Earth Sciences New Zealand

Git Basics Training

Hands-on Workshop on the Fundamental concepts of Git version control and Github

Faye Nielsen, Mario Krapp, Jack Drummond, Quyen Nguyen

Agenda

1. Intro & Key Concepts
2. Git (local) Fundamentals
3. GitHub and Collaboration
4. Discussion & Wrap up

What is Git and Github?

Git is known as a version control system

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.*

Github is an online platform to store files in the cloud which enables backed up storage, sharing/collaborating with others, easy integration with other cloud softwares

Why is Git useful?

Git is like a time machine for your code that creates snapshots of your work, letting you easily revert to previous versions and experiment safely.

Example: You're analysing earthquake data with analyse_data.py. You add a new visualisation function, but it breaks your entire analysis. What changed?

Without Git: analyse_data_v1.py, analyse_data_v2_final.py etc.

With Git: One file. Instant revert to the last working version with a single command. Essential when working with complex research code or collaborating with others.

How does it work?

Essentially, Git is a turbocharged save button.

A save in Git is called a **commit**. When you make a commit, you take a snapshot of all the files you have made visible to git – otherwise known as your git **repository**, or repo for short.

This snapshot saves everything in it's current state, and that particular commit can then be accessed further down the line if you mess anything up.

Key Concepts / Commands

Repository (repo for short)

A folder/directory that is tracked by git (i.e. it has a .git folder)

Repository (repo for short)

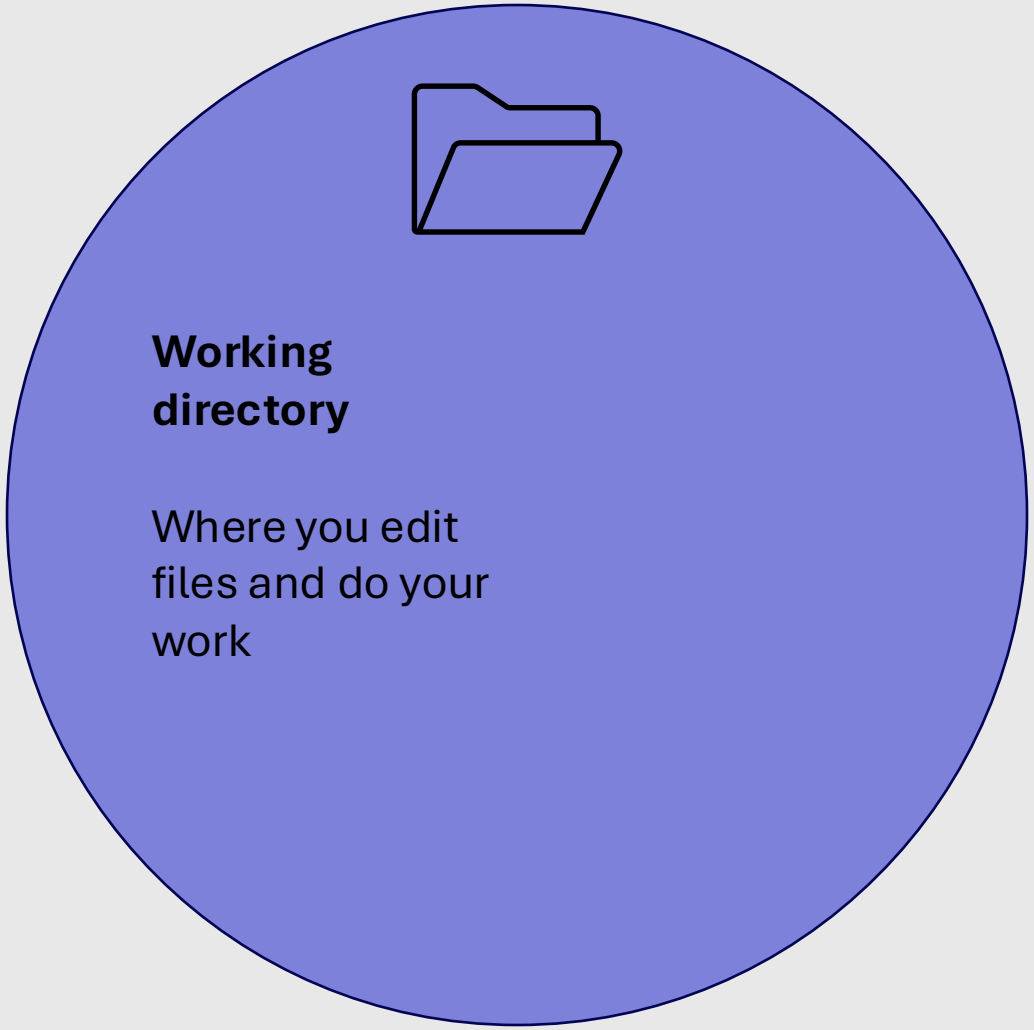
A folder/directory that is tracked by git (i.e. it has a .git folder)

A local repository is on your local machine e.g. laptop, PC

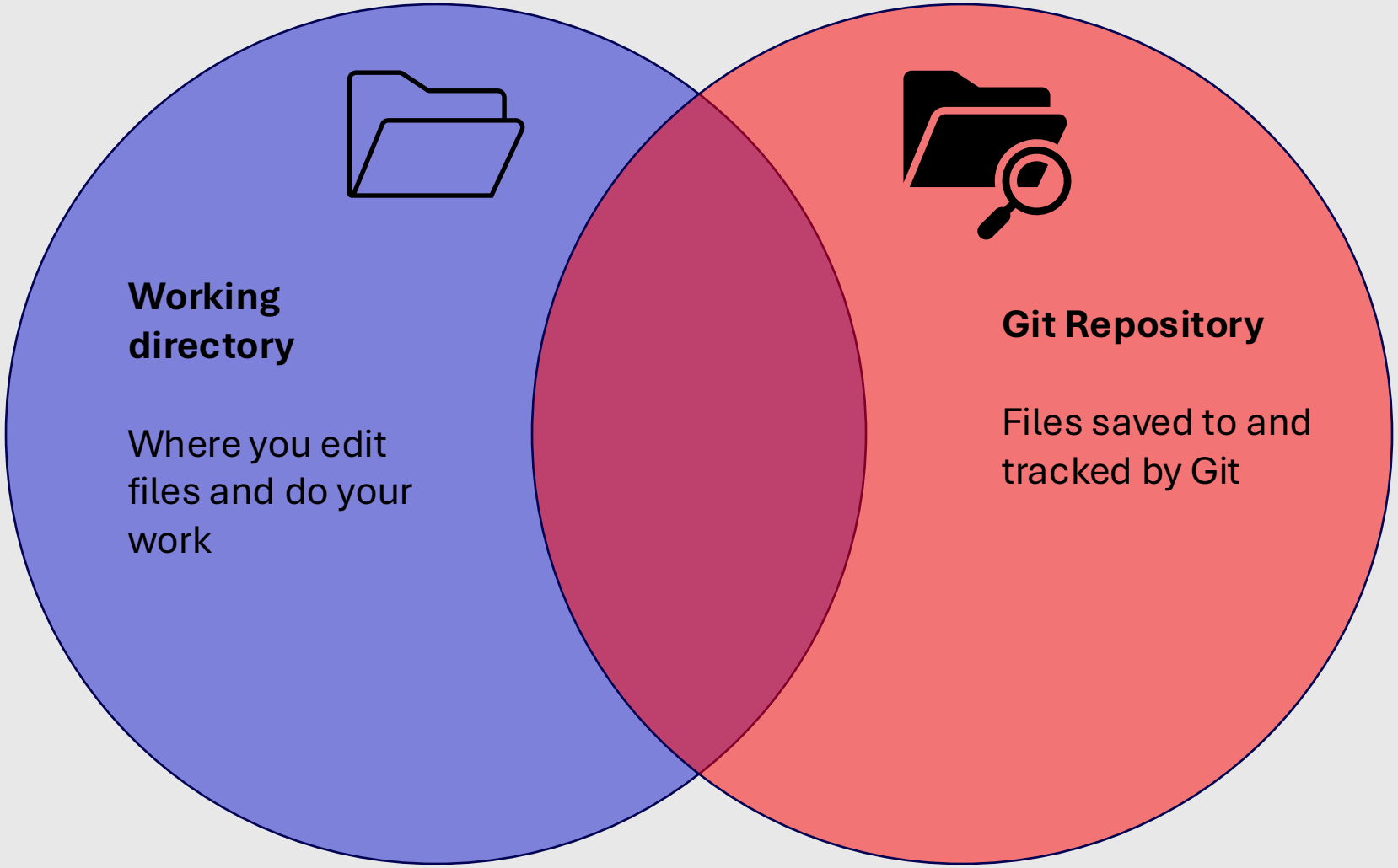
A remote repository is hosted on a server e.g. Github, Gitlab

The Three States / Areas

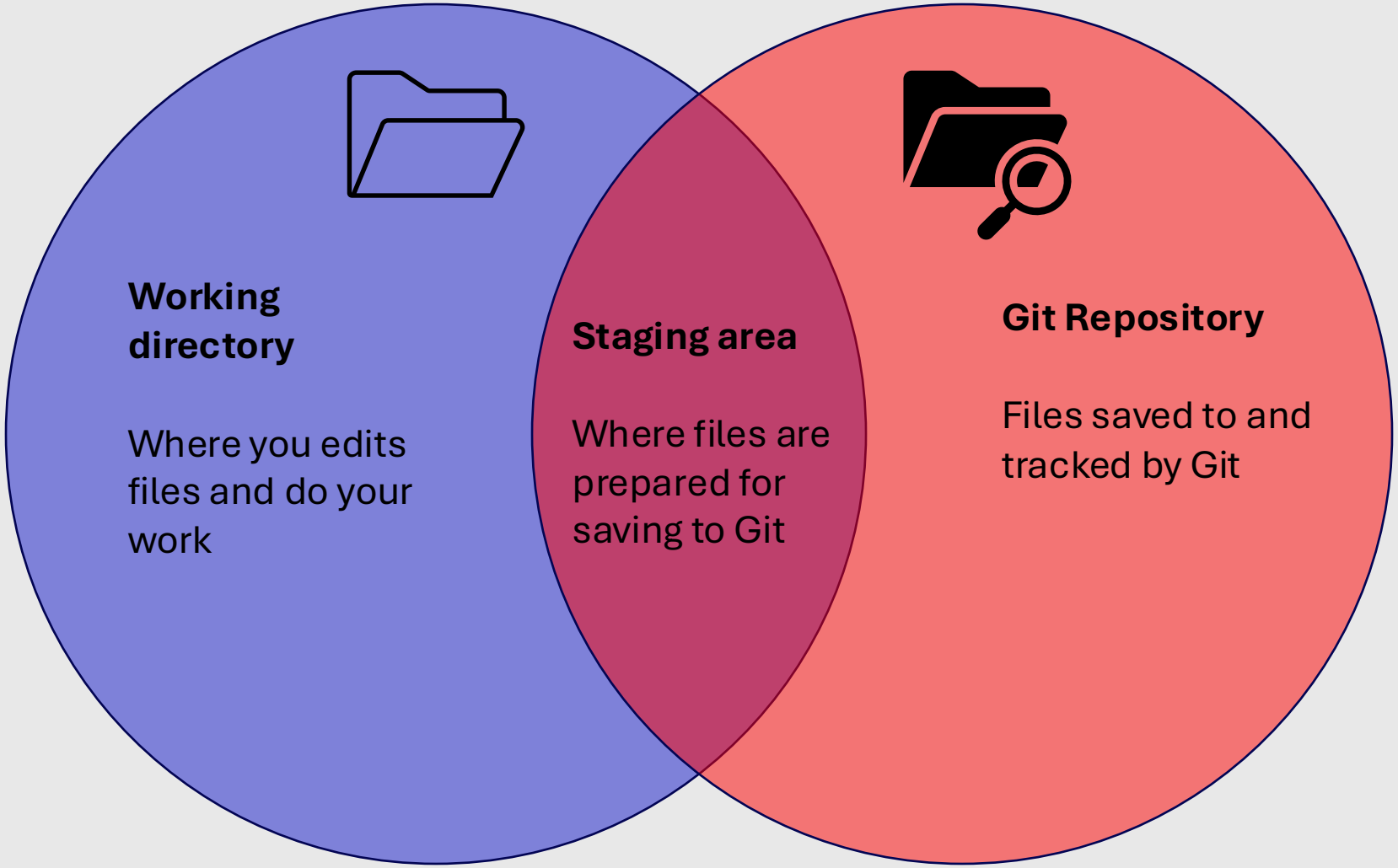
The Three States / Areas



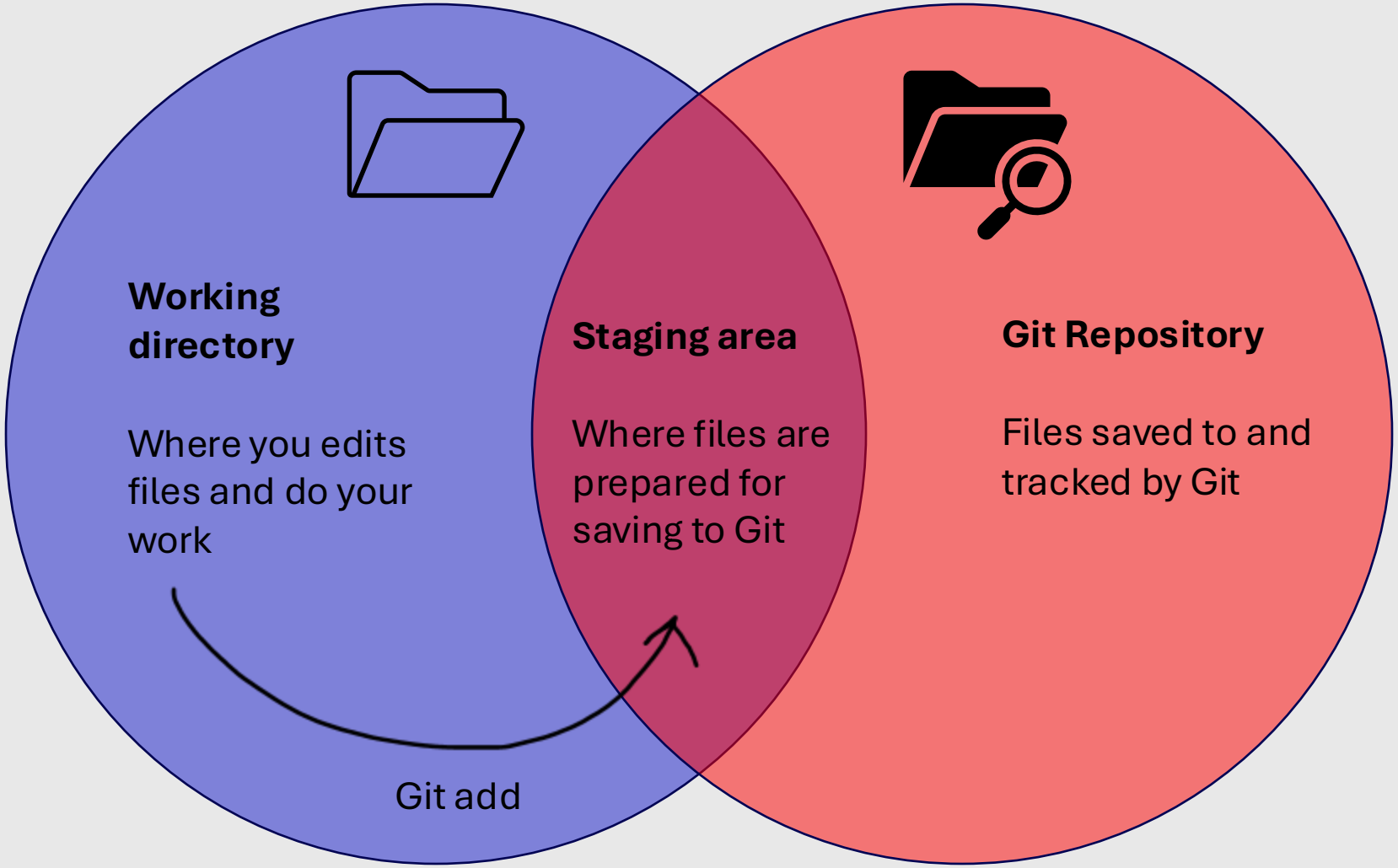
The Three States / Areas



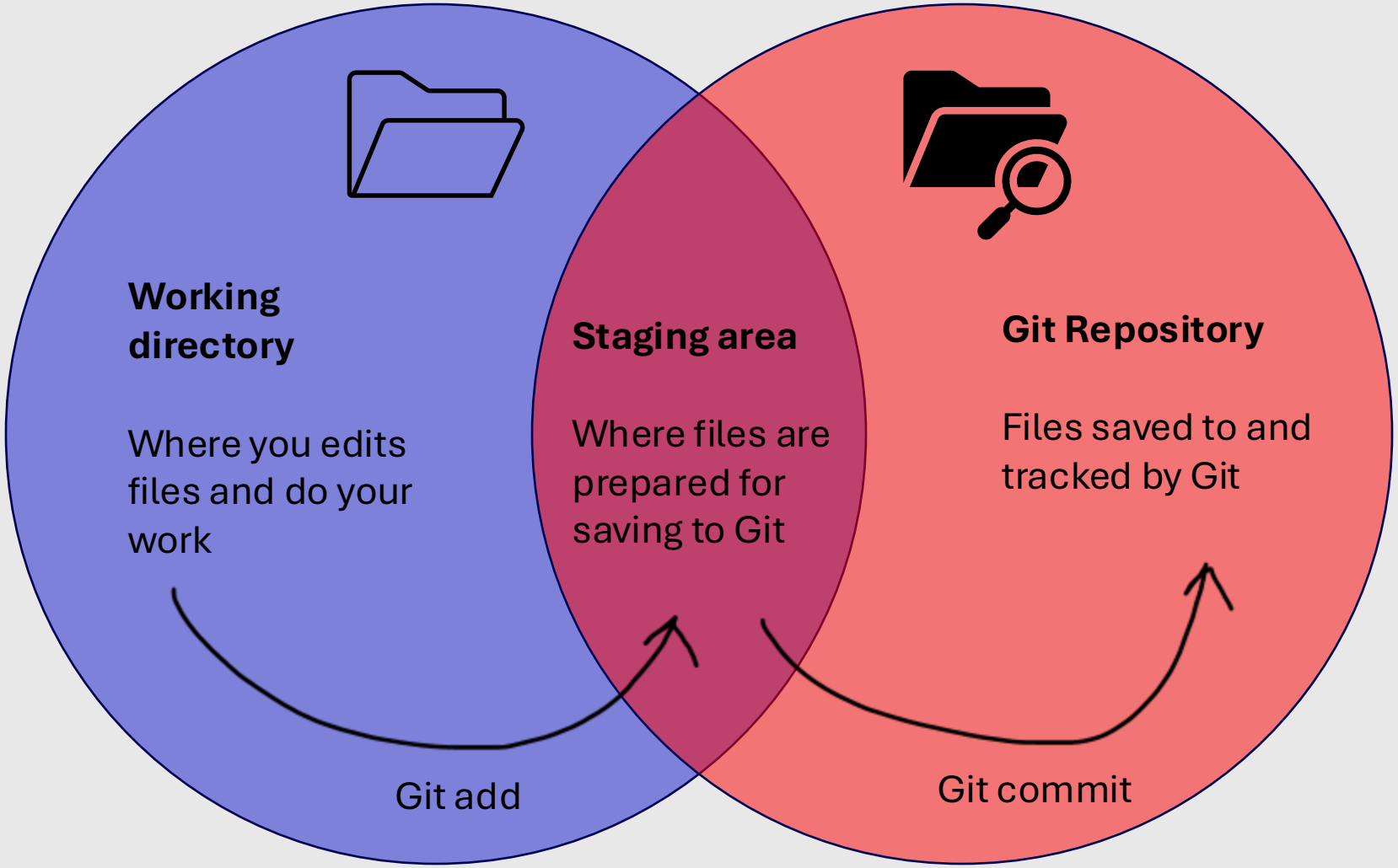
The Three States / Areas



The Three States / Areas



The Three States / Areas



Add

‘Stage’ files

Prepare files to track in Git by selecting which ones you want to save

This is useful as you might only want to save one file, and leave the others out as they are not needed

Commit

Snapshot of all files at that moment in time, including the changes you ‘staged’ by using the ‘add’ command

These are the building blocks of your version control history, when you look back through them you will see what changed between one commit and the next

Has a unique ID, message, author and timestamp

Branch

Parallel version of a project

Enables you to work on different features on different branches, rather than everything all together where it can get confusing

Enables multiple people to work on the same codebase without disrupting/breaking what someone else is working on

Gets merged back into the ‘main’ branch

Merge

Merging branches back into the ‘main’ branch

The main branch is the source of truth. For example, if you have a software product, what is live in the product will be what is on the main branch.

When changes on a branch are finished, they can be merged into main and be made ‘official’

Set up First Repository

Step 1: Configure identity

You only ever need to do this once

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Check it worked:

```
git config --global --list
```

Step 2: Create first repository

I recommend creating a folder called repositories in your home folder

```
cd ~                cd C:\Users\yourUserName

mkdir repositories

cd repositories

mkdir my-first-project

ls                  dir

cd my-first-project

git init
```

Step 2: Create first repository

I recommend creating a folder called repositories in your home folder

Check for hidden .git file:

```
ls -la        dir /a
```

Check what git can currently see:

```
git status
```

Step 3: Create first file

1. Create a README.md file with your text editor:

```
echo # My Git Project > README.md
```

2. Check the file exists:

```
ls        dir
```

3. Look at the contents on the file:

```
more README.md          less README.md              cat README.md
```

4. Exit:

```
q
```

Step 4: First commit

1. Check what Git sees:

```
git status
```

2. Stage the file (prepare it for commit):

```
git add README.md
git status
```

3. Make your first commit (save the snapshot):

```
git commit -m "Initial commit: Add README”
git status
```

Step 4: First commit

You want to use descriptive messages in each commit.

It is really easy to forget what or why you made a commit, and even harder for other people if you are collaborating.

Step 3: View History

```
git log
```

The Git Workflow

The Git Workflow

Edit -> Check -> Add -> Commit -> Repeat

Step 1: Single file change

1. Edit README.md, add these lines:

```
I am learning about version control
```

2. Check what changed:

```
git status
```

3. See the actual changes:

```
git diff
```

4. Stage and commit:

```
git add README.md
```

```
git commit -m "Add learning section to README"
```

5. Check the log:

```
git log --oneline
```

Step 2: Multiple file change

1. Create a python file called ‘test.py’:

```
print(“hello, world!”)
```

```
echo 'print('hello, world!')' > test.py
```

2. Edit README.md:

```
# My Progress
Learning git basics
```

```
Echo ‘# My Progress\nLearning git basics’ >> README.md
```

Step 2: Multiple file change

3. Check status:

```
git status
```

4. Stage only README.md:

```
git add README.md
```

```
git status
```

5. Unstage it:

```
git restore --staged README.md
```

```
git status
```

Step 2: Multiple file change

6. Add all changes at once:

```
git add -A
```

```
git status
```

Stages everything: new files, modified files, deleted files

7. Commit everything together:

```
git commit -m "Add test.py and update README with progress”
```

8. View your history:

```
git log --oneline
```

Quick Practice of the workflow

Challenge: On your own, try one more cycle:

- Make any change to any file, create a new one, or delete one
- Use git status to see it
- Use git add -A to stage everything
- Commit with a descriptive message

Fixing Common Mistakes

Scenario 1: Added the wrong file

1. Create a temporary file:
echo "temporary stuff" > temp.txt
Windows Command Prompt: echo temporary stuff > temp.txt
2. Oops, add it by mistake:
git add temp.txt
git status
Point out: It is green and ready to be committed
3. Unstage it:
git restore --staged temp.txt
git status
Point out: Back to red (unstaged)

Scenario 2: I want to undo my changes

1. Edit test.py but don't commit :

Print(“1+1 = 3”)

2. Check you have uncommitted changes:

git status

git diff

3. Discard the changes:

git restore test.py

git status

File is back to how it was at last commit

Scenario 3: Committed too early

1. Make a small change to notes.txt and commit:
echo "- git restore" >> notes.txt
git add notes.txt
git commit -m "Update notes"
Windows Command Prompt: echo - git restore >> notes.txt
2. Oops! You realize you wanted to add more. Let's undo that commit but keep the changes:
git reset --soft HEAD~1
Explain:
HEAD~1 means "one commit before current"
--soft keeps your changes staged (green)
The commit disappears but your work is safe
3. Check status:
git status
Point out: Your changes are still there, staged and ready
4. Now you can add more changes and commit them together:
echo "- git reset" >> notes.txt
git add notes.txt
git commit -m "Update notes with restore and reset commands"
Windows Command Prompt: echo - git reset >> notes.txt
5. Check the log:

Scenario 4: Committed the wrong thing

1. Let's create a mistake on purpose - add a file we don't want:
echo "This is a mistake" > mistake.txt
git add mistake.txt
git commit -m "Add mistake file (oops!)"
2. View log:
git log --oneline
Point out: There's our mistake commit
3. Undo the commit completely:
git reset HEAD~1
Explain:
No --soft this time (default is --mixed)
Changes are kept but unstaged (red)
More common than --soft
4. Check status:
git status
Point out: mistake.txt is there but unstaged
5. We don't want this file at all, so discard it:
git restore mistake.txt
6. Verify it's gone:
git status