Version Control

Maintaining a software project is challenging, especially when multiple developers are working on the same codebase. When introducing new features, maintainers may face the following issues:

- Multiple developers modifying the same file simultaneously, making it difficult to merge changes.
- New code breaking existing features, affecting downstream users.

The solution to these problems is **version control**. Among all version control software, **git** is the most popular.

Git - a distributed version control system

Version control is a system that records changes to files over time, allowing you to track modifications, compare changes, and revert to previous versions if needed. It is a critical component of modern software development, especially when a project has multiple developers. It prevents unwanted overwrites and conflicts.

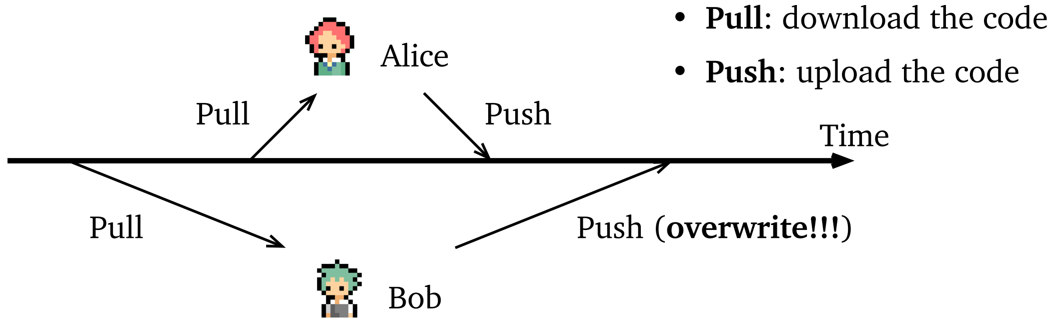


Figure 1: Without version control, may overwrite 's work due to conflicts.

In the early days, *Centralized version control systems* (e.g. SVN) is the main way to manage version control. All changes are made to a central repository. **Locking** the repository is a common practice to prevent conflicts. Whenever someone wants to make changes to the repository, they need to lock some part of the repository, which blocks other developers from making changes to the same part of the repository. Just like the figure Figure 2 shows, some long lasting work may harm the productivity of the whole team.

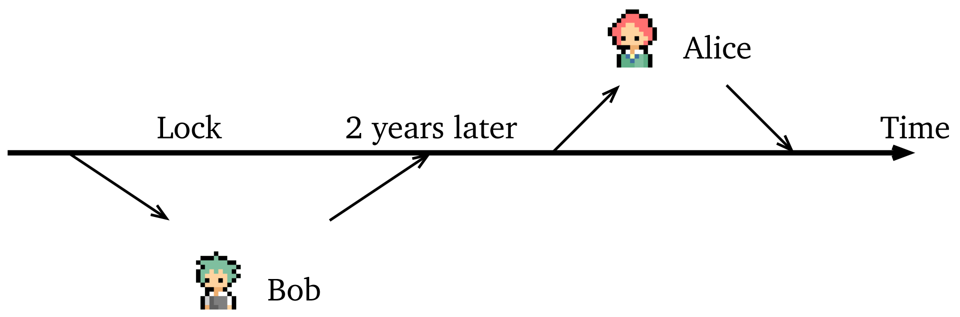


Figure 2: With centralized version control, has to wait to complete the work.

Can we do version control without a lock? *Git* is an open source distributed version control system that proposed to solve this issue. It was initially developed by Linus Torvalds (yes, the same guy who developed Linux) in 2005, and now it is the most popular version control system.

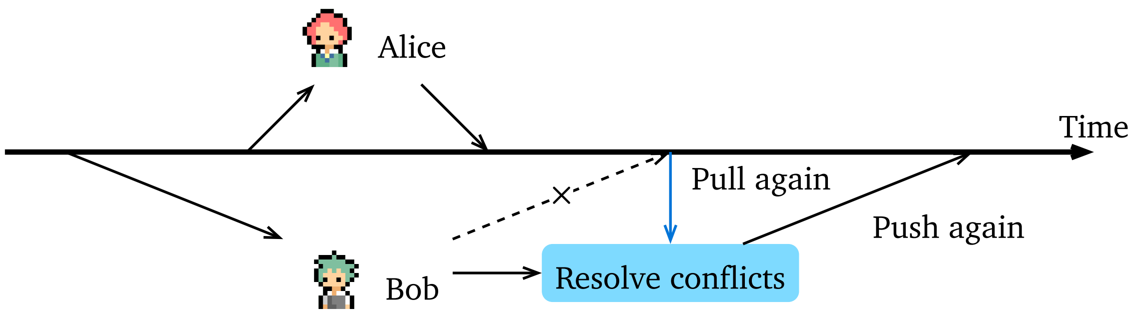


Figure 3: With Git, and collaborate peacefully.

As shown in Figure 3. A project managed by Git, or a *repository*, may have multiple *branches*, each being a "copy" of the project. The *main* branch (the thick line) is the default branch that accessible to users. Developers can freely derive one branch from another, and make changes to the their own branches without blocking others, however, updating the main branch needs to be done carefully, since it will affect all users and developers. Consider the scenario where both Alice and Bob want to make changes to the main branches. Each of them *checkout* a new branch from the main branch, which effectively "copies" the main branch (not a hard copy).

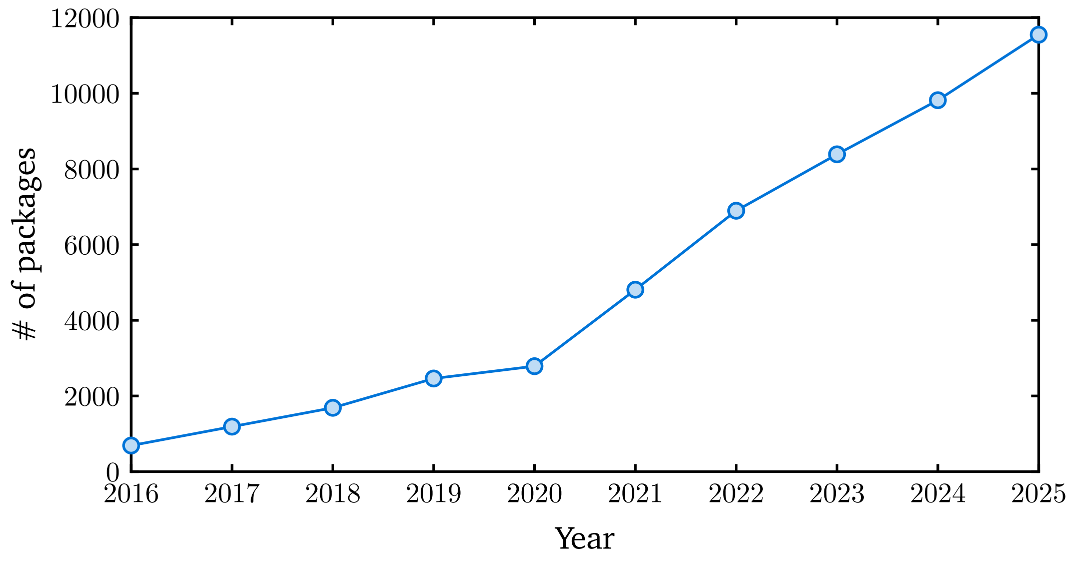
Alice and Bob do not know the existence of each other's branches, inevitably, they will make changes to the same file. When Bob tries to push his changes, he will be notified about a previous commit by Alice, and his version is out of date. To resolve the conflict, Bob pulls the latest changes from Alice, and resolve the conflict locally. After that, Bob can push his changes to the repository.

You may wonder: Where do they store their code if they do not know each other? How can they verify the correctness of their code? e.g. what if Bob just delete all the changes made by Alice? These questions are related to the Git hosting services, *pull requests*, the automated *unit tests*, and *continuous integration/continuous deployment* (CI/CD) that we will discuss later.

Git Hosting Services

To collaborate effectively using Git, you need a server to store your Git repository, known as a **remote**. These remote repositories can be hosted on platforms like GitHub and GitLab. GitHub is the most popular platform for hosting and collaborating on open-source projects. While GitHub is like a "centralized" hub, GitLab is more like a piece of software that you can deploy on your own server.

Many famous projects are hosted on GitHub, including machine learning framework PyTorch and Jax. The Julia community has a tradition of hosting their packages on GitHub as well. Actually, all Julia packages are hosted on GitHub! They are registered on GitHub repository: JuliaRegistries/General. By the time of writing, there are more than 10,000 packages registered in the Julia registry!



Collaborating with others - issues and pull requests

When collaborating with others, you may want to propose changes to a repository and discuss them with others. This is where **issues** and **pull requests** come in. Issues and pull requests are features of git hosting services like GitHub and GitLab.

- **Issue** is relatively simple, it is a way to report a bug or request a feature.
- **Pull request** (resource: how to create a pull request) is a way to propose changes to a repository and discuss them with others. It is also a way to merge code from source branch to target branch. The source branch can be a branch in the same repository or a branch in a **forked repository** - a copy of the repository in your account. Forking a repository is needed when you want to propose changes to a repository that you do not have write access to.

To update the main branch, one should use pull requests as much as possible, even if you have write access to the repository. It is a good practice to discuss the changes with others before merging them to the main branch. A pull request also makes the changes more traceable, which is useful when you want to revert the changes.

A Minimum Introduction to Git

Install git

In Ubuntu (or WSL), you can install git with the following command:

```
sudo apt-get install git
```

In MacOS, you can use Homebrew to install git:

```
brew install git
```

Then you need to configure your git with your name and email:

```
git config --global user.name "Your Name"
git config --global user.email "xxx@example.com"
```

Create a git repository

A git repository, or repo, is a directory managed by git. To create one, open a terminal and type

```
cd path/to/working/directory
git init
echo "Hello, World" > README.md
git add -A
git commit -m 'this is my initial commit'
git status
```

- Line 1: changes the directory to the specified working directory, which can be an existing or a new directory.
- Line 2: initializes the working directory as a git repository, creating a `.git` directory that contains all data managed by git.
- Line 3: creates a file named `README.md` with the content `Hello, World`. This file is a **markdown** file, a lightweight markup language with plain-text-formatting syntax. More information about markdown can be found in the markdown tutorial. This step can be skipped if the working directory already contains files.
- Line 4: adds files to the **staging area**, which temporarily stores changes to be committed.
- Line 5: commits the changes to the repository, creating a **snapshot** of your current work.
- Line 6: displays the status of the working directory, staging area, and repository. If the previous commands were executed correctly, the output should be `nothing to commit, working tree clean`.

Track the changes - checkout, diff, log

Git enables developers to track changes in their codebase. Continuing the previous example, we can analyze the repository with the following commands:

```
echo "Bye Bye, World" > README.md
git diff
git add -A
git commit -m 'a second commit'
git log
git checkout HEAD~1
git checkout main
```

- Line 1: modifies the `README.md` file.
- Line 2: displays the changes made to `README.md`.
- Line 3-4: stages the changes and commits them to the repository.
- Line 5: displays the commit history. The output should look like this:

```
commit 02cd535b6d78fca1713784c61eec86e67ce9010c (HEAD -> main)
Author: GiggleLiu <cacate0129@gmail.com>
Date:   Mon Feb 5 14:34:20 2024 +0800

    a second commit

commit 570e390759617a7021b0e069a3fbe612841b3e50
Author: GiggleLiu <cacate0129@gmail.com>
Date:   Mon Feb 5 14:23:41 2024 +0800

    this is my initial commit
```

- Line 6: Check out the previous snapshot. Note that `HEAD` represents your current snapshot, and `HEAD~n` refers to the `n`th snapshot before the current one.
- Line 7: Switch back to the `main` **branch**, which points to the latest snapshot. We will discuss more about **branches** later in this tutorial.

You can use `git reset` to reset the current HEAD to the specified snapshot, which can be useful when you committed something bad by accident.

Upload your repository to the cloud - remote

To collaborate effectively using Git, you need a server to store your Git repository, known as a **remote**. Begin by creating an empty repository (without README files) on a Git hosting service. You can follow this tutorial on creating a new GitHub repository. Once created, a URL for cloning the repository will be provided, typically using the `SSH` or `HTTPS` protocol. To ensure your repository's security, configure necessary security settings such as `SSH` or two-factor authentication (2FA).

After setting up your repository, you can upload your local repository to the remote by using the following commands:

```
git remote add origin <url>
git remote -v
git push origin main
```

Add a Remote Repository

1. Add a remote repository using the command `git remote add origin <url>`. Here, `origin` serves as a label for the remote repository, and `<url>` is its web address.
2. Display all remote URLs with `git remote -v`, which includes the newly added `origin`.
3. Push commits to the `main` branch of the `origin` remote using `git push origin main`. If a collaborator has pushed changes before you, this command might fail. To resolve this, execute

`git pull origin main` to fetch the latest updates and manually merge any conflicting commits. For more details, refer to the Git Branching and Merging guide.

Develop Features Safely with Branches

Working solely on the `main` branch is not advisable. A branch in Git is a lightweight pointer to a specific commit. Here are some reasons why relying on a single branch can be problematic:

- **Lack of Usable Code:** The `main` branch should always be stable and usable, as developers build features from it. Working on a single branch can disrupt this stability.
- **Conflict Resolution Challenges:** When multiple developers edit the same file simultaneously, conflicts can arise, making synchronization difficult. Using multiple branches allows for more independent feature development.
- **Difficulty in Discarding Features:** Experimental features might need to be discarded after testing. Reverting a commit on the `main` branch is not straightforward.

Understanding branches is crucial when multiple developers collaborate on a project. In the following example, we will create a new branch named `me/feature` to develop a feature independently.

```
git checkout -b me/feature
echo "Hello, World - Version 2" > README.md
git add -A
git commit -m 'this is my feature'
git push origin me/feature
```

- Line 1: create and switch to the new branch `me/feature`, which is a copy of the current branch, e.g. the main branch. The branch name `me/feature` follows the convention `<username>/<feature>`, which is useful when working with others.
- Line 2-5: makes some changes to the file `README.md` and commits the changes to the repository. Finally, the changes are pushed to the remote repository `origin`. The remote branch `me/feature` is created automatically.

While developing a feature, you or another developer may want to develop another feature based on the current `main` branch. You can create another branch `other/feature` and develop the feature there.

```
git checkout main
git checkout -b other/feature
echo "Bye Bye, World - Version 2" > feature.md
git add -A
git commit -m 'this is another feature'
git push origin other/feature
```

In the above example, we created a new branch `other/feature` based on the `main` branch, and made some changes to the file `feature.md`.

Finally, when the feature is ready, you can merge the feature branch to the main branch.

```
git checkout main
git merge me/feature
git push origin main
```

Git cheat sheet

It is not possible to cover all the feature of git. We will list a few useful commands and resources for git learning.

```
# global config
git config  # Get and set repository or global options

# initialize a repo
git init    # Create an empty Git repo or reinitialize an existing one
git clone   # Clone repository into new directory

# info
git status  # Show the working tree status
git log     # Show commit logs
git diff    # Show changes between commits, commit and working tree, etc

# work on a branch
git add     # Add file contents to the index
git rm      # Remove files from the working tree and from the index
git commit  # Record changes to the repository
git reset   # Reset current HEAD to the specified state

# branch manipulation
git checkout # Switch branches or restore working tree files
git branch  # List, create, or delete branches
git merge   # Join two or more development histories together

# remote synchronization
git remote  # Manage set of tracked repositories
git pull  # Fetch from and integrate with another repo or a local branch
git fetch   # Download objects and refs from another repository
git push    # Update remote refs along with associated objects
```

Correctness - Unit Tests and CI/CD

In terms of scientific computing, accuracy of your result is most certainly more important than anything else. Checking the correctness of contributed code is definitely one of the most challenging tasks in the open-source community. As checking the code line by line and test over all the edge cases is too expensive and time-consuming, clever software engineers have come up with a more efficient way to check the correctness of the code, which is to use **Unit Tests** and **CI/CD**.

Unit Test

Unit Tests is a software a testing method for the smallest testable **unit** of an application, e.g. functions. Unit tests are composed of a series of individual test cases, each of which is composed of:

- a collection of inputs and expected outputs for a function.
- an **assertion** statement to verify the function returns the expected output for a given input.

For example, in Julia language, we can test the `sin` function with the `@test` macro:

```
julia> using Test  # import the Test package
julia> @test sin(π/2) ≈ 1.0  # test if sin(π/2) is equal to 1.0 up to machine precision
Test Passed
```

Here, the input is `π/2` and the expected output is `1.0`. The `@test` macro will check if the followed statement is true. If it is, the test will pass. Otherwise, the test will fail.

In practice, unit tests will be written in a separate file, e.g. `test/runtests.jl` in a Julia package. They will be run automatically with the `]test` command in the Julia REPL. The more test cases, the more **robust** the code is. As a metric, **test coverage** is often used to measure the quality of the unit tests. It is the percentage of the code that is covered by tests, i.e. the higher the coverage, the more robust the code is.

Automate your workflow - CI/CD

Even with unit tests, you still need to set up three clean machines to run the tests. Is there a convenient way to run the tests on the cloud? The key to solving this problem is to automate the workflow on the cloud with the containerization technology, e.g. Docker. You do not need to configure the dockers on the cloud manually. Instead, you can use a Continuous Integration/ Continuous Deployment (CI/CD) service to automate the workflow of

- (CI) **build**, **test** and **merge** the code changes whenever a developer commits code to the repository.
- (CD) **deploy** the code or documentation to a cloud service and **register** the package to the package registry.

CI/CD are often provided by git hosting services, e.g. Github Actions. When a developer commits code to the repository, or create a pull request, the GitHub action will detect the change and trigger a CI/CD pipeline. The pipeline will initialize a virtual machine on the cloud, install the dependencies, run the tests, and report the test results. To configure the CI/CD pipeline, you need to create a configuration file, e.g. `.github/workflows/ci.yml`. We will learn how to set up a CI/CD pipeline in the section My First Package.

Resources

- The Official GitHub Training Manual
- MIT online course missing semester.