Telstra
Purple

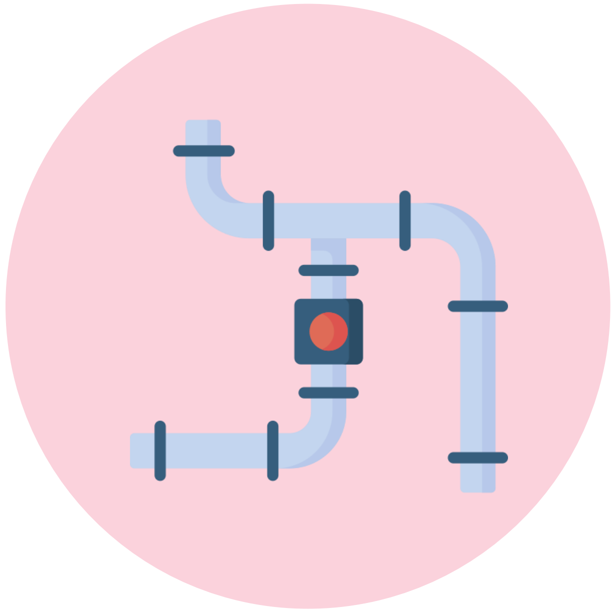
Git Crash Course

Jonathan Neo | Consultant

ADS Go Fast Training Sessions

Session 1
Git Crash Course

Session 2
Azure DevOps Crash Course

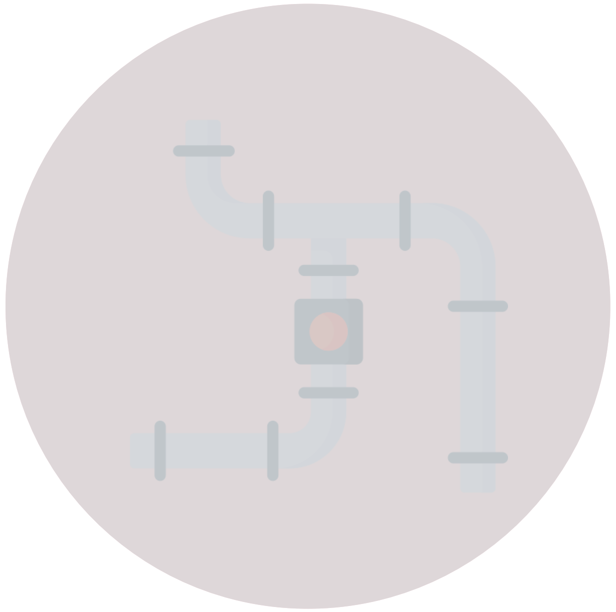


Session 3
ADS Go Fast CI/CD

ADS Go Fast Training Sessions

Session 1
Git Crash Course

Session 2
Azure DevOps Crash Course



Session 3
ADS Go Fast CI/CD

Agenda / navigation

Pre-requisites

It is recommended that you have the following software installed on your computer so that you can follow along the exercises.

Git

https://git-scm.com/downloads

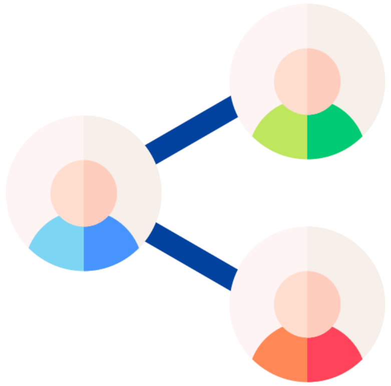
Visual Studio Code

https://code.visualstudio.com/download

Git Basics – What is Git?

A Version Control System (VCS) that supports

Change tracking



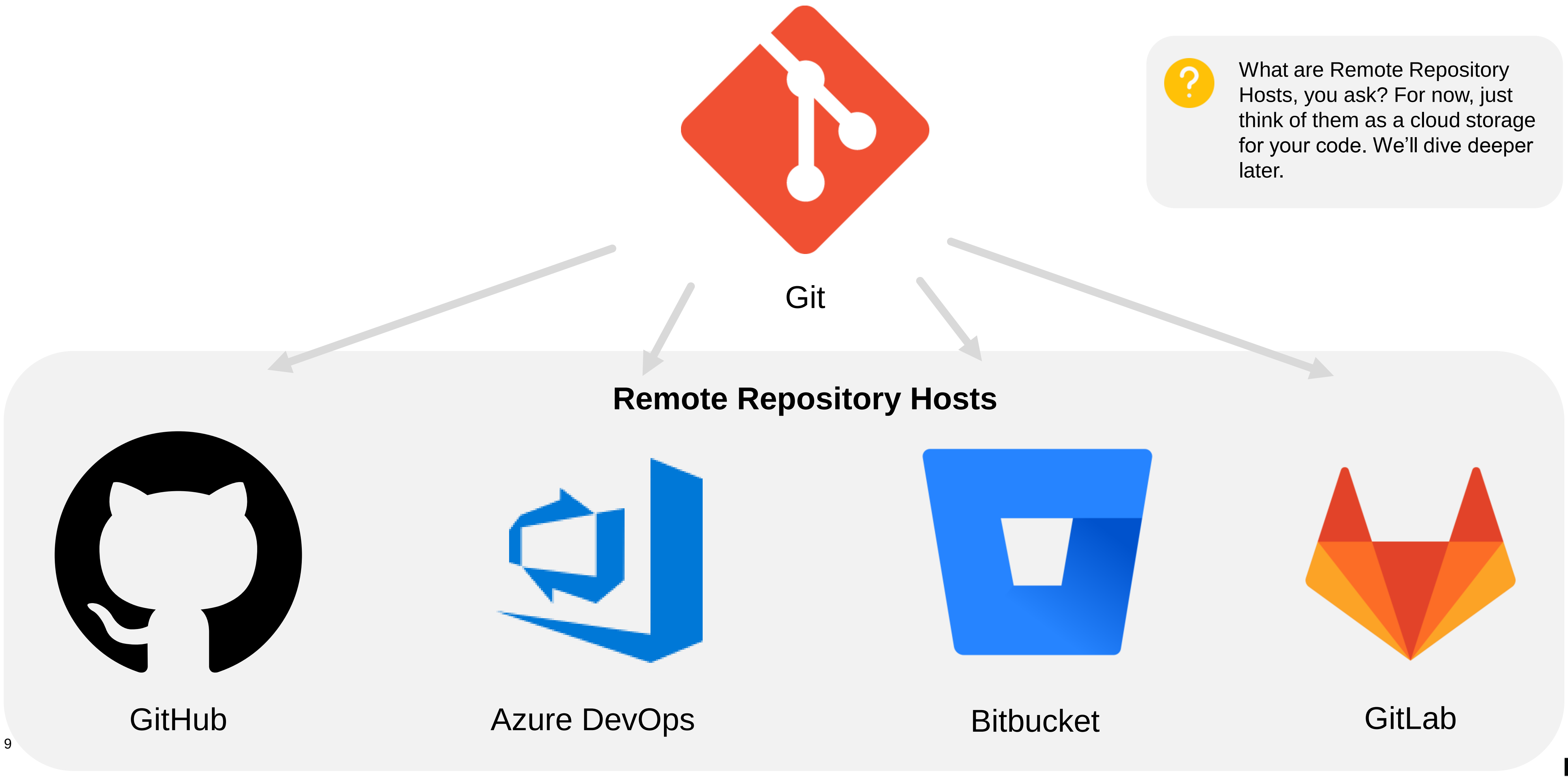
Collaboration

Learn more about Version Control Systems here.

Git Basics – What is Git?

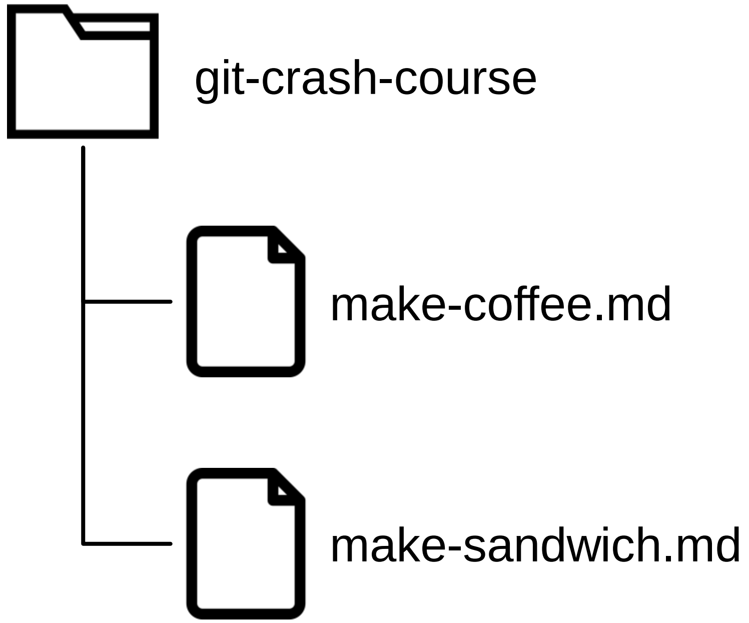
Git $\neq$ GitHub

Git Basics – What is Git?



? What are Remote Repository Hosts, you ask? For now, just think of them as a cloud storage for your code. We'll dive deeper later.

Git Basics – Tracking Change



What is wrong with the make-coffee instructions?

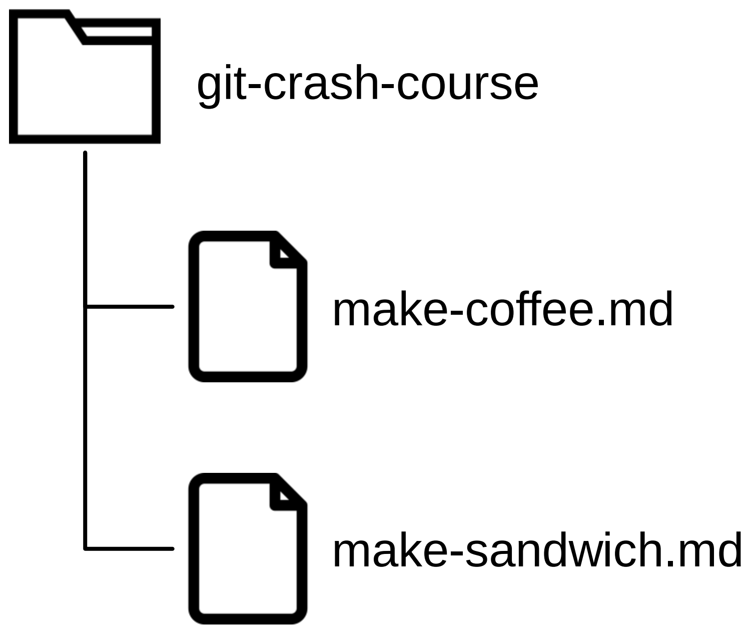
make-coffee.md

```
# Instructions - Pour over coffee

1. Bring cold water to a boil in a kettle
2. Grind the beans
3. Put a filter in the brewer and rinse with hot water
4. Add the grounds to the filter
5. Heat water to 100°C
6. Slowly and steadily pour just enough water over the grounds to satu
rate them completely, starting from the middle and working your way ou
tward. This should take 3 to 4 minutes.
7. Carefully remove the filter
8. Serve and enjoy
```

make-sandwich.md

```
# Instructions

1. Take two slices of bread from bag
2. Spread butter on each slice
3. Take two slices of ham out from fridge and add one to each slice
4. Take two lettuce leaves out from fridge and add one to each slice
5. Take one slice of cheese and add to one slice
6. Take one half of the slice stack and add to the other. Done.
```

Git Basics – Tracking Change



make-coffee.md

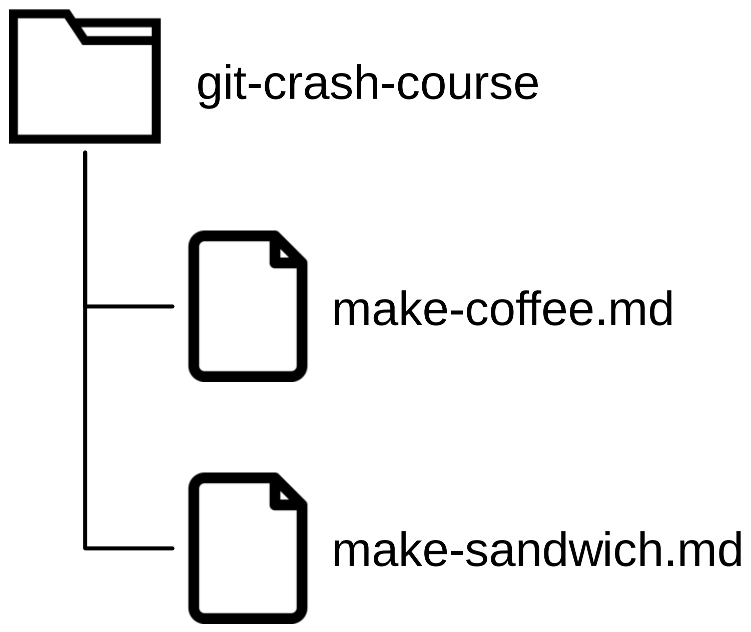
```
# Instructions - Pour over coffee

1. Bring cold water to a boil in a kettle
2. Grind the beans
3. Put a filter in the brewer and rinse with hot water
4. Add the grounds to the filter
5. Heat water to ~~100°C~~ 90°C and 95°C
6. Slowly and steadily pour just enough water over the grounds to satu
rate them completely, starting from the middle and working your way ou
tward. This should take 3 to 4 minutes.
7. Carefully remove the filter
8. Serve and enjoy
```

make-sandwich.md

```
# Instructions

1. Take two slices of bread from bag
2. Spread butter on each slice
3. Take two slices of ham out from fridge and add one to each slice
4. Take two lettuce leaves out from fridge and add one to each slice
5. Take one slice of cheese and add to one slice
6. Take one half of the slice stack and add to the other. Done.
```

Coffee enthusiasts know that you will burn the grounds if you use 100 degrees water!

Git Basics – Tracking Change



make-coffee.md

```
# Instructions - Pour over coffee

1. Bring cold water to a boil in a kettle
2. Grind the beans
3. Put a filter in the brewer and rinse with hot water
4. Add the grounds to the filter
5. Heat water to ~~100°C~~ 90°C and 95°C
6. Slowly and steadily pour just enough water over the grounds to satu
rate them completely, starting from the middle and working your way ou
tward. This should take 3 to 4 minutes.
7. Carefully remove the filter
8. Serve and enjoy
```

make-sandwich.md

```
# Instructions

1. Take two slices of bread from bag
2. Spread butter on each slice
3. Take two slices of ham out from fridge and add one to each slice
4. Take two lettuce leaves out from fridge and add one to each slice
5. Take one slice of cheese and add to one slice
6. Take one half of the slice stack and add to the other. Done.
```

Let's explore how Git can help track these changes for us.
The >> symbol represents sections where you can code along.

Useful: A cheatsheet of Git commands created by GitHub here.

Git Basics – Initialize a Git repository

To get Git to start tracking changes to a directory, you must initialize that directory as a Git repository.

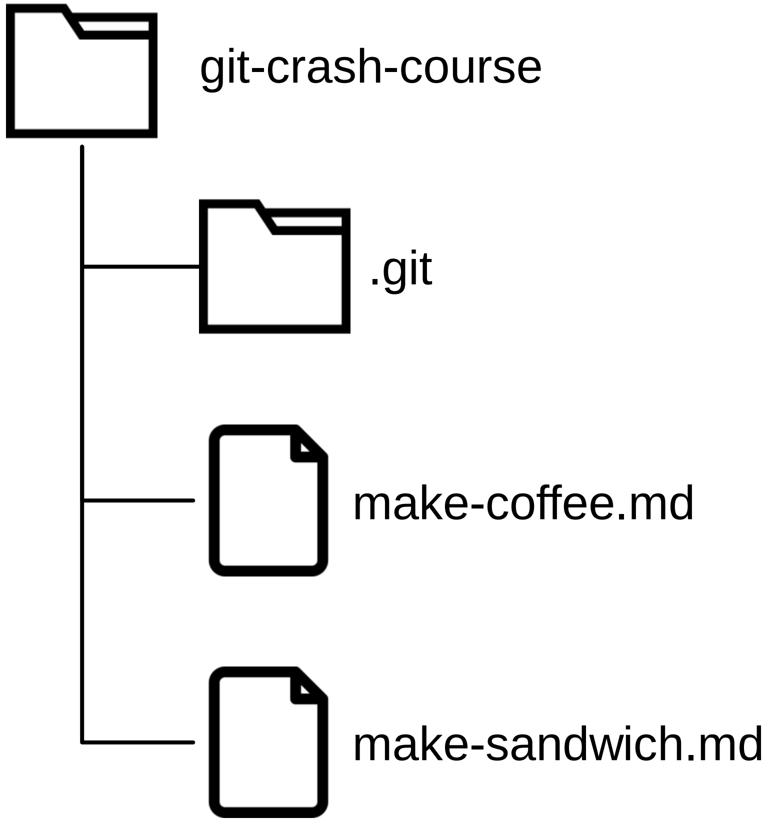
What is a repository?
A repository is just a name that Git uses to represent a folder or directory.

terminal

```
> cd "<the folder you want to initialize as a git repository>"
> git init
```

Git Basics – Initialize a Git repository

After executing “git init”, you will notice that a folder called “.git” will have been created. “.git” will contain the files that Git uses to track changes to your repository.



Can’t find “.git”? You will have to show hidden files and folders on your computer. Windows, Mac.

Git Basics – Make changes!

Now that you have initialized your directory as a Git repository, you can start making changes to files just like how you would edit a text file.
In order to create a snapshot of those changes, we need to understand 3 phases in Git.

make-coffee.md

```
# Instructions - Pour over coffee

1. Bring cold water to a boil in a kettle
2. Grind the beans
3. Put a filter in the brewer and rinse with hot water
4. Add the grounds to the filter
5. Heat water to 100°C
6. Slowly and steadily pour just enough water over the grounds
 to saturate them completely, starting from the middle and wor
king your way outward. This should take 3 to 4 minutes.
7. Carefully remove the filter
8. Serve and enjoy
```

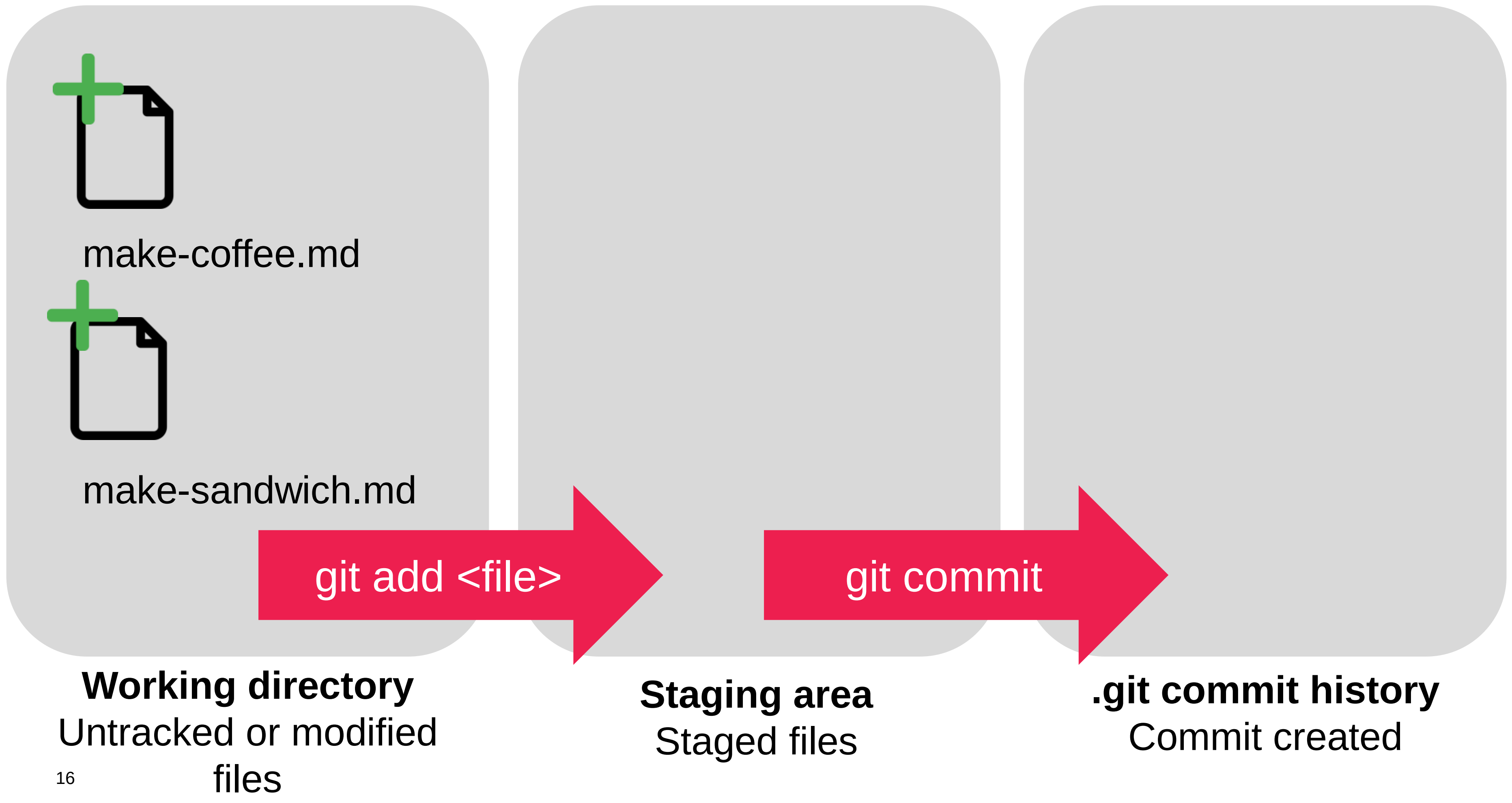
make-coffee.md

```
# Instructions - Pour over coffee

1. Bring cold water to a boil in a kettle
2. Grind the beans
3. Put a filter in the brewer and rinse with hot water
4. Add the grounds to the filter
5. Heat water to ~~100°C~~ 90°C and 95°C
6. Slowly and steadily pour just enough water over the grounds
 to saturate them completely, starting from the middle and wor
king your way outward. This should take 3 to 4 minutes.
7. Carefully remove the filter
8. Serve and enjoy
```

changes

Git Basics – Git Phases

Git uses the concept of staging your changes before creating commit.



What are untracked files?

Tracked files are files that were in the last snapshot. Untracked files are new files.

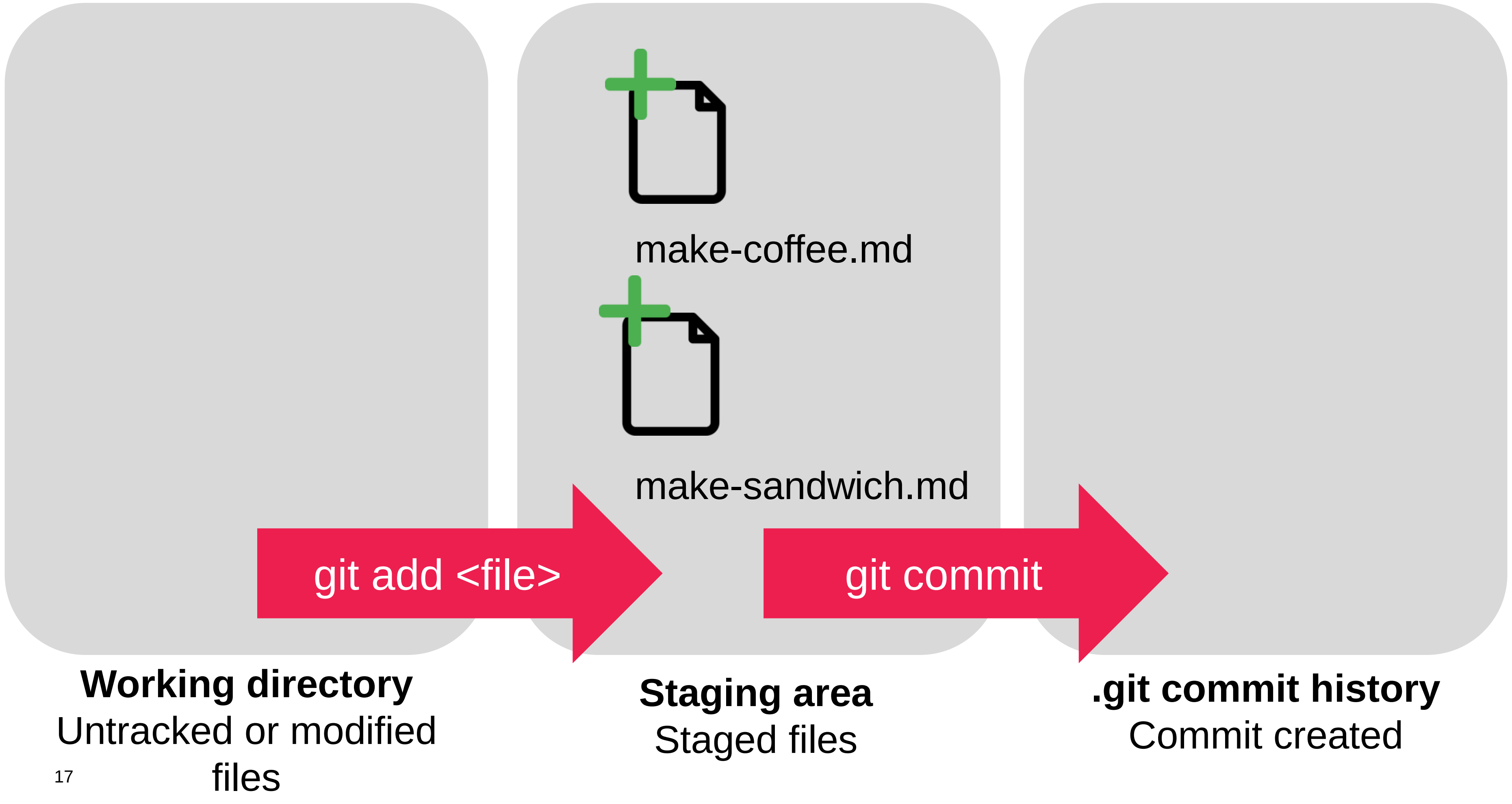
Why do we need to stage changes? Why not just commit all changes?

You may not always want to commit all your changes straight away – thus “git add” allows you to commit only certain changes.

A more detailed explanation of tracking changes here.

Git Basics – Git Phases

Git uses the concept of staging your changes before creating commit.



What are untracked files?

Tracked files are files that were in the last snapshot. Untracked files are new files.

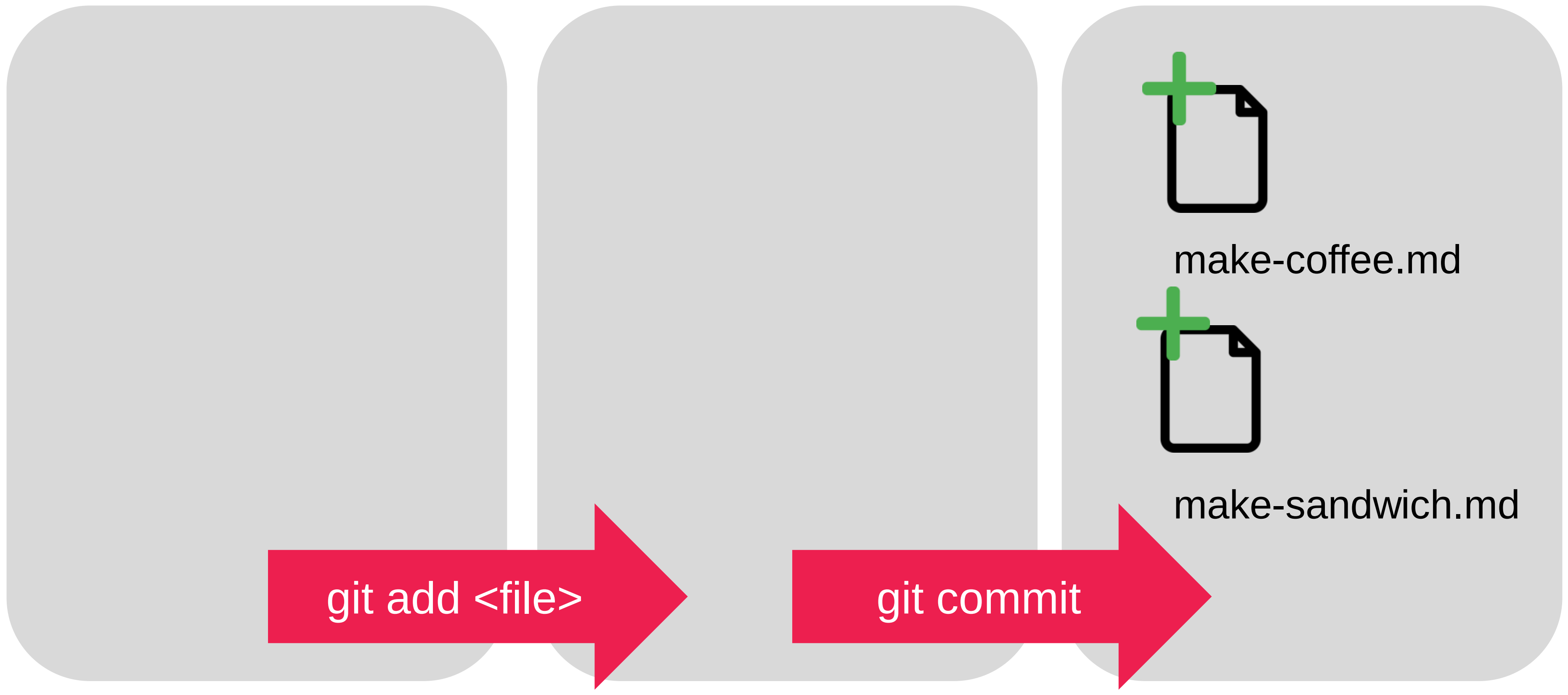
Why do we need to stage changes? Why not just commit all changes?

You may not always want to commit all your changes straight away – thus "git add" allows you to commit only certain changes.

A more detailed explanation of tracking changes here.

Git Basics – Git Phases

Git uses the concept of staging your changes before creating commit.



Working directory
Untracked or modified files

Staging area
Staged files

.git commit history
Commit created

What are untracked files?

Tracked files are files that were in the last snapshot. Untracked files are new files.

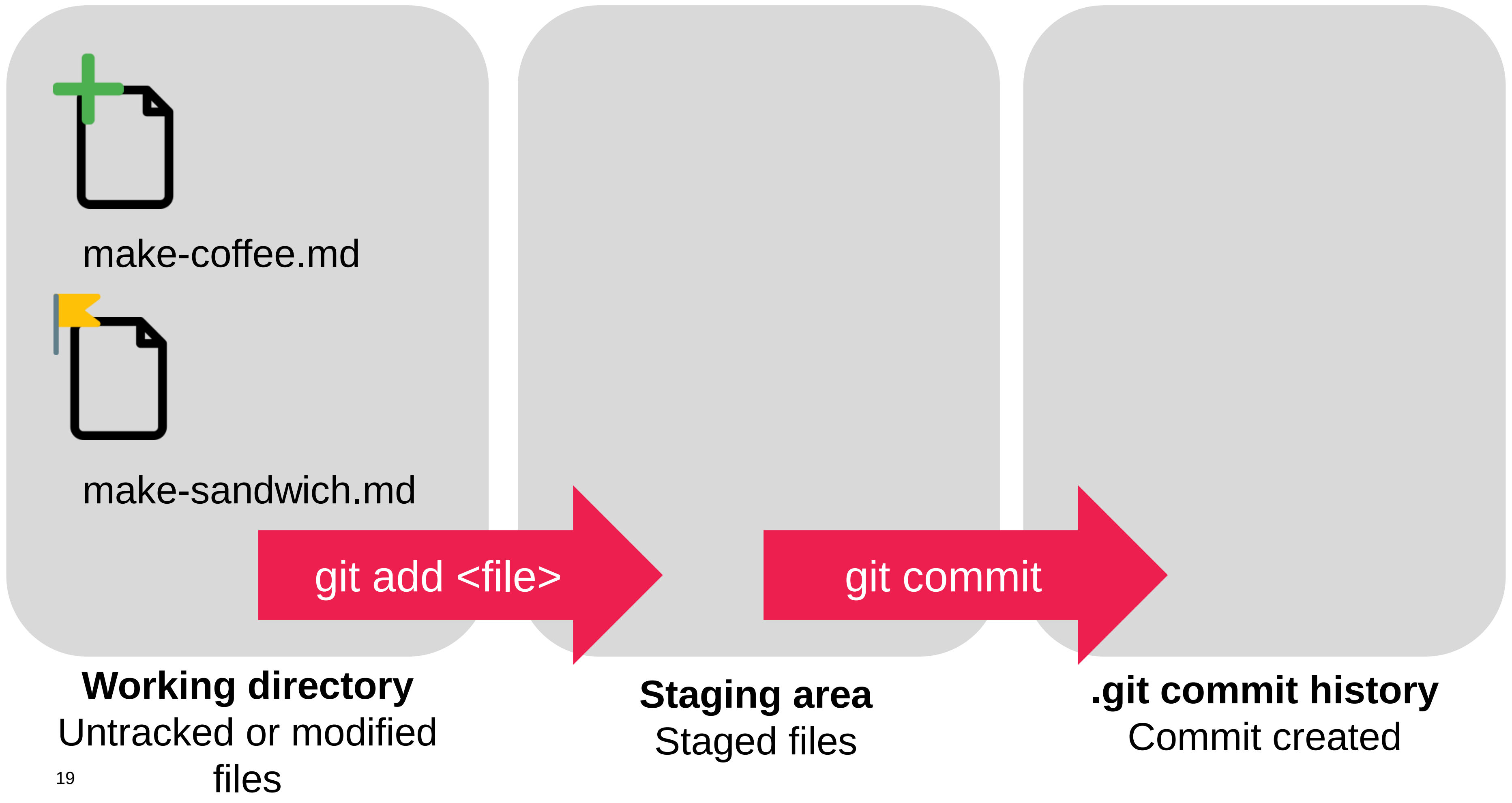
Why do we need to stage changes? Why not just commit all changes?

You may not always want to commit all your changes straight away – thus “git add” allows you to commit only certain changes.

A more detailed explanation of tracking changes here.

Git Basics – Git Phases

You can also add only one file to the staging area and only commit that change. This gives you control over what changes you want to commit, and what files you're still working on.



What are untracked files?

Tracked files are files that were in the last snapshot. Untracked files are new files.

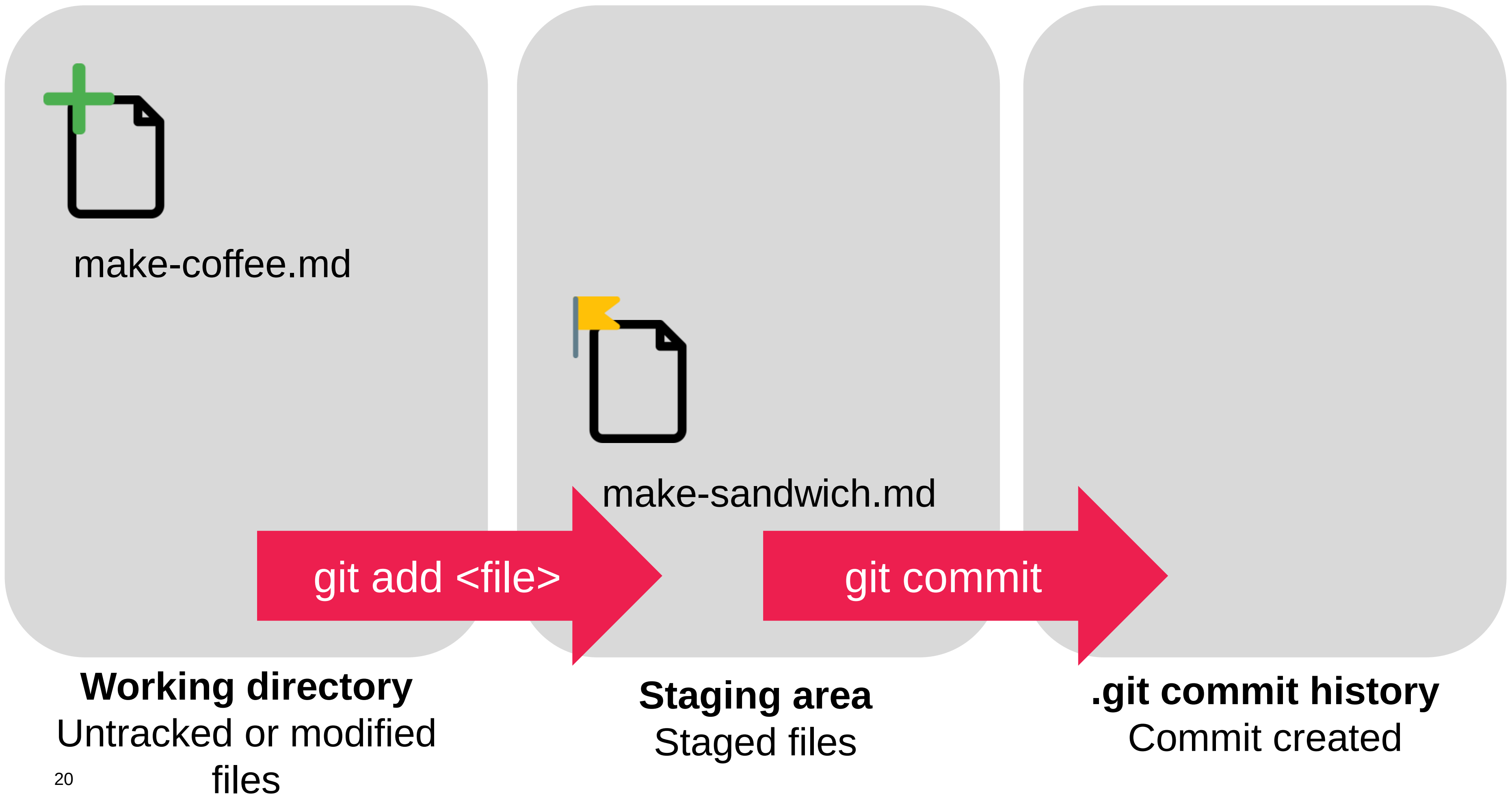
Why do we need to stage changes? Why not just commit all changes?

You may not always want to commit all your changes straight away – thus "git add" allows you to commit only certain changes.

A more detailed explanation of tracking changes here.

Git Basics – Git Phases

You can also add only one file to the staging area and only commit that change. This gives you control over what changes you want to commit, and what files you're still working on.



What are untracked files?

Tracked files are files that were in the last snapshot. Untracked files are new files.

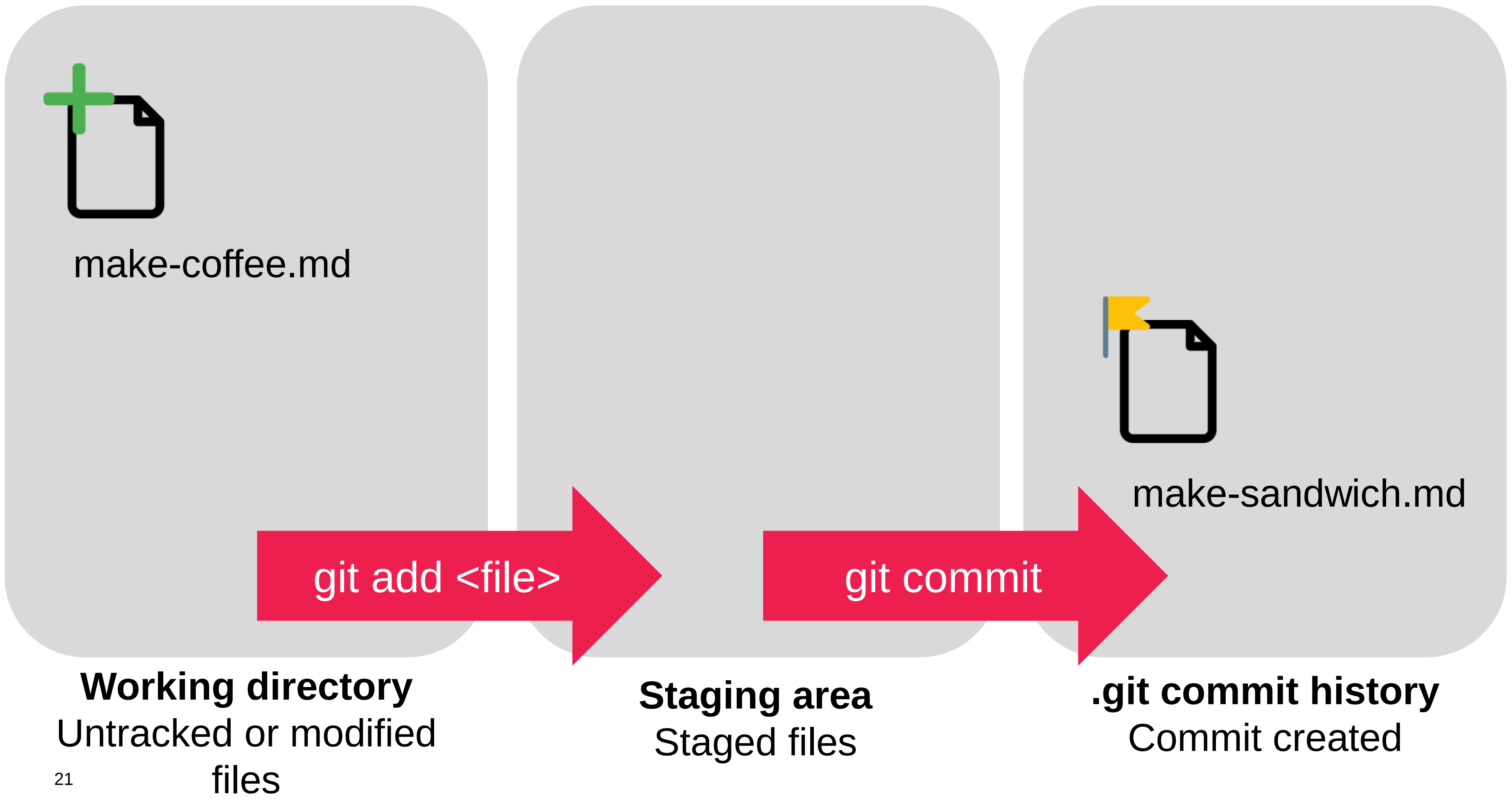
Why do we need to stage changes? Why not just commit all changes?

You may not always want to commit all your changes straight away – thus "git add" allows you to commit only certain changes.

A more detailed explanation of tracking changes here.

Git Basics – Git Phases

You can also add only one file to the staging area and only commit that change. This gives you control over what changes you want to commit, and what files you're still working on.



What are untracked files?

Tracked files are files that were in the last snapshot. Untracked files are new files.

Why do we need to stage changes? Why not just commit all changes?

You may not always want to commit all your changes straight away – thus "git add" allows you to commit only certain changes.

A more detailed explanation of tracking changes here.

Git Basics – Add <filename>

Now that you have initialized your directory as a Git repository, you can start making changes to files just like how you would edit a text file.
When you're ready for Git to save a snapshot of those changes, you will need to tell Git which files you wish to save snapshots of.
Use "git add <filename>". This will tell Git what files you wish to take snapshots by registering (adding) the changes to a staging area before creating the snapshot.

terminal

```
> git add make-coffee.md make-sandwich.md
```

Use "git status" before and after executing "git add" to see how Git tracks changes.

Git Basics – git add .

A short-cut to add all files is to use “git add .”
This will add all untracked or modified files in your repository to Stage, and then you can commit using “git commit”.

terminal

```
> git add . # adds all untracked or modified files to the staging area
> git commit -m “change temperature from 100 to 90-95”
```

Git Basics – Commit

After you’ve told Git what files you wish to save snapshots for, you can now save those snapshots by using “git commit”.
“git commit -m <message>” allows you to write more descriptive information about the commit.

terminal

```
> git commit -m “change temperature from 100 to 90-95”
```

Use “git show” to show changes of your current commit. You can use “git show <commit>” to show changes to past commits too!

You will also notice that Git uses a hash string to uniquely identify each commit. The hash string is always unique for every change and you will never have two same commit ids! Read more here.

Writing good Git commit messages is something we won’t cover in this course. But you can read blogs on it here.

Git Basics – Log

To view all the changes you have made, use “git log”. This will print out a log of the past commits.

terminal

```
> git log
```

Use “git log --all --graph --decorate” to show a pretty looking timeline of your commits!

What is HEAD?

HEAD just tells you which commit you are currently looking at (checked-out).

Git Basics – Diff

To view changes between two commits, you can use “git diff <old_commit> <new_commit>”.

terminal

```
> git diff <old_commit> <new_commit>
```

You would typically execute “git log” first to give you the commit history, and then use “git diff” to compare two commits.

Git Basics – Checkout <commit >

Git allows you to jump back in time to a previous version of your repository by using “git checkout <commit>”.
If you do “git checkout <commit>” now, you will notice that the contents of your file has reverted to a previous commit.
Git is clever enough to update your repository to the commit version that you’ve checked-out.

terminal

```
> git checkout <commit>
```

Git Basics – Exercise Time!

Now it's your turn! Download or recreate the "git-crash-course" folder and repeat steps from the previous slides.
A summarized list of steps is shown below. Execute them in order.
Note: "#" are comments – do not copy those into terminal.

Help! I did a Git commit, and now I'm stuck in this weird text screen.

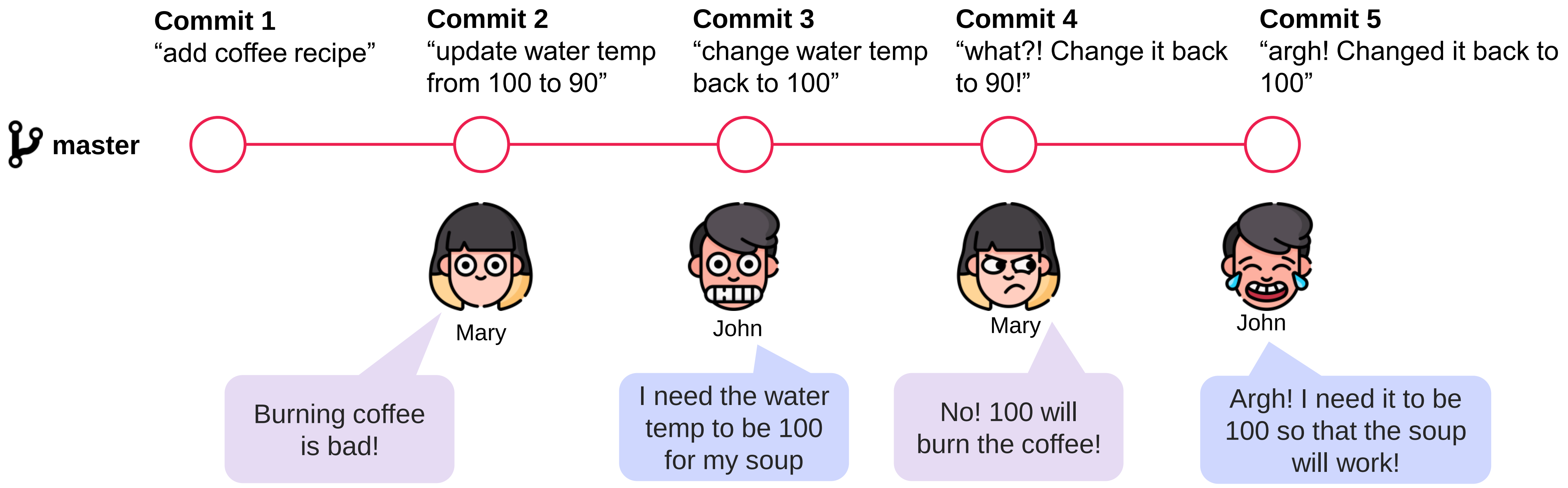
That is the vim editor. Hit <Esc> and type ":wq" + <Enter> to save and exit.

terminal

```
> cd "<the demo folder>" # cd: changes directory to the directory where the "demo" folder is at
> git init # initialize the repository
> git add make-coffee.md make-sandwich.md # adds make-coffee.md and make-sandwich.md to the staging area
> git commit -m "add make-coffee and make sandwich"
> # modify the coffee temperature from "100 degrees" to "90 to 95 degrees"
> git add make-coffee.md # stage changes that you have made to make-coffee. Alternatively, use "git add ." to stage all your
untracked or modified files.
> git commit -m "change coffee temp from 100 to 90-95"
> git show # show your current commit and the changes compared against the previous commit
> git log # show the git commit history
> git diff <old_commit_id> <new_commit_id> # perform a diff to compare two commit versions
> git checkout <old_commit_Id> # checkout to the old commit – observe how the file contents have changed!
> git checkout <new_commit_Id> # checkout back to the latest commit
```

Git Branches

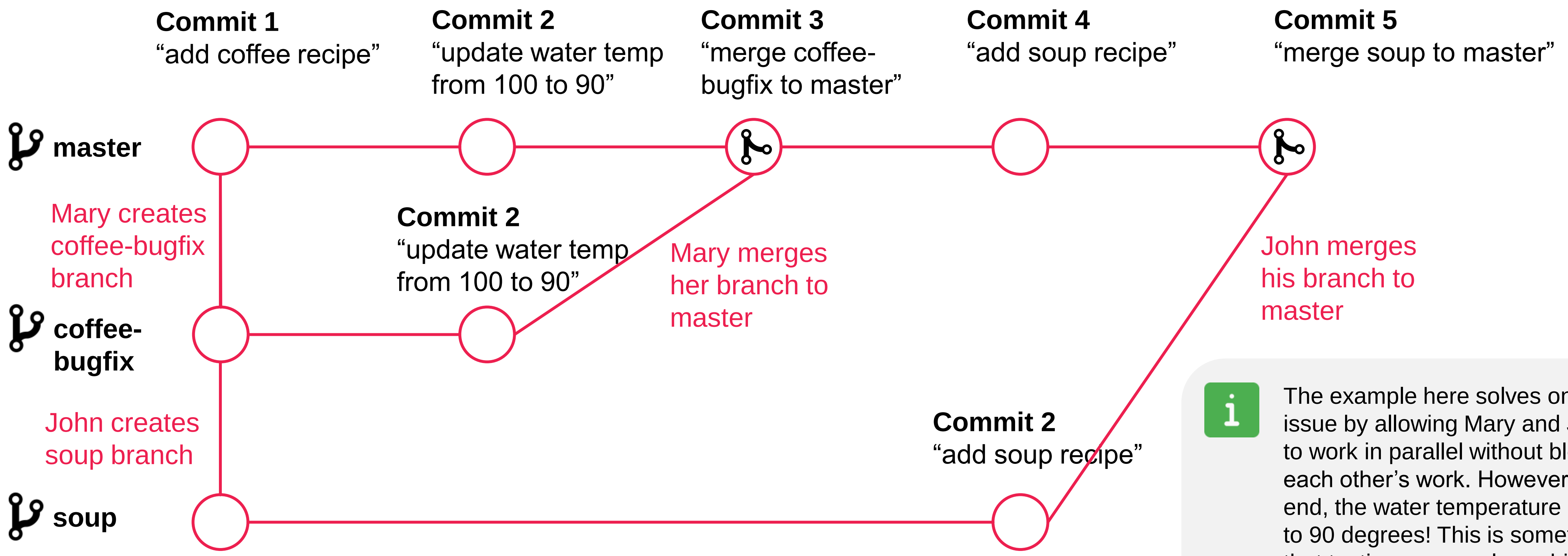
So far, we have been making commits to the “master” branch. The “master” branch is the default branch that is created by “git init” and is commonly used by developers to represent the main branch that code is published from. But having multiple people make changes on the “master” branch all at once can get messy quickly. Let’s look at an example below.



Commit 1
“add coffee recipe”
Commit 2
“update water temp from 100 to 90”
Commit 3
“change water temp back to 100”
Commit 4
“what?! Change it back to 90!”
Commit 5
“argh! Changed it back to 100”
master
Mary
John
Mary
John
Burning coffee is bad!
I need the water temp to be 100 for my soup
No! 100 will burn the coffee!
Argh! I need it to be 100 so that the soup will work!

Git Branches

Let's now see how Mary and John can work in feature branches to avoid overwriting each other's work. Let's assume Mary creates the coffee-bugfix branch and John creates the soup branch.

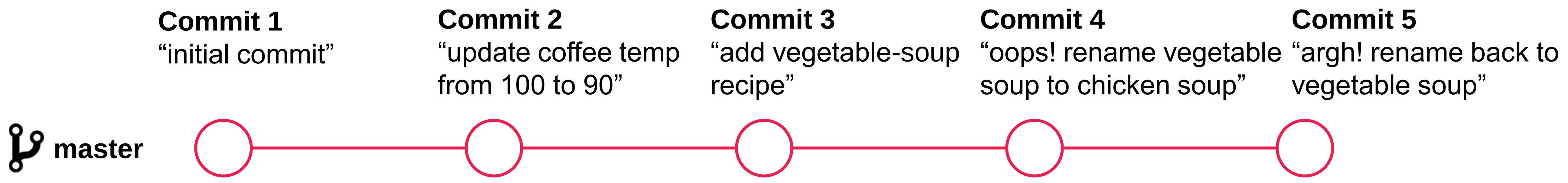


Commit 1
"add coffee recipe"
Commit 2
"update water temp from 100 to 90"
Commit 3
"merge coffee-bugfix to master"
Commit 4
"add soup recipe"
Commit 5
"merge soup to master"
master
Mary creates coffee-bugfix branch
Commit 2
"update water temp from 100 to 90"
Mary merges her branch to master
John merges his branch to master
coffee-bugfix
John creates soup branch
Commit 2
"add soup recipe"
soup

The example here solves one issue by allowing Mary and John to work in parallel without blocking each other's work. However, at the end, the water temperature is back to 90 degrees! This is something that testing can resolve, which we won't cover in this course.

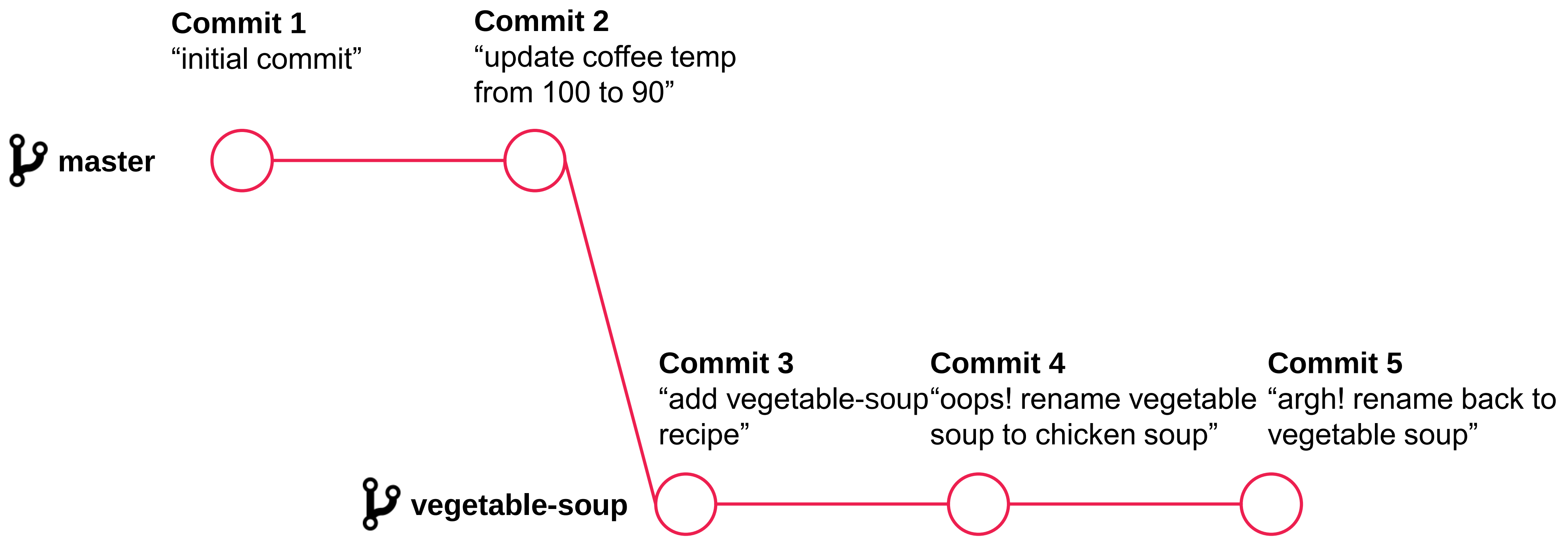
Git Branches

Another benefit of working in branches is having a clean branch history. The example below shows a messy branch history.


Commit 1
"initial commit"
Commit 2
"update coffee temp from 100 to 90"
Commit 3
"add vegetable-soup recipe"
Commit 4
"oops! rename vegetable soup to chicken soup"
Commit 5
"argh! rename back to vegetable soup"
master

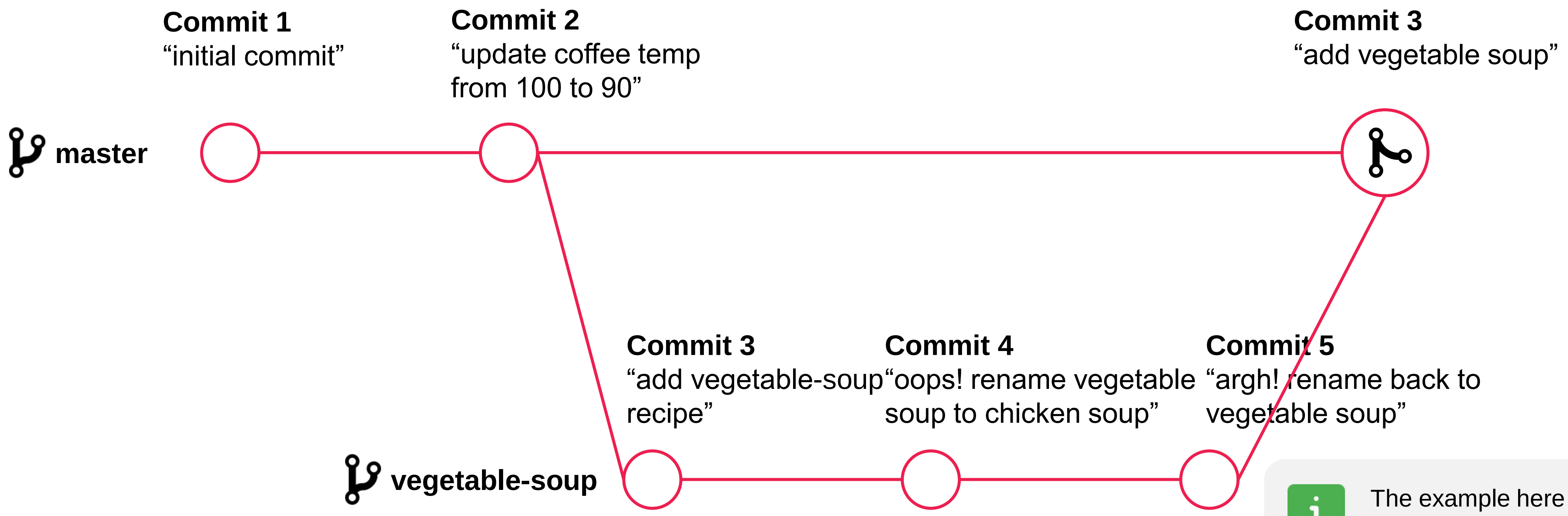
Git Branches

We can instead create a new branch called vegetable-soup and make our changes there.


Commit 1
"initial commit"
Commit 2
"update coffee temp from 100 to 90"
master
Commit 3
"add vegetable-soup recipe"
Commit 4
"oops! rename vegetable soup to chicken soup"
Commit 5
"argh! rename back to vegetable soup"
vegetable-soup

Git Branches

And when you are happy with your feature, you can **merge** that feature back to master by using "git merge --squash". Notice that the commits from the vegetable-soup branch is "squashed" into a single commit in the master branch.



Commit 1
"initial commit"
Commit 2
"update coffee temp from 100 to 90"
Commit 3
"add vegetable soup"
master
Commit 3
"add vegetable-soup recipe"
Commit 4
"oops! rename vegetable soup to chicken soup"
Commit 5
"argh! rename back to vegetable soup"
vegetable-soup

The example here uses squash and merge for cleaner branch history. More info here.

Git Branches – branch <branch_name>

Let's now look at making our changes using branches!
Use "git branch <branch_name>" to create a new branch from the branch you are currently on.

terminal

```
> git branch "vegetable-soup"
```

Use "git branch -a" to list all branches!

Git Branches – Checkout <branch_name>

After creating the new branch, you need to checkout the new branch to start using it.

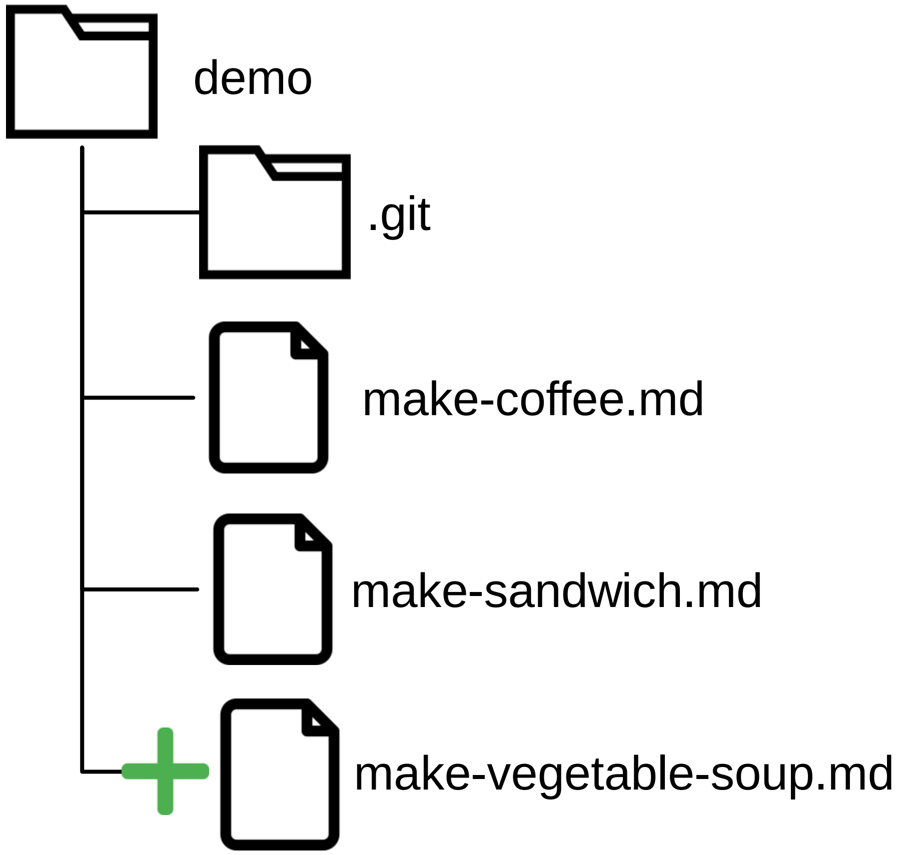
terminal

```
> git checkout “vegetable-soup”
```

Checkout is not only used for commits but for branches too!

Git Branches

Let’s make changes to the newly created branch. Let’s create a new file called “make-vegetable-soup.md” and insert the code below.



make-vegetable-soup.md

```
# Instructions

1. Boil 1L of water on the stove
2. Chop carrots, onions, celery and tomato
3. Add vegetables to the boiling pot of water
4. Boil for 30 minutes until vegetables are soft
5. Add salt to taste
6. Serve
```

Git Branches

Now let’s add and commit our changes.
We can visualize the branches by using “git log --all --graph --decorate”

terminal

```
> git add “make-vegetable-soup.md”
> git commit “add vegetable soup recipe”
> git log --all --graph --decorate
```

Git Branches – Checkout <branch_name>

Let’s return to “git checkout”. Git checkout does more than changing branches.
If you do “git checkout master” now, notice that “make-vegetable-soup.md” has disappeared!
Don’t worry, the file isn’t deleted. Git has just cleverly updated your repository to the commit or branch that you’ve checked-out.

terminal

```
> git checkout “master”
```

Git Branches – log

While we’re on master, let’s make some quick changes to master by adding a new recipe “make-chocolate-cake.md”. We’re just doing this for demonstration purposes only – in practice, it is recommended that you create a new branch called “chocolate-cake” if you want to add a new feature. After we’ve made our changes, let’s execute “git log --all --graph --decorate”. Notice how the branches have forked!

terminal

```
> git add “make-chocolate-cake.md”
> git commit –m “add chocolate cake recipe”
> git log --all --graph --decorate
```

make-chocolate-cake.md

```
# Instructions

1. Grab flour
2. Add chocolate
3. Chuck into oven
4. Boom! Cake!
```

Git Branches – Merge

Now let's bring the changes we've made to the "vegetable-soup" branch back to the "master" branch using "git merge <branch_to_bring_in>".
But before we do that, let's first check what changes we're actually bringing back to "master" by using "git diff <first_branch> <second_branch>". It is good practice to do this before any merge.

terminal

```
> git diff vegetable-soup master
> git checkout master # just make sure we are on the master branch
> git merge vegetable-soup # bring changes from vegetable-soup to master
> # notice that "make-vegetable-soup.md" has now been added to master!
> git log --all --graph --decorate # to view the graph again
```

Use "git branch -a" to list all branches!

Help me! I've executed "git log" but now it's taken me to this weird text editor screen in terminal and I can't exit!

Type "q" to exit. When there are long logs, Git will take you to the `less program` which makes the logs scrollable.

Git Branches – Exercise Time!

Excellent! We now have a way to work on features separately and combine them when ready.
Now it’s your turn to try! Repeat the steps from the previous slides.
A summarized list of steps is shown below. Execute them in order.
Note: “#” are comments – do not copy those into terminal

terminal

```
> git branch "vegetable-soup" # create a new branch called vegetable-soup
> git checkout "vegetable-soup" # checkout to the newly created branch
> # create a new file called "make-vegetable-soup.md" and copy code from slides
> git add "make-vegetable-soup.md" # add newly created file to the staging area
> git commit -m "add vegetable soup recipe" # commit the newly created file
> git checkout master # checkout to the master branch. Notice that the make-vegetable-soup.md file has disappeared!
> # create a new file called "make-chocolate-cake.md" and copy code from slides
> git add “make-chocolate-cake.md”
> git commit –m “add chocolate cake recipe”
> git log --all --graph --decorate # notice how there is a fork in branch history!
> git diff vegetable-soup master # view the difference between the vegetable-soup branch and the master branch
> git checkout master # just make sure we are on the master branch
> git merge vegetable-soup # bring changes from vegetable-soup branch into the current (master) branch
> git log --all --graph --decorate # observe how the branches are now merged
```

Git Collaboration

Until now, we've only touched on features of Git that interact with our local repository.
To allow collaboration to happen between developers, we need a copy of the repository stored somewhere where it can be accessed by others. This is called a "**remote repository**".
The remote repository can be hosted by yourself, but it is more common and not to mention easier to use hosts such as GitHub, Azure DevOps, Bitbucket and GitLab.

Remote Repository Hosts

GitHub

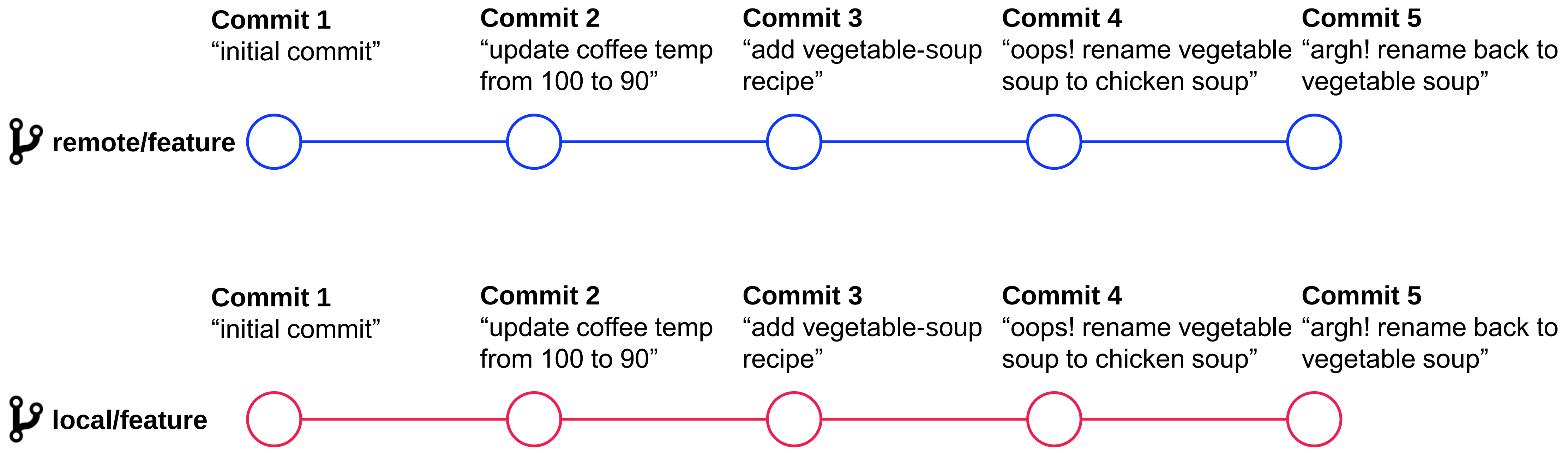
Azure DevOps

Bitbucket

GitLab

Git Collaboration – What's a remote repo?

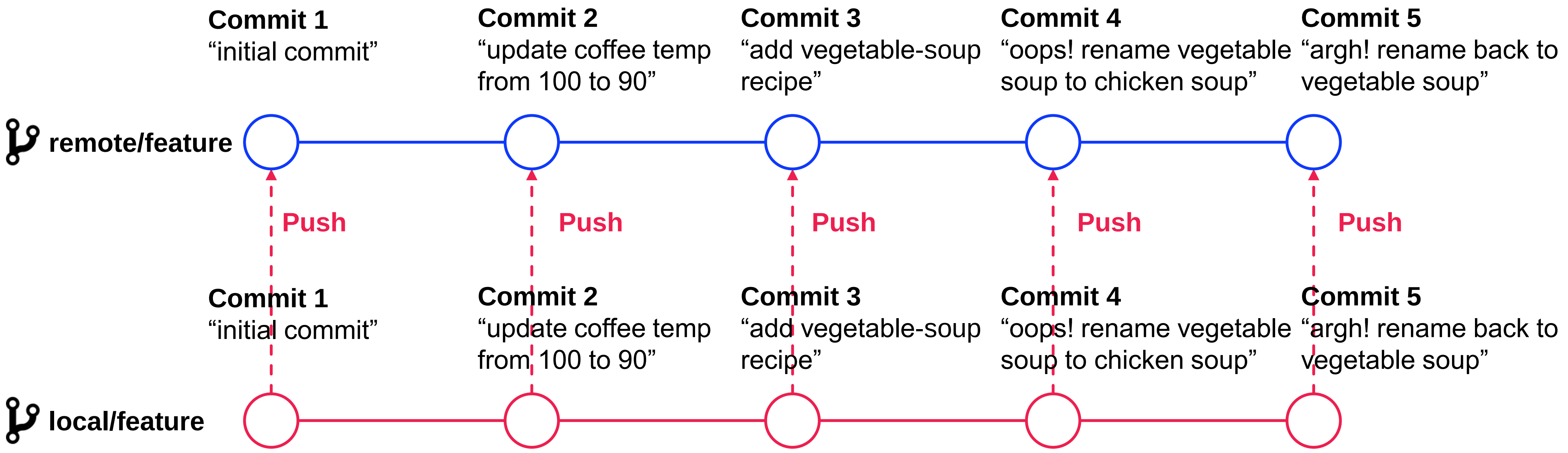
So what's a remote repository anyway?
A remote repository is just a copy of the local repository hosted somewhere else so that others can access it.



Commit 1
"initial commit"
Commit 2
"update coffee temp from 100 to 90"
Commit 3
"add vegetable-soup recipe"
Commit 4
"oops! rename vegetable soup to chicken soup"
Commit 5
"argh! rename back to vegetable soup"
remote/feature
Commit 1
"initial commit"
Commit 2
"update coffee temp from 100 to 90"
Commit 3
"add vegetable-soup recipe"
Commit 4
"oops! rename vegetable soup to chicken soup"
Commit 5
"argh! rename back to vegetable soup"
local/feature

Git Collaboration – Push

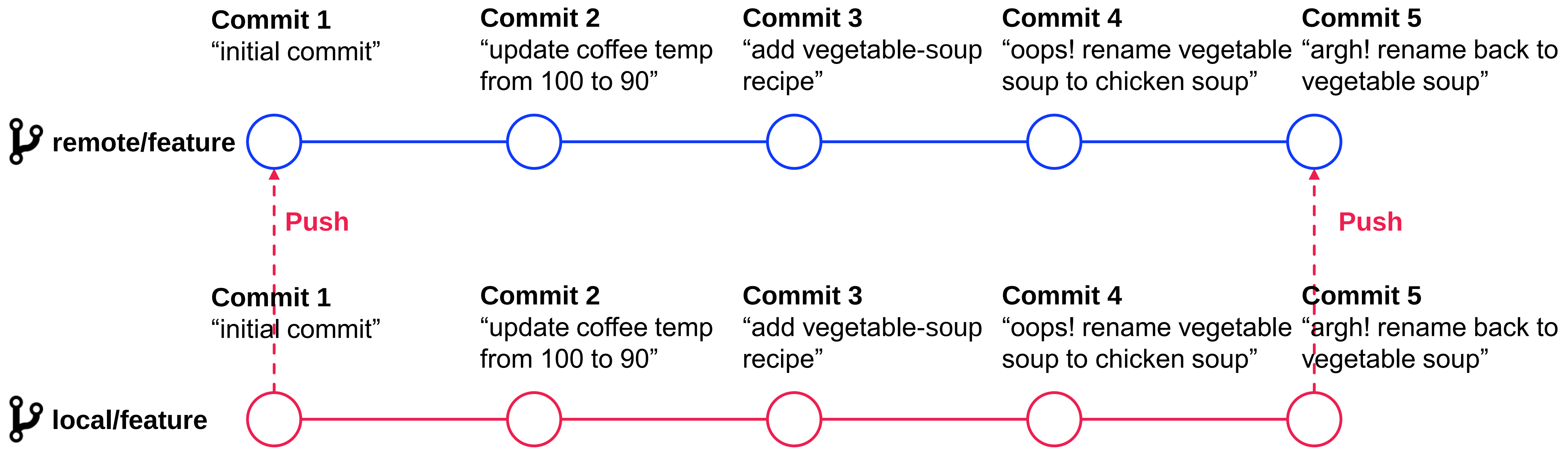
We can bring our changes to the remote repository by **pushing** our changes to the remote.



Commit 1
“initial commit”
Commit 2
“update coffee temp from 100 to 90”
Commit 3
“add vegetable-soup recipe”
Commit 4
“oops! rename vegetable soup to chicken soup”
Commit 5
“argh! rename back to vegetable soup”
remote/feature
Push
Push
Push
Push
Push
Commit 1
“initial commit”
Commit 2
“update coffee temp from 100 to 90”
Commit 3
“add vegetable-soup recipe”
Commit 4
“oops! rename vegetable soup to chicken soup”
Commit 5
“argh! rename back to vegetable soup”
local/feature

Git Collaboration – Push

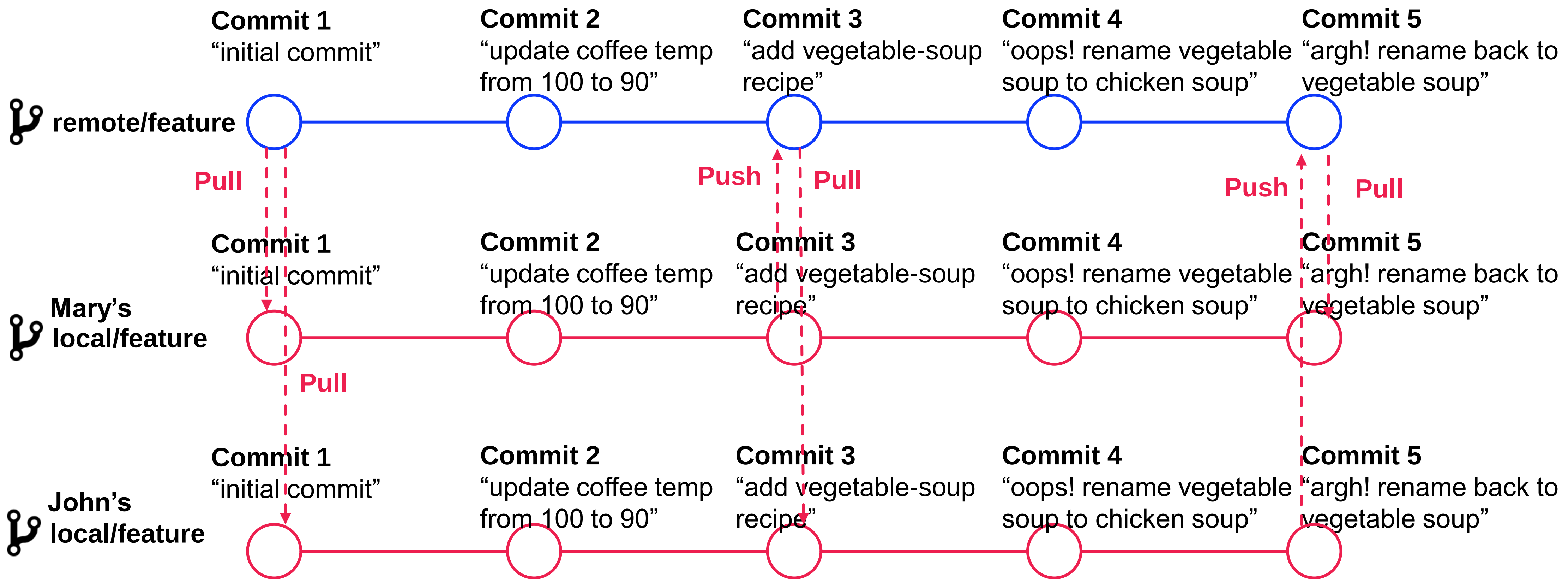
You don't need to push your changes on every commit. You could push your changes after a few commits. Pushing just brings the new commits into the remote repository.



Commit 1
"initial commit"
Commit 2
"update coffee temp from 100 to 90"
Commit 3
"add vegetable-soup recipe"
Commit 4
"oops! rename vegetable soup to chicken soup"
Commit 5
"argh! rename back to vegetable soup"
remote/feature
Push
Push
Commit 1
"initial commit"
Commit 2
"update coffee temp from 100 to 90"
Commit 3
"add vegetable-soup recipe"
Commit 4
"oops! rename vegetable soup to chicken soup"
Commit 5
"argh! rename back to vegetable soup"
local/feature

Git Collaboration – Pull

You can **pull** changes from the remote repository to your local repository. This feature is useful when working with others. Push and pulls are the crux of enabling collaboration between developers.



remote/feature
Commit 1
“initial commit”
Commit 2
“update coffee temp from 100 to 90”
Commit 3
“add vegetable-soup recipe”
Commit 4
“oops! rename vegetable soup to chicken soup”
Commit 5
“argh! rename back to vegetable soup”
Pull
Push
Pull
Push
Pull
Mary’s local/feature
Commit 1
“initial commit”
Commit 2
“update coffee temp from 100 to 90”
Commit 3
“add vegetable-soup recipe”
Commit 4
“oops! rename vegetable soup to chicken soup”
Commit 5
“argh! rename back to vegetable soup”
Pull
John’s local/feature
Commit 1
“initial commit”
Commit 2
“update coffee temp from 100 to 90”
Commit 3
“add vegetable-soup recipe”
Commit 4
“oops! rename vegetable soup to chicken soup”
Commit 5
“argh! rename back to vegetable soup”

Git Collaboration – Remote

Now we have the ingredients necessary to facilitate collaboration between developers. Let's add a reference to a remote repository by using "git remote add <name> <url>". The common practice is to always give the remote the name "origin" i.e. "git remote add origin <url>".

terminal

```
> git remote add origin "<url_here>"
```

What if we didn't have a local repository and wanted to pull from a remote repository?
You would use "git clone <url>" to clone an existing repository into your current working directory.

terminal

```
> git clone <url>
```

What URL do I use for my remote?

The facilitator will provide you with a URL that you can use. But if you are trying the exercises out on your own, you can create a free GitHub account and create a remote repository there. Here are instructions to do so: Creating a GitHub account, Creating a GitHub Repo.

Git Collaboration – Push

Now that we have added a remote, let's push our changes to the remote.
First, check which branch you are on. For this exercise, we will make sure we are in a **feature branch**, and push that feature to the remote.
Use "git push -u origin <branch_name>".

terminal

```
> git checkout vegetable-soup # let's checkout to a local feature branch first
> git push -u origin vegetable-soup  # -u flag is used to link the local branch with the
origin branch
> git show # notice how the remote has been added in a different colour
"origin/vegetable-soup"
> git branch -a # notice a new branch is added
```

The -u flag stands for "upstream" and is used once to link the local with the remote repository. That way you don't have to always tell Git which remote branch corresponds to your local branch.

Git Collaboration – Fetch and Pull

Now let’s make change directly in the remote repository to mimic a change from another developer.
Next, let’s fetch any changes from the remote repository that we can pull down using “git fetch”.
After that, let’s before a “git diff <local_branch_name> <remote_branch_name” to review the changes, and pull the changes using “git pull” if we are happy with it.

terminal

```
> # make a change directly to the remote repository to mimic change from another developer
> git fetch # fetch a list of changes to the branch you’re currently on
> git checkout <local_branch_name> # checkout to the local branch name if you’re not on it
> git diff <local_branch_name> <remote_branch_name> # review the differences before pulling
> git pull # pull the changes to your branch. You can run “git pull” independently of “git fetch”, but it is good practice to do fetch before pull.
```

It is good practice to perform a “git fetch” before a “git pull” especially when you’re working in teams. Another developer may have made changes which you may not be aware of. Performing a “git fetch” will help you be aware of what changes were made before pulling it to your repository.

“git pull” is a combination of “git fetch” and “git merge”.

Git Collaboration – Remote merge

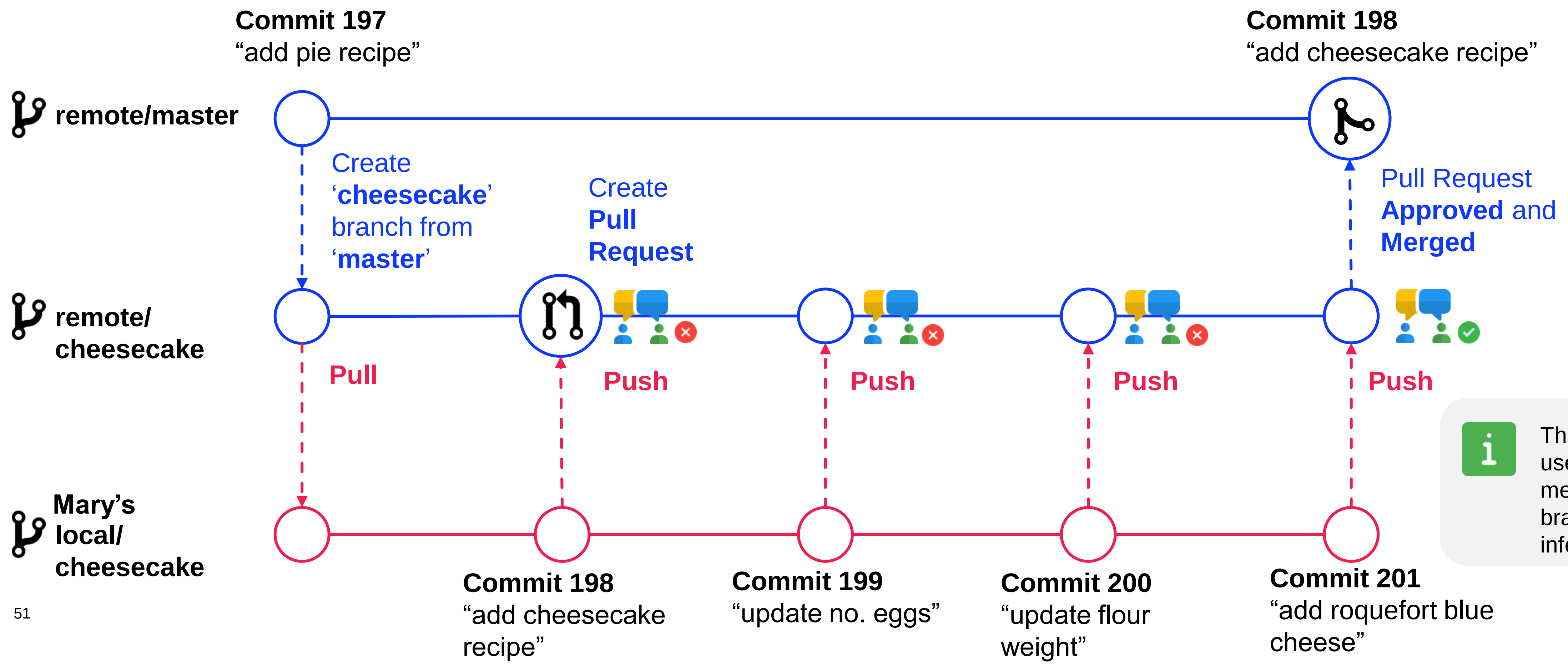
After changes are made to the **remote feature branch** and we are happy with it, we can merge it to the **remote master branch**. To do so, it is as simple as doing something like:

terminal

```
> git checkout “<feature_branch>” # checkout to the local feature branch
> git pull “<feature_branch>” # bring the latest version of the remote feature branch to
local feature branch
> git checkout master # checkout local master
> git pull master # bring the latest version of remote master to local master
> git merge “<feature_branch>” # merge local feature branch to local master branch
> git push origin master # push local master to the remote (origin) master
```

Git Collaboration – Pull Requests

However, Git hosts such as GitHub and Azure DevOps have features to support remote-to-remote branch merges called **Pull Requests**.



The example here uses squash and merge for cleaner branch history. More info here.

Git Collaboration – Exercise Time!

For this exercise, let's split into teams working on different cuisine features that constitutes a two-course menu:

Team	Last Name	Branch/Feature Name	Task Description
English Cuisine	A-G	english-cuisine	Create two markdown files: • Main `<dish_name>.md` • Dessert `<dish_name>.md`
Italian Cuisine	H-P	italian-cuisine	Create two markdown files: • Main `<dish_name>.md` • Dessert `<dish_name>.md`
Thai Cuisine	Q-Z	thai-cuisine	Create two markdown files: • Main `<dish_name>.md` • Dessert `<dish_name>.md`

May the best dish win!

Hint: First discuss different roles within the team, and how work should be divided.
See next slide for suggested roles.

Git Collaboration – Exercise Time!

Use the summarized list of steps below to guide you.

Note: “#” are comments – do not copy those into terminal

terminal

```
# ------- Role: Developer | Task: Creating and pushing new dishes to remote repo ------------
> git remote add origin “<url_here>” # add the remote if you haven't already
> git branch -a # notice a new branch is added
> git branch <cuisine-branch-name> # create a branch for the cuisine name if you haven't already
> git checkout <cuisine-branch-name> # checkout to the cuisine branch
> # develop changes for the cuisine dish
> git push -u origin <cuisine-brach-name> # push the local cuisine branch to the remote repository
> # create a pull request once the entire cuisine feature is complete, or wait for other dishes to arrive before creating a pull request
> # tag a reviewer for the pull request

# ------- Role: Reviewer | Task: Reviewing a pull request ------------
> # review the pull request -> provide comments and request for changes  if necessary
> # once changes look good -> approve the pull request
> # generally the creator of the pull request will merge (complete) the pull request, which places ownership on the creator. But other approaches could be used to.

# ------- Role: Reviewer | Task: Pulling dishes to local repo ------------
> git add remote origin “<url_here>” # add the remote if you haven't already
> git fetch # fetch all remote branches
> git checkout <cuisine-branch-name> # checkout to a local branch fetched from remote
```

Git Merge Conflicts

Merge conflicts can occur when you have two branches that change the same line of code and you merge one branch to another.

master

make-coffee.md

```
# Instructions - Pour over coffee

1. Bring cold water to a boil in a kettle
2. Grind the beans
3. Put a filter in the brewer and rinse with hot water
4. Add the grounds to the filter
5. Heat water to ~~50°C~~ 100°C
6. Slowly and steadily pour just enough water over the grounds
 to saturate them completely, starting from the middle and wor
king your way outward. This should take 3 to 4 minutes.
7. Carefully remove the filter
8. Serve and enjoy
```

coffee-greatness

make-coffee.md

```
# Instructions - Pour over coffee

1. Bring cold water to a boil in a kettle
2. Grind the beans
3. Put a filter in the brewer and rinse with hot water
4. Add the grounds to the filter
5. Heat water to ~~50°C~~ 90°C and 95°C
6. Slowly and steadily pour just enough water over the grounds
 to saturate them completely, starting from the middle and wor
king your way outward. This should take 3 to 4 minutes.
7. Carefully remove the filter
8. Serve and enjoy
```

Git Merge Conflicts

Let’s demonstrate this by making a change directly to a file in the remote repository to mimic a change from another developer. And let’s also make a different change to the same file and the same line to our local repository.

terminal

```
> # make a change directly to a file in the remote repository to mimic change from another developer
> # make a different change to the same file in the local repository
> git fetch # fetch a list of changes to the branch you’re currently on
> git diff <local_branch_name> <remote_branch_name> # review the differences before pulling
> git pull # pull the changes to your branch – this should throw a merge conflict!
> git status
```

? What if I want to cancel the merge?

Use “git merge --abort”. This will exit the merge process and return the branch to a state before the merge.

? I didn’t do a merge, so why is Git complaining there is a merge conflict?

Merge conflicts typically arise when executing “git merge” between two branches. We are experiencing it now in this example because “git pull” is a combination of “git fetch” and “git merge”.

? But didn’t you say that merge conflict only happens when changes are made to same line of code but different branches?

Yes, but the remote branch is a separate branch from the local branch. This is a common issue when two developers are working on the same branch. Communication beforehand is key to avoid this issue.

Git Merge Conflicts – How to resolve

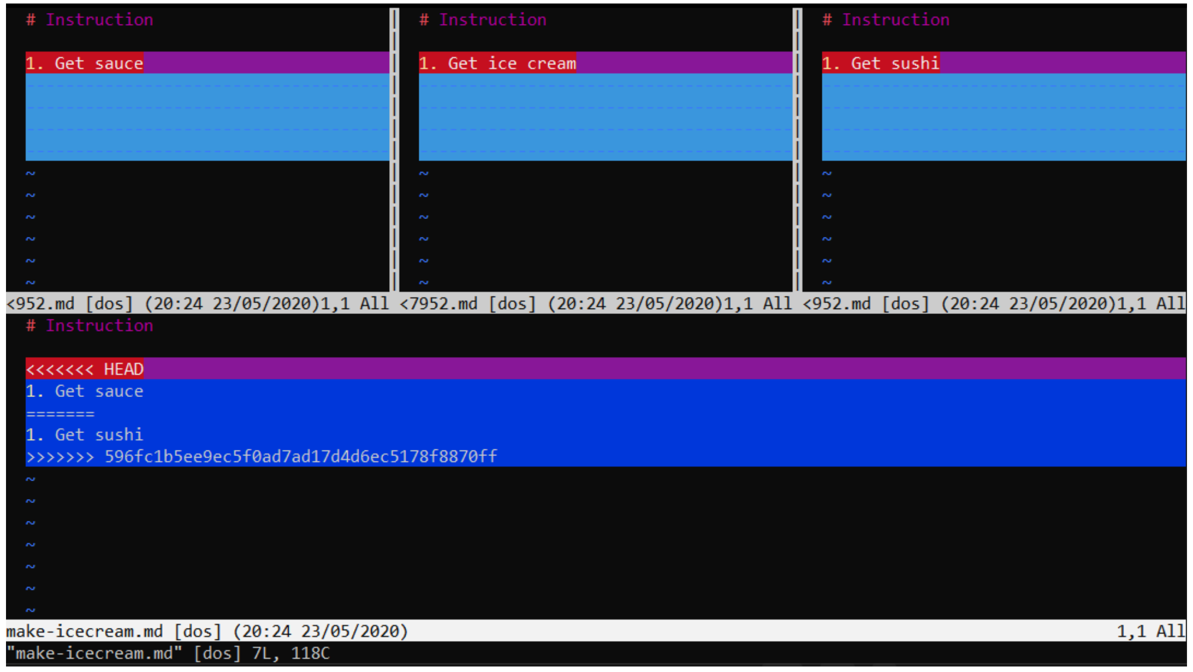
Without the use of tools, the way to resolve merge conflicts is to compare line-by-line what the conflicts are between two files and make the necessary changes. That can be very tedious.

Luckily, there are tools available to help us. In the terminal, we can use “git mergetool” to help us resolve merge conflicts. However, mergetool does not have the most intuitive UI.

terminal

```
> git mergetool # mergetool will prompt to use vimdiff to compare files. Hit return to use vimdiff. You should see something similar to the right.
> # type :qa! and hit enter to escape vimdiff
```

If you are serious about using mergetool, you can learn more about it here. However most modern code editors have built-in merge tools which are much more intuitive to use.



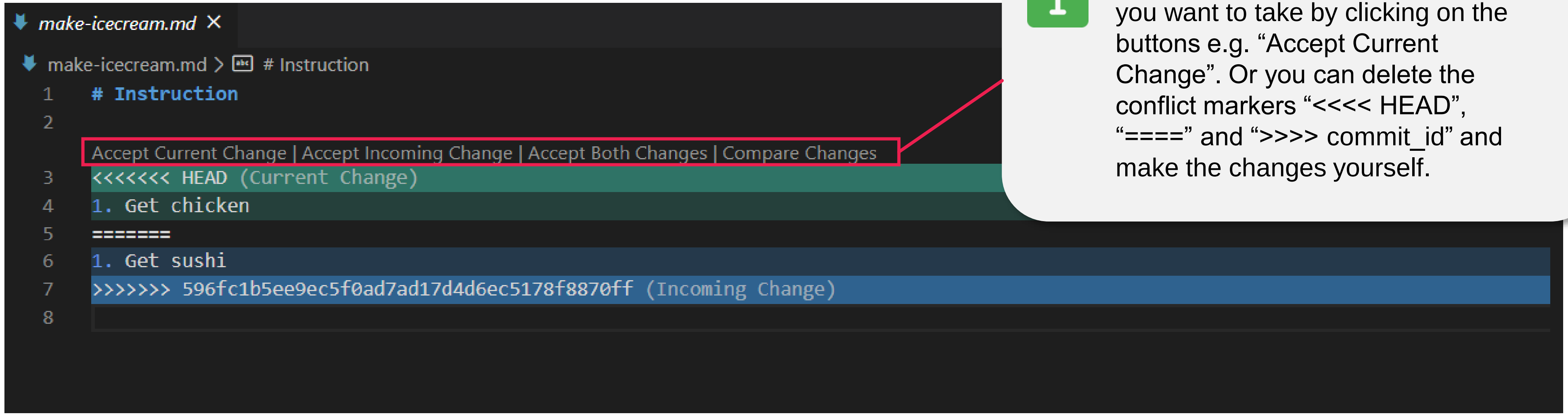
```
# Instruction                      # Instruction                      # Instruction

1. Get sauce                       1. Get ice cream                   1. Get sushi







<952.md [dos] (20:24 23/05/2020)1,1 All <7952.md [dos] (20:24 23/05/2020)1,1 All <952.md [dos] (20:24 23/05/2020)1,1 All
# Instruction

<<<<<<< HEAD
1. Get sauce
=======
1. Get sushi
>>>>>>> 596fc1b5ee9ec5f0ad7ad17d4d6ec5178f8870ff
~
~
~
~
~
~
~
make-icecream.md [dos] (20:24 23/05/2020)                                                  1,1 All
"make-icecream.md" [dos] 7L, 118C
```

Git Merge Conflicts – How to resolve

Modern code editors come with easy-to-use versions of “merge tool” to help developers resolve merge conflicts.

Open up **Visual Studio Code** and you should see something similar to below.

- HEAD: the branch or commit you have checked-out
- <commit_id>: the incoming change


make-icecream.md
make-icecream.md > # Instruction
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes

```
# Instruction

<<<<<<< HEAD (Current Change)
1. Get chicken
=======
1. Get sushi
>>>>>>> 596fc1b5ee9ec5f0ad7ad17d4d6ec5178f8870ff (Incoming Change)

```

You can easily decide what action you want to take by clicking on the buttons e.g. “Accept Current Change”. Or you can delete the conflict markers “<<<< HEAD”, “====” and “>>>> commit_id” and make the changes yourself.

Git Merge Conflicts – How to resolve

Once the issue has been resolved, we need to let Git know by executing “git add <filename>” to mark the resolution.

Then we tell Git to continue the merge process by using “git merge --continue”. Git will prompt for a merge message, just hit enter to use the default.

And then we send our changes back to the remote using “git push”.

terminal

```
> git add make-icecream.md # mark the resolution
> git merge --continue # git will prompt for a merge message.
> git log --all --graph --decorate # observe how the branches are now merged
> git push # push changes to remote
```

Git Merge Conflicts – Exercise Time!

Recreate a Merge Conflict by using the steps below.
Note: “#” are comments – do not copy those into terminal

terminal

```
> # make a change directly to a file in the remote repository to mimic change from another developer
> # make a different change to the same file in the local repository
> git fetch # fetch a list of changes to the branch you’re currently on
> git diff <local_branch_name> <remote_branch_name> # review the differences before pulling
> git pull # pull the changes to your branch – this should throw a merge conflict!
> # resolve the merge conflict using either mergetool or VS Code
> git add make-icecream.md # mark the resolution
> git merge --continue # git will prompt for a merge message.
> git log --all --graph --decorate # observe how the branches are now merged
> git push # push changes to remote
```

Modern IDEs

Most Integrated Development Environments (IDEs) come with Git and offers varying levels of ease when using Git. Each IDE offers different features and limitations of interacting with Git. Thus it is important that you understand the fundamentals of Git rather than relying on the features that the IDE provides.
We shall do a demonstration using Visual Studio Code, a popular light-weight IDE.

Visual Studio Code

Sublime Text

Atom

PyCharm

Resources

Resource	URL
Git cheat sheet	https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf
Git Pro Book	https://git-scm.com/book/en/v2
Git Handbook	https://guides.github.com/introduction/git-handbook/
GitHub Tutorials	https://guides.github.com/activities/hello-world/
Azure DevOps Tutorials	https://docs.microsoft.com/en-us/azure/devops/repos/git/?view=azure-devops
BitBucket Tutorials	https://www.atlassian.com/git/tutorials
GitLab Tutorials	https://docs.gitlab.com/ee/gitlab-basics/

Course Complete!

References

Resource	URL
Icons from Pixel Perfect	https://www.flaticon.com/authors/pixel-perfect
Icons from Freepik	https://www.flaticon.com/authors/freepik

Telstra Purple

Thank you