the demo document for detailed steps:
[03_Resolving_Merge_Conflicts_in_Git](#)

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a new branch
2. Make changes to the new branch
3. Switch to the main branch and make conflicting changes
4. Merge the new branch and resolve the merge conflicts

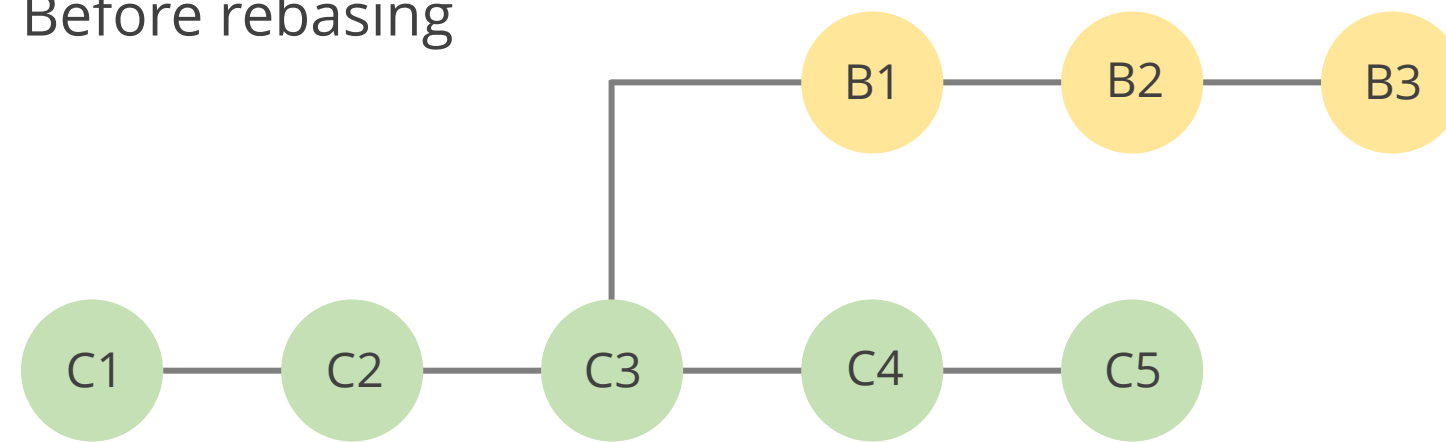


Rebasing in Git

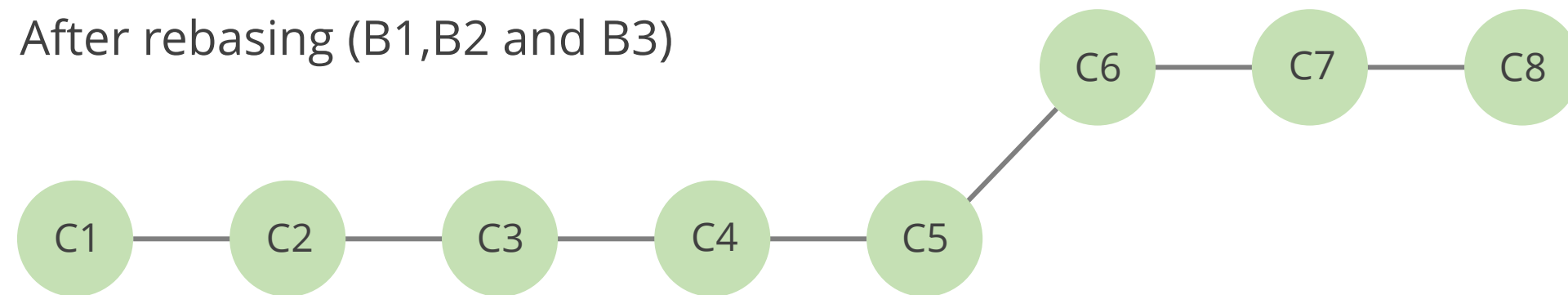
Rebasing in Git

It is a process of integrating a series of commits on top of another base tip. It takes all the commits of a branch and appends them to the commits of a new branch.

Before rebasing

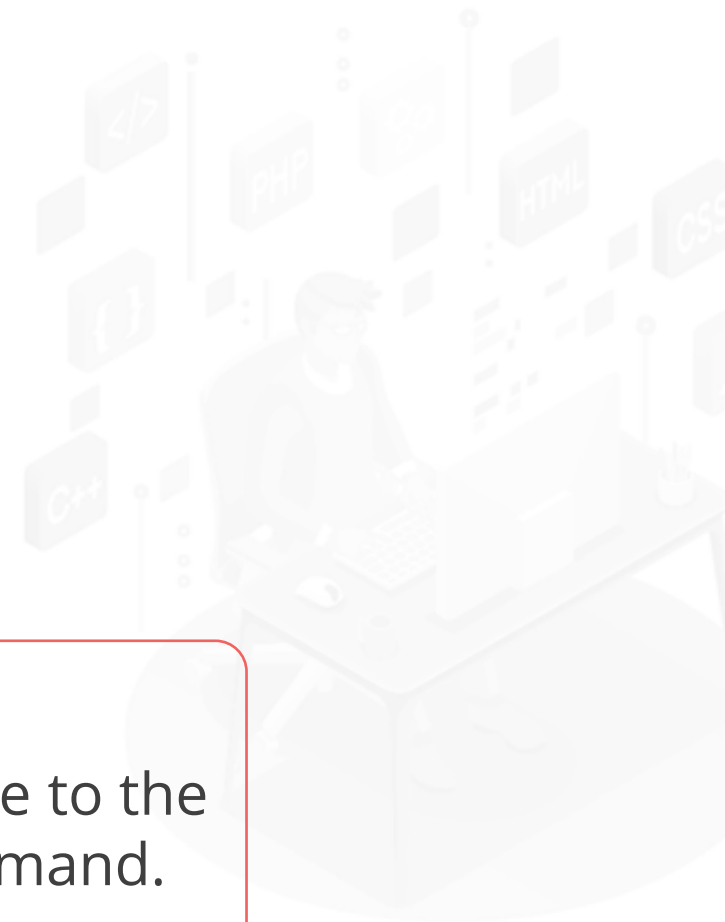
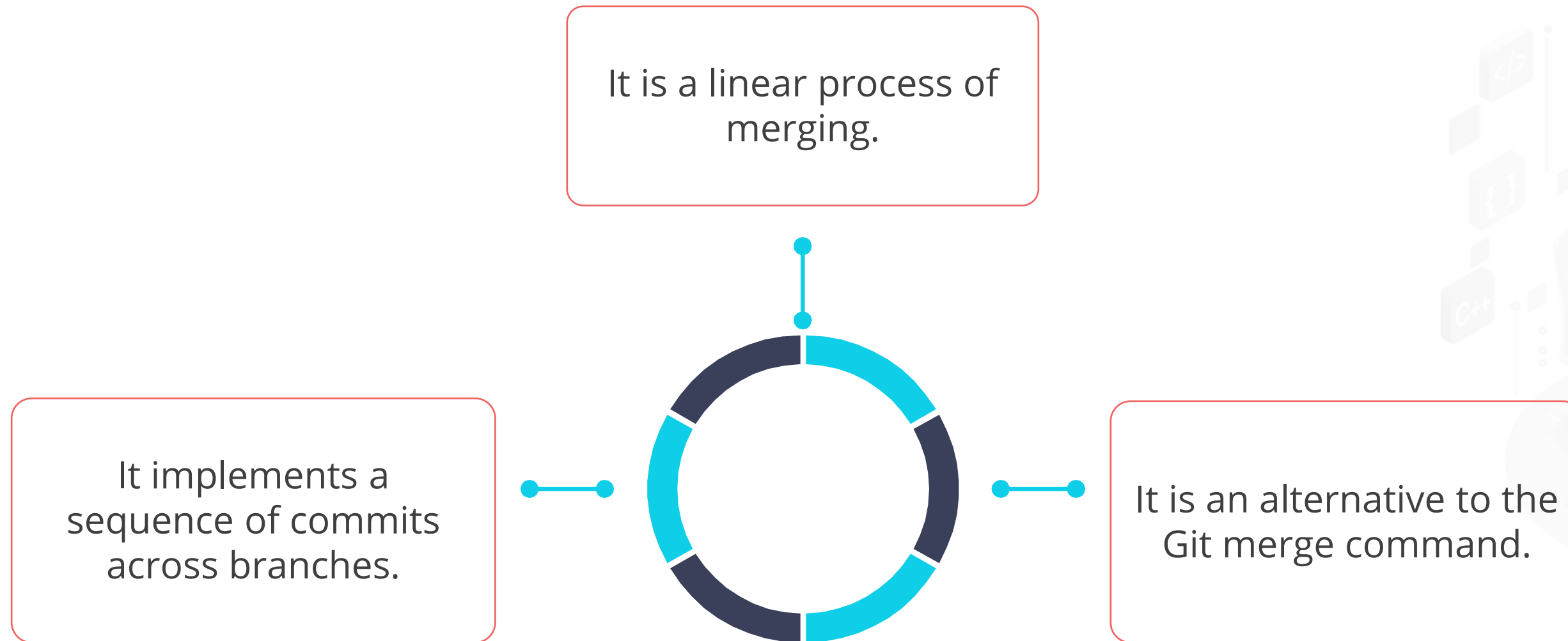


After rebasing (B1,B2 and B3)



Rebasing in Git

Git rebase is a process to reapply commits on top of another base trip.



Rebasing in Git

It simplifies the project history, enabling seamless tracing from feature completion back to the start.

Syntax:

```
git rebase [-i | --interactive] [ options ] [--exec cmd] [--onto newbase |  
--keep-base] [upstream [branch]]
```



Rebasing in Git



Duration: 10 Min.

Problem Statement:

You have been assigned a task to perform a rebase in Git for integrating changes from one branch to another while maintaining a linear commit history.

Outcome:

By completing this task, you will be able to perform a rebase in Git to integrate changes from one branch to another while maintaining a linear commit history by creating a repository, cloning it, and rebasing the branch.

Note: Refer to the demo document for detailed steps:
04_Rebasing_in_Git

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a repository
2. Clone the Git repository and rebase the branch to integrate the changes



Key Takeaways

- A branch in Git allows developers to create isolated copies of the codebase at specific points of the development line within the project.
- A Git branching model or Git flow is a set of guidelines or conventions that define how to create, manage, and use branches in a Git workflow.
- The main or master is the default branch available in the Git repository.
- The **git branch** command is used to manage branches in your Git repository. Branches are essential for organizing and managing different lines of development within a project.



Key Takeaways

- 🕒 The Git merge combines multiple sequences of commits into one unified history.
- 🕒 A basic merge in Git is the process of integrating changes from one branch into another.
- 🕒 A fast-forward merge occurs when the current branch is an ancestor of the branch being merged.
- 🕒 Rebasing is a process of integrating a series of commits on top of another base tip. It takes all the commits of a branch and appends them to the commits of a new branch.



TECHNOLOGY

Thank You