VERSION CONTROLLING & WORKFLOWS WITH GIT

APPLICATION FRAMEWORKS (SE3040)

LEARNING OUTCOMES

- After completing this lecture, you will be able to,
  - Apply the basic concepts of Git for version controlling
  - Apply an efficient branching strategy for a team
  - Describe what a git workflow is and why a workflow is important.
  - Apply a simple git workflow to your own projects.

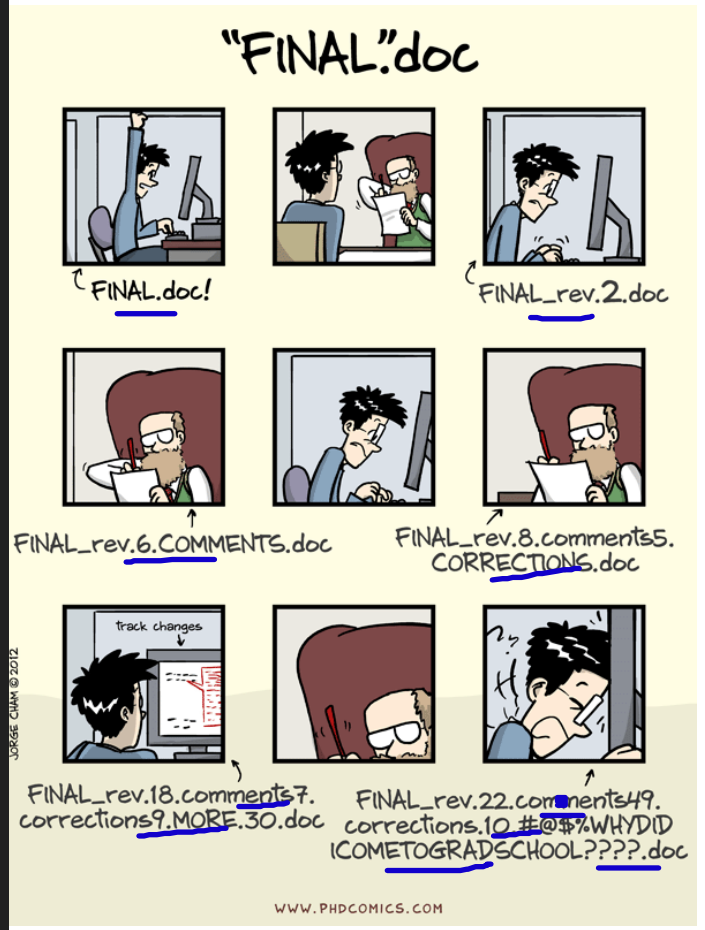
CONTENTS

VERSION CONTROLLING: WHAT?

- “Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.” [1].

WHAT YOU KNOW ABOUT VERSION CONTROLLING

Feature / Aspect	First-Gen VCS (Local VCS)	Centralized VCS (CVCS)	Distributed VCS (DVCS)
Repository Location	Stored locally on the developer's system	Single central server	Multiple repositories (central + local)
Collaboration Support	Poor; designed for individual use	Good; allows multiple users	Excellent; full history available locally
Internet/Network Dependency	None	Required for most operations	Not needed for most operations
Access Speed	Fast (local access)	Slower (network-dependent)	Fast (local operations)
Data Loss Risk	High (no backup if local data is lost)	High if central server fails	Low (every user has full copy)
Branching and Merging	Limited or manual	Basic, often complex	Advanced and efficient
History Management	Basic (may not track all changes)	Maintains full history centrally	Maintains full history locally
Examples	SCCS, RCS	CVS, Subversion (SVN), Perforce	Git, Mercurial, Bazaar
Offline Work	Fully offline	Mostly not possible	Fully supported
Security Control	Basic	Centralized control and access	Decentralized; can enforce security policies



Source: https://swcarpentry.github.io/hg-novice/01-basics/

A BIT OF HISTORY ON VERSION CONTROLLING SYSTEMS (VCS)

- The first proper Version Control System (VCS), named *Source Code Control System (SCCS)* was released in 1973 by Bell Labs.
- It was a first-generation VCS, which are now collectively known as **Local Version Control Systems**.
- Local VCS were intended to track changes for individual files and checked-out files could only be edited locally by one user at a time.
- Facilitating collaboration among large teams was problematic.

A BIT OF HISTORY ON VERSION CONTROLLING SYSTEMS (VCS)

- To address the problem of collaboration, a new type of VCS were developed.
- This second generation of VCS are collectively known as **Centralized Version Control Systems**.
- Centralized VCS allowed users to refer files stored on a centralized repository through a network and allowed multiple users to use the files concurrently.
- Follows a *Client-Server* approach.
- However, they would all have to commit their code to the centralized repository, which would require network access.
- A central authority is required to maintain the Centralized repository.

A BIT OF HISTORY ON VERSION CONTROLLING SYSTEMS (VCS)

- With the advent of open-source software came the third generation of VCS. They are collectively known as **Distributed Version Control Systems**.
- The earlier Centralized VCS required a central authority to maintain the centralized repository. No such authorities are usually found in open-source projects.
- Instead, there are just contributors who may develop features which may or may not be relevant to the main open-source project.
- Therefore, Distributed VCS have *peer-to-peer (P2P)* approach to version controlling.
- Distributed VCS are widely used today.
- De-facto standard in Distributed VCS now?
  - **Git!**

GIT: THE DE-FACTO STANDARD TODAY IN VCS

- Created by **Linus Torvalds** (creator of Linux) in 2005.
- Pronounced as *Ghi-T*. Not *Ji-T (*A "git" is a cranky old man. According to online sources, Linus meant himself).
- Originally designed to do version control on Linux kernel.
- Git supports non-linear development (thousands of parallel branches) and is fully distributed.

GIT BASIC CONCEPTS

- As mentioned, git does not necessarily need a Centralized Repository to work (It can work in somewhat centralized manner too, which we will see).
- Every user maintains a **local repository** of their own.
- When starting out they will obtain this local repository,
  - from another user or,
  - they may **clone** it from a central location or,
  - They may even initialize a brand-new repository from scratch.
- This repository will contain what others have done originally, what the user has done themselves and even work of others as selected by the user.
- They can **commit** and **update** their local repository without any interference from any others.

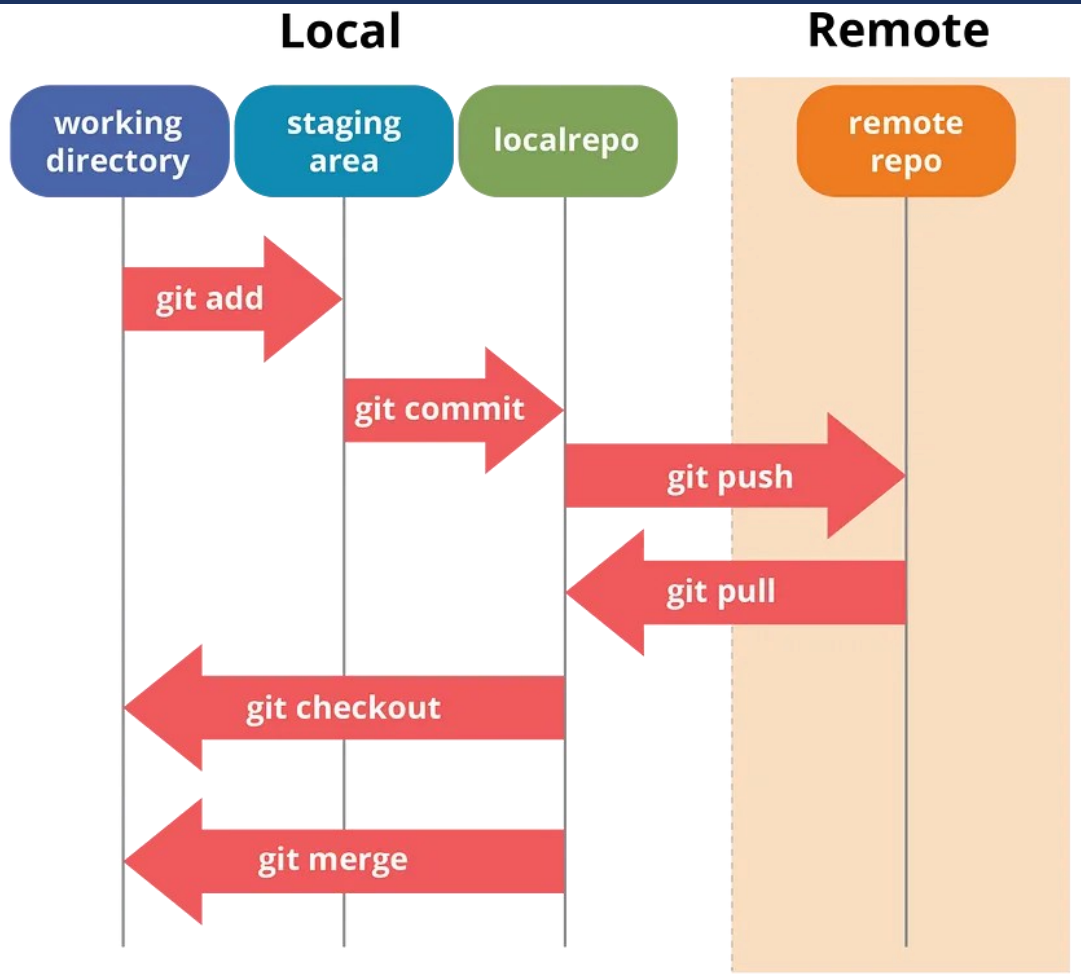
GIT CONCEPTS

- Git has two types of repositories.
  - Local repositories – which is owned by each user.
  - Remote repository – a centralized location accessible to all users. A single source of truth.
    - The users will **push** their new updates to this repository so those can be shared with others as well.
    - The users will **pull** new updates from others from this repository.
  - Why do we need Remote repositories?

    magine you're working on a group project with friends. If each person keeps their version of the project only on their own computer, it gets confusing and hard to share updates.
    - Scenario Discussion: The problem with Distributed Version Controlling. Which project is the "right" project when multiple versions of the project exist?
    - Solution? Remote repositories as a single source of truth!
  - Git != GitHub (Git is not Github)!

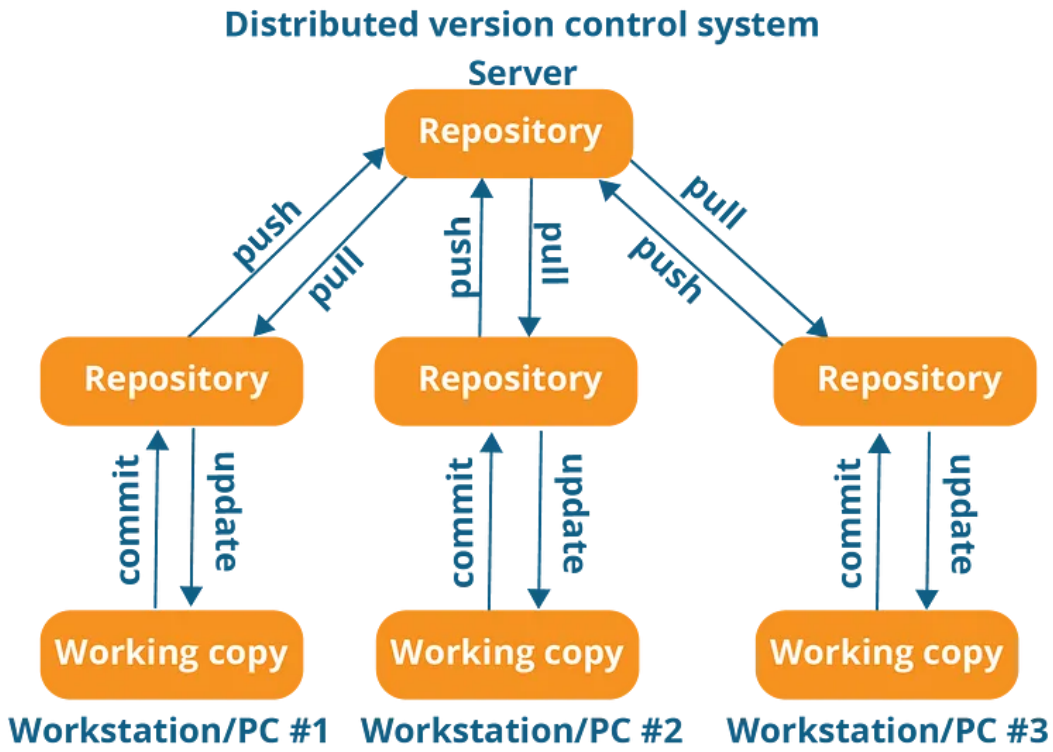
GIT CONCEPTS

1. **Add** a new file to the local Repo.
2. Do the required changes to the file.
3. **Commit** the file to the local repo.
4. **Pull** the changes from the remote repo.
5. **Push** the local changes to the remote repo.



Source: https://medium.com/@sahoosunilkumar/how-does-git-works-5cc8444ea383

GIT CONCEPTS



Source: https://medium.com/@sahoosunilkumar/how-does-git-works-5cc8444ea383

GIT CONCEPTS

- Git works on **Commits**.
- A Commit is very simply **a snapshot of all changes in the local repository** (more correctly, the working directory).
- Each Commit has a unique Commit ID.
- A Commit ID is a unique SHA-1 hash that is created whenever a new commit is recorded.
- The most recent commit is called the **Head** (more correctly, it refers to the currently checked-out branch's latest commit).
- We will extensively deal with Commit IDs for many operations on git.

GIT BRANCHING: THE BASICS

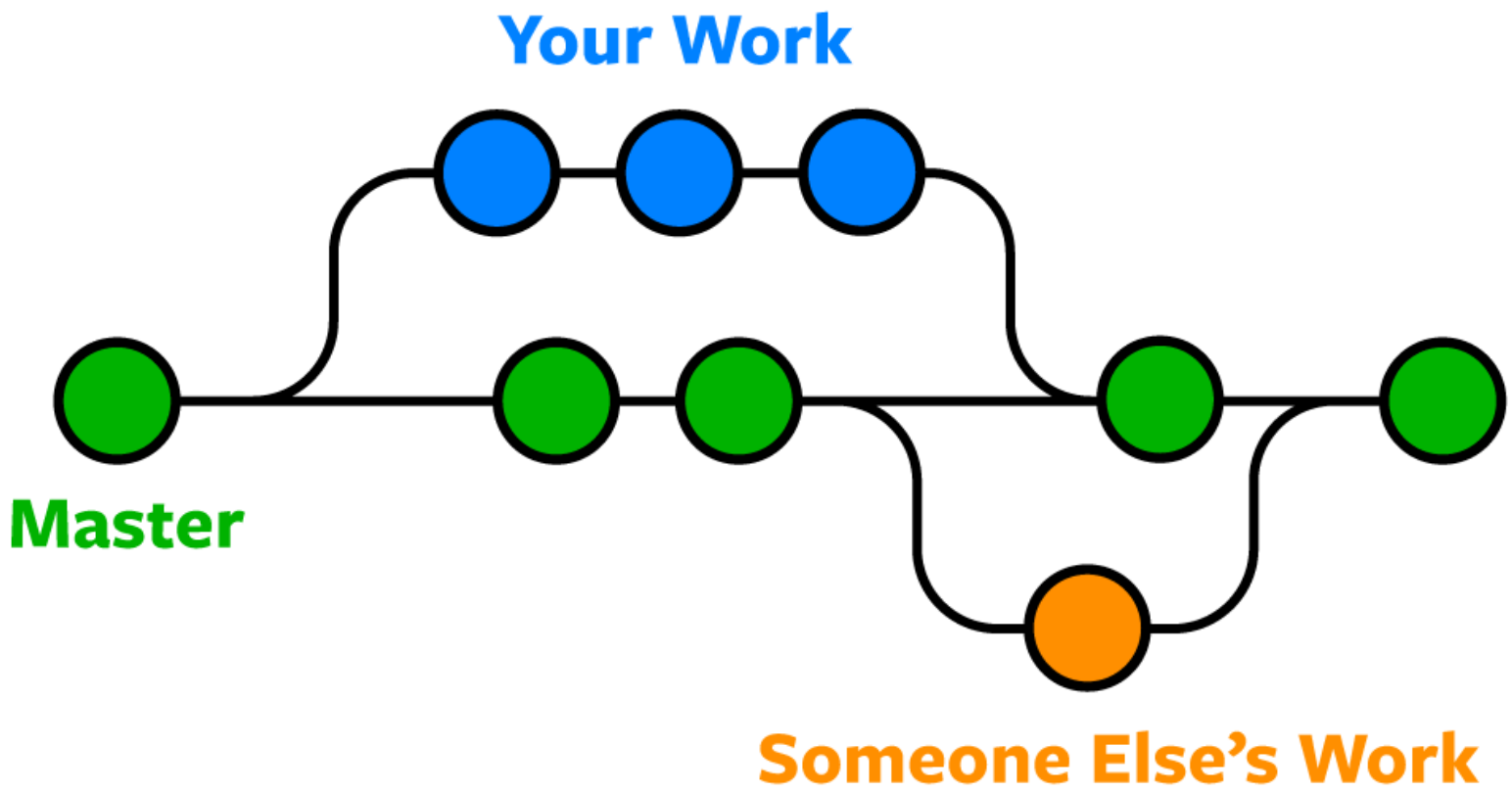
- Think of a tree. What is a branch in a tree?
  - An offshoot of the trunk of a tree.
- What is a branch in Git?
  - The default branch in Git is the ***master/ main*** branch (It is like the trunk of a tree).
  - Very simply put, a branch is a detour you make from the main path of development.

Source: https://www.rainforest-alliance.org/wp-content/uploads/2021/06/kapok-tree-profile-1.jpg.optimal.jpg

GIT BRANCHING: THE BASICS

- Why is branching needed?
  - Branching lets developers work independently inside a single code repository.
    - E.g., In product X, Amith works on feature A, while Binesh works on feature B.
    - If both have their code in just their local repositories, then no problem.
    - Now, it is a good practice to push your changes to the remote repository regularly.
    - What sort of problems can be expected when they do that?
      - **Merge Conflicts!**
  - Any solution to the above problem?
    - Yes, let each of them work on their own branch.

GIT BRANCHING: THE BASICS



Example on git branching. Source: https://www.nobledesktop.com/learn/git/git-branches

GIT BRANCHING: THE BASICS

- Why is branching needed? (continued)
  - Allows a developer to switch among different tasks easily.
    - Suppose while Amith is working on feature A, he gets a request to attend to a major issue in the product X.
    - If Amith is working on his local master branch, he will now have to,
      - Undo all the changes he did.
      - Fix the issue.
      - Restart work on feature A.

GIT BRANCHING: THE BASICS

- If Amith had been working on a separate branch even if he was on his local repository, he could have,
  - Stopped the work on the feature and switch back to master/ main.
  - Created new branch for the **"hotfix"** and work on it.
  - Pushed the hotfix branch to the remote repo (and merge to remote master).
  - Switched back to his feature branch and continue where he left off.

GIT BRANCHING: THE BASICS

- Referring to the previous scenario,
  - Using git branches during development time is clearly more efficient rather than not using any.
  - Using branches is a must for collaborations with multiple developers.
  - Whatever you do on a branch is isolated from the rest of the code – great when you want to experiment!
  - It is not a bad idea to use branching for individual projects too.
- Branching on Git is a very lightweight operation. Therefore, use it whenever possible.

HOW DO WE CREATE A BRANCH ON GIT?

- Very simple.
  - *git branch <branch_name>* - just creates the branch based on your current branch.
  - *git checkout <branch_name>* - checks out the branch.

OR

  - *git checkout –b <branch_name>* - creates the branch, and switches to the new branch (checks it out) from your current branch immediately.

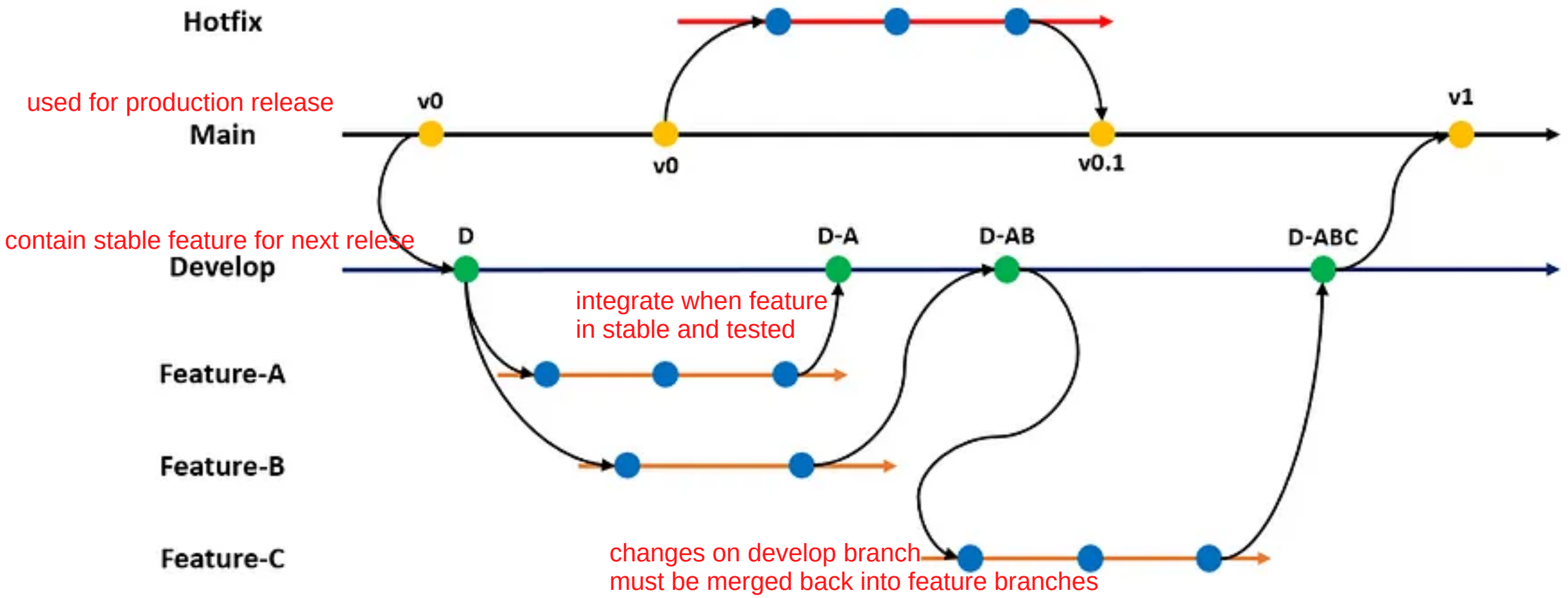
WORKFLOWS (BRANCHING STRATEGIES)

- Git is **very flexible** when it comes to how-to branch. There is no one standard method, but some well accepted how-to branch 'recipes' are there.
- They are called workflows or branching strategies.
  - A Git workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner.

How developers should create branches
How they should merge code
How to collaborate efficiently
Keeping the codebase clean and stable

- Some popular workflows are,
  - **GitFlow**
  - **GitHub Flow**
  - GitLab Flow
  - Trunk Based Development (goes with CI/ CD)

GITFLOW WORKFLOW



Source: https://betterprogramming.pub/git-branching-strategy-for-better-team-collaboration-aacb5f235d05

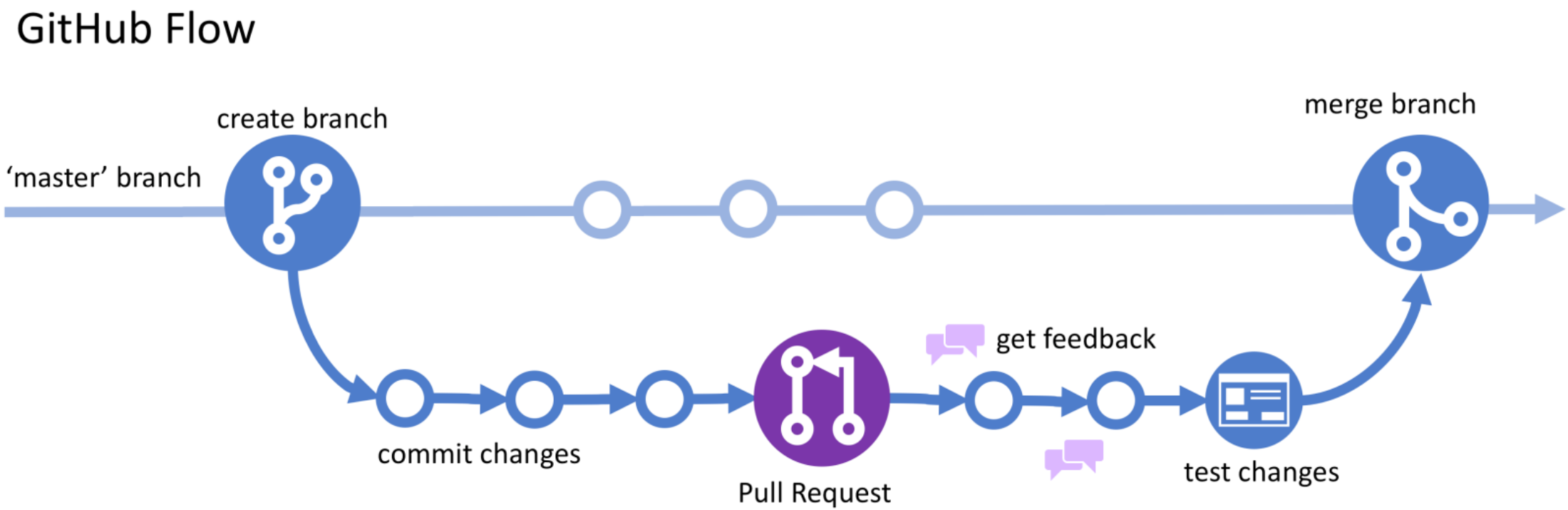
GITFLOW WORKFLOW

- GitFlow is very comprehensive.
  - [YouTube] GitFlow Workflow
  - Can become unnecessarily complex for small projects.
- It is more suited to Enterprise grade software development efforts.
  - Requires enforcement of the practices for the proper functioning of the workflow.
  - For most use cases, it is overkill.

GITHUB WORKFLOW

- A simpler alternative is preferable for most use cases.
  - One such workflow is **GitHub Flow** which can be used by anybody.
  - [YouTube] GitHub Flow

THE GITHUB FLOW



Source: https://build5nines.com/introduction-to-git-version-control-workflow/

GITHUB FLOW

- This is a GitHub based simple workflow.
  1. Create a new feature branch from the main (master) branch.
  2. Make the necessary changes to the feature branch – continue until you think your feature is ready.
  3. Once you think the feature is done, ask for feedback from the other developers – create a Pull Request (PR) for this.
  4. Address their review comments received on the Pull Request (PR).
  5. Merge your Pull Request (PR) to the main branch.
  6. Delete your feature branch.

Refer the official documentation here.

PULL REQUESTS (PR)

- A Pull Request (PR) is used to propose a new change to a repository (usually to the main branch).
- Usually, the proposed changes are in a feature branch, which ensures that the main branch will only contain code which is tested and working.
- The other collaborators will have to verify that the proposed change (PR) is OK first.
- If the PR is OK, they will accept it. Or else, they will suggest changes until it is acceptable.
- Once the PR is accepted, the changes in the feature branch are merged automatically to the main branch, bringing the proposed change to the main branch.
- Find the official documentation on Creating PRs here.

GIT BRANCHING: BEST PRACTICES

1. Create a branch for each feature. Do not work on multiple features in a single branch.
2. Ideally, only one person should work on one feature branch.
3. Feature branches should be short-lived. They **should not exist** once the feature is complete.
4. Delete the local and remote feature branch after merging to the master branch (see point 3).

GIT BRANCHING: BEST PRACTICES

5. Merge early and often to avoid merge conflicts.
6. Keep the master branch clean. Only *production ready* code should be in the master branch. It is better to **avoid** working directly on the master branch.
7. Use a good naming convention to name your feature branches. Everyone should stick to the same naming convention.
   - feature/<username>/task-name-in-lowercase-spaces-replaced-with-dashes
     - feature/vishan.j/awesome-feature-x-to-our-app

GIT GENERAL BEST PRACTICES

1. Make small, incremental changes.
   - Don't make huge changes in one go. If you have a big feature to add, break it down to smaller features and add one at a time.
2. Keep commits atomic.
   - It is better if each commit is focused on one part of the feature. This will minimize the damage if you must revert a commit.
3. Commit often.
   - Commit your changes often. Even if the work is not done.
4. Pull the changes in origin/main branch before you create a new branch from your local/main.
   - This ensures that the new feature branch is created with all the latest changes in master. This reduces the possibility of merge conflicts later.

GIT GENERAL BEST PRACTICES

5. Push local feature branch commits to Remote often.
   - Even if it is an unfinished feature, push it. That's good for safekeeping.
6. Use branches.
   - See the section on branching for a comprehensive set of reasons.
7. Write good commit messages.
   - Make the lives of other Software Engineers, DevOps engineers, QA engineers easier.
8. Select one branching strategy (Git Workflow) and stick with it.
   - Select one that suits you and use it well.

SUMMARY

- Git branching
  - The basics
  - How do we create a branch on git?
  - Workflows (Branching strategies)
    - GitFlow
    - The GitHub Flow
    - Pull Requests (PRs)
  - Best practices for branching
- Git general best practices

ADDITIONAL TOPICS

Read on these topics to be up to date on some current trends.

- Continuous Integration and Continuous Deployment (CI/ CD).
- Compare Trunk based development workflow with GitFlow workflow.
  - Understand when to use which.
- Read on Pull Requests.
- Read on Git Rebase as an alternative to Git Merge.

REFERENCES

1. Pro Git - Scott Chacon and Ben Straub [Free eBook]
2. Dangit, Git!?! [Git quick reference]
3. Branch in Git
4. What Are the Best Git Branching Strategies
5. 10 Git Branching Strategy Best Practices
6. Trunk-based Development vs. Git Flow
7. How to Use Git and Git Workflows – a Practical Guide

THANK YOU

VISHAN.J@SLIIT.LK