Submitted By : **Rithika**

Source – Google, websites link Geekforgeeks, datacamp, Github:discover-devops , so on

Branching Strategies in Git:

- To manage the process of writing, merging, and deploying code with the help of VCS\
- Keep the Prj repo – organized, error-free and avoid merge-conflicts(many users push and pull code at same time from same repo)

1.) Creation of a branch:

- Commands used:

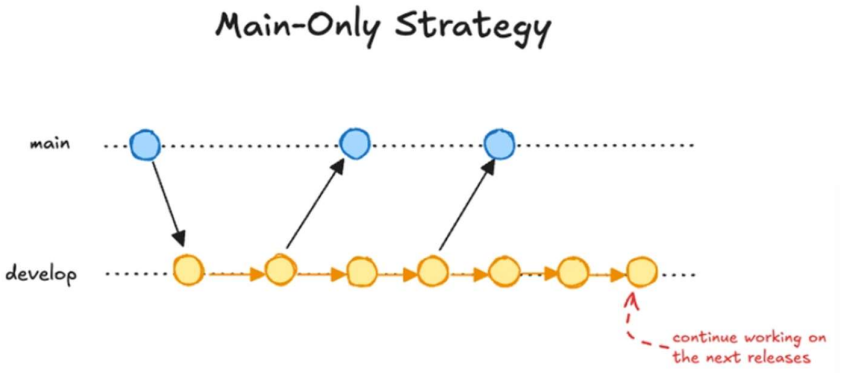
git branch <branch__name>
git checkout <branch_name>(To move to that parti branch from curr)
git checkout -b <branch_name>(create & move at same time)
git branch(list all local branch , current branch with *)
git branch -r(remote branches)
git branch -a(local+ remote branches)
git branch -d <bname>(Dlt branch only if merged already)
git branch -D <bname>(Force dlt even if not merged)
git branch -m <oldname> <newname> (renaming)
git branch -v(last commit)
git branch –merged(List the merged branches)
git branch –no-merged(list branches that are not merged yet)

2.) Merging (Overview of it in general):

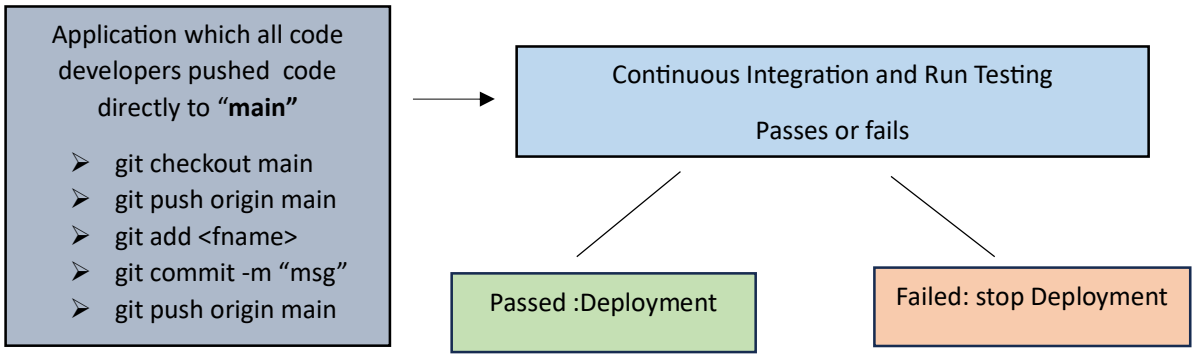
I have a prj with "main" branch(with commit-A,B,C) , and now I plan with add a new branch-"Feature"(have commit-D,E) to it , in-order to combine the work from 1 branch into another branch, I use "**Merging"**

- git merge <branch_name>

3.) Main-Only Stragety:



- All Development happens on single branch – main or master

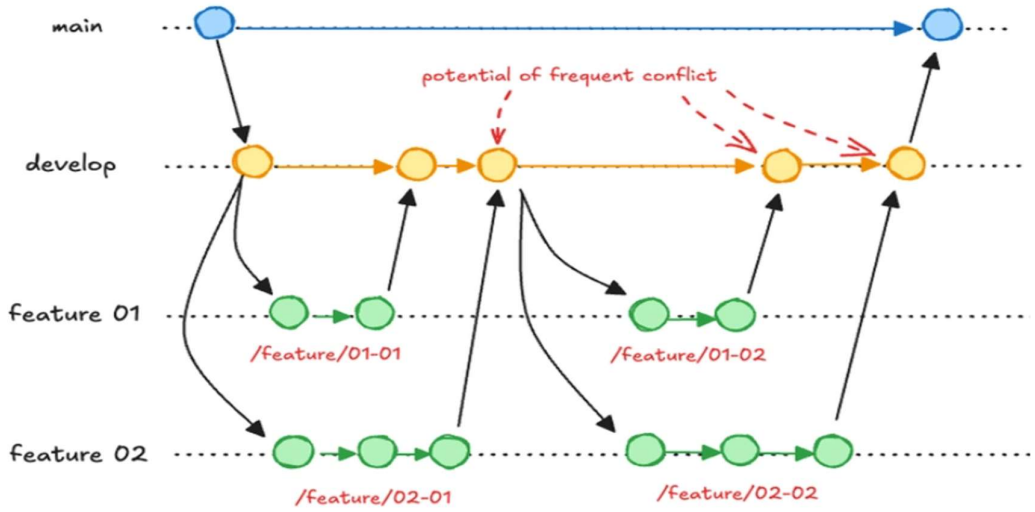


- **Regression**: New change that affect the working as like adding a new feature that affect the previous functionality of the application
  git revert <new_commit> (returning to stable previous state)
  git push origin main

- If that bad commit is deployed then we can **"rollback"** to previous deployment
  deploy <prev_version> again

Pros	Cons
• No additional branch, No conflict	➢ Difficult to track process of each features
• Simple , easy to undertand	➢ Only unit test can be done
• Used when few dev are involved	➢ Hard to review code

4.) <u>Feature Branching:</u>



- Each feature= Each separate branch

- After pushing a branch, the developer opens a PR(Pull Request) to request review and approval from a teammate before the branch is merged into main
- After Testing, branches are merged into "**main"**
- **CI** fails: Blocks the merge to protect the shared codebase

Commands:

- ✓ git checkout main
- ✓ git checkout -b develop
- ✓ git pull
- ✓ git checkout -b feature
- ✓ git add <file_name>
- ✓ git commit -m "Message"
- ✓ git merge feature
- ✓ git push origin develop
- ✓ git branch -d feature
- ✓ git checkout main
- ✓ git merge develop
- ✓ git push origin main

Pros:	**Cons:**
• Clear, modular code history	• Merge conflicts- 2 ppl editing same file
• Easy to Rollback if a branch affect or have a bug	• Longer the branch lives, more it creates risk and conflict
• Multiple developers can work on different features simultaneously	• Updating branches frequently- overhead

Notes:

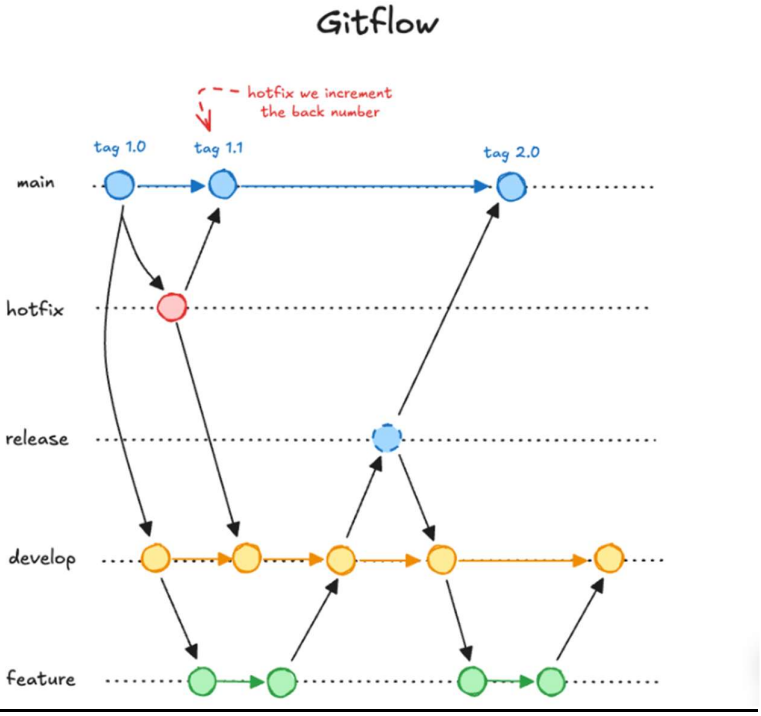
- ✓ **Pull Request**:
  - Asking permission to add changes to "**main",** provide a controlled merging.
  - **Example**: If there are 2 person (A,B) working on 2 diff features – login, payment , B completed and merged its changes to main and now A wants the latest commit from main so it uses **"** git pull origin main"

- ✓ When to use Feature branch strategy?
  - Working on multiple features
  - Clear tracking of features

- ✓ Why to delete a branch after merging?

  Branches are just a pointer now on merging main, feature they gets are common pointer , so on deleting it deletes the pointer not the code , thus making clean and organized workspace.

5.) **GitFlow:**



- Structured , multi-stage s/w development
- Persistent and ephemeral branches.

Persistent Branches

1.) Main – Stable Production code
2.) Develop-Work in progress, merge all completed features
3.) Release- Before Publish, create in develop , fix bugs

Ephemeral Branches:

1.) Feature- created and merged in develop, separate functionality
2.) Hotfix- Emergency usage, created and merged in main

Example:

I have a Cake shop where main- cake ready to sell, develop- making cake, feature-cream, toppin, release- taste testing, hotfix-emergency if too much sugar or other bug

Commands:

git checkout main

git checkout -b develop

git push origin develop

git checkout develop

git checkout -b fetaure1

Commands:

git add .

git commit -m “msg”

git push origin feature1

git checkout develop

git merge feature1

git branch -d feature1

git push origin --delete feature1

git checkout -b release1.0

git checkout main

git merge release1.0

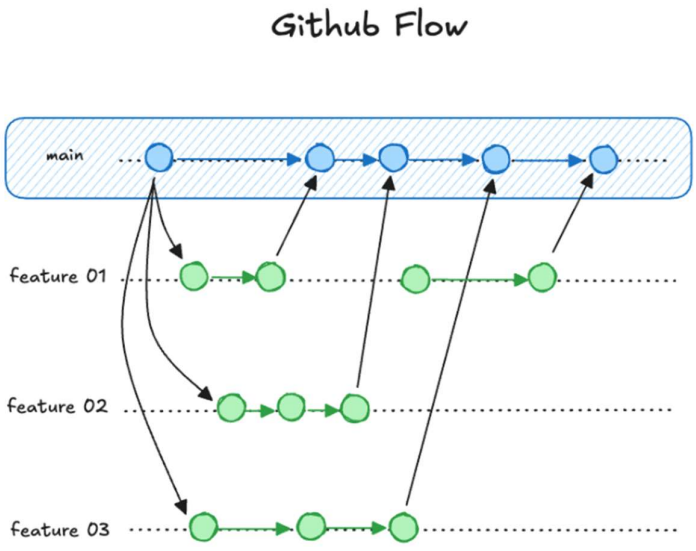
git checkout -b hotfix

git checkout main

git merge hotfix

Pros:	**Cons:**
• Frequent release, short production cycle	• Not ideal for rapid and continuous deployment needs
• Well-suited for parallel development,Safe Hotfix, large teams	• Time consuming- overhead

6.) <u>GitHub Flow:</u>



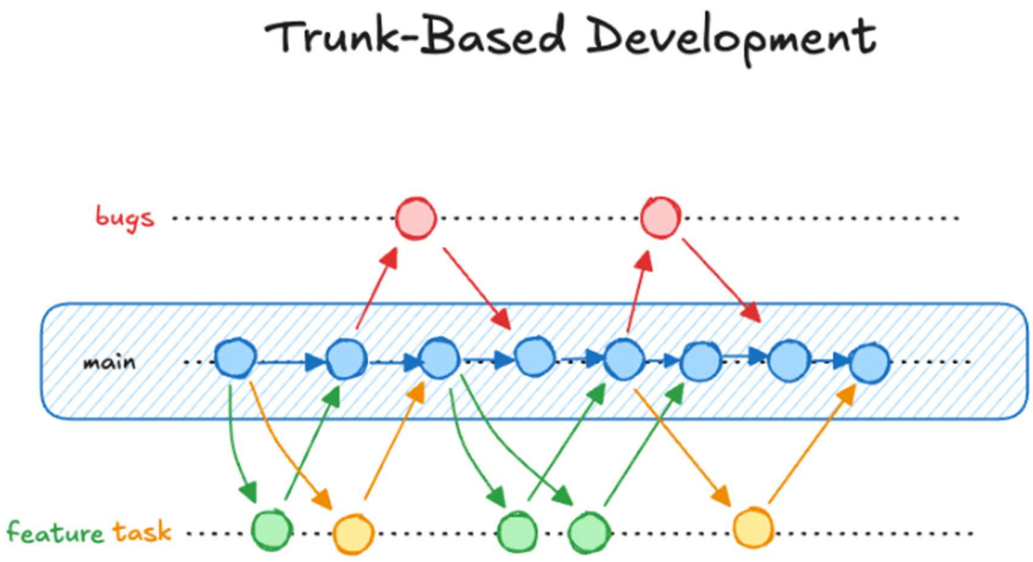
- Developers create feature branches, merge them into the main branch, and deploy immediately.
- Branch naming convention - /{author}/{short_description}

Commands

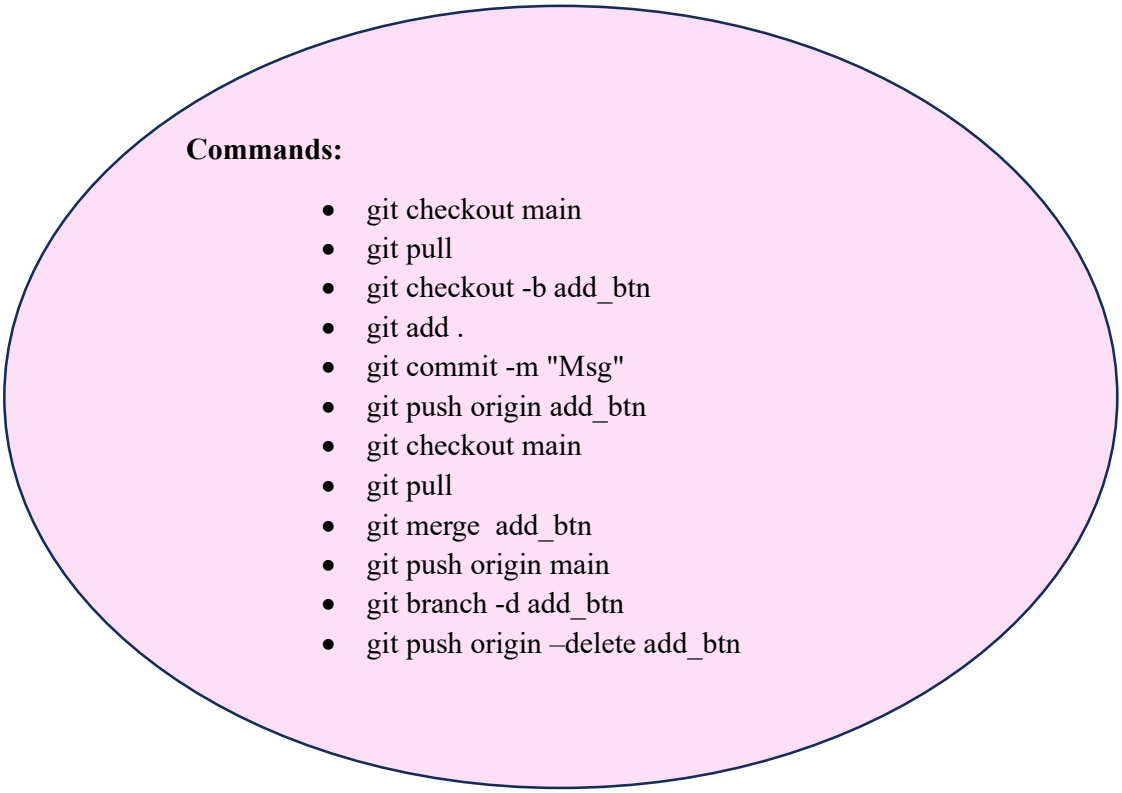
- git checkout main
- git pull
- git checkout -b rithika/add_btn
- git add .
- git commit -m “msg”
- git push origin rithika/add_btn
- Pull Request in GitHub UI -> Review and Merge
- git checkout main
- git pull
- git merge rithika/add_btn
- git push origin main
- git branch -d rithika/add_btn
- git push origin –delete rithika/add_btn

Pros:	**Cons:**
• It have only feature branch with PR thus easy to use	• Not ideal for long term development
• Continuous deployment with rapid feedback	• Potential instability with main branch should always be ready

7.) <u>Trunk-Based Development:</u>



- Instead of long feature branches, they make **small branches** or commit straight to main.
- Changes are small, frequent, Branches are short lived for few hrs
- Beast for Agile Team, Strong CI/CD
- Trunk = main branch
- Ex: Imagine having 3 ppl to change bg clr, add btn, fix spelling mistake , then they create separate 3 small branch and merge them .



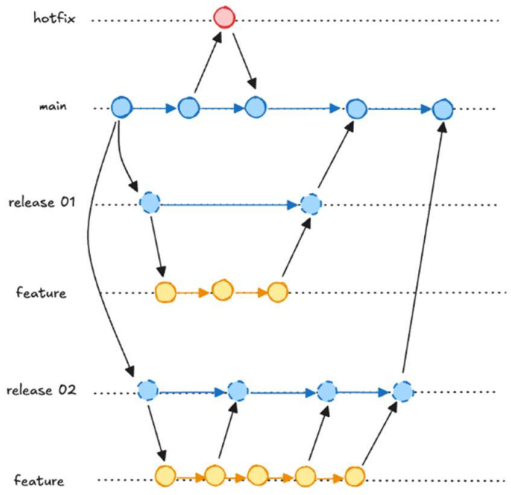
Pros	Cons
• Fewer Merge Conflicts , Development cycle are faster	• All commit go directly to "main"- Main branch can become unstable if proper testing isn't enforced.
• Encourage collaboration	• Strict Testing process

8.) <u>Release Branching:</u>

Ex: I have 3 edition of textbook -of year 2023, 2024, 2025 , If there is mistake in edition 2023 , then only that is corrected not all future one

- main → stable code
- release branches → prepare and test version
- feature branches → build new features
- hotfix branch → urgent fixes for production
- Separate branches are maintained for each release version, often labeled with version numbers.

Release Branching



Pros:	Cons:
• Clear history of all releases.	• Too many branches can become unmanageable.
• Projects with long-term support (LTS) requirements	• Not ideal for fast-moving project

Commands:

- git checkout main
- git checkout -b feature1
- git add .
- git commit -m "Msg"
- git push -u origin feature1
- git checkout main
- git checkout -b release/v1.1
- git merge feature1
- git push origin release/v1.1
- git branch -d feature1
- git push origin --delete feature1

- git checkout main
- git merge release/v1.1
- git tag -a v1.1 -m "version 1.1"
- git push origin main --follow-tags
- git checkout main
- git checkout -b hotfix1
- git checkout main
- git merge hotfix1
- git tag -a v1.1.1 -m "Fix "

Common Issues and Their Solutions: (Learning few Advanced Commands)

1.) **Merge Conflict:** change in 2 branch affects same file lines
   - ✓ Open & remove conflict markers(>>>>>>>, =======,..)
   - ✓ Add the resolved file
   - ✓ Regularly pull changes from base branch

2.) **Forgotten Branch Merge, Branch Bloat(**Too many branch causes clustering repo)**:**
   - ✓ Branch being forgotten
   - ✓ git branch -d <bname>(Dlt branch only if merged already)
   - ✓ git branch -D <bname>(Force dlt even if not merged)

3.) **Large Divergences Betwwen Branches:**
   - ✓ When two branches are separate for a long time, they accumulate many commits independently.
   - ✓ So when you finally try to merge them, many merge conflicts occur because the same files or lines have changed in different ways
   - ✓ Use : git rebase

4.) **Unclear Branch Purpose:**
   - ✓ Use descriptive name, enforce a branch naming convention.
   - ✓ Like : rithika/add_btn(work in a team)

5.) **Accidental Pushes to Wrong Branch:**
   - ✓ Use : git reset –hard origin/main(All local changes and commits that are not on the remote will be deleted permanently.)
   - ✓ git push –force(Pushing changes to remote even if it overwrite)

6.) **Deleted or Lost Branch:**
   - ✓ Branch deleted accidently by use of " **git reset –hard <prev_commit>** "
   - ✓ Use : git reflog (history of everything your HEAD has pointed to)
   - ✓ git checkout -b restore <commit_hash>(connect the restored branch with old)