# DEVOPS - GIT AND VERSION CONTROL SYSTEM

---

## PROBLEM 1: SHARING CODE

**Scenario:**

- Dev 1 and Dev 2 are working on the same application
- Dev 1 is writing addition functionality for calculator
- Dev 2 is writing subtraction functionality for calculator
- End of the day, we have to get both their codes and build a centralized application

**Problem:** This is a very basic application, but when we work on complex applications with thousands of lines of code and multiple developers, sharing code becomes very complex and practically impossible without a proper system.

**Challenges:**

- How to merge different developers' code?
- What if both developers modified the same file?
- How to track who made what changes?
- How to avoid code conflicts?
- How to maintain a single source of truth?

---

## PROBLEM 2: VERSIONING

**Scenario:**

**Day 1:** Dev 1 writes a function for addition of 2 numbers:

```
function add(a, b) {
  return a + b;
}
```

**Day 2:** Manager tells Dev 1 to change it to addition of 3 numbers:

```
function add(a, b, c) {
  return a + b + c;
}
```

**Day 3:** Manager tells Dev 1 to change it to addition of 4 numbers:

```
function add(a, b, c, d) {
  return a + b + c + d;
}
```

**Day 4:** Code went to QA, discussions were made, and manager says: "Go back to the code where there is addition of 2 numbers"

**Problem:** Without proper versioning, Dev 1 has to:

- Remember the old code
- Manually rewrite it
- Hope they didn't forget anything
- Risk losing work

**Solution:** We need very strong versioning to go back to that specific piece of code easily!

---

## VERSION CONTROL SYSTEMS (VCS)

**Definition:** A system that records changes to files over time so you can recall specific versions later.

**Types:**

1. Centralized Version Control System (CVCS)
2. Distributed Version Control System (DVCS)

---

## CENTRALIZED VERSION CONTROL SYSTEM (CVCS)

**Examples:** CVS, SVN (Subversion), Perforce

**Architecture:**

Central Server (SVN) is at the top

Dev 1 and Dev 2 connect to it from below

---

**HOW CVCS WORKS (Scenario 1)**

**Setup:**

- Dev 1 is writing addition functionality
- Dev 2 is writing subtraction functionality
- Both communicate with central SVN server

**Workflow:**

**Step 1: Dev 1 wants to share addition functionality with Dev 2** Dev 1 uploads code to Central SVN Server

**Step 2: Dev 2 gets the code** Dev 2 downloads code from Central SVN Server

**Note:** All communication happens through the central server!

---

**PROBLEMS WITH CVCS**

**Problem 1: Single Point of Failure**

If Central SVN Server goes down:

- Dev 1 CANNOT share code
- Dev 2 CANNOT get code
- Work STOPS completely

**Problem 2: Network Dependency**

- Need internet connection to commit changes
- Slow operations over network
- Can't work offline

**Problem 3: Limited Backup**

- Only one central copy
- If server crashes and backups fail, all history lost

**Problem 4: Collaboration Issues**

- Must always communicate through central server
- Can't directly share code between developers

---

## DISTRIBUTED VERSION CONTROL SYSTEM (DVCS)

**Examples:** Git, Mercurial

**Why Git is Very Popular?** Git is a distributed version control system which solves all problems of centralized systems!

---

## HOW DVCS WORKS (Scenario 2)

**Architecture:**

Original Repository (GitHub/GitLab) at the top

Dev 1 and Dev 2 below it

Each developer has their own Copy 1 and Copy 2

Both copies contain the full repository

---

## GIT WORKFLOW EXPLAINED

**Step 1: Dev 1 creates changes** Dev 1 makes changes, then commits locally, and has full repository history

**Step 2: Multiple sharing options**

**Option A: Share via distributed system (GitHub)** Dev 1 pushes to GitHub, then Dev 2 pulls from GitHub

**Option B: Dev 1 creates copy (Fork)** Dev 1 creates fork (Copy 1), then Dev 2 gets Copy 1

**Option C: Direct sharing** Dev 1 shares their local copy, then Dev 2 gets it

---

## KEY CONCEPT: FORK (Important Interview Question)

**What is Fork?** Fork means we are creating a copy of the entire original source code.

**Example:**

- Original repository: example.com organization repo
- Dev 1 creates fork: dev1-example.com
- All the code is now present with Dev 1

**Benefit:** If the central repository goes down, Dev 1's copy can be shared with other developers. This is the major benefit of distributed version control system.

---

**ADVANTAGES OF DISTRIBUTED VCS (Git)**

**Advantage 1: No Single Point of Failure**

If GitHub goes down:

- Dev 1 still has full copy
- Dev 2 still has full copy
- Work CONTINUES

**Advantage 2: Full History Locally**

- Every developer has complete repository history
- Can work offline
- Fast operations (no network needed)

**Advantage 3: Multiple Backup Copies**

- Every clone is a backup
- If one copy is lost, others remain

**Advantage 4: Flexible Workflows**

- Can work independently
- Merge changes when ready
- Branch and experiment safely

**Advantage 5: Better Collaboration**

- Can share directly between developers
- Pull requests for code review
- Fork and contribute to open source

---

**GIT VS GITHUB**

**Git:**

- Open source version control tool

- Command-line tool
- Installed locally on your machine
- Core functionality: version control

**GitHub:**

- Web-based platform built on top of Git
- Provides additional features
- Cloud-based repository hosting
- Not the only option (GitLab, Bitbucket also available)

---

**DETAILED COMPARISON**

**Aspect: Type**

- Git: Software tool
- GitHub: Web service

**Aspect: Installation**

- Git: Install on local machine
- GitHub: Cloud-based, no installation

**Aspect: Purpose**

- Git: Version control
- GitHub: Repository hosting + extras

**Aspect: Cost**

- Git: Free, open source
- GitHub: Free for public repos, paid for private

**Aspect: Offline**

- Git: Works offline
- GitHub: Requires internet

**Aspect: Interface**

- Git: Command line
- GitHub: Web UI + command line

**Aspect: Features**

- Git: Version control only
- GitHub: Project management, code review, CI/CD, issues, wikis

---

**ANALOGY**

**Git = Microsoft Word**

- Software you install
- Create and edit documents locally

**GitHub = Google Drive**

- Online platform
- Store documents
- Share with others
- Collaborate

---

**GIT ALTERNATIVES WITH ADDITIONAL FEATURES**

**1. GitHub**

- Most popular
- Owned by Microsoft
- Best for open source
- GitHub Actions for CI/CD

**2. GitLab**

- Self-hosted option available
- Built-in CI/CD
- Better for enterprises

**3. Bitbucket**

- Owned by Atlassian
- Integrates with Jira
- Good for enterprise teams

**All these are solutions on top of Git that help with:**

- Project management
- Code review
- Commenting
- Pull requests
- Issue tracking
- CI/CD pipelines
- Other usability features

---

## PRACTICAL GIT COMMANDS - HANDS-ON TUTORIAL

---

### STEP 1: CREATE PROJECT DIRECTORY

**Open Git Bash on Desktop**

**Command:** mkdir example.com

**Explanation:**

- mkdir: Create directory
- example.com: Your project folder name

**Verify:** ls

You should see example.com folder.

---

### STEP 2: NAVIGATE TO PROJECT

**Command:** cd example.com

**Explanation:**

- cd: Change directory
- Now you're inside example.com folder

---

### STEP 3: CREATE A FILE

**Command:** vi calculator.sh

**Inside the file, write:** x=a+b

**Save and exit:** Esc + :wq!

**Verify file creation:** ls

You should see calculator.sh

---

**STEP 4: INITIALIZE GIT REPOSITORY**

**Command:** git init

**Output:** Initialized empty Git repository in /path/to/example.com/.git/

**Explanation:**

- git init: Initialize a new Git repository
- Creates a hidden .git folder
- This folder contains all Git history and configuration

---

**STEP 5: VERIFY .git FOLDER**

**Command:** ls -la

**Explanation:**

- ls: List files
- -la: Show all files (including hidden) with details

**Output:**

```
drwxr-xr-x  .

drwxr-xr-x  ..

drwxr-xr-x  .git

-rw-r--r--  calculator.sh
```

You can see .git folder!

**CRITICAL NOTE:** If we delete this .git folder, then our entire folder will NOT be watched by Git anymore. All history will be lost!

---

**STEP 6: EXPLORE .git FOLDER**

**Command:** ls .git

**Output:**

```
config

description

HEAD

hooks/

info/

objects/

refs/
```

---

## UNDERSTANDING .git FOLDER STRUCTURE

### 1. objects/ folder:

- Contains all the content for your repository
- Basically all that you create inside your repo
- Stores commits, trees, blobs

### 2. hooks/ folder:

- Contains scripts that run automatically on certain Git events
- Used to prevent commit of API tokens, secrets, sensitive data
- Example: pre-commit hook to check code quality

### 3. config file:

- Used for configuring credentials
- Store secrets
- TLS certificates
- Remote repository URLs
- User settings

### 4. refs/ folder:

- Contains pointers to commits (branches, tags)
- refs/heads/ contains local branches

- refs/remotes/ contains remote branches

**5. HEAD file:**

- Points to current branch
- Tells Git which branch you're on

---

## STEP 7: CHECK GIT STATUS

**Command:** git status

**Output:**

```
On branch master


No commits yet


Untracked files:
  (use "git add <file>..." to include in what will be committed)
        calculator.sh


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

---

## UNDERSTANDING GIT STATUS

**git status command:** Used to see the number of tracked and untracked files.

**Untracked files:**

- Shown in RED color
- Git is not watching these files
- Changes won't be saved

**Tracked files:**

- Shown in GREEN color
- Git is watching these files
- Changes will be saved on commit

---

## STEP 8: ADD FILE TO STAGING (TRACK FILE)

**Command:** git add calculator.sh

**Explanation:**

- git add: Stages file for commit
- calculator.sh: File to track
- Now Git will watch this file

**Verify:** git status

**Output:**

```
On branch master


No commits yet


Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
      new file:   calculator.sh
```

Now calculator.sh is in GREEN color, indicating it has been tracked by Git!

---

## STEP 9: MODIFY THE FILE

**Open calculator.sh again:** vi calculator.sh

**Change content to:**

```
x=a+b
y=a*b
```

**Save and exit**

---

## STEP 10: CHECK STATUS AFTER MODIFICATION

**Command:** git status

**Output:**

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   calculator.sh

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   calculator.sh
```

---

## UNDERSTANDING THE OUTPUT

**Two sections:**

**1. Changes to be committed (GREEN):** new file: calculator.sh

This is the original version that was added earlier.

**2. Changes not staged for commit (RED):** modified: calculator.sh

This is the new modification that hasn't been staged yet.

**Git shows:**

- File is tracked
- But changes are not staged for commit
- Need to add again to stage new changes

---

## STEP 11: SEE EXACT CHANGES

**Command:** git diff

**Output:**

```
diff --git a/calculator.sh b/calculator.sh
```

```
index abc123..def456 100644
--- a/calculator.sh
+++ b/calculator.sh
@@ -1 +1,2 @@
 x=a+b
+y=a*b
```

**Explanation:**

- Minus sign (-): Lines removed
- Plus sign (+): Lines added
- Shows exact changes made to the file

git diff shows the exact changes that have been made in the file.

---

**STEP 12: RESET FILE TO PREVIOUS VERSION**

**Restore original content:** vi calculator.sh

**Change back to:** x=a+b

**Save and exit**

---

**STEP 13: CHECK STATUS AGAIN**

**Command:** git status

**Output:**

```
On branch master


No commits yet


Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   calculator.sh
```

Now everything is tracked by Git, all in GREEN.

## STEP 14: COMMIT CHANGES (CREATE VERSION 1)

**Command:** git commit -m "This is my first version of addition"

**Explanation:**

- git commit: Save the current state
- -m: Message flag
- "This is my first version of addition": Commit message describing changes

**Output:**

```
[master (root-commit) abc1234] This is my first version of addition
 1 file changed, 1 insertion(+)
 create mode 100644 calculator.sh
```

## UNDERSTANDING GIT COMMIT

**What is a commit?** A commit is a snapshot of your code at a specific point in time.

**Why commit?** We have to keep track of the versioning or the changes every time. This is done with the help of git commit command.

**Commit = Save point in a video game**

- You can always go back to this save point
- You can see what changed since last save
- You can compare different save points

## STEP 15: CHECK STATUS AFTER COMMIT

**Command:** git status

**Output:**

```
On branch master
nothing to commit, working tree clean
```

**Explanation:**

- nothing to commit: All changes are saved

- working tree clean: No new modifications
- Everything is up to date!

---

### STEP 16: MODIFY FILE AGAIN (VERSION 2)

**Open file:** vi calculator.sh

**Change content to:**

```
x=a+b

y=a-b
```

**Save and exit**

---

### STEP 17: CHECK STATUS

**Command:** git status

**Output:**

```
On branch master

Changes not staged for commit:

  (use "git add <file>..." to update what will be committed)

  (use "git restore <file>..." to discard changes in working directory)

        modified:   calculator.sh


no changes added to commit (use "git add" and/or "git commit -a")
```

calculator.sh is now in RED, indicating changes are not staged.

---

### HOW DOES GIT UNDERSTAND ALL OF THIS?

**Answer:** Git is keeping track of all of this with the help of the .git folder which we learned about previously.

**What Git tracks:**

- Every file content
- Every change made

- When changes were made
- Who made changes
- Commit history
- Branches
- Remote repository information

All stored in .git folder!

---

### STEP 18: STAGE NEW CHANGES

**Command:** git add calculator.sh

**Check status:** git status

**Output:**

```
On branch master

Changes to be committed:

 (use "git restore --staged <file>..." to unstage)

      modified:   calculator.sh
```

Everything is now in GREEN, so Git is tracking our new modified code.

---

### STEP 19: COMMIT VERSION 2

**Command:** git commit -m "This is my second version with subtraction"

**Output:**

```
[master def5678] This is my second version with subtraction

 1 file changed, 1 insertion(+)
```

---

### STEP 20: VIEW COMMIT HISTORY

**Command:** git log

**Output:**

```
commit def567890abcdef (HEAD -> master)

Author: Sushmita Hubli <sushmita@example.com>
```

```
Date:   Wed Nov 6 15:30:45 2024 +0530

    This is my second version with subtraction

commit abc123456789abc
Author: Sushmita Hubli <sushmita@example.com>
Date:   Wed Nov 6 15:25:30 2024 +0530

    This is my first version of addition
```

---

**UNDERSTANDING GIT LOG**

**Information shown:**

1. Commit ID: Unique identifier (abc123456789abc)
2. Author: Who made the commit
3. Date and time: When commit was made
4. Commit message: Description of changes

**Use case:** We can see all commit details like author, date and time of commit, commit message, etc.

---

**STEP 21: GO BACK TO PREVIOUS VERSION**

**Scenario:** Manager told you to go back to the previous version of code (version 1 - only addition).

**Step 1: View commit history** git log

**Step 2: Copy commit ID of previous commit** abc123456789abc

**Step 3: Reset to that commit** git reset --hard abc123456789abc

**Output:** HEAD is now at abc1234 This is my first version of addition

---

**UNDERSTANDING GIT RESET**

**Command:** git reset --hard <commit_id>

**Explanation:**

- git reset: Move HEAD to specified commit
- --hard: Discard all changes after that commit
- commit_id: The commit you want to go back to

**WARNING:** --hard is destructive! It permanently deletes changes after that commit.

**Safer alternatives:**

git reset --soft <commit_id>  (Keep changes staged)

git reset --mixed <commit_id> (Keep changes unstaged)

---

### STEP 22: VERIFY FILE CONTENT

**Command:** cat calculator.sh

**Output:** x=a+b

**Success! You're back to the previous version with only addition!**

---

### VISUAL REPRESENTATION OF VERSIONING

**Version 1 (Commit abc1234):** calculator.sh: x=a+b

**Then make changes**

**Version 2 (Commit def5678):** calculator.sh: x=a+b y=a-b

**Then git reset --hard abc1234**

**Back to Version 1:** calculator.sh: x=a+b

---

### SHARING CODE WITH GITHUB

**What we've done so far:**

- Created Git repository locally
- Made commits
- Tracked versions
- Everything is on your machine only

**Problem:** How do we share this code with other developers?

**Solution:** Use a distributed system like GitHub!

---

## STEP 23: CREATE GITHUB ACCOUNT

1. Go to https://github.com
2. Click "Sign up"
3. Enter email, password, username
4. Verify account
5. Login

---

## STEP 24: CREATE REPOSITORY ON GITHUB

**Steps:**

1. Click "New repository" button (green button on homepage)
2. Fill in details:
   - Repository name: example-calculator
   - Description: Calculator application for learning Git
   - Public or Private: Choose Public
   - Initialize with README: Don't check (we already have code)
3. Click "Create repository"

---

## STEP 25: CONNECT LOCAL REPO TO GITHUB

**GitHub will show you commands like:**

```
git remote add origin https://github.com/yourusername/example-calculator.git

git branch -M main

git push -u origin main
```

---

## UNDERSTANDING THE COMMANDS

**Command 1:** git remote add origin https://github.com/yourusername/example-calculator.git

**Explanation:**

- git remote add: Add a remote repository
- origin: Name for the remote (convention)
- URL: GitHub repository URL

This connects your local repo to GitHub repo!

---

**Command 2:** git branch -M main

**Explanation:**

- git branch -M main: Rename current branch to "main"
- (Older Git versions use "master", newer use "main")

---

**Command 3:** git push -u origin main

**Explanation:**

- git push: Upload local commits to remote
- -u: Set upstream (tracking)
- origin: Remote name
- main: Branch name

This uploads your code to GitHub!

---

**STEP 26: VERIFY ON GITHUB**

1. Refresh your GitHub repository page
2. You should see:
   - calculator.sh file
   - Your commit messages
   - Commit history

Your code is now on GitHub!

---

**STEP 27: FORK - CREATING A COPY**

**What is Fork?** Fork creates a complete copy of the repository under your account.

**How to Fork:**

1. Go to any GitHub repository
2. Click "Fork" button (top right)
3. Choose your account
4. Wait for fork to complete

**Result:**

- Original: https://github.com/originaluser/repo
- Your fork: https://github.com/yourusername/repo

You now have your own copy with full history!

---

### BENEFIT OF FORK

**Scenario:**

- Organization repository: github.com/company/project
- You fork it: github.com/yourname/project

**Benefits:**

1. You have complete copy of entire code
2. You can experiment without affecting original
3. If company's GitHub goes down, you still have code
4. You can contribute back via Pull Requests
5. Multiple developers can work independently

This is how distributed version control works!

---

### COMPLETE GIT WORKFLOW

**Local Machine:**

1. git init (initialize)
2. git add (stage changes)
3. git commit (save version)

4. git push (upload to GitHub)

**GitHub (Remote):** 5. Code stored on cloud 6. Others can clone/fork 7. Collaborate via Pull Requests

**Other Developer:** 8. git clone (download) 9. git pull (get updates) 10. Make changes 11. git push (upload changes)

---

**COMMON GIT COMMANDS SUMMARY**

**Command: git init** Purpose: Initialize new repository

**Command: git status** Purpose: Check file status

**Command: git add <file>** Purpose: Stage file

**Command: git add .** Purpose: Stage all files

**Command: git commit -m "msg"** Purpose: Save version

**Command: git log** Purpose: View history

**Command: git diff** Purpose: See changes

**Command: git reset --hard <id>** Purpose: Go back to version

**Command: git remote add origin <url>** Purpose: Connect to GitHub

**Command: git push** Purpose: Upload to GitHub

**Command: git pull** Purpose: Download from GitHub

**Command: git clone <url>** Purpose: Copy repository

**Command: git branch** Purpose: List branches

**Command: git checkout <branch>** Purpose: Switch branch

**Command: git merge <branch>** Purpose: Merge branches