ENPM702

Version Control

v2.1

Lecturer: Z. Kootbally

Semester/Year: Summer/2025

MARYLAND APPLIED
GRADUATE ENGINEERING

Table of Contents

- Version Control
  - Types of Version Control Systems
    - Centralized Version Control
    - Distributed Version Control
- Git
  - Example
    - Setting Up the Project
    - Adding GPS Navigation
    - Bug Alert - Emergency Fix Needed
    - Handling Merge Conflicts
    - Completing the Feature
    - End of Day Review
- GitHub
  - Connect Local Repo to Remote.
  - Collaboration Workflows
    - Branch Workflow
    - Fork Workflow
  - Pull Requests

Changelog

- **v2.1**: Added a section on git pointers.
- **v2.0**: Updated examples and files used in examples.
- **v1.0**: Original version.

Learning Objectives

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

- Set up and configure Git for any project.
- Create repositories and track changes effectively.
- Use branching for feature development and hotfixes.
- Collaborate using GitHub and pull requests.
- Apply version control best practices to team projects.

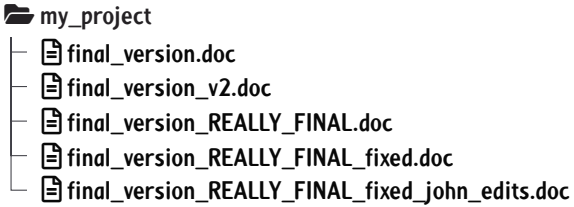
Version Control

📜 Version Control

Version control is a system that tracks changes to files over time, allowing you to:

- **Track History**: See exactly what changed, when, and who made the change.
- **Revert Changes**: Go back to any previous version of your files.
- **Branch and Merge**: Work on different features simultaneously without conflicts.
- **Collaborate**: Multiple people can work on the same project without overwriting each other's work.
- **Backup**: Distributed copies serve as automatic backups.

Without Version Control



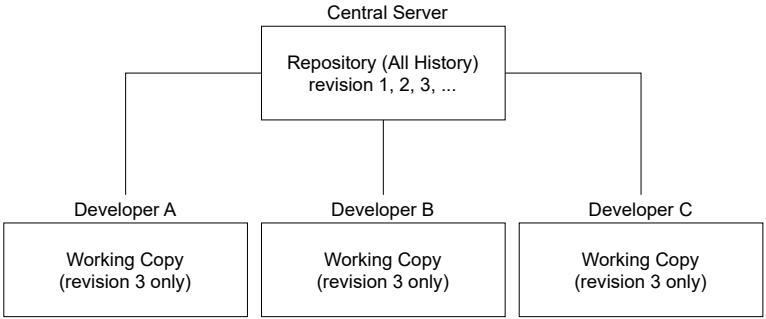
With Version Control

- **document.doc** has complete history.
- **.git/** tracks all versions automatically.

Types of Version Control Systems

1. Local Version Control:
   - Copies files to different directories.
   - Simple but error-prone.
   - **Example**: RCS (Revision Control System)
2. Centralized Version Control:
   - Single server contains all versions.
   - Clients check out files from central place.
   - **Examples**: CVS, Subversion (SVN), Perforce.
3. Distributed Version Control:
   - Every client has complete repository copy.
   - No single point of failure.
   - **Examples**: Git, Mercurial, Bazaar.

Centralized Version Control



- **Single source of truth**: Central server has master repository.
- **Client-server model**: Developers have working copies only.
- **Network dependent**: Most operations require server connection.

Typical Workflow

```
# SVN Example
svn checkout https://server.com/repo/trunk # Get working copy
svn update # Get latest changes
# Make changes...
svn commit -m "My changes" # Save to server
svn log # View history (needs network)
```

Advantages & Disadvantages

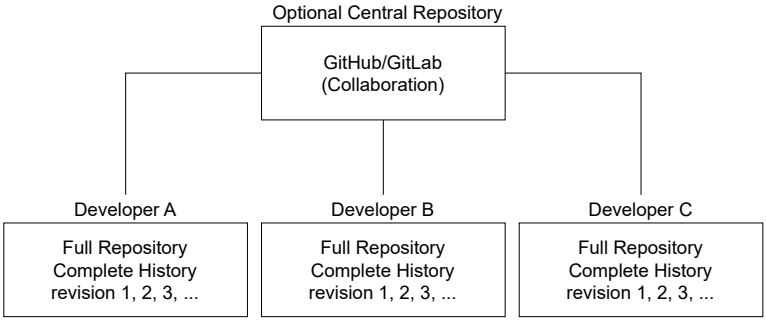
Advantages

- Simple mental model - one central authority
- Fine-grained access control
- Storage efficient for clients
- Easy administration and compliance

Disadvantages

- Single point of failure
- Network required for most operations
- Limited offline capabilities
- Expensive branching operations

Distributed Version Control



- **Complete local repositories**: Full history everywhere.
- **Peer-to-peer model**: Repositories sync with any other.
- **Offline capable**: All operations work without network.

Typical Workflow

```
# Git Example
git clone https://github.com/user/project.git   # Get complete repo
git log                                  # View history (offline)
git branch feature                       # Create branch (instant)
git checkout feature                     # Switch branch (instant)
# Make changes...
git commit -m "My changes"                     # Save locally
git push origin feature                  # Share when ready
```

Advantages & Disadvantages

Advantages

- No single point of failure
- Fast local operations
- Excellent offline capabilities
- Flexible workflows
- Cheap branching and merging

Disadvantages

- Steeper learning curve
- More complex concepts
- Larger local storage requirements
- Less granular access control

Git

Git

Git is a version control system that tracks changes in your files over time.

- Save snapshots of your project at different points.
- See what changed between versions.
- Collaborate with others without overwriting each other's work.
- Revert to previous versions if something breaks.

- **Installation:** `sudo apt update  sudo apt install git`
- **Configure Git:**
  - `git config --global user.name "Your Full Name"`
  - `git config --global user.email "your.email@university.edu"`
- **Check:**
  - `git config --list`

Why Git dominates.

Technical Advantages

- **Speed** - Operations like commits and diffs execute in milliseconds.
- **Distributed Nature** - Every developer has a complete copy of the project history.
- **Branching Model** - Creating and merging branches is lightweight and fast.
- **Data Integrity** - SHA-1 checksums ensure your code history cannot be corrupted.

Ecosystem Advantages

- **GitHub Integration** - Seamless hosting with powerful collaboration features.
- **Tool Support** - Every major IDE and editor has excellent Git integration.
- **Industry Adoption** - Used by virtually all major tech companies and open source projects.

Daily Commands

```
git status              # Check status
git add .               # Stage changes
git commit -m "msg"     # Commit changes
git push                # Upload to GitHub
git pull                # Download updates
```

Branching Commands

```
git branch          # List branches
git checkout -b new # Create & switch
git merge branch    # Merge branch
git branch -d old   # Delete branch
```

Multiple Git cheat sheets are available on Canvas.

Example: A Day in the Life of a Robotics Engineer.

To illustrate how these concepts work in practice, we will walk through a realistic, day-long scenario. You will take on the role of a robotics engineer at ENPM702Tech Labs.
Throughout the day, you will use Git to:

- Initialize a new project for an autonomous robot's configuration.
- Use branching to develop a new feature (GPS navigation).
- Handle a critical hotfix for a performance bug.
- Navigate and resolve a merge conflict that arises from these parallel lines of work.

This hands-on example will demonstrate the complete lifecycle of branching, merging, and conflict resolution in a typical engineering workflow.

Monday Morning - Setting Up the Project

You begin your week by creating the baseline configuration for a new autonomous delivery robot platform.

1. **Create a new project directory.**

```
# Create a new project directory
mkdir robot-config #note kebab-case
```

2. **Initialize version control.**

```
cd robot-config
git init
```

`git init` creates a new Git repository in your current directory.

- Creates the .git directory: This hidden folder contains all of Git's internal files and metadata for the repository, including the object database, configuration files, and references.
- Initializes an empty repository: The repository starts with no commits, branches, or tracked files. It is essentially a blank slate ready for you to start adding content.
- Sets up the default branch: Modern Git versions create a default branch.

What is the default branch?

Think of the default branch as the official, stable version of your project, like the master copy of a document that everyone refers to. The default branch is usually called master or main.

main vs. master

GitHub and many organizations switched from master to main for inclusive language reasons.

- This change happened because `master` has historical associations with slavery and the `master/slave` terminology used in various technologies. The tech industry began reconsidering this language as part of broader efforts to make computing more inclusive and welcoming.
- Git itself updated to allow configurable default branch names.

ToDo

Rename master to main

Understanding Git's Foundation: Commits and Pointers

Before diving into our robotics example, let's understand how Git actually works under the hood.

- **Commit**: A snapshot of your entire project at a specific moment in time.
- **Branch**: A lightweight, movable pointer to a specific commit.
- **HEAD**: A special pointer that shows you where you currently are.

Visual Analogy

Think of commits as photographs in a photo album, branches as bookmarks, and `HEAD` as your current page.

What is a Commit?

A commit is like taking a complete snapshot of your project folder at a specific moment.

- Contains all your files exactly as they were when you committed.
- Has a unique identifier (SHA hash): `a1b2c3d4e5f6...`
- Includes metadata: author, timestamp, commit message.
- Points to its parent commit(s), creating a chain of history.

```
# Each commit creates a permanent snapshot
git commit -m "Add GPS sensor configuration"
# Creates commit: f7a8b9c with complete project state
```

Key Insight

Git doesn't store differences between versions. It stores complete snapshots efficiently using content addressing.

HEAD Pointer: Your Current Location

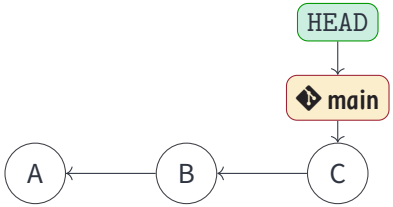
HEAD tells Git “this is where you are right now” in your project’s history.

```
# See where HEAD is pointing
git log --oneline --graph
# * c3d4e5f (HEAD -> main) Latest commit
# * a1b2c3d Previous commit

# HEAD typically points to the tip of your current branch
git branch
# * main  <- HEAD is here
```

- HEAD usually points to a branch name (like **main**).
- That branch points to the latest commit.
- When you make a new commit, the branch pointer moves forward.
- HEAD moves with your current branch.

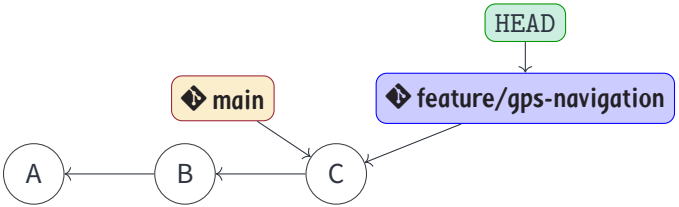
Git Pointers: Initial State



- **main** branch points to commit C (the latest).
- `HEAD` points to **main** branch.
- You are “on” the **main** branch.
- Each commit points to its parent, forming a chain of history.

Creating a Branch: Just a New Pointer

```
git checkout -b feature/gps-navigation
```

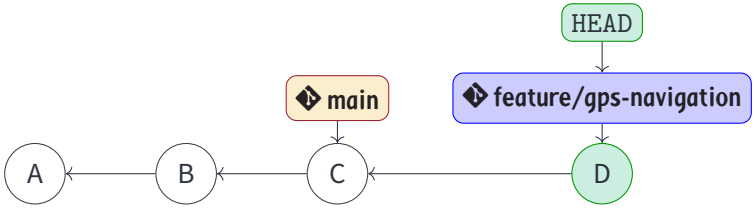


ToDo

- Git creates a new pointer called feature/gps-navigation.
- New pointer points to the same commit as current branch (C)
- HEAD switches to point to the new branch.
- **No files are copied!** Git just creates a lightweight pointer.

Making a Commit: Branch Pointer Moves Forward

After you make changes and commit on the feature branch:



ToDo

- Git creates new commit D with your changes.
- Current branch pointer (**feature/gps-navigation**) moves forward to D
- `HEAD` stays with current branch.
- Other branches (**main**) remain unchanged at C.
- Each commit knows its parent, preserving history.

Why Understanding Pointers Matters

Understanding that branches are just lightweight pointers explains:

- Why creating branches is instant (no file copying)
- How Git can switch between branches so quickly
- Why you can have hundreds of branches without bloating your repository
- How merging actually works (moving pointers and creating new commits)
- Why Git is so efficient compared to older version control systems

Ready for the Example

Now that you understand commits and pointers, let's see them in action with our robotics project!

3. Create the files you need to track in the current folder.
   - **robot_config.yaml** – Configuration file for autonomous delivery robot.
   - **sensitive_config.yaml** – Contains sensitive information that should not be tracked by Git.
   - **README.md** – File that serves as the front door, or the initial point of information, for anyone encountering a project for the first time. The `.md` extension signifies that the file is written in Markdown, a lightweight markup language that allows for easy formatting and readability.
   - **.gitignore** – File tells Git which files and folders to ignore when tracking changes. This prevents unnecessary or sensitive files from being committed to your repository.

git status

```
On branch main

No commits yet

Untracked files:
    (use "git add <file>..." to include in what will be committed)
        .gitignore
        README.md
        robot_config.yaml
        sensitive_config.yaml

nothing added to commit but untracked files present (use "git add" to track)
```

Why Use .gitignore?

Avoid tracking files that:

- Are generated automatically (build artifacts, compiled code).
- Contain sensitive information (passwords, API keys).
- Are user-specific (IDE settings, OS files).
- Are too large or binary (datasets, videos, executables).
- Change frequently but aren't important (log files, cache).

Common .gitignore Patterns

```
# Compiled code and build artifacts
*.o
*.so
*.exe
build/
dist/
target/

# IDE and editor files
.vscode/

# Operating system files
.DS_Store       # macOS
Thumbs.db       # Windows
*.tmp

# Configuration with secrets
robot_secrets.yaml
wifi_passwords.txt
```

- **Create early**: Add .gitignore before making your first commit.
- **Already tracked files**: .gitignore won't affect files already being tracked.
- **Remove tracked files**: Use `git rm --cached filename` to stop tracking.

```
# Remove file from tracking but keep locally
git rm --cached sensitive_config.yaml
```

ToDo

Add sensitive_config.yaml and .vscode to .gitignore

4. **Stage the files.**

```
git add .
```

Staging puts your changes in a **draft box** called the **staging area**. You are preparing what you want to include in your next save point, but you have not saved it yet.

- **Analogy**: Drafting an email.
  - Your email is ready to be sent but not sent yet.

5. **Commit changes.**

```
git commit -m "Initial commit: Add basic robot configuration

- Add robot_config.yaml with hardware and navigation settings
- Add README.md with project description
- Add .gitignore"
```

Committing takes everything from your staging area (draft box) and creates a permanent save point in your project's history.

- **Analogy**: Sending an email.
  - The email is now permanently sent.
  - It becomes part of your email history.

6. **Verify the commit.**

```
git log
```

You can pass other options to `git log`. Run `git log --help` to get more information.

Git Branch in Visual Studio Code

Multiple extensions provide an overview of branches, e.g., Git Graph

Open the folder `robot-config` in VS Code and use the Git Graph extension to visualize branches and commits.

Feature Development - Adding GPS Navigation

10:00 AM - You receive a task to add GPS navigation capabilities to the robot.

1. **Create a feature branch for GPS navigation.**

```
git checkout -b feature/gps-navigation
```

- **Part 1:** `git checkout -b`
  - `git checkout` = "Switch to a branch"
  - `-b` = "But first, create a new branch from wherever you are right now"
  - **Combined:** "Create a new branch AND switch to it immediately"
- **Part 2:** `feature/gps-navigation`
  - This is the name of the new branch.
  - `feature/` is a naming convention (like organizing folders).
  - `gps-navigation` describes what this branch is for.
- **Long way (two separate commands)**

```
git branch feature/gps-navigation # Create the branch
git checkout feature/gps-navigation # Switch to it
```

When you use **git checkout -b <new-branch-name>**, the new branch is created from the current branch you are currently on. For instance, to create **feature/gps-navigation** from **main**, you can use:

```
git checkout main # Switch to main if you are not already on main
git checkout -b feature/gps-navigation # Create the new branch and switch to it
```

Alternative: **git checkout -b <new-branch-name> <existing-branch-name>**

```
git checkout -b feature/gps-navigation main
```

2. **Check you are on the current branch:** As a sanity check, verify you are on the correct branch before doing any work.

```
git branch

# output
* feature/gps-navigation
  main
```

3. **Modify robot_config.yaml for GPS Feature**
   - Implement GPS feature in **robot_config.yaml**.
   - Uncommon the following lines.

```yaml
gps:
  enabled: true
  provider: "u-blox ZED-F9P"
  accuracy: "high"
  datum: "WGS84"
```

   - Fix the typo in the comment.

```yaml
navigation:
  # General navigation settings for the robot
  update_rate: 5 # HZ
  max_speed: 1.5   # meters/sec
  ...
```

4. **Check what changed.**

```
git diff
```

```
diff --git a/robot_config.yaml
↪  b/robot_config.yaml
index c055b1f..a7ca272 100644
--- a/robot_config.yaml
+++ b/robot_config.yaml
@@ -38,11 +38,11 @@ power_management:
```

git diff shows changes between your working directory (current files) and the staging area (what's ready to commit).

- `diff --git a/robot_config.yaml b/robot_config.yaml` - Shows this is a diff between two versions of the same file: **robot_config.yaml**
- `index c055b1f..a7ca272 100644` - Internal Git object hashes.
  - `c055b1f` – Git hash of the old version (before changes)
  - `a7ca272` – Git hash of the new version (after changes)
  - `100644` – File permissions (standard file, readable/writable by owner)
- `--- a/robot_config.yaml` - Old version (before changes).
- `+++ b/robot_config.yaml` - Modified version (after changes).
- `a/` and `b/` are just Git's way of labeling old vs new.
- `@@ -38,11 +38,11 @@` - This is called a "hunk header" and tells you:
  - `-38,11` = In the old file, starting at line 38, showing 11 lines of context.
  - `+38,11` = In the new file, starting at line 38, showing 7 lines of context.
  - `power_management:` = Shows some context of what section this change is in.

5. Stage and commit the changes.

```
git add robot_config.yaml

git commit -m "Feature: Add GPS navigation capabilities to the robot"

# Check commit history or use Git Graph
git log --oneline --all --graph
```

Emergency Fix Needed

11:30 AM - You get an urgent message: the robot's overall responsiveness is sluggish! The default navigation `update_rate` of 5Hz is too slow for real-time obstacle avoidance.

This is a critical issue. You must fix this on main immediately.

1. Let's see what we are working on. **git status**

```
# On branch feature/gps-navigation
# nothing to commit, working tree clean
```

   - Your GPS work is safe on its own branch. You can switch away without losing anything.

2. Create and switch to the new branch. The fix needs to be based on the official **main** branch, not your **feature/gps-navigation** branch.

```
git checkout main # switch to main
git checkout -b hotfix/fix-navigation-rate
```

   - or, more concisely: **git checkout -b hotfix/fix-navigation-rate main**

3. Increase navigation `update_rate` to 10Hz. You change the `update_rate` in **robot_config.yaml** from 5Hz to 10Hz to improve the robot's reaction time.

```yaml
navigation:
  # General navigation settings for the robot
  update_rate: 10 # Hz
  ...
```

4. Check the changes: **git diff**

5. Stage and commit.

```
git add robot_config.yaml
git commit -m "HOTFIX: Increase navigation update rate to 10Hz

- Critical performance fix to improve robot responsiveness."
```

6. Merge hotfix back to **main**.

```
git checkout main # switch to main
git merge hotfix/fix-navigation-rate
```

   6.1 Go to the **main** version of the project.
   6.2 Apply all the changes made on the **hotfix/fix-navigation-rate** branch into this **main** branch.

7. Clean up the hotfix branch: `git branch -d hotfix/fix-navigation-rate`
8. Check current state: `git log --oneline --all --graph`

Handling Merge Conflicts

12:30 PM - After the hotfix is deployed, you return to your GPS feature. To stay up-to-date, you must merge the changes from `main` into your feature branch.

1. Switch back to the GPS feature branch: `git checkout feature/gps-navigation`
2. Attempt the merge: `git merge main`

```
# Auto-merging robot_config.yaml
# CONFLICT (content): Merge conflict in robot_config.yaml
# Automatic merge failed; fix conflicts and then commit the result.
```

3. Check the conflict status: `git status`

```
# On branch feature/gps-navigation
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#   (use "git merge --abort" to abort the merge)
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#   both modified:     robot_config.yaml
```

We have a Merge Conflict because both branches modified the same line in robot_config.yaml. Instead of guessing, Git pauses the merge and inserts conflict markers into the file, asking you to resolve the situation manually.

```
<<<<<<< HEAD
  update_rate: 5 # Hz
=======
  update_rate: 10 # Hz
>>>>>>> main
```

- `<<<<<<< HEAD`: This marks the beginning of the change from your current branch ('feature/gps-navigation').
- `=======`: This separates the two conflicting changes.
- `>>>>>>> main`: This marks the end of the change from the branch you are merging in ('main').

4. Open robot_config.yaml in your editor to see the conflict markers.
5. Edit the file to contain only the final, correct code and remove all conflict markers.
6. Stage the file: `git add robot_config.yaml`
7. Complete the merge with a descriptive commit message

```
git commit -m "Merge main into feature/gps-navigation

- Resolved navigation update_rate conflict"
```

8. Verify the merge history: `git log --oneline --graph` or use Git Graph.

Completing the Feature

2:00 PM - You add final touches to the GPS feature.

- ☐ Add waypoint information to define how the robot handles mission destinations based on GPS coordinates.
- ☐ Update `README.md` file.

1. Add GPS waypoint configuration in **robot_config.yaml**. Uncomment lines 67 – 79
2. Update **README.md**. Uncomment line 14
3. Review all changes: `git status`
4. Stage: `git add .`
5. Commit:

```
git commit -m "Complete GPS navigation feature

- Add waypoint navigation configuration
- Set home coordinates for auto-return functionality
- Update README with GPS feature documentation
- Support up to 50 waypoints with 1-meter precision"
```

6. The feature is complete, merge back to **main**

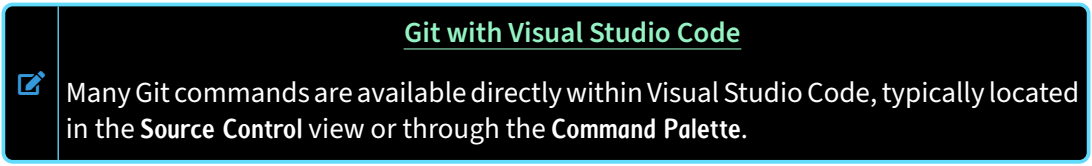
```
git checkout main
git merge feature/gps-navigation
```

7. Clean up feature branch: `git branch -d feature/gps-navigation`

End of Day Review

5:00 PM - You review your day's work.

1. View complete project history: `git log --oneline --graph --all`
2. Check final file status: `git status`
3. View the complete configuration file: `cat robot_config.yaml`
4. Create a summary of what was accomplished: `git log --oneline --since="1 day ago"`
5. View detailed changes for the day: `git diff a1b2c3d..HEAD --stat`


Git with Visual Studio Code
Many Git commands are available directly within Visual Studio Code, typically located in the Source Control view or through the Command Palette.

GitHub

GitHub

GitHub is a cloud-based platform that hosts Git repositories online. It adds features like:

- Remote storage for your projects.
- Collaboration tools.
- Issue tracking.
- Project management features.
- Portfolio showcase for your work.

LIVE DEMO: GitHub Integration

Time to connect our local repository to GitHub!

ToDo

- Create a GitHub account (only if you don't have one already).
- Set up Authentication.
- Connect local repo to remote.

1. Create a GitHub account.
   - Go to github.com
   - Sign up with your university email.
   - Verify your email address.
2. Set up authentication.
3. Create a public GitHub repository.

Public Repository

- Anyone can view
- Anyone can clone/download
- Shows up in search engines
- Others can star/follow
- Completely open source

vs.

Private Repository

- Only you can view
- Only invited collaborators can access
- Hidden from search engines
- Cannot be cloned by public
- Perfect for private work

Connecting Local Git Repository to GitHub

Scenario #1 - You already have a local repository.

1. Create an empty repository on GitHub: **robot-config**
   - **IMPORTANT**: When creating on GitHub, **DON'T check**:
     - ✗ "Add a README file"
     - ✗ "Add .gitignore"
     - ✗ "Choose a license"
2. Connect local repository to GitHub:
   - Add GitHub as remote origin

     ```
     git remote add origin git@github.com:yourusername/robot-config.git
     ```
   - Verify the remote was added: `git remote -v`
3. Push your local **main** branch to GitHub: `git push -u origin main`
   - This command pushes your local **main** branch to the remote repository named **origin**, and establishes a tracking relationship so that in the future you can simply use:

     `git push`

Connecting Local Git Repository to GitHub

Scenario #2 - Start with GitHub repository first.

1. Create an empty repository on GitHub: **new-project**
   - **IMPORTANT**: When creating on GitHub, check the following since you are starting fresh:
     - ✅ "Add a README file"
     - ✅ "Add .gitignore" (optional)
     - ✅ "Choose a license" (optional)
2. Clone to your local machine: `git clone git@github.com:yourusername/new-project.git`

```
# Move into the directory
cd new-project

# Start working
 echo "# My New Project" >> README.md
git add README.md
git commit -m "Update README"
git push origin main
```

Branch vs Fork: Collaboration Strategies

When working with teams, you need to choose between **branching** and **forking** workflows. The choice depends on your team structure and project permissions.

- **Branch Workflow (Private Repo)**
  - **Use when: You have write access** to the repository (team member, collaborator)
    - Direct access to main repository.
    - Simpler workflow for team members.
    - Better for small to medium teams.
    - Easier to manage releases.
- **Fork Workflow (Public Repo)**
  - **Use when: You don't have write access** (open source contributor, external collaborator)
    - Works without write permissions.
    - Safe for open source projects.
    - Each contributor has complete copy.
    - Maintainers control what gets merged.

Branch Workflow

1. Clone the main repository.

```
git clone git@github.com:zeidk/branch-workflow-demo.git
cd branch-workflow-demo
```

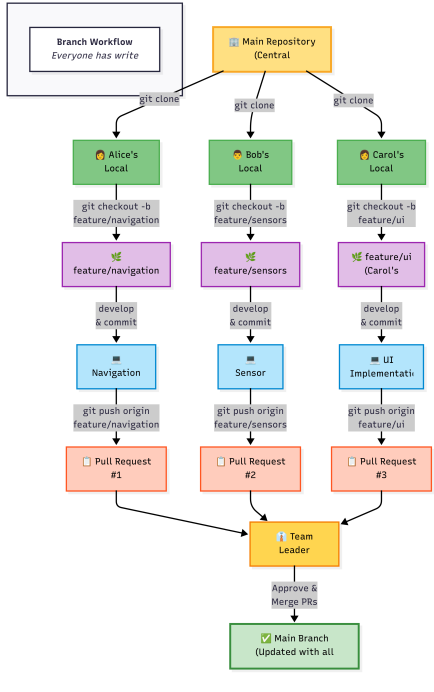
2. Create feature branch.

```
git checkout -b feature/sensor-integration
```

3. Work on your changes.
4. Commit and push to shared repository.

```
git add .
git commit -m "Add ultrasonic sensor integration"
git push origin feature/sensor-integration
```

5. Create Pull Request on GitHub: After review and approval, merge to main.


Branch Workflow
Everyone has write
🏢 Main Repository (Central
git clone
git clone
git clone
👩 Alice's Local
👨 Bob's Local
👩 Carol's Local
git checkout -b feature/navigation
git checkout -b feature/sensors
git checkout -b feature/ui
🌿 feature/navigation
🌿 feature/sensors
🌿 feature/ui (Carol's
develop & commit
develop & commit
develop & commit
💻 Navigation
💻 Sensor
💻 UI Implementati
git push origin feature/navigation
git push origin feature/sensors
git push origin feature/ui
📋 Pull Request #1
📋 Pull Request #2
📋 Pull Request #3
👔 Team Leader
Approve & Merge PRs
✅ Main Branch (Updated with all

Repository Structure

```
main repository: team/robot-project
├─ main branch
├─ feature/sensor-integration (your branch)
├─ feature/camera-module (teammate's branch)
└─ hotfix/battery-issue (another branch)
```

Fork Workflow (External Contributors)

1. Fork this repository on GitHub (creates your copy)
   Original: zeidk/enpm702-summer-2025.git → **Your fork**: yourusername/enpm702-summer-2025.git
2. Clone YOUR fork.

```
git clone git@github.com:yourusername/enpm702-summer-2025.git
cd enpm702-summer-2025
```

3. Add original repository as upstream.

```
git remote add upstream https://github.com/zeidk/enpm702-summer-2025.git
git remote -v
# origin    git@github.com:yourusername/enpm702-summer-2025.git (your fork)
# upstream  https://github.com/zeidk/enpm702-summer-2025.git (original)
```

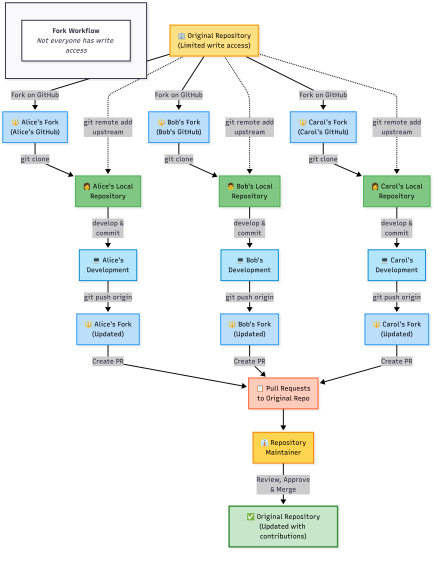
4. Create feature branch: `git checkout -b feature/new-algorithm`
5. Work on your changes.
6. Commit and push to YOUR fork.

```
git add .
git commit -m "Implement new pathfinding algorithm"
git push origin feature/new-algorithm
```

7. Create Pull Request from your fork to original repository.

Keeping Your Fork Updated

1. Fetch latest changes from original repository: `git fetch upstream`
2. Switch to main branch: `git checkout main`
3. Merge the latest changes from the upstream repository into your local repository: `git merge upstream/main`
4. Push updates to YOUR fork: `git push origin main`
5. Now create new feature branches from updated main: e.g., `git checkout -b feature/next-feature`


Fork Workflow
Not everyone has write access
Original Repository (Limited write access)
Fork on GitHub
Fork on GitHub
Fork on GitHub
Alice's Fork (Alice's GitHub)
git remote add upstream
Bob's Fork (Bob's GitHub)
git remote add upstream
Carol's Fork (Carol's GitHub)
git remote add upstream
git clone
git clone
git clone
Alice's Local Repository
Bob's Local Repository
Carol's Local Repository
develop & commit
develop & commit
develop & commit
Alice's Development
Bob's Development
Carol's Development
git push origin
git push origin
git push origin
Alice's Fork (Updated)
Bob's Fork (Updated)
Carol's Fork (Updated)
Create PR
Create PR
Create PR
Pull Requests to Original Repo
Repository Maintainer
Review, Approve & Merge
Original Repository (Updated with contributions)

Repository Structure

Original: `zeidk/enpm702-summer-2025 (upstream)`

└ **main** `branch`

Your Fork: `yourusername/enpm702-summer-2025 (origin)`

├ **main** `branch (synced with upstream)`

└ **feature/new-algorithm** `(your work)`

When to Use Each Approach

Scenario	Recommended Approach
Team member with repository access	Branch Workflow
Contributing to open source project	Fork Workflow
External contractor/collaborator	Fork Workflow
Company internal project	Branch Workflow
Public project accepting contributions	Fork Workflow
Small team (< 10 people)	Branch Workflow
Large community project	Fork Workflow

- Clear branch naming: feature/name, bugfix/name, hotfix/name
- Regular commits: Small, focused commits with clear messages.
- Pull requests: Always use pull requests for code review.
- Stay updated: Regularly sync with main branch/upstream.
- Test before merging: Ensure changes don't break existing functionality.

ToDo

- Practice both workflows with your team.
- Set up proper branch protection rules.
- Establish team conventions for branch naming.
- Configure automated testing for pull requests.

Pull Requests

A Pull Request (PR) is a method of submitting contributions to a project. It allows you to tell others about changes you have pushed to a branch in a repository.

Why Use Pull Requests?

- **Code Review**: Team members can review changes before merging.
- **Discussion**: Collaborate and discuss proposed changes.
- **Testing**: Automated tests run on proposed changes.
- **Quality Control**: Maintain code standards and catch bugs.
- **Documentation**: Track what changes were made and why.

Basic Workflow

```
git checkout -b feature/add-lidar-support
# Make changes, add, commit
git push origin feature/add-lidar-support
# Create PR on GitHub  Review  Merge  Delete branch
```

Creating a Good Pull Request

```
Title: Add LiDAR sensor integration for obstacle detection

## What this PR does
- Adds support for Velodyne VLP-16 LiDAR sensor
- Implements point cloud processing for obstacle detection
- Updates robot configuration with LiDAR parameters

## Testing
- [x] Unit tests pass
- [x] Integration tests with physical sensor

## Related Issues
Fixes #123: Robot needs better obstacle detection
```

- Use clear, descriptive titles and explain WHAT and WHY.
- Include testing information and link to related issues.
- Keep PRs focused and reasonably sized (< 400 lines).
- Use “Draft PR” for work-in-progress to get early feedback.

PR Merge Strategies

Merge Commit
- Preserves branch history
- Shows when merged
- Can create “bubbles”

Squash and Merge
- Combines all commits
- Cleaner history
- ✅ **Recommended**

Rebase and Merge
- Linear history
- Preserves commits
- Advanced technique

Common Mistakes to Avoid

- Mixing unrelated changes in one PR.
- Vague descriptions or poor commit messages.
- Not testing before submitting.
- Ignoring review feedback or force-pushing after reviews.
- Committing sensitive data (passwords, keys).

ToDo

Set up branch protection rules requiring reviews before merging.