Introduction to Version Control and Project Management with Git and GitHub: Part I

Dmitri Svetlov

School of Biomedical and Chemical Engineering, Colorado State University

Dmitri.Svetlov@ColoState.edu

CM 515 Spring 2026

Agenda

About Me

- Training
  - Chemistry
  - Philosophy (undergraduate only)
  - Computer Science
  - Biomedical Engineering (graduate only)
- Academia + National Labs
  - Molecular virology and biology/biophysical chemistry (OSU and Wisconsin)
  - Computational materials chemistry (USC and Los Alamos)
  - Soil and Crop Sciences and Biomedical Engineering (CSU)
- Industry
  - Scientific software development (De Novo Software; analysis and visualization of flow cytometry data)

Reproducibility in Science

- *Per* the National Academies in *Reproducibility and Replicability in Science*:
  - **Reproducibility**: "consistent computational results using the same input data, computational steps, methods, code, and conditions of analysis"
  - **Replicability**: "consistent results across studies aimed at answering the same scientific question, each of which has obtained its own data"
  - **Generalizability**: "the extent that results of a study apply in other contexts or populations that differ from the original one"

Additional Considerations for Computational Science

- "The training of scientists in best computational research practices has not kept pace [with the emergence or expansion of 'fields of science focused solely on computation']."
- "Many decisions related to data selection or parameter setting for code are made throughout a study and can affect the results", and these are not always captured and included in the record.
- Authorship and assignment of credit is greatly complicated by the nature of computational research: contributions to data, experiment design, and code are not automatically separable.

Additional Considerations for Computational Science, continued

- The scientific research enterprise is *cumulative*: others (and you) cannot extend, generalize, or build upon your research if they cannot repeat it.
  - This is almost impossible unless **code and data** are *open-source* and remain publicly available in at least one dedicated archive for an appreciable length of time.

Key Concepts

- A **version control system (VCS)** is a software tool that effectively manages multiple versions of a project (represented by a set of files called a **repository**) by providing the following:
  - A record of all changes to all files and directory structures (creation, modification, deletion)
  - A record of authorship of such changes
  - An ability to compare the state of the project at any two (or more) **revisions** (or **commits**)
  - An ability to undo (**revert**) any changes
  - A facility for **branching** the project into arbitrarily many, orthogonal copies that can evolve independently of one another

Centralized and Decentralized VCS

Centralized

- A single server holds the authoritative repository.
- Users copy ("check out") some or all of the contents of the repository to a **working copy** on their local machines and make modifications there.
- To modify the repository itself, users must **commit** their changes by transmitting, *via* a network connection, their changes to the server.
- The leading centralized VCS is Apache Subversion (SVN), introduced in 2000.

Decentralized

- No single authoritative repository exists - all copies of the repository are peers!
- Each "client" has a copy of the entire repository and (its own) change history.
- The relationship between two repositories depends upon the direction of transferring changes:
  - A **push** occurs when changes on a local repository are sent to a remote server.
  - A **pull** occurs when changes are fetched from a remote server and applied to a local repository.
- The leading decentralized VCS is Git, introduced in 2005.

How Git Works: Theory I

- While some VCSes record changes, Git takes **snapshots** of the entire repository - each commit is a snapshot - and then calculates a checksum to ID each snapshot and accelerate comparison of snapshots.
  - The full checksum (40 hexademical characters) is guaranteed to be unique across all Git repositories ever!
  - Within a given repository, each commit is referenced by the first seven characters of its checksum, *e.g.* `b4488c5`, since this is generally enough.
- From snapshots, Git can create **patches** to allow modifications to be exchanged arbitrarily, even across repositories.
- Because of this, a commit has both an **author** and a **committer**, who may or may not be the same user.

How Git Works: Theory II

- **Branches** are used to group commits that share a common sequential history - each commit is an "ancestor" of all subsequent commits made on that branch.
- All repositories start with a single branch, often called `main` or `trunk` (`master` was also used historically).
- Additional branches can be created from a source branch, which will then be "upstream" of the new branch. This allows the repository to evolve in multiple, orthogonal directions.
- Changes can be readily **merged** between branches that share history. **Reintegration** is often used to describe merging upstream, particularly into main.

How Git Works: Theory III - Some Questions

- Can you make any analogies to evolutionary biology?
  - What is a Git branch?
  - What is happening when a new Git branch is created?
  - What is a Git commit?
  - What is the most recent Git commit?
  - What is the checksum of a Git commit?
  - What is happening if a Git committer isn't the same as the author?
  - What is happening when two Git branches are merged into one?

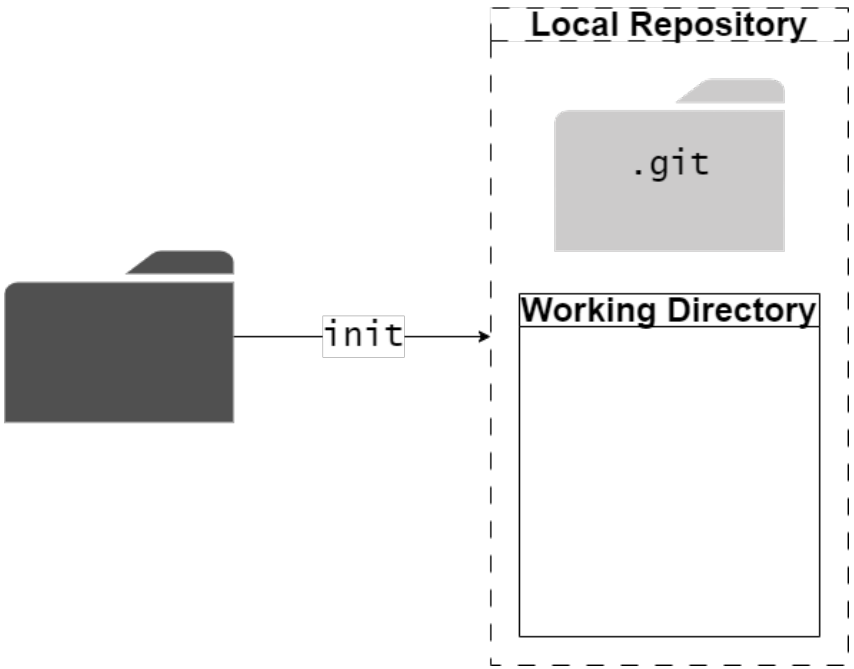
How Git Works: Theory III - The Answers

- Can you make any analogies to evolutionary biology?
  - A Git branch is a branch of a *phylogenetic tree*.
  - A new Git branch being created is an instance of *divergent evolution*.
  - A Git commit is an individual *species* in that (sub)tree.
  - The most recent Git commit is the latest species to evolve in that tree.
  - The checksum of a Git commit is a *genomic barcode* for the corresponding species.
  - If a Git committer isn't the same as the author, that means that the genomic changes were grafted from elsewhere...
  - Two Git branches being merged into one is an instance of true convergent evolution: two species creating a new one. (Take-home questions: Can this happen? How? Has it happened?)

How Git Works in Practice: Interfaces

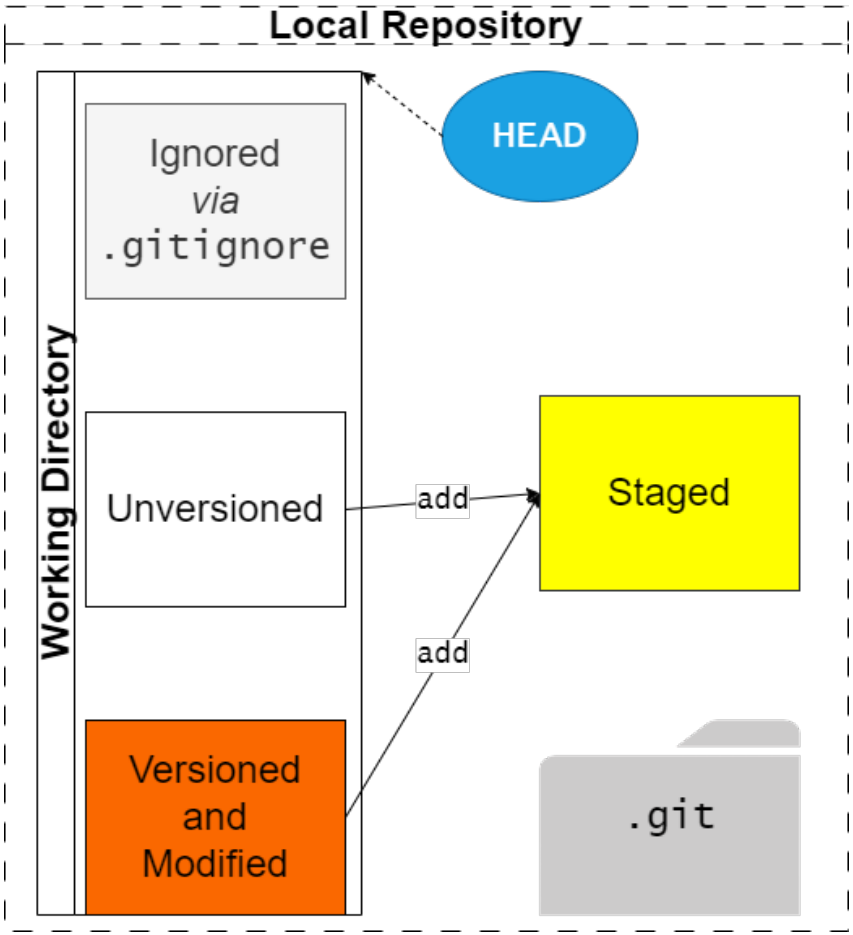
- To work with Git repositories, there are numerous tools:
  - The Git command-line program, `git`, which has sub-commands for all Git operations.
  - GUI tools, *e.g.* Git GUI and GitHub Desktop
  - Various IDEs integrate with Git, *e.g.* Visual Studio Code.
- Here, we will discuss the commands in the CLI: the other tools just provide graphical wrappers around these commands, but it's important to understand what the commands themselves do and how.
- On the subsequent slides, every command is a sub-command, *e.g.* `commit` is actually invoked as `git commit`.
- Also, in keeping with Unix and Linux philosophy, folders are also files, so I use "files" to refer to both.

How Git Works in Practice: Initialization



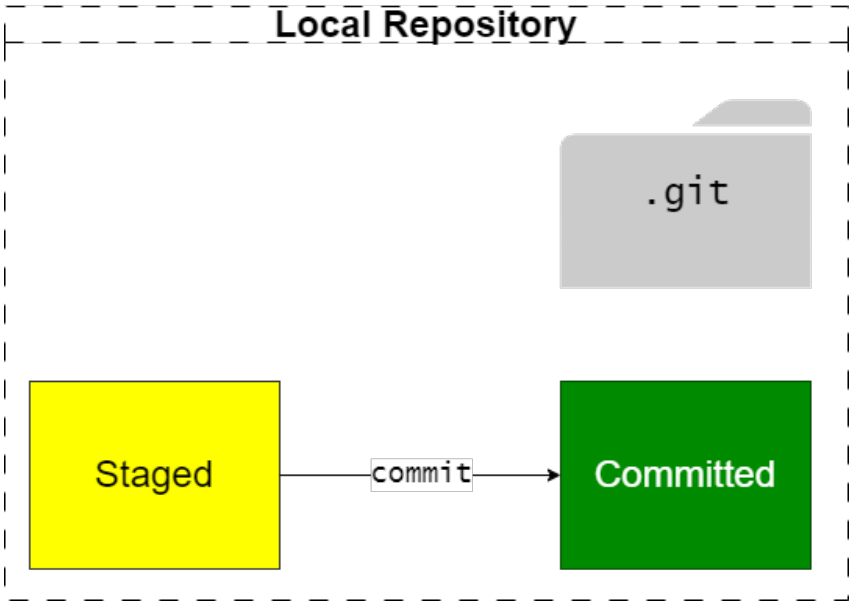
- One way is to call `init` on a directory on a local machine to convert that directory to a repository.
- The database itself (changesets, snapshots) is in a (new) subfolder called `.git`. **NEVER** modify this yourself!
- The repository uses a pointer called `HEAD` to track where the working directory is "supposed" to point, *i.e.* a particular branch or commit.

How Git Works in Practice: Staging



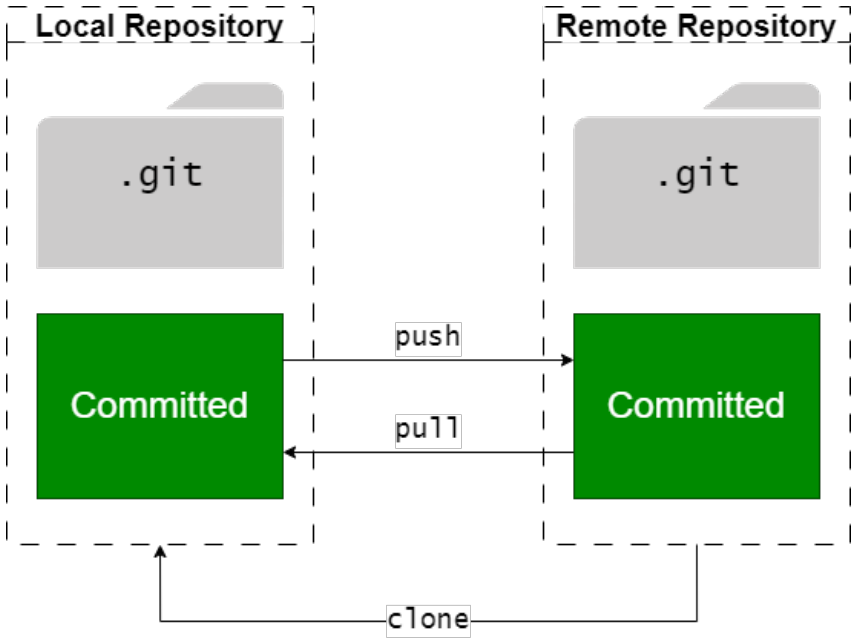
- **Staging** prepares files for **snapshotting**, *i.e.* their preservation in a commit.
- Three kinds of files are relevant here:
  - Files with a name or extension listed in a special `.gitignore` file are ignored by Git by default.
  - Unversioned files are also ignored until you choose to version them *via* `add`.
  - Even if a versioned file has been modified, the changes won't be snapshotted unless you `add` the file again.

How Git Works in Practice: Committing



- **Committing** actually takes the snapshot so that changes persist in the database.
- A **message** is used to describe what you are doing and why, not how you are doing it.
  - 50 characters in the headline
  - Additional lines if you must add more detail
  - Present tense
  - The code should have its own documentation!

How Git Works in Practice: Pushing and Pulling

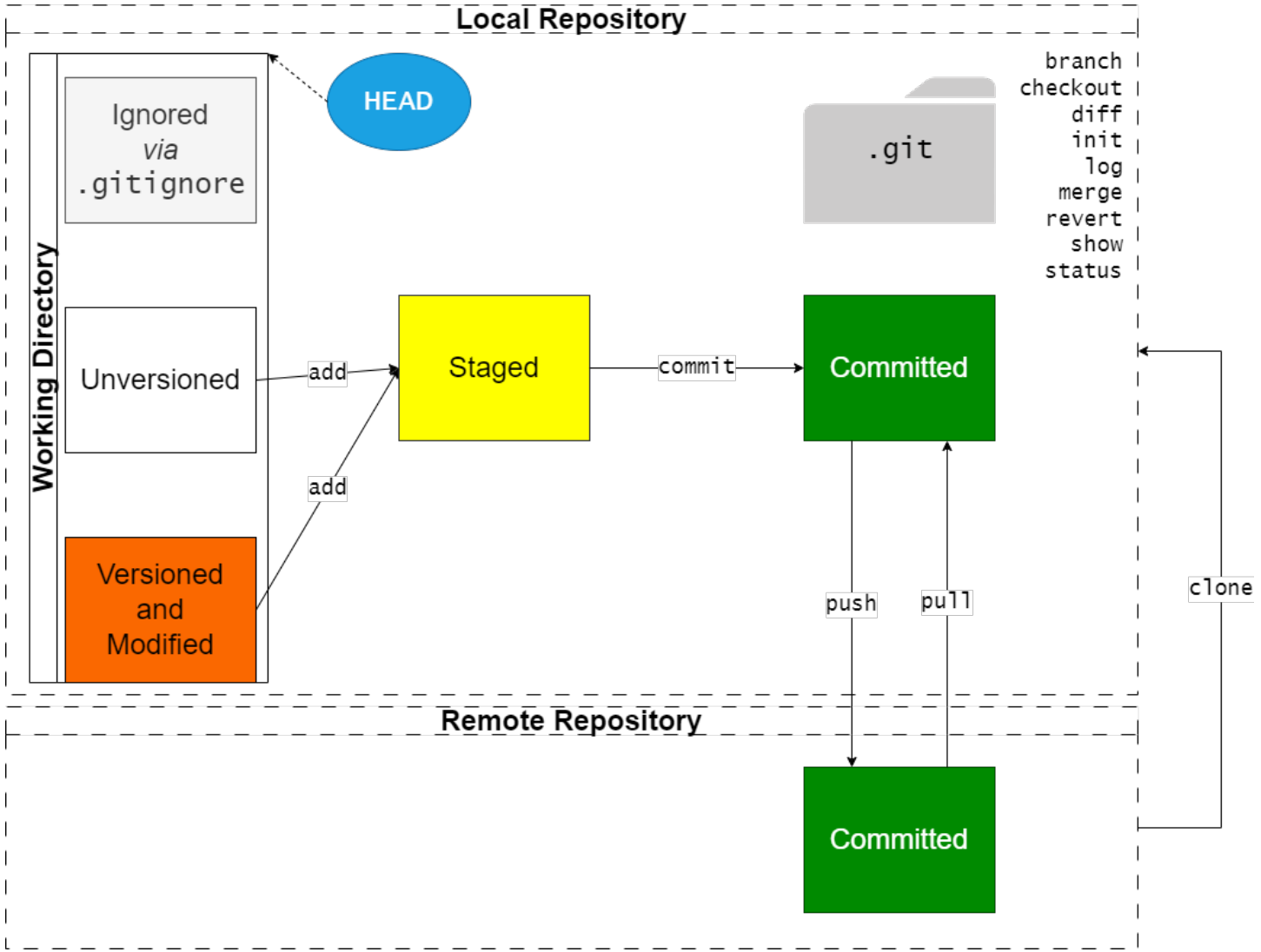


- Commits can be exchanged between (related!) local and remote repositories.
  - A **push** occurs when changes are sent from the local to the remote.
  - A **pull** occurs when changes are fetched from the remote and applied to the local.
- **Cloning** creates a local repository that is a complete copy (contents, histories, branches) of the remote.

How Git Works in Practice: Other Operations

`branch`	Creates a new branch
`checkout`	Switches `HEAD` to show a different branch or commit
`diff`	Shows content differences between two branches or commits
`log`	Lists version history for current branch
`merge`	Combines specified branch's history into current branch *via* a new commit
`revert`	Rolls back the changes made in the specified commit
`show`	Outputs metadata and content changes of the specified commit
`status`	Describes the state of the working directory (staged, unstaged, untracked) and of the local repository relative to an associated remote (if applicable)

How Git Works in Practice: Putting It All Together



Guided Practice

1. Visit the course GitHub repository and create your own fork.
2. Clone your fork onto your local machine, either using the CLI or a client (*e.g.* GitHub Desktop).
3. Create a new file called `NAME.md`, replacing `NAME` with your name (you can use Visual Studio Code to make a Markdown file). Add your name, your email address, GitHub username, and a photo of yourself to this file. See these instructions on GitHub Markdown.
4. Using the mechanism of your choice (CLI, GitHub Desktop, Visual Studio Code, *etc.*), add, commit, and push the changes. Visualize the history of your repository and compare it to that of the remote fork.
5. Visit your fork online and inspect your Markdown file. Correct any mistakes.

Guided Practice, continued

6. Online, modify your Markdown file to add hyperlinks to your email address (*e.g.* `mailto:svetlov@colostate.edu`) and to your GitHub username, *i.e.* link to your GitHub profile.
7. Update the clone on your local machine - which command(s) do you need to run? Visualize the history of your repository and compare it to that of the remote fork.

Want More Guided Practice?

- Want a **tutorial**? Read and work through either Tsitoara's book or Skoulikari's book. These will walk you through:
  - Installing Git and connecting to GitHub on local machines, with OS-specific instructions
  - All major Git operations: locally, on GitHub, and involving communication between the two
  - Conflict resolution, and several approaches to it
  - Important details of all of the above that were omitted here
- You can go through either in a (long) afternoon - but it will be very worthwhile!

Want The Full Details?

- Want a **definitive reference**?
  - Read the official *Pro Git* book, available for free online. While its discussion of Git is authoritative, some details about IDE integrations may be out of date.
  - Read alternative books, such as *Version Control with Git*.
  - Note that both of the above books give very little coverage of GitHub specifically - you should consult GitHub's own official documentation.
- On that note, the O'Reilly Learning Platform is available for free to CSU, offering thousands of computing-related books, videos, and on-demand courses.

Want A Quick Command-Line Reference?

- Want a **cheatsheet**?
  - Here's the official one.

Back to start