Monday, 2 February 2026 11:36 PM

PAGE 1: Version Control Notes

1. Version Control

- **Definition:** A system that tracks and manages changes to software projects over time.
- **Why Important?**
  - Maintains a **complete history** of all work.
  - Allows **multiple versions** (branches).
  - Enables collaboration among developers without overwriting each other's work.
- **Version Control Systems (VCS):**
  - To implement version control, we need a **Version Control System**.
  - Popular: **Git**, others include **Mercurial**, **Subversion (SVN)**, **Perforce**.
- **Key Points:**
  - You **cannot realistically do version control without a VCS**.
  - A VCS is a **tool**, not magic—requires learning.
  - Helps in **tracking changes**, **collaboration**, and **undoing mistakes**.

2. Git is NOT GitHub

What is Git?

- Git**:** is a **Version Control System (VCS)**—a piece of software you install on your computer.
- It helps you track changes, manage versions, and collaborate on code.

What is GitHub.com / Bitbucket.org:

- **Hosting services that use Git/ VCS tools like Mercurial.**
- A **central place to store ("host") your project code**.
- Features for **team collaboration**, such as:
  - Tracking progress
  - Discussing issues
  - Conducting code reviews

Important Distinction:

- **Git (the VCS)** and **GitHub (the hosting service)** are **separate things**.
- You can use Git **by itself** on your computer without any hosting service.
- Or, you can use Git **together with a hosting service** for easier collaboration and remote access.
- Git was **created by Linus Torvalds** to help manage the Linux OS kernel.
  - The Linux kernel is a huge project:
    - **22 million lines of code**
    - **4,600 new lines added per day**
    - **13,500+ developers**
    - Running since **1991**
  - Git was created to handle this complexity.
- **We'll use Git via the command-line:**
  - There are GUI and web tools available.

- But the **command-line is better for learning**.

3. Undoing Mistakes

At some point in development, you might experience this cycle:

1. **It works!**
2. *Now I'll just add the next feature…*
3. **Damn. Now it's broken.**
4. *Argh! I can't figure out how to get back to the original version.*
5. **Start again from scratch.**

- **Problem with manual backups:**
  - Leads to messy folders: new, old, working.3, etc.
  - Hard to remember what's what.
- **Solution:** Use VCS like GIT to revert easily and keep history.
- As VCS allows you to:
  - **Revert to previous versions easily**.
  - **Track changes without creating multiple folders**.
  - **Experiment safely** using branches.

4. Why Text Editor Undo Isn't Enough

- **Limitations:**
  1. You often want **both old and new versions**.
  2. Undo history is **temporary** (lost when editor closes).
  3. Too many changes across files to remember.
  4. Track changes across **multiple files and commits**.
- **Better Solution:** VCS keeps all versions permanently.

5. How Does a VCS Help?

- For every update (**commit**) , the VCS records:
  - **What was changed**
  - **Date and time of the change**
  - **The person responsible** (important for team projects)
  - **A description** entered by that person

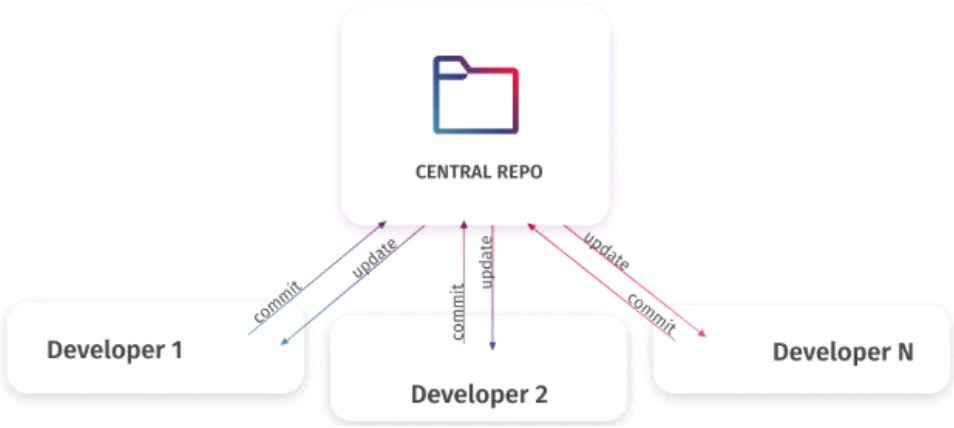
- You can:
  - View commit history
  - Retrieve old versions
  - Compare differences
- **Workflow:** Make changes → Commit with description → Safe to undo anytime.

6. Repositories

Repository (repo): Stores the complete project and all versions

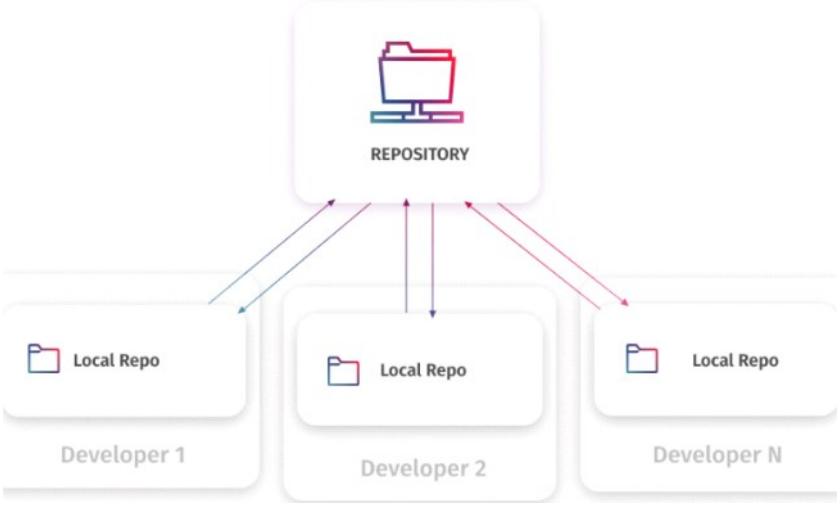
Types of Repositories

a. **Older VCS – Centralized Repository**
   - One central repository.
   - Users check out files and check in updates.
   - **Limitation:** Nobody else can modify the same files at the same time.

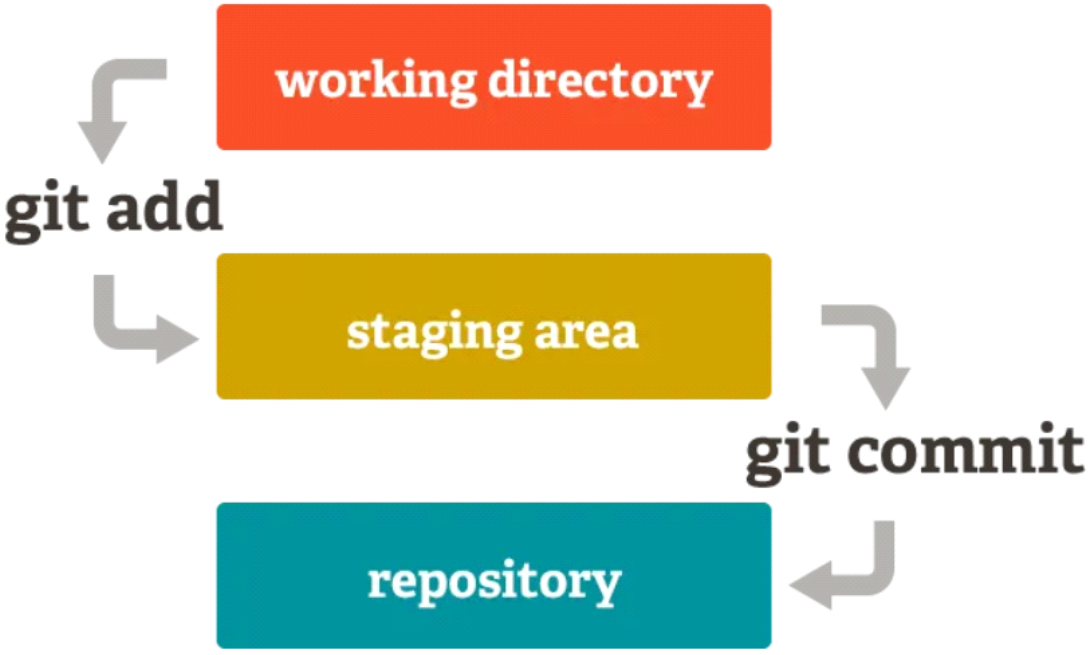


b. Newer VCS – Distributed Repository (e.g., Git, Mercurial)

- Each user has a local repo + central repo.
- Typically, there's also a **central repository** (often hosted on **GitHub**, **Bitbucket**, etc.).
- Different repositories are **periodically synced** with each other.
- People can **update the same files at the same time**.
- Intelligent algorithms help **merge changes together**.



7. Local Repository vs Working Directory



[Git Gud: The Working Tree, Staging Area, and Local Repo | by Lucas Maurer | Medium]
[Why Most New Developers Get Git Wrong: Understanding the Four Code Stages | Rabin's Blog]

- **Working Directory:**
  - Stores The version you're currently working on.
  - Usually the **latest version** (but branches can change this).
  - It's just a **normal directory** on your computer.

- **Local Repository:**
  - This folder is your **local repository**. It stores:
    - **All saved versions of your code** (every commit you make).Except any uncommitted changes in the working directory).
    - Information about branches, tags, and configuration.
  - Typically located in a **hidden sub-directory** inside the working directory:
    - For Git: .git/
    - For Mercurial: .hg/
  - Uses a **VCS-specific format** (not directly human-readable).

- **Important Points**
  - **Committed changes** = Changes you saved using **git commit**. ☑ stored in the .git folder.
  - **Uncommitted changes** = Changes you made but **haven't committed yet**. ❌ stored in **working directory**, not in .git.

- **What the VCS Does:**
  - Save working directory/ current work into repo when u commit
  - Load any version from repo into working directory.

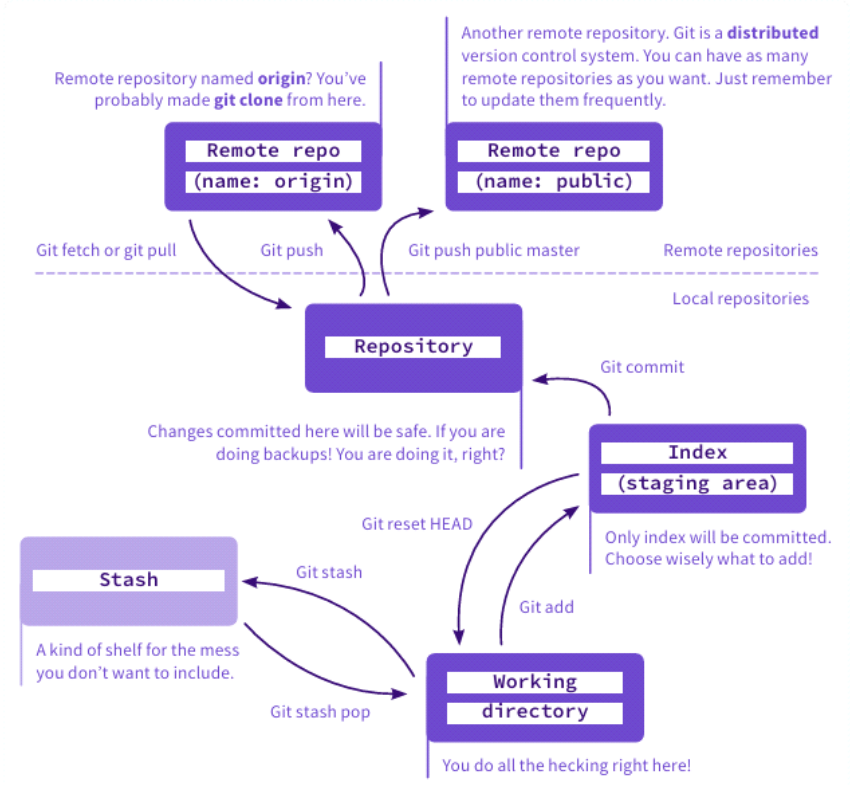
Example

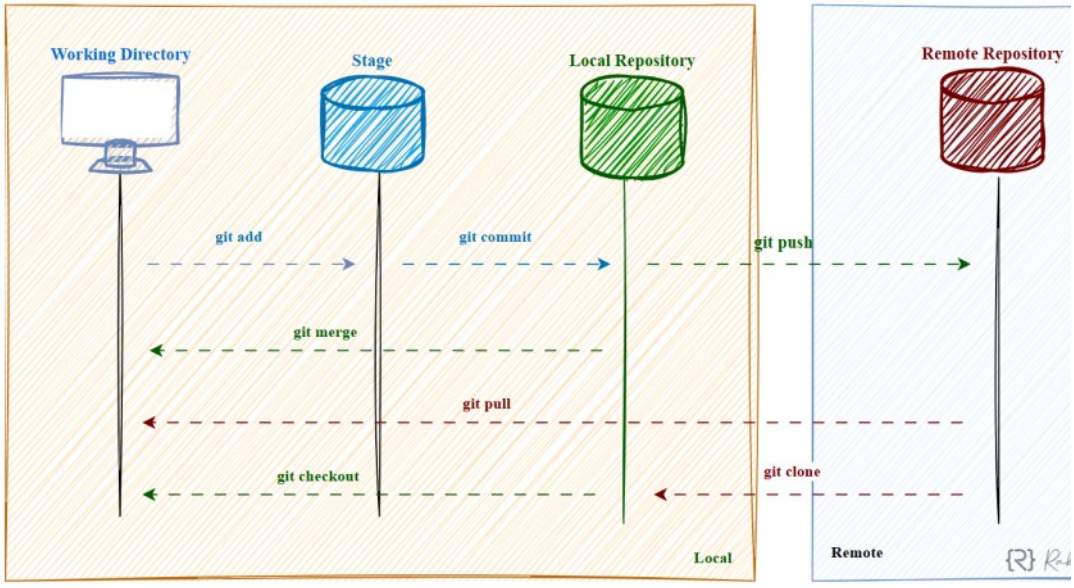
i. You edit file1.txt → This change is in the **working directory**.
ii. You run **git add** file1.txt → Now it's staged (ready to commit).
iii. You run git commit -m "Update file1" → Now the change is stored in the **local repository (.git folder)**.

So:

- .git = Stores all committed versions.
- Working directory = Your current files, including uncommitted changes.

☑ Git workflow (Working Directory → Staging → Local Repo → Remote Repo)





🔁 **How Everything Flows (Big Picture)**

▨ **1** Working Directory
▧ **2.** Index (Staging Area)
▨ **3.** Local Repository

▨ 4. Remote Repository 📍 GitHub / GitLab / Bitbucket
Remote repo -> Local repo = git fetch
Remote repo -> working directory = git pull

▩ **5.** Stash (Temporary Shelf)
📍 **Hidden storage**
- Used when:
  - You're not ready to commit
  - But need to switch branches

git stash
git stash pop
Think of stash as: "Put this mess aside, I'll deal with it later"
⏳ **TEMPORARY COMMITS (Stashing)**
Temporarily store modified changes so you can switch branches safely
- **git stash** #Stash (save) your **uncommitted changes** (tracked files by default) and revert your working directory to a clean state.
- **git stash list** -Show all stashes (stack order).
- **git stash pop** **#**Apply the latest stash **and remove it** from the stash list.
- **git stash apply** *(useful to know)* # Apply the latest stash **but keep it** in the stash list.
- **git stash drop #**Delete a stash. ☑ Best written as: git stash drop stash@{0} (or whichever stash index)

Extra paths:
- git reset HEAD → Repo → Index
- git checkout -- file → Repo → Working Directory
- git stash → Working Directory → Stash

Staging and Committing
☑ **Commit Best Practices**
- A commit should represent **one thing** – one bug fix or one small feature.
- If you make two distinct changes, create **two separate commits**.
  - This makes it easier for others (and your future self) to understand what was done.

☑ **Unstage Files Before Commit**
- If you accidentally staged a file, you can unstage it:
  git reset MyCode.java
  git reset

☑ **Fixing Mistakes**
- If you mess up a commit, **don't edit the old commit** (it's risky).
- Instead, make **another commit** to correct the mistake.
  - This keeps history clear and avoids breaking version control.

⚠ Changing already-made commits is possible but **not recommended** because it can defeat the purpose of version control. In most cases, commits should be considered **permanent**.

8. Branching

Why Do We Use Branches?

- A branch allows working on **different versions of a project at the same time**.
- Normally, commits usually form a straight line: c1 → c2 → c3 → c4.
- Each commit is based on the previous one.
- Useful for:
  - Adding new features
  - Fixing bugs
  - Experimenting without breaking main code
- Common setup:
  - **Main branch** (stable version)
  - **Feature branches** (new changes/experiments)

In a Team

- Different developers can work on **different branches** at the same time.
- When ready, you **merge** the branch back into the main project
- This avoids conflicts and keeps the main branch stable.

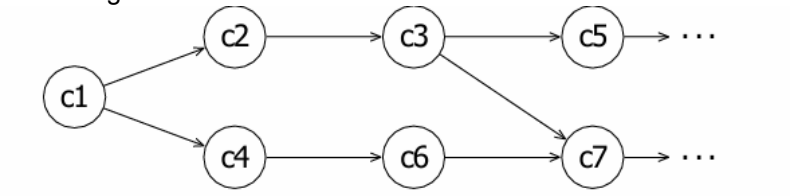
Why Is It Important?

- Main branch = **latest working version**.
- Branching helps:
  - Keep main version safe
  - Make experimental changes without affecting others
  - Work independently while collaborating

Branches Example

Commits can **split into branches**:

- Two branches start from c1
- Each branch has its own commits
- Merge later at c7



Steps

Default Branch When you create a new Git repository, you start with one branch called **main** (or sometimes master).

i. **Create a New Branch :**git branch updating_amze
ii. **Switch to New Branch** git checkout branch2
iii. **Create and Switch in One Step git checkout -b branch2**
iv. **Merge Branch into Main** git merge mynewbranch
v. Shows all commits in the current branch's history: **git log**
vi. **List All Branches** git branch
vii. **Display Current Branch Name** :git status
viii. **Rename a Branch :** git branch -m mynewbranch featurexyz
(Renames mynewbranch to featurexyz.)
ix. ***Create a Branch from a Specific Commit:****git branch mynewbranch ff823e*
x. **Del from GITHUB REPO /remote:** git push origin --delete (branch name)
xi. **Delete Branches :** git branch -d mynewbranch. You can delete branches, usually after merging use -D instead of -d forces deletin

9.Merging in Git

1. Why Do We Merge?

- Branching is useful **only if branches can be merged later**.
- Most development happens in **feature branches**.
- Eventually, all completed work must be merged into the **main (master) branch**.
- Merging allows multiple lines of development to come together into one stable codebase.

2. Basic Merge Commands

Switch to the main branch:git checkout main

Merge another branch into main:git merge mynewbranch

If conflicts were fixed

git add conflicted-file.java

git commit -m "Resolved merge conflict"

3. What Happens During a Merge?

Example: merging mynewbranch into main

- Work from mynewbranch is integrated into main.
- mynewbranch is usually no longer needed after merging.
- Git creates a **merge commit** that:
  - Combines the latest commits from both branches
  - Preserves the history of each branch

4. How Git Performs a Merge

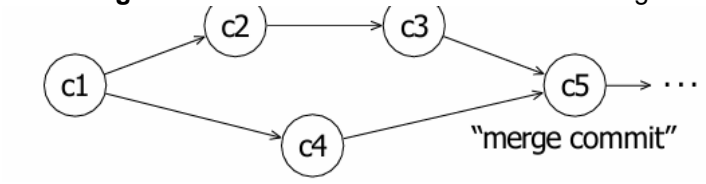
Why Is Merging Hard?

- Branches may contain **different or conflicting changes**.
- Git must combine multiple versions of code into one working result.
- This is complex, but Git handles most cases automatically.

5. How Git Knows What to Merge

Git uses a **common ancestor**:

- Finds the commit where branches originally split
- Compares:
  - Changes in the current branch
  - Changes in the branch being merged
- Creates a **merge commit** that combines both sets of changes
  -



6. Three-Way Merge (Key Concept)

Git performs a **three-way merge** using:

i. **Base commit** (common ancestor)
ii. **Current branch**
iii. **Branch being merged**

If changes don't overlap → merge is automatic

If changes overlap → merge conflict occurs

7. Automatic vs Manual Merging

Can Git Merge Automatically?

- Often yes

- Automatic if branches modified **different parts of the code**

When Does a Merge Conflict Occur?

- Both branches modify the **same lines** of the same file

8. Merge Conflicts Explained

What Git Shows

CONFLICT (content): Merge conflict in Xyz.java

This means:

- Both branches changed the same section
- Git cannot decide which change to keep

How Git Handles It

- Leaves conflict markers inside the file
- Shows which change came from which branch
- Requires **manual resolution**

9. Resolving a Merge Conflict (Steps)

i. Open the conflicted file (e.g., Xyz.java)
ii. Review both changes
iii. Decide what to keep / combine
iv. Remove conflict markers
v. Stage and commit

git add Xyz.java

git commit -m "Merged abc → xyz changes"

Helpful Command: git status Shows files that need attention.

🔟 Multiple Repositories in Git

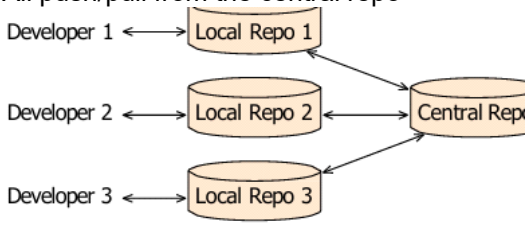
Git is a **distributed version control system (DVCS)**.

Repository Structure

- Each developer has a **local repository**
- A **central repository** (e.g., GitHub) is used for sharing

Example:

- Developer 1 → Local Repo
- Developer 2 → Local Repo
- Developer 3 → Local Repo
- All push/pull from the central repo



1□1□ Why Multiple Repositories Matter

i. **Safety**

- In a **centralized VCS**, if the central repo is corrupted or deleted, you lose everything.
- Distributed VCS acts as a **natural backup system** because every developer has a full copy.

ii. **Reliability:**If the central server goes down:

- With **distributed VCS**, developers can still work using their local repo.
- With **centralized VCS**, work stops completely → huge productivity loss.

iii. **Performance:**For large teams/projects:
- Centralized VCS can get overloaded with CPU tasks and network traffic.
- Distributed VCS reduces network dependency and spreads load across developers' machines.
- Team size doesn't affect performance much because each developer works locally.

☑ **Key takeaway:** Distributed VCS (like Git) is safer, more reliable, and better for performance compared to centralized systems.

🎯 **Step-by-step: Merge Conflict Resolution**

☑ **1. Switch to main branch:**git checkout main
☑ **2. Merge if_replace first:git merge if_replace** This should succeed automatically.
☑ **3. Merge rename_vars next:git merge rename_vars** This will trigger a **merge conflict** in Maze.py.
☑ **4. Check the conflict:git status**

You'll see:
Unmerged paths:
both modified: Maze.py

☑ 5. Open Maze.py and scroll to the conflict **. Delete the conflict markers and both versions** & save
☑ **7. Replace with the correct merged version**

Paste this clean block:

```
cell = grid[row][col]
if cell == EMPTY:
   grid[row][col] = VISITED
elif cell == WALL:
   done = True
   print("MESSAGE 3:You stumble blindly into a solid concrete wall.")  # Hit wall.
elif cell == END:
   done = True
   solved = True
   print("MESSAGE 4:SOLVED!")  # Solved.
else:
   pass  # Do nothing
```

☑ **8. Save Maze.py:**Make sure the file is clean and runs without syntax errors.
☑ **9. Stage the resolved file:** git add Maze.py
☑ **10. Commit the merge resolution:git commit -m "Resolved merge conflict between if_replace and rename_vars"**
☑ **11. Confirm everything is merged:git log --branches --graph**
You'll see the merge commits and how the branches came together.

☑ Tag

1 This creates a permanent label so you can refer back to the work later.
git tag archive/updating_maze updating_maze
git tag archive/updating_messages updating_messages

These tags are just names — archive/ is a naming convention to show they're archived.

2 run**:git tag:**
archive/updating_maze
archive/updating_messages

🤔 Why this matters

- Tags preserve the history of your work
- Deleting unused branches keeps your project clean
- This is good Git hygiene — especially for team projects