MODULE 6

VERSION CONTROL

GIT & GITHUB

Zero to Advance

Complete DevOps Guide

PRESENTATION OVERVIEW

Learning Roadmap

01 Version Control Fundamentals

Understanding what version control is, exploring Git and GitHub, and learning why they are essential for modern software development and DevOps practices.

02 Git Basics & Setup

Installing Git, mastering essential Linux commands, initializing repositories, understanding the three-stage workflow, and making your first commits.

03 GitHub & Remote Collaboration

Connecting local repositories to GitHub, pushing changes, managing multiple remotes, and understanding different Git hosting providers.

04 Branching & Merging

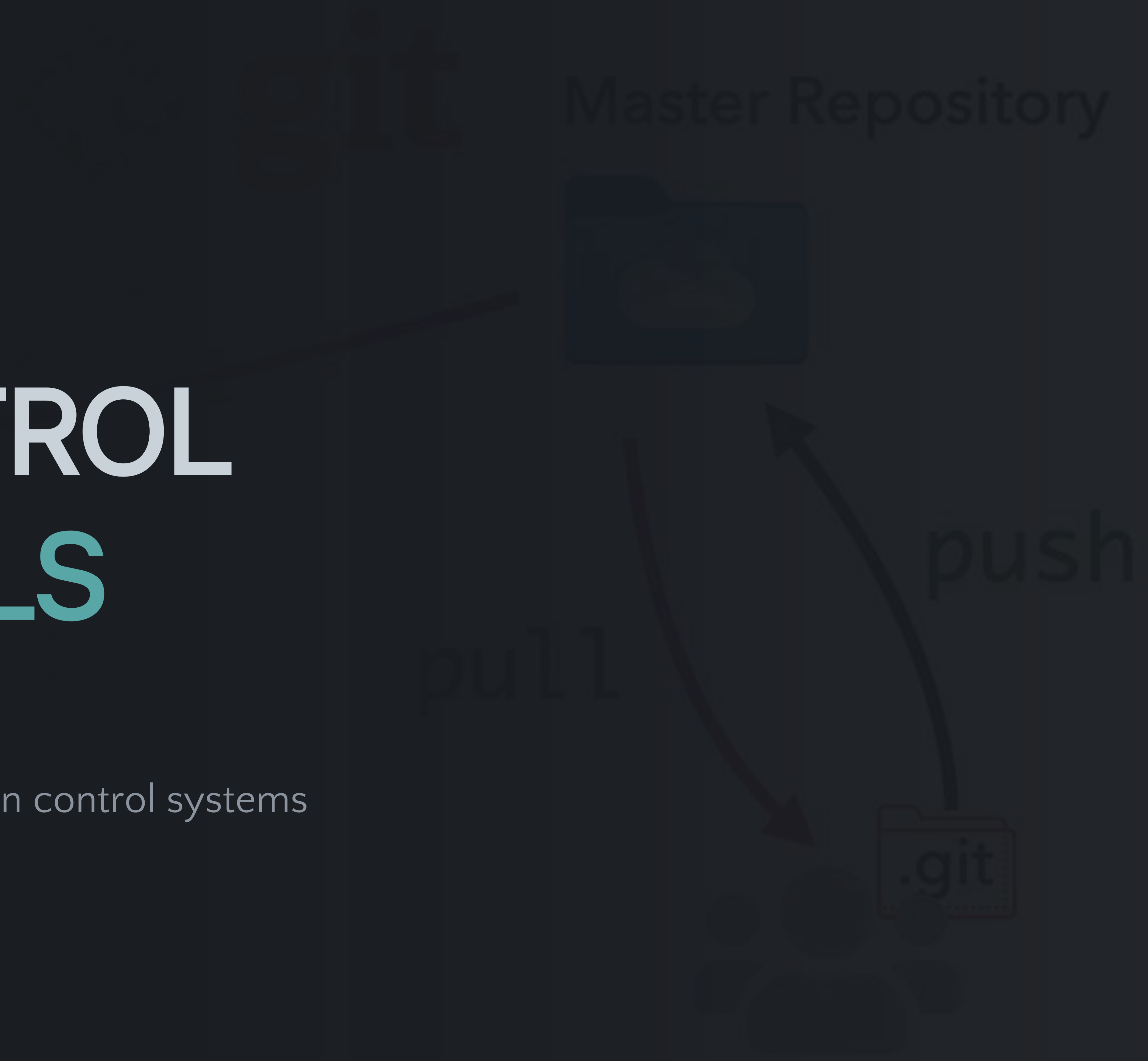
Mastering parallel development with branches, creating pull requests, understanding the fork-clone workflow, and collaborating safely.

05 Advanced Git Operations

Squashing commits, stashing changes, rebasing for clean history, and effectively resolving merge conflicts in collaborative environments.

06 DevOps Workflows & Best Practices

Industry-standard workflows, essential Git commands cheat sheet, interview preparation, and MCQs to test your knowledge.



01

VERSION CONTROL FUNDAMENTALS

Understanding the what, why, and how of version control systems

VERSION CONTROL BASICS

What is Version Control?

Version Control is a system that records changes to files over time so you can recall specific versions later. It's an essential tool for any developer, enabling you to track project evolution, collaborate safely, and recover from mistakes.

Tracks Changes

Monitors every modification to your files, creating a detailed log of what changed, when it changed, and who made the change.

Maintains History

Preserves the complete project timeline, allowing you to travel back in time and review or restore any previous state.

Enables Collaboration

Allows multiple developers to work on the same project simultaneously without overwriting each other's changes.

Real-World Example

If a mistake happens, you can go back to old versions. Imagine accidentally breaking your website's login feature. With version control, you can instantly revert to the last working version while you fix the bug.

Safety Net for Your Code

Version control acts as a safety net, allowing experimentation without fear. Break something? Simply roll back to a stable state.

CORE CONCEPTS

Git vs GitHub: Understanding the Difference

Git

The Tool

Git is a distributed version control system used to track changes in source code during software development. It's designed for coordinating work among programmers.

- ✓ **Distributed:** Every developer has a complete copy of the project history
- ✓ **Local:** Works entirely on your system without internet
- ✓ **Fast:** Operations are nearly instantaneous
- ✓ **Secure:** Cryptographic integrity checking

Created by: Linus Torvalds (creator of Linux)

GitHub

The Platform

GitHub is a cloud-based hosting platform for Git repositories. It provides a web-based graphical interface and collaboration features.

- ✓ **Hosting:** Store your Git repositories in the cloud
- ✓ **Share:** Make your code publicly accessible or keep it private
- ✓ **Collaborate:** Work with teams through pull requests and issues
- ✓ **Social:** Build a developer profile and contribute to open source

Access: Available anywhere with internet

Key Analogy

Git = Your local tool (like a word processor) · GitHub = Cloud service (like Google Drive) · You use Git to create and manage code, then push it to GitHub to share and collaborate.

VALUE PROPOSITION

Why Do We Use Git & GitHub?

Track Project History

Every change is recorded with who, what, when, and why. Never lose work again.

Team Collaboration

Multiple developers can work on the same project without conflicts. See who changed what and when.

Backup Code Online

Your code is safely stored in the cloud. Hard drive crashes? No problem. Your code lives on GitHub.

Manage Branches

Work on new features in isolation. Experiment without fear. Merge when ready.

DevOps Integration

Automated CI/CD pipelines. Deploy code with confidence. Test automatically on every change.

Open-Source Contribution

Contribute to projects worldwide. Build your reputation. Learn from the best developers.

Industry Requirement

Every DevOps & Software Engineer must know Git. It's not optional. Version control is the backbone of modern software development. Without it, you're not a professional developer – you're a hobbyist.

95%
of professional dev teams use Git

100M+
developers use GitHub

83M+
repositories on GitHub

02

GIT BASICS
& SETUP

Installation, configuration, and fundamental commands

GETTING STARTED

Installing Git & Essential Linux Commands

Downloading & Installing Git

Step 1: Visit the official Git website

```
https://git-scm.com
```

Step 2: Download for your OS

Windows / Linux / Mac – All platforms supported

Step 3: Verify installation

```
git --version
```

Should return something like: git version 2.42.0

Essential Linux Commands

These commands are crucial for navigating the terminal when working with Git:

`ls`	List files and directories
`pwd`	Print current directory path
`cd`	Change directory
`mkdir`	Create a new directory
`touch`	Create a new empty file
`rm`	Remove files or directories
`clear`	Clear the terminal screen
`cat`	Display file contents

Pro Tip

Master these Linux commands first. Git is primarily used through the command line interface (CLI). While GUI tools exist, professional developers use CLI for its speed, power, and flexibility. These commands are your foundation.

Windows: Use Git Bash

Mac: Use Terminal

Linux: Use bash/zsh

FIRST STEPS

Initializing a Git Repository

The git init command transforms any ordinary folder into a Git repository. This is the first step in starting version control for your project.

What happens when you run git init?

1. A hidden .git folder is created
2. This folder contains all Git metadata
3. Your project is now a Git repository
4. You can start tracking changes

The .git Directory

This hidden directory contains:

- **objects:** All your commits and files
- **refs:** Branch and tag references
- **HEAD:** Points to current branch
- **config:** Repository settings

Step-by-Step Process

Step 1: Navigate to your project folder

```
cd my-project
```

Step 2: Initialize Git repository

```
git init
```

Step 3: See the success message

```
Initialized empty Git repository in /path/to/my-project/.git/
```

Step 4: Verify (optional – shows hidden folder)

```
ls -la
```

Important Note

Never manually edit the .git folder. Let Git handle it.

One Repository

Each project gets its own Git repository.

Reversible

Delete .git folder to remove version control.

CORE CONCEPT

Git Working Areas: The Three-Stage Workflow

Understanding Git's three working areas is CRITICAL. This is the foundation of everything you do in Git. Miss this, and you'll be lost.

1 Working Directory

Where you write and edit code. Files here may or may not be tracked by Git.

Location: Your project's main folder

State: Files with unstaged changes

→

2 Staging Area

A buffer zone where you prepare changes before committing. Select exactly what you want to include.

Command: `git add filename`

State: Files ready to be committed

→

3 Repository

The .git folder containing all committed snapshots of your project. This is your permanent history.

Command: `git commit -m "message"`

State: Permanent project history

The Workflow Flow

Working Directory → Staging Area → Repository · You modify files in your working directory, stage the changes you want to include, then commit them to the repository. This gives you full control over what goes into each commit.

HANDS-ON

Making Your First Commit

A commit is a snapshot of your project at a specific point in time. Follow these steps to create your first commit and start tracking your project history.

1 Add File

Stage a specific file

```
git add file.txt
```

Stages only one file to the staging area

2 Stage All Files

Stage all changes at once

```
git add .
```

The dot means "current directory" - stages everything

3 Commit

Create a snapshot

```
git commit -m "First commit"
```

-m flag adds a descriptive message

What Git Records

What Changed

Exact line-by-line differences (diffs)

When It Changed

Timestamp with timezone information

Who Changed It

Author name and email address

✓ Good Commit Messages

- "Add user authentication"
- "Fix login button alignment"
- "Update API documentation"

✕ Bad Commit Messages

- "fixed stuff"
- "asdfghjkl"
- "WIP" (Work in Progress)

VERSION CONTROL IN ACTION

Managing Changes & Commit History

Removing Changes from Stage

Accidentally staged a file? Use git reset to unstage it. This removes the file from staging but keeps it on disk.

```
git reset file.txt
```

Removes from staging, NOT from disk

Viewing Project History

See all commits with detailed information about each change in your project.

```
git log
```

Commit ID: Unique hash (e.g., a1b2c3d)
Author: Name and email
Date: When it happened
Message: What was changed

Making Multiple Commits

Each commit should represent one logical change. This makes history clean and easy to understand.

Good:

```
git commit -m "Added login feature"
```

```
git commit -m "Fixed password validation bug"
```

Bad:

```
git commit -m "Fixed stuff"
```

Useful Git Log Options

One-line summary:

```
git log --oneline
```

Graph view:

```
git log --graph --oneline
```

Last 5 commits:

```
git log -5
```

Pro Tip: Commit Often, Perfect Later

Make commits frequently with clear messages. You can always clean up your history later with advanced commands like squash and rebase. It's better to have too many commits than to

ADVANCED UNDO

Resetting & Undoing Changes

Made a mistake? Git provides powerful ways to undo changes. But be careful – some methods are permanent and dangerous.

Soft Reset

Moves the last commit back to the staging area. Safe and reversible.

Command:

```
git reset --soft HEAD~1
```

Use case: Forgot to include something in the last commit

What it does:

- Keeps changes in staging area
- Removes commit from history
- No data loss

Hard Reset

PERMANENTLY DELETES commits and changes. Cannot be undone easily.

Command:

```
git reset --hard HEAD~1
```

⚠️ **DANGEROUS:** Deletes commits permanently

What it does:

- Removes commit from history
- Deletes all changes from working directory
- Data is GONE

Safety First: When to Use Each

Use Soft Reset When:

- You forgot to include files in the last commit
- You want to amend the commit message
- You're being cautious

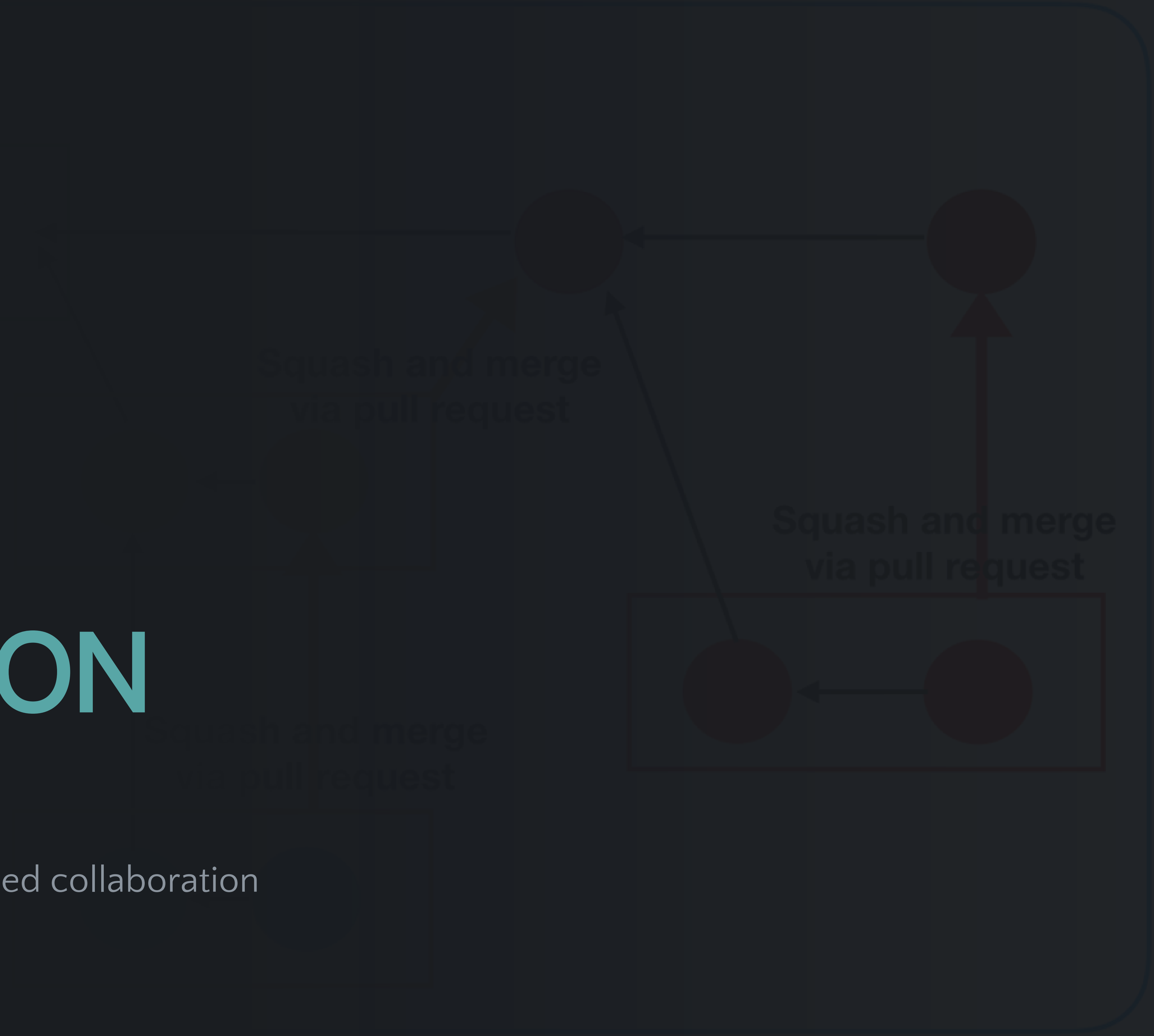
Use Hard Reset When:

- You absolutely need to delete sensitive data
- You're cleaning up experimental commits
- You understand it's permanent

03

GITHUB & REMOTE COLLABORATION

Connecting local development with cloud-based collaboration


Squash and merge via pull request
Squash and merge via pull request
Squash and merge via pull request

CLOUD PLATFORM

Getting Started with GitHub

GitHub brings your local Git repositories to the cloud, enabling collaboration, backup, and project management. Here's how to start:

1 Create Account

Sign up at github.com for a free account.

- Choose a professional username
- Use your real name and email
- Add a profile picture and bio

2 Create Repository

Click "New repository" to create a new repo.

- Choose a descriptive name
- Add a README if desired
- Select Public or Private

3 Copy URL

Copy the repository URL for remote connection.

- HTTPS: https://github.com/...
- SSH: git@github.com:...
- Use HTTPS for beginners

Repository Best Practices

Repository Name Tips:

- Use lowercase with hyphens
- Be descriptive and concise
- Avoid generic names

Visibility Options:

- Public: Anyone can see, great for portfolios
- Private: Only you and collaborators can see

CONNECTIVITY

Connecting Local Git to Remote Repository

Adding Remote Connection

Connect your local repository to GitHub using the git remote add command. This creates a link between your local project and the cloud.

Command:

```
git remote add origin
```

- remote: Manage remote connections
- add: Create a new connection
- origin: Default name for your main remote

Verifying Remote Connection

Check that your remote was added correctly and see all connected remotes.

Command:

```
git remote -v
```

Example Workflow

1. Add remote (HTTPS):

```
git remote add origin https://github.com/username/my-project.git
```

2. Verify connection:

```
git remote -v
```

Expected output:

```
origin https://github.com/username/my-project.git (fetch) origin
https://github.com/username/my-project.git (push)
```

Other Git Providers

GitHub isn't the only option. Alternatives include:

- **GitLab**
  DevOps platform with CI/CD
- **Bitbucket**
  Atlassian's solution
- **Azure Repos**
  Microsoft's cloud hosting

Multiple Remotes

You can add multiple remotes. This is common in open-source. You might have origin (your fork) and upstream (original repo). Use descriptive names.

SYNCING

Pushing Changes & Managing Multiple Remotes

Pushing Local Changes to Remote

Send your local commits to GitHub so others can see and collaborate on your work.

Command:

```
git push origin main
```

- push: Send commits to remote
- origin: The remote repository
- main: The branch name to push

First Push Setup

Git may ask you to set the upstream branch:

Command:

```
git push --set-upstream origin main
```

Or use the shorthand:

Shorthand:

```
git push -u origin main
```

Adding Multiple Remotes

Connect to multiple remote repositories. Essential for open-source collaboration.

Add upstream remote:

```
git remote add upstream
```

Common Multi-Remote Setup

origin
Your fork on GitHub (you push here)

upstream
Original repository (you pull updates from here)

Verify all remotes:

```
git remote -v
```

Why multiple remotes? In open-source, you fork a repo, clone your fork, then add upstream to sync with the original project.

Push Workflow Summary

1) git add . → 2) git commit -m "message" → 3) git push origin main · This is the most common Git workflow. Master these three commands, and you can use Git professionally.

04

BRANCHING & MERGING

Parallel development and safe collaboration workflows

CORE CONCEPT

Understanding Branches

Branches are the superpower of Git. They let you work on multiple features simultaneously without affecting the main codebase. Think of them as parallel universes for your code.

What is a Branch?

A branch is a separate line of development. It creates an isolated environment where you can make changes without affecting other work.

- **Default branch:** main (or master in older repos)
- **Purpose:** Work without breaking the main code
- **Benefit:** Experiment safely and merge when ready

Branch Visualization

Timeline view:

```
main: ●——●——● \ feature: ●——●——●
```

After merging:

```
main: ●——●——●————————● \ feature: ●——●——●
```

Why Branches Matter

Branches enable parallel development. The main branch stays stable while features are developed in isolation. Each developer can work on their own branch without conflicts. When a feature is complete and tested, it's merged back. This is how professional teams ship quality software.

BRANCH OPERATIONS

Creating, Switching & Merging Branches

Creating a Branch

Create new branch:

```
git branch feature1
```

What it does: Creates a copy of current branch named "feature1"

Switching Branches

Switch to branch:

```
git checkout feature1
```

What it does: Changes your working directory to the feature1 branch

Combined Command

Create + switch in one step:

```
git checkout -b feature1
```

Pro tip: This is the most common way professionals create branches

Merging Branch to Main

After completing work on your feature, merge it back to main:

Step 1: Switch to main

```
git checkout main
```

Step 2: Merge feature branch

```
git merge feature1
```

What happens: All changes from feature1 are integrated into main

Pushing Branch to GitHub

Push new branch:

```
git push origin feature1
```

Result: Your branch is now visible on GitHub for collaboration

Branch Naming

Use descriptive names: feature/login, bugfix/header, hotfix/crash

Delete Branch

After merge: `git branch -d feature1`

List Branches

See all: `git branch`

COLLABORATION

Pull Requests: The Heart of Collaboration

What is a Pull Request?

A Pull Request (PR) is a request to merge changes from one branch into another. It's the primary collaboration method on GitHub.

- Purpose: Propose changes before merging
- Review: Team reviews code quality
- Discuss: Comments and suggestions
- Approve: Merge after approval

Think of it as: "Hey team, I made some changes. Can you review and merge them?"

Creating Your First Pull Request

1. Push your branch

```
git push origin feature-branch
```

2. Open GitHub
   Navigate to your repository on GitHub.com
3. Click "New Pull Request"
   GitHub shows a "Compare & pull request" button
4. Compare → Create PR
   Add title, description, then click "Create"

Why PRs Are Game-Changers

Pull requests make code review systematic. They catch bugs, improve quality, and spread knowledge. No code reaches main without review. This is how professional teams maintain quality at scale.

ADVANCED COLLABORATION

Force Push, Merging & Fork Workflow

Force Push (Use with Caution)

Overwrites remote history. Used when you need to rewrite commits after pushing.

Command:

```
git push --force
```

⚠️ DANGEROUS in teams: Can overwrite others' work

When to use:

- Fixing PR commits
- Squashing before merge
- Personal branches only

Merging a PR

1. Review code: Check changes
2. Resolve conflicts: Fix if needed
3. Click Merge: Confirm merge
4. Delete branch: Clean up (optional)

Fork → Clone Workflow

Standard workflow for contributing to open-source projects you don't have write access to.

1 **Fork repository**
Click "Fork" on GitHub to create your copy

2 **Clone your fork**

```
git clone
```

3 **Make changes**
Create branch, commit changes

4 **Create PR**
Push to your fork, create PR to original

What is Upstream?

Add upstream remote:

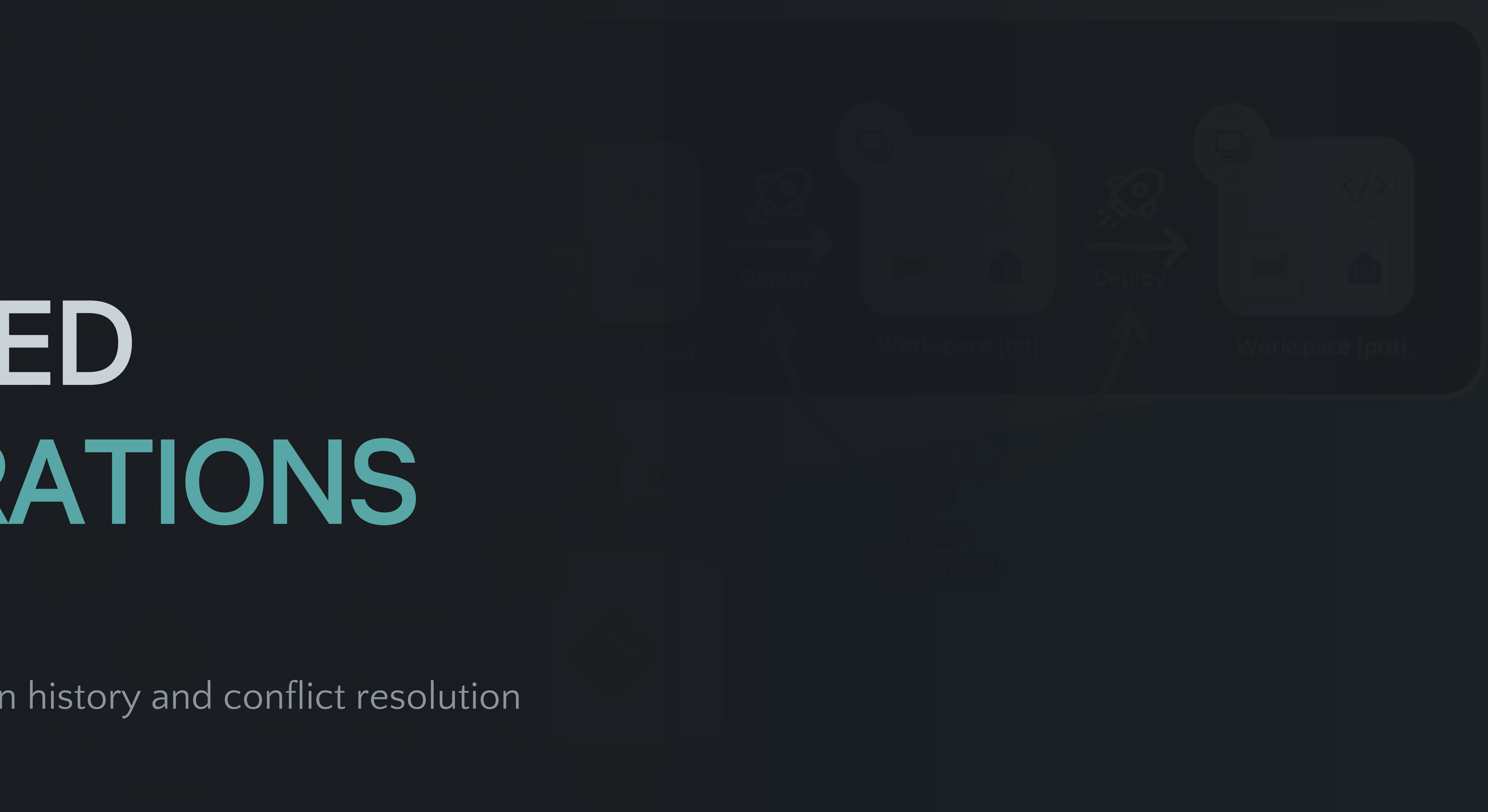
```
git remote add upstream
```

Upstream = original repository. Keep your fork synced by pulling from upstream.

05

ADVANCED GIT OPERATIONS

Powerful commands for clean history and conflict resolution



HISTORY MANAGEMENT

Squashing Commits & Stashing Changes

Squashing Commits

Combine multiple commits into one clean commit. Perfect for cleaning up messy history before merging.

Interactive rebase (squash last 3 commits):

```
git rebase -i HEAD~3
```

> **What happens:** Git opens an editor where you choose which commits to squash

Example Squash Workflow

1. Start interactive rebase:

```
git rebase -i HEAD~3
```

2. Git shows:

```
pick a1b2c3d Add login feature pick d4e5f6g Fix password validation pick g7h8i9j Update tests
```

3. Change to:

```
pick a1b2c3d Add login feature squash d4e5f6g Fix password validation squash g7h8i9j Update tests
```

Stashing Changes

Temporarily save uncommitted changes when you need to switch contexts quickly.

Save current changes:

```
git stash
```

Restore stashed changes:

```
git stash pop
```

Clear all stashes:

```
git stash clear
```

When to Use Stash

- Scenario 1: Need to switch branches but have unfinished work
- Scenario 2: Pull changes but have local modifications
- Scenario 3: Quick fix needed on another branch

Pro Tips

Squash before merging to keep history clean. Use stash when interrupted - it saves time and prevents lost work. These are advanced but essential for professional workflows.

HISTORY REWRITING

Rebase & Reset: Maintaining Clean History

Using Rebase Command

Rebase moves commits to a new base commit, creating a linear history without unnecessary merge commits.

Rebase feature branch onto main:

```
git rebase main
```

What it does: Takes your feature branch commits and applies them on top of main

Rebase vs Merge

Merge	Rebase
Creates merge commit, preserves history	Linear history, rewrites commits

Use rebase for: Clean, linear project history

Reset with Hard Flag

Reset to a specific commit, permanently deleting changes. This is the nuclear option – use with extreme caution.

Reset to specific commit:

```
git reset --hard
```

⚠️ **PERMANENT:** Deletes changes forever. Cannot be undone.

Finding Commit ID

View log:

```
git log --oneline
```

Commit ID: First 6–8 characters are enough (e.g., a1b2c3d)

Safety Comparison

- git reset --soft ← Safe
- git reset --mixed ← Medium
- git reset --hard ← DANGEROUS

⚠ **Golden Rule: Never Rewrite Shared History**

Only use rebase and reset on your own branches. Never on shared branches like main. Rewriting shared history confuses teammates and can break deployments. When in doubt,

CONFLICT RESOLUTION

Merge Conflicts: Causes & Resolution

What Causes Merge Conflicts?

A merge conflict occurs when Git cannot automatically merge changes because the same file was edited in conflicting ways.

- Same line edited: Two people change the same line differently
- File deleted: One person edits, another deletes
- Overlapping changes: Changes are too close for Git to decide

Git's dilemma: "I see two different changes to the same place. Which one is correct?"

Conflict Markers

What Git shows in conflicted file:

```
<<<<<<< HEAD Current branch's changes ======= Incoming branch's changes
>>>>>>> feature-branch
```

Resolving Merge Conflicts

1. **Open conflicted file**
   Find the file with conflict markers
2. **Fix conflict markers**
   Edit the file to desired final state
3. **Add file**
   ```
   git add conflicted-file.js
   ```
4. **Commit**
   ```
   git commit -m "Resolve merge conflict"
   ```

Prevention Strategies

- Pull frequently to stay updated
- Communicate with team about file changes
- Make smaller, focused changes
- Use feature branches to isolate work

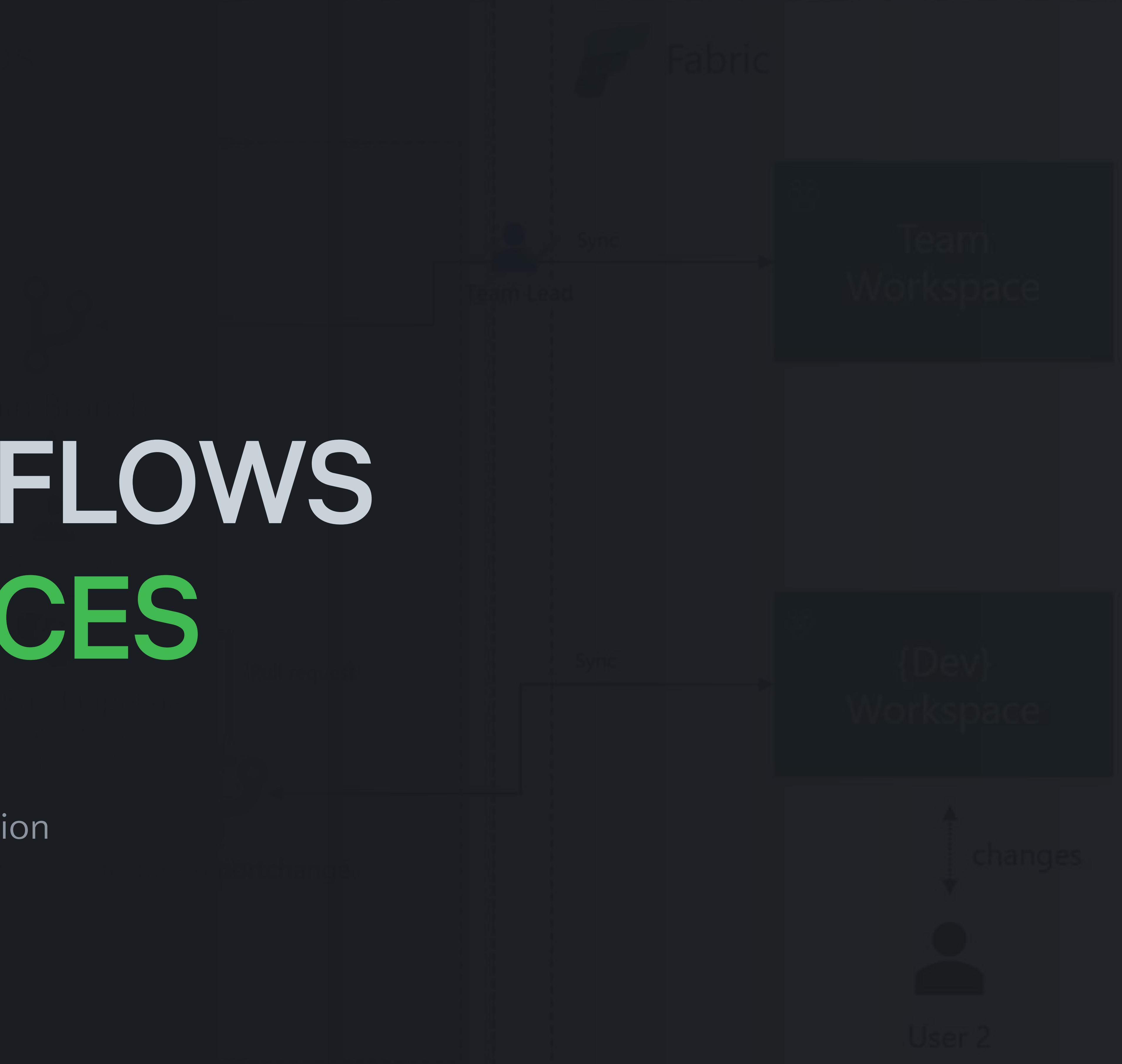
Pro Tip: Conflicts Are Normal

Don't panic when conflicts happen. They're a natural part of collaboration. Good communication and frequent pulls minimize conflicts. When they occur, resolve them calmly and carefully.

06

DEVOPS WORKFLOWS & BEST PRACTICES

Industry standards, tools, and interview preparation



INDUSTRY STANDARD

Real DevOps Git Workflow

This is the exact workflow used by professional DevOps teams worldwide. Master this flow, and you're ready for enterprise development.

1 Pull Latest Code

```
git pull origin main
```

Start with the most up-to-date codebase

2 Create Feature Branch

```
git checkout -b feature-login
```

Isolate your work in a dedicated branch

3 Make Changes & Commit

```
git add . && git commit -m "Added login feature"
```

Stage and commit your changes with clear messages

4 Push Branch

```
git push origin feature-login
```

Upload your branch to GitHub for collaboration

5 Create Pull Request

Open PR on GitHub for code review

6 Code Review + CI Pipeline

Team reviews code while automated tests run

7 Merge & Deploy

After approval, merge PR and auto-deploy to production

DevOps Tools

- ✓ Jenkins
- ✓ GitLab CI
- ✓ Kubernetes
- ✓ GitHub Actions
- ✓ Docker
- ✓ Ansible

∞ The Power of Automation

Git is the backbone of DevOps automation. Every push triggers pipelines that test, build, and deploy. This enables teams to ship code multiple times per day with confidence. Git isn't just version control – it's the foundation of modern software delivery.

REFERENCE

Git Commands Cheat Sheet

Bookmark this page! These are the most important Git commands you'll use daily.

◆ Setup

Set name:

```
git config --global user.name "Name"
```

Set email:

```
git config --global user.email "email"
```

◆ Repository

Initialize:

```
git init
```

Clone:

```
git clone
```

◆ Status & History

Status:

```
git status
```

Log:

```
git log
```

Diff:

```
git diff
```

◆ Add & Commit

Stage all:

```
git add .
```

Commit:

```
git commit -m "message"
```

◆ Branching

List:

```
git branch
```

Create + switch:

```
git checkout -b branch
```

Merge:

```
git merge branch
```

◆ Remote

Add remote:

```
git remote add origin
```

Push:

```
git push origin main
```

Pull:

```
git pull origin main
```

◆ Stash

Stash:

```
git stash
```

Pop:

◆ Undo

Soft reset:

```
git reset --soft HEAD~1
```

Hard reset:

◆ Advanced

Rebase:

```
git rebase main
```

Interactive rebase:

INTERVIEW PREPARATION

Interview Questions & Answers

These are the most common Git interview questions from beginner to advanced levels. Know these answers by heart.

Q1: What is Git?
Git is a distributed version control system used to track changes in source code and manage collaboration.

Q2: Difference between Git and GitHub?
Git is a local tool for version control, while GitHub is a cloud platform that hosts Git repositories.

Q3: What is a repository?
A repository is a storage location that contains project files and their complete version history.

Q4: What is a commit?
A commit is a snapshot of changes saved in the Git repository with a message.

Q5: What is staging area?
The staging area is where files are prepared before committing to the repository.

Q6: What is a branch?
A branch is a separate line of development used to work on features independently.

Q7: What is merge?
Merge combines changes from one branch into another branch.

Q8: What is rebase?
Rebase moves commits to a new base commit to maintain a clean, linear history.

Q9: What is stash?
Stash temporarily saves uncommitted changes without committing them.

Q10: What is a pull request?
A pull request is a request to merge code changes into another branch after review.

Q11: What is upstream?
Upstream refers to the original repository from which a fork was created.

Q12: What is merge conflict?
A merge conflict occurs when Git cannot automatically merge changes.

Q13: Reset vs revert?
Reset removes commits from history, while revert creates a new commit that undoes changes.

Q14: git fetch vs git pull?
Fetch downloads changes without merging, while pull fetches and merges changes.

Q15: Why Git in DevOps?
Git enables CI/CD automation, collaboration, rollback, and version tracking.

Interview Success Tips

Don't just memorize – understand the concepts. Be ready to explain with examples. Practice commands. Show enthusiasm for learning. Every DevOps role requires Git proficiency.

TEST YOUR KNOWLEDGE

MCQs: Test Your Knowledge

Quick knowledge check! Answers are highlighted in green.

1. Git is a:
- A) Centralized VCS
- ✓ B) Distributed VCS
- C) Database
- D) Editor

2. Which command initializes a repository?
- A) git start
- ✓ B) git init
- C) git begin
- D) git repo

3. Where are files after commit?
- A) Working directory
- B) Staging area
- ✓ C) Git repository
- D) RAM

4. Which command stages all files?
- A) git stage
- B) git add *
- ✓ C) git add .
- D) git commit

6. Which command creates a new branch?
- A) git checkout
- ✓ B) git branch new
- C) git new branch
- D) git create

7. Which command pushes code?
- A) git send
- B) git upload
- ✓ C) git push
- D) git move

8. What causes merge conflicts?
- ✓ A) Same file edited differently
- B) Different branches
- C) Large files
- D) Wrong commit message

9. Which command temporarily saves changes?
- A) git save
- ✓ B) git stash
- C) git hold
- D) git pause

CONGRATULATIONS

From Zero to Hero

✓ Git tracks changes · Every commit is a snapshot

✓ GitHub shares code · Cloud collaboration

✓ Branches = safe development · Isolate features

✓ PR = collaboration · Code review workflow

✓ Rebase & squash = clean history · Professional

✓ Stash = temporary save · Context switching

Remember These Fundamentals

Git = Local	GitHub = Remote	Commit Often
Your tool on your machine	Cloud platform for sharing	Small, frequent commits
One Feature = One Branch	DevOps Relies on Git	Practice Daily