Git & GitHub Workshop

Kazuharu Yanagimoto

September 23, 2022

Why Do People Use Git?

📝 Recording

take logs of all coding activity of you and your collaborators

🕒 Restorable

go back to a previous version of codes

🔍 Comparable

focus on the change in the codes, and detect bugs

🔀 Branch

separate things completed and under development

Is Git Easy?

No. I am sorry.

- Git has many commands with many options
- Need some knowledge to recover from a trouble
- Git allows various styles to use, which are different across people and organization

I propose

- First follow my workflow, which requires the minimum knowledge
- Once you’re comfortable with it, learn the detail

Is GitHub Git?

Git Is a Version Control Tool

- Command Line App
- Works in a local machine

GitHub Is a Web-Service

- Publish Codes
- Collaboration

CUI vs GUI?

GUI Applications for Git & GitHub

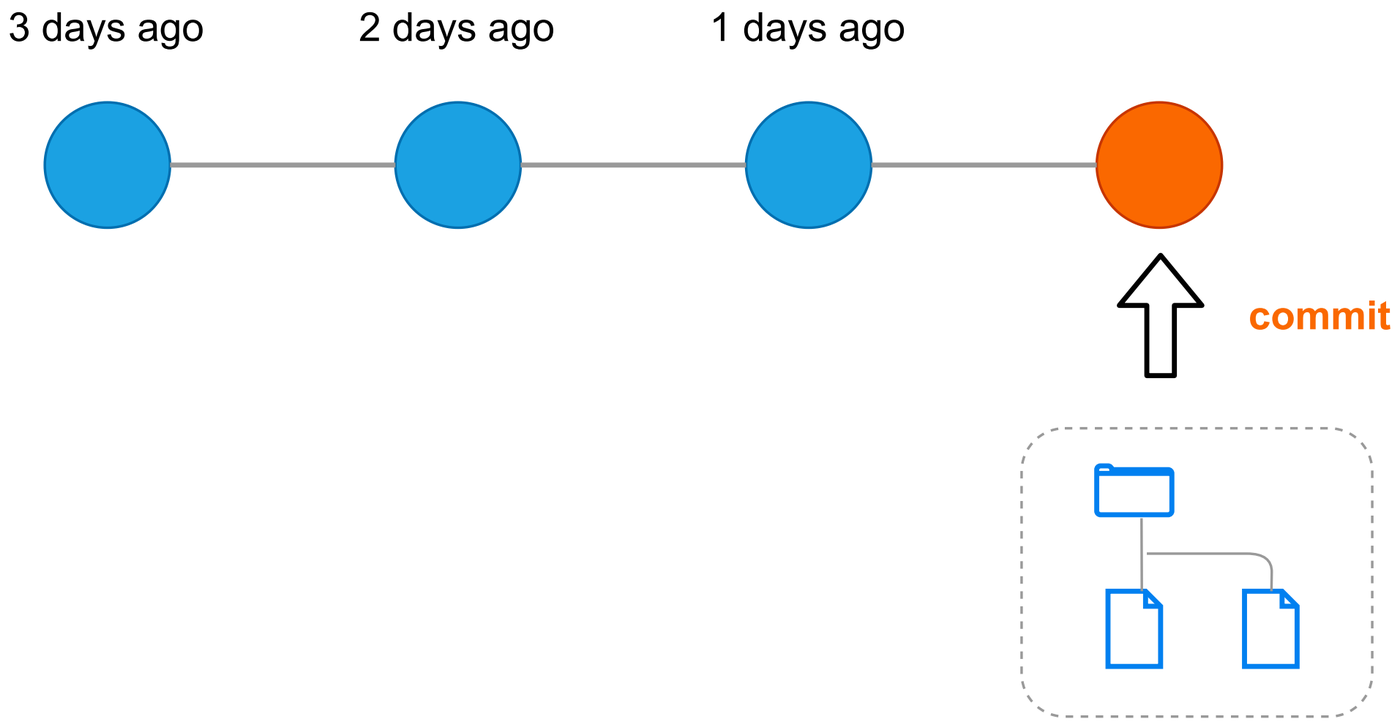
- We are going to use VSCode
- Other options: GitHub Desktop, Fork, RStudio

I recommend to learn CUI first

- CUI knowledge is necessary to use GUI
- Some troubles can be solved only through command line
- DVC (explained later) only works with command line

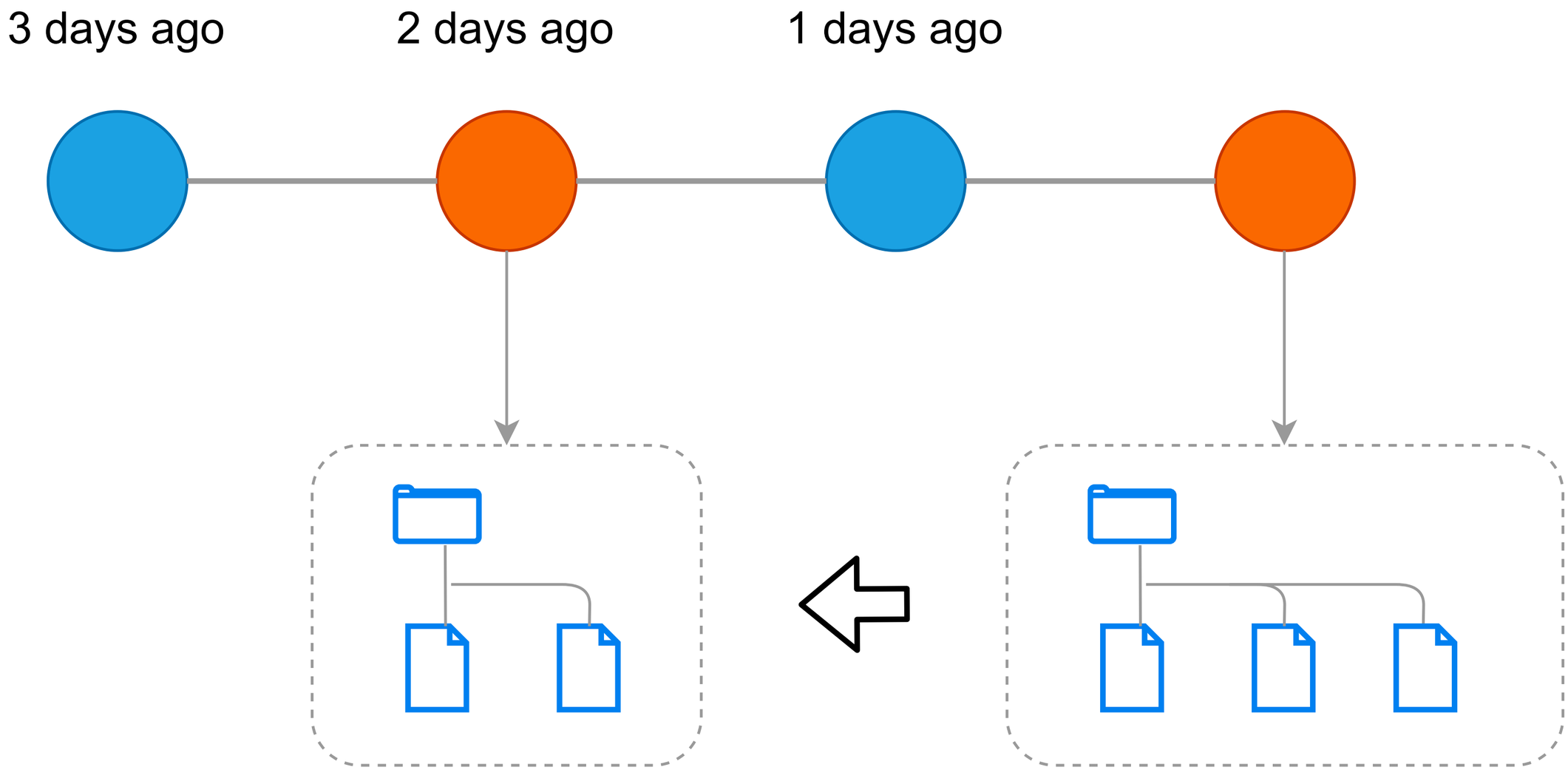
Basics of Git

Commit Is a Save Point!

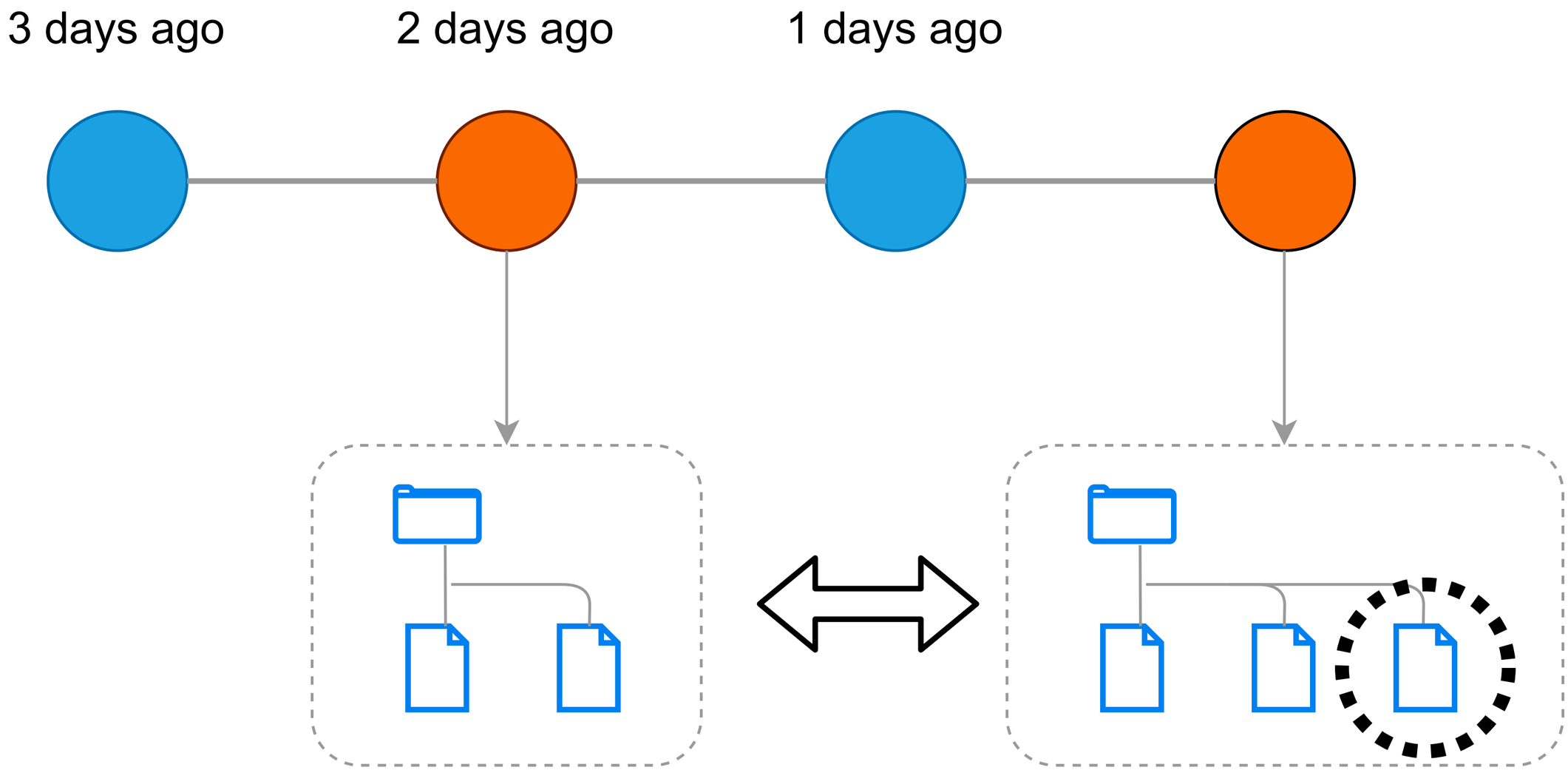


- “The purpose of Git” is to create a COMMIT
- A commit is a save point of a state of the project folder

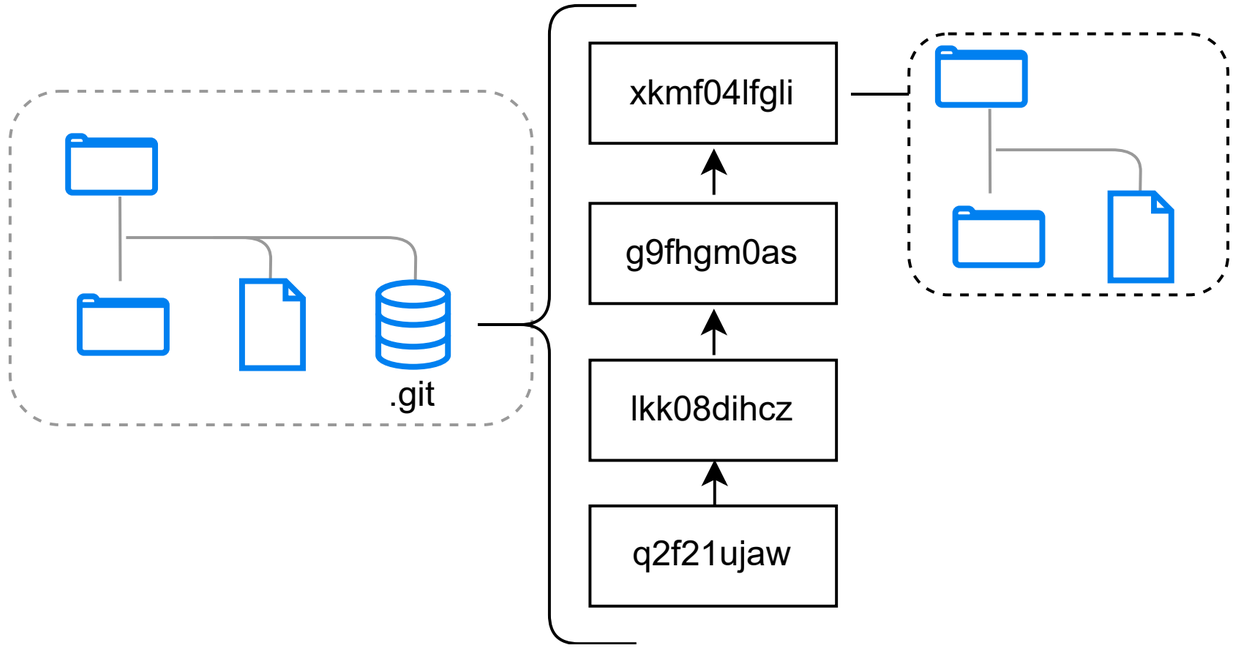
You Can *Go Back* to Any Commit



You Can *Compare* Any Two Commits

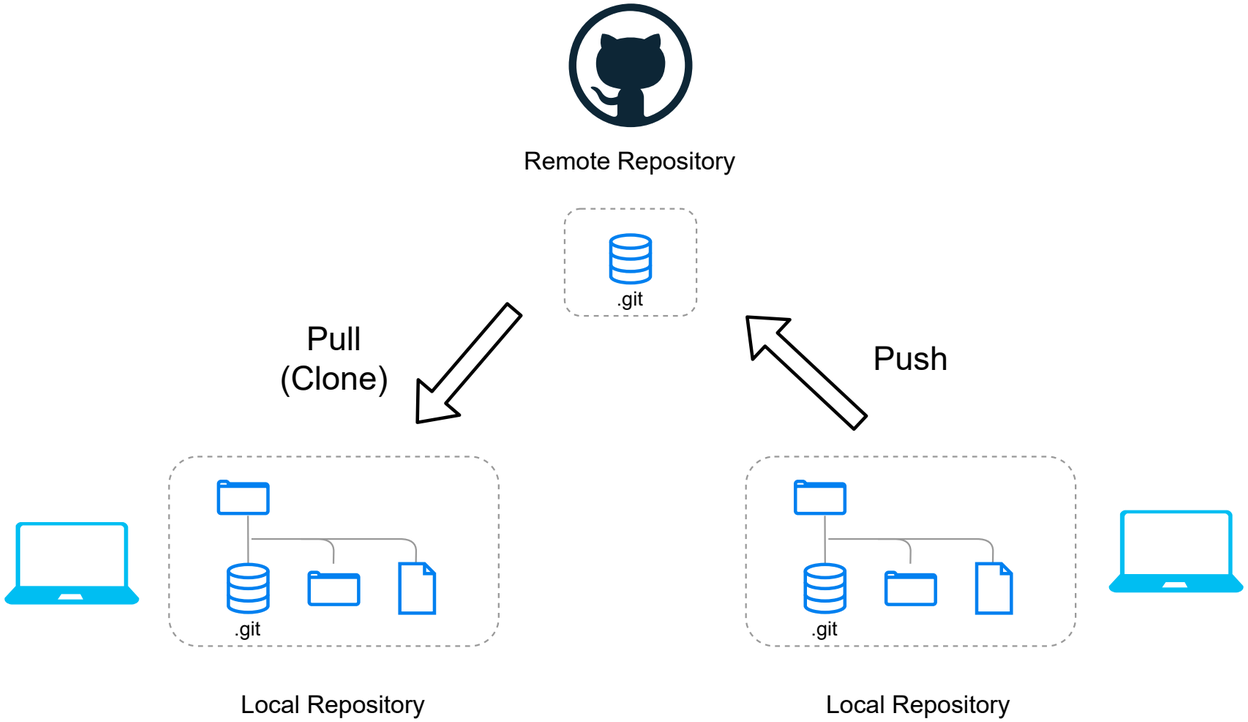


What Is *Commit* in a Computer?



- If you init a git project on a folder, a *hidden* folder `.git` will be made
- When you create a commit, the state of the folder will be stored in `.git` folder
- The name of commit is just a random hash string

Local & Remote Repository



CUI and GUI Operations

Stage

- When you modify some files, you can create a commit only for specific changes
- This choosing process is called, **Staging**

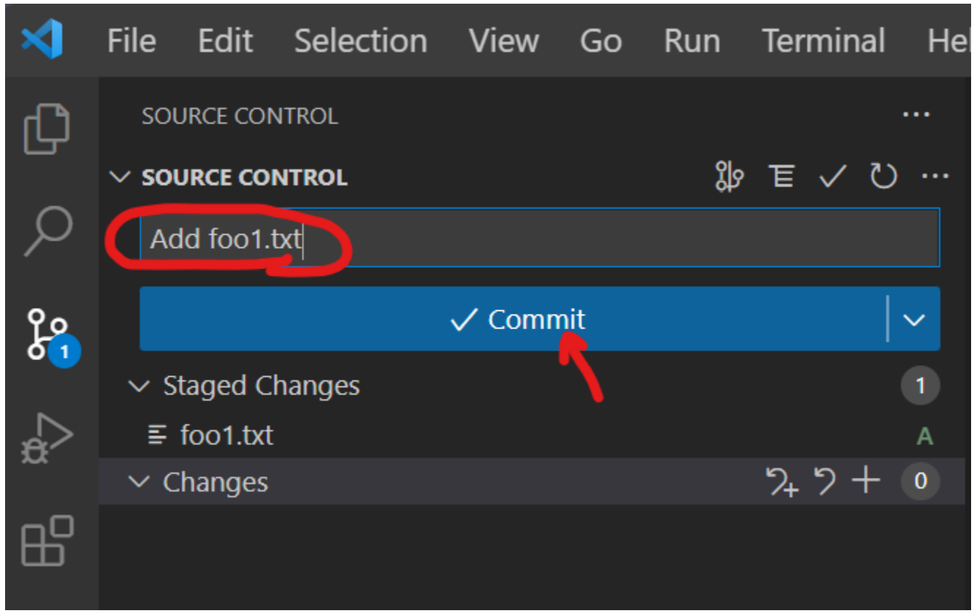
Commit

- You have to add a message for each commit

Command Line

```
1  git commit -m "YOUR COMMENT"
```

VS Code

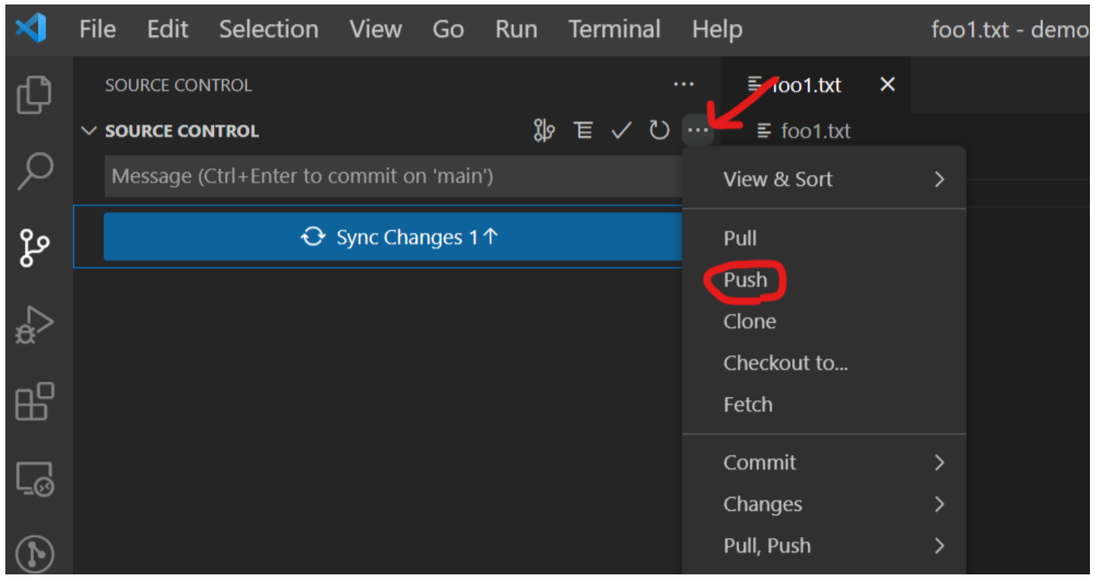


Push

Command Line

```
1 git push origin main
```

VS Code

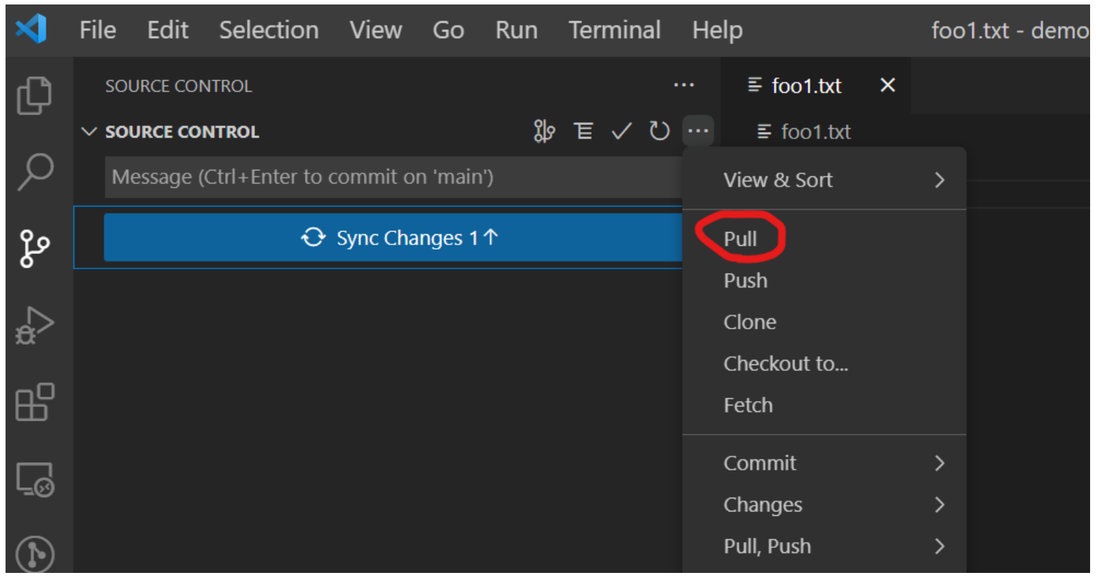


Pull

Command Line

```
1 git push origin main
```

VS Code



For now, forget about `origin main`

Handson 1: First Commit and Push

1. Go to GitHub webpage and create a repository

1. Choose the private repository (if it is for research)
2. Choose the language you are using for .gitignore
3. Add README
4. No need to choose license (because it is private)

2. Clone it to your local machine

1. The easist way is to open a new window of VS Code and choose “clone a repository”
2. You can also do `git clone URL_OF_REPOSITORY`

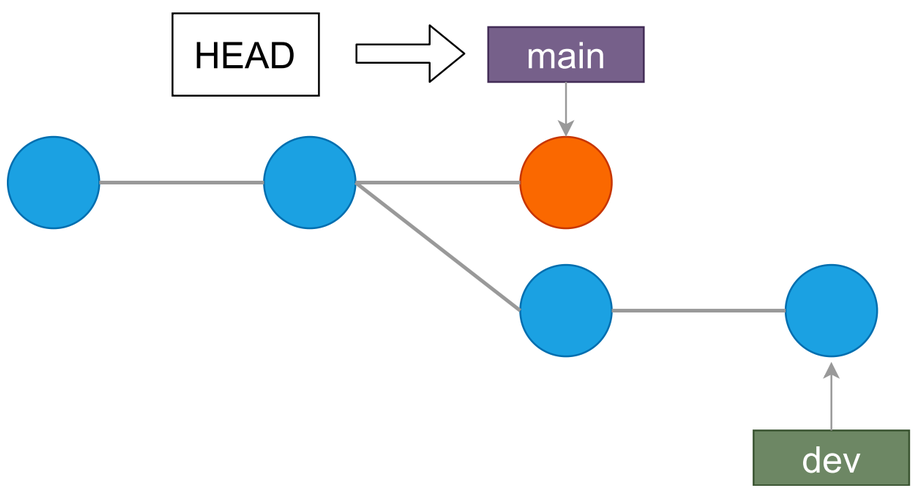
3. Create a Commit and Push It

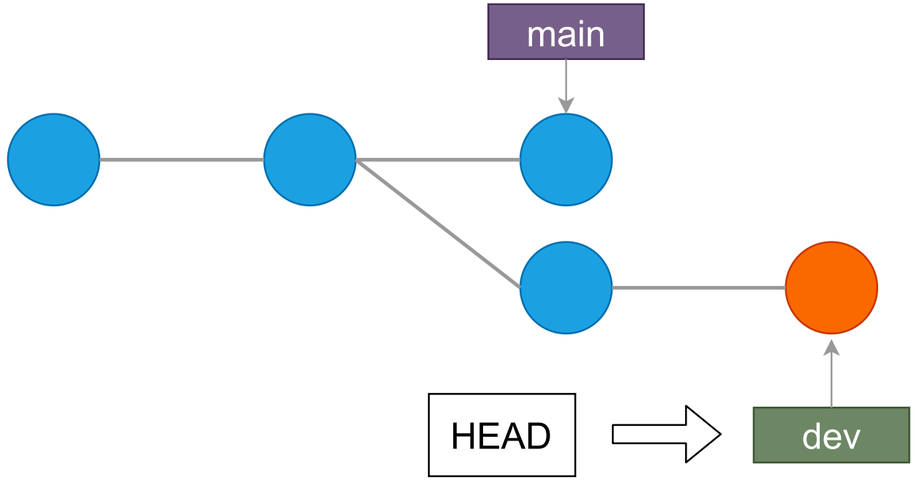
1. Create a text file (say foo1.txt) with a random text (say “Hello Git!”)
2. Create a commit (don’t forget to stage it before)
3. Push to the remote repository

Branch, Merge, and Conflict

HEAD and Branch

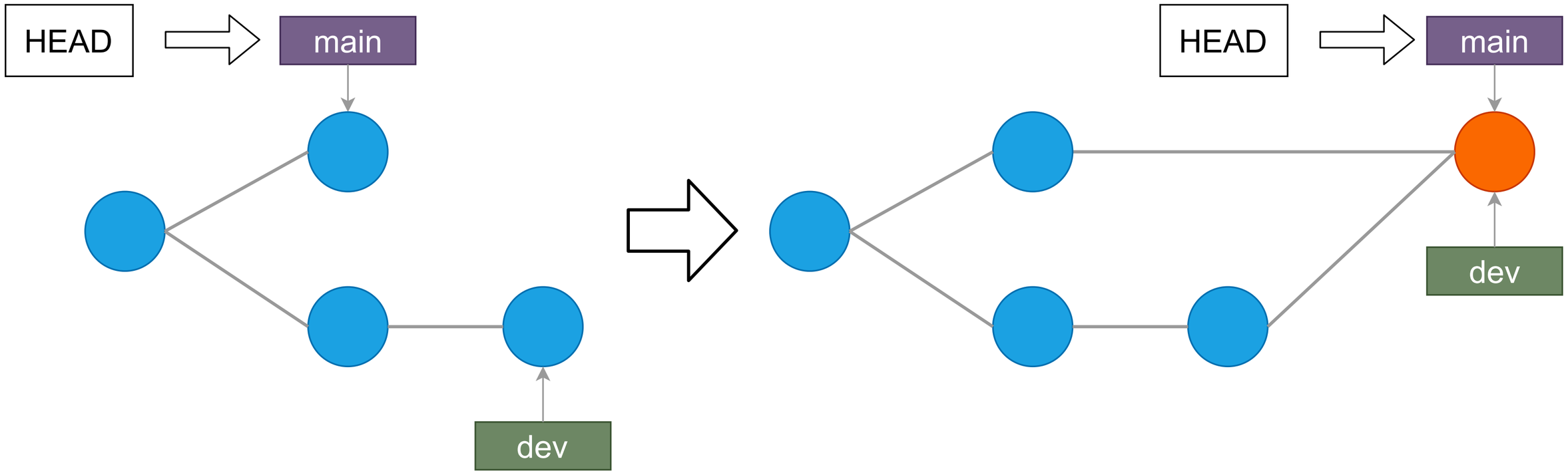
- Branch: a label of a commit
- HEAD: the branch you are seeing





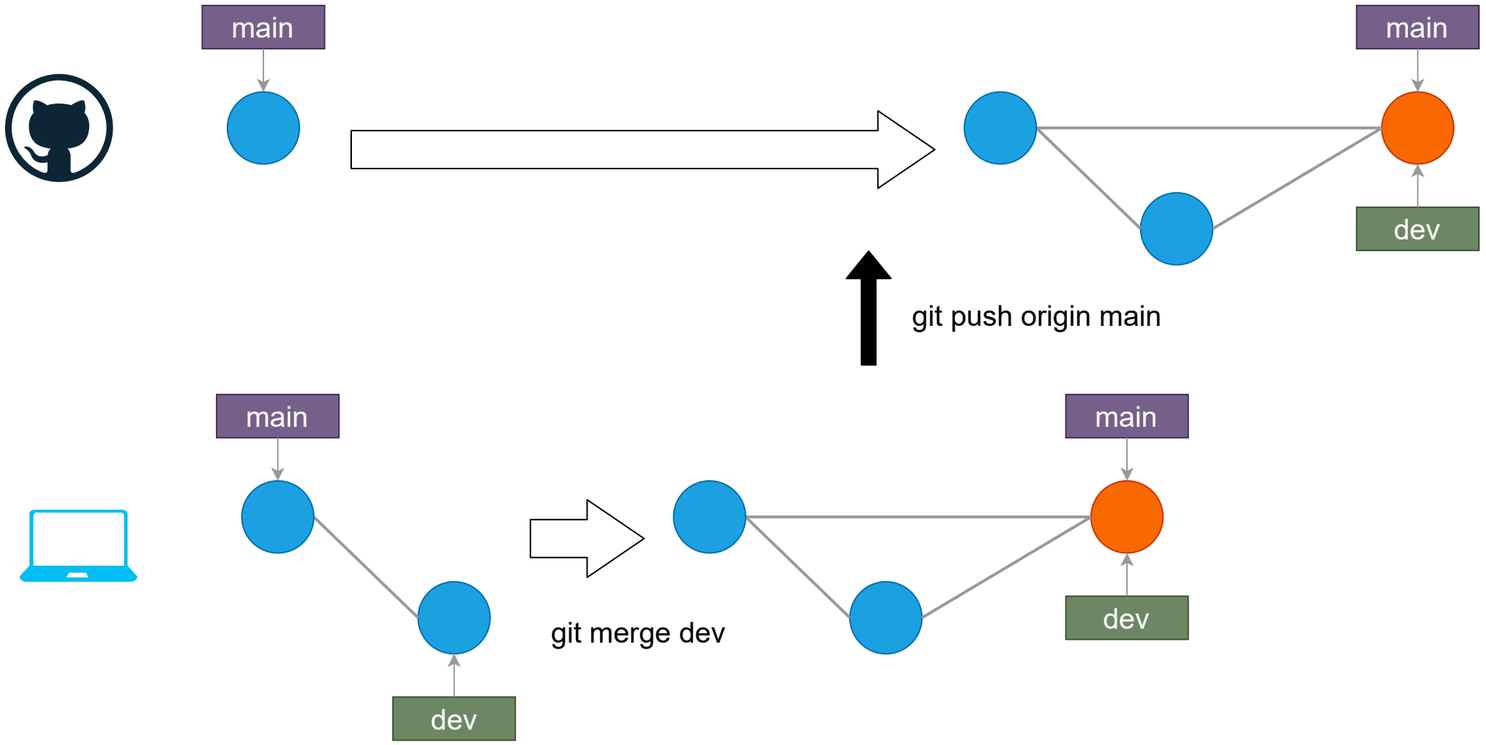
- To create a branch, `git branch BRANCH_NAME`
- To move the HEAD, `git switch BRANCH_NAME`

Merge



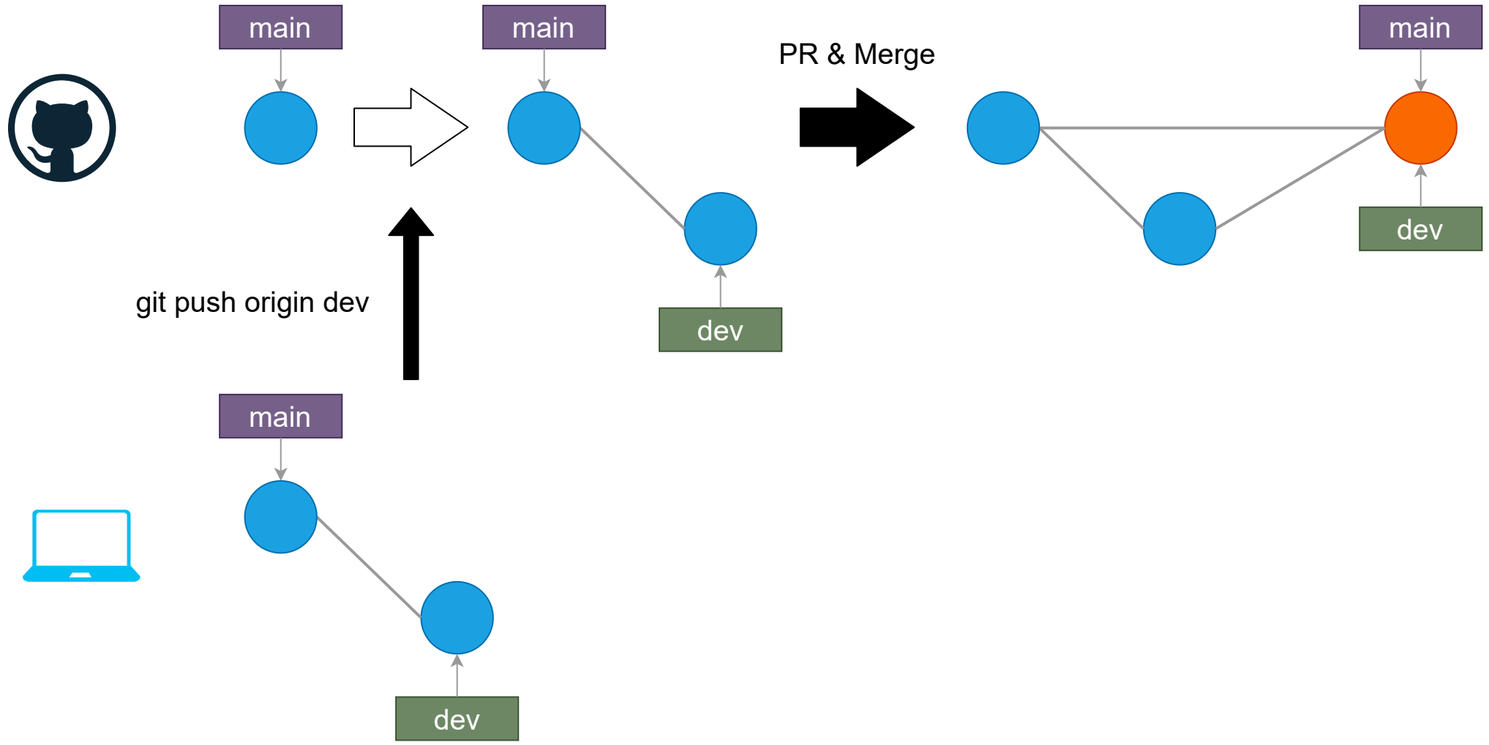
- You can merge a branche into HEAD by `git merge BRANCH_NAME`

1. Merge at Local Repository



- Merge it to main on your local repository
- Push the local main into the remote main
- You can use it for solo project

2. Merge at Remote Repository



- Create a **Pull Request** on GitHub Repository
- Decide to merge it into main or not on GitHub
- Used for collaboration (but I use it even for solo project)

Questions about Branches

Why Do We Use Branch?

- Keep **main** branch clean and work correctly
- Easy to detect a bug (because “main” works perfectly)
- Collaborators create a branch from main and work on it

When Should I Create a Branch?

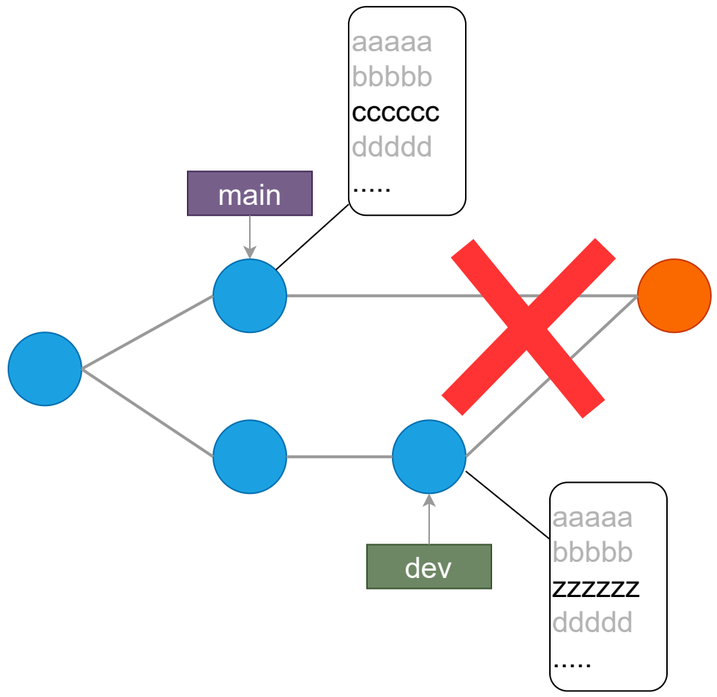
A simple suggestion is *main-dev workflow*

- You only work on **dev** branch (with a collaborator, I use my name for the developing branch)
- When you finish a meaningful chunk of works, cleanup uneccessary files and merge **dev** into **main**

I merge into *main* just before the meeting with my supervisor or collaborators.

Conflict

A different code is written in the same line of two branches in merging



Produced File

```
1 aaaaa
2 bbbbb
3 <<<<<<< HEAD
4 cccccc
5 =======
6 zzzzzz
7 >>>>>>> dev
8 ddddd
```

Most conflicts can be avoided by a rule prohibitting members from co-editting the same file

Solve a Conflict

1. Delete the lines you don't need and save the file
2. Create a commit for this merge and solving the conflict

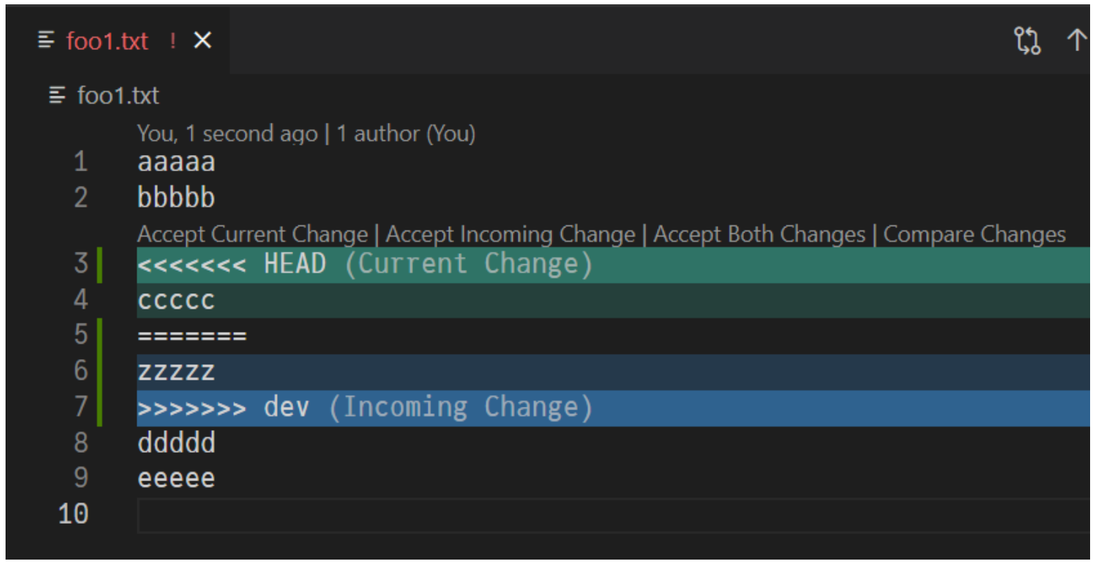
Regular Editor

```
aaaaa
bbbbb
<<<<<<< HEAD
cccccc
======
zzzzzz
>>>>>>> dev
ddddd
```

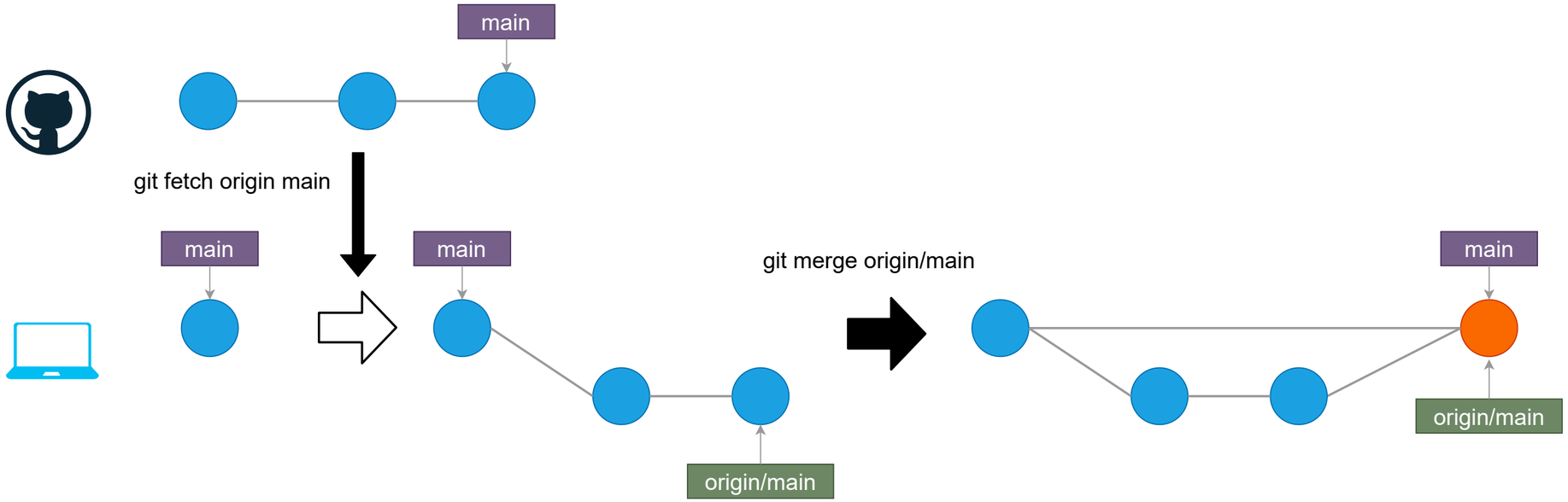
\[\Downarrow\]

```
aaaaa
bbbbb
zzzzz
ddddd
```

VS Code



Pull = Fetch + Merge



- Fetch : download a branch from the remote repository (origin) to a local machine, named `origin/BRANCH_NAME`
- `git pull origin main` is equivalent to `git fetch origin main && git merge origin/main`
- Because it is a merge, `git pull` could return a CONFLICT!

Handson 2: Two Types of Merge

1. Local Merge

1. Create a new branch “dev”
2. Switch the HEAD into *dev*
3. Create some commits
4. Switch HEAD into *main*
5. Merge *dev* into *main*
6. Push it to *origin/main*
7. Delete (local) *dev* branch

2. Remote Merge

1. Create *dev* branch
2. Switch to *dev*
3. Create some commits
4. Push it to *origin/dev*
5. Go to GitHub and create a Pull Request
6. Merge *dev* to *main* and delete the remote *dev* branch
7. Go back to local and switch to *main*
8. Pull main branch and delete (local) *dev* branch

Handson 3: Three Types of Conflicts

1. Local Conflict

1. Switch to *dev* and create `foo1.txt` with a line of text (e.g. “Tortilla sin Cebolla”) and create a commit
2. Switch to *main* and create `foo1.txt` with a line of different text (e.g. “Tortilla con Cebolla”) and a create a commit
3. Merge *dev* into *main*, which produces a conflict
4. Solve the conflict by choosing either of the lines

2. Pull Request Conflict

1. Switch to *dev* and create `foo2.txt` with a line of text (e.g. “But the Emperor has nothing at all on!”). Push it into *origin/dev*
2. Switch to *main* and create `foo2.txt` with a line of different text (e.g. “Oh! How beautiful are our Emperor’s new clothes!”). Push it into *origin/main*
3. Go to GitHub and create a pull request. Solve the conflict on GitHub

3. Pull Conflict

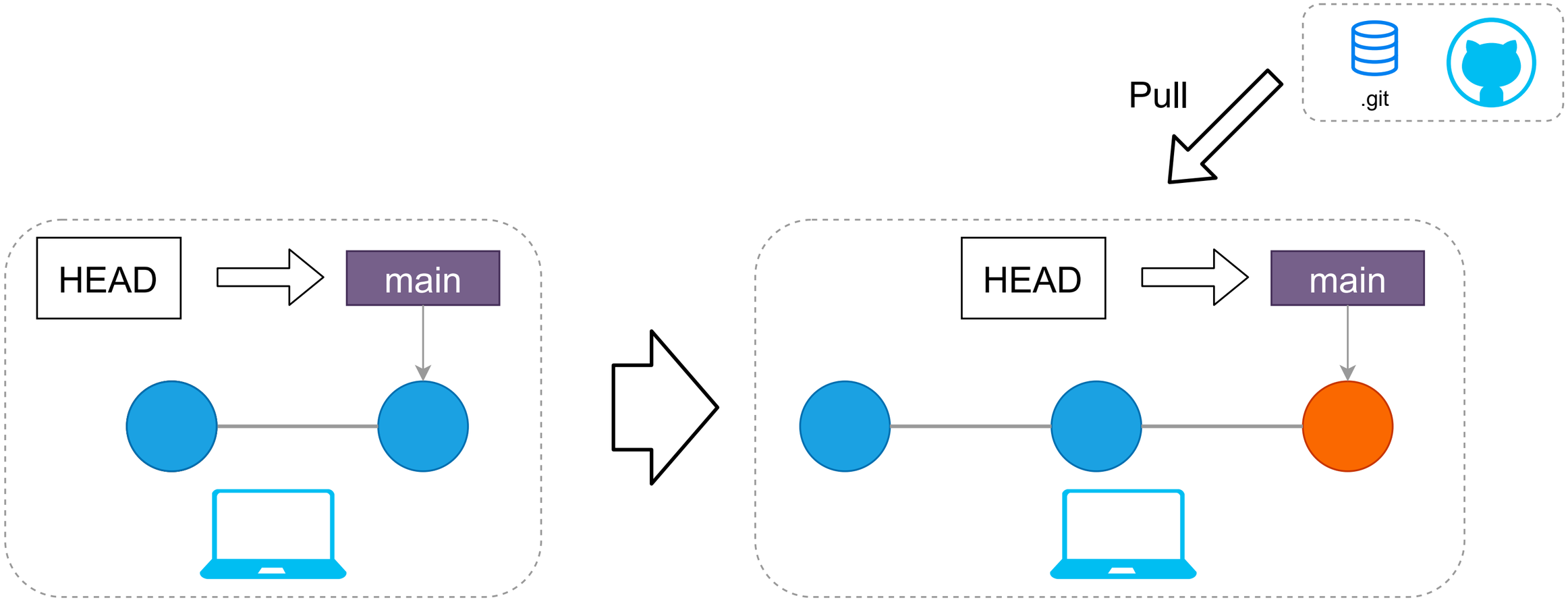
1. Switch to *dev* and create `foo3.txt` with a line of text (e.g. “Mountain of Mushroom”), create a commit, and push into *origin/dev*
2. Go to GitHub and merge *dev* into *main*
3. Go back to local. Switch to *main* and create `foo3.txt` with a different line of text (e.g. “Village of Bamboo Shoot”), create a commit
4. Pull from *origin/main*, which produces a conflict. Solve it.

Mountain of Mushroom

Village of Bambooshoot

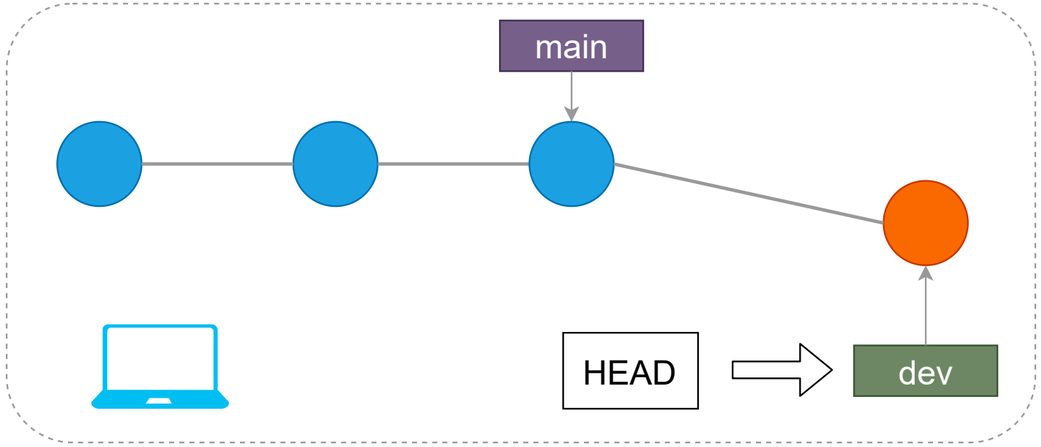
Git & GitHub Workflow

1. Sync Local Repository




```
1  git switch main
2  git pull origin main
```

2. Write Your Codes on Dev branch



1. Switch to a new developing branch

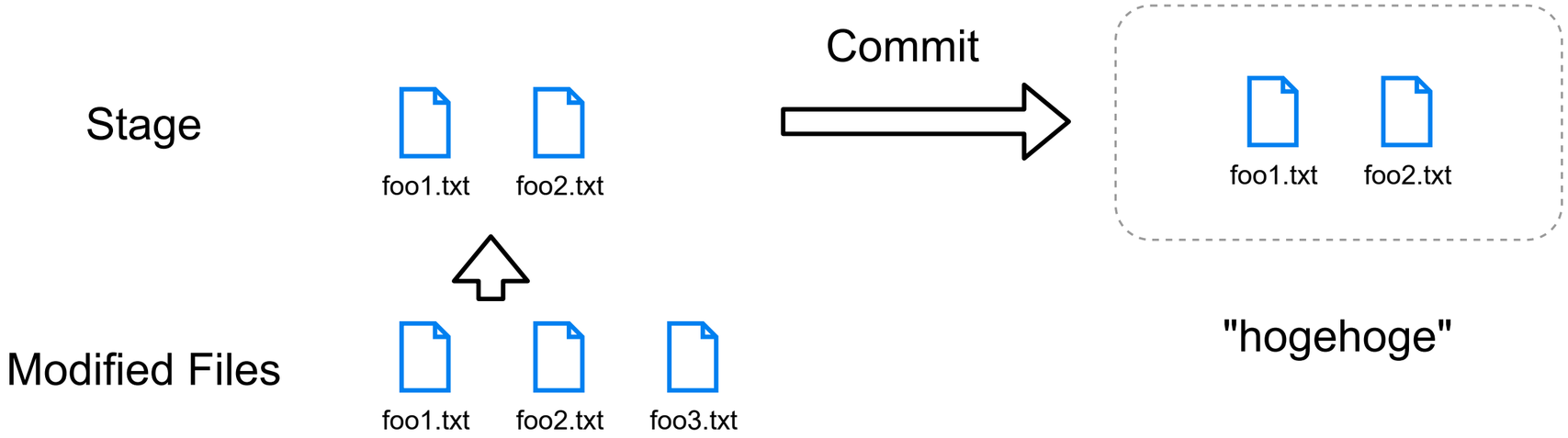
```
1  git branch dev
2  git switch dev
```

Or

```
1  git checkout -b "dev"
```

2. Write your codes

3. Commit



1. Stage files you want to commit

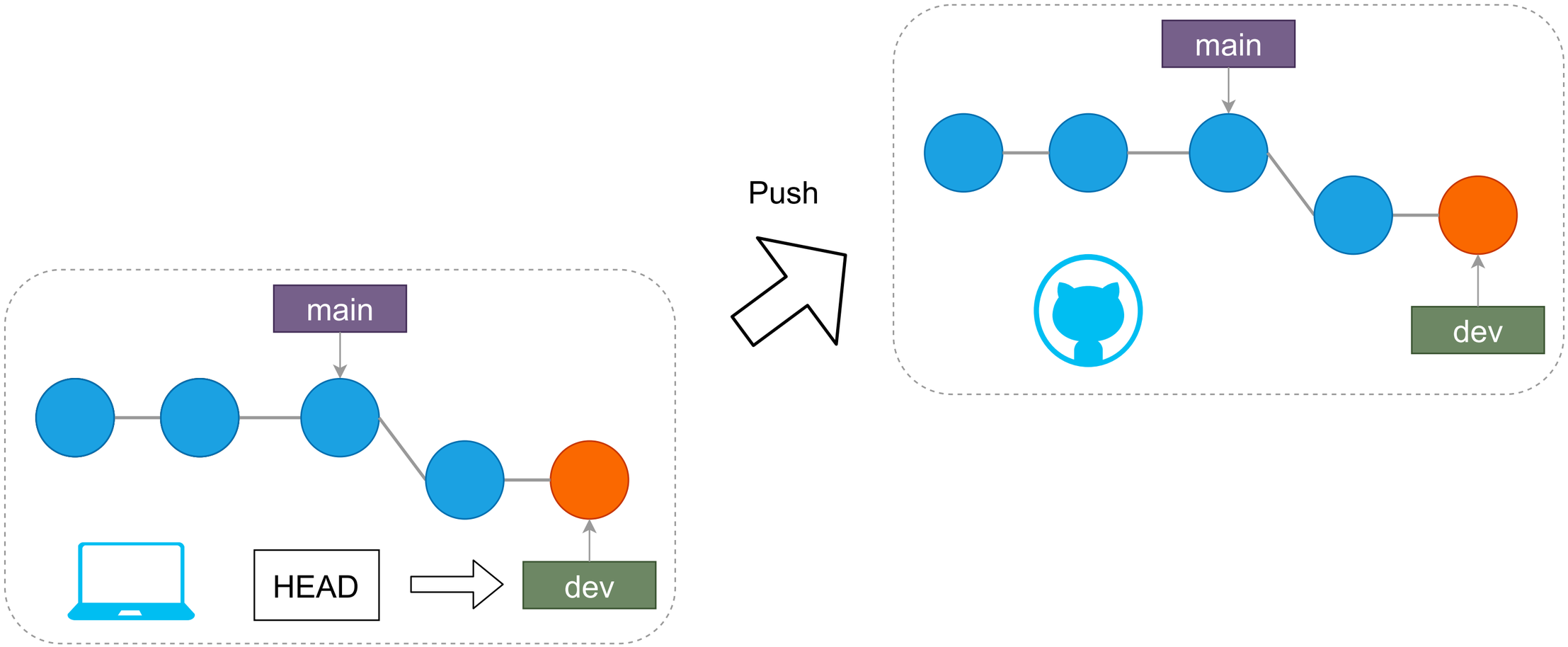
```
1  git add foo1.txt foo2.txt
```

If you want to stage all modified files, `git add .`

2. Commit

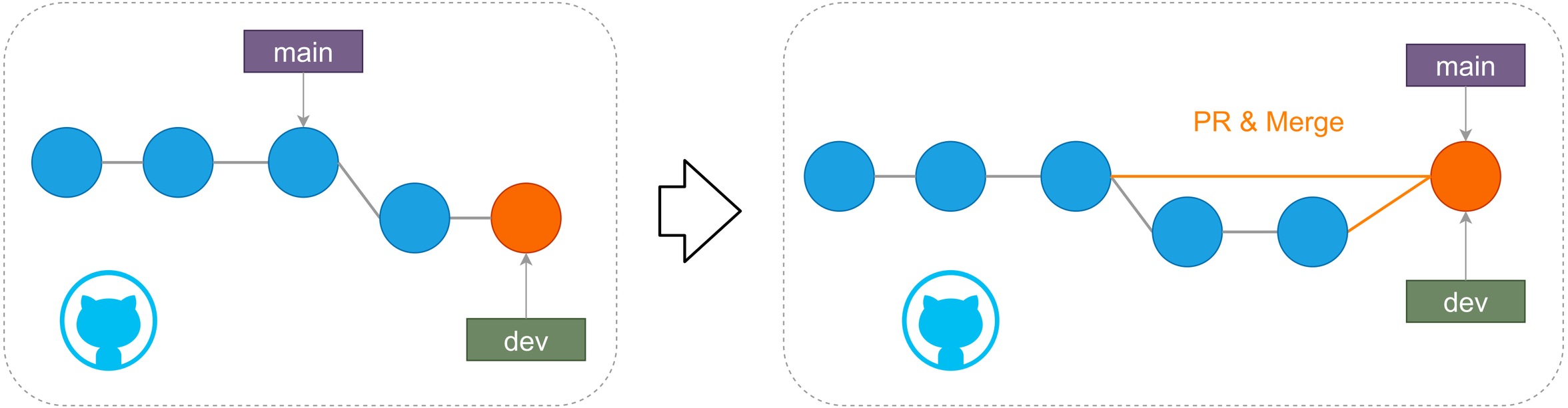
```
1  git commit -m "hogehoge"
```

4. Push to the Remote Repository




```
1  git push origin dev
```

5. Pull Request and Merge



I recommend you to delete the remote and local dev branch here.

Handson 4: Follow My Workflow

- Follow my workflow 1-5 with a few commits
- Try both command line and VS Code

GitHub with Data

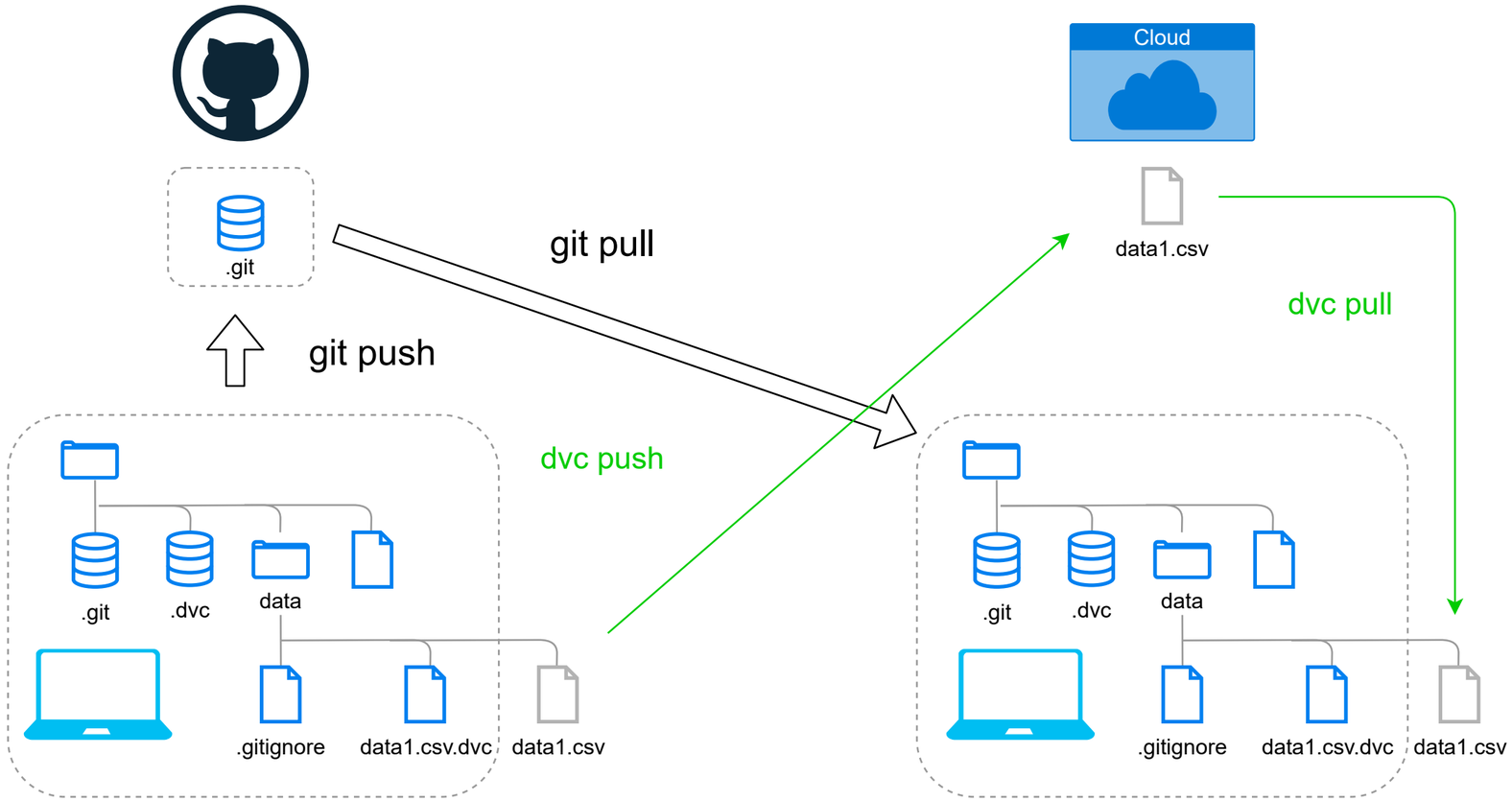
How Can We Work with Data in Git?

- There is a limit for the file size in GitHub (100 MB)
- GitHub's Git Large File Storage (GLFS) is costly (and a trap)
- You need to **git-ignore** these files

What is `.gitignore` ?

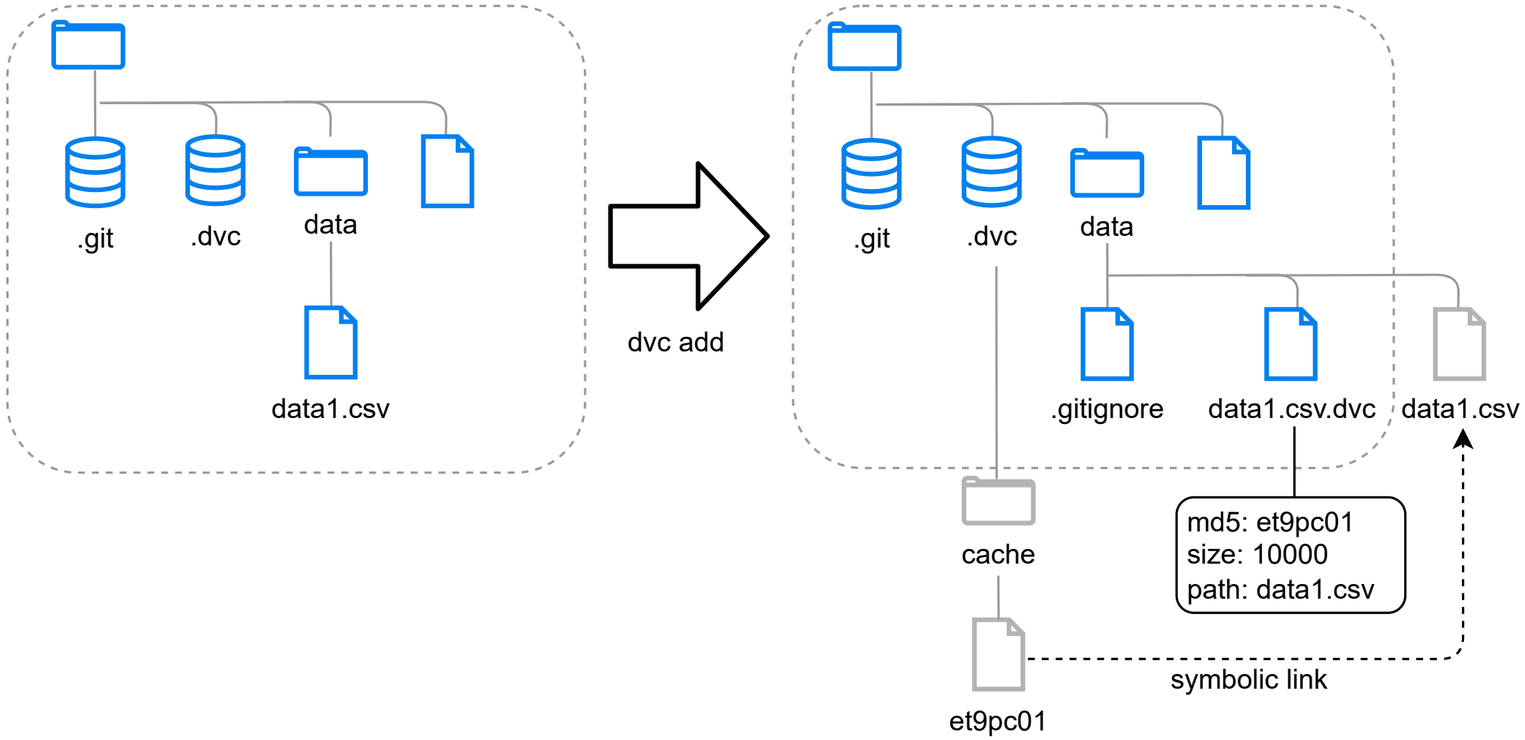
- A text file shows files you don't want track by git
- Usually, large files, auxiliary files, and secret information are git-ignored
- You can specify by folder (e.g. `data/`) and by extension (e.g. `*.csv`)

Data Version Control (DVC)



- Create a text file (`*.dvc`) for the meta-information of the data
- Git tracks only dvc files and ignores the data (by `.gitignore`)
- The original data is stored in the remote storage (Google Drive, Amazon S3, ..)

How Does DVC Work?



- A data file is actually stored in `.dvc/cache` with a random (hashed) name
- In the data folder, the data file is just a symbolic link to the original file
- Data will be stored with this hashed name in the cloud storage

DVC Setup Commands

1. Init a DVC project

```
dvc init
```

2. Setup the remote DVC Repository (Google Drive)

- Create a new Google drive folder and copy the folder id from its URL
- Run the follwoing

```
dvc remote add --default myremote gdrive://GDRIVE_FOLDER_CODE
```

DVC Workflow Commands

1. Track a file by DVC

```
dvc add foo1.csv
```

You can specify a folder (e.g. data) with *-R* option

```
dvc add -R data
```

2. Push data to the remote storage

```
dvc push
```

For the first push (or pull), you will see an authentication URL.

- Click the link and log in to your Google account
- Check both checkboxes

3. Pull data from the remote storage

```
dvc pull
```

Handson 5: Setup DVC and Follow My Workflow

1. Init a DVC project and setup its remote on Google Drive
2. Go to the repository of this workshop and read my workflow
3. In step 2, add a csv file in `data/csv` folder and create a commit (say, “add foo1.csv”)
4. Follow the rest of my workflow

Troubleshoot

Git Reference log

You can see all the git activity (commit, merge, rebase, ...)

```
1 git reflog
```

Git Reset

You can reset any git activity

```
1 git reset --soft COMMIT_ID_OR_REFLOG_ID
```

Keep in Mind

- Nothing bad happens with Git. It just stores savepoints.
- In practice, you rarely restore a file from a commit. But it gives you a lot of relief and freedom.

Learn More

Introduction to Git • Introduction to GitHub