Introduction to Version Control and Project Management with Git and GitHub

Dmitri Svetlov

School of Biomedical Engineering, Colorado State University

Dmitri.Svetlov@ColoState.edu

2024 UQ-Bio Summer School

Agenda

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 (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**
  - An ability to undo 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.
- For brevity, each commit is referenced by the first seven characters of its checksum, *e.g.* `b4488c5`.
- 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.

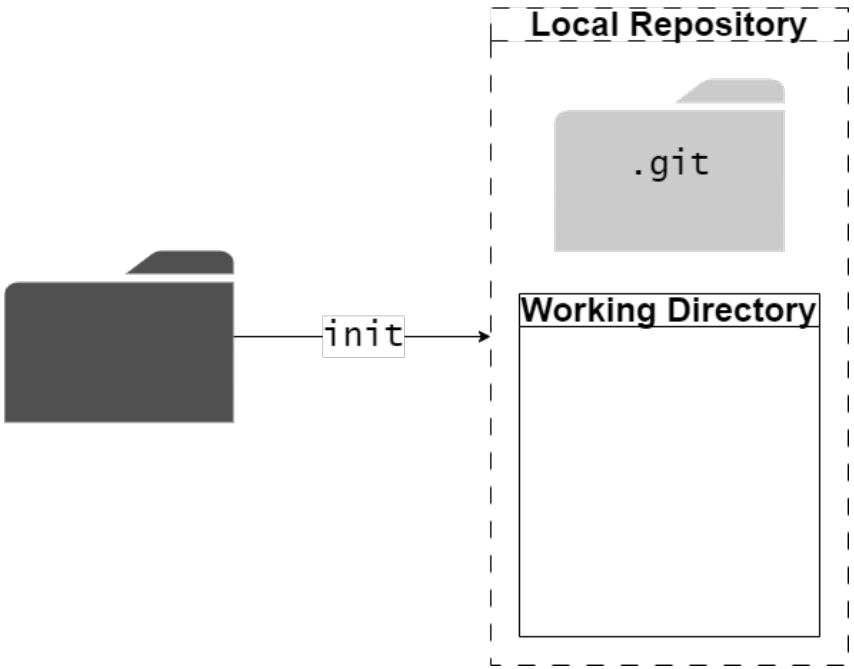
How Git Works: Theory II

- **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 but is now deprecated).
- 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

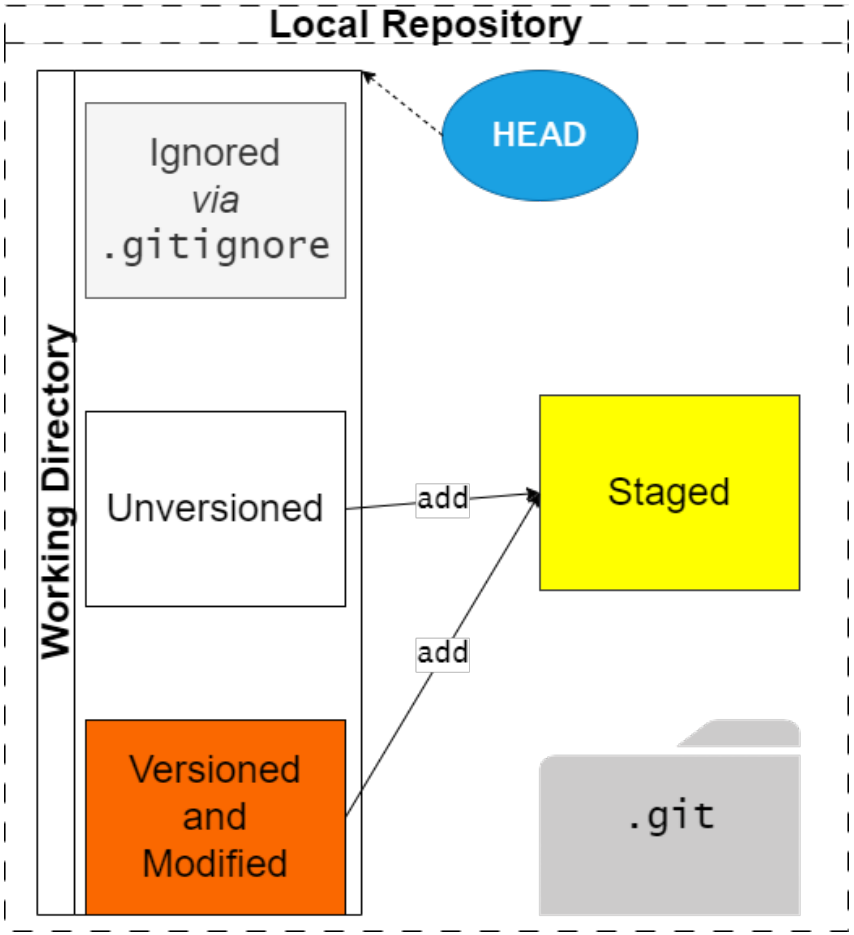
- To work with Git repositories, there are numerous tools:
  - The Git command-line program, `git`, which has sub-commands for all Git operations.
  - GUI tools, *e.g.* Git GUI and GitHub Desktop
  - Various IDEs integrate with Git, *e.g.* Visual Studio Code.
- Here, we will discuss the commands in the 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`.

How Git Works in Practice: Initialization



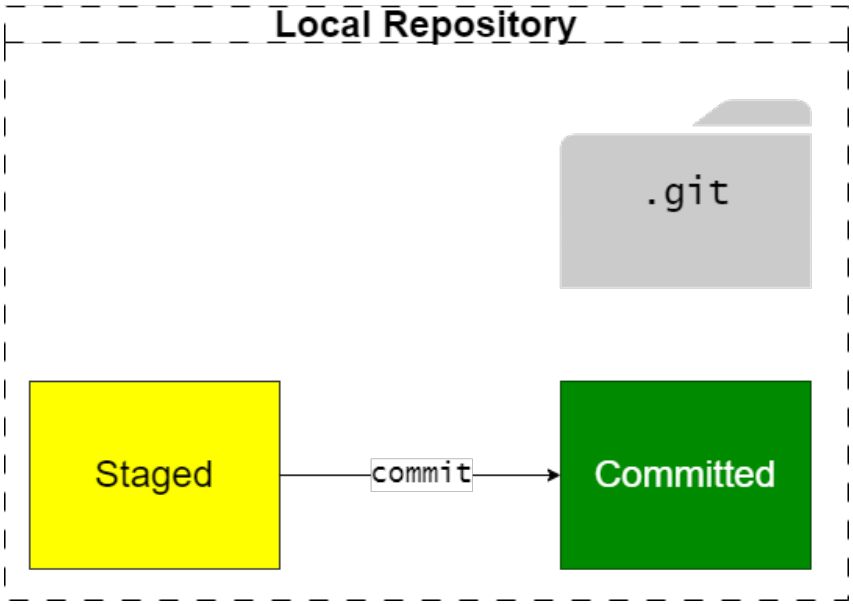
- One way is to call `init` on a directory on a local machine to convert that directory to a repository.
- The database itself (changesets, snapshots) is in a (new) subfolder called `.git`. **NEVER** modify this yourself!
- The repository uses a pointer called `HEAD` to track where the working directory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

How Git Works in Practice: Staging



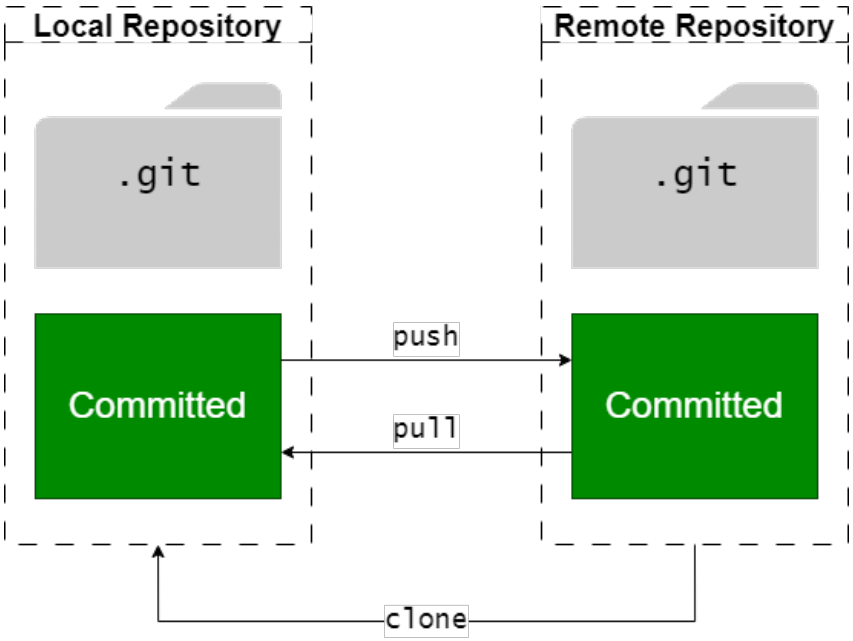
- **Staging** prepares files and folders for snapshotting, *i.e.* their preservation in a commit.
- Three kinds of files are relevant here:
  - Files with an extension listed in a special `.gitignore` file are ignored by Git by default.
  - Unversioned files are also ignored until you choose to version 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

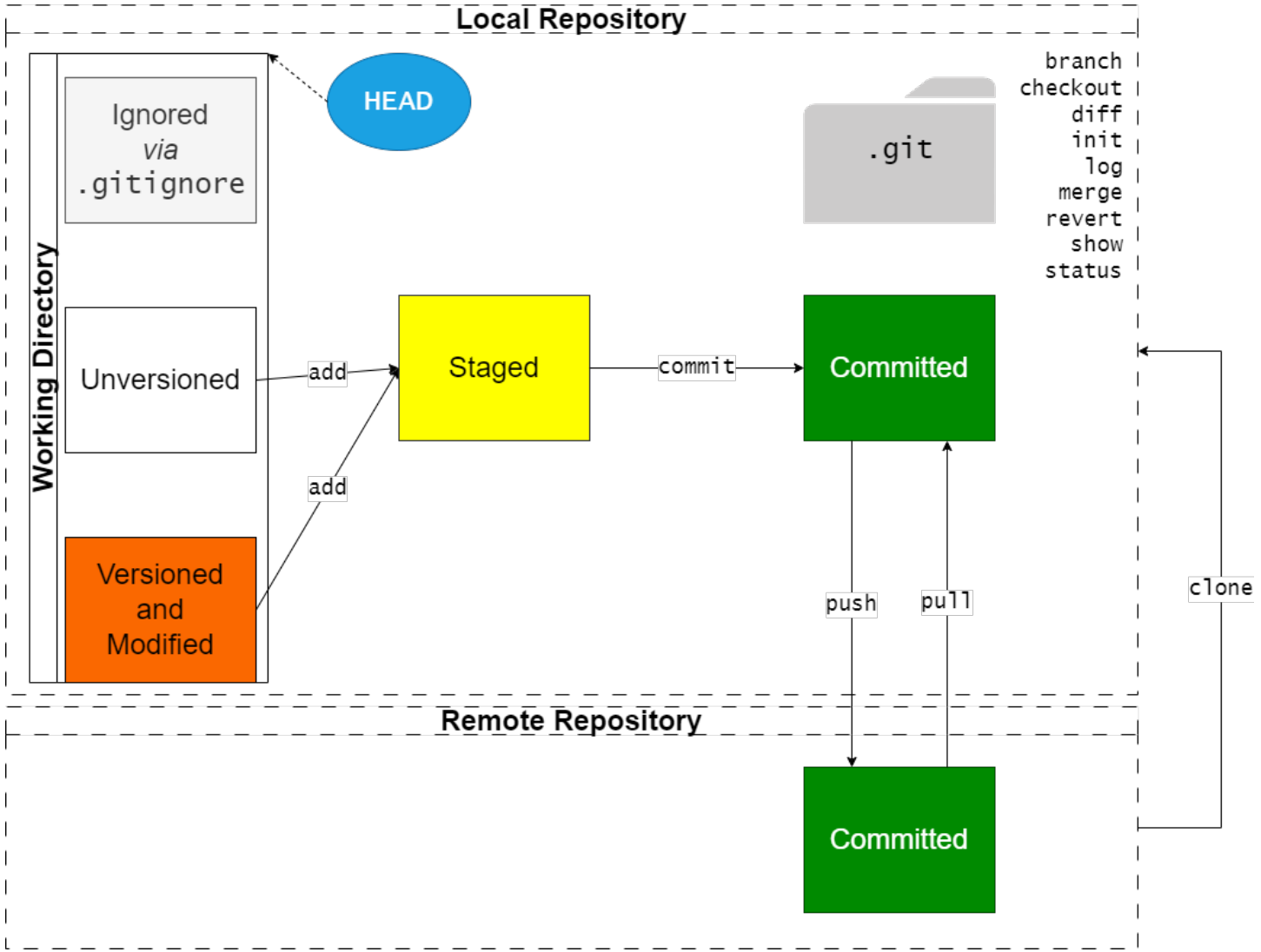


- Commits can be exchanged between (related!) local and remote repositories.
  - A **push** occurs when changes are sent from the local to the remote.
  - A **pull** occurs when changes are fetched from the remote and applied to the local.
- **Cloning** creates a local repository that is a complete copy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



Branching Model

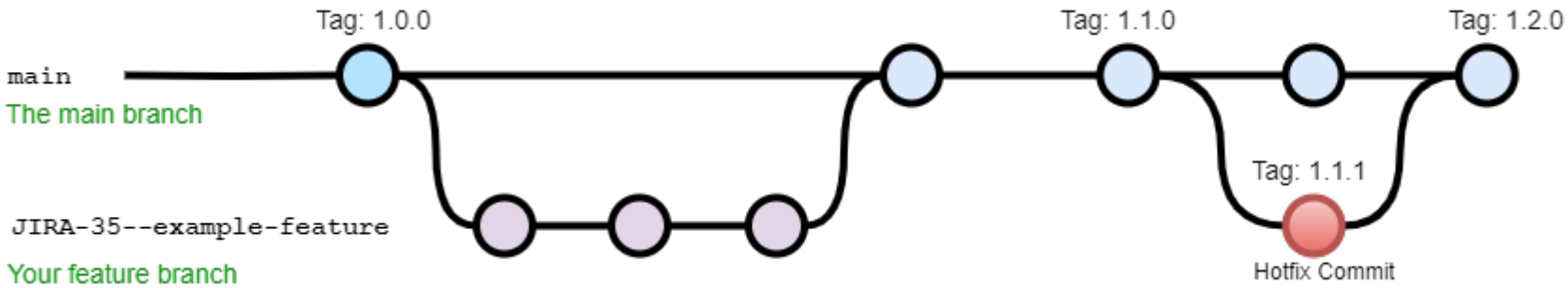
- Because of the intrinsic branching functionality of Git, it lends itself well to teams (and groups of teams) of developers.
- To be successful, any development team needs to adopt, and adhere to, a particular **branching model**:
  - What does each branch *mean*, philosophically and operationally?
  - Under what conditions can branches be created, deleted, and tagged?
  - Under what conditions can changes from one branch be merged into another?
  - How do the branches relate to other aspects of the team's/company's work, *e.g.* help-desk requests?

Naive Branching Models

- One approach is to simply have each developer own a "personal" branch, particularly if peers are working largely independently of one another and without much supervision. There are many downsides, especially at scale:
  - "Divergent evolution" to an extreme degree - merging across branches without conflicts will become prohibitively difficult.
  - The code will become highly idiosyncratic, with n developers adopting $\geq n$ ways of doing the same thing.
  - Someone ultimately has to call the shots - who will decide what is in the authoritative version of the code, and how?
- A better approach is to associate one branch with each intellectual change to the code, *i.e.* one bug fix or one new, standalone feature.
  - These tend to be called **feature branches** even when they are to fix a bug.

Trunk-and-Branch Model

- You can now see why `trunk` is often used for the name of the `main` branch - it is because of this model.
- This is very similar to what GitHub uses for their own development.
- A **tag** is what you'd expect: simply a human-readable reference to a particular commit. In this model, tags are used to identify versions for public ("production") release.
- See also the following webpages: [1] and [2]

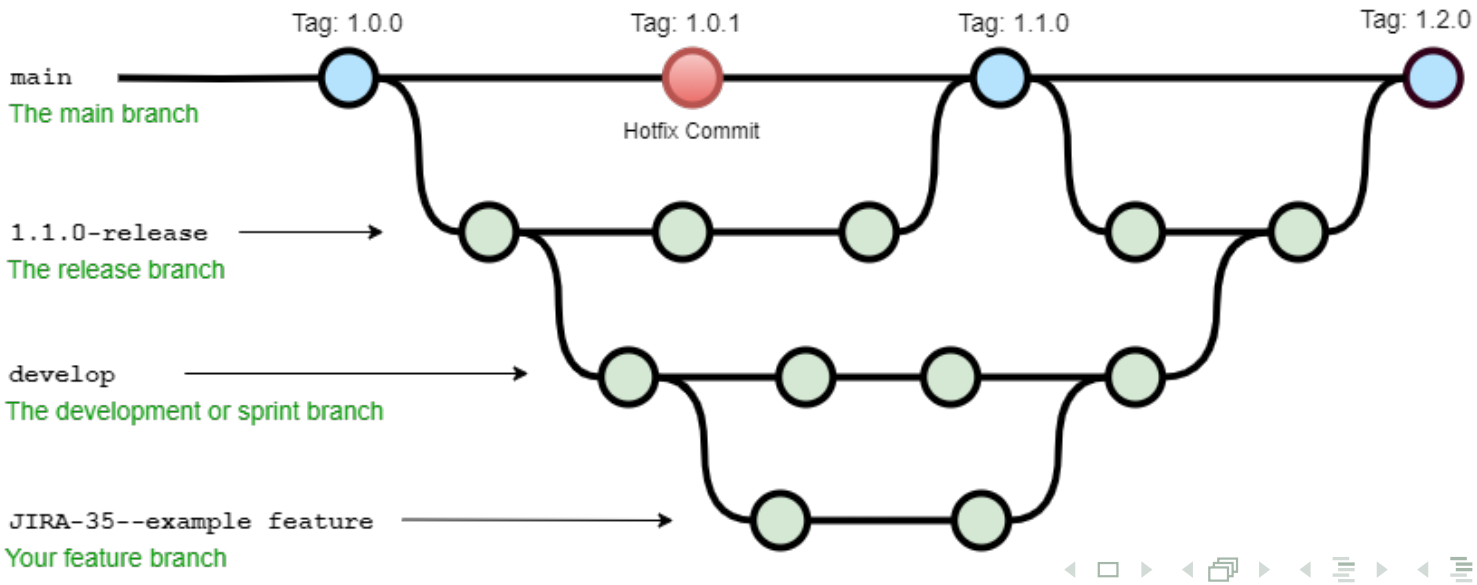


Toward More Complex Branching Models

- Several problems with having just one "mainline" branch:
  - `main` must **ALWAYS** be deployable, to whatever standard of correctness and reliability your code requires.
    - As a package grows in complexity and size (think millions of lines of code), this can become very difficult in practice.
  - You may need to simultaneously maintain, *i.e.* (customer) support, multiple versions that are very far apart historically.
  - You may have other constraints on information flow, the software development life cycle - *e.g.* for regulatory reasons, a public release might require validation that automated testing cannot handle.
- As a result, some workflows maintain multiple mainlines:
  - `stable`
  - `development`
  - `nightly`
  - `release`
  - *etc.*

A More Complex Gitflow

- This is for an Agile workflow, where releases are frequent (often, timed) and development accordingly occurs in "sprints"; therefore, the branches have different interrelationships.
- There are other workflows that work, but how well suited they are is very situation-dependent.
- Find one that works for you - and stick to it!



What is GitHub?

- Partly, GitHub is a hosting service for Git repositories (and associated webpages, README files, wikis, *etc*.
  - There are competitors, *e.g.* GitLab and Bitbucket.
- But GitHub also provides additional functionality that is very useful.
  - Project management tools, *e.g.* issue tracking.
  - Software and hardware for automated testing.
  - Workflow-related tools - most notably, the **pull request**.

Issue Tracking

- In project management, any task that must be done can be represented by an **issue**; in computing terms, think of an issue as the data structure encapsulating a task.
- Each issue has its own properties: assignee(s), milestone(s), priority, labels, related issue(s), dependencies, *etc*.
- Any project, in turn, can be represented by a set of constituent tasks/issues interrelated in logical and/or sequential ways.
- This concept generalizes: if you've ever emailed ETS, your help-desk "ticket" was an issue in ETS' instance of the FreshService ticketing system.
- Each ticket receives a unique, alphanumeric ID, *e.g.* `JIRA-35`.

Issue Tracking in Software Development and GitHub

- In software development, the best practice is that any new feature or bug fix will have a corresponding issue.
  - If the work is customer-originated, *e.g.* a customer reported the bug, the fixing issue will be "linked" to the help-desk issue (even if that's in a different system) so that the customer can be notified when the fix is made.
  - In some workflows, a feature branch's name will contain the issue ID (recall the mentions of `JIRA-35` in the diagrams above).
- Issues are an integral feature of GitHub repositories:
  - All of the fundamental issue-tracking functionality is provided, plus
  - Each issue has a webpage, where the comments can reference contributors, branch(es), and commits for the issue.
  - You can also reference - and even resolve! - issues in commit messages. If those commits are initially made to a local repository, the issues will get updated when you `push`.

Automated Testing in Software Development

- Testing, particularly automated testing, is an integral and indispensable best practice of software development.
- **Test-driven development** (TDD) actually makes tests primary, in a sense:
  - In this philosophy, a software package is defined not by its code but the functionality it implements.
  - Tests are the means by which functionality can be empirically, reliably verified.
  - Therefore, you write the test(s) for a new feature first, then the code to implement the feature.
  - Upon any changes to the code, and on a regular schedule, *e.g.* nightly, all tests associated with a package are run, to ensure that the package is functioning as expected.

Continuous Integration and Deployment (CI/CD)

- The practice of frequently (re)integrating code changes from distributed repositories to a shared one, and then automatically testing the result, is called **continuous integration** (CI).
- Analogously, **continuous deployment** (CD) refers to the automated (re)deployment of (new) versions of software upon the success of the CI checks.
- These practices are not universal, *e.g.* highly regulated code will probably not use CD, because of the validation procedures required.
- Nonetheless, any move toward greater CI/CD catches errors soon, speeds up and eases debugging, and minimizes merge conflicts.

CI/CD in GitHub

- GitHub provides a wealth of CI/CD functionality, much of which is available for free for public (*i.e.* open source) repositories.
- To any repository, workflows of GitHub **Actions** can be added, which are then executed by **runners** in response to **events**.
- That's pretty vague, so here's an example:
  - Event: A push is made to a repository containing Python code and utilizing some third-party packages.
  - Actions: Install a given version of Python and the third-party packages on a machine. Execute all automated tests within the repository. Email the results to the committer of the push.
  - Runner: GitHub provides its own runners for the three major desktop OSes. (Additionally, you can "self-host" a runner wherever you choose.)

Pull Requests

- Think back to our discussion of pushes and pulls.
- There are two common situations where you would like to push but cannot or should not:
  - You don't have write access, *e.g.* you want to make a contribution to an open-source repository but are not a full member.
  - You are reintegrating a feature branch, *i.e.* seeking to commit to `main`.
- Since you cannot *push*, you can only request that someone on the other end perform a *pull* (the same activity, remember, but the perspective is reversed).
- GitHub invented a formalization of this process: the **pull request**.

A Word about Forks

- In the first scenario above (lack of membership in a repository), the common solution is the following:
  1. Create (on GitHub) a **fork** of the repository. This will clone a specified branch of the repository and all upstream branches (up to and including `main`).
  2. Make changes on a branch of the forked repository (which you will own).
  3. When satisfied, create a pull request from the branch of your repository to the desired upstream branch of the upstream repository.
- In the second scenario, you should not create a fork. The pull request is created in the same way, but entirely within the one repository.

Anatomy of a Pull Request

- A pull request (PR) will contain some or all of the following, with its own GitHub webpage:
  - A discussion.
  - A list of all included commits and `diff` of all changes - this allows for integrated code review.
  - The results of any CI checks that were run as part of the PR. In particular, if this included pass/fail tests, their success or failure will be clearly visually indicated.
  - A resolution.
    - Using certain keywords, *e.g.* "close" and "resolve", you can simultaneously approve the PR, merge the feature branch into the upstream branch, delete the feature branch, and successfully resolve the associated issue!
- The best practice is that any merge into `main` will only occur as the result of an approved PR, and GitHub will even enforce this requirement if you choose.

What Now?

- Are you already on GitHub?
  - GitHub Skills offers mini-courses on various topics:
    - GitHub Actions
    - Workflows
    - Markdown
    - Merge conflicts and how to resolve them
    - And more...
- If not...

What Now? (continued)

- Read and work through Tsitoara's book [3]. This will walk you through:
  - Installing Git on your local machine, with OS-specific instructions
  - Creating an account on GitHub
  - 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 it in a (long) afternoon - but it will be very worthwhile!
- There are other tutorials - and reference books - but this one will be the most accessible. Indeed, this talk's structure largely follows that of the tutorial.

What Then?

- Practice, practice, practice!
- Words of advice:
  - Don't invent a non-standard workflow unless none of the tried-and-true ones works for you.
  - Team members should all use the same workflow.
  - Always understand where you are and what you're trying to accomplish with your commit/push/pull request.
    - If you must, draw a workflow diagram like the ones above.
    - Graphical tools, *e.g.* Git GUI, will show you where you actually are, so compare that to where you think you are.
  - Do only one thing per commit/push/pull request (think incremental and atomic).
  - **DON'T REWRITE HISTORY!**
    - One way to minimize conflicts is to frequently merge from `main` into all open feature branches. Keep them only ahead of `main`, not behind.
- Good luck!

Parting Thoughts

- Version control isn't just for software!
  - Drafts of papers, slide decks, *etc.*
  - Experimental protocols
  - Analyses and procedures for generating figures
- And neither is project management!
  - The same principles are broadly applicable, including to scientific research.
  - Many of the (general-purpose) tools are also adaptable.

◂ Back to start

References

[1] Juan Benet. *a simple git branching model (written in 2013)*. URL: https://gist.github.com/jbenet/ee6c9ac48068889b0912. (accessed: January 11, 2024).

[2] End of Line Blog. *OneFlow: a Git branching model and workflow*. URL: https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow.

[3] Mariot Tsitoara. *Beginning Git and GitHub: A Comprehensive Guide to Version Control, Project Management, and Teamwork for the New Developer*. eng. 1st ed. Berkeley, CA: Apress L. P, 2019. ISBN: 1484253124.