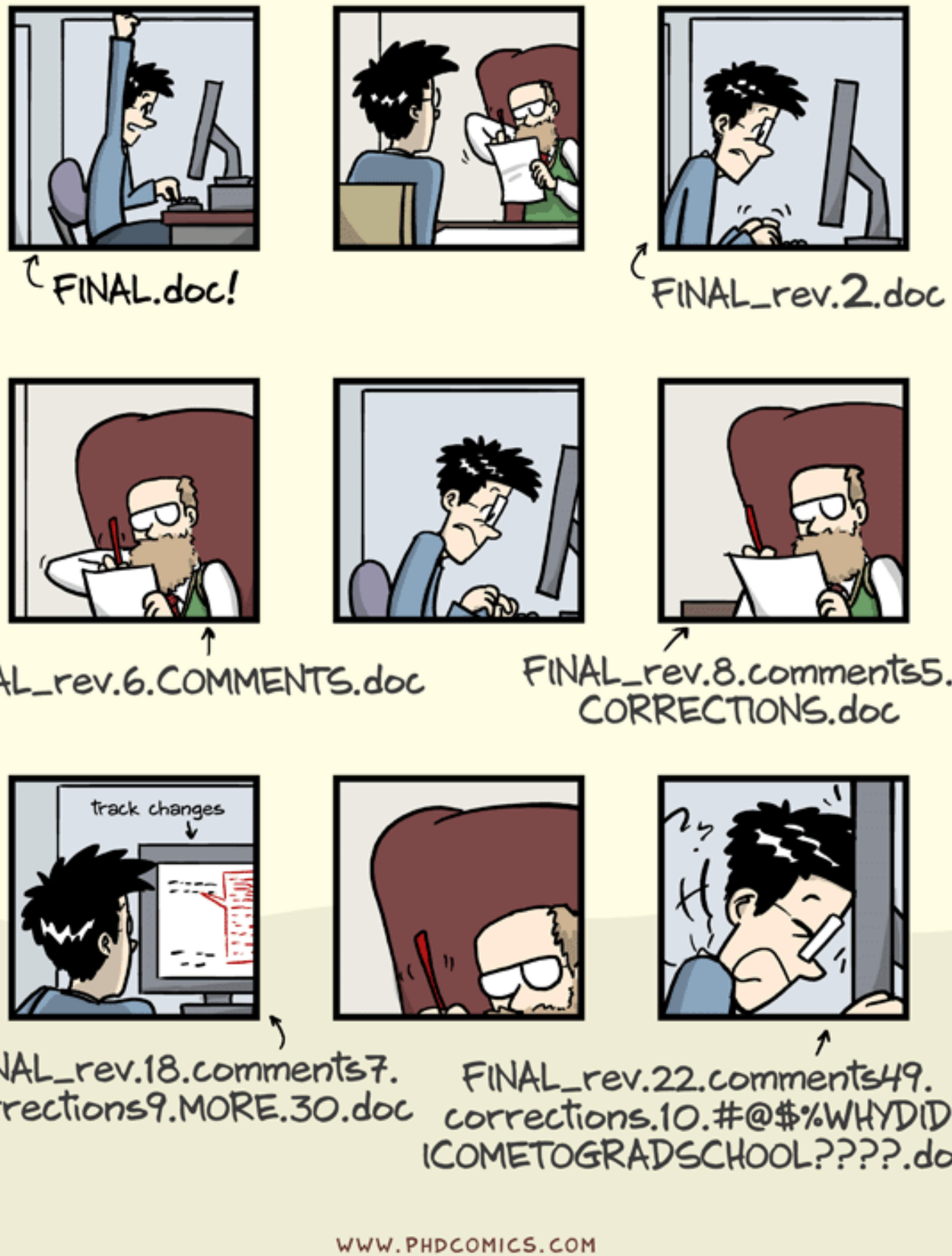
# Modern Analysis Techniques for Large Data Sets

Miguel F. Morales
Bryna Hazelton

# "FINAL".doc



# git and GitHub

- git
  - version control
  - local + remote

- GitHub
  - Tools for collaboration
    - Issue tracking, pull requests with code review, forking
  - hosting & public access

# Why version control (and git)

- simplicity
  - only need one copy
  - always clear what the current version is
  - git just tracks changes
  - backup!
- freedom to delete code
  - git keeps the full history, you can always resurrect old code
  - don’t need to keep commented code around ‘just in case’
- provenance and reproducibility
  - makes it possible to track exactly what code was run for any analysis
  - fine-grained history
- good support for branching and merging
  - supports development separate from a stable ‘main’ branch
  - aids parallel development

# GitHub collaboration tools

- Issue tracking
  - with labelling, assignments and links between issues and PRs
- Pull Requests (PRs)
  - build code reviews into the process of merging in new functionality
- integrations with other services
  - Continuous integration: tests and other checks run every time the repo is updated
  - Documentation hosting: rebuild the documentation every time the repo is updated
- user interface for exploring code changes
  - graphical diffs between any commits or branches

# git basics

- local repository
  - a complete copy (with the full history) on your local machine
  - self-contained and self-sufficient

- remote repository
  - a complete copy hosted remotely (e.g. on GitHub) — the repository all collaborators have access to

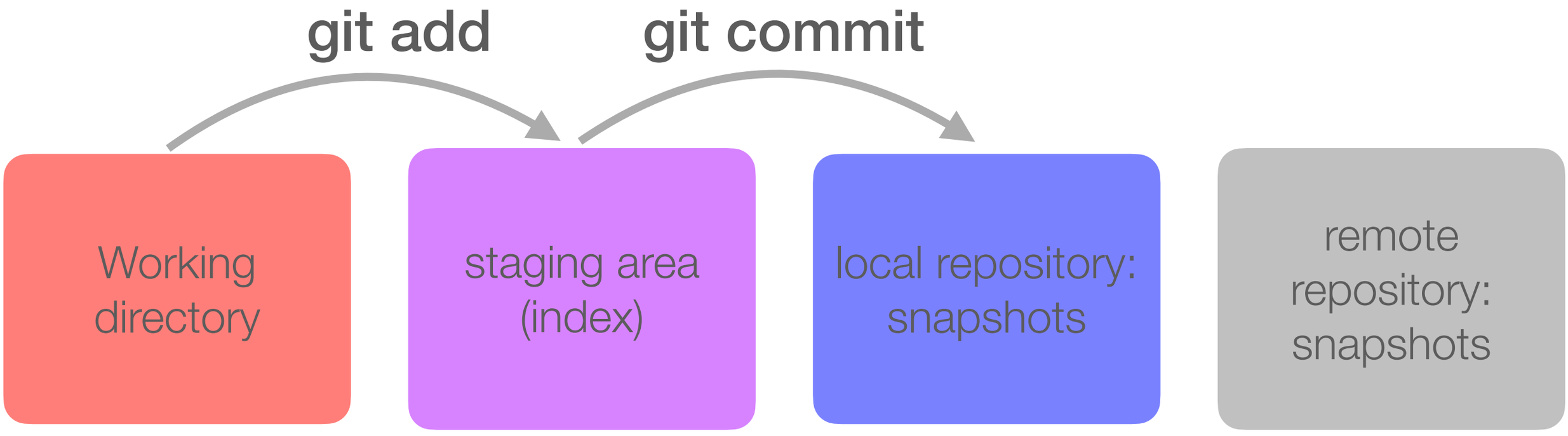
- snapshots (commits)
  - the unit of tracking within git — can be multiple changes to multiple files
  - should be used to identify ‘atomic’ changes — things that go together
  - commit early & often — fine-grained commits make the history more useful

Working directory

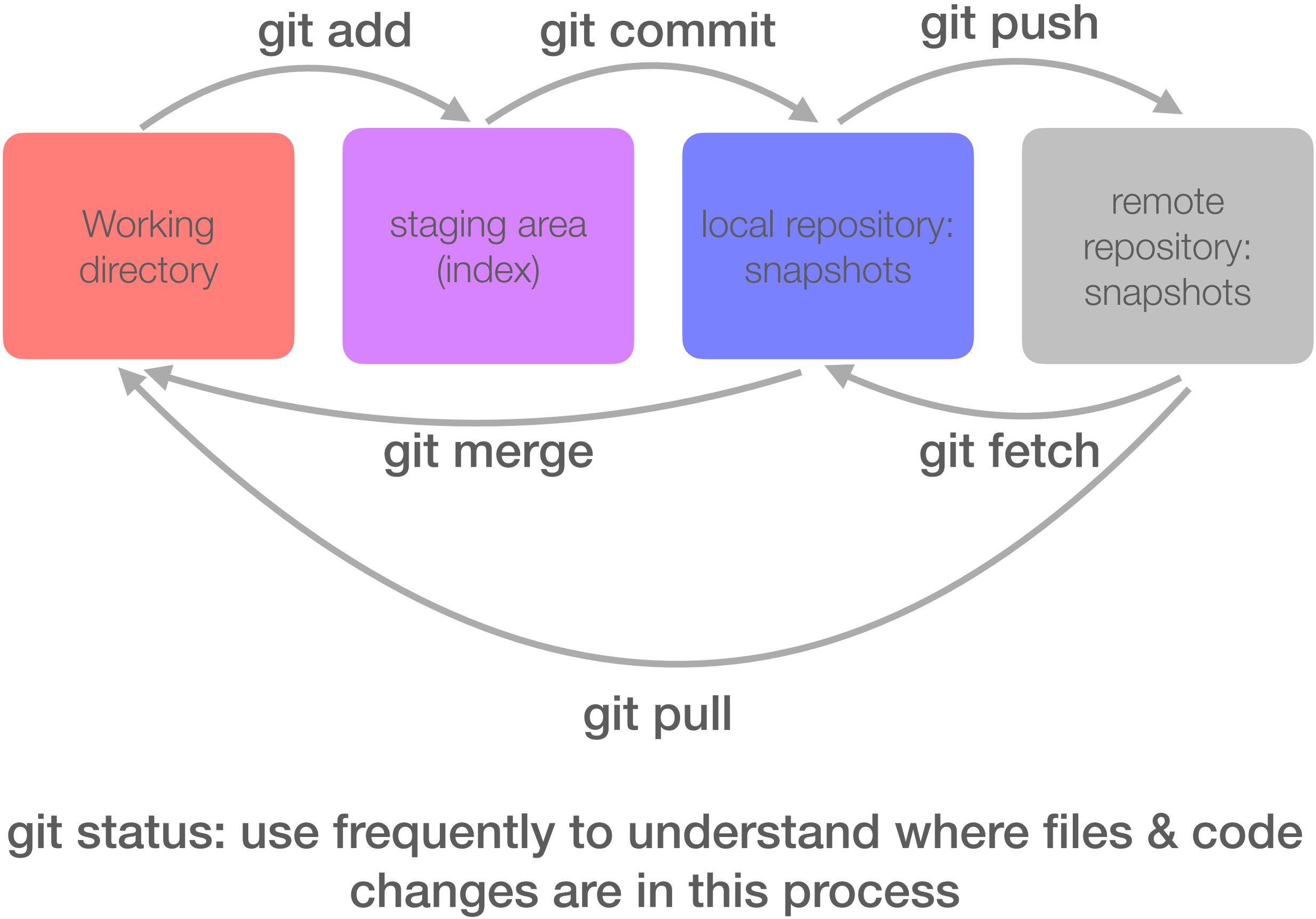
staging area (index)

local repository: snapshots

remote repository: snapshots




git status: use frequently to understand where files & code changes are in this process




git status: use frequently to understand where files & code changes are in this process

# Live Demo with GitKraken

# git user interfaces

- command line
  - most control and awareness of what you are doing
  - can be hard to visualize the process
- gui (GitKraken, SourceTree, Lazygit — terminal gui!)
  - good for visualizing the process
  - great interface for viewing history and diffs
  - encourages some good practices (viewing changes before adding)
  - easy to do powerful things (add parts of files, deal with merge conflicts, undo)
  - can obscure details or make it too easy to make mistakes
- GitHub
  - great interface for viewing history and diffs, but restricted to what's on the remote
  - required for issue tracking and pull request management

# Making a new repository

- On GitHub
  - choose public or private
  - initialize with a readme
  - choose a .gitignore (also see https://github.com/github/gitignore for more language options, including matlab)
  - choose a license

- Clone the repository locally
  - ssh vs https
  - `git clone <repo-address>`
  - `git remote -v` to see the address of the remote

# git config

- Global settings for git
  - name and email address (to identify who made changes) — should match email associated with your GitHub account
  - preferred text editor (for commit messages)
- To see current settings:
  - `git config –list`
- To change these settings:
  - `git config --global user.name "Vlad Dracula"`
  - `git config --global user.email "vlad@tran.sylvan.ia"`
  - `git config --global core.editor "nano -w”`

# making changes

- Check the status
  - use `git status` to see what things have changed
  - use this command liberally — it's always safe and helps you know what's going on
- Identify all the changes you want to snapshot together
  - use `git diff` to see what the changes are
  - use `git add <file>` to move changes to the staging area
  - only include changes that go together
- make the snapshot
  - use `git commit` to make the snapshot: brings up a browser to add a commit message
  - or `git commit -m 'your message here'`
  - commit messages should be **descriptive**
- view the history
  - `git log`, `git show`

# syncing with the remote

- get snapshots from the remote
  - use **`git fetch`** to get the snapshots but not apply them to the local repo
  - use **`git status`** to see differences between the local and remote
  - use **`git merge`** to apply the snapshots to the local repo
  - **`git pull`** is **`git fetch`** immediately followed by **`git merge`** but doesn't let you examine the snapshots before applying them
- send your snapshots to the remote
  - use **`git push`** to send your local snapshots to the remote
  - git will not let you push if there are snapshots on the remote that you have not yet merged into your local repository
- view the history
  - **`git log`**, **`git show`**

# branching

- create a new branch and switch to it
  - use `git checkout -b <branch_name>` to create the branch and switch to it
  - for existing branches, use `git checkout <branch_name>` to switch to that branch
- make changes and snapshots on that branch
- push the branch up to the remote
  - use `git push --set-upstream origin <branch_name>` to make a branch on the remote that tracks your new local branch
  - make more changes and snapshots and push/pull
- to merge the branch into the main, make a pull request
  - leads to a code review

# merging vs rebasing branches

- merging in branches is straightforward, but can result in a somewhat complicated graph
- rebasing is an alternative approach that results in a neat, linear graph at the expense of rewriting history
- rebasing effectively moves the location that a branch leaves the tree
  - can be used to place branches at the tip of the master branch to avoid having to merge
  - effectively replays the changes in the branch after the end of the master branch

# Resources

- git parable (conceptually building up why git is the way it is): https://tom.preston-werner.com/2009/05/19/the-git-parable.html

- Software Carpentry hands-on tutorial: http://swcarpentry.github.io/git-novice/

- Lab-style git intro: https://github.com/HERA-Team/CHAMP_Bootcamp/blob/master/Lesson2_IntroToComputing/git-lab-handout.pdf

- eScience office hours (https://escience.washington.edu/office-hours/#eScienceDataScientists)

# Collaborating with GitHub

# Collaborative Data Analysis

- Issues for discussion threads (e.g. https://github.com/EoRImaging/FHD/issues/39)
  - threaded logbook linked to code
  - making your advisor useful to you
  - use plots!
  - closing to mark as resolved

# Collaborating on code

- Branching workflow
  - make a branch for a specific topic
  - you can have multiple branches!
- Pull Requests for code reviews (pyuvdata link)
  - linking to issues
- GitHub Milestones/Projects
  - track sets of issues towards a larger goal

# Reproducibility & Open Science

- public code
  - complementary (some times required) to publishing
  - documentation and readability are key
  - the analysis is the code — allows others to see what you did

- open source code
  - requires an open source license
  - API documentation
  - unit testing and continuous integration
  - building a user community

- reproducibility
  - capture of versions & settings
  - open data
  - full stack capture, containers (docker)