Guide to GitHub

Using GitHub for Group Projects

Purpose

This supplementary guide covers the main skills you'll need to use GitHub for group projects. You'll learn to:

- Invite team members to collaborate.
- Use GitHub Flow.
- Merge conflicts.
- Use Git Large File Storage (LFS).

Prepare

Before continuing, prepare with the following tasks:

- Create a project repository, or be invited to join it as a collaborator.
- Clone the project repository.
- Become familiar with your command-line interface (Terminal for Mac users and Command Prompt for Windows users).
- Understand and install Git LFS *before* adding any files larger than 100MB.

What Is GitHub?

More than 40 million people use GitHub. That is higher than most countries' populations on the planet! Almost every developer working individually or on a group project uses GitHub. No wonder it is so popular. Not only is it recognized as an essential tool for best code practices, but it is the biggest platform for developers to showcase their work. Using GitHub makes it easier to collaborate with colleagues and peers and to review previous versions of one's work.

GitHub is an essential tool for your academic development, and it will transition with you through your professional career. Think of your knowledge blueprint—it will raise your visibility among potential employers and could translate into a high-paying job. As you'll discover, there are many advantages to mastering GitHub—too numerous to list here. Take advantage of this project's opportunity to explore GitHub fully, and learn how to leverage that knowledge to your advantage, academically and professionally. Above all, have fun!

Invite Team Members to Collaborate

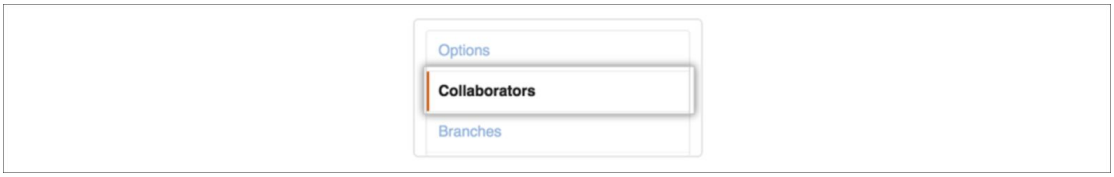
After you create your project repository, invite your team members to join it to collaborate on the project. Collaborators will be given read/write access to your single repository. Let's start the process.

1. Go to the main page of your repository.

2. On the top right, click the "Settings" tab.

Alt-Text: The "Settings" tab is on the top right of your GitHub repository page.

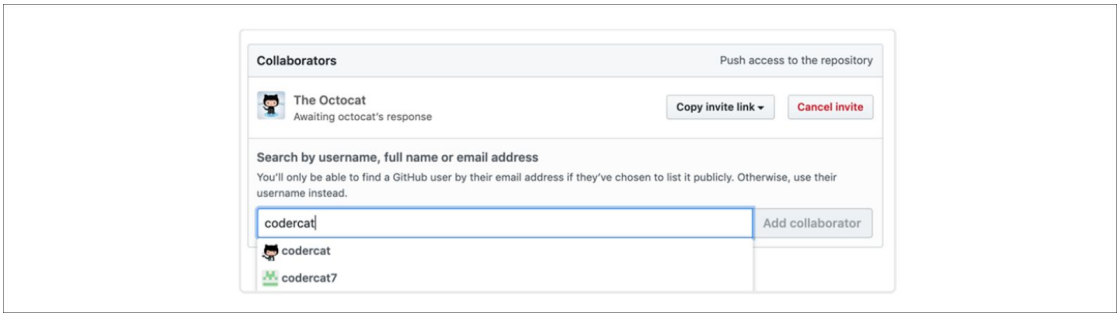
3. In the left sidebar, click "Collaborators."



Alt-Text: Click "Collaborators" in the left sidebar.

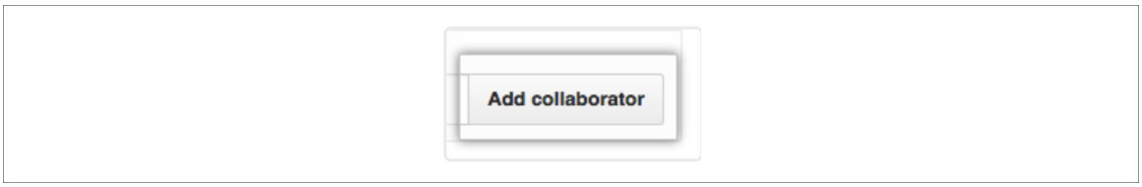
4. Under "Collaborators," type the collaborator's username in the search field.

5. Select the collaborator's username from the drop-down menu.



Alt-Text: Use the search field to locate the collaborator's username.

6. Click the "Add collaborator" button.



Alt-Text: Click the “Add collaborator” button.

7. Your teammates will receive an email invitation to join the repository. Once they accept the invitation, they’ll have collaborator access to the project repository you created.

Use GitHub Flow

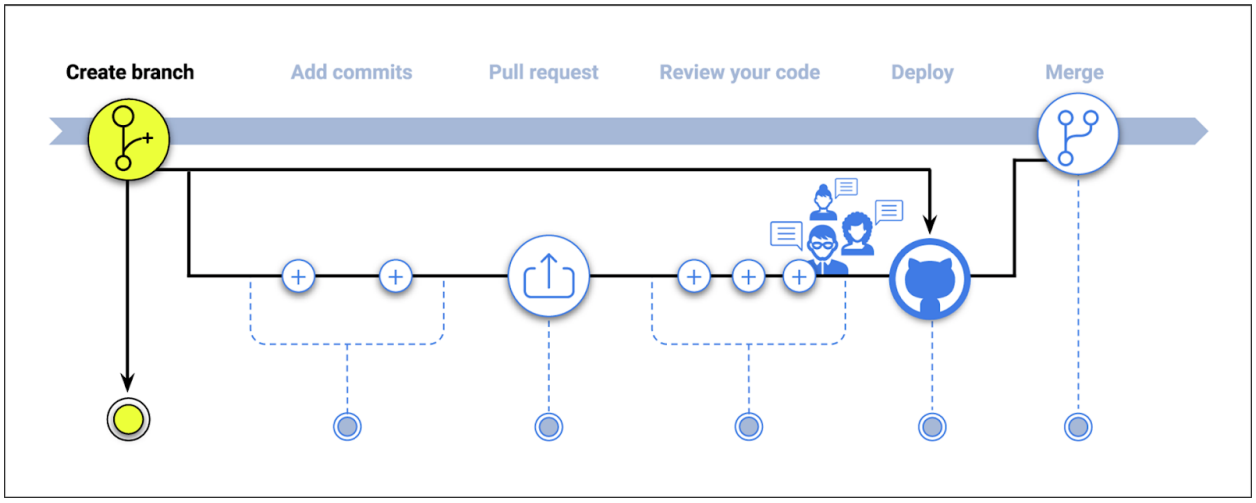
GitHub flow is a lightweight, branch-based workflow designed for teams and projects with regular deployments. This section explains how GitHub flow works and provides important core knowledge about GitHub's underlying concepts. Understanding workflow is essential as you'll apply this throughout your developer career.

GitHub flow is a six-step process:

1. Create a branch
2. Add commits
3. Pull requests
4. Review your code
5. Deploy for final testing
6. Merge your code

Step 1: Create a Branch

During your project, you'll have a few distinct ideas in progress that may or may not be ready to go. Branching was created to manage this workflow. When you add a branch in your project, you're creating an environment for trying out new ideas. Changes you make on a branch won't affect the main branch, so you're free to experiment and commit changes, assured that your branch won't be merged until it is ready for a collaborator's review.



Alt-Text: GitHub flow's first step is to create a branch.

There are a few ways you can create new branches in Git. The most common follows:

```
git checkout -b <branch-name>
```

The above method is used often because it will create a branch from your current branch and will switch you to that branch in a single command line.

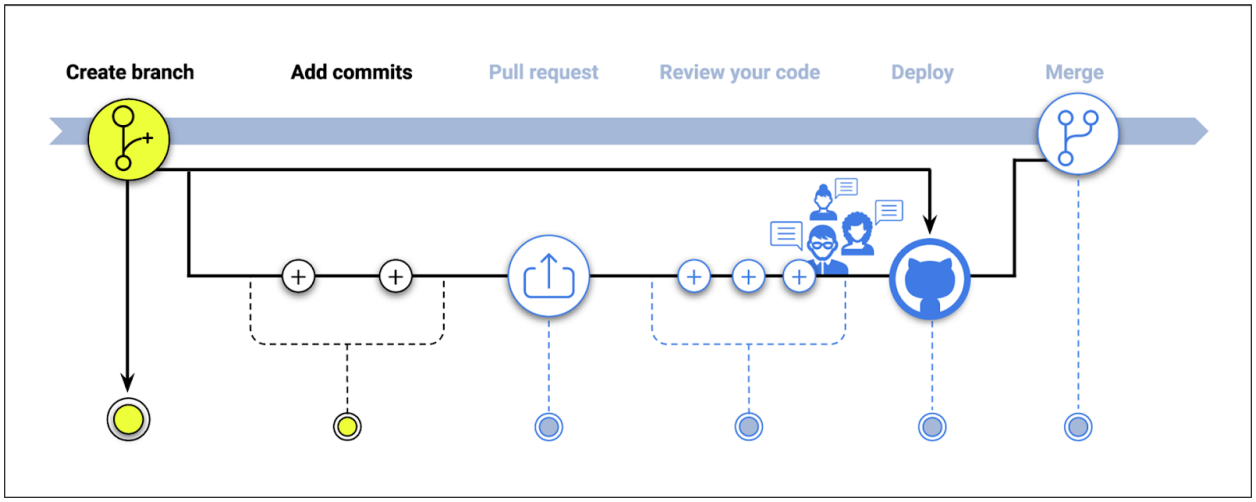
Another common option is to use the `branch` command directly. Note that this command will not automatically switch over to the newly created branch. However, if you do wish to switch, you can use the `checkout` command.

In the code below, there are two commands. With the first, a branch is created. The second command will “checkout” the branch, and switch you to it.

```
git branch <branch-name>
git checkout <branch-name>
```

Step 2: Add Commits

Once your branch has been created, you can start making changes. Whenever you add, edit, or delete a file, you’re making a commit and adding them to your branch. This process of adding commits keeps track of your progress as you work on a feature branch. Commits also create a transparent history of your work that others can follow to understand what you’ve done and why. Each commit has an associated commit message, which is a description explaining the reason a particular action was made. Moreover, each commit is considered a separate unit of change. This lets you roll back changes if a bug is found, or if you decide to head in a different direction.



Alt-Text: GitHub flow’s second step is to add commits to your branch.

```
git status
git add .
git commit -m 'first commit'
```

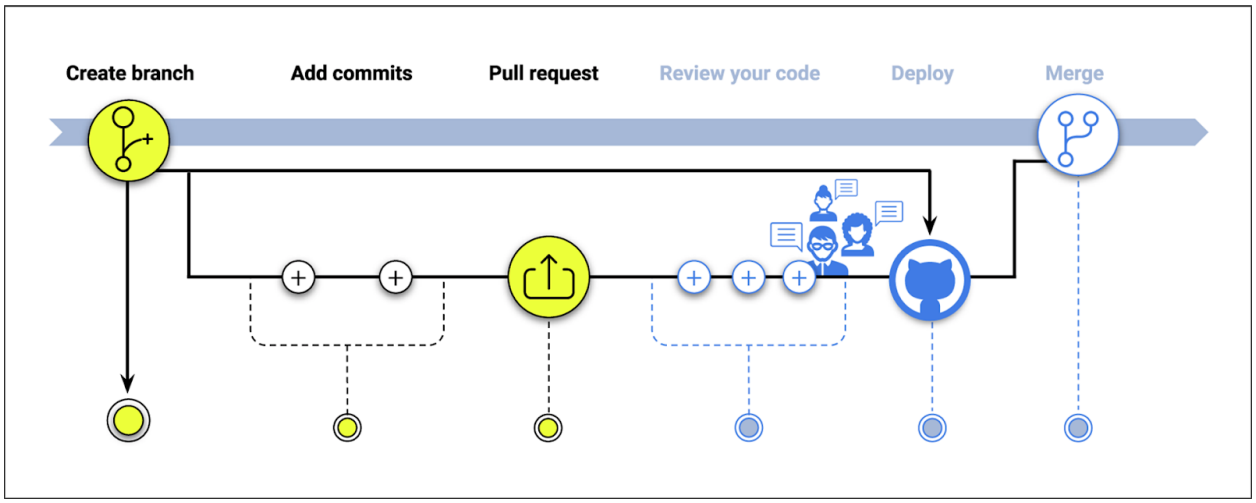
Please note, the `git add .` command will add *all* paths to the staged changes if they're either changed or are new and not ignored. To fully grasp the `add` command, take a few minutes to read the documentation.

Once you've committed your changes, the natural next step is to push our branch. Remember, do commit and push early and often—creating periodic checkpoints helps you understand how you broke something.

```
git push origin <branch-name>
```

Step 3: Pull Requests

Pull requests initiate discussion about your commits. Since they're tightly integrated with the underlying Git repository, anyone can see exactly what changes would be merged if they accept your request. Pull requests can be opened at any point during the development process: when you have little or no code but want to share some screenshots or general ideas, when you're stuck and need help or advice, or when you're ready for someone to review your work. Also, pull requests are useful for contributing to open source projects and for managing changes to shared repositories. If you're using a Fork & Pull Model, pull requests provide a way to notify project maintainers about the changes you'd like them to consider. If you're using a Shared Repository Model, pull requests help start a code review and conversation about proposed changes before they're merged into the main branch.

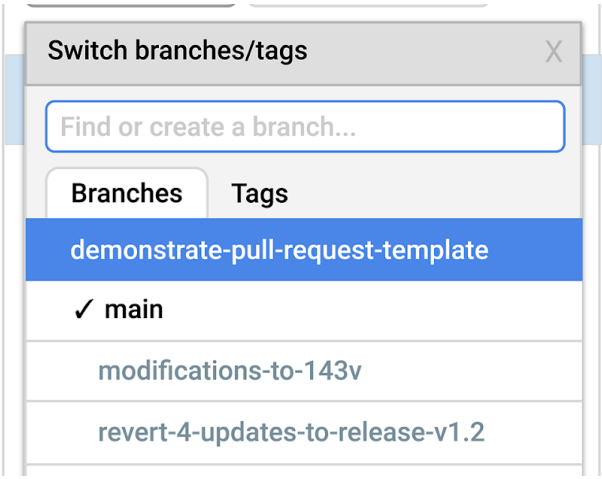


Alt-Text: GitHub flow's third step is to create a pull request.

To create a pull request, make sure you've already added, committed, and pushed your code to the main repository, and follow the instructions below:

1. On GitHub (webpage), navigate to the main page of the repository.

2. In the "Branch" drop-down menu, choose the branch that contains your commit.

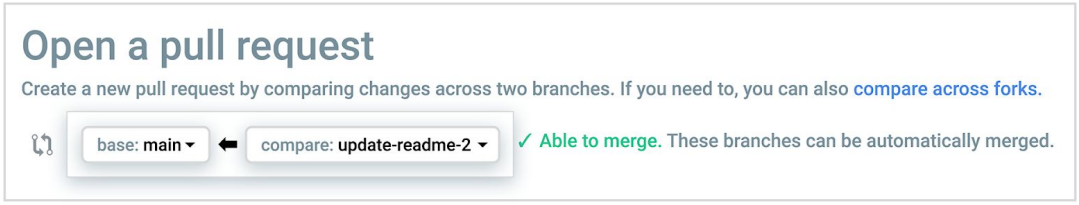


Alt-Text: In the “Branch” menu, choose the branch that contains your commit.

3. To the right of the “Branch” drop-down menu, click the “New pull request” button.

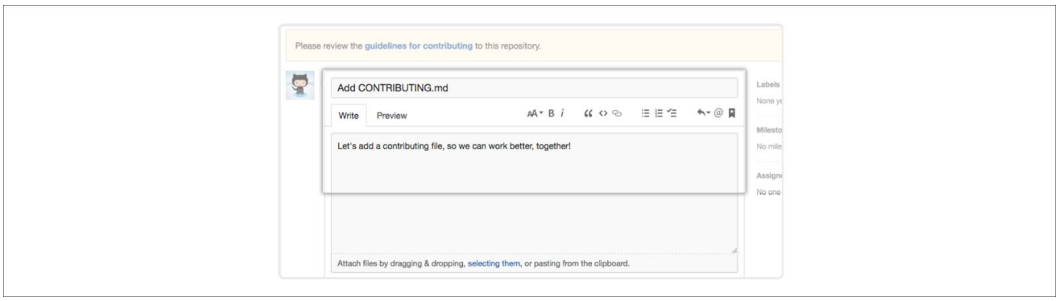
Alt-Text: Click the “New pull request” button.

4. In the “Base” branch drop-down menu, select the branch to merge your changes into. In the “Compare” branch drop-down menu, choose the topic branch you made your changes in.



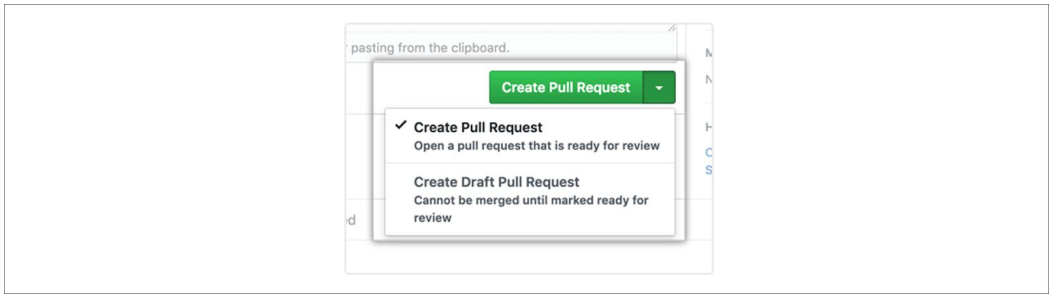
Alt-Text: In the “Base” drop-down menu, on the left, select the branch. In the “Compare” branch drop-down menu, on the right, select the topic branch.

5. Type a title and description for your pull request.



Alt-Text: Type a pull request title and description into the text box.

6. The drop-down menu offers two options: Click “Create Pull Request” to open a pull request ready for review, or click “Create Draft Pull Request” to create a draft. For more on draft pull requests, see “About pull requests.”

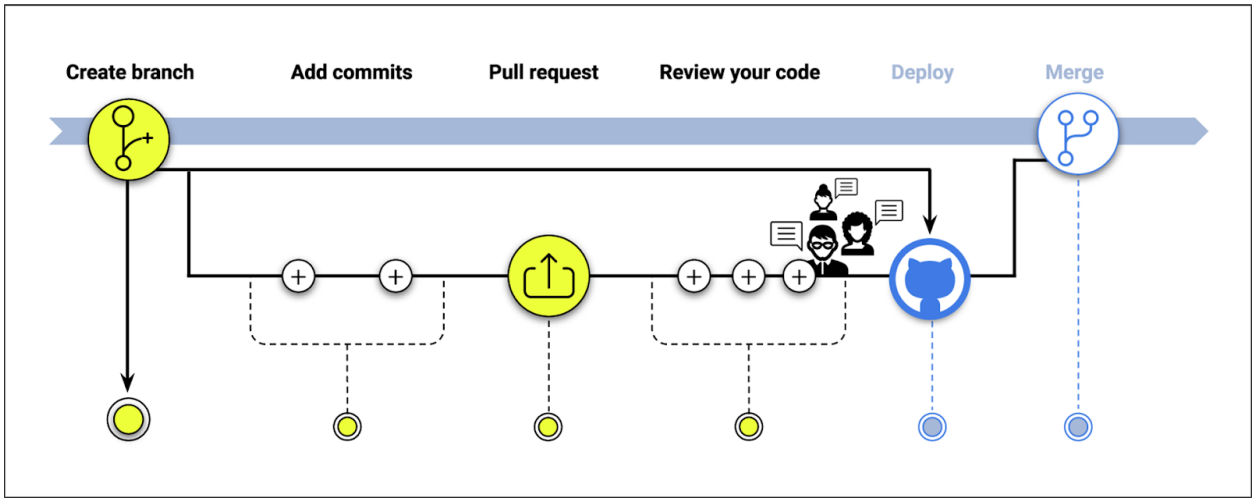


Alt-Text: The final drop-down menu offers two options: “Create Pull Request” and “Create Draft Pull Request.”.

Step 4: Review Your Code

Once a pull request has been opened, the person reviewing your changes might have questions or comments. Perhaps the coding style doesn’t match project guidelines, the change is missing unit tests, or maybe everything looks great and props are in order. Pull requests are designed to encourage and capture this type of conversation. Also, you can continue to push to your branch in light of feedback about your commits. If someone comments that you forgot to do something or if there is a bug in the code, you can make corrections in your branch and push up the change. GitHub will show your new commits and any additional feedback you might receive in

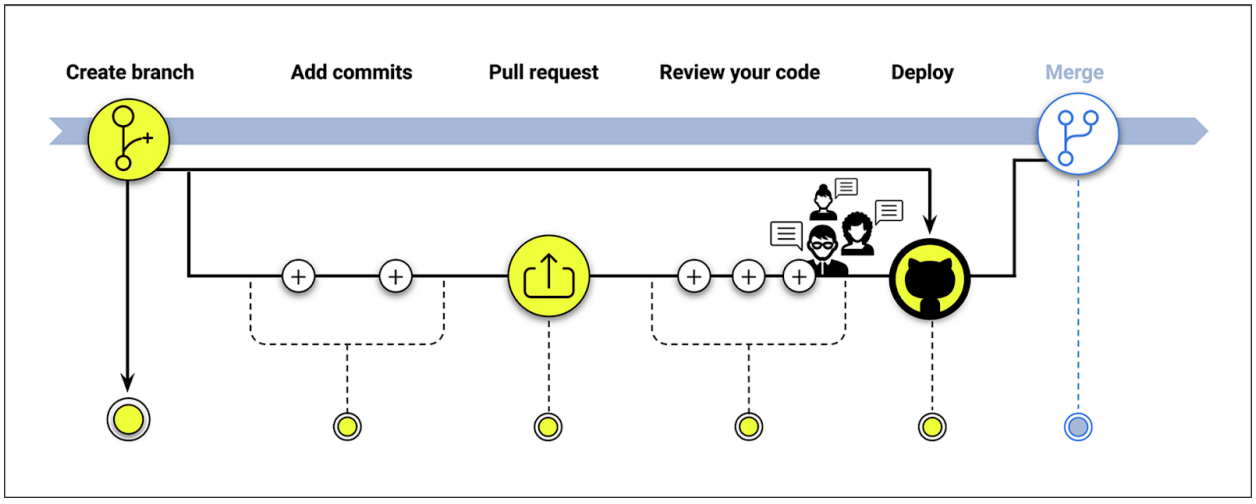
the unified pull request view.



Alt-Text: GitHub flow's fourth step is to review your code.

Step 5: Deploy for Final Testing

With GitHub, you can deploy from a branch for final testing in production before merging to main. Once your pull request has been reviewed and the branch passes your tests, you can deploy your changes to verify them in production. If your branch causes issues, you can roll it back by deploying the existing main branch into production.

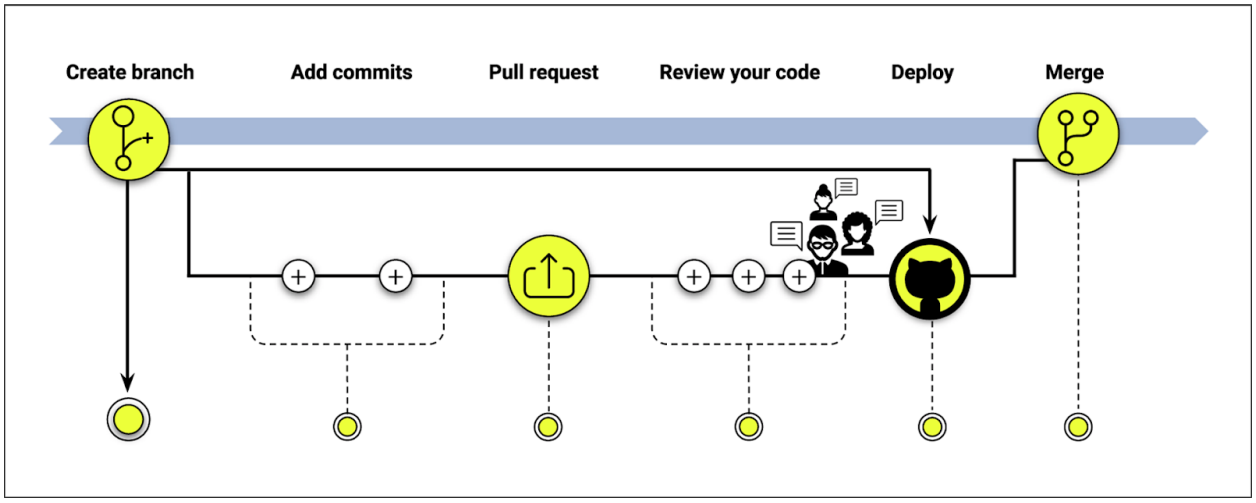


Alt-Text: GitHub flow's fifth step is to deploy for final testing.

Step 6: Merge Your Code

Now that your changes have been verified in production, it is time to merge your code into the main branch. Once merged, pull requests preserve a record of the historical changes to your code. These records are searchable so that anyone can review them easily to understand why

and how a decision was made.



Alt-Text: GitHub flow's sixth step is to merge your code into the main branch.

Before performing a merge, there are a couple of preparation steps to take to ensure the merge is performed smoothly. Open the command line interface, if you have not already, and follow the instructions below:

1. Execute `git status` to ensure that `HEAD` is pointing to the correct merge-receiving branch. If needed, execute `git checkout <receiving>` to switch to the receiving branch or `checkout` directly to the merge-receiving branch if you know its name.

2. Make sure the receiving branch and the merging branch are up to date with the latest remote changes. Execute `git fetch` to pull the latest remote commits. Once the fetch is complete, ensure the main branch has the latest updates by executing `git pull`.

3. Now, a merge can be initiated by executing `git merge <branch name>`, whereby `<branch name>` is the name of the branch that will be merged into the receiving branch.

4. Once the merge is completed, deleting the branch you created is not necessary; however, it is best practice to prune unneeded branches.

Merge Conflicts

Merge conflicts can be an extremely frustrating experience. The good news is that Git offers powerful tools to help navigate and resolve conflicts. Git can manage most merges on its own with automatic merging features. A conflict arises when two separate branches have made edits to the same line in a file, or when a file has been deleted in one branch but edited in the other. Conflicts will most likely happen when working in a team environment.

There are many tools to help resolve merge conflicts. Git has plenty of command line tools that we'll discuss here. In addition to Git, many third-party tools offer streamlined merge conflict support features.

Version control systems are all about managing contributions between multiple distributed collaborators. Sometimes multiple collaborators may try to edit the same content, if collaborator A tries to edit code that collaborator B is editing, a conflict may occur. To minimize the occurrence of conflicts, collaborators will work in separate isolated branches. The `git merge` command's primary responsibility is to combine separate branches and resolve any conflicts edits.

Understand Merge Conflicts

Merging and conflicts are a common part of the Git experience. Conflicts in other version control tools like Subversion (SVN) can be costly and time-consuming. Git makes merging easy. Most of the time, Git will figure out how to automatically integrate new changes. Conflicts generally arise when two people have changed the same lines in a file, or if one collaborator deleted a file while another collaborator was modifying it. In these cases, Git cannot automatically determine what is correct. Conflicts only affect the collaborator conducting the merge, while the rest of the team is unaware of the conflict. Git will mark the file as being conflicted and halt the merging process. It is then the collaborator's responsibility to resolve the conflict.

Types of Merge Conflicts

A merge can enter a conflicted state at two separate points: when starting and during a merge process. The following covers how to address each of these conflict scenarios.

Git Fails to Start the Merge

A merge will fail to start when Git sees there are changes in either the working directory or staging area of the current project. Git fails to start the merge because these pending changes could be written over by the commits that are being merged in. When this happens, it is not because of conflicts with other collaborators but because of conflicts with pending local changes. The local state will need to be stabilized using `git stash`, `git checkout`, `git commit`, or `git reset`. A merge failure on start will output the following message:

```
error: Entry '<fileName>' not uptodate. Cannot merge. (Changes
in working directory)
```

Git Fails During the Merge

A failure during a merge indicates a conflict between the current local branch and the branch being merged. This indicates a conflict with another collaborator code. Git will do its best to merge the files, but it will leave things for you to resolve manually in the conflicted files. A mid-merge failure will output the following error message:

```
error: Entry '<fileName>' would be overwritten by merge. Cannot
merge. (Changes in staging area)
```

Create a Merge Conflict

The following will simulate a conflict instance to later examine and resolve. The simulation will be executed using command line Git interface.

```
mkdir git-directory-name
cd git-directory-name
git init .
echo "this is some content to mess with" > merge.txt
git add merge.txt
git commit -am "initial content commit"
[main (root-commit) d48e74c] initial content commit
1 file changed, 1 insertion(+)
create mode 100644 merge.txt
```

This code simulation executes a sequence of commands that accomplishes the following:

1. Creates a new directory named “`git-directory-name`,” makes changes to that directory, and initializes it as a new Git repository.

2. Creates a new text file named “`merge.txt`” with some content in it.

3. Add `merge.txt` to the repository and commit it.

At this stage, we have a new repository with one branch `main` and a file `merge.txt` with content in it. The next step is to create a new branch to use as the conflicting merge.

```
git checkout -b new_branch
echo "new content in the new branch to merge later" > merge.txt
```

```
git commit -am "changed the content on merge.txt to cause a conflict"
[new_branch 6282319] changed the content on merge.txt to cause a
conflict
1 file changed, 1 insertion(+), 1 deletion(-)
```

The preceding command sequence achieves the following:

1. Creates and checks out a new branch named "`new_branch`."
2. Overwrite the content in `merge.txt`.
3. Commit the new content.

With the `new_branch`, we have created a commit that overrides the content of `merge.txt`.

```
git checkout main
Switched to branch 'main'
echo "content to append" >> merge.txt
git commit -am "appended content to merge.txt"
[main 24fbe3c] appended content to merge.tx
1 file changed, 1 insertion(+)
```

This chain of commands checks out the main branch, appends content to `merge.txt`, and commits it. This now puts our example repository in a state where we have two new commits. One in the `main` branch and one in the `new_branch` branch. At this stage, let's use `git merge` on the second branch and observe the outcome.

```
git merge new_branch
Auto-merging merge.txt
CONFLICT (content): Merge conflict in merge.txt
Automatic merge failed; fix conflicts and then commit the result.
```

As predicted, a conflict appears.

How to Identify Merge Conflicts

As shown in the preceding example, Git will produce some descriptive output to inform us that a conflict has occurred. We can gain further insight by running the `git status` command.

```
git status
On branch main
```

```
You have unmerged paths.
(fix conflicts and run "git commit")
(use "git merge --abort" to abort the merge)

Unmerged paths:
(use "git add <file>..." to mark resolution)

both modified:   merge.txt
```

The output from `git status` indicates that there are unmerged paths due to a conflict. The `merge.txt` file now appears in a modified state. Let's examine the file and see what's modified.

```
cat merge.txt
<<<<<<< HEAD
this is some content to mess with
content to append
=======
totally different content to merge later
>>>>>>> new_branch_to_merge_later
```

Here we have used the `cat` command to put out the contents of the `merge.txt` file. We can see some strange new additions:

- <<<<<<< HEAD
- `=======`
- `>>>>>>> new_branch_to_merge_later`

Think of these new lines as "conflict dividers." The `=======` line is the "center" of the conflict. All the content between the center and the `<<<<<<< HEAD` line is content that exists in the current branch main, which the `HEAD` ref is pointing to. Alternatively, all present content is in our merging branch.

Git Commands for Resolving Merge Conflicts

General Tools

```
git status
```

The `status` command is in frequent use when working with Git. It helps identify conflicted files as it checks the status of the current state of the working directory. It is a great command for monitoring the states of both the working directory and repository.

```
git log --merge
```

Passing the `--merge` argument to the `git log` command will produce a log with a list of commits that conflicts between the merging branches.

```
git diff
```

The `diff` command is very helpful to predict and prevent merge conflicts. It helps find differences between states of a repository/files.

Tools for When Git Fails to Start a Merge

```
git checkout
```

The `checkout` command can be used for undoing changes to a file or for changing branches.

```
git reset --mixed
```

The `reset` command can be used to undo changes to the working directory and staging area.

Tools for When Git Conflicts Arise During a Merge

```
git merge --abort
```

Adding `--abort` to `git merge` will exit from the merge process and return the branch to the state before the merge began.

```
git reset
```

The `reset` command allows us to reset conflicted files to a known good state during a merge conflict.

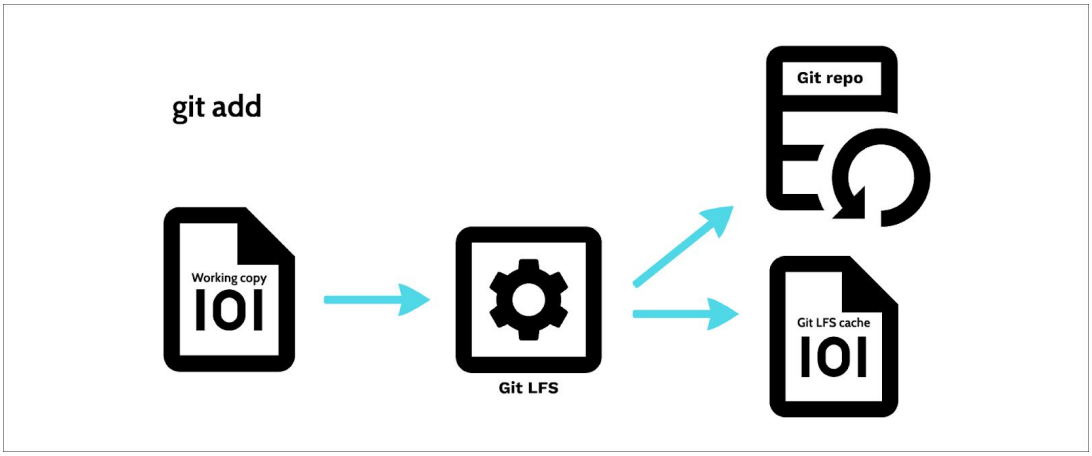
Use Git Large File Storage

Not many people are aware that Git and GitHub have a file size limit of 100MB. File sizes of 50MB trigger a warning message but can still be pushed through. To circumvent the file size restriction, developers created Git large file storage (LFS), which is a git extension programmed in Go.

Understand the Mechanics of Git LFS

Before we dive in, let's understand how Git LFS works under the hood. In essence, the large files are stored separately in a distinct location from your repository by just placing a pointer file in your repository directing to your actual file. Let's see how it actually works.

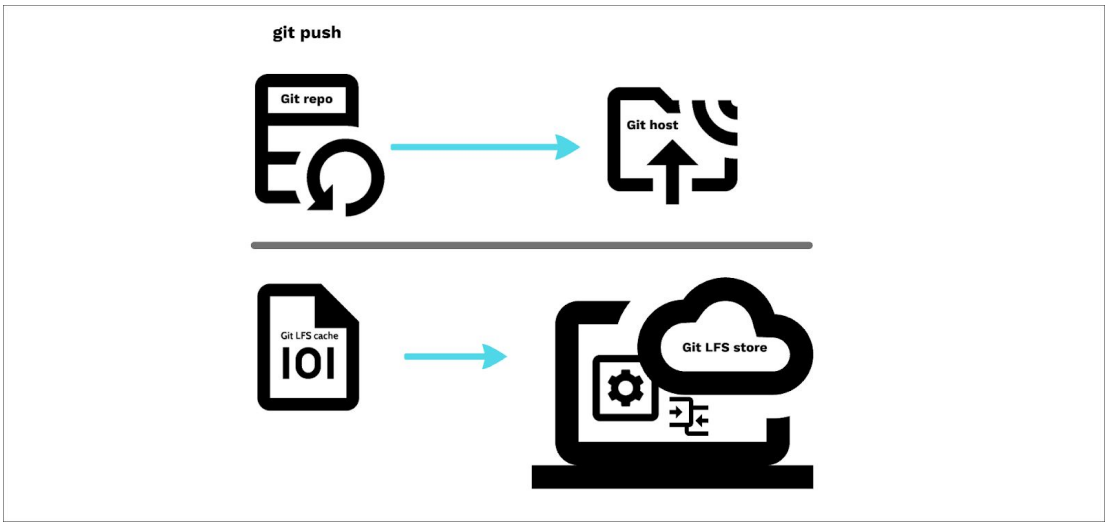
1. When you add a file to your repository, Git LFS replaces its contents with a pointer and stores the file contents in a local Git LFS cache to the remote Git LFS store tied to your Git repository.



Alt-Text: Visual workflow illustrates what happens when adding a large file with Git LFS.

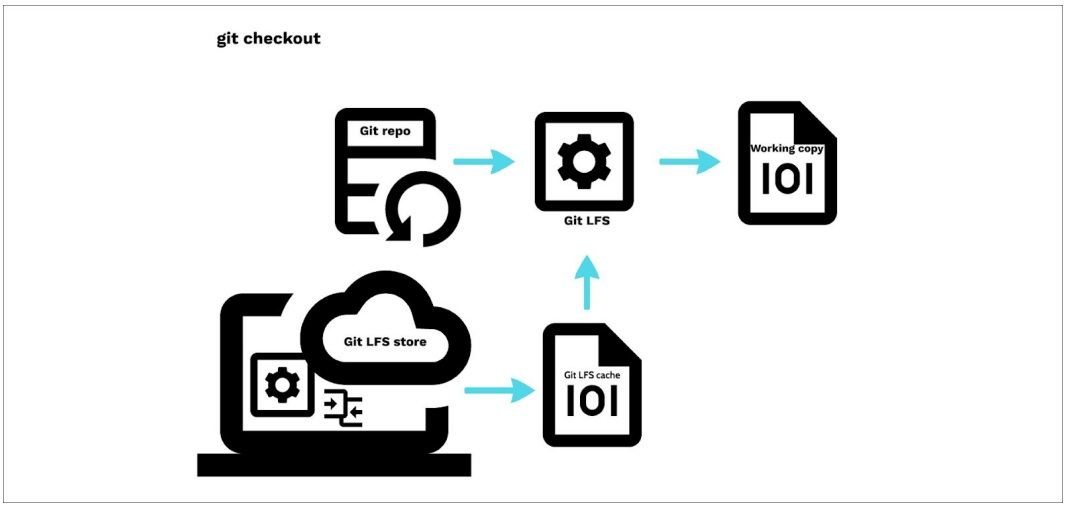
2. When you push new commits to the server, any Git LFS files referenced by the newly pushed commits are transferred from your local git LFS cache to the remote Git LFS

store tied to your Git repository.



Alt-Text: Visual workflow illustrates what happens when pushing new commits.

3. When you use `checkout` for a commit that contains Git LFS pointers, they are replaced with files from your local Git LFS cache, or downloaded from the remote Git LFS store.



Alt-Text: Visual workflow illustrates what happens when using git checkout for a commit.

Install Git LFS

There are few ways to install Git LFS, but for sake of simplicity, let's stick to the basics:

1. From GitHub's website, download and install Git LFS.

2. Once Git LFS is installed, run `git lfs install` from the command line. Although we have installed Git LFS on our machine, we need to run the command on the specific

repository. In case you have other repositories handling large files, you need to initiate Git LFS individually per repository.

```
git lfs install
```

Track Files

Git LFS won't automatically start handling any large files until we specify what types of files we would like it to track. Let's see how we tell Git LFS to track these files:

1. Run `git lfs track "*.file_extension_to_tack"`. Note that the quotes around the file type are important and omitting them will cause errors, the * is telling Git LFS to track all files with that specific type of extension. Let's see an example on the command line with a comma-separated values (CSV_ file.

```
git lfs track "*.csv"
```

2. Also, we can specify to track a single file.

```
git lfs track "final_ml_project_data.csv"
```

Run .gitattributes

Once a file is tracked by Git LFS, a `.gitattributes` file is created and the file being tracked will be listed in this file. Note that you must run `git add .gitattributes` to commit and push to your github repository. In fact, Git LFS automatically creates or updates `.gitattributes` files; however, you need to commit any changes to the `.gitattribute` file to your repository yourself.

1. Run `git add`, and proceed with `commit` and `push` as usual.

```
git add .gitattributes
git commit -m 'adding .gitattributes file'
git push origin main
```

2. Once pushed, you'll see the file on your GitHub repo. Note that you'll only see that file in your local directory.

Alt-Text: The GitHub repository displays a list of files..

If you open the file, you'll see a list of the files being tracked by Git LFS, in our case `.csv`.

Alt-Text: Open .gitattributes file to see .csv files are tracked.

Add Files

We could add files during the previous `.gitattributes` step. Instead, we separated the step for adding files to avoid unnecessary confusion.

Now that we have our project GitHub repository, Git LFS installed, type of files tracked, and `.gitattributes` file created and pushed, it is time to add the large files.

From your local project directory, add the files or file and run `add`, `commit`, and `push` commands as usual.

```
git add your_file_name.csv
git commit -m 'adding file to the repo'
git push origin main
```

Once you've executed the final command, all of the files you've been working on will be pushed from your computer to your repository.