version control : git + GitHub

Marco Morales
marco.morales@columbia.edu

Nana Yaw Essuman
nanayawce@gmail.com

GR5069: Applied Data Science
for Social Scientists

Spring 2025
Columbia University

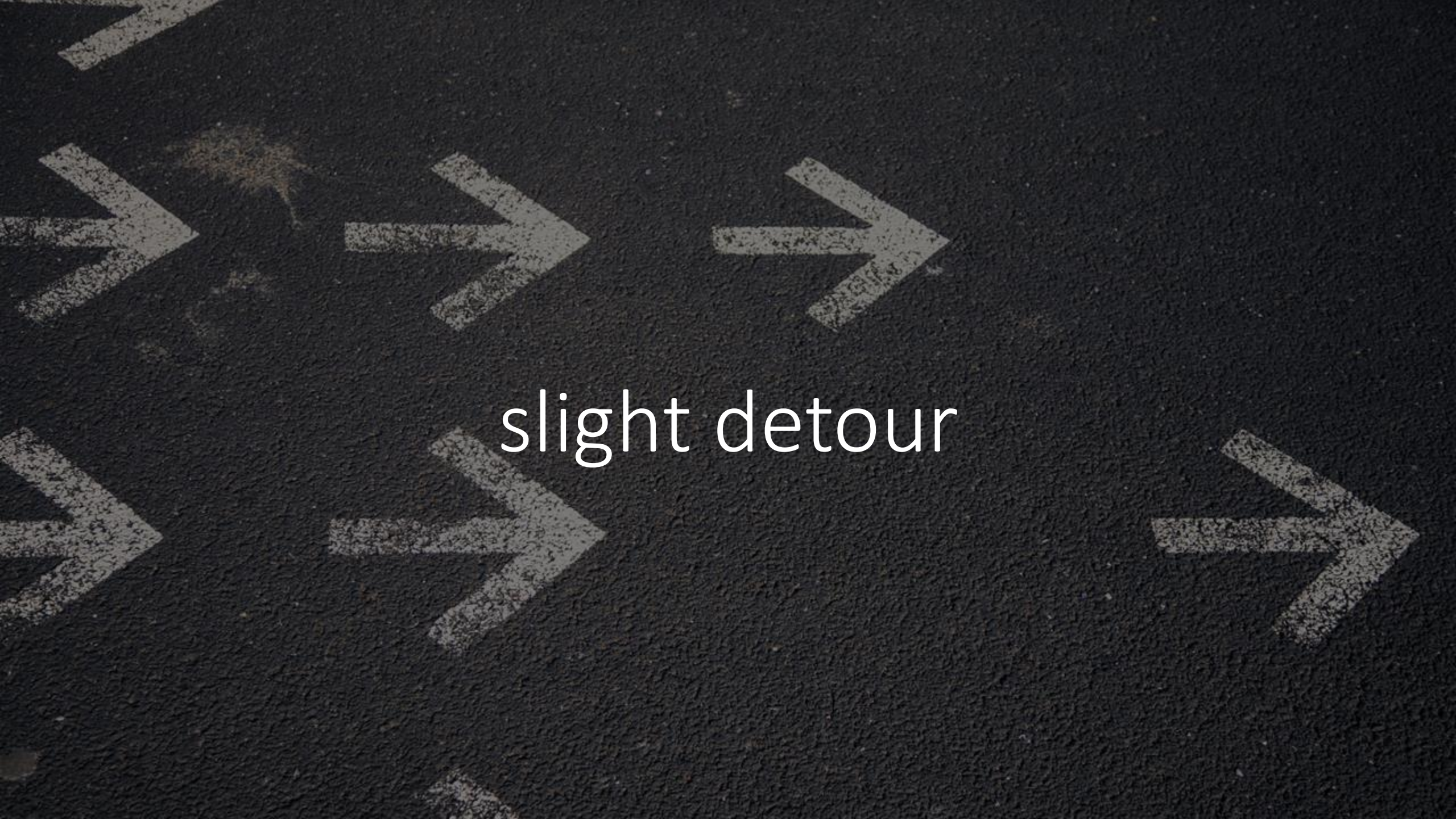
housekeeping

housekeeping!

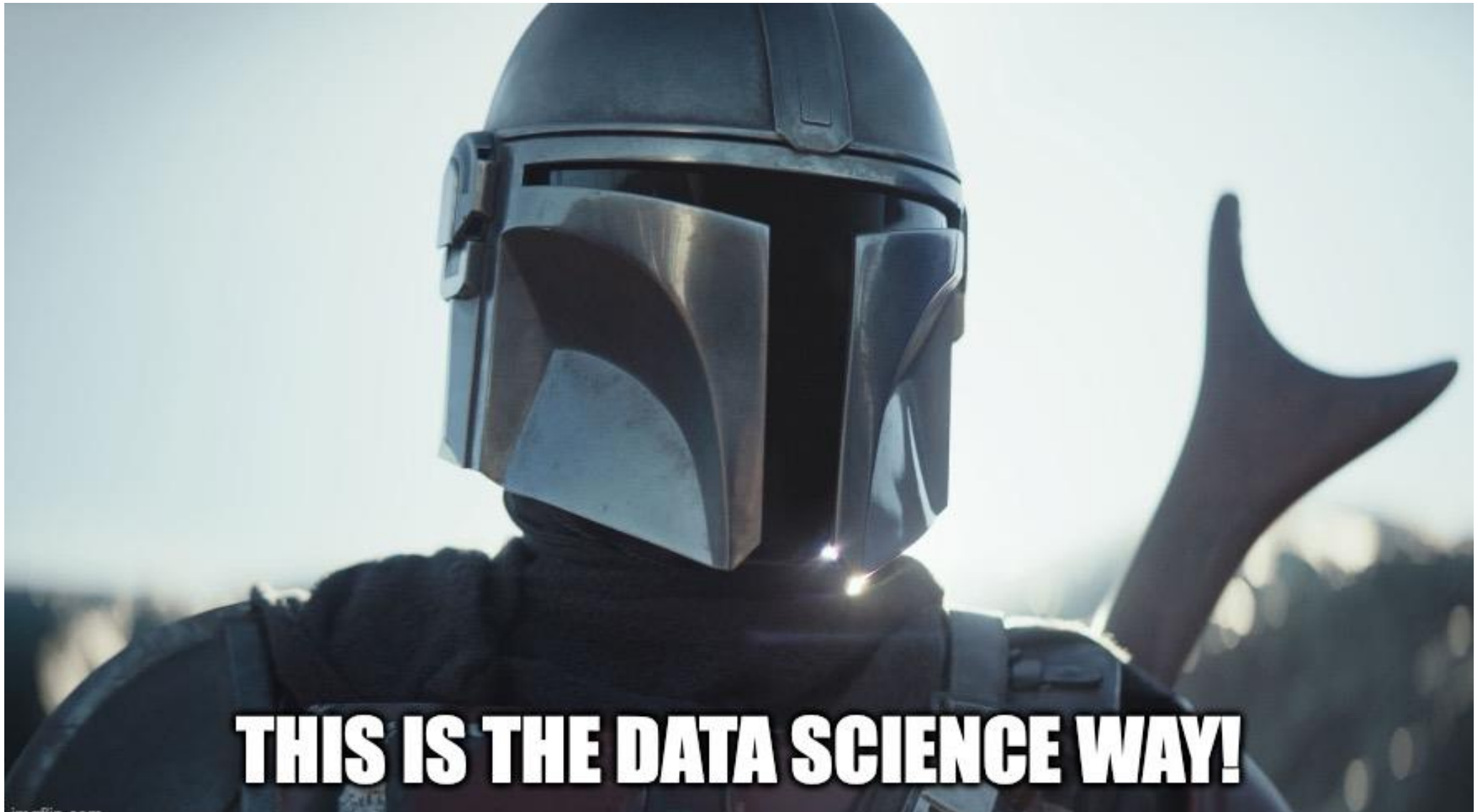
- invites to the course's **Slack** workspace have been sent out to your Columbia email. **Accept invite to Slack**

- please **submit** your **GitHub handle** in the Courseworks assignment.

- you'll receive invites to AWS and Databricks as needed. **Follow the instructions to accept the invites**.


slight detour

recap: collaborate, collaborate, collaborate!



behold messaging systems!

- your team will likely use a **messaging system** to collaborate
- important that you get used to it!
- we'll use Slack for this class as a real-world simulation

slack

remember the 15 min rule?

- first, **try to figure it out** on your own
- if you haven’t figured it out **after 15 minutes**, ask your teammates / classmates for **help**
- never helpful to go down rabbit holes on your own!
- likely that someone else has **faced and solved your question/problem** and may help you with it
- post your question in the **appropriate Slack channel**!

a bit of Slack etiquette

- mention people (i.e. **@marco-morales**) when speaking to them directly on a channel
  - people will not be notified unless you mention them
- use **@channel** and **@here** with care
  - **@here** notifies all people currently <u>active</u> in the channel
  - **@channel** notifies all members of the channel
  - **@everyone** notifies all members of the workspace
- be mindful of other people's time and schedules (and time zone differences!)

a few useful Slack gimmicks!

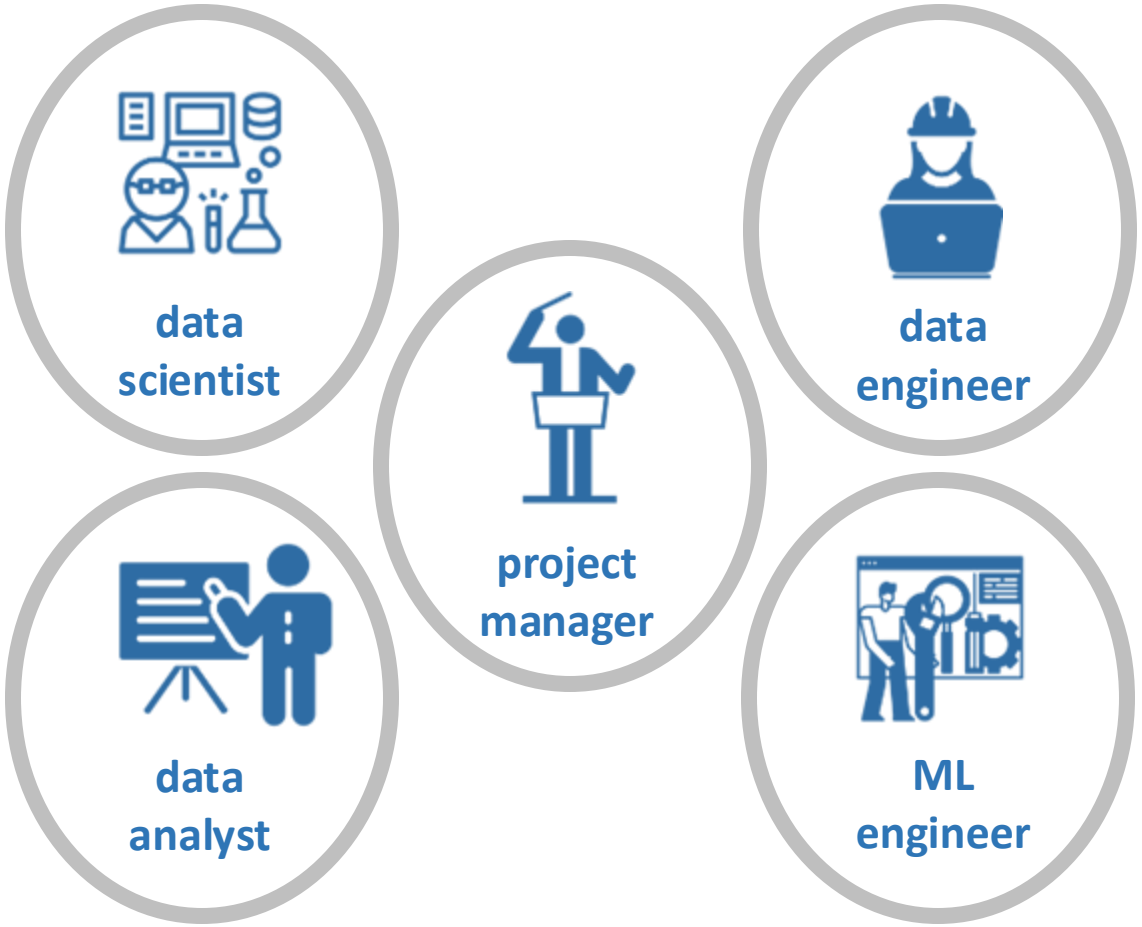
- Slack works on **Markdown**, so it's simple to format the text of your messages
- easy to share snippets of **code**, **text**, **data**
- can **edit messages after sending** them (nice alternative for documenting)
- **integrations** with other apps


lets get started!

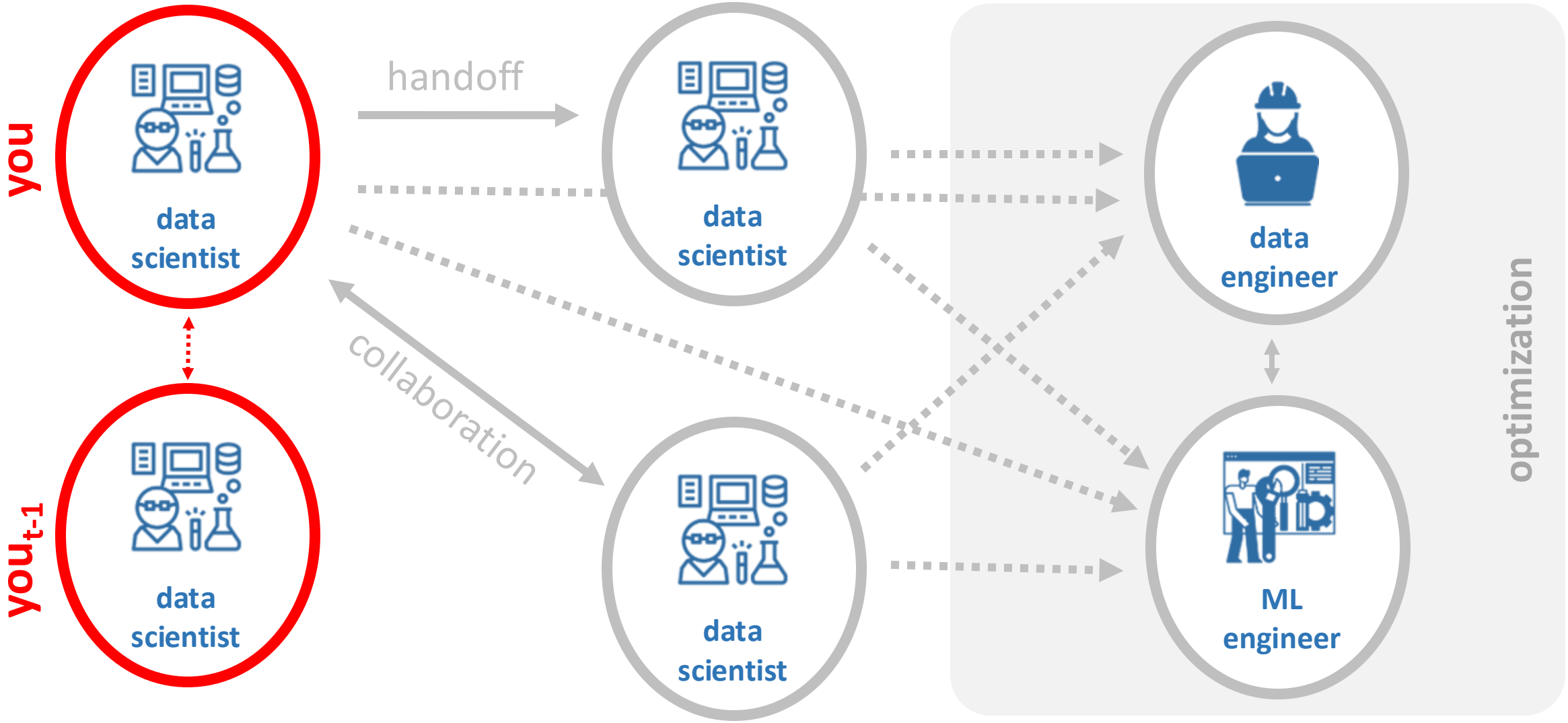
recap: what does a Data Scientist do?

Instagram vs **reality**

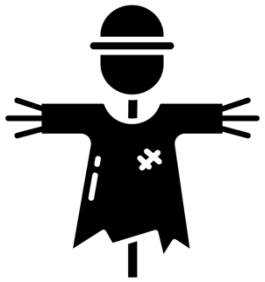
recap: collaboration to develop Data Products



workflow collaboration in Data Science



recap: iteration to build Data Products



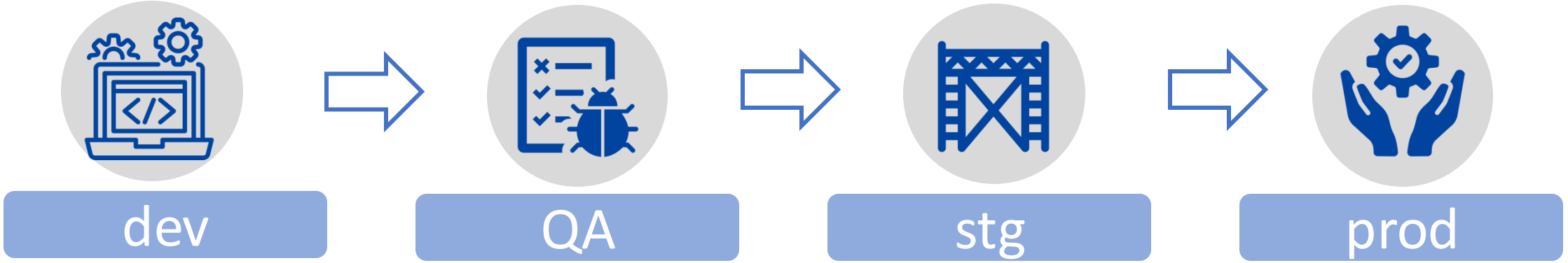
start small
(MVP)

fail fast

scale up

iterate

working environments to build Data Products



development

quality assurance / testing

staging

production

operational concepts in Data Science

portability

anyone should be able to **pick up where you left off** from any machine

replicability

anyone should be able to arrive at your **same results**

scalability

your prototype should also work for **larger data sets** and/or be on the path of **automation**

operational concepts in Data Science

	portability	replicability	scalability
what	• flexible references • structured and documented code • replicate original environment	• documentation: data, software, hardware, environments • commented code • no manual processes	• high quality code • flexible functions • modularized code
why	• seamless handoff • frictionless transitions across environments	• seamless examination, review or validation • cordial troubleshooting • harmonious optimization	• simplified review and validation • reduce time optimizing, automating and deploying

why do we start here?

all **Data Products** involve code, and

code is the backbone of all Data Products

why version control?

1. keeps **“snapshots”** of your code over time
2. expedites the process to **debug** code (yours and your team’s)
3. regulates **team collaboration** (everyone can see who changed what! + “air traffic control”)
4. supports the **lifecycle of a Data Product**
5. enables **reproducibility**, **portability**, and **scalability**

central to any **activity** that involves **code**

Software Engineering

Data Engineering

Machine Learning Engineering

Data Science

many flavors, but we'll focus on these two

version control software

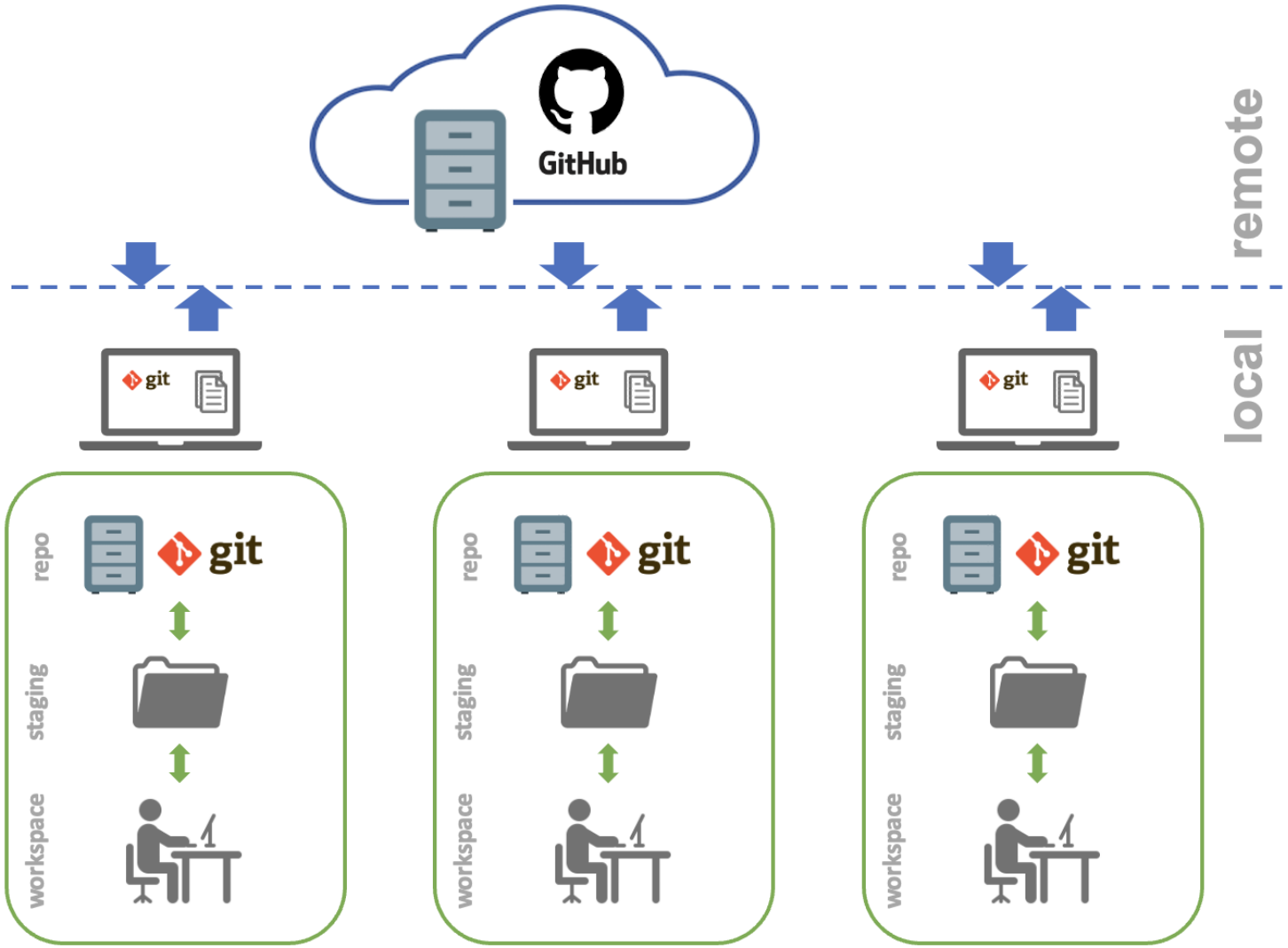
open-source cloud service

two minutes to make sure you've:

1) downloaded/installed git

2) created a GitHub account

an ideal **version control setup**

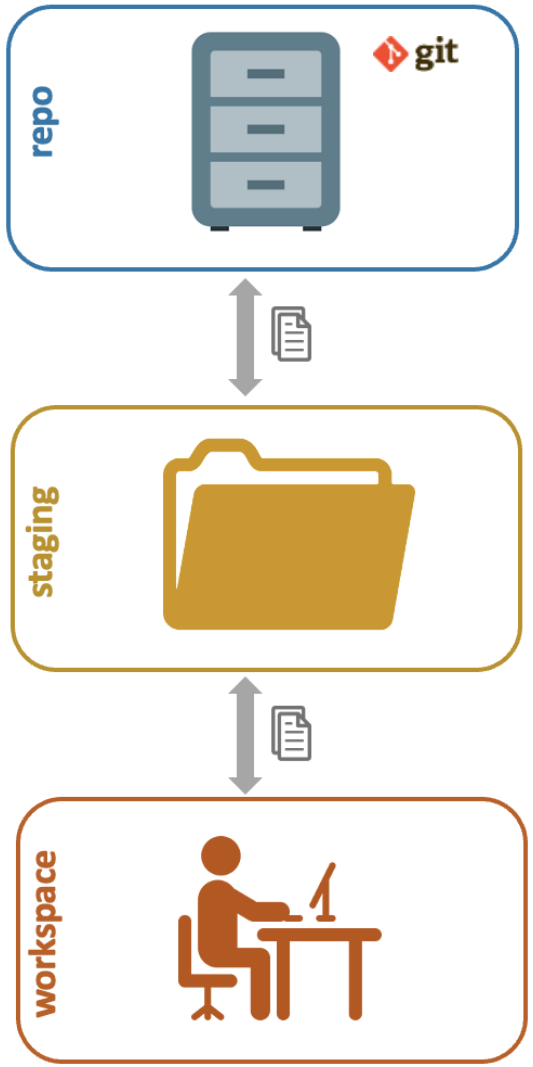


git locally

recap: what was this **git** thing?

- git is a **version control software**
  - installed “locally” on your computer (or virtual machine or computing platform)
  - keeps snapshots of your (coding) work
- helps with
  - “time travel” (insert your favorite “Back to the future” gif here)
  - keep collaboration organized when multiple people are working on the same project
- a vehicle to be nice to your fellow collaborators (and to the you of the future)

git: a mental model



git, meet your new user!

- set your **username** and **email address**

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

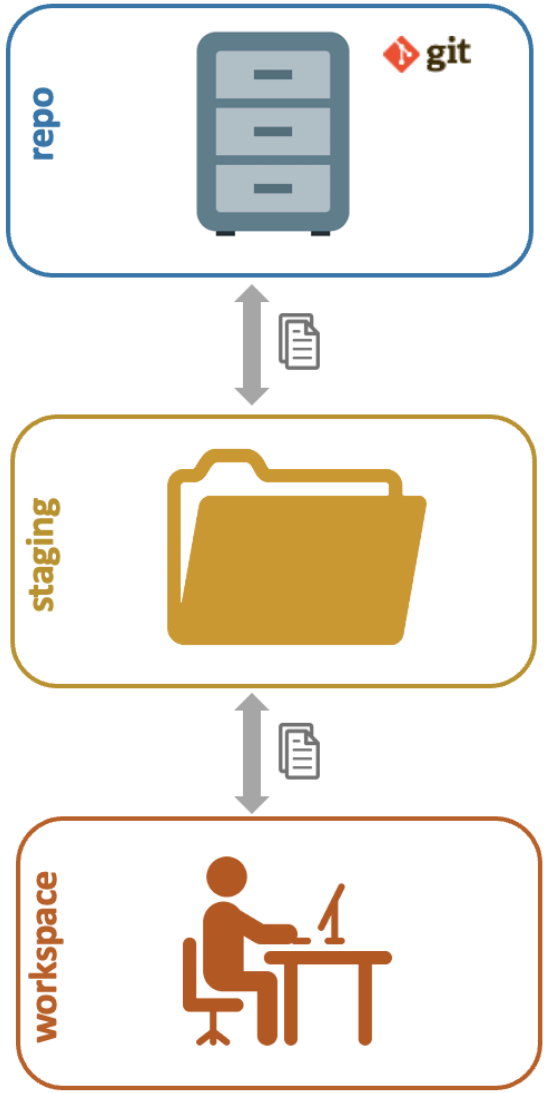
- **verify** that information was successfully entered

```
$ git config –-list
```

- this information gets baked in your commits

ProTip: other useful information (e.g. proxy settings) also goes on git config

now, turn your folder structure into a git repo



now, turn your folder structure into a git repo

- go to the **root** of your project and initialize the repo

```
$ git init
```

- there are **files you never want tracked** by git (e.g. log files, access keys), even by mistake

- that's the purpose of a .gitignore file

now, turn your folder structure into a git repo

- from the root of your local repository, create a .gitignore file

```
$ touch .gitignore
$ echo > .gitignore
```

(Mac)
(Windows)

- add files, file types and folders you want git to ignore

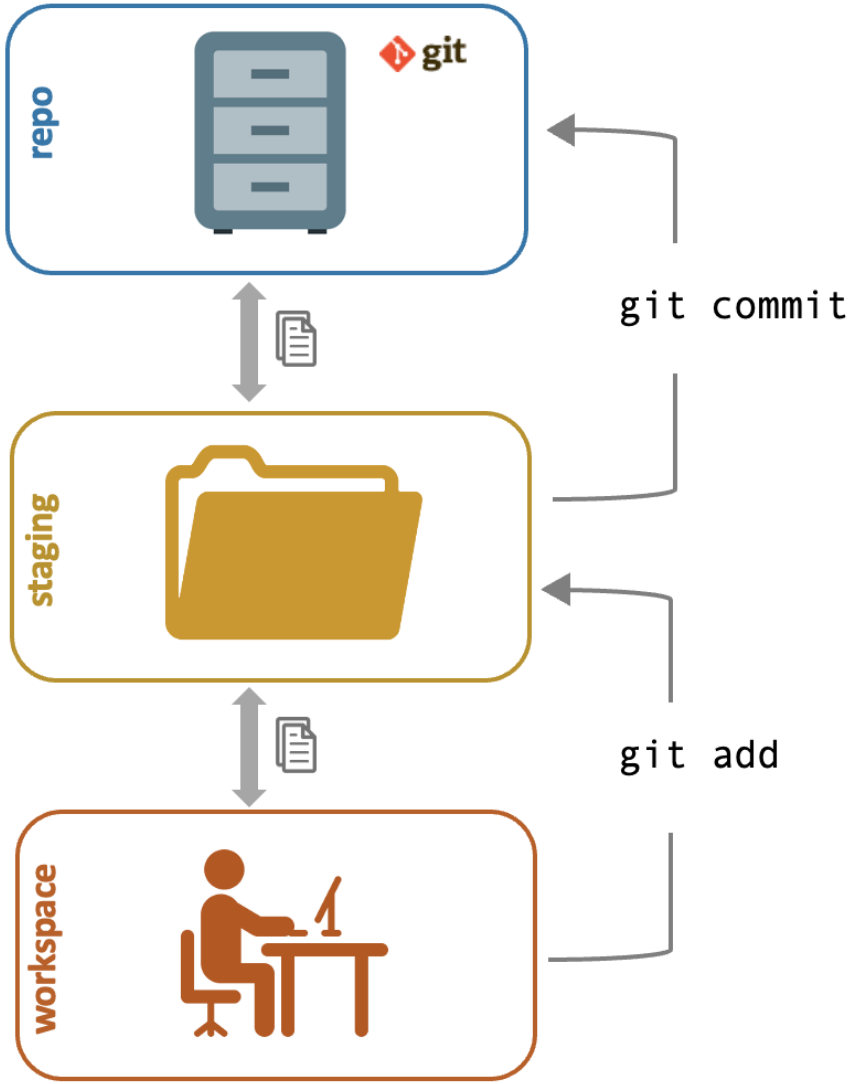
what do you add to a **.gitignore** file?

```
# OS generated files #
*.DS_Store

# Jupyter Notebook
.ipynb_checkpoints
```

```
# RStudio files
*.Rproj.user/

# all data folders
data/
```

ProTip: further info/templates: https://github.com/github/gitignore

your basic git workflow



your basic git workflow

- indicate a file to be tracked by git

```
$ git add samplefile.R
```

- verify what’s being tracked

```
$ git status
```

- commit your tracked files (with an informative message)

```
$ git commit -m "Commit initial files"
```

a few confusing things about git

- a file will be committed **exactly** as it was when you git add-ed it
- if you change the file after you git add it and want to commit the new changes, you need to git add again before the git commit
- use git status to assess what's being staged and commited

git workflow **ProTips**

- NEVER use `git add .`
- use `git status` often as validation
- only `add` and `commit` source files
  - omit files you can reproduce using source files
- `commit` **small chunks of logically grouped changes**
  - you may want to undo a change, and only that change
- `commit` with **informative** (imperative mood) **messages**
  - [*this commit will*] Rename income variable

quick detour: what is a **branch** in git?

a **divergence** from the main line of development

that **isolates** development work
without affecting other branches

---

a branch can **merge** into another branch

repos always have a **default branch**

git workflow **ProTips**

- current best practice is to use main for your default branch; used to be master
- by default, git will create a main branch after your first commit
- easy tor rename your branch to main

```
$ git branch -M main
```

- for a permanent solution (in git >= 2.28)

```
$ git config --global init.defaultBranch main
```

push globally
(to GitHub)

recap: what was this **GitHub** thing?

- **GitHub** is a **cloud service** that hosts git repositories
  - lives in the cloud
  - understands the git dialect!
  - can speak with multiple git users simultaneously
- helps with
  - persisting repository storage (your dog cannot eat your repo!)
  - synchronizing work
  - minimizing risk of people stepping on each other's toes (while working on the same project)
  - seamless transition between environments (dev > qa> staging > prod)

first, create a GitHub repo

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

Repository template

No template

Start your repository with a template repository's contents.

Owner *

marco-morales

Repository name *

mytestrepo

mytestrepo is available.

Great repository names are short and memorable. Need inspiration? How about friendly-memory ?

Description (optional)

Public
Anyone on the internet can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

Initialize this repository with:

Add a README file
This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. Learn more about ignoring files.

Choose a license

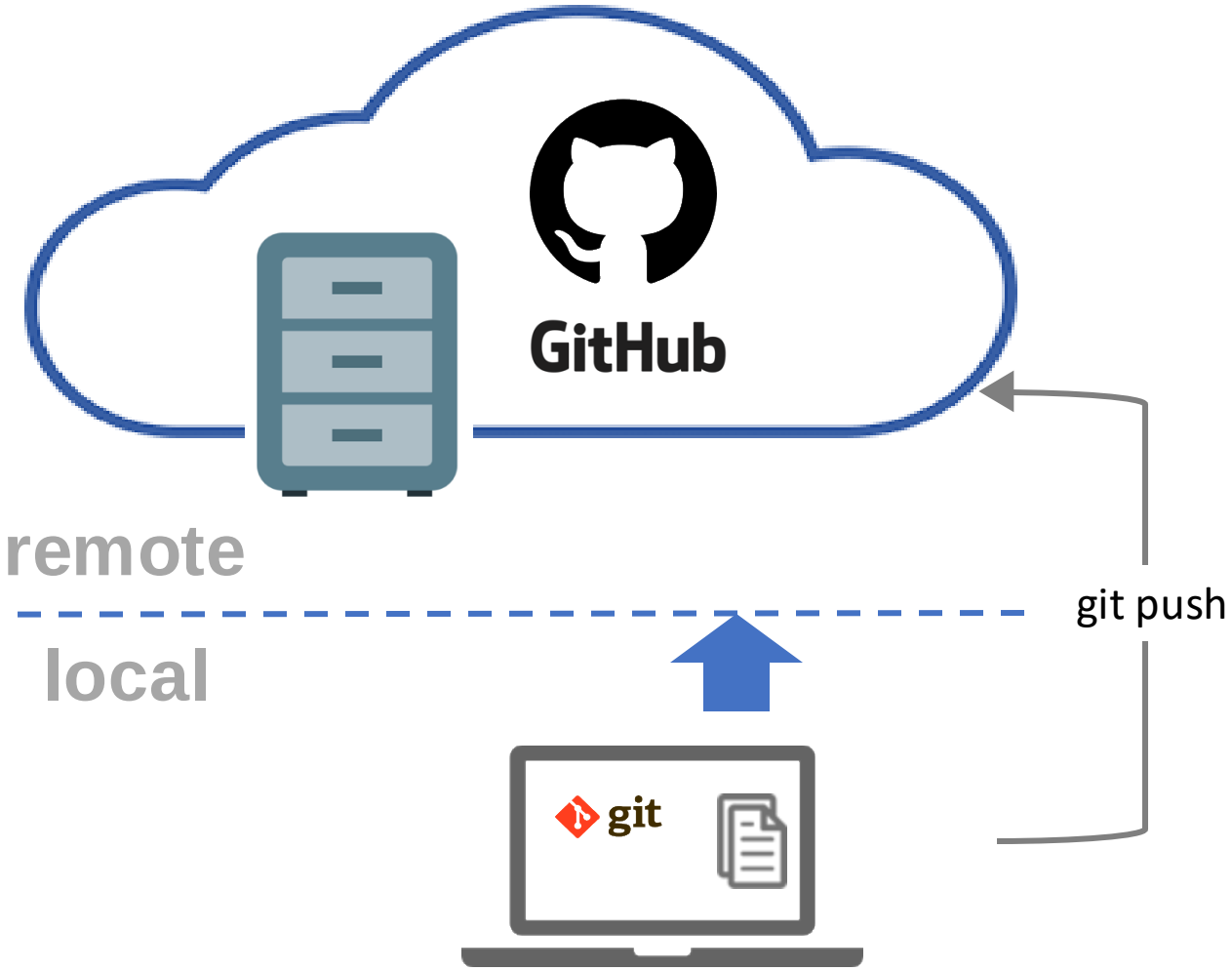
License: None

A license tells others what they can and can't do with your code. Learn more about licenses.

You are creating a public repository in your personal account.

Create repository

then, push to that GitHub repo



then, push to that GitHub repo

- tell git the location of the remote GitHub repo you just created (typically nicknamed “origin”)

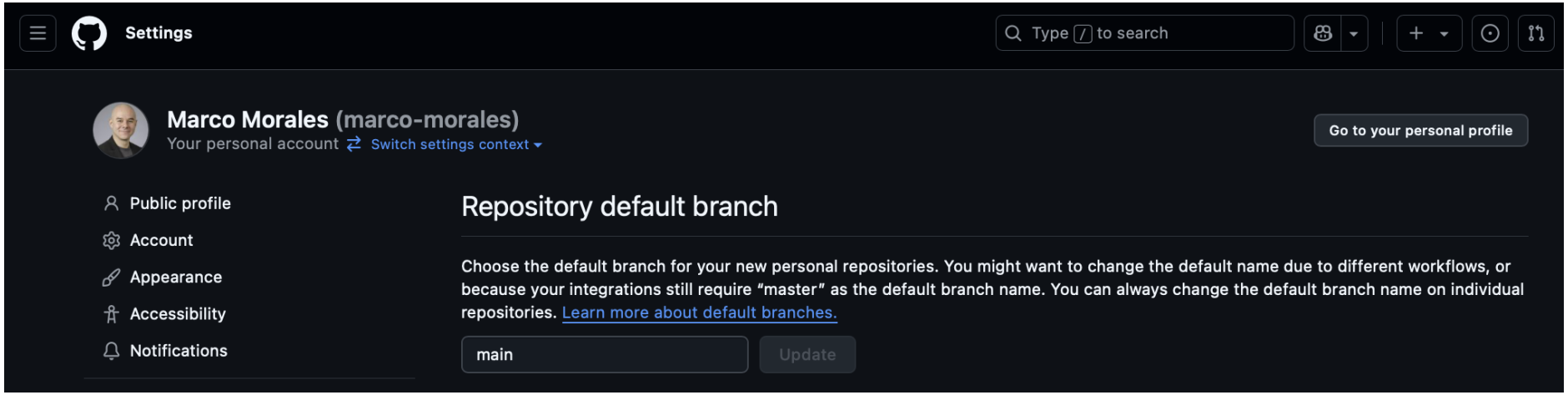
```
$ git remote add origin https://github.com/marco-morales/testrepo.git
```

- send committed files to your GitHub (“origin”) repo from your local git branch (“main”)

```
$ git push -u origin main
```

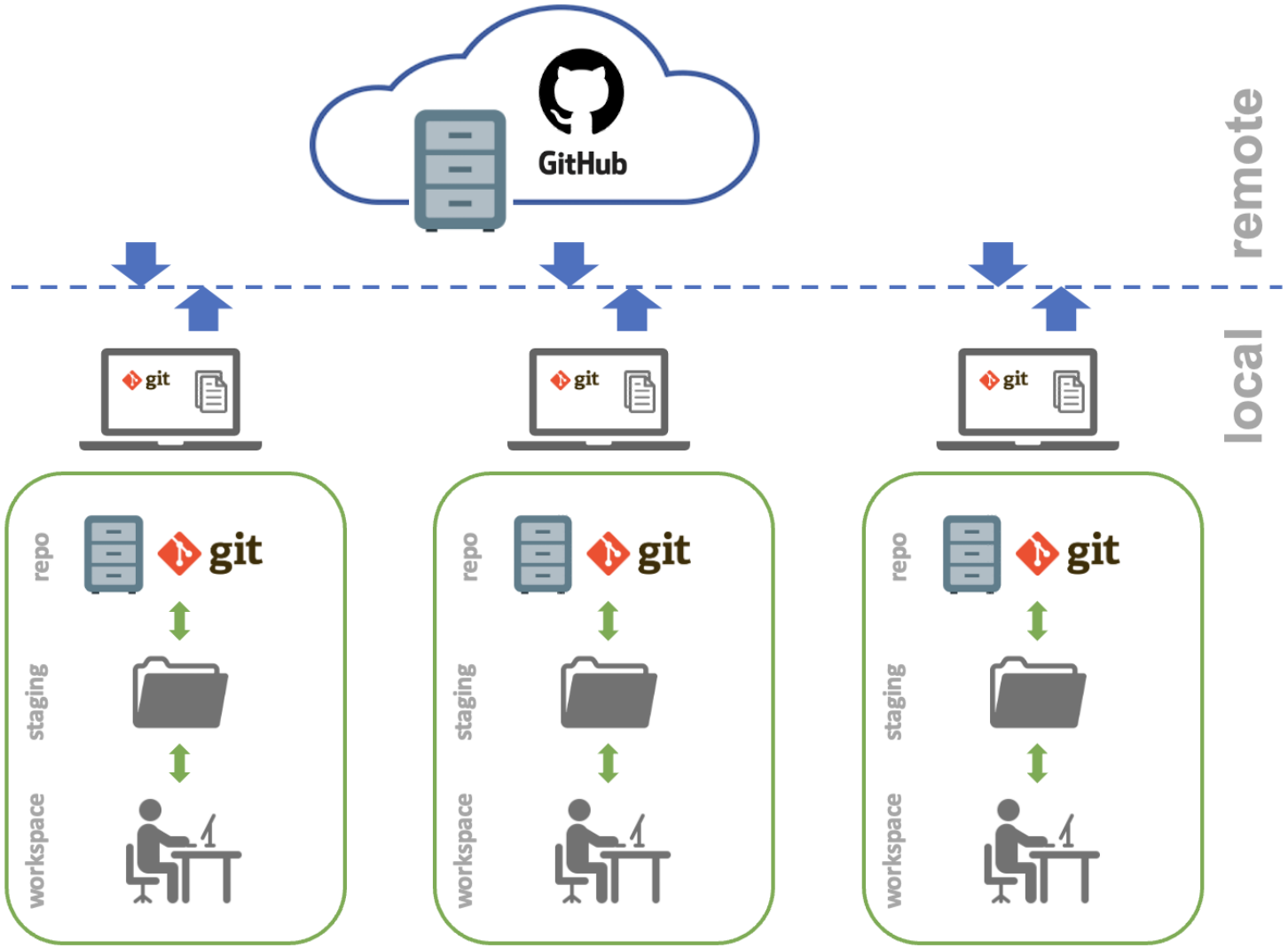
GitHub workflow ProTips

- current best practice is to use main for your default branch; used to be master
- by default, GitHub will create a master branch after you first create a repo if you do not change defaults
- easy to change permanently in your GitHub settings

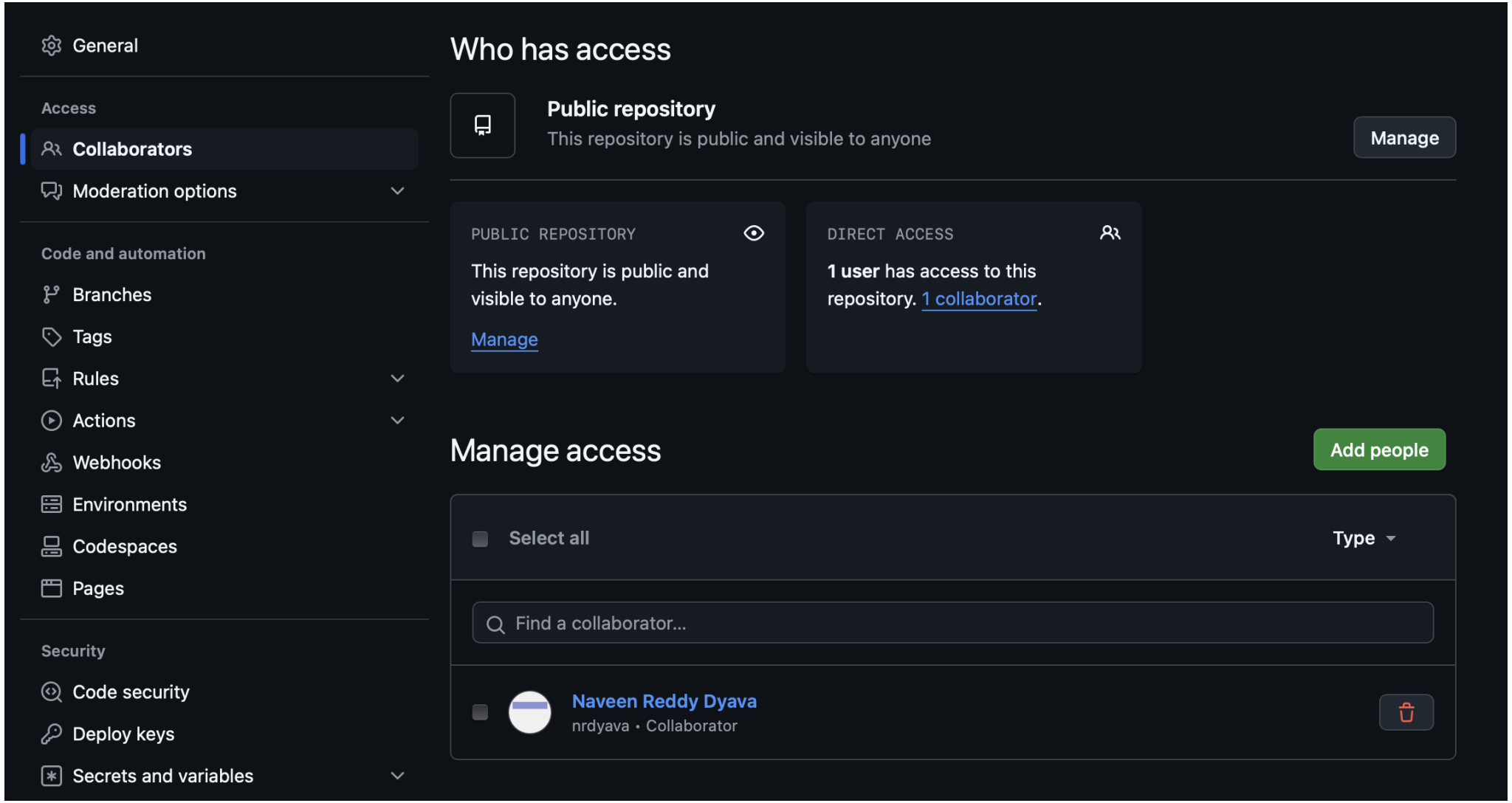


git + GitHub
for team collaboration

all the building blocks are now in place



now, enable collaborators in your GitHub repo



important to know what each role can do

- add **collaborators** to your repo
  - as a repo **owner** you have control over what gets changed
  - **collaborators** will be able to push to the repo

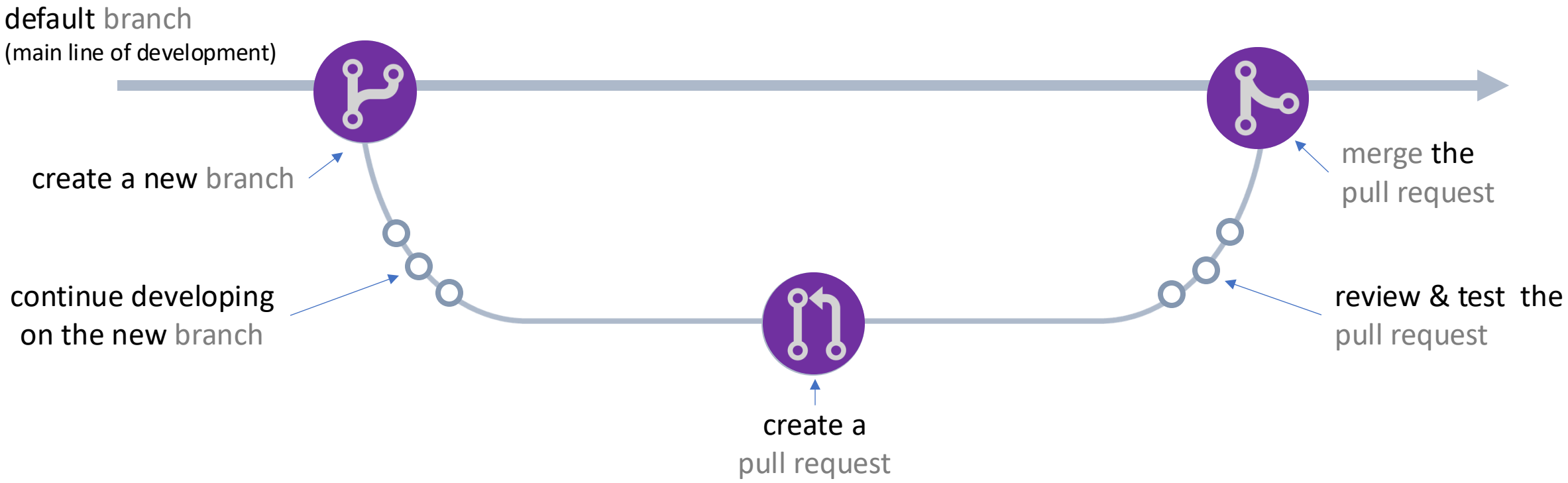
a) collaborators:

- work on a branch on the repo and create code
- send a pull request to add that code to the master repo

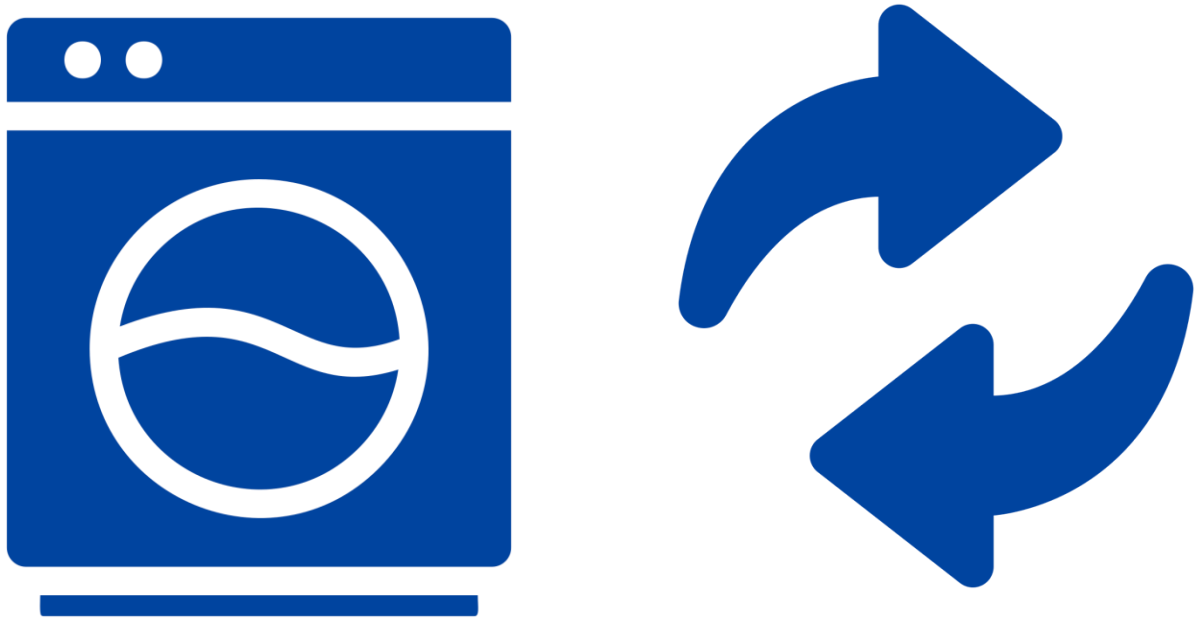
b) owner:

- comment on the pull request
- accept the pull request and/or merge the code

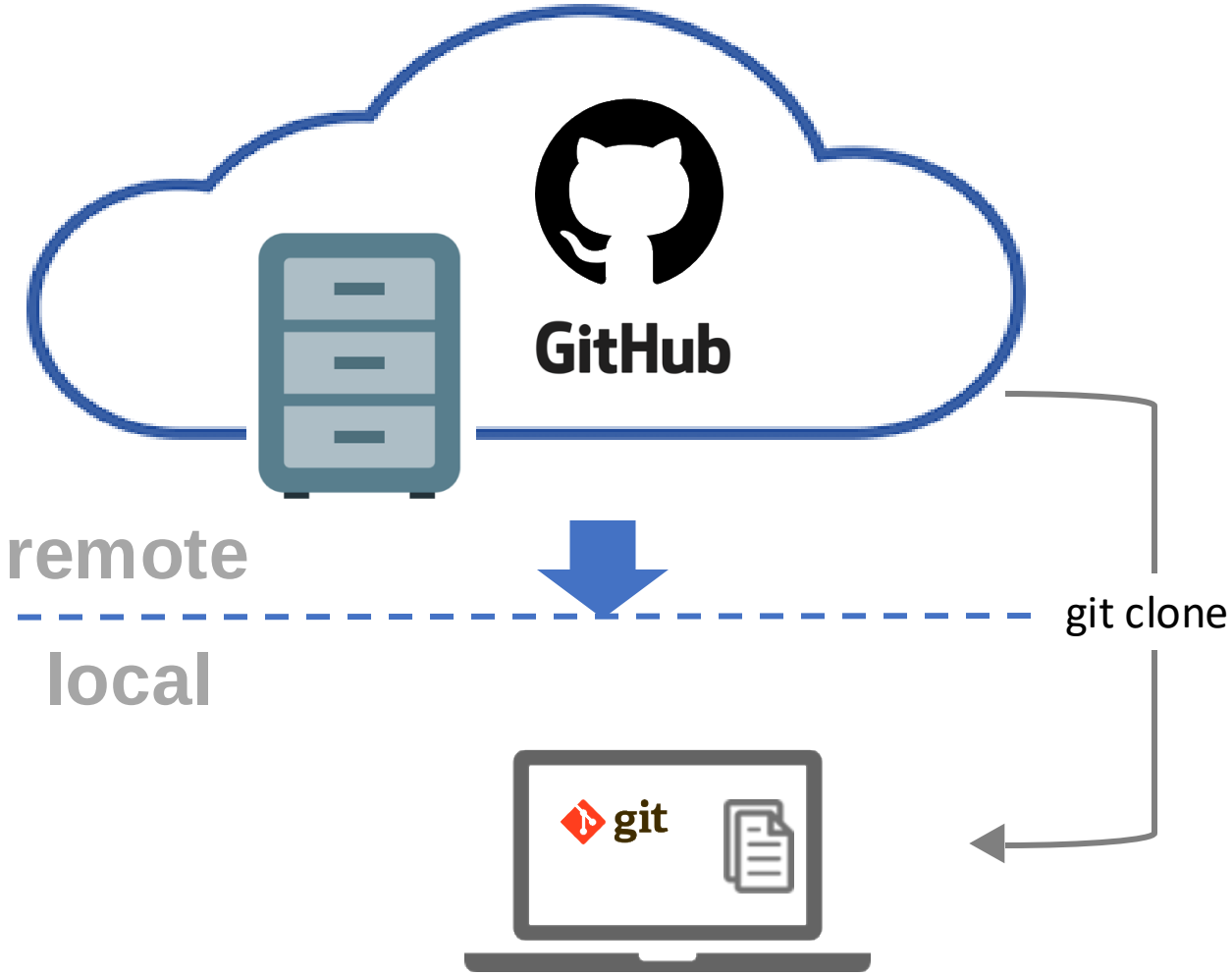
the magic of **branches** in git repos



rinse and repeat



final piece: getting an existing GitHub repo



a quick exercise

a quick exercise

1. go to a brand-new location

```
$ cd <your selected location>
```

2. clone somebody else's remote repo

```
$ git clone https://github.com/<your chosen repo>
```

3. go inside the repo you just cloned

```
$ cd <name of cloned repo>
```

4. (checkout and) create a branch

```
$ git checkout -b <mytestbranch-myname>
```

a quick exercise

5. make a change in your code file
6. go on, verify that git is tracking the change

```
$ git status
```

7. do your usual git routine

```
$ git add testfile.R
$ git commit -m "Add hubris to the code"
```

8. now, you'll create a pull request

```
$ git push origin <mytestbranch-myname>
```

9. time for the repo owner to intervene!

your friendly neighborhood Visual Studio Code

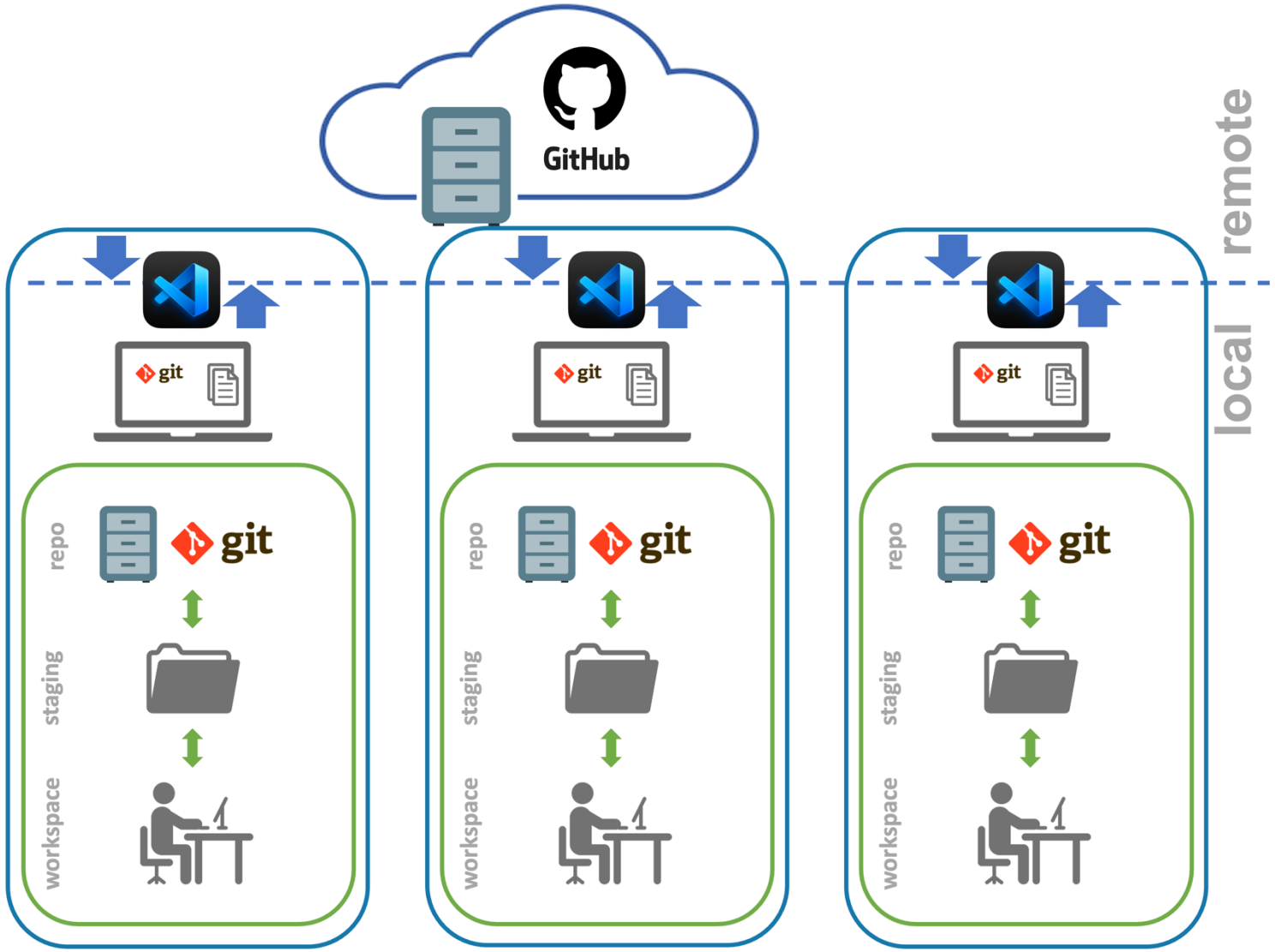
a simpler workflow is possible

- we've been working with git+GitHub from the **command line**
  - it can get confusing and harder that necessary at times

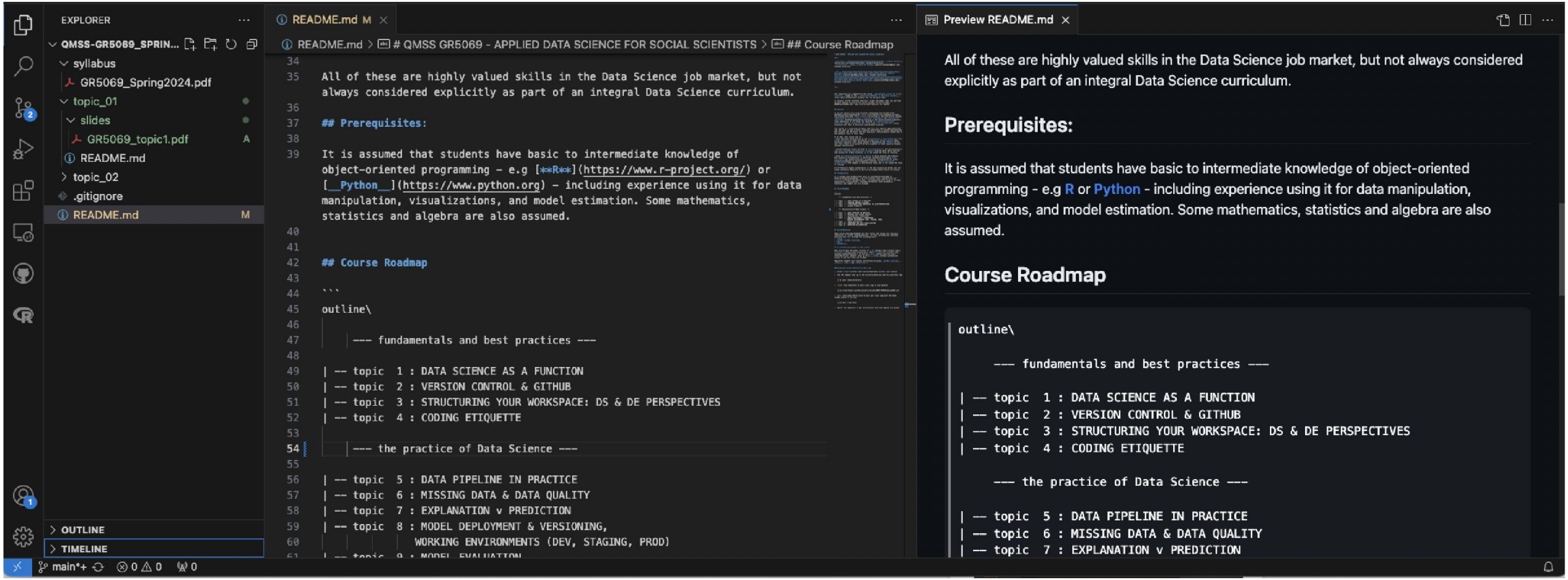
- now that you understand the flow, we can make our lives easier with an **IDE** (Integrated Development Environment)

- our preferred IDE today → **Visual Studio Code** aka **VS Code**
  - it can seamlessly handle git + GitHub interaction, and coding needs

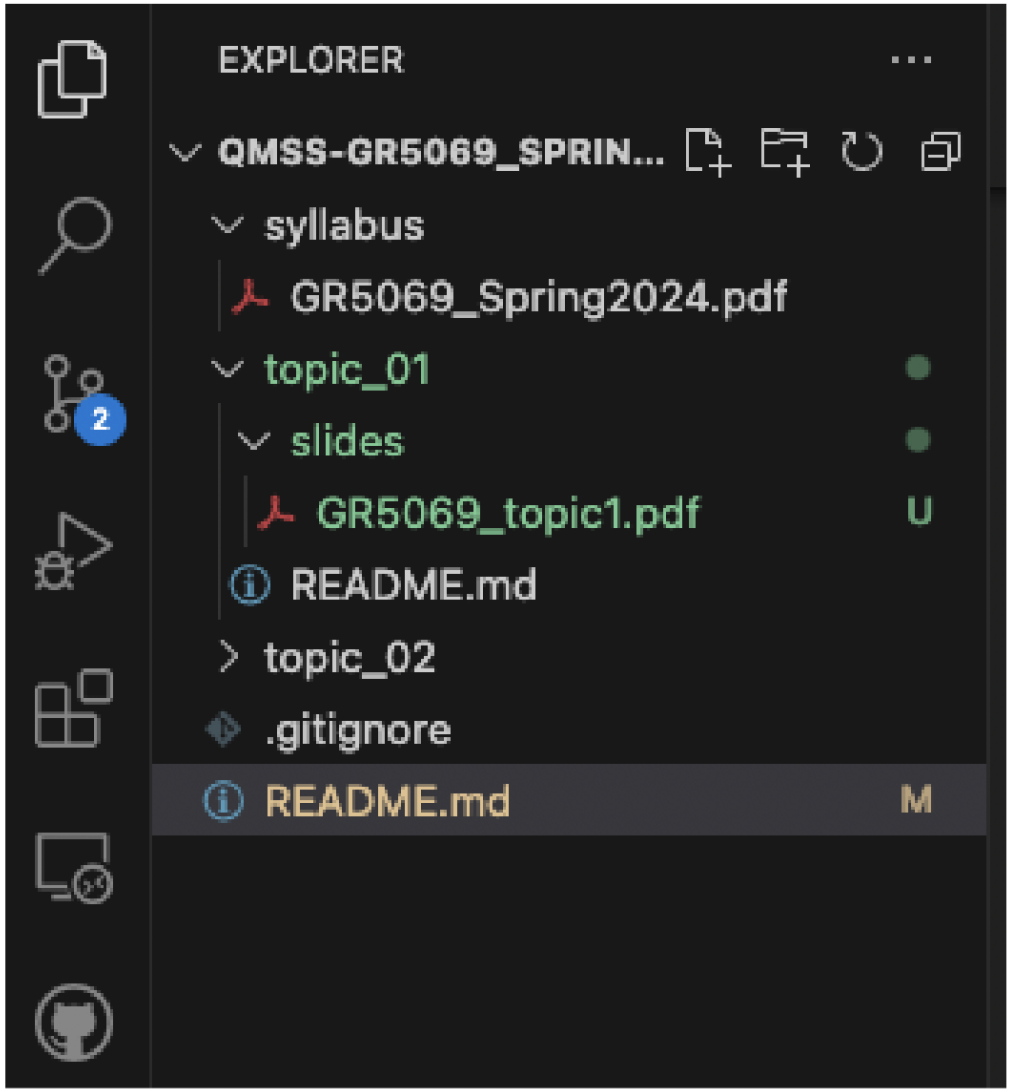
a git + GitHub workflow through an IDE



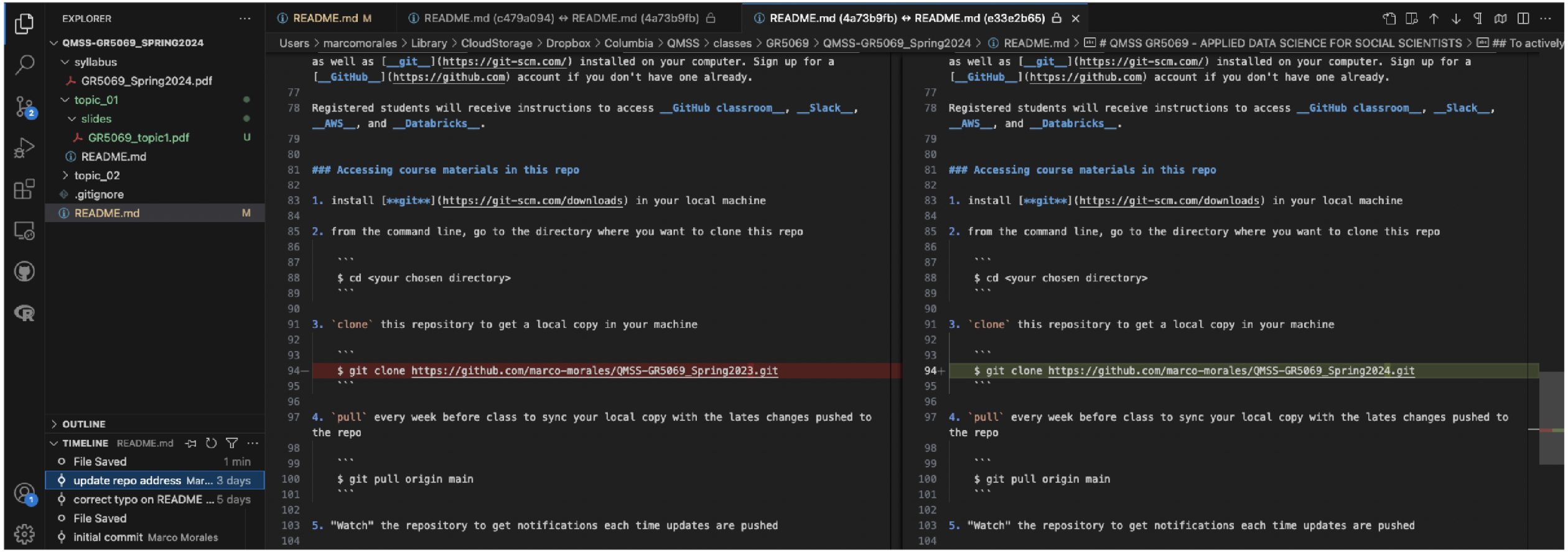
a git + GitHub workflow through an IDE



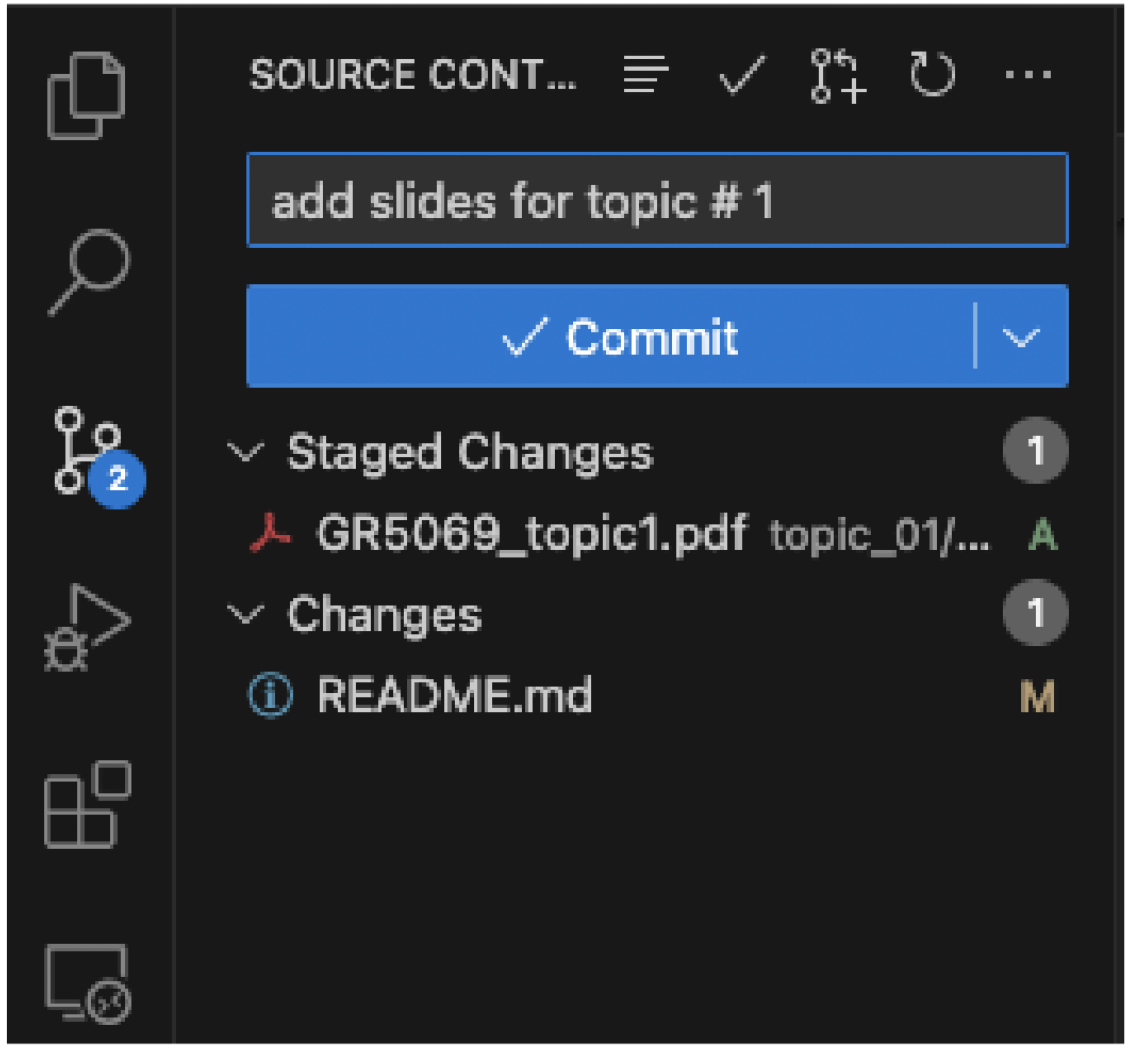
a simple way to git status



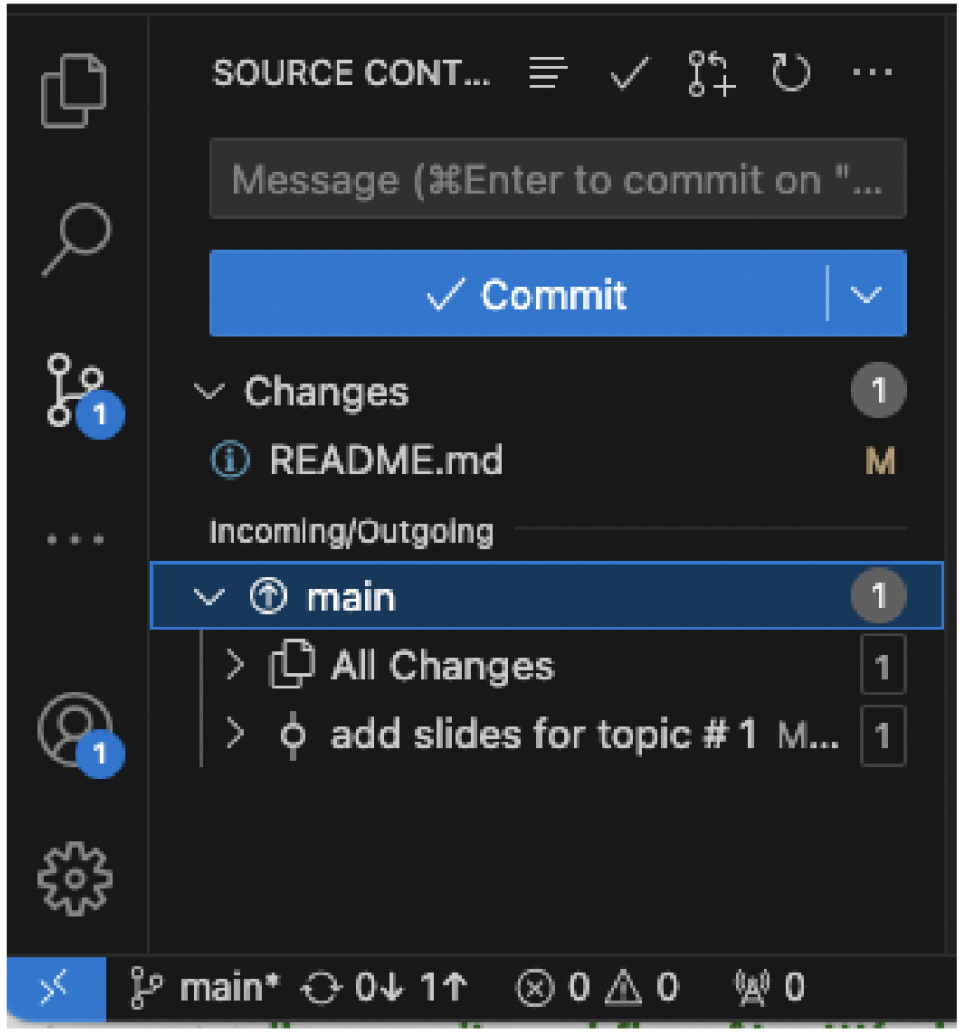
a simple way to git diff



all your git workflow in a single screen



all your git workflow in a single screen



though this be madness,
yet there’s method in’t

think of **version control** as a **time machine**…

- go **back in time** to when the digital solution worked well
- go **forward in time** to improvements in the digital solution
- go to **parallel universes** where other versions of the digital solution exist

the method to this version control madness

- git init - initializes git, and indicates that the folder should be tracked
- git add - brings new files to the attention of git to be tracked as well
- git commit - takes a snapshot of alerted files
- git push - sends changes committed in your branch (of your local repo) to the remote branch (of the GitHub repo)

the method to this version control madness

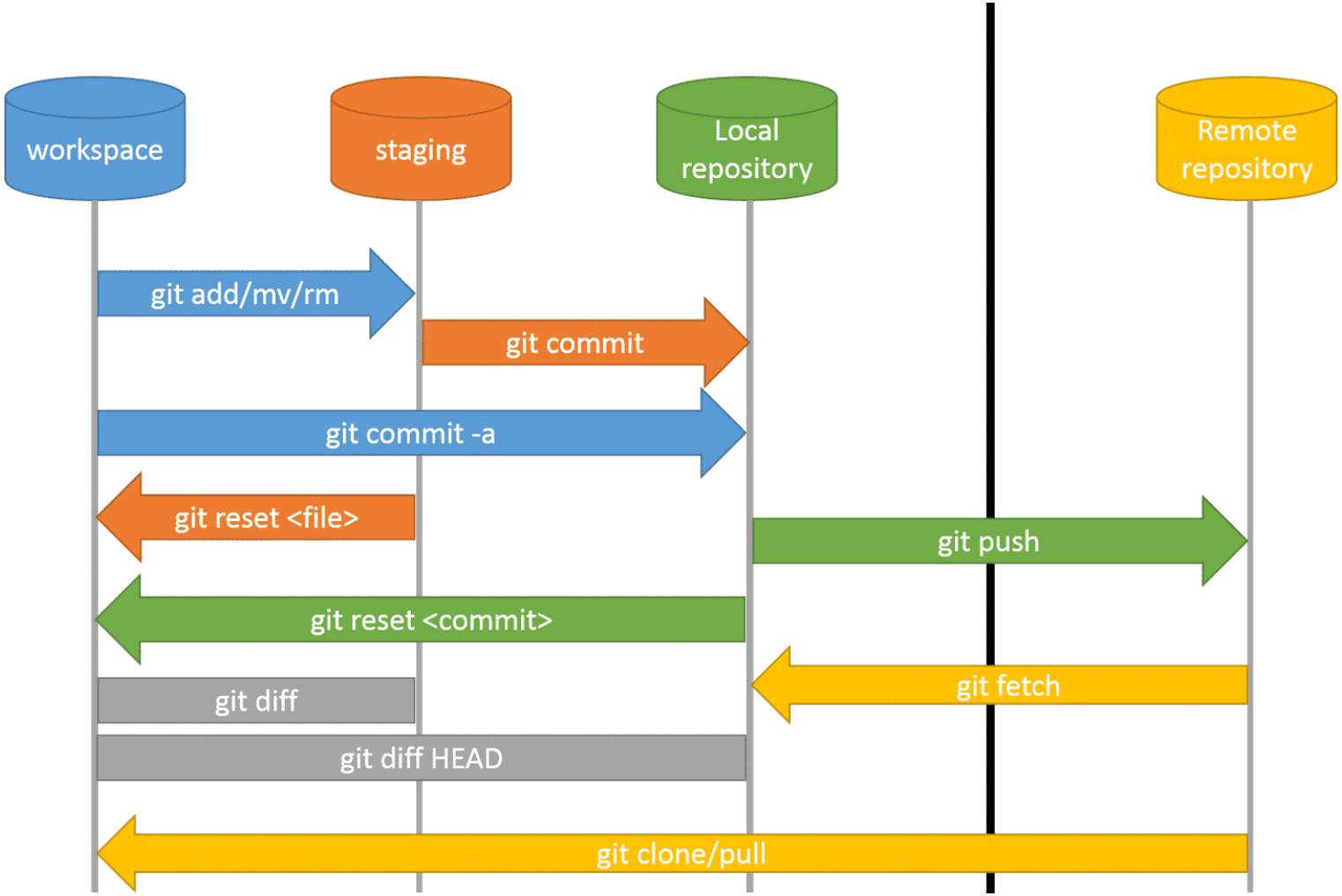


figure: Pro Git (2nd ed)

the method to this version control madness

- **clone**; a local copy of a repository that can be updated as changes happen
- **fork**; a thread in a repository
- **branch**; a local mirror copy of a repository at a given point in time
- **pull**; brings changes into master repository

version control : git + GitHub

Marco Morales
marco.morales@columbia.edu

Nana Yaw Essuman
nanayawce@gmail.com

GR5069: Applied Data Science
for Social Scientists

Spring 2025
Columbia University