VERSION CONTROL

Definition

- It is a software tool that helps a development team manage changes to the source code of a project over time.
- It keeps track of every modification to the code, allowing rollbacks in case of a mistake done to the code.
- Allows for easy synchronization in between different developers.

DevOps Toolchain

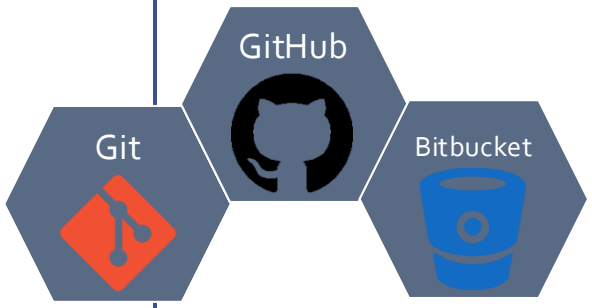
Plan

- Planning
- Requirement Gathering
- Project Management

Develop

- Issue Tracking
- Source Control
- Dev Env
- CD

Test



- Continuous Testing

Release

- Deployment
- Config Management
- Containers
- Scripting

Operate

- Monitoring
- Issue Discovery

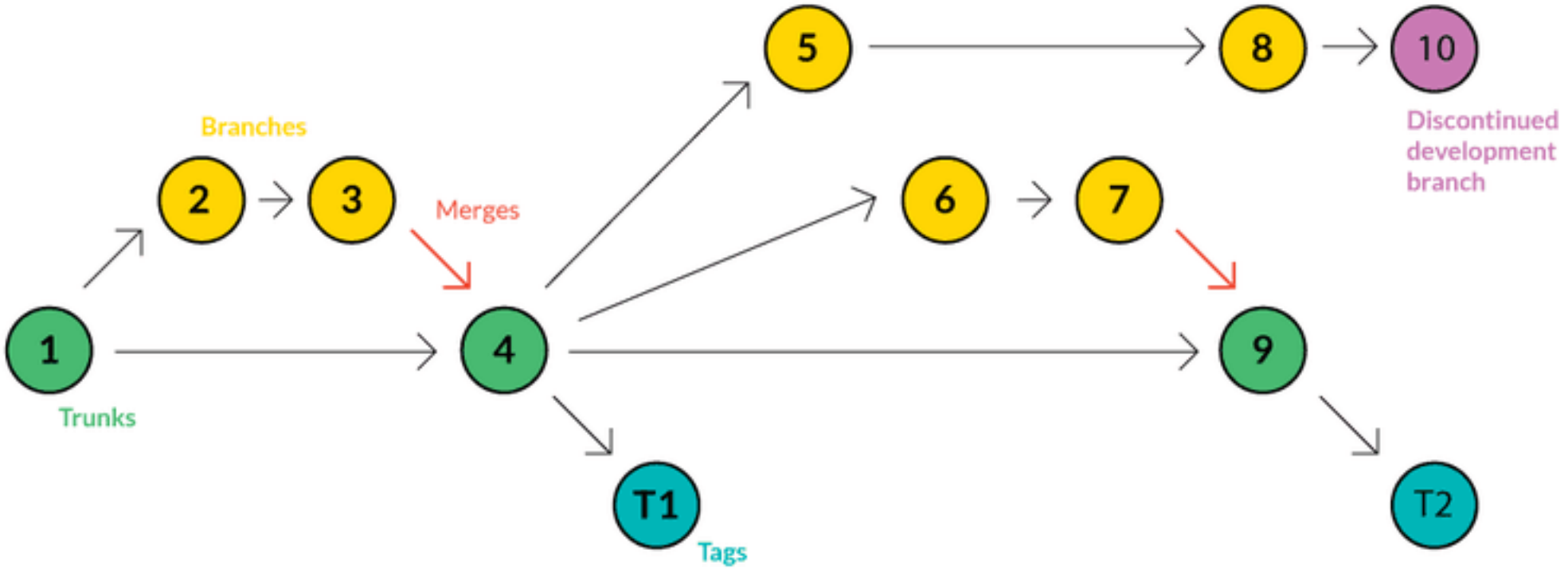
What is Version Control?

- A system that records changes to files over time.
- Allows you to revert to previous versions, track history, and collaborate safely.
- Essential for teamwork, CI/CD, and maintaining production stability.
- Examples: Git, Mercurial, Subversion, CVS (Git being most common).

Core Concepts

- Repository (Repo)
  - A directory where the version control system tracks all changes.
- Commit
  - A snapshot of changes made to the codebase at a point in time.
- Branch
  - A parallel line of development that isolates work from the main code.
- Merge
  - Combines changes from one branch into another.
- Remote
  - A shared version of your repository (e.g., GitHub, GitLab, Bitbucket).

Overview


5
8
10
Discontinued development branch
Branches
2
3
Merges
6
7
1
4
9
Trunks
T1
Tags
T2

Initializing a Git Repository

```
# Create a new folder and initialize Git
mkdir project && cd project
git init

# Add files and commit changes
echo "print('Hello, world!')" > app.py
git add app.py
git commit -m "Initial commit"

# Check status and history
git status
git log
```

How Version Control Works

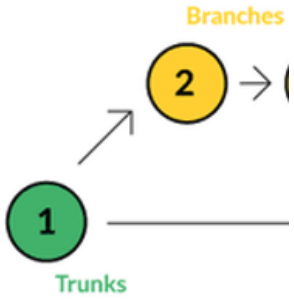
- Tracks file changes by comparing snapshots (diffs) between versions.
- Each commit creates a unique hash (SHA) representing a snapshot of the repo.
- Changes are stored efficiently by referencing previous versions.
- You can navigate the project's full history or revert to earlier commits.

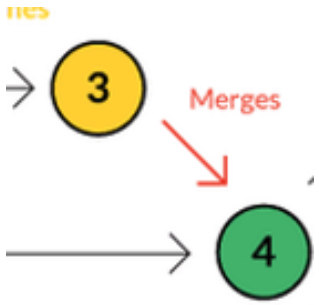
Branching and Merging

```
# Create and switch to a new branch
git checkout -b feature-login

# Make changes and commit
echo "def login(): pass" >> app.py
git add app.py
git commit -m "Add login function"

# Merge branch into main
git checkout main
git merge feature-login
```





Why Use Branches?



Allows developers to work independently on features or fixes.



Keeps the main (production) branch stable.



Supports workflows like Git Flow or GitHub Flow.

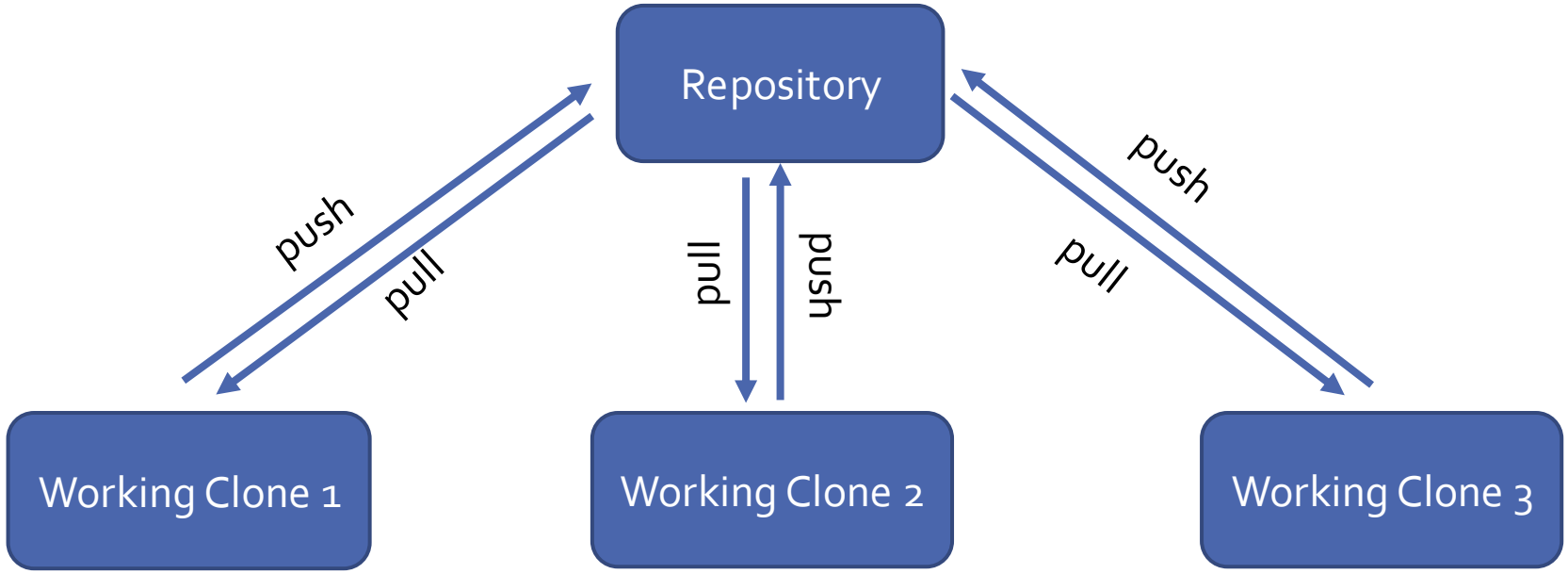


Facilitates peer review and continuous integration.

Note

- **Version Control =/= Online repository**

Remote Repo (Repository)



Collaborating with Remotes

```
# Add a remote repository
git remote add origin https://github.com/user/project.git

# Push local commits to remote
git push -u origin main

# Fetch and pull updates
git fetch origin
git pull origin main
```

Collaboration Workflow



Clone the repository from a remote (GitHub, GitLab).



Create a new branch for your feature or bug fix.



Commit and push your changes to the remote.



Open a Pull Request (PR) or Merge Request (MR).



Team reviews, approves, and merges the change into the main branch.

Commit Best Practices

- Write descriptive commit messages (e.g., 'Add user authentication route').
- Group related changes together; avoid large unrelated commits.
- Use present tense: 'Add', 'Fix', 'Update'.
- Commit early and often for smaller, manageable history.

How Git Stores Data

- Git doesn’t store file differences (deltas); it stores snapshots.
- Each commit points to a tree of files and a parent commit.
- Branches are pointers (references) to specific commits.
- Efficiently handles merges by using commit ancestry graphs (DAG).

Viewing History and Differences

```
# Show commit history
git log --oneline --graph --decorate --all

# Compare changes between commits
git diff HEAD~1 HEAD

# View changes in a specific file
git diff HEAD app.py
```

Undoing Changes

- Revert
  - Creates a new commit that undoes a previous one (safe for shared history).
- Reset
  - Moves the HEAD pointer to a specific commit (use carefully).
- Checkout
  - Temporarily switch to another commit or branch for inspection.

Undo Examples

```
# Undo last commit but keep changes
git reset --soft HEAD~1

# Discard local changes
git checkout -- app.py

# Create a new commit that reverts a bad one
git revert <commit_hash>
```

Advanced Concepts (Overview)

- Rebasing: Rewrites commit history to create a linear progression.
- Cherry-pick: Apply a specific commit from another branch.
- Stashing: Temporarily saves uncommitted changes for later use.
- Tagging: Marks specific commits as releases or milestones.

Some Available Version Control Systems

- git: https://git-scm.com/



- subversion: https://subversion.apache.org/



- mercurial: https://www.mercurial-scm.org/



Some Available Online Repositories

- GitHub: https://github.com 
- GitLab: https://about.gitlab.com 
- BitBucket: https://bitbucket.org 

References

- Git Handbook: https://guides.github.com/introduction/git-handbook/
- Git Cheat Sheet: https://training.github.com/downloads/github-git-cheat-sheet/
- Learn Git Branching: https://learngitbranching.js.org/