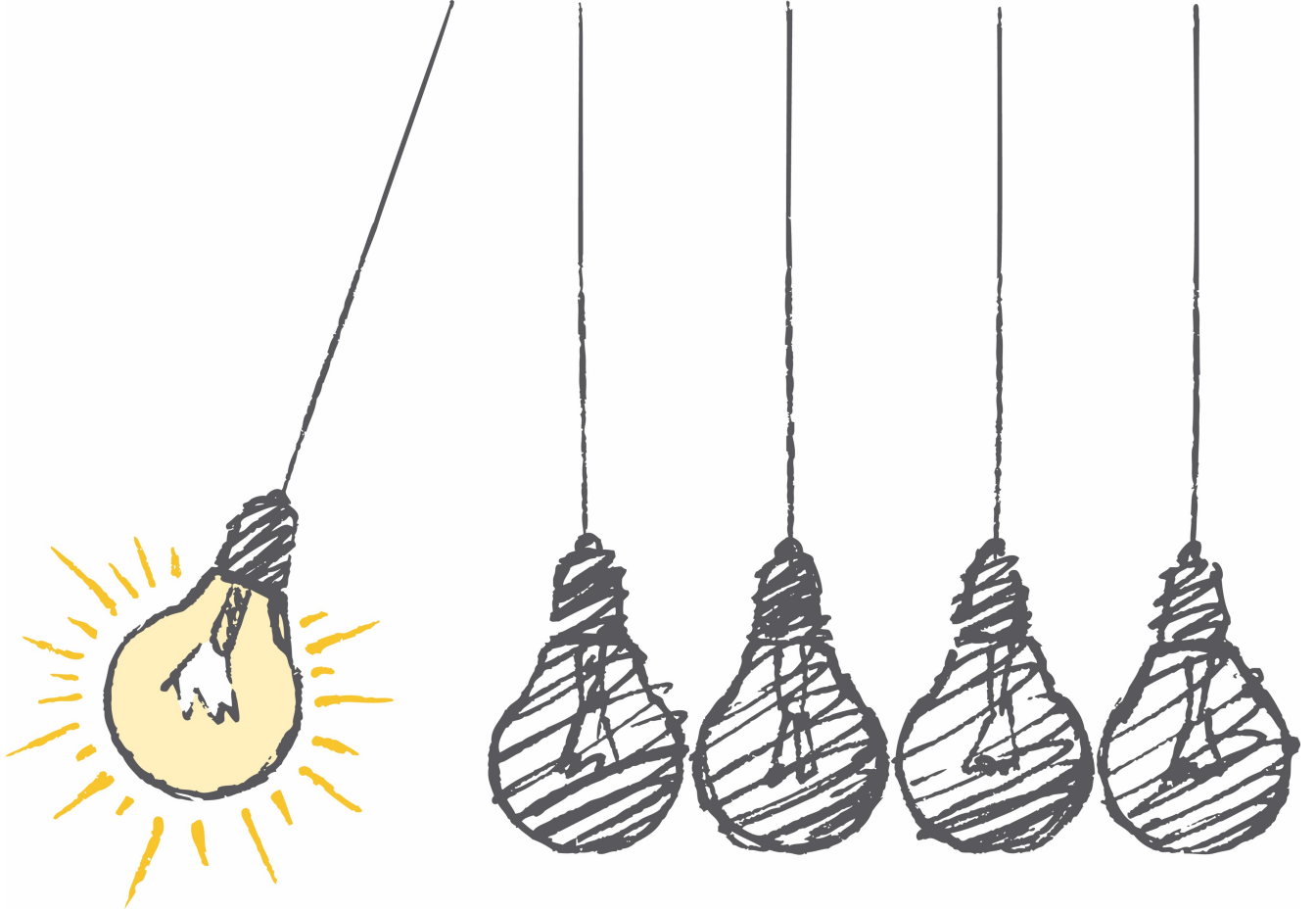
Making Analytics Reusable:

Using Git and GitHub

Prepared by **DIME Analytics**
dimeanalytics@worldbank.org

Presented by **Benjamin Daniels**
bdaniels@worldbank.org



WORLD BANK GROUP

UKaid
from the British people

Norad

What is Git?
What is GitHub?

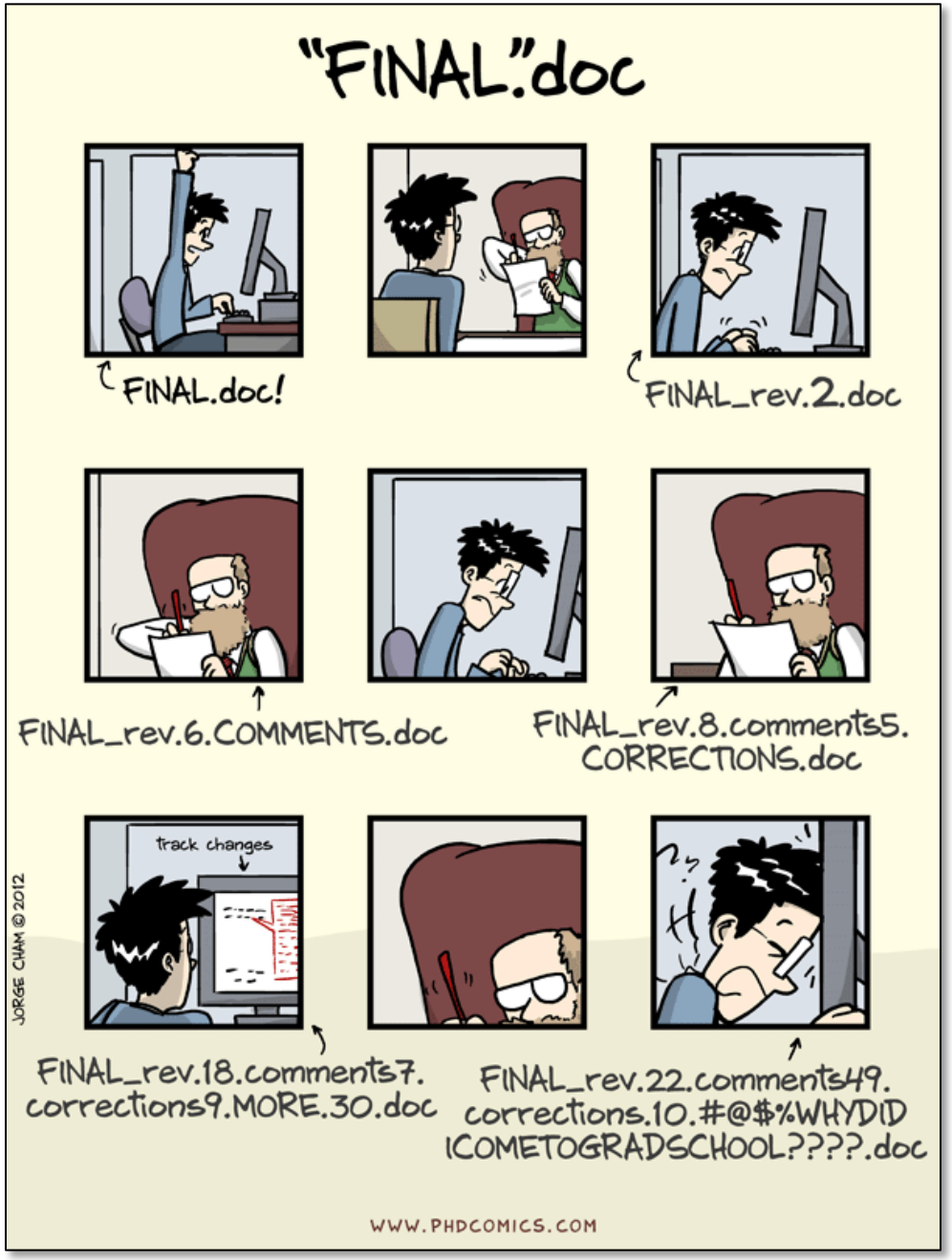
First, Git is not GitHub.

Second, GitHub is not Git.

Git is a **version control software**. It organizes *versions* of every file in a project in an orderly way.

GitHub is a "**development platform**". It manages *contributions* to projects in an orderly way.

Together, Git and GitHub allow you to avoid the "FINAL".doc problem.

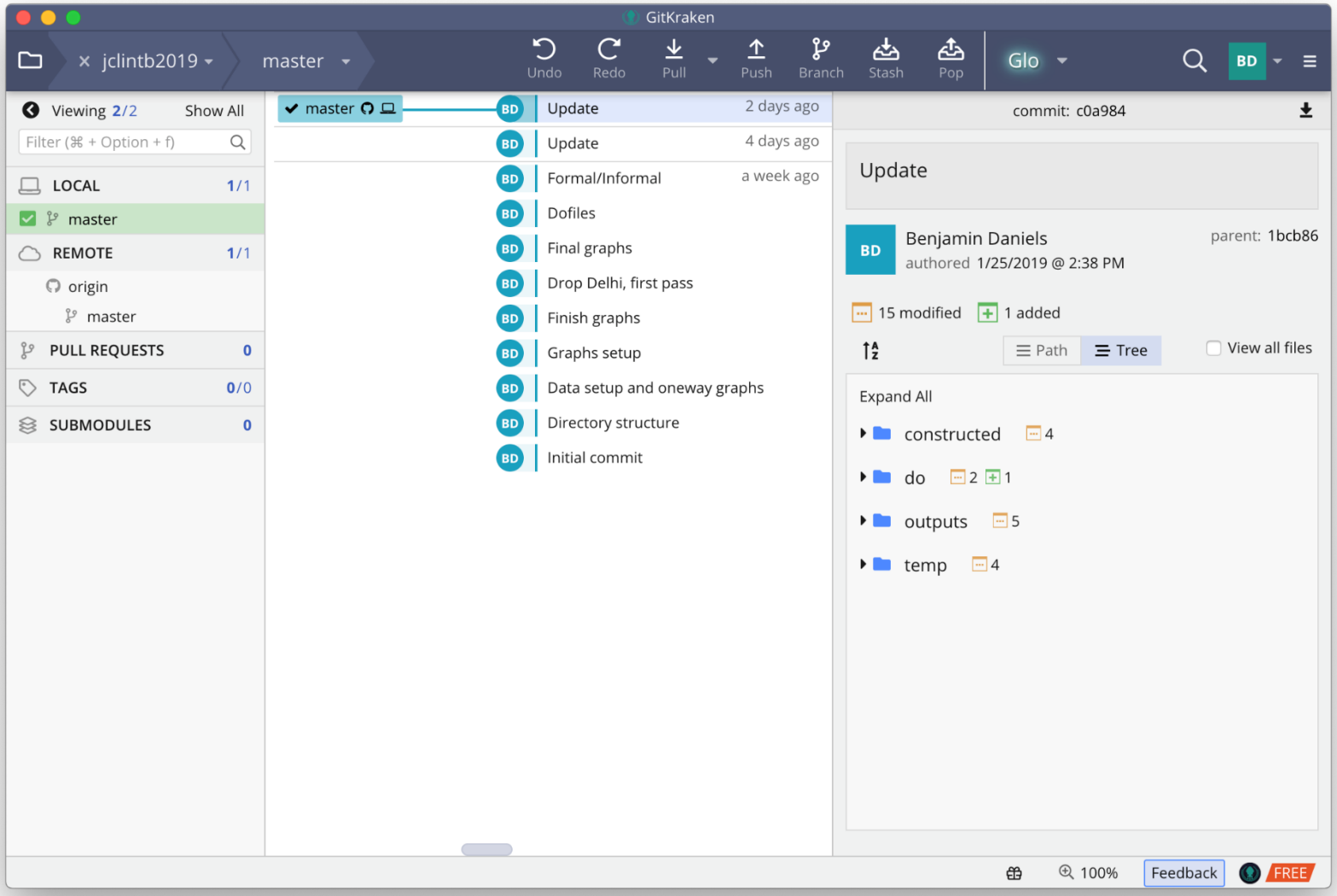


Git:
The basics

Git is a software that runs on your computer. It stores “version histories” of your files in a way that is really useful, but that you mainly don’t need to understand.

The complete history of a project is called a “repository”. A repository can be viewed in a desktop “client”, such as GitKraken, shown here.

A very simple repository would look much like this one: it has a series of versions called *commits* that make up its history. Each commit is a record of the changes between itself and the previous commit.

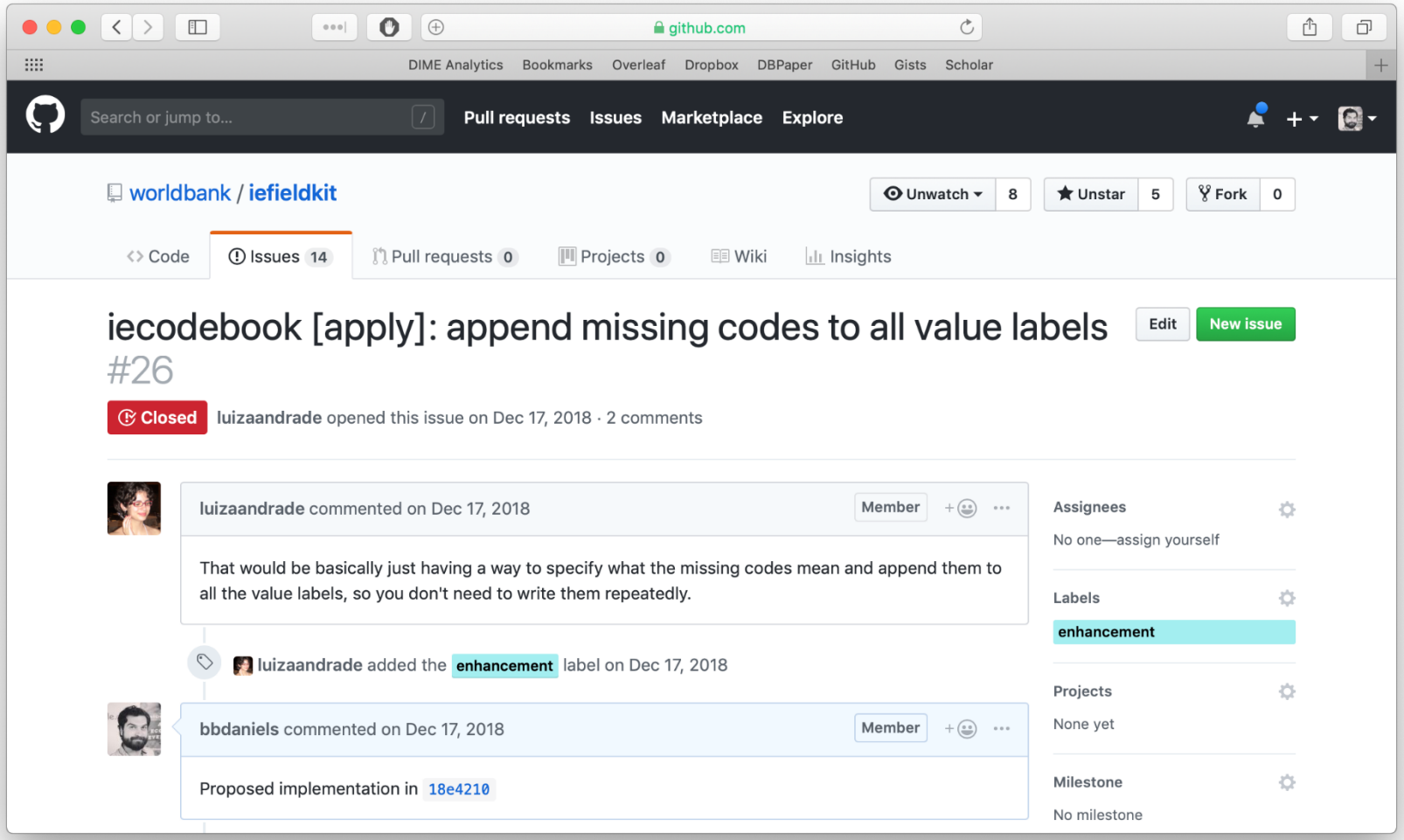


GitHub:
The basics

GitHub is a website that you can use to manage collaboration with other people.

This has two main forms: first, managing and tracking tasks (known generally as “issues”), and second, allowing people to submit new materials to the repository.

A very simple issue looks like this one: A problem is posed, and after some discussion a solution is submitted as a “pull request” to the repository.



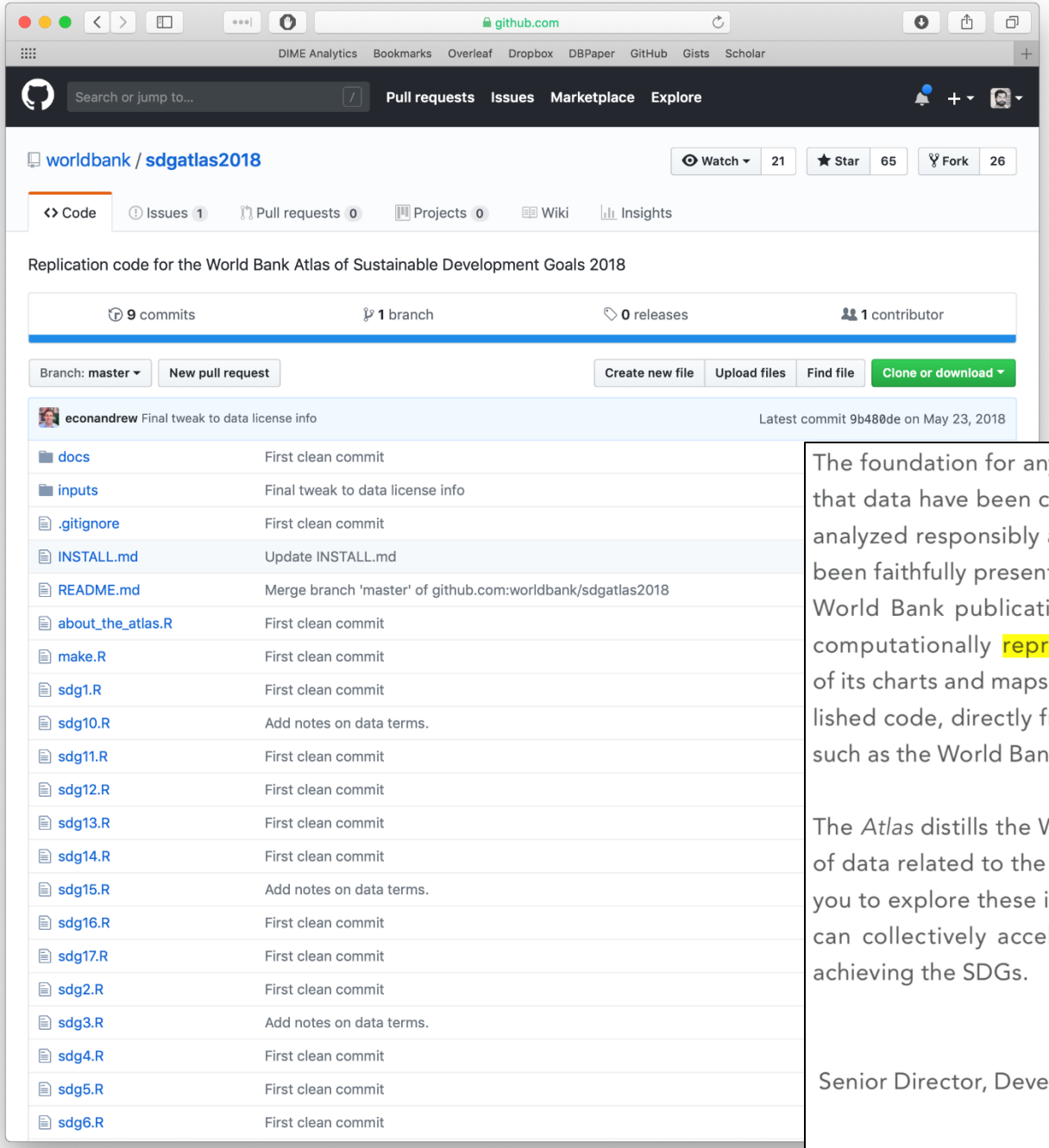
Git & GitHub: The basics

Mastering Git with GitHub gives you an awesome tool for managing your own team's workflow, as well as for making and releasing big projects.

Having both your data, code, and *code history* available for all authors and the public makes an incredibly useful tool as a product of your work.

Plus, Git protects all your files much better than other software.

(Shown here: The World Bank Atlas of Sustainable Development Goals.)



The foundation for any evidence is trust: trust that data have been collected, managed, and analyzed responsibly and trust that they have been faithfully presented. The *Atlas* is the first World Bank publication that sets out to be computationally reproducible—the majority of its charts and maps are produced with published code, directly from public data sources such as the World Bank's Open Data platform.

The *Atlas* distills the World Bank's knowledge of data related to the SDGs. I hope it inspires you to explore these issues further so that we can collectively accelerate progress toward achieving the SDGs.

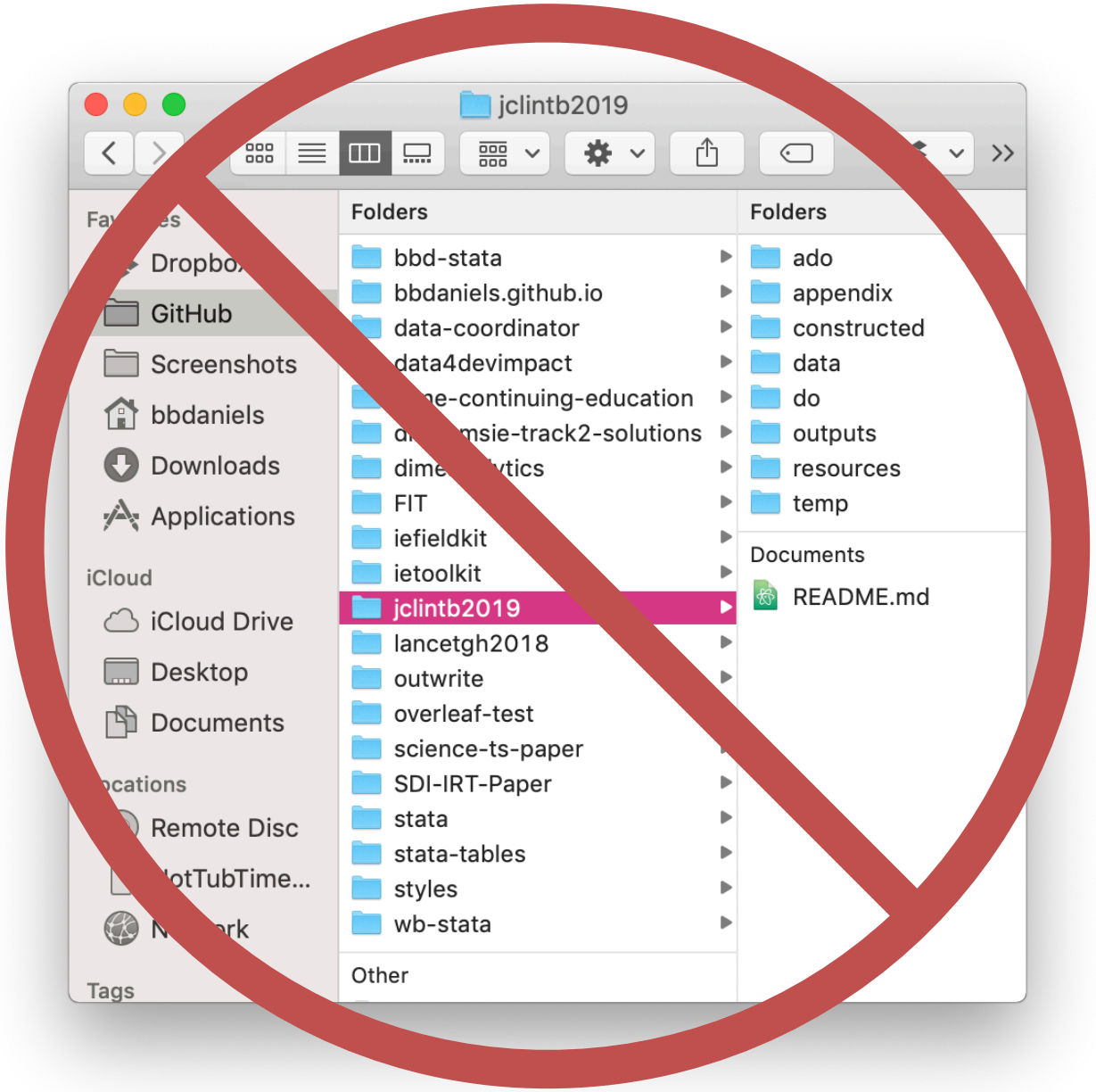
Shanta Devarajan
Senior Director, Development Economics and
Acting Chief Economist
World Bank Group

The files are not the repository

Your Git repositories will probably mainly be used to manage a lot of files. But it is important to remember that although the terms are often used semi-interchangeably, the “repository” is *not* the files themselves.

The files live in what we will call the “working directory” – the project folder (“directory”) where you are working.

A Git repository manages the contents of the working directory.



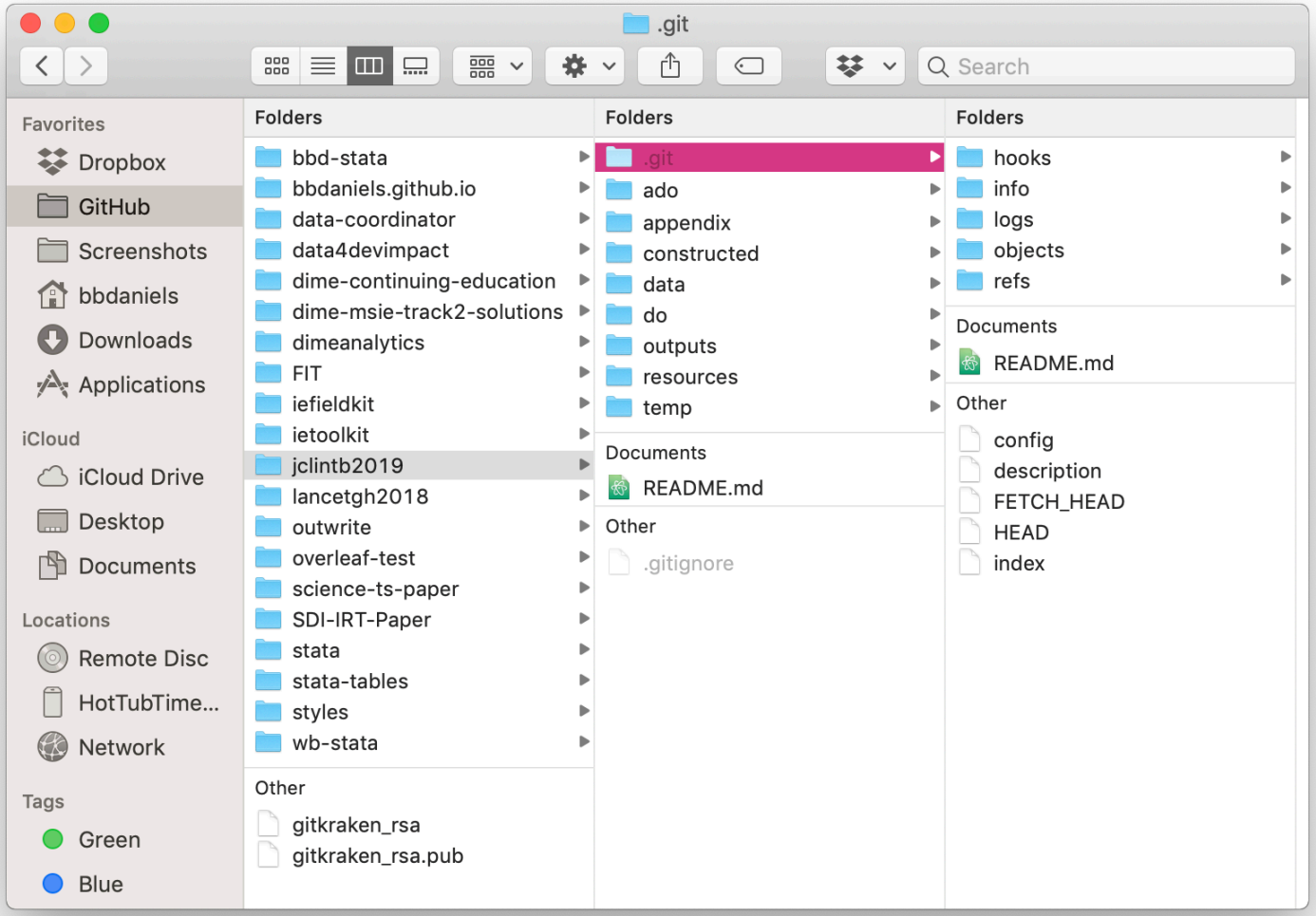
What is the repository then?

You *really* don't need to remember this, but:

The Git repository is a bunch of hidden files *inside* the working directory. This is where the magic happens.

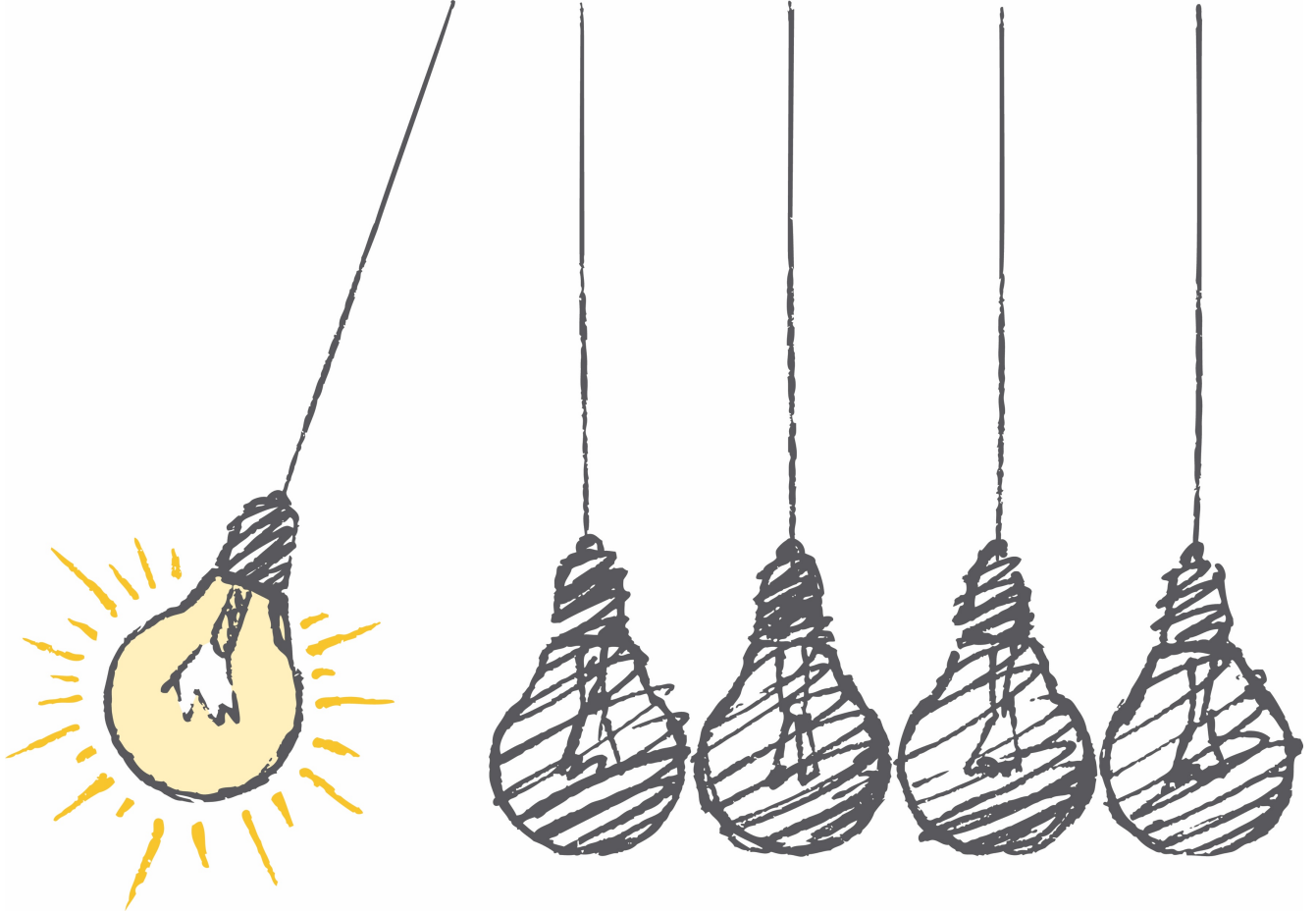
It's hidden because you never need to touch these. NEVER.

Just pretend this doesn't exist – everything important happens entirely inside your Git client.



So how does it *work*?

Part 1: Review commits and branches



WORLD BANK GROUP

UKaid
from the British people

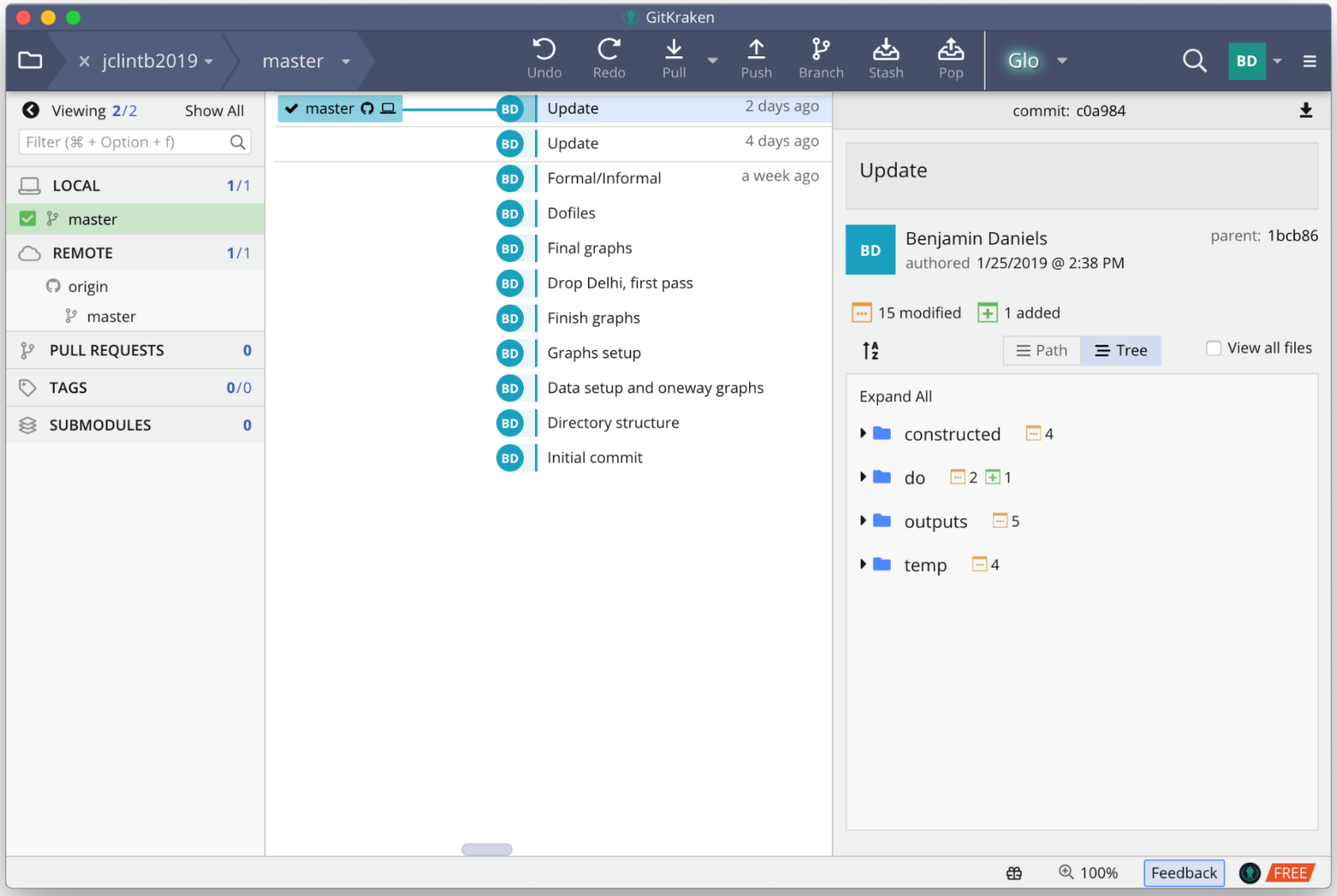
Norad

Git creates **commits**

The most important thing Git does is create “commits”.

A commit is a snapshot of the entire working directory. Each commit has a *name* and a *timestamp*.

You have to tell Git when to create new commits. This is probably the biggest difference from whatever you do now.



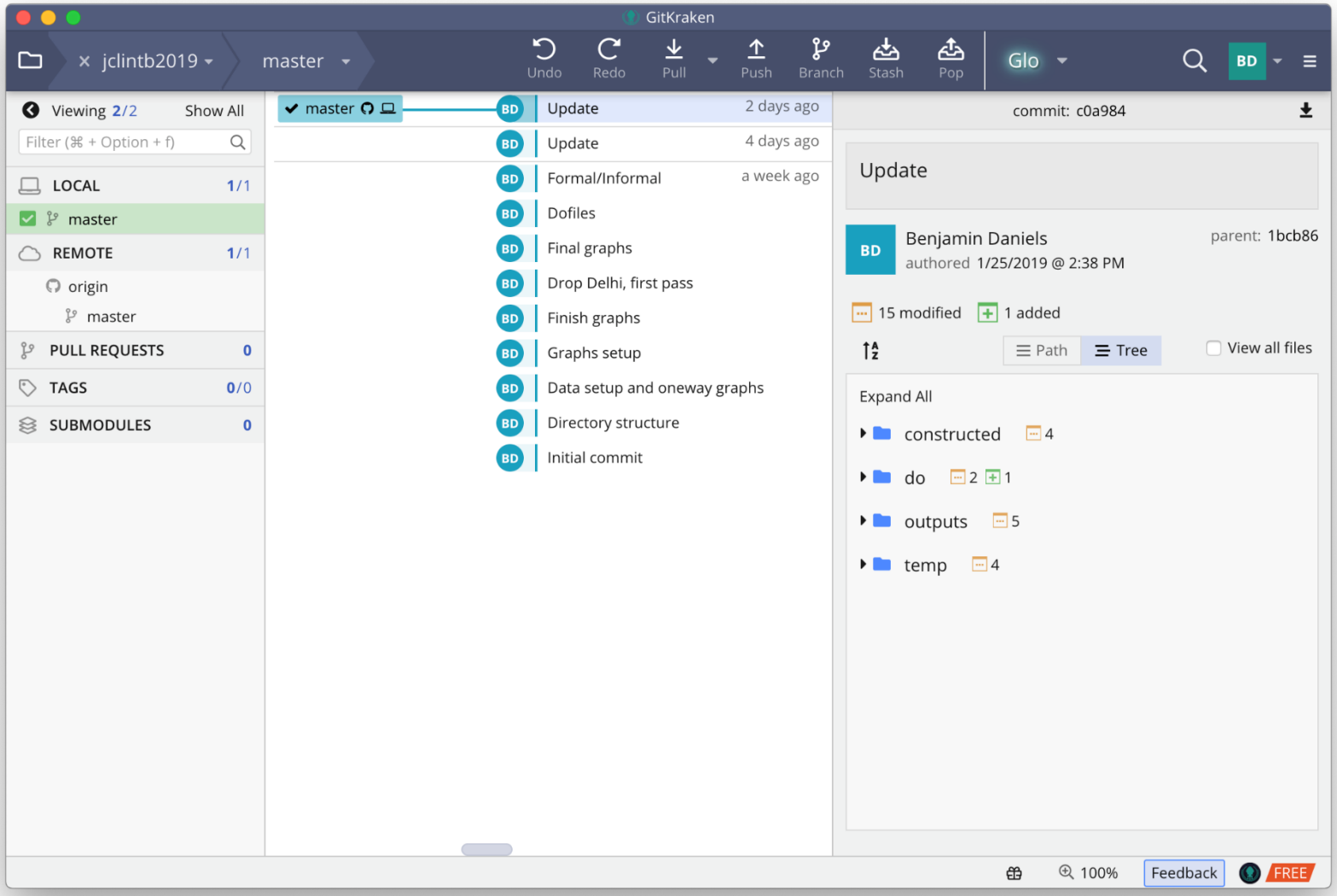
Commits name past versions

Git *watches* your saved changes

But it only *stores* them when you tell it to: this is a “commit”

You can name these so that they make sense and are easy to find when you need them in the future (you *always* need them when you least expect, right?)

Required Reading: Git vs Dropbox
https://michaelstepner.com/blog/git-vs-dropbox

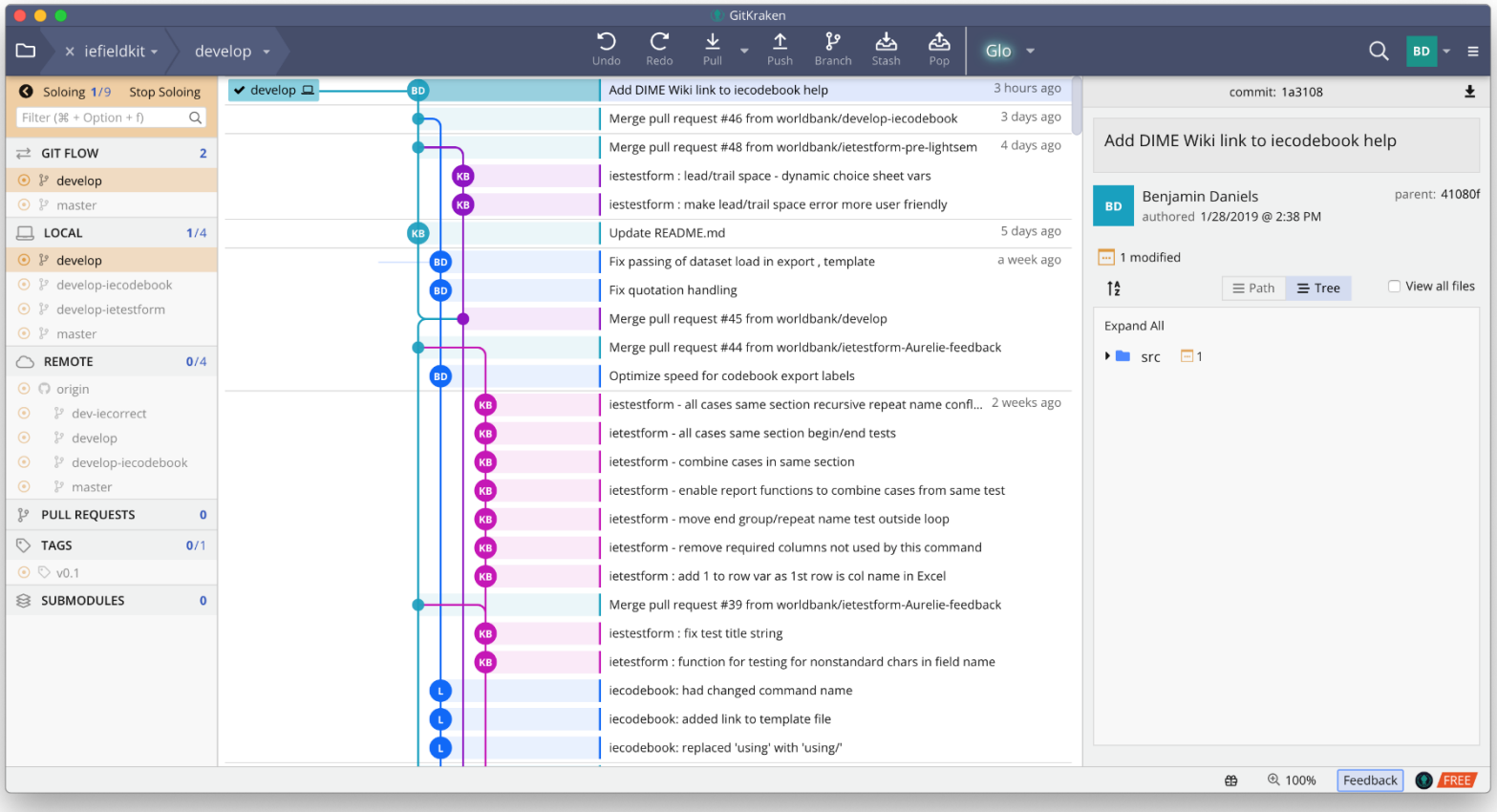


Git creates **branches**

Branches are the “killer feature” of Git. Nearly every advanced Git usage you learn will be about how to manage branches.

Branches enable different people to work on the same thing at the same time, and they enable you to view different versions of your files.

Branching allows you to move forwards or backwards in time; and to move “horizontally” in time through various concurrent versions.

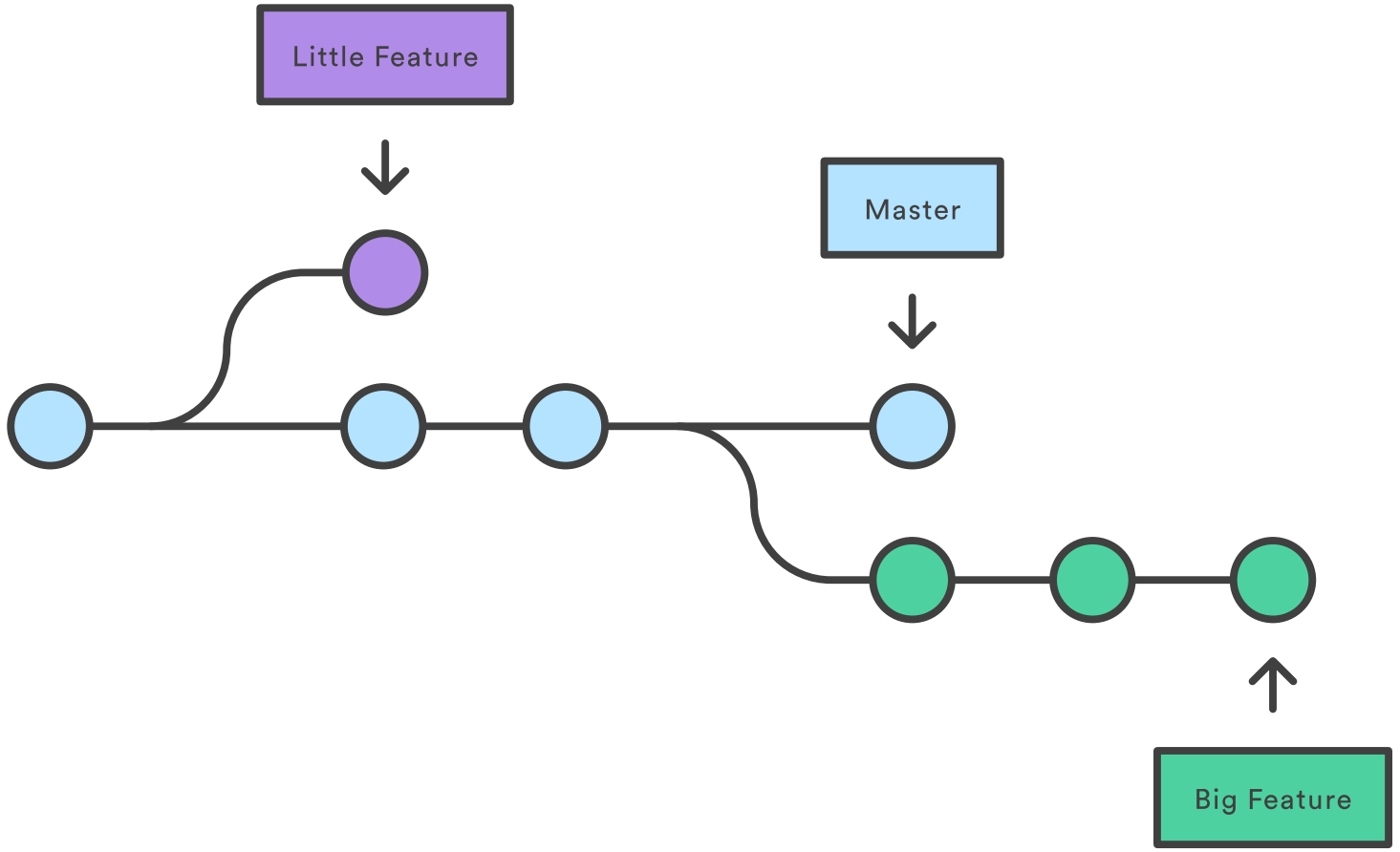


Branches name current versions

Each “branch” points to a commit, usually the latest version in some development workflow. You can switch between branches *locally*. When you do, your working directory will look exactly as it did when that commit was made.

Every contributor can be on any branch they like at any time – past or current.

This means various experimental changes can be made simultaneously without affecting the current edition of a product.



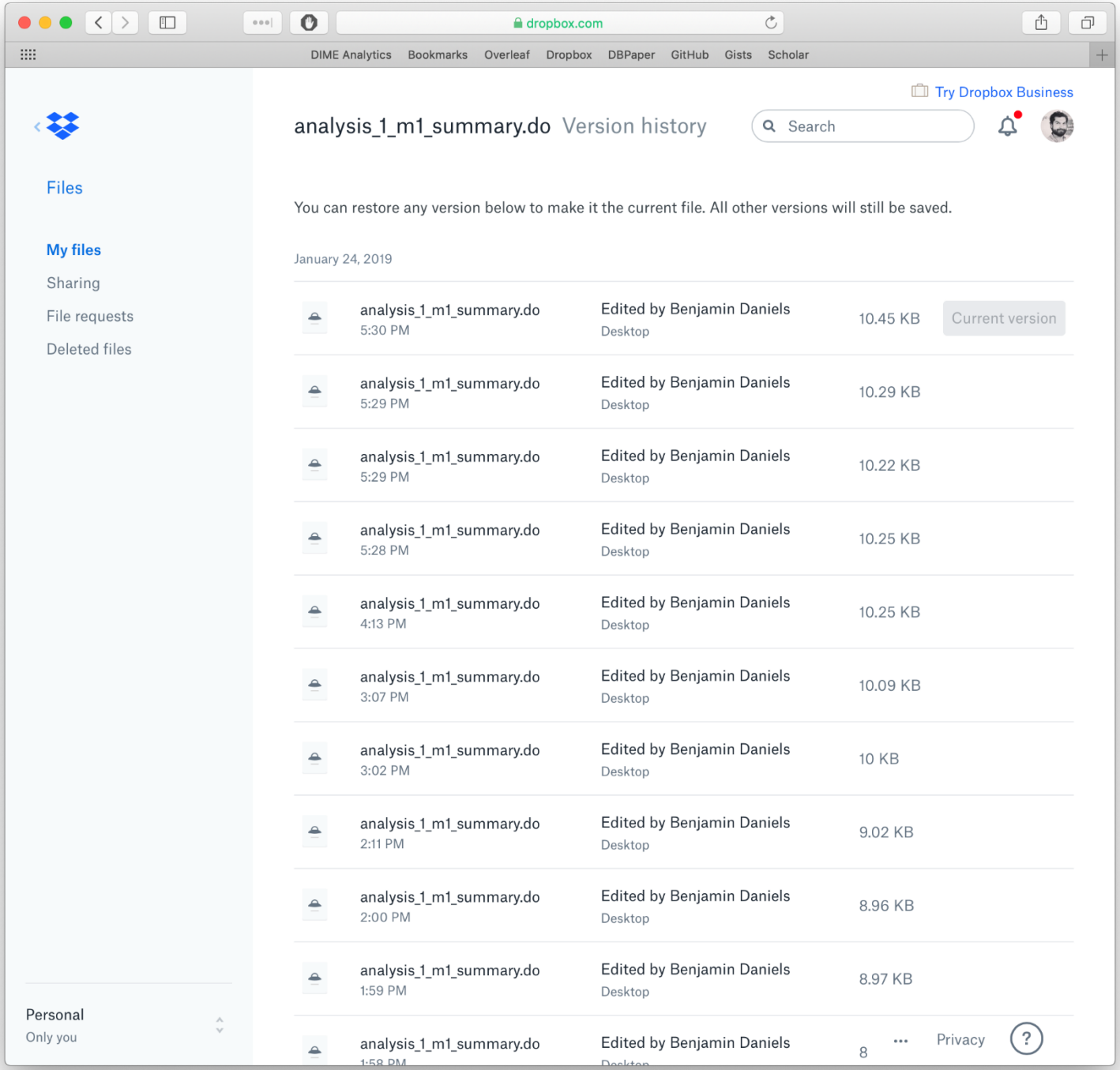
How is that better than Dropbox?

Dropbox stores a new snapshot *every time you save* each file.

You save your work *often* (right?)

So Dropbox stores *many* similar images of your files

But you can't tell which is which!

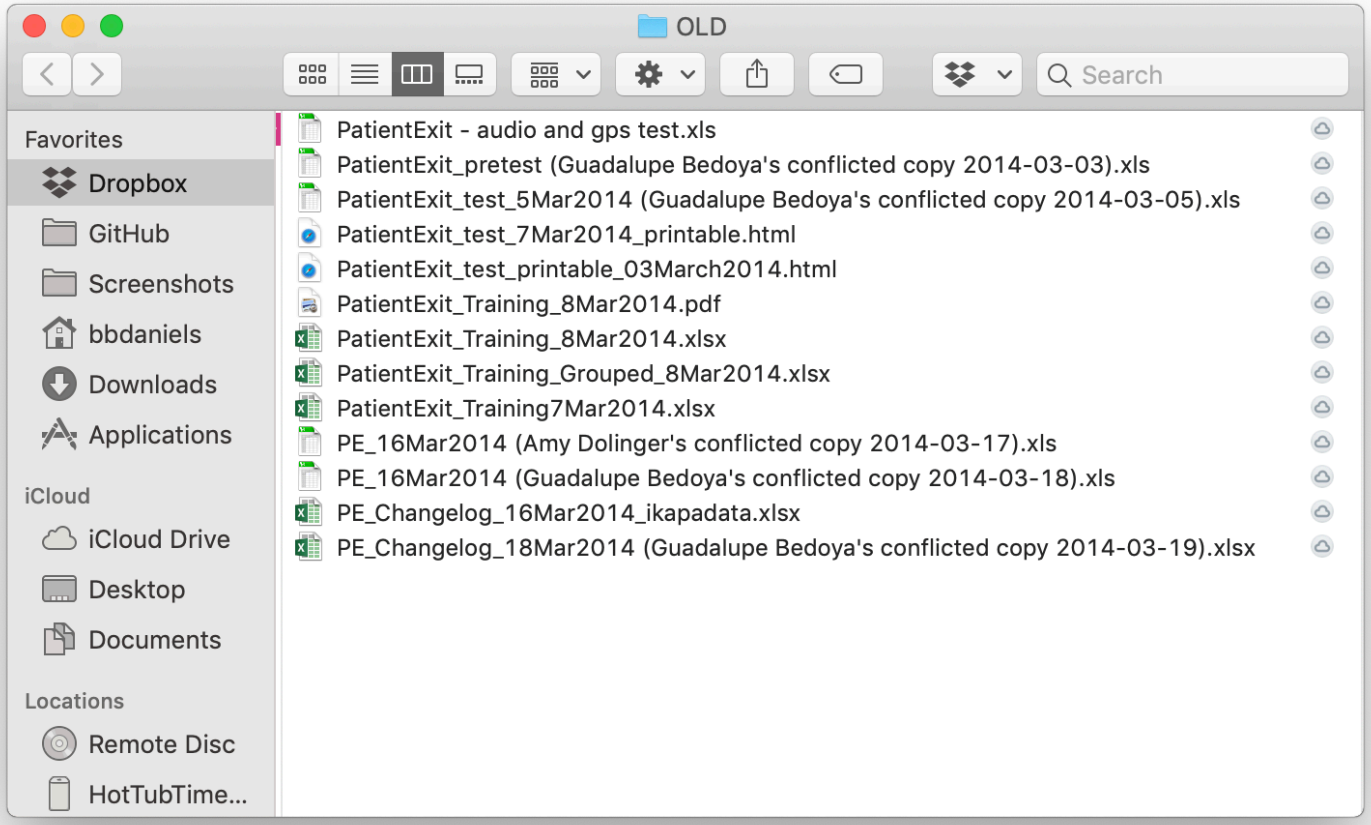


How is that better than Dropbox?

In Dropbox, there can be only *one* living version of a file at any time – otherwise a “conflicted copy” is created. This means nobody can edit the same file at the same time: Dropbox has no concurrency.

Worse, if you “roll back” a file to a previous version to see what it looked like, this affects everyone’s *current* version, even if you don’t want it to.

Finally, Dropbox versions are costly (computationally) – so they delete them often without telling you.



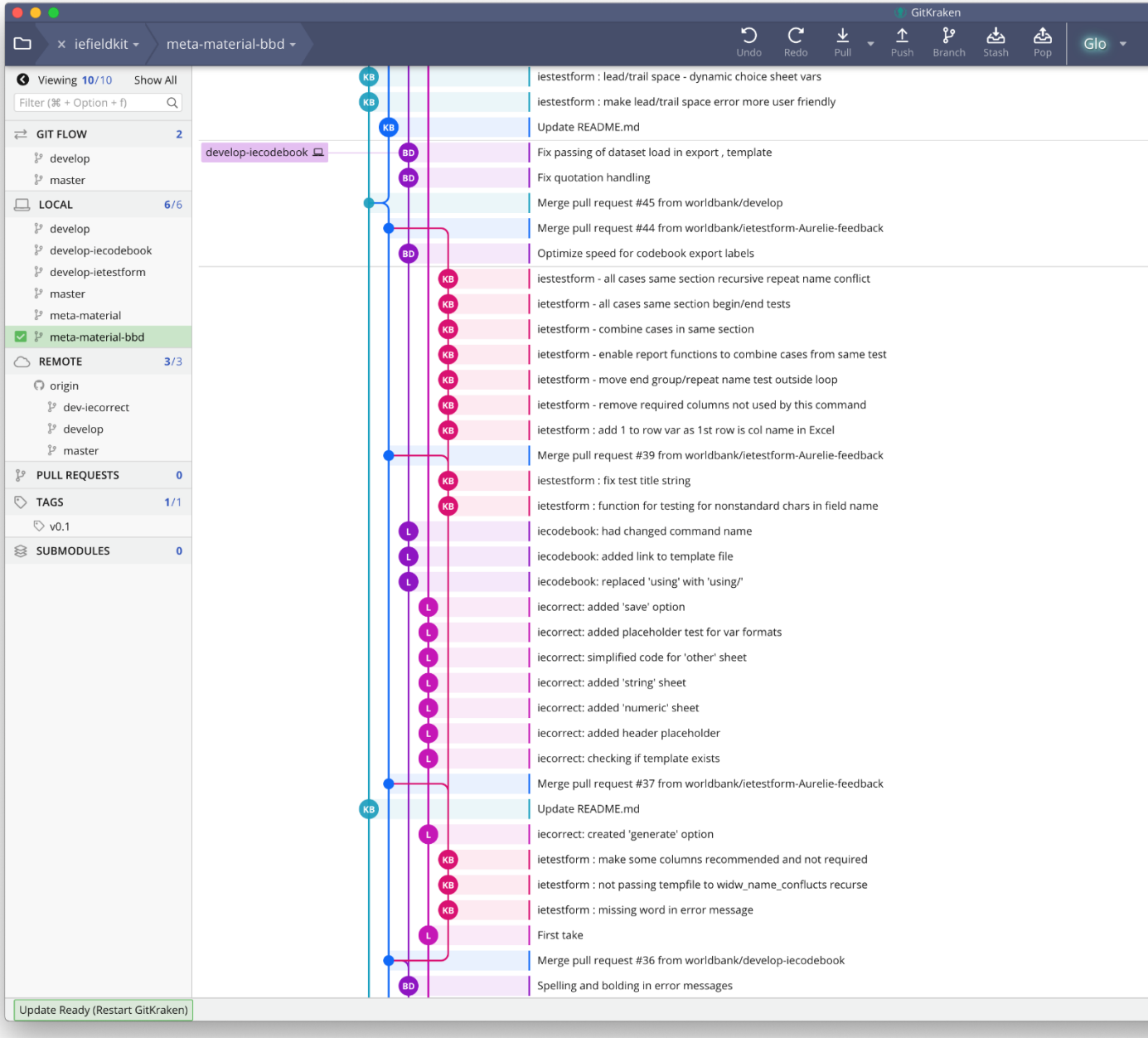
This can get complicated.

To make sure we know what we are doing, we need to *organize* our commits and branches.

This is non-trivial: it was more than 5 years between the release of Git and the development of a “successful” branching model.

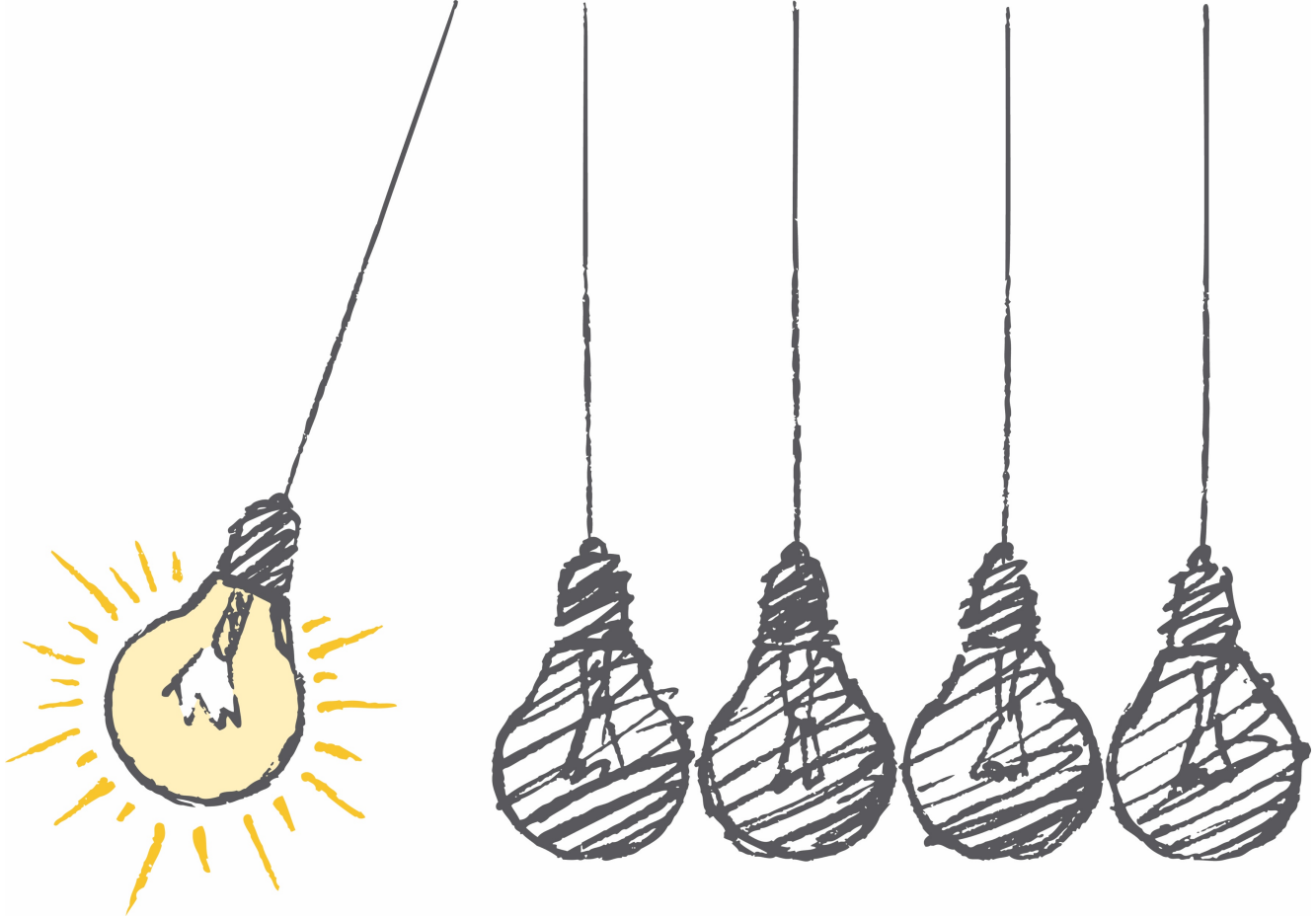
You have to think in ways you are not used to thinking when you start using Git, so these workflows will not feel intuitive at all.

But they are worth learning, since Git will not be as useful otherwise.



So how does it *work*?

Part 2: Organize your commits and branches



WORLD BANK GROUP

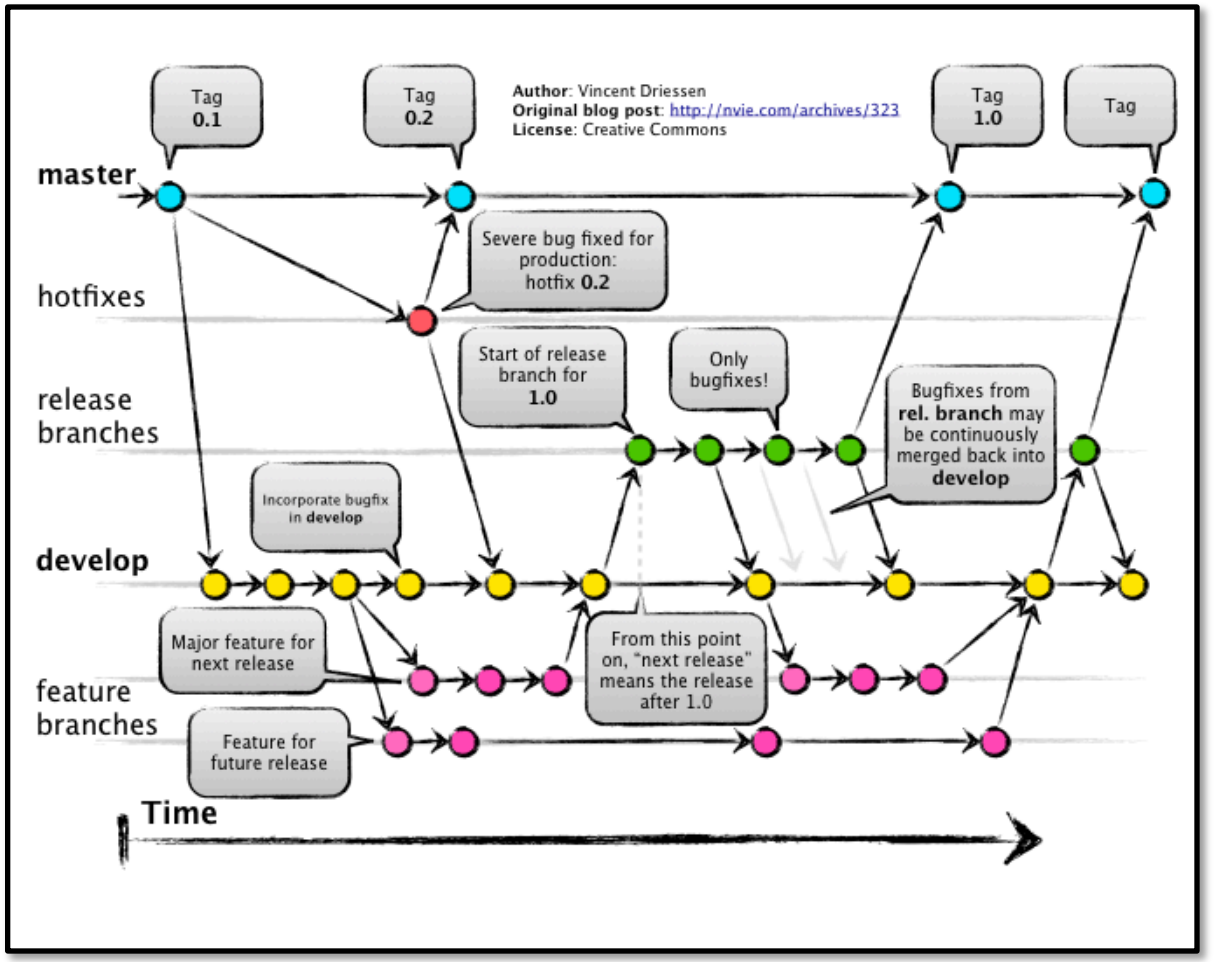
UKaid
from the British people

Norad

Git Flow organizes your commits and branches

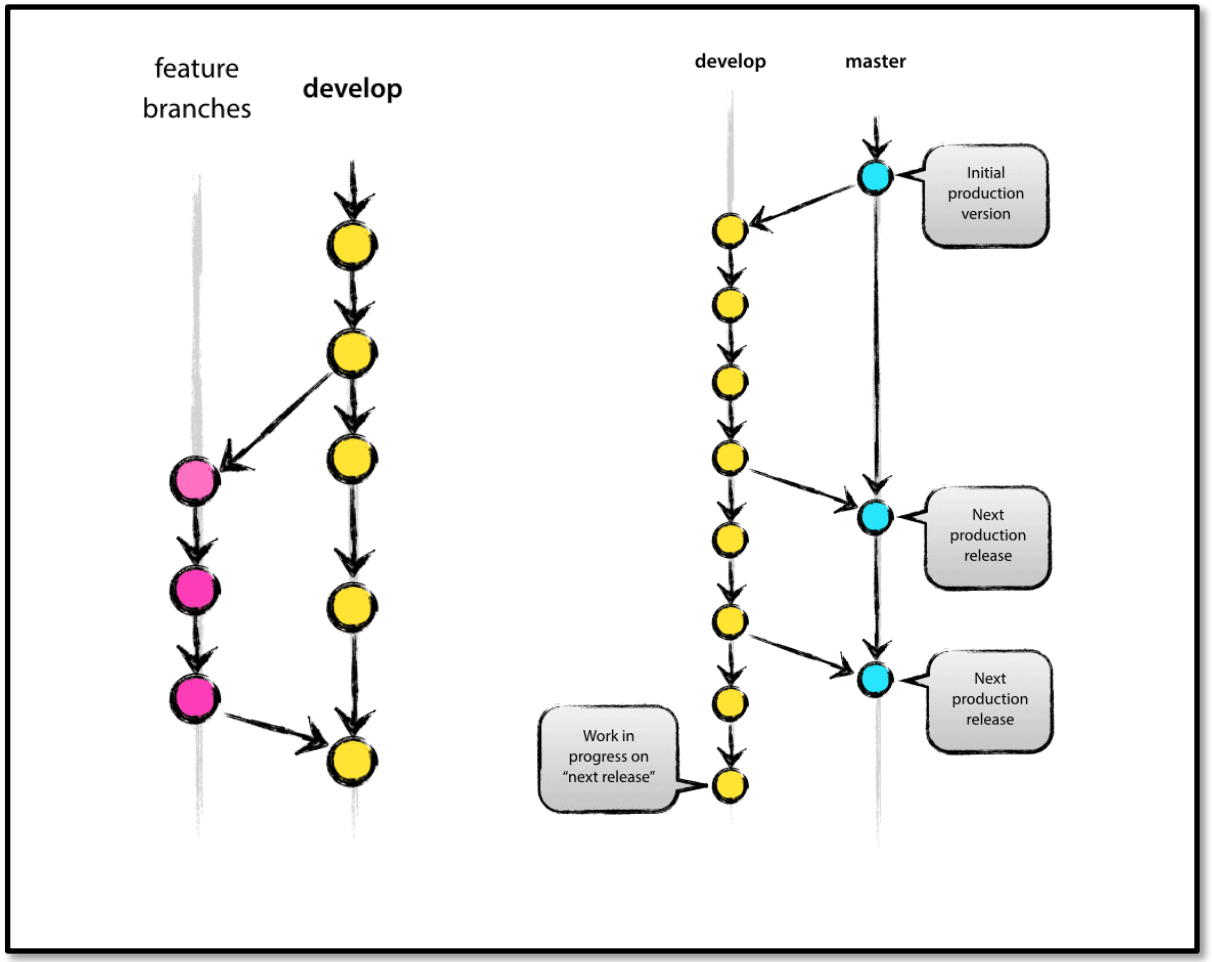
- Git Flow is a similar idea to a standardized folder structure
- When you move to another project using Git Flow, you will know what everything is for and how you should be working

- Required Reading: Git Flow https://nvie.com/posts/a-successful-git-branching-model/



Core structure of Git Flow workflow

1. [master] branch is protected, always ready to run, and has sparse commits from [develop]
2. [develop] branch is protected, “nearly” ready to run, and has frequent commits from [feature] branches
3. [feature] branches are frequent, specific, personal, and hold all new work



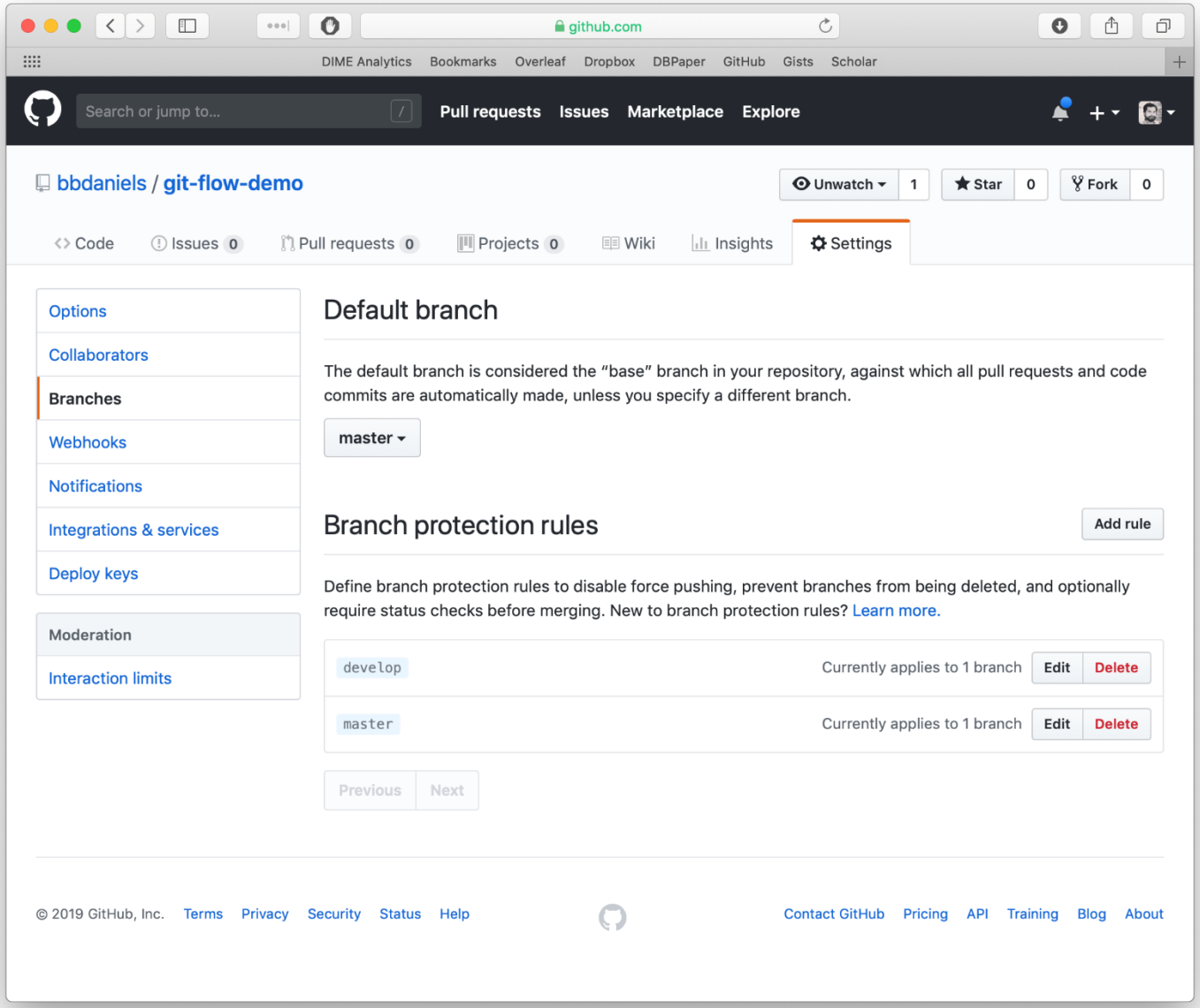
Two “protected” branches

These have rules set on GitHub which prevent:

- “Force pushing” (this can be used to overwrite history and is not allowed)
- Merging without approval

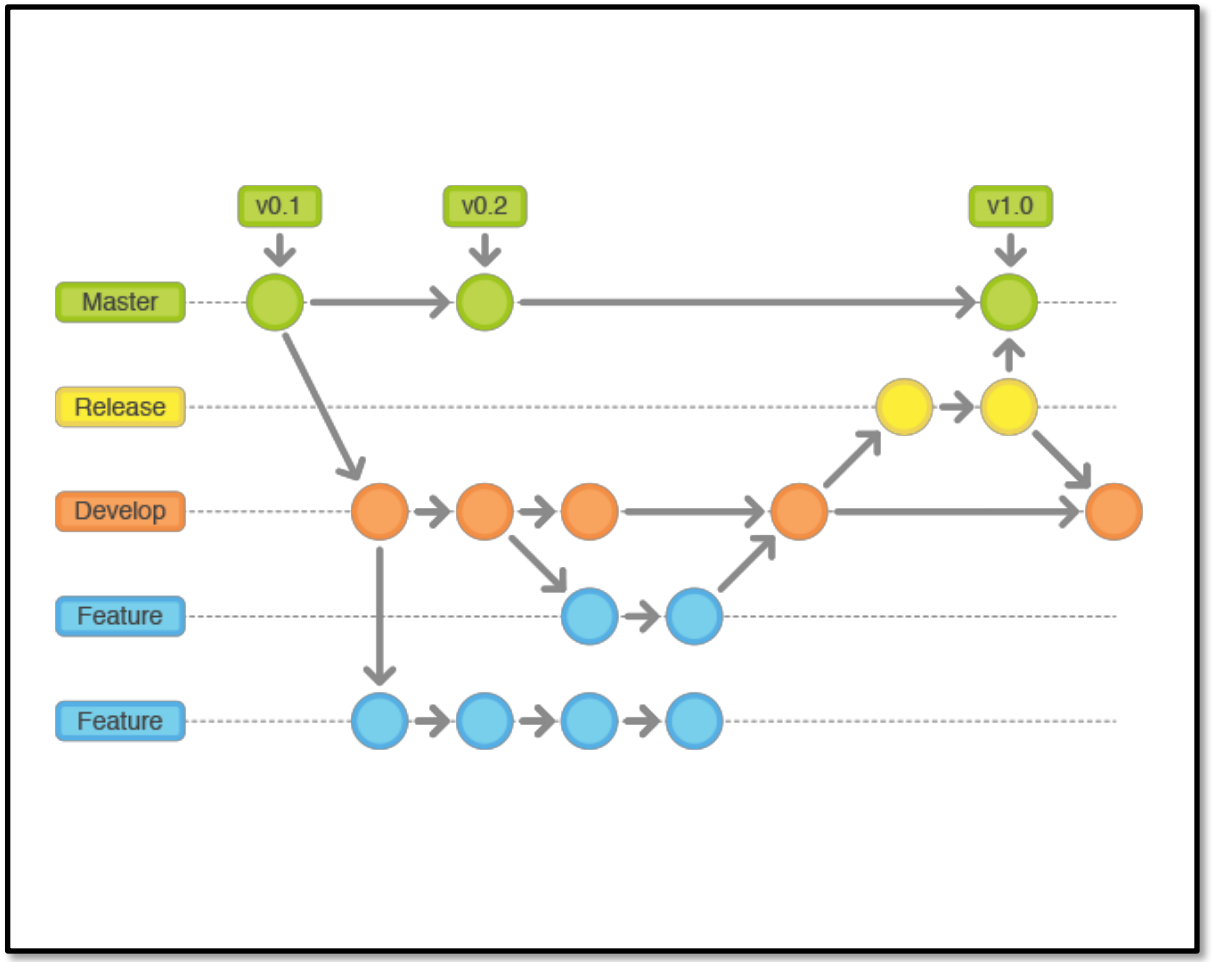
With these protections in place, the [master] branch is a preserved common status record that nobody edits directly.

[develop] is the common workspace where ready changes are merged after being built in a [feature] branch.



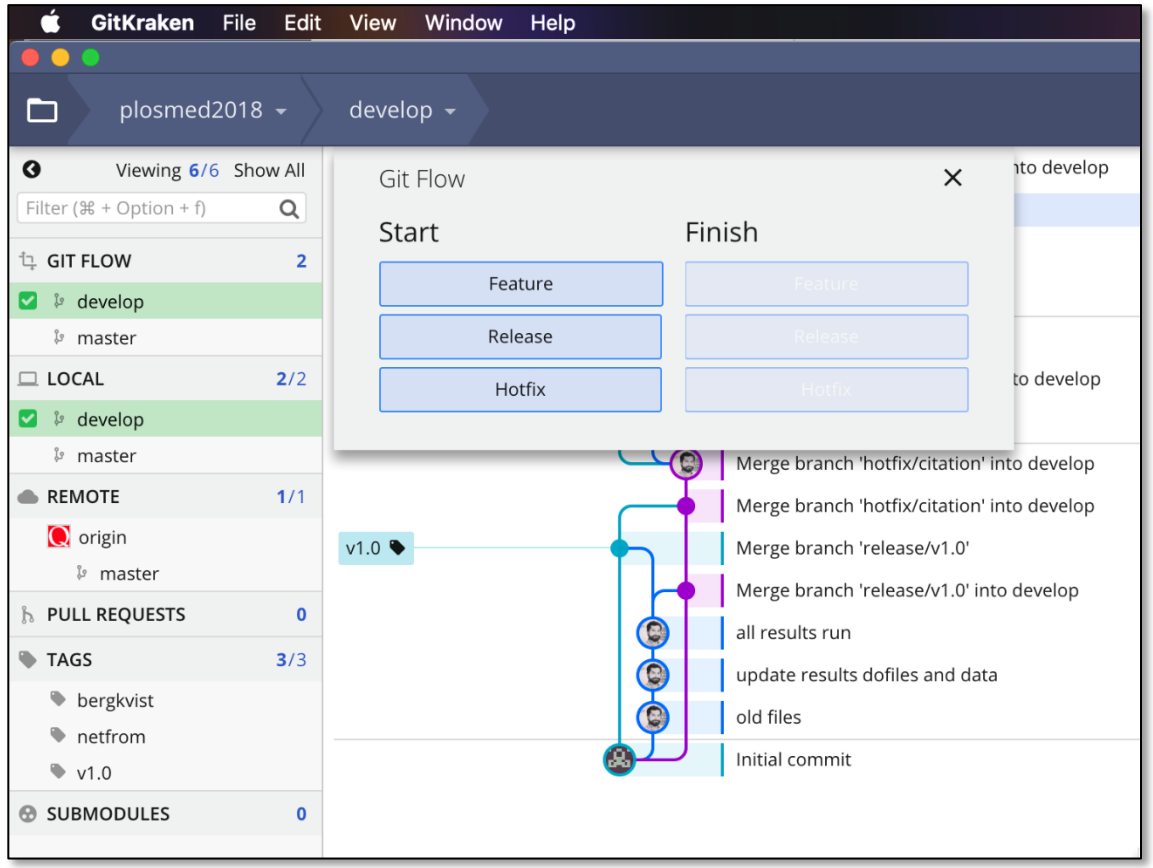
Benefits of Git Flow

- Anyone (like the PI) can *always* open [master] branch and see “where we are”.
- Anyone (like the PI) can create a [feature] branch and ask “what happens if I do X?”
- You can always look back and see what happened when someone tried something else (abandoned [feature] branches)

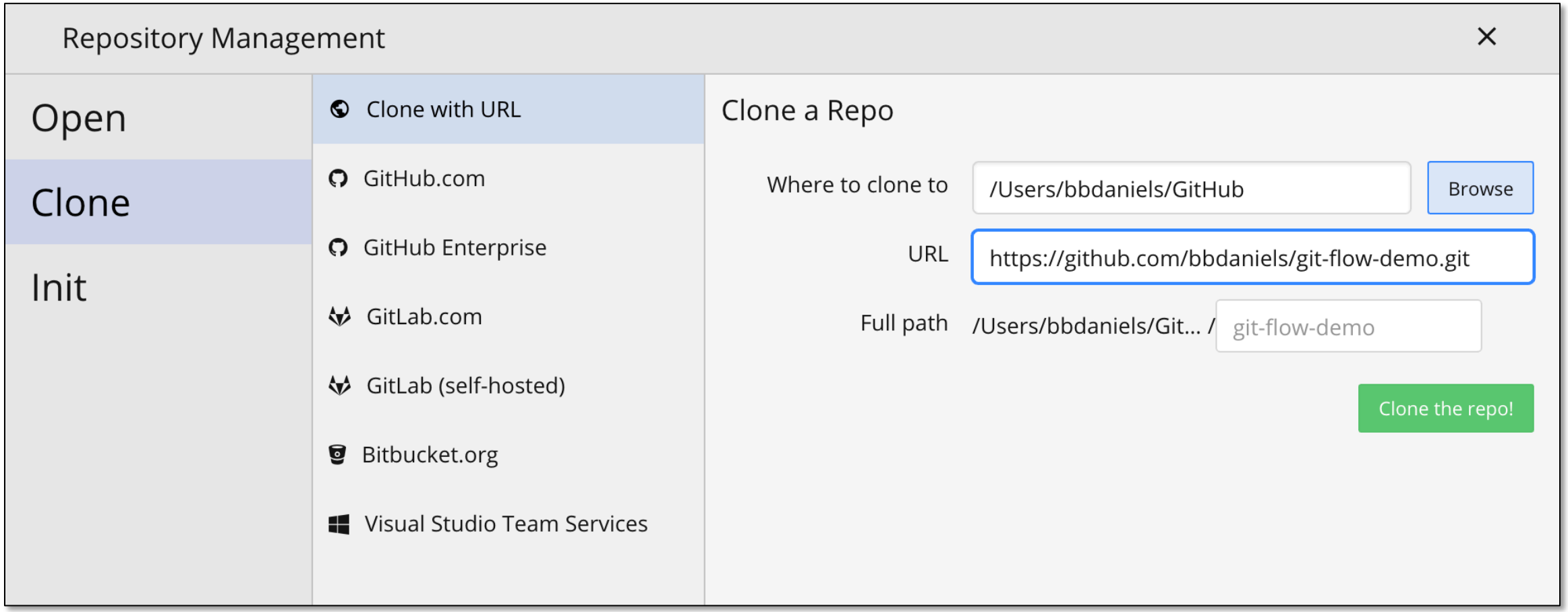


Using Git Flow with GitKraken

- Git Flow is so successful it is the default in many organizations, and GitKraken has powerful automatic features for supporting Git Flow.
- Send me your GitHub username so I can give you edit access on the git-flow-demo repo



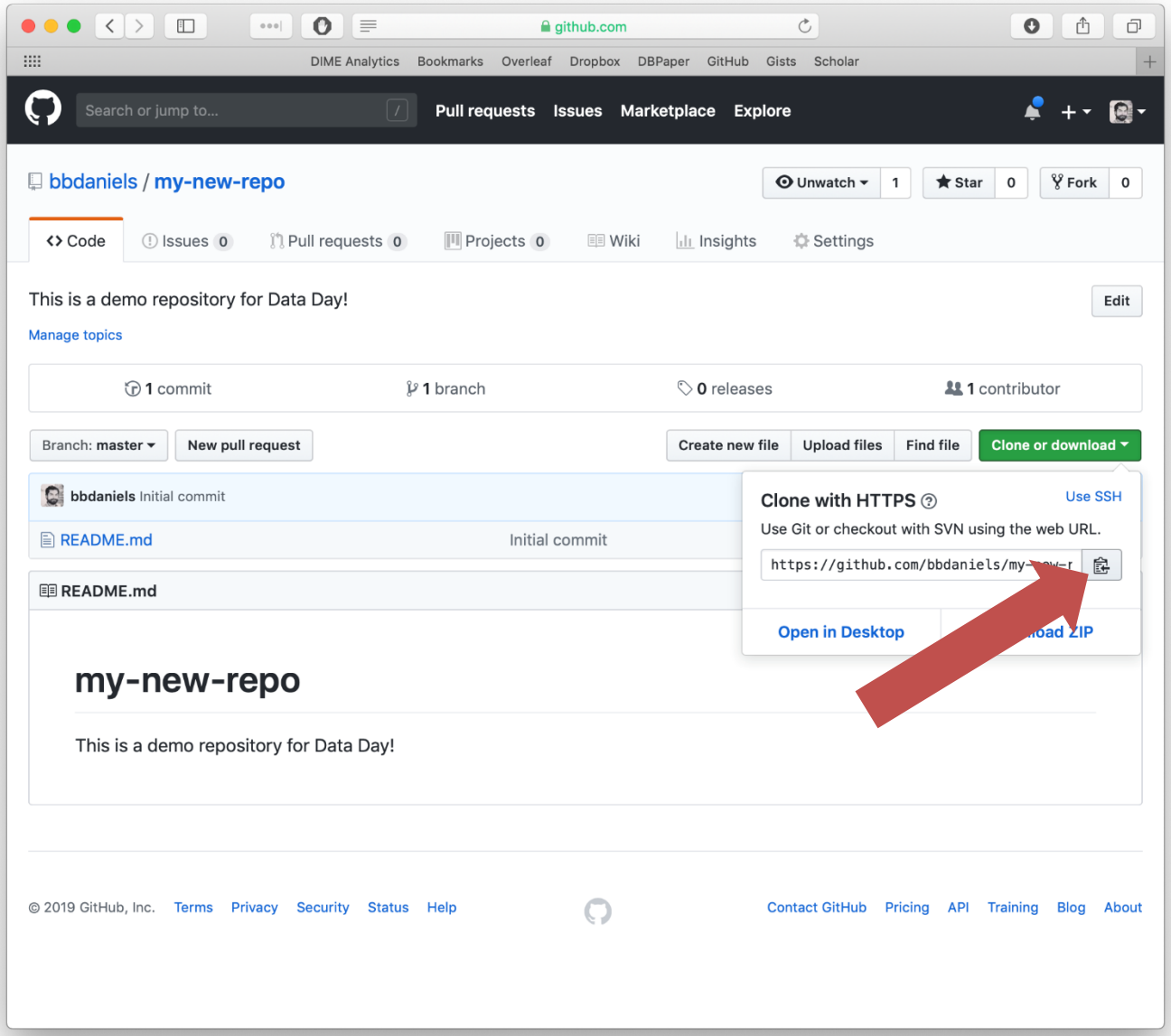
Clone the git-flow-demo repo locally



Clone the repo with your desktop client

Cloning the repository makes a complete copy of it at the new location, including the entire history (remember, the repository is the history).

Cloning from GitHub to your local computer is a good way to start a repository.

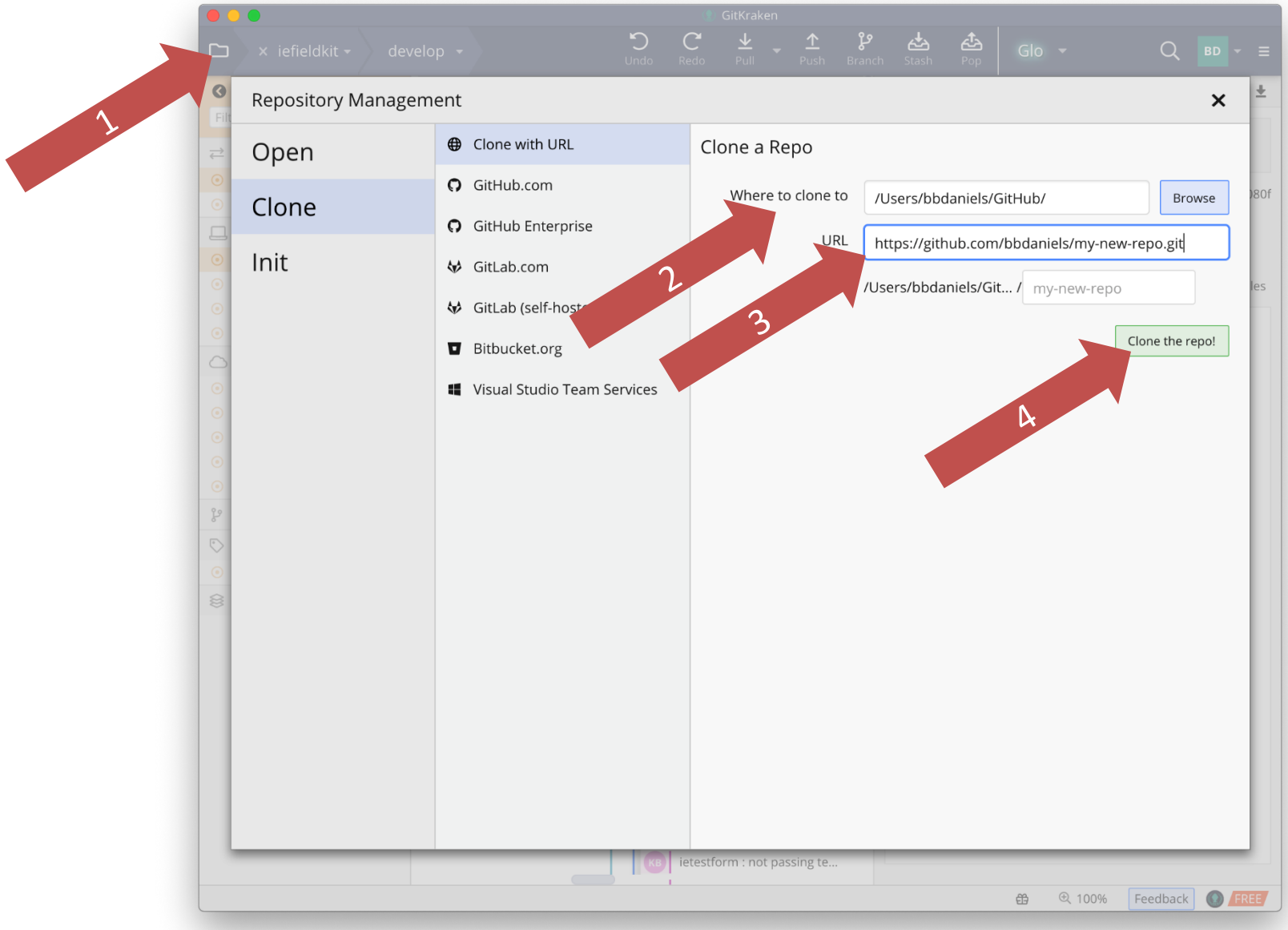


Clone the repo with your desktop client

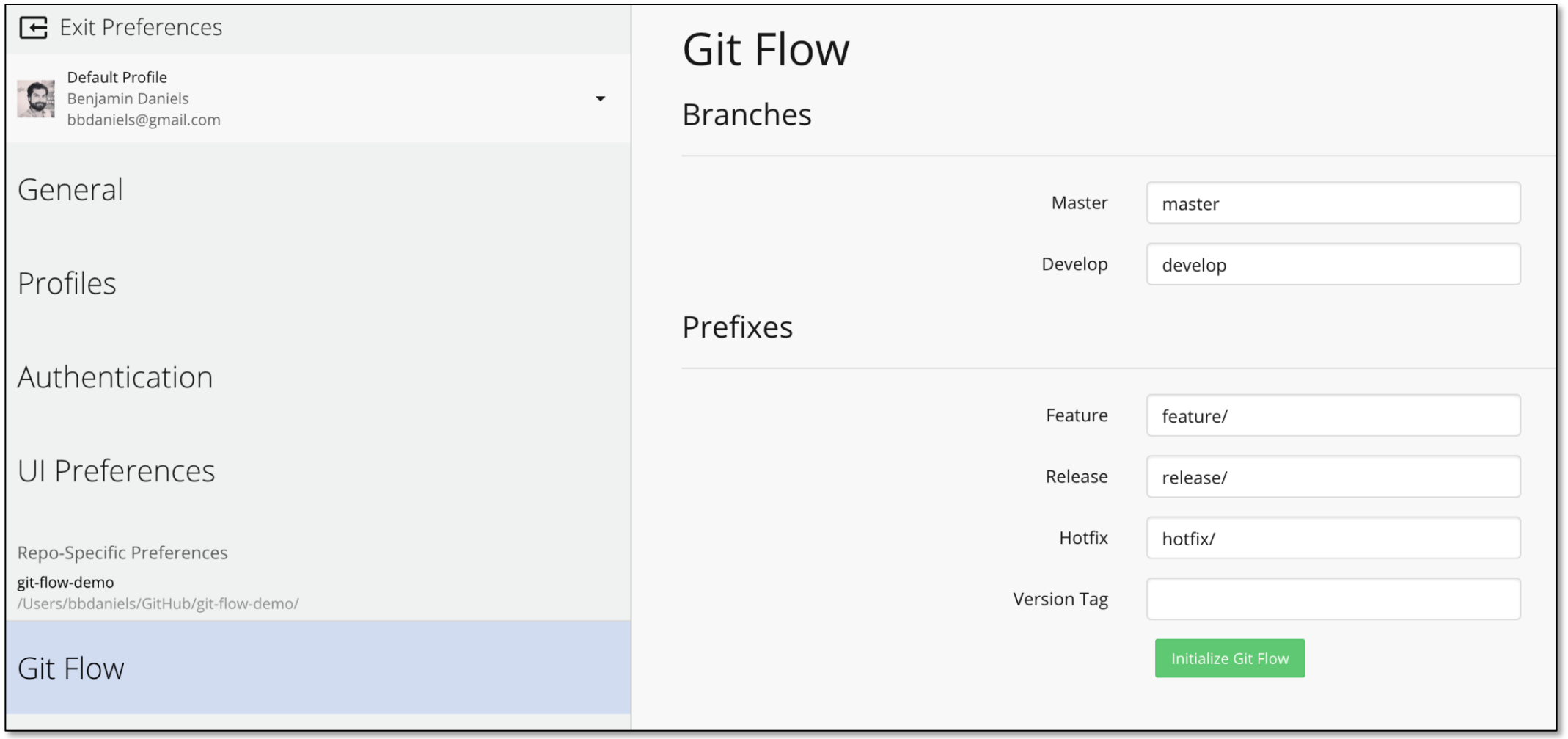
Cloning the repository makes a complete copy of it at the new location, including the entire history (remember, the repository is the history).

Cloning from GitHub to your local computer is a good way to start a repository.

(I know it seems like a lot of steps now, but this becomes second nature. Bear with me, and use this presentation as a guide!)



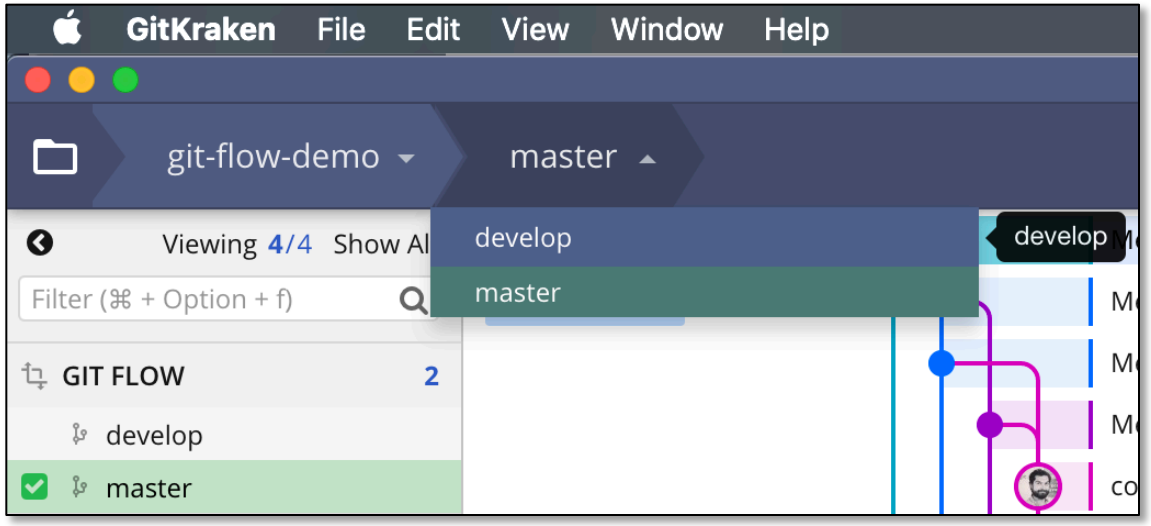
Initialize Git Flow in “Preferences”



Checkout “develop”

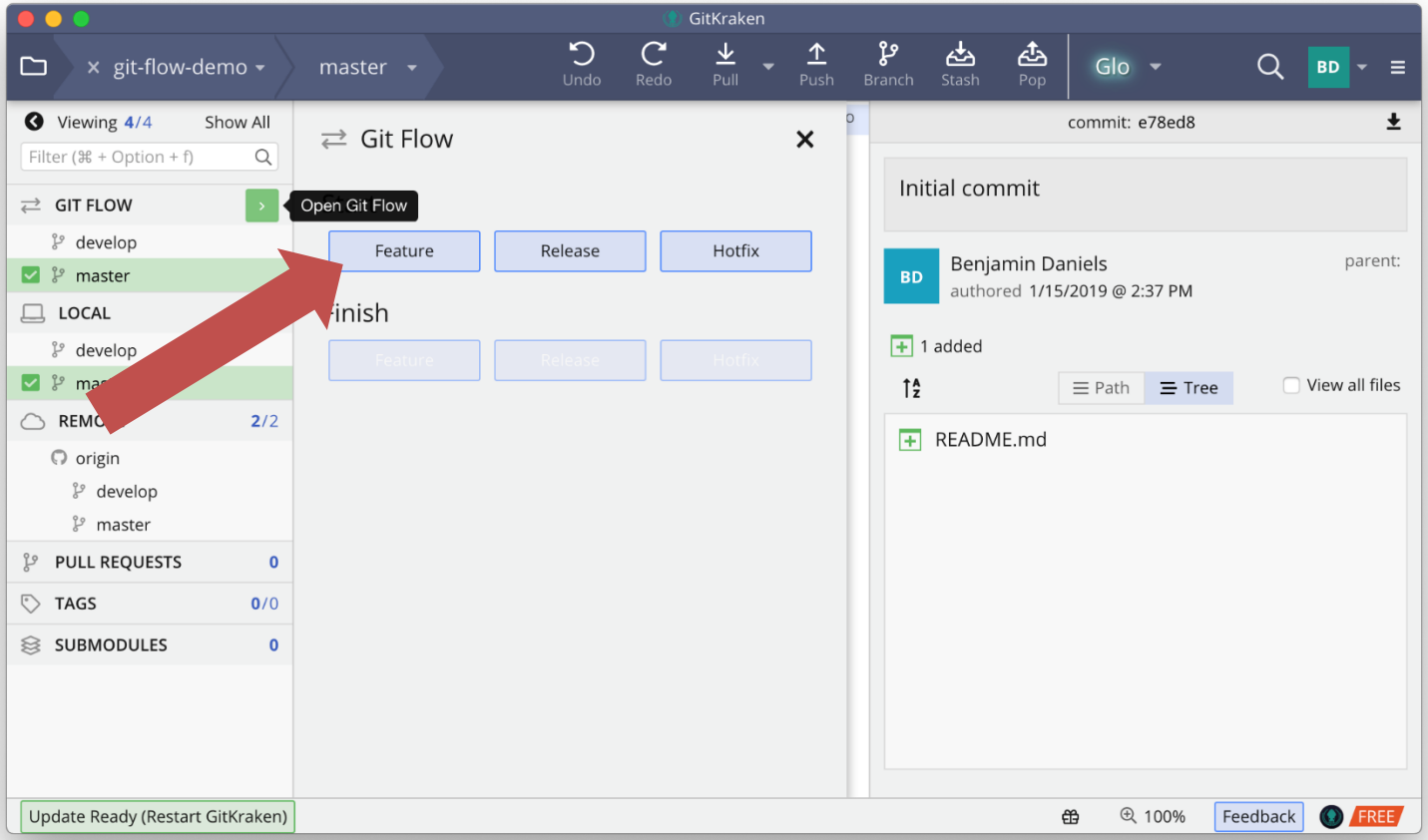
When you “check out” a branch, everything on your computer is instantly updated to what that branch looks like.

Always be conscious to remember “where” you are – this is the core functionality of Git. You’ll get used to it!



Start a new [feature] branch

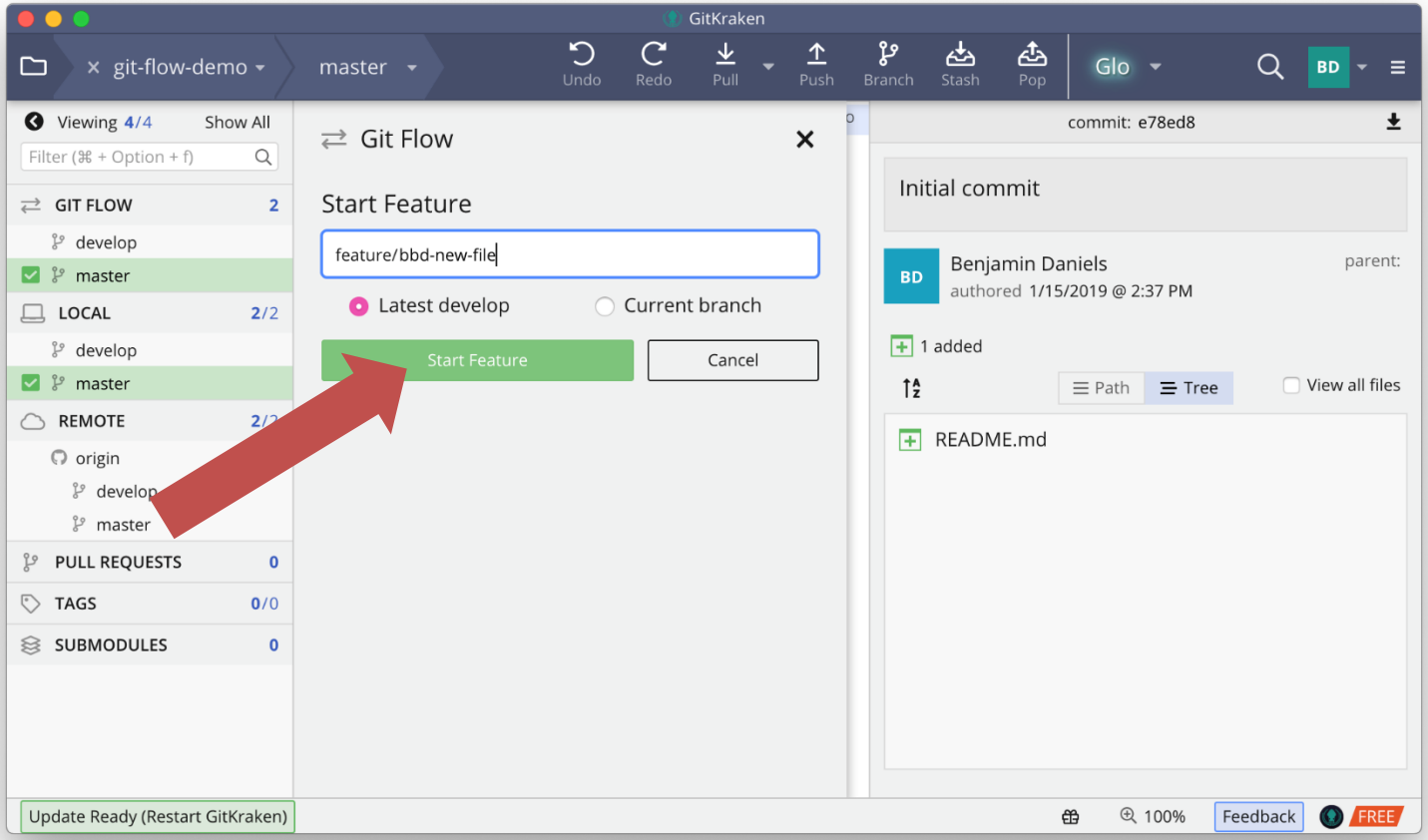
On your local machine, use the Git Flow popout to start a new feature. Call it [*xx*-new-file] where *xx* is your initials.



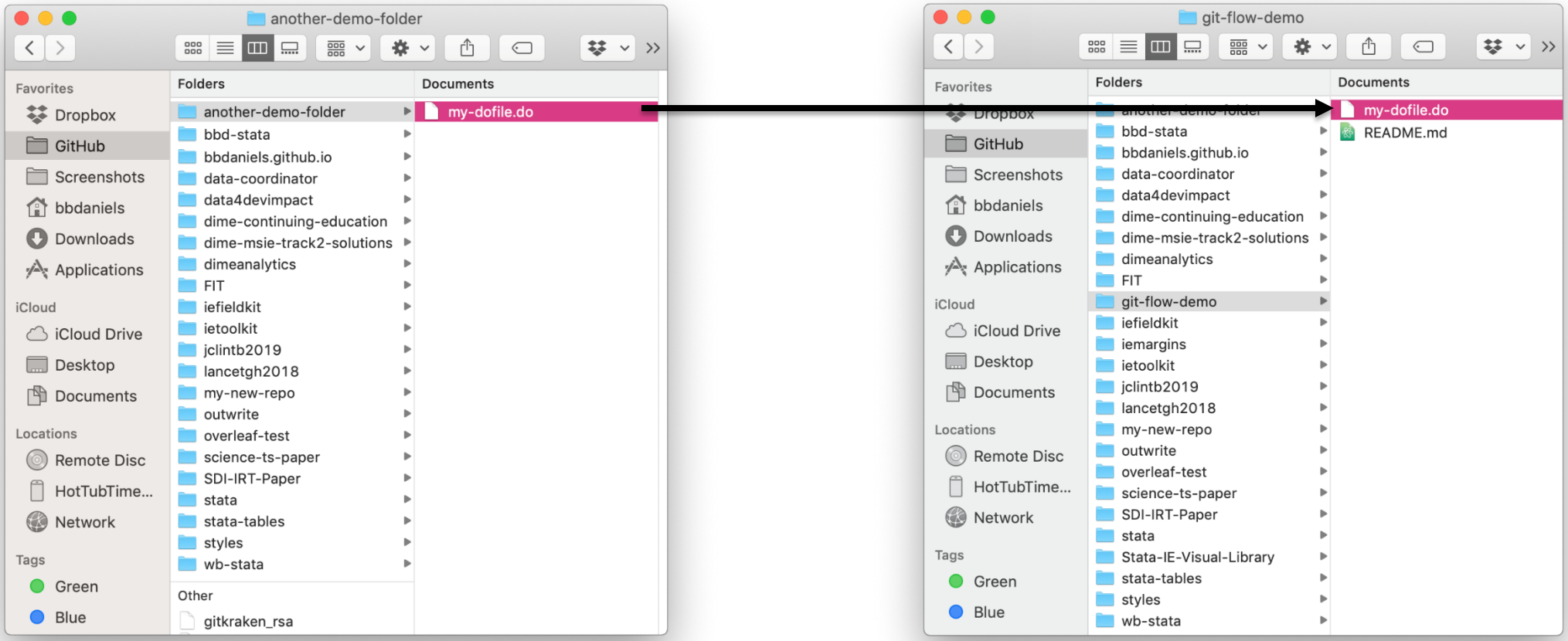
Start a new [feature] branch

On your local machine, use the Git Flow popout to start a new feature. Call it [*xx*-new-file] where *xx* is your initials.

[feature] branch names should be single-person and single-purpose: the name is a great place to be clear about those.



Add some kind of file locally (xx-file initials)



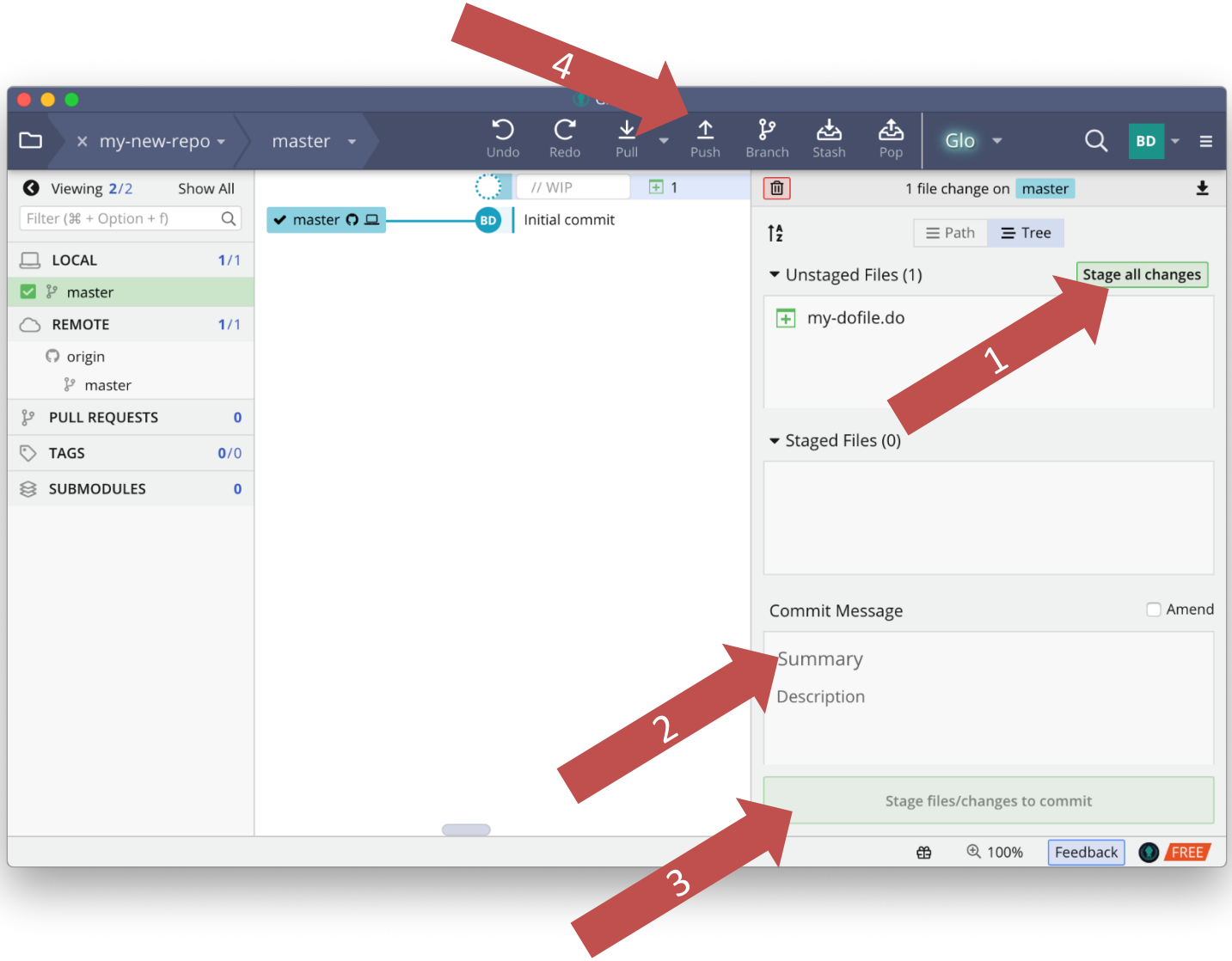
Git notes changes as “work in progress”

1. “Stage” your changes to add them to the queue to be committed.

2. “Name” your changes informatively – a short sentence will do nicely.

3. “Commit” your changes to add them to the version history.

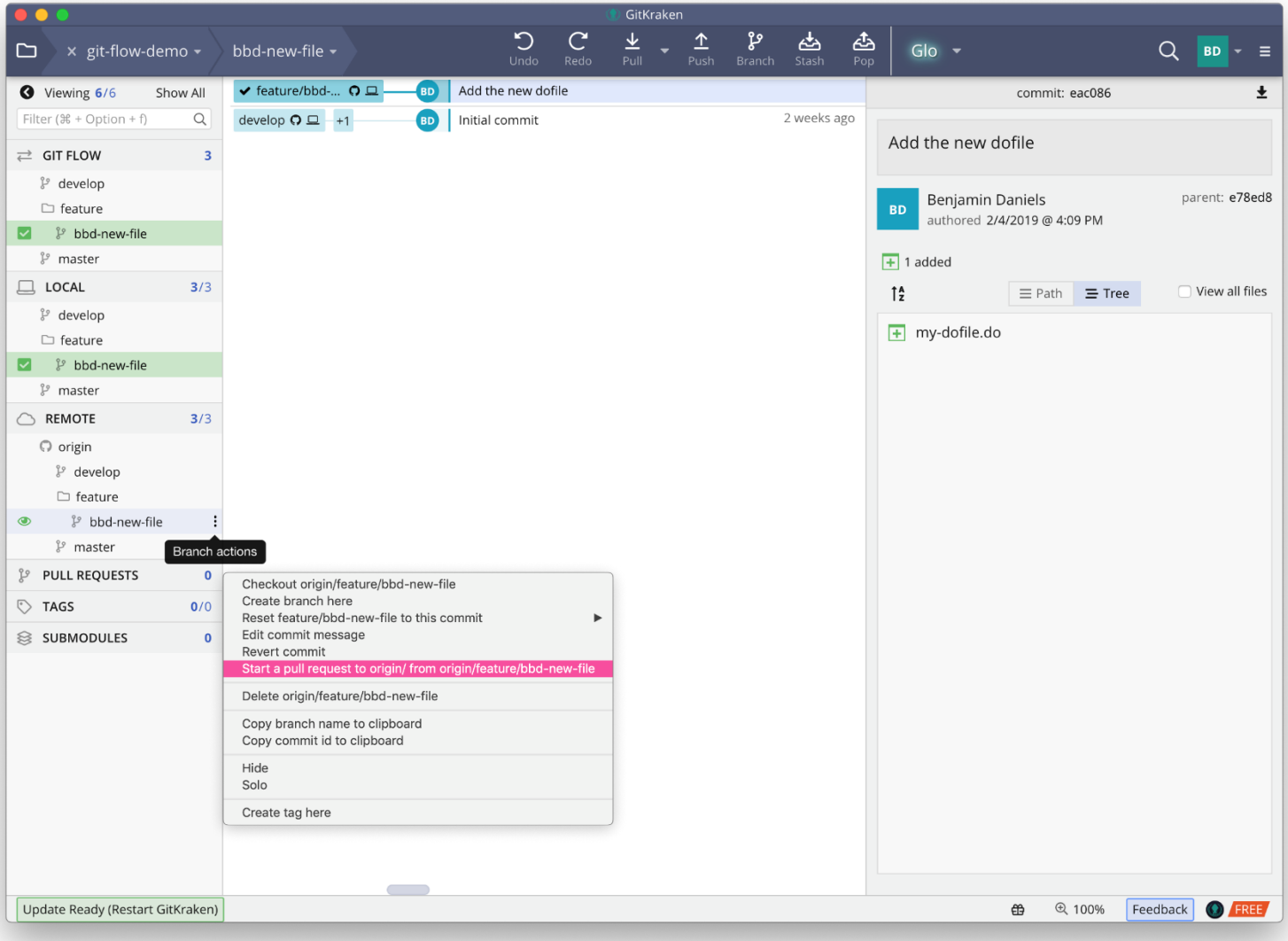
4. “Push” your changes to sync the remote (GitHub) repository with your local copy.



Start a pull request

This can be done by using the context menu from the feature branch under the “remote/origin” header – because this is a GitHub operation, not a Git operation

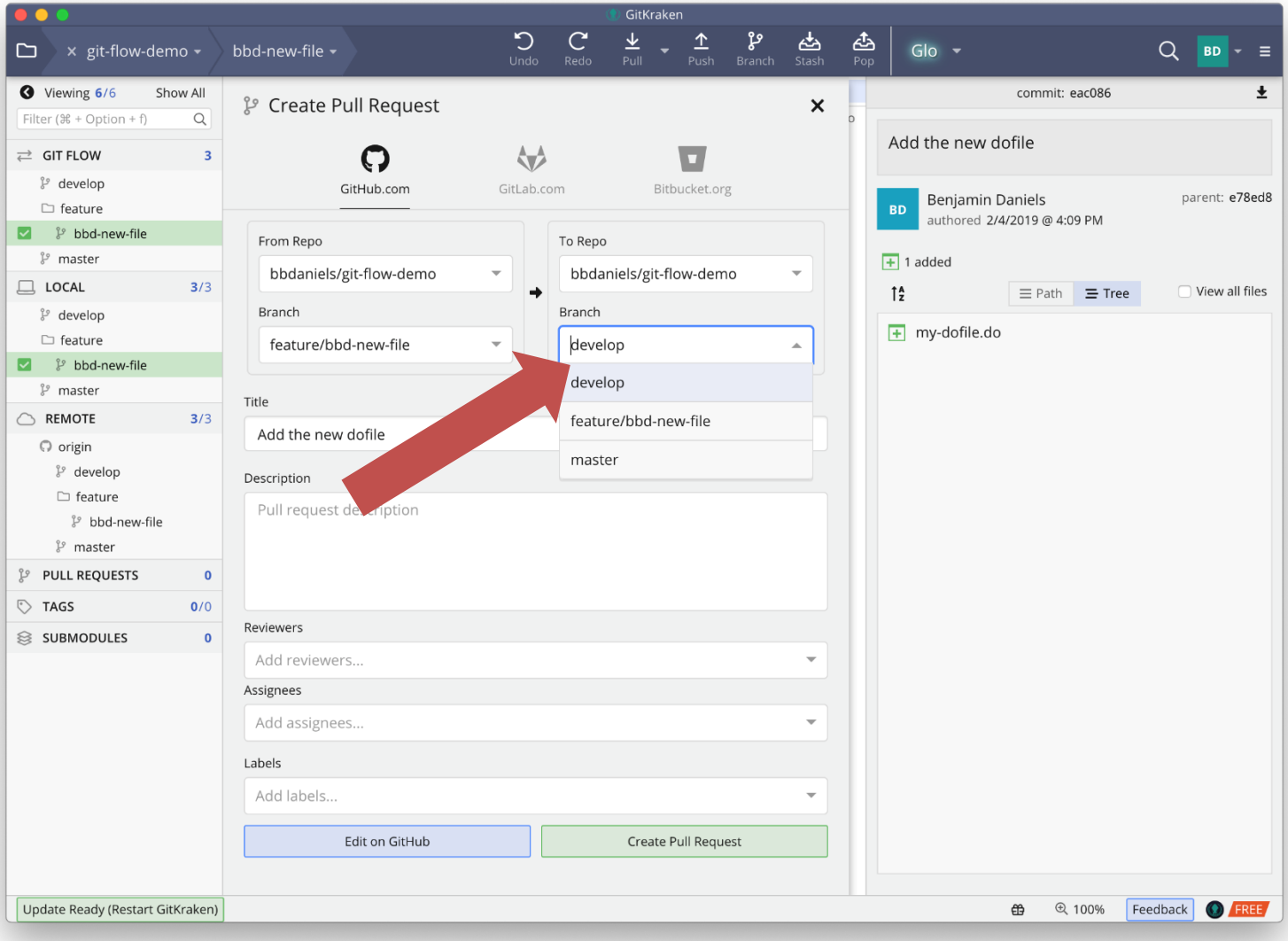
You can also drag-and-drop the feature branch onto “origin/develop”.



Make sure you target *develop*

Next, as the administrator, I'll resolve these on GitHub.

Once the branch is pushed and PRed, you can delete the branch locally.



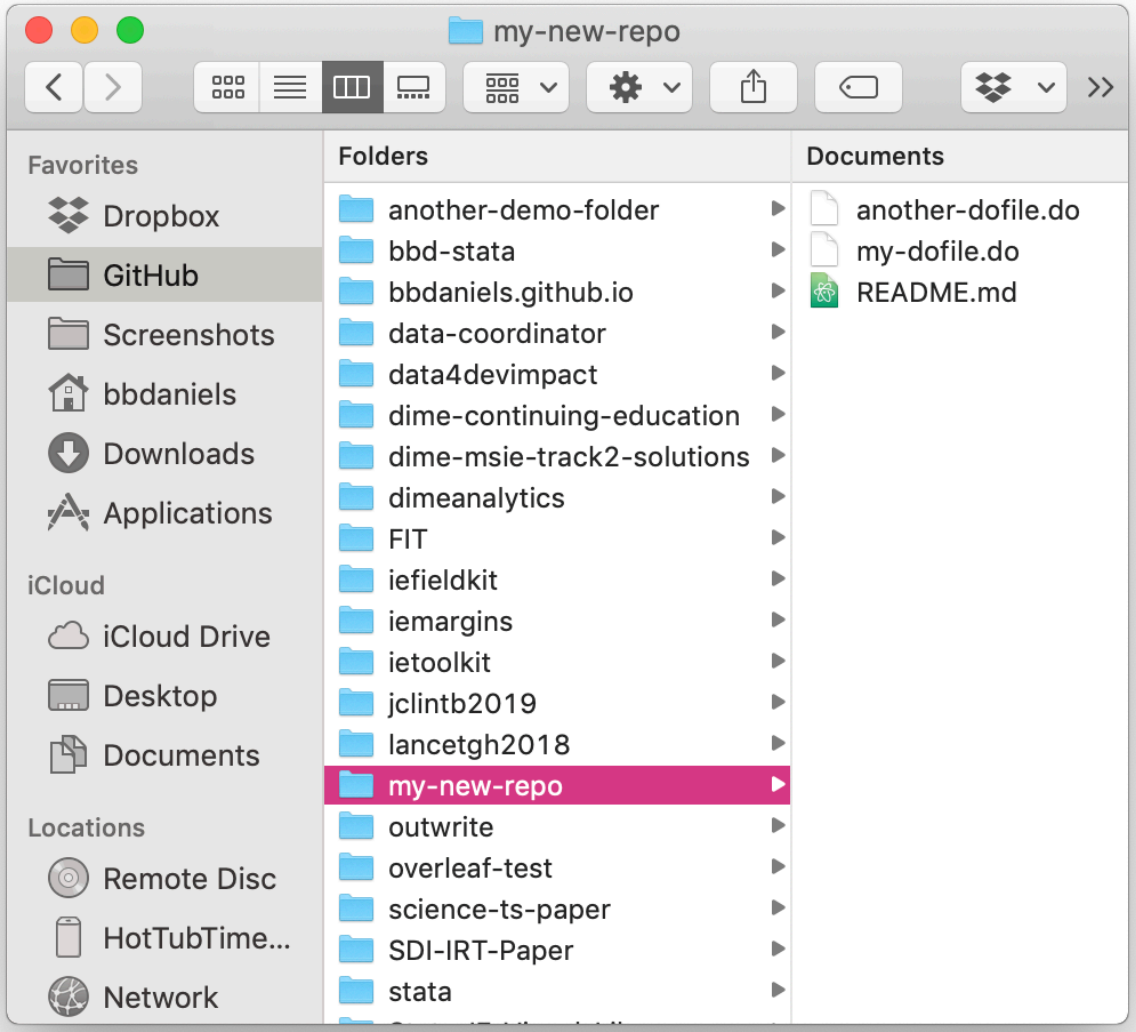
Merging consolidates changes

In the local working directory, you will now see that *both* files now exist in the same commit, “Merge pull request #1 from bbdaniels/feature/…”, on the *develop* branch.

Then I will merge these to *master* so that all those changes are only reflected in a single *master* commit: a new version!

If people made *conflicting* changes to the same files, this process also gives you a chance to resolve those, and this and other workflows will be covered in a later session.

That’s all for now!



Thank you!