Collaborative Coding with Git and GitHub

Getting Started with Git and GitHub Part 2
Coffee, Cookies, and Coding (C-cubed) Workshops

January 26, 2026

This workshop provides three examples demonstrating common uses of a configured Git and GitHub setup. The first two, in another handout, show how to establish a connection by starting with either a local project directory or an existing remote repository. In this example, students will find a practice collaboration exercise to complete with peers, where they propose code adjustments for collaborators to review via Pull Requests.

To effectively follow these guidelines, we suggest that you use the following file structure. By the end of this module, your workshop directory should contain:

```
dsde-workshops/
  |- start-local/
  |- JHU-CRC-Vaccinations/
  |- JHU-CRC-Cases-and-Deaths/
```

> **!** Important
>
> Specific directions are provided for establishing these repositories. To interact with the pre-prepared JHU-CRC examples, you must first create a "clean-break" copy in your own GitHub account. When practicing collaboration, only one team member needs to complete this step; all others can then clone that member's copy to their local device.
>
> Find the directions for creating a "clean-break" on the Book of Workshops webpage for this workshop: Getting Started with Git and GitHub - Accessing the Codespaces

Exercise Framework and Approach

In the following exercises, you and your team are going to assume roles commonly seen in DevOps teams. DevOps teams—combining software development (Dev) and IT operations (Ops)—embody a collaborative philosophy where everyone shares responsibility, to varying degrees, for different phases of code development, integration, and deployment [1]. This team structure and distribution of responsibilities is popular across code development teams of all types. Git's version control and code history management, combined with GitHub's codebase distribution and collaboration features, have become instrumental tools in supporting these workflows [2].

DevOps is commonly understood as a continuous cycle of development, testing, integration, and deployment. Canonically, there are eight distinct phases in a DevOps cycle, where team members may be responsible for specific aspects and different pieces of the codebase may exist in different phases of the process at any given time [3]. In your professional lives, you may experience the DevOps framework on a spectrum, as roles are separated and divided depending on factors such as the scope and complexity of the project, as well as the bandwidth and expertise of each team member.

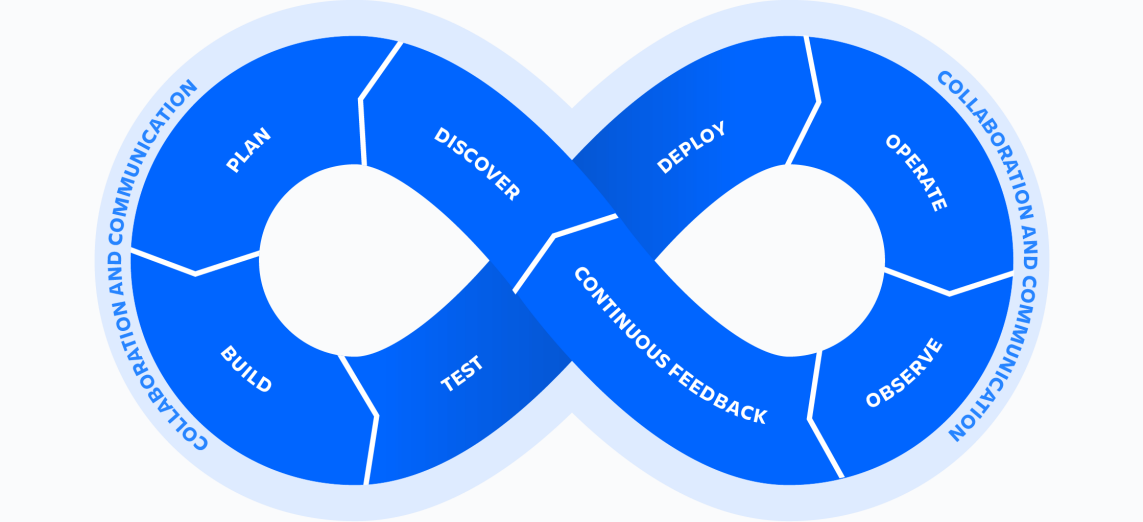


Figure 1: The DevOps lifecycle from Atlassian's "What Is DevOps?" article [3].

For data analysis projects, where there typically is no code-based product that requires deployment or continual integration, team members will assume to varying degrees the roles of Developer, Reviewer, and codebase Maintainer:

- **Developer:** Create and implement features through code changes.
- **Reviewer:** Review and approve proposed changes submitted through Pull Requests.
- **Maintainer:** Manage the repository and ensure branch protection rules are configured correctly to enforce review structures and code integrity for the team.

In practice, project administrators or team leads configure protection rules for the codebase, which is beyond the scope of this exercise. We'll provide these rules, and each person will take turns maintaining code alignment with these standards and discussing their implications.

Why do we need protection rules? In this framework, the `main` branch on GitHub serves as the most current version of the project—the basis everyone uses for their work. To preserve its integrity, collaborators use branches to create isolated work environments where they can test and implement intended changes [2,4,5].

Additionally, since this codebase is intentionally simple, participants are not restricted to one role. Instead, the exercises encourage collaborative participation where students discuss outcomes as team members propose code changes and review results together. The exercises are designed for teams of 2 to 4 people. We encourage everyone to go through each step together to experience the process from different perspectives and encounter various challenges.

Accessing the Codebase

In this example, you will copy a pre-prepared codebase from the remote repository "ysph-dsde/JHU-CRC-Cases-and-Deaths" on GitHub. One team member will copy this repository to their GitHub account (directions below) and add the remaining members as collaborators. Each team member can then clone the copied repository to their local device.

Using this codebase, you will be guided through collaborative GitHub workflows. What you explore is ultimately up to your team, but we have included simple exercises covering branching and merge rules, submitting Pull Requests, and resolving merge conflict. Articles are provided at each section that every team member is recommended to reference and review so they can fully participate in the exercises.

Making a Clean-Break Copy

1. Open the Getting Started with Git and GitHub - Accessing the Codespaces webpage. From there, open the "ysph-dsde/JHU-CRC-Cases-and-Deaths" GitHub repository in a new tab and scroll down to the *Making a Clean-Break Copy* section.
2. Follow **Method #1** or **Method #2** to create a clean-break copy of the remote repository in your personal GitHub account. We recommend starting with **Method #1**, but if that takes too long or does not work, **Method #2** is available as a backup option.

Inviting Team Members

1. In the YOUR-USER-ID/JHU-CRC-Cases-and-Deaths repository, click the Settings tab at the top.
2. In the left sidebar under the "Access" heading, click Collaborators and teams.
3. Under "Manage Access," click Add people and enter your teammates' GitHub usernames. Because this workshop is intended to allow everyone the opportunity to play different roles on a DevOps team, we suggest you give each teammate "Admin" rights.
4. Your teammates will receive email invitations that they must accept to gain access. Once they have accepted the invitations, they are ready to proceed with cloning the repository.

Clone the Project Codebase

1. Open the command-line application (i.e. Terminal for Macs and Git Bash for Windows) and navigate to the file location you want to temporarily store the repository copy.

Command-Line Interface

```
cd "~/dsde-workshops/"     # Go to workshop directory
```

2. In the copied GitHub repository, "YOUR-USER-ID/JHU-CRC-Cases-and-Deaths", find the repository's SSH or HTTPS URL. Copy the SSH or HTTPS URL from your GitHub repository by clicking the  button. Which one you choose depends on your preference and configurations (see Configurations and Credentials - Transfer Protocols for additional details).

   For example the URLs will look like this:

```
# SSH
git@github.com:YOUR-USER-ID/JHU-CRC-Cases-and-Deaths.git

# HTTPS
https://github.com/YOUR-USER-ID/JHU-CRC-Cases-and-Deaths.git
```

3. Clone the repository to your local device.

Command-Line Interface

```
# SSH
git clone git@github.com:YOUR-USER-ID/JHU-CRC-Cases-and-Deaths.git

# HTTPS
git clone https://github.com/YOUR-USER-ID/JHU-CRC-Cases-and-Deaths.git
```

4. Open the cloned project directory.

Command-Line Interface

```
cd "JHU-CRC-Cases-and-Deaths"
```

5. Open the R project environment.

Command-Line Interface

```
open JHU-CRC-Cases-and-Deaths.Rproj
```

6. In RStudio, open "`Plot Cases and Deaths.R`" and initialize the environment.

RStudio

```
renv::init()     # Set up renv environment
renv::restore()  # Install locked package versions
```

1. Pull Requests

Most new users of GitHub know it for its role in codebase sharing, acting as the central server for distributed version control. GitHub also provides valuable tools that support collaborative workflows by establishing clear expectations, roles, and project timelines. Pull Requests are one such central feature.

Pull Requests allow teams to systematically organize and review proposed modifications to the project's core code, typically stored on the `main` branch. They also serve as a way to inform your team about individual work, create a contribution track record, and refine changes before finalizing a merge [6]. Because of its core role in collaborative projects, this is one of the most important tools to learn and implement in an effective collaborative team.

There are many ways to use the Pull Request feature, including setting rulesets, which we will briefly cover later. Unfortunately, we do not have time to explore all of these options here. Instead, we will focus on a general introduction to creating, reviewing, and approving Pull Requests in GitHub. Below are GitHub Docs links for additional Pull Request features and uses.

- Creating a pull request: Step-by-step guides for initiating and managing Pull Requests using different operating systems, the GitHub browser, or the Git Command-Line Interface (CLI) [7].
- Reviewing proposed changes in a pull request: How-to guide for accessing, viewing, and managing Pull Request reviews [8].
- Requesting a pull request review: Step-by-step guide to requesting review from a specific reviewer [9].
- About pull request merges: An explanation of how merges are interpreted in the project's version control history, their default settings and restrictions, what a "squash" merge means, and how to initiate different types of merge protocols [10].

1. Create a new branch from `main` where you can modify the provided script and submit your changes as a pull request. Each team member should use a unique branch name. In the example below, the branch is called `little-feature`.

   > **Note**
   >
   > This assumes you haven't modified `main` since cloning. If you have, either sync with the remote repository and have everyone pull the changes, or revert to the original version.

   Command-Line Interface

   ```
   git checkout -b little-feature  # Create and switch to branch
   ```

2. In RStudio, open "`Plot Cases and Deaths.R`" and initialize the environment.

   Command-Line Interface

   ```
   open JHU-CRC-Cases-and-Deaths.Rproj       # Open the R Project
   ```

   RStudio

   ```
   renv::init()     # Set up renv environment
   renv::restore()  # Install locked package versions
   ```

3. Each team member should implement one of the following code changes. Make sure each person chooses a different change to avoid conflicts for now—we'll cover handling conflicts in the next section.

   RStudio - Team Member 1: Add Captions

```
plot_cases <- plot_cases +
    labs(subtitle = "Data Source: CSSE at Johns Hopkins University",
         caption = "Visualization by YOUR-TEAM-NAME")

ggsave("plot_cases_captioned.jpeg", plot_cases, width = 20, height = 12,
↪  units = "cm")

plot_deaths <- plot_deaths +
    labs(subtitle = "Data Source: CSSE at Johns Hopkins University",
         caption = "Visualization by YOUR-TEAM-NAME")

ggsave("plot_deaths_captioned.jpeg", plot_deaths, width = 20, height =
↪  12, units = "cm")
```

RStudio - Team Member 2: Change `plot_cases` to Show Data for "New England"

```
plot_cases <- covid19_cases_deaths |>
  # Filter out the "New England" census division
  filter(Province_State == "New England") |>
  ggplot(aes(Week, Confirmed_Cases_Daily)) +
    geom_line(color = "#00356b") +
    scale_y_continuous(labels = unit_format(unit = "M", scale = 1e-6)) +
    scale_x_date(date_breaks = "4 month", date_labels = "%b %Y") +
    labs(x = NULL, y = "Daily Counts",
         title = "Daily Confirmed Counts of COVID-19 in New England") +
    theme_minimal()

ggsave("plot_cases_NE.jpeg", plot_cases, width = 20, height = 12, units
↪  = "cm")
```

RStudio - Team Member 3: Change `plot_deaths` to Show Data for "New England"

```
plot_deaths <- covid19_cases_deaths %>%
  # Filter out the "New England" census division
  filter(Province_State == "New England") |>
  ggplot(aes(Week, Deaths_Daily)) +
    geom_line(color = "#A353FF") +
    scale_x_date(date_labels = "%m/%Y",
```

```
                  breaks = as.Date(c("2020-01-01", "2020-06-30",
                                     "2021-01-01", "2021-06-30",
                                     "2022-01-01", "2022-06-30",
                                     "2023-01-01", "2023-06-30"))) +
    scale_y_continuous(labels = scales::label_comma()) +
    labs(x = NULL, y = "Daily Counts",
         title = "Daily Death Counts of COVID-19 in New England") +
    theme_minimal()

ggsave("plot_deaths_NE.jpeg", plot_deaths, width = 20, height = 12,
↪  units = "cm")
```

RStudio - Team Member 4: Create a Third Graph for Two Districts

```
plot_districts <- covid19_cases_deaths |>
  filter(Province_State %in% c("West South Central", "Pacific")) |>
  ggplot(aes(Week, Confirmed_Cases_Daily, color = Province_State)) +
    geom_line() +
    scale_color_manual(values = c("West South Central" = "#A353FF",
                                  "Pacific" = "#00356b")) +
    scale_y_continuous(labels = unit_format(unit = "K", scale = 1e-3)) +
    scale_x_date(date_breaks = "4 month", date_labels = "%b %Y") +
    labs(x = NULL, y = "Daily Counts",
         title = "Daily Confirmed Counts of COVID-19 in the West South
         ↪  Central and Pacific",
         color = "Districts") +
    theme_minimal()

plot_districts

ggsave("plot_districts.jpeg", plot_districts, width = 20, height = 12,
↪  units = "cm")
```

4. Commit your changes to "`Plot Cases and Deaths.R`" and the corresponding plot JPEG with a descriptive commit message. For example, here's the message I used for my changes:

Command-Line Interface

```
git add .
git commit -m "Move some details to the Book of Workshops. Update
↪  README.md to reflect these changes, and simplify the repo contents."
```

5. Push these changes to your branch on the GitHub repository. If this is your first time pushing from this branch, you will need to set the upstream (remote URL). There are two approaches for this, but Option #1 is most appropriate since we have already made local changes.

   a. **Option #1:** Push and set the upstream from your local device. Use this when local changes have already been made to this branch.

Command-Line Interface

```
git push -u origin little-feature      # Set origin as upstream
```

   b. **Option #2:** Create the branch manually on GitHub first, then pull it to your local device. This approach is only for when you are first setting up the branch on your device.

   On the repository landing page, click main ⟶ "View all branches." Click New branch and enter the exact name of your local branch. Keep the source as `main`, then click Create branch. Finally, push your changes from your local device.

Command-Line Interface

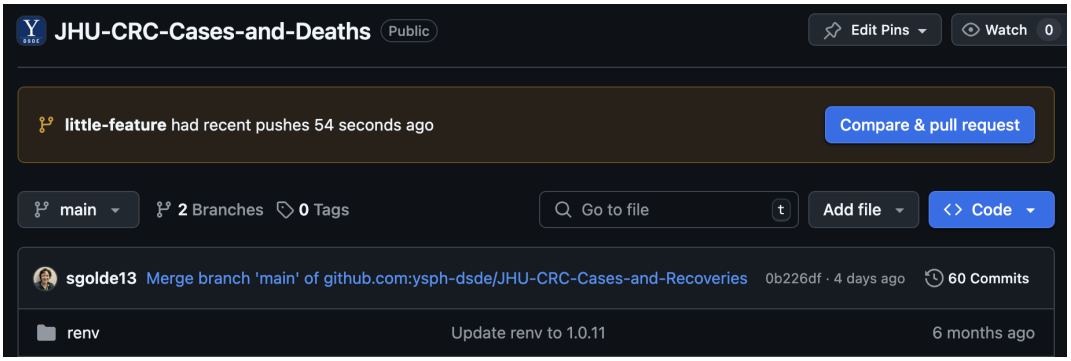
```
git fetch origin little-feature    # Fetch remote branch

# Make local copy with upstream
git checkout -b little-feature origin/little-feature
```

Output

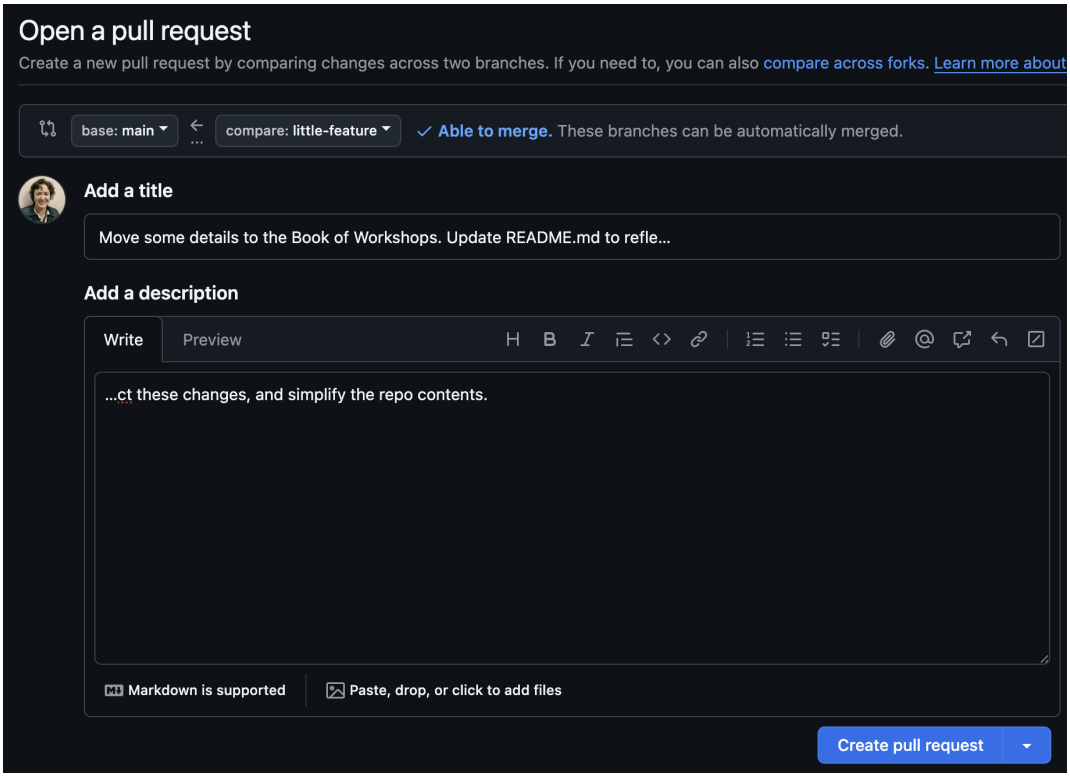
```
branch 'little-feature' set up to track 'origin/little-feature'.
Switched to a new branch 'little-feature'
```

6. Refresh the GitHub page. A yellow banner should appear at the top prompting you to create a Pull Request. Click Compare & pull request. If no banner appears, navigate to the Pull requests tab and create one manually.

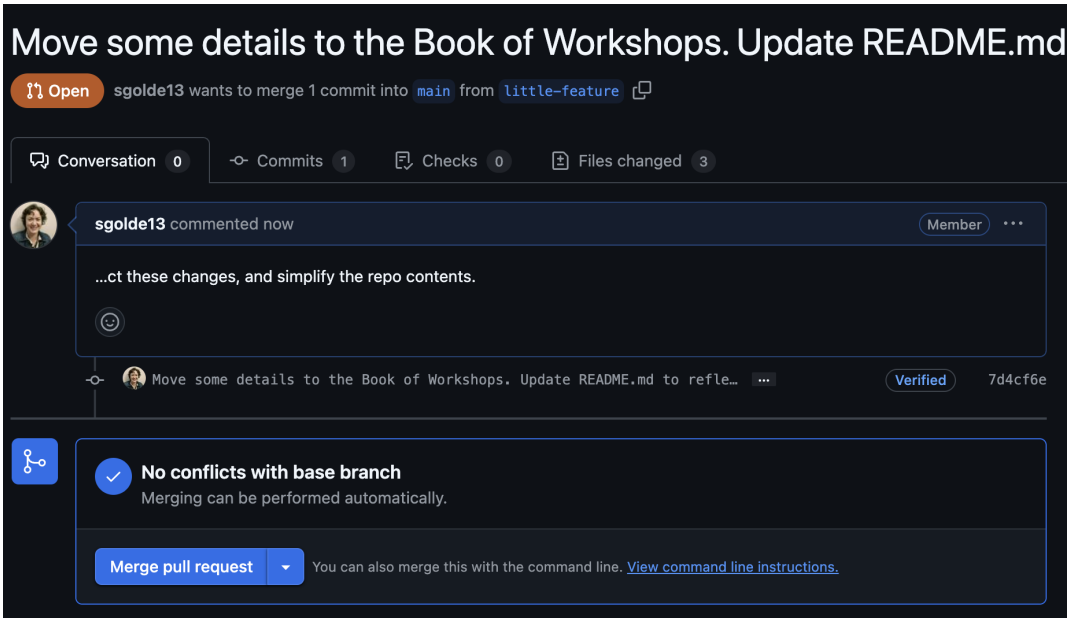
> **One Reviewer, One Pull Request**
>
> GitHub's default settings require only one approval to merge a Pull Request. Have each person select one Pull Request to review and merge into `main`.



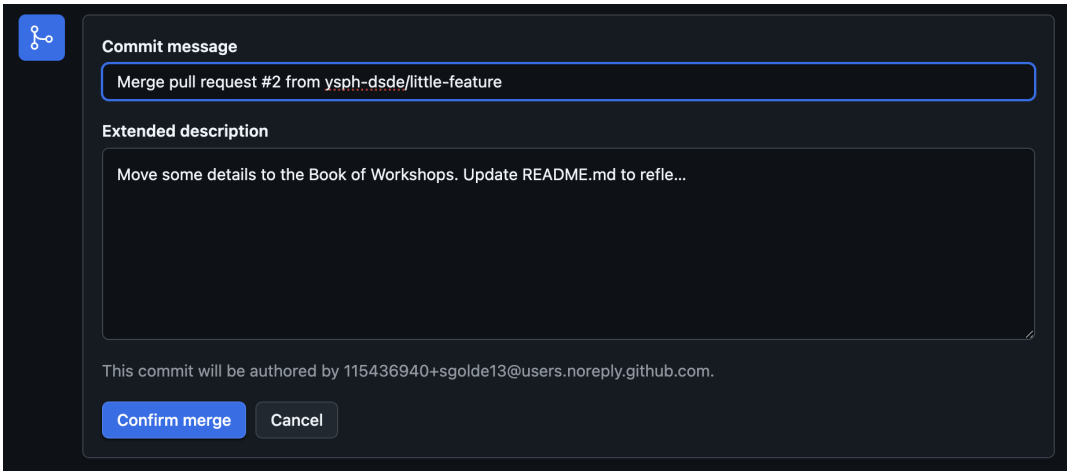
7. **OPTIONAL:** Edit the title or add a description for the Pull Request.



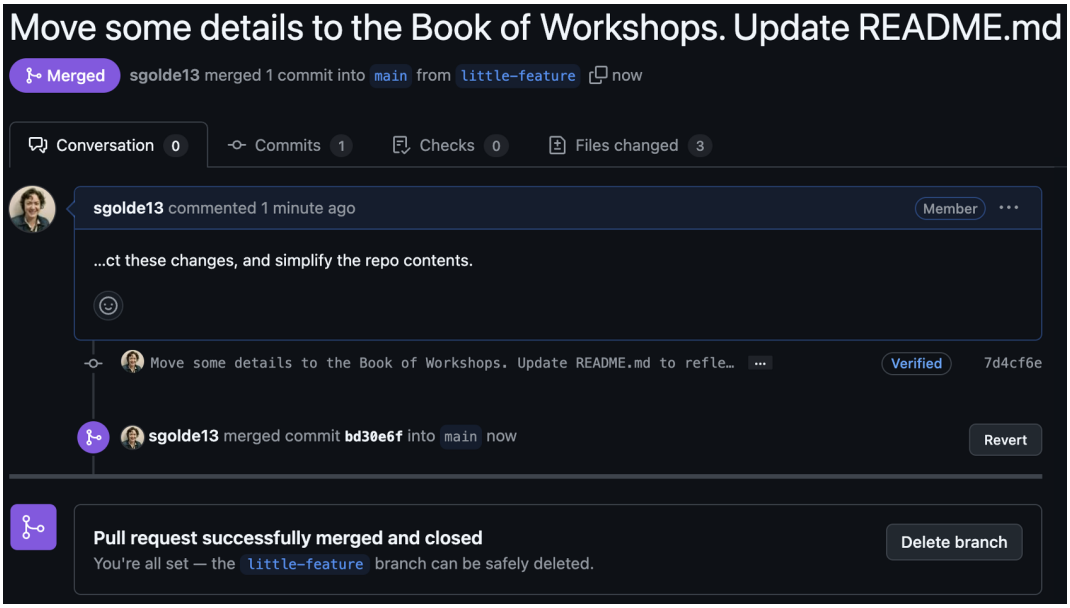
8. Clicking **Create pull request** opens the Pull Request and compares your branch with `main` for potential merge conflicts. Since each team member modified different parts of the code, no conflicts should be detected.



9. Clicking **Merge pull request** merges the Pull Request, with the option to include a custom merge commit message.



10. Clicking Confirm merge completes the merge and closes the Pull Request.



11. As a final step, ensure your local repository reflects these changes and clean up branches based on the old version of `main`.

Command-Line Interface

```
git checkout main                 # Checkout main branch
git pull                          # Pull in the latest changes
git branch -D little-feature      # Delete the outdated branch
```

2. Reconciling Merge Conflicts

As seen in the previous section, GitHub calls the branch receiving changes (in this case, `main`) the "base" and the branch proposing changes (in this case, `little-feature`) the "compare." Merge conflicts arise when the base and compare branches propose contradictory edits to the same line of code. In these cases, Git cannot automatically reconcile the differences and prompts you to either integrate both changes manually or choose which version to keep.

While we call this a "merge" conflict, it can arise from any coalescing operation, whether a `git merge` or `git rebase` [11]. Merge conflicts can occur when directly merging two branch heads, cherry-picking a historical commit, or when one branch deletes a file that the other branch modifies [12,13]. Regardless of how the merge conflict arises, there are two approaches for reviewing and reconciling the differences: resolving conflicts in GitHub or locally through the CLI [14,15].

Resolving conflicts works the same in both places, but GitHub only handles simple conflicts while the CLI can resolve any conflict. We'll walk you through a local resolution example, as this approach applies to all merge conflicts. After that, we've prepared two scenarios to try: one simple conflict resolvable in GitHub, and one complex conflict requiring local resolution.

Before jumping into the exercises, let's look at how Git highlights code impacted by merge conflicts. Consider the following scenario: two team members are updating the study inclusion/exclusion criteria saved in the project repository. They each create a branch (`john-revisions` and `alice-updates`), make their edits, and commit their changes. Before merging into `main`, they first merge their branches together to combine their work. The file on each branch looks like:

`inclusion_exclusion_criteria.txt` **on** `john-revisions`

```
# Study Protocol

1. Inclusion Criteria
   - Age above 18
   - Confirmed diagnosis of condition X

2. Exclusion Criteria
   - Diagnosis of condition Y
```

`inclusion_exclusion_criteria.txt` **on** `alice-updates`

```
# Study Protocol

1. Inclusion Criteria
   - Age above 21
   - Diagnosis of condition X

2. Exclusion Criteria
   - Diagnosis of condition Y
   - Recent major surgery
```

As you can see, some lines differ between the branches, and some lines appear in only one version. When they attempt to merge Alice's branch into John's, a conflict occurs.

Command-Line Interface

```
git checkout john-revisions     # Switch to the john-revisions branch
git merge alice-updates         # Merge changes from the alice-updates branch
```

Output

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

A status check shows the merge has been paused, awaiting user input to resolve the conflicts. The repository stays in merge mode until you either resolve the conflicts and finish the merge or abort it.

Command-Line Interface

```
git status    # Show current status
```

Output

```
On branch john-revisions
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 added:      inclusion_exclusion_criteria.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Opening the conflicted file shows that Git has added annotations highlighting the problem areas:

Command-Line Interface

```
cat inclusion_exclusion_criteria.txt      # Display file content
```

Output

```
# Study Protocol

1. Inclusion Criteria
<<<<<<< HEAD
   - Age above 18
   - Confirmed diagnosis of condition X

2. Exclusion Criteria
   - Diagnosis of condition Y
=======
   - Age above 21
   - Diagnosis of condition X

2. Exclusion Criteria
   - Diagnosis of condition Y
   - Recent major surgery
>>>>>>> alice-updates
```

The conflicting lines are marked by `<<<<<<< HEAD` at the top and `>>>>>>> alice-updates` at the bottom; non-conflicting content falls outside these markers. This shows that

`alice-updates` is being merged into `HEAD` (the latest commit of `john-revisions`). The section between `<<<<<<< HEAD` and `=======` contains `john-revisions` content, while the section between `=======` and `>>>>>>> alice-updates` contains `alice-updates` content.

> **Code Like a Pro**
>
> Git frequently uses the `HEAD` nomenclature to indicate that you are referencing or interacting with the most recent commit status of that branch, as reflected in the project's `.git` directory.

To reconcile these changes, edit the file and remove all conflict markers.

Command-Line Interface

```
vim inclusion_exclusion_criteria.txt      # Edit the file in the CLI
cat inclusion_exclusion_criteria.txt      # Display file content
```

Output

```
# Study Protocol

1. Inclusion Criteria
   - Age above 21
   - Confirmed diagnosis of condition X

2. Exclusion Criteria
   - Diagnosis of condition Y
   - Recent major surgery
```

With these changes made, we are ready to reattempt coalescing the two branches.

Command-Line Interface

```
git add .     # Stage the updated file
git status    # Show current status
```

Output

```
On branch john-revisions
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
    modified:   inclusion_exclusion_criteria.txt
```

Command-Line Interface

```
git commit -m "Merged and resolved the conflict in
↪  inclusion_exclusion_criteria.txt."
```

Output

```
[john-revisions cf29589] Merged and resolved the conflict in
↪  inclusion_exclusion_criteria.txt.
```

Best practice is to delete branches that are no longer active or based on outdated code. Since Alice's updates have been merged, we can now delete the `alice-updates` branch locally and remotely [16].

Command-Line Interface

```
git push -d origin alice-updates    # Delete remote copy first
git branch -D alice-updates         # Delete local copy
```

Output

```
Deleted branch alice-updates (was e00a860).
```

2.a. Reconciling Merge Conflicts in GitHub

1. Two team members should propose minor edits from their local machines. Each team member starts by creating a branch based on `main` to make their edits.

Command-Line Interface

```
git checkout main                            # Ensure base is main

git checkout -b modify-x-axis-plot-cases  # Team member 1 ONLY
git checkout -b change-title-plot-cases   # Team member 2 ONLY

open JHU-CRC-Cases-and-Deaths.Rproj        # Open the R Project
```

2. Both team members should modify the x-axis of `plot_cases`: change the breaks from every four months to every two months and add an x-axis label that says "Week" (currently blank). One team member should also modify the plot title.

RStudio - Team Member 1: Change x-axis Breaks and Label

```
plot_cases <- covid19_cases_deaths |>
  filter(Province_State == "New England") |>
  ggplot(aes(Week, Confirmed_Cases_Daily)) +
    geom_line(color = "#00356b") +
    scale_y_continuous(labels = unit_format(unit = "M", scale = 1e-6)) +
    # Format the x-axis to show dates as Jan 2020 from 01/01/2020,
    # spaced every two months (changed from four months).
    scale_x_date(date_breaks = "2 month", date_labels = "%b %Y") +
    # Add labels and title to the plot (x-axis label changed
    # from NULL to "Week")
    labs(x = "Week", y = "Daily Counts",
         title = "Daily Confirmed Counts of COVID-19 in New England") +
    theme_minimal()
```

RStudio - Team Member 2: Change x-axis Breaks and Label with New Title

```
plot_cases <- covid19_cases_deaths |>
  filter(Province_State == "New England") |>
  ggplot(aes(Week, Confirmed_Cases_Daily)) +
    geom_line(color = "#00356b") +
    scale_y_continuous(labels = unit_format(unit = "M", scale = 1e-6)) +
    # Format the x-axis to show dates as Jan 2020 from 01/01/2020,
    # spaced every two months (changed from four months).
    scale_x_date(date_breaks = "2 month", date_labels = "%b %Y") +
    # Add labels and title to the plot (x-axis label changed
    # from NULL to "Week" and new title)
    labs(x = "Week", y = "Daily Counts",
         title = "Daily COVID-19 Confirmed Counts in New England") +
    theme_minimal()
```

3. Both team members commit their local changes and push them to the GitHub repository. GitHub will then prompt them to create a Pull Request for each branch.

Command-Line Interface

```
git add "Plot Cases and Deaths.R"          # Stage the edits
git commit -m "Update plot-cases x-axis"   # Commit changes

git push origin modify-x-axis-plot-cases   # Team member 1 ONLY
git push origin change-title-plot-cases    # Team member 2 ONLY
```

4. On GitHub, open the Pull requests page and click Compare & pull request next to `change-title-plot-cases`. This generates a Pull Request merging `change-title-plot-cases` (compare branch) into the base branch (`main` by default).

> **Note**
>
> Notice that the merge of `change-title-plot-cases` (compare branch) into `main` (base branch) can be completed automatically, indicating no conflicts are detected.

5. Change the base branch from `main` to `modify-x-axis-plot-cases`. Update the merge title and description as desired, then click Create pull request.

> **Note**
>
> Notice that GitHub now shows that merge conflicts have been detected.

6. On the opened Pull Request page, scroll to the merge assessment section, marked with  on the left of the box. A warning will indicate that merge conflicts were detected in "`Plot Cases and Deaths.R`." Click Resolve conflicts to continue.

7. This will open a text editor on the GitHub page where you can reconcile the changes as demonstrated above. When you are done, click Mark as resolved in the top-right corner of the text editor and then click Commit merge to finalize the changes.

> **Note**
>
> Notice that GitHub provides three hyperlinked options to quickly implement changes.
>
> Also note that the base branch version is labeled "(Current change)" and the compare branch version is labeled "(Incoming change)" in their respective sections.

8. Complete the Pull Request by merging it into the base branch as shown above.

2.b. Reconciling Merge Conflicts in the CLI

1. Two team members should propose minor edits from their local machines. Each team member starts by creating a branch based on `main` to make their edits.

Command-Line Interface

```
git checkout main                       # Ensure base is main

git checkout -b modify-plot-deaths      # Team member 1 ONLY
git checkout -b delete-plot-deaths      # Team member 2 ONLY

open JHU-CRC-Cases-and-Deaths.Rproj     # Open the R Project
```

2. The team member on `modify-plot-deaths` should change the title of `plot_deaths` and save the updated plot.

RStudio - Team Member 2: Adjust the Title

```
plot_deaths <- covid19_cases_deaths %>%
  filter(Province_State == "New England") |>
  ggplot(aes(Week, Deaths_Daily)) +
    geom_line(color = "#A353FF") +
    scale_x_date(date_labels = "%m/%Y",
                 breaks = as.Date(c("2020-01-01", "2020-06-30",
                                    "2021-01-01", "2021-06-30",
                                    "2022-01-01", "2022-06-30",
                                    "2023-01-01", "2023-06-30"))) +
    scale_y_continuous(labels = scales::label_comma()) +
    # Add labels and title to the plot (adjust the title)
    labs(x = NULL, y = "Daily Counts",
         title = "Daily COVID-19 Death Counts in New England") +
    theme_minimal()

ggsave("plot_deaths_NE.jpeg", plot_deaths, width = 20, height = 12,
↪  units = "cm")
```

3. The same team member should then commit and push their changes to GitHub.

Command-Line Interface

```
git add .                                  # Stage all edits
git commit -m "Update plot_deaths title"   # Commit changes
git push origin modify-plot-deaths         # Push to GitHub
```

4. The team member on the `delete-plot-deaths` branch should delete the `plot_deaths_NE.jpeg` file.

Command-Line Interface

```
rm plot_deaths_NE.jpeg                     # Delete the plot

git add plot_deaths_NE.jpeg                # Stage all edits
git commit -m "Delete plot_deaths JPEG"    # Commit changes
git push origin delete-plot-deaths         # Push to GitHub
```

5. On GitHub, open the Pull requests page and click Compare & pull request next to `delete-plot-deaths`. This generates a Pull Request merging `delete-plot-deaths` (compare branch) into the base branch (`main` by default).

6. Change the base branch from `main` to `modify-plot-deaths`. Update the merge title and description as desired, then click Create pull request.

7. On the Pull Request page, scroll to the merge assessment section (marked with  on the left). A warning indicates merge conflicts were detected but the merge status cannot be loaded. This means you must resolve the conflict locally using the CLI.

8. On one team member's device, pull the missing branch from GitHub, check out the `modify-plot-deaths` branch, and attempt to merge `delete-plot-deaths` into it. In the example below, `modify-plot-deaths` is already local, so we fetch `delete-plot-deaths` from GitHub to prepare for the merge.

Code Like a Pro

We use fetch here instead of pull because we don't need a fully integrated local copy of `delete-plot-deaths`. Our goal is simply to merge our team member's remote changes into `modify-plot-deaths`, then merge the result into `main`. Once complete, these feature branches can be deleted.

Command-Line Interface

```
git checkout modify-plot-deaths     # Ensure on modify-plot-deaths
git fetch origin delete-plot-deaths # Retrieve delete-plot-deaths
```

Output

```
From github.com:EXAMPLE-USER/JHU-CRC-Cases-and-Deaths
 * branch            delete-plot-deaths -> FETCH_HEAD
```

Command-Line Interface

```
git merge origin/delete-plot-deaths   # Merge from FETCH-HEAD
```

Output

```
CONFLICT (modify/delete): plot_deaths_NE.jpeg deleted in HEAD and
↪  modified in origin/delete-plot-deaths.  Version
↪  origin/delete-plot-deaths of plot_deaths_NE.jpeg left in tree.
Automatic merge failed; fix conflicts and then commit the result.
```

9. In this example, we will implement the plot modifications and keep the resulting JPEG.

> **Discussion Questions**
>
> To complete this scenario, we did not need to remove conflict markers like we did in previous examples, and we did not need to recreate the plot. Why was that the case?
>
> How would the commit progression on each branch need to change in order to produce conflict markers during this step?

Command-Line Interface

```
git status      # Show current status
```

Output

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

Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark resolution)
    deleted by them:   plot_deaths_NE.jpeg
```

Command-Line Interface

```
git add plot_deaths_NE.jpeg               # Keep the plot
git commit -m "Merged and resolved the conflict with
↪  plot_deaths_NE.jpeg."    # Commit changes
```

Output

```
[modify-plot-deaths 8625245] Merged and resolved the conflict with
↪  plot_deaths_NE.jpeg.
```

8. Push the changes to GitHub and delete the `delete-plot-deaths` branch, if you have a local copy, as it no longer contains unique information [16].

> **Discussion Questions**
>
> How would the process differ if you instead merged `modify-plot-deaths` into `delete-plot-deaths`?

Command-Line Interface

```
git push origin modify-plot-deaths        # Push to GitHub
git push -d origin delete-plot-deaths     # Delete remote copy first
git branch -D delete-plot-deaths          # Delete local copy
```

3. Branch Rules

Branch protection rules in GitHub help maintain the integrity of important branches by enforcing certain policies before changes can be merged. Common rules used by DevOps teams include requiring Pull Request reviews, passing status checks, and restricting who can push changes directly to the branch. To learn more, see GitHub's documentation on About protected branches and Managing a branch protection rule [17,18].

> **Code Like a Pro**
>
> More generally, you can set rulesets that are applicable to branches and tags in the repository. You can learn more about the generalized application of rulesets in About rulesets and Creating rulesets for a repository, where the usage of `fnmatch` syntax for setting "Target branches" patterns is explained [19,20].

Only one team member needs to do this part. We recommend either taking turns setting up rulesets that will automatically enforce codebase maintenance expectations, or having one person screenshare the configuration process so everyone can observe.

> **Note**
>
> Branch protection rulesets may require the repository to be Public. If the ruleset doesn't apply, convert the repository from Private to Public.

1. In the YOUR-USER-ID/JHU-CRC-Cases-and-Deaths repository, click the Settings tab at the top.
2. In the left sidebar under the "Code and automation" heading, click Branches then the Add branch ruleset under the "Branch protection rules" section.

   Alternatively, click Rules then "Rulesets" followed by New ruleset ⟶ "New branch ruleset" under the "Rulesets" section.
3. In the ruleset window, fill out the fields with the following settings, then click Create.

   - **Ruleset Name:** Main Branch Protection

- **Enforcement status:** Active
- **Bypass list:** Since we recommend all team members have administrator rights, leave this blank for now so the ruleset applies to everyone. If you later adjust roles, you can use this list to exempt specific collaborators from these rules.
- **Target branches:** Click Add target ⟶ Include by pattern. In the popup window, type "main" to apply the ruleset to the `main` branch.
- **Rules:** Check "Require a pull request before merging" and "Block force pushes". Leave the default pull request settings, but set the number of required approvals to one less than your total team size.

4. With the branch ruleset in place, try pushing changes directly to the main branch to test the protection rules.

Command-Line Interface

```
git checkout main                          # Checkout main branch
echo "Direct push test" >> test-file.txt   # Create test file
git add test-file.txt                      # Stage changes
git commit -m "Test direct push to main"   # Commit changes
git push origin main                       # Push to main branch
```

5. Confirm that this commit attempt was blocked.

> **Discussion Questions**
>
> How does this ruleset help maintain the integrity of the main branch?
> Do you see this helping in your own project workflows?

6. Try creating a Pull Request to merge changes from your feature branch to `main`. Confirm that it requires the configured number of reviewers before allowing the merge.

> **Discussion Questions**
>
> How does this ruleset help maintain the integrity of the main branch?
> Thinking of your own projects, what other rulesets would be helpful to apply?

References

1. GitHub. What is DevOps? (2024).
2. Atlassian. Is GitOps the next big thing in DevOps? *Atlassian Tutorial*.
3. Atlassian. What is DevOps? *Atlassian Tutorial*.
4. Atlassian. Git branch. *Atlassian Tutorial*.
5. Atlassian. Creating and merging branches in git - git guides (2020) - YouTube. *YouTube* (2020).
6. GitHub. About pull requests. *GitHub Docs*.
7. GitHub. Creating a pull request. *GitHub Docs*.
8. GitHub. Reviewing proposed changes in a pull request. *GitHub Docs*.
9. GitHub. Requesting a pull request review. *GitHub Docs*.
10. GitHub. About pull request merges. *GitHub Docs*.
11. Atlassian. Merging vs. rebasing. *Atlassian Tutorial*.
12. Atlassian. How to resolve merge conflicts in git? *Atlassian Tutorial*.
13. GitKraken. How to resolve merge conflicts in git. *GitKraken*.
14. GitHub. Resolving a merge conflict on GitHub. *GitHub Docs*.
15. GitHub. Resolving a merge conflict using the command line. *GitHub Docs*.
16. Rankin, M. Version control - how do i delete a git branch locally and remotely? *StackOverflow* (2025).
17. GitHub. Managing a branch protection rule. *GitHub Docs*.
18. GitHub. About protected branches. *GitHub Docs*.
19. GitHub. Creating rulesets for a repository. *GitHub Docs*.
20. GitHub. About rulesets. *GitHub Docs*.