Version Control with Git

Your First Steps into Collaborative Development

Mehdi Lotfinejad

🎯 Learning Objectives

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

1. **Explain** what version control systems are and why Git has become the industry standard
2. **Clone** a remote repository and create branches for independent development
3. **Merge** branches and understand the basic workflow of collaborative coding
4. **Push** your local changes to a remote repository and **pull** updates from teammates
5. **Identify** merge conflicts and resolve them using basic conflict resolution strategies

The Group Project Nightmare

Without Version Control:

- Everyone edits the same files
- Changes get overwritten
- No way to undo mistakes
- Chaos and confusion
- "Who broke the code?!"

With Git:

- Track every change
- Work independently
- Merge intelligently
- Undo anything
- "When did this break?"

Why Git Dominates Development

"Git is a widely used distributed version control system that allows software development teams to have multiple local copies of the project's source code that are independent of each other." — Red Hat Developer

Statistics:

- **93%** of professional developers use Git (Stack Overflow 2023)
- **Distributed:** Everyone has complete history
- **Fast:** All operations are local
- **Branching:** Work safely without breaking production

What is Version Control?

Version Control Systems (VCS) track changes to files over time:

- **Recall specific versions** from history
- **Collaborate** without overwriting each other
- **Experiment safely** with new features
- **Backup** your entire project history
- **Understand** what changed, when, and why

Essential Tool: "Version control tools are essential for backing up code, collaborating on codebases, and rolling back to earlier versions if required." — Mozilla Developer Network

🔄 Traditional vs. Distributed Version Control

Traditional VCS (Centralized)

```
Central Server
      |
      |
  [Users]
```

- Single point of failure
- Requires network access
- All history on server
- Example: SVN

Git (Distributed)

```
Remote Repository
     |
  +---+---+
[User A] [User B]
(Full)   (Full)
```

- Complete local copy
- Work offline
- Full history everywhere
- Fast operations

Git's Three-Stage Architecture

Understanding Git's workflow is essential:

```
Working Directory  →  Staging Area  →  Repository  →  Remote
   (edit files)      (git add)       (git commit)   (git push)
```

1. **Working Directory:** Your actual files where you make changes
2. **Staging Area (Index):** Preparation zone for your next commit
3. **Local Repository:** Database of all committed snapshots
4. **Remote Repository:** Shared repository (GitHub, GitLab)

This three-stage process gives you precise control over what you commit.

📚 Core Concept 1: Cloning

Cloning creates a complete local copy of a remote repository.

```
# Clone a repository from GitHub
git clone https://github.com/username/project-name.git

# This creates a folder named 'project-name' with all files and history
cd project-name

# Check the repository status
git status
```

When you clone, Git automatically sets up a connection to the original repository (called "origin"). This lets you push and pull changes later.

What Cloning Gives You

After cloning, you have:

- **Complete history:** Every commit, branch, and tag
- **Remote connection:** Automatically linked to "origin"
- **Working copy:** All files at the latest commit
- **Independence:** Can work offline, commit locally

```
# See remote repositories
git remote -v

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

🌿 Core Concept 2: Branching

Branches are Git's killer feature—parallel development spaces.

```
# Create a new branch named 'feature-login'
git branch feature-login

# Switch to the new branch
git checkout feature-login

# Or do both in one command (modern approach)
git checkout -b feature-login

# See all branches (* marks your current branch)
git branch
```

Think of branches as parallel universes for your code—try new features without affecting the main codebase.

Branch Workflow Visualization

```
main branch:      A---B---C---F---G
                       \       /
feature branch:         D---E
```

Key Points:

- **main branch:** Stable, production-ready code
- **Feature branches:** Independent development
- **Merge:** Integrate features back to main
- **Safety:** Bugs in features don't affect main

"A simple example of branching and merging with a workflow that you might use in the real world demonstrates how branches enable parallel development." — Git Documentation

Making Changes and Committing

Once on a branch, edit files and commit changes:

```
# Edit some files in your editor
# Then check what changed
git status

# Stage specific files for commit
git add login.py
git add templates/login.html

# Or stage all changed files
git add .

# Commit with a descriptive message
git commit -m "Add user login functionality"
```

Good Commit Messages Matter

Your commit history is your project's story.

❌ **Bad Messages**

```
git commit -m "changes"
git commit -m "fixed stuff"
git commit -m "updates"
git commit -m "asdf"
```

✅ **Good Messages**

```
git commit -m "Add login validation"
git commit -m "Fix empty username bug"
git commit -m "Refactor auth logic"
git commit -m "Update dependencies"
```

Best Practice: Write messages that explain *why* you made changes, not just *what* changed.

🔀 Core Concept 3: Merging

Merging integrates changes from one branch into another.

```
# First, switch to the branch you want to merge INTO
git checkout main

# Then merge the feature branch
git merge feature-login
```

Two types of automatic merges:

- **Fast-forward merge:** Target branch hasn't changed
- **Automatic merge:** Git intelligently combines independent changes

> If branches haven't touched the same lines of code, Git automatically combines them!

Understanding Merge Conflicts

Conflicts occur when two branches modify the same lines differently.

```
def calculate_total(items):
<<<<<<< HEAD
    # Current branch version
    total = sum(item.price for item in items)
    return total * 1.1  # Add 10% tax
=======
    # Incoming branch version
    total = sum(item.price * item.quantity for item in items)
    return total
>>>>>>> feature-cart
```

Conflict markers:

- `<<<<<<< HEAD`: Your current branch's version
- `=======`: Separator
- `>>>>>>> feature-cart`: Incoming branch's version

Resolving Conflicts Step-by-Step

"The standard approach involves examining both versions, understanding their intent, and creating code that incorporates both changes appropriately." — Atlassian

```
# Step 1: Attempt the merge
git merge feature-cart
# Git reports: CONFLICT (content): Merge conflict in cart.py

# Step 2: Open conflicted file and edit it
# Remove conflict markers and choose correct code

# Step 3: Stage the resolved file
git add cart.py

# Step 4: Complete the merge with a commit
git commit -m "Merge feature-cart, combine quantity and tax"
```

Conflict Resolution Example

Before (Conflicted):

```
<<<<<<< HEAD
total = sum(item.price for item in items)
return total * 1.1  # Add 10% tax
=======
total = sum(item.price * item.quantity for item in items)
return total
>>>>>>> feature-cart
```

After (Resolved):

```
# Calculate total with quantities and add tax
total = sum(item.price * item.quantity for item in items)
return total * 1.1  # Add 10% tax
```

Key: Understand what both versions tried to accomplish, then create code that honors both intentions.

📤 Core Concept 4: Pushing

Pushing uploads your local commits to a remote repository.

```
# Push your current branch to the remote repository
git push origin feature-login

# Push and set upstream tracking (first time)
git push -u origin feature-login

# After setting upstream, simply use
git push
```

The `-u` flag (short for `--set-upstream`) sets up **upstream tracking**, which means Git remembers which remote branch corresponds to your local branch.

📥 Core Concept 5: Pulling

Pulling downloads changes from remote and merges them.

```
# Pull changes from the remote main branch
git checkout main
git pull origin main

# Or simply (if tracking is set up)
git pull
```

Pull = Fetch + Merge:

- **Fetch:** Download changes from remote
- **Merge:** Integrate changes into current branch

Best Practice: Always pull before starting new work to minimize conflicts!

The Push-Pull Cycle

A typical collaborative workflow:

```
# Morning: Get latest changes
git checkout main
git pull

# Create feature branch
git checkout -b feature-notifications

# Work and commit
git add notifications.py
git commit -m "Add email notification system"

# Push your branch
git push -u origin feature-notifications
```

The Push-Pull Cycle (continued)

```
# Later: Someone else updates main
# Update your feature branch with latest main
git checkout main
git pull
git checkout feature-notifications
git merge main

# Resolve any conflicts, then push
git push
```

This cycle keeps your work synchronized with your team while maintaining independent development spaces.

📋 Practical Example: Task Manager Project

Scenario: You're joining a team building a task management app.

Step 1: Clone the Project

```
# Clone the team's repository
git clone https://github.com/team/task-manager.git
cd task-manager

# Check the current state
git status
# Output: On branch main
#         Your branch is up to date with 'origin/main'
```

Step 2: Create Your Feature Branch

Your task: Add a priority system for tasks.

```
# Create and switch to feature branch
git checkout -b feature-task-priority

# Verify you're on the new branch
git branch
# Output:
#   main
# * feature-task-priority
```

Step 3: Implement and Commit

```python
# Edit task.py
class Task:
    def __init__(self, title, description, priority='medium'):
        self.title = title
        self.description = description
        self.priority = priority  # New feature: low, medium, high

    def is_urgent(self):
        return self.priority == 'high'
```

```bash
# Stage and commit your changes
git add task.py
git commit -m "Add priority field to Task class with urgency check"

# Make another change to the UI
git add templates/task_list.html
git commit -m "Display priority badges in task list"
```

Step 4: Push Your Branch

```
# Push your feature branch to remote
git push -u origin feature-task-priority
```

What happens:

- Your commits are uploaded to GitHub
- A new remote branch is created
- Upstream tracking is set up
- Teammates can see your work

> Your feature is now backed up and visible to the team!

Step 5: Merge and Handle Conflicts

Meanwhile: A teammate updated the main branch.

```
# Switch to main and update
git checkout main
git pull origin main

# Switch back and merge main into your feature
git checkout feature-task-priority
git merge main

# Conflict! Your teammate also modified task.py
# CONFLICT (content): Merge conflict in task.py
```

Resolving the Conflict

Conflicted file:

```
class Task:
    def __init__(self, title, description, priority='medium'):
        self.title = title
        self.description = description
<<<<<<< HEAD
        self.priority = priority  # Your version
=======
        self.due_date = due_date  # Teammate's version
>>>>>>> main
```

Resolved file:

```
class Task:
    def __init__(self, title, description, priority='medium', due_date=None):
        self.title = title
        self.description = description
        self.priority = priority  # Your feature
        self.due_date = due_date  # Teammate's feature
```

Complete the Merge

```
# Mark as resolved and complete merge
git add task.py
git commit -m "Merge main into feature-task-priority, combine priority and due_date"

# Push the updated branch
git push
```

Complete cycle: clone → branch → commit → push → pull → merge → resolve conflicts

🚫 Common Pitfall 1: Forgetting to Pull

The Problem:

You start coding without pulling latest changes. When you push, Git rejects it because your branch is behind.

The Solution:

Make pulling the first thing you do each session.

```
# ✅ Good workflow
git checkout main
git pull
git checkout -b new-feature

# ❌ Not this
git checkout -b new-feature  # Might be based on old code!
```

Create a habit: `git checkout main && git pull` before creating new branches.

🚫 Common Pitfall 2: Wrong Branch

The Problem:

You make changes on main instead of a feature branch.

The Solution:

Always check your current branch. If wrong, use stash to move changes:

```
# Oh no, I'm on main!
git status  # Shows you're on main with changes

# Stash your changes temporarily
git stash

# Switch to correct branch
git checkout -b correct-feature-branch

# Apply your stashed changes
git stash pop
```

🚫 Common Pitfall 3: Panic During Conflicts

The Problem:

You see conflict markers and abandon the merge or randomly delete code.

The Solution:

Conflicts are normal and fixable. Follow the process:

1. Read both versions carefully
2. Understand what each tried to accomplish
3. Edit the file to combine both intentions
4. Remove all conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`)
5. Test that your code works
6. Stage and commit

If you mess up, abort the merge: `git merge --abort`

🚫 Common Pitfall 4: Vague Commit Messages

The Problem:

Messages like "fixed stuff" make history useless.

The Solution:

Write clear, descriptive messages.

❌ Bad

```
git commit -m "changes"
git commit -m "stuff"
git commit -m "updates"
```

✅ Good

```
git commit -m "Fix login validation
to prevent empty username"
git commit -m "Add priority field
to Task class"
```

Format: Start with verb, be specific, explain why if not obvious.

🚫 Common Pitfall 5: Not Testing After Conflicts

The Problem:

You resolve conflicts but don't test, potentially introducing bugs.

The Solution:

Always test after resolving conflicts!

```
# After resolving conflicts
git add resolved-file.py
git commit -m "Merge and resolve conflicts"

# Before pushing, TEST!
python -m pytest tests/
# or
python resolved-file.py
```

Remember: Merged code might be syntactically correct but logically broken.

📊 Comparison: Git vs. SVN

Subversion (SVN)

- Centralized model
- History on central server
- Requires network access
- Simpler permissions
- Less flexible branching

Git

- Distributed model
- Full history locally
- Work offline
- More complex permissions
- Flexible branching

Use Git when: You need flexible branching, offline work, or distributed collaboration.

📊 Comparison: Git vs. Mercurial

Mercurial

- Distributed VCS
- Easier to learn
- More consistent commands
- Smaller ecosystem
- Less industry adoption

Git

- Distributed VCS
- Steeper learning curve
- Inconsistent commands
- Massive ecosystem
- Industry standard (93%)

Use Git when: You want maximum ecosystem support and industry standard skills.

📊 Comparison: Git vs. GitHub

Common Confusion: Git ≠ GitHub

Git (the tool):

- Version control system
- Tracks changes locally
- Runs on your computer
- Open source software

GitHub (the platform):

- Git hosting service
- Provides remote repositories
- Adds collaboration features
- Pull requests, issues, CI/CD

🎯 Key Takeaways

- **Version control systems track changes over time**, enabling collaboration, backup, and experimentation
- **Git is distributed**, meaning every developer has complete history locally
- **Cloning creates a complete local copy** with full history for independent work
- **Branches are parallel development spaces** that protect the main codebase
- **Merging combines work from branches**, automatically when possible

🎯 Key Takeaways (continued)

- **Merge conflicts require manual resolution** by examining both versions and creating code that honors both
- **Push uploads commits to remote**, while **pull downloads and integrates** remote changes
- **The basic Git workflow**: pull → branch → edit → commit → push → merge
- **Always pull before starting new work** to minimize conflicts
- **Good commit messages** are essential for understanding project history

📝 Practice Quiz: Question 1

What's the difference between `git clone` and `git pull`?

A) They do exactly the same thing

B) Clone is for repositories, pull is for files

C) Clone creates a new local copy (used once), pull updates existing copy (used regularly)

D) Pull is faster than clone

📝 Answer 1

Correct: C

`git clone` creates a new local copy of a remote repository (used once when starting).

`git pull` updates an existing local repository with changes from the remote (used regularly to stay synchronized).

> **Remember:** Clone once, pull often!

📝 Practice Quiz: Question 2

You're on the `main` branch and want to start a new feature. What commands do you use?

A) `git branch new-feature`

B) `git checkout -b feature-name`

C) `git pull feature-name`

D) `git merge feature-name`

📝 Answer 2

Correct: B

```
git checkout -b feature-name
```

This creates a new branch and switches to it in one command.

Alternative (two commands):

```
git branch feature-name
git checkout feature-name
```

The `-b` flag means "create a new branch".

📝 Practice Quiz: Question 3

You see this in your file after a merge:

```
<<<<<<< HEAD
code version A
=======
code version B
>>>>>>> feature-branch
```

What do you do?

A) Delete all the markers and keep nothing

B) Keep both versions with markers

C) Edit to keep correct code, remove markers, then git add and commit

D) Run git merge --abort always

📝 Answer 3

Correct: C

Steps to resolve:

1. Edit the file to keep correct code (A, B, or combination)
2. Remove all conflict markers (`<<<<<<<`, `=======`, `>>>>>>>`)
3. Save the file
4. `git add filename`
5. `git commit` to complete the merge

> **Option D** is only for when you want to give up and start over!

📝 Practice Quiz: Question 4

What's the purpose of the staging area (using `git add`)?

A) It's where Git stores remote repositories

B) It lets you prepare a logical commit by selecting which changes to include

C) It's a backup of your files

D) It's where conflicts are resolved

📝 Answer 4

Correct: B

The staging area lets you prepare a logical commit by selecting which changes to include.

Example:

```
# You edited 5 files but only want to commit 2 related changes
git add feature.py
git add feature_test.py
git commit -m "Add new feature with tests"

# Later, commit the other changes separately
git add config.py
git commit -m "Update configuration"
```

> This keeps your commit history clean and focused.

📝 Practice Quiz: Question 5

Your `git push` is rejected with "Updates were rejected because the remote contains work that you do not have locally." What should you do?

A) Force push with `git push -f`
B) Delete your local changes
C) Run `git pull` to merge remote changes, then push again
D) Create a new branch

📝 Answer 5

Correct: C

Steps:

1. `git pull` to download and merge remote changes
2. Resolve any conflicts if they occur
3. `git push` again

Never force push (`git push -f`) unless you're absolutely sure! It can overwrite teammates' work.

This ensures your push includes all the latest remote changes.

📝 Practice Quiz: Question 6

What does the `-u` flag do in `git push -u origin feature-branch`?

A) Updates all files

B) Sets up upstream tracking so you can use `git push` without arguments later

C) Creates a new branch on the remote

D) Uploads faster

📝 Answer 6

Correct: B

The `-u` flag (short for `--set-upstream`) sets up upstream tracking.

What this means:

- Git remembers the relationship between your local and remote branch
- After setting this once, you can simply use `git push` and `git pull`
- No need to specify `origin feature-branch` every time

```
# First time
git push -u origin feature-login

# After that
git push  # Git knows where to push!
```

📝 Practice Quiz: Question 7

What's the difference between Git and GitHub?

A) Git is old, GitHub is new

B) Git is the version control tool, GitHub is a hosting platform

C) GitHub is faster than Git

D) They're the same thing

📝 Answer 7

Correct: B

Git (the tool):

- Version control system software
- Tracks changes locally
- Open source

GitHub (the platform):

- Git hosting service (provides remote repositories)
- Adds collaboration features (pull requests, issues, CI/CD)
- One of many options (alternatives: GitLab, Bitbucket)

Remember: You can use Git without GitHub!

🔧 Hands-On Practice

Practice Repository:
https://github.com/datatweets/git-collaboration-practice

Structured Exercises:

- **Exercise 1:** Fork, clone, make your first contribution
- **Exercise 2:** Create feature branch, add priority system
- **Exercise 3:** Find and fix intentional bugs
- **Exercise 4:** Create and resolve merge conflicts
- **Exercise 5:** Review classmates' pull requests

> Each exercise builds on the previous, teaching complete Git workflow!

Getting Started with Practice

Steps:

1. Visit: https://github.com/datatweets/git-collaboration-practice
2. Click "Fork" button to create your copy
3. Clone your fork to local machine
4. Follow instructions in README.md
5. Start with Exercise 1 and progress through all five

By completing these exercises, you'll gain practical experience with all Git operations: cloning, branching, committing, merging, pushing, pulling, and resolving conflicts.

Essential Git Commands Summary

```
# Clone a repository
git clone https://github.com/user/repo.git

# Create and switch to new branch
git checkout -b feature-name

# Stage and commit changes
git add .
git commit -m "Descriptive message"

# Push to remote
git push -u origin feature-name

# Pull latest changes
git pull

# Merge branch
git merge feature-name
```

Git Workflow Cheat Sheet

Daily workflow:

1. **Morning:** `git checkout main && git pull`
2. **Start feature:** `git checkout -b feature-name`
3. **Make changes:** Edit files
4. **Commit:** `git add . && git commit -m "message"`
5. **Push:** `git push -u origin feature-name`
6. **Update:** `git pull origin main` (merge main into feature)
7. **Complete:** Open pull request on GitHub

> This workflow keeps you synchronized with your team!

📚 Additional Resources

Official Documentation:

- Git Documentation: https://git-scm.com/doc
- GitHub Guides: https://guides.github.com/

Interactive Learning:

- Learn Git Branching: https://learngitbranching.js.org/
- Git Immersion: http://gitimmersion.com/

Reference:

- Atlassian Git Tutorials: https://www.atlassian.com/git/tutorials
- Oh Sh*t, Git!?: https://ohshitgit.com/

Quick Reference Card

Command	Purpose
`git clone <url>`	Create local copy
`git status`	Check current state
`git branch`	List branches
`git checkout -b <name>`	Create & switch branch
`git add <file>`	Stage changes
`git commit -m "msg"`	Save snapshot
`git push`	Upload to remote
`git pull`	Download from remote
`git merge <branch>`	Combine branches

Final Thoughts

Git is the foundation of modern collaborative development.

What you've learned:

- The mental model of distributed version control
- Essential Git operations (clone, branch, commit, merge, push, pull)
- How to resolve merge conflicts systematically
- Common pitfalls and how to avoid them
- The complete collaborative workflow

Next steps:

- Complete the hands-on exercises
- Start using Git for your projects
- Practice with teammates

🎉 Lesson Complete!

You now understand:

- ✅ What version control is and why Git dominates
- ✅ How to clone repositories and work independently
- ✅ How to create branches and merge work
- ✅ How to push and pull changes with teams
- ✅ How to resolve merge conflicts confidently

> **Practice makes perfect!** The more you use Git, the more natural it becomes. Start with the hands-on exercises and apply these skills to every project.

Ready to collaborate? Fork the practice repository and begin!