Collaborating on Research Code – Session 2

Pair work with branches, forks, and pull requests

In this session you will work in **pairs** on a small 2D advection–diffusion code hosted on GitHub. You will:

- fork a shared repository,
- create and use feature branches,
- open and review pull requests (PRs),
- resolve a simple merge conflict.

This document guides you step by step. You can use either the **terminal** or **VS Code**'s Git tools throughout.

0 Pre-flight context

- You will work in pairs as **review buddies**. Decide who is:
  - **Person A** and
  - **Person B**.
- The code is in `https://github.com/mandli/RESCUER_workshop`.
  Navigate to `Collaborative_Software_Development/hands-on/session2/code/`
- Each of you will:
  - fork this repository into your own GitHub account,
  - clone your fork locally,
  - create a feature branch,
  - implement a small change,
  - open a PR on your own fork.
- Then you will **review each other's PRs**, comment, and update your code.

1 Step 1 – fork and clone

1.1 Fork the workshop repository on GitHub

For both Person A and Person B:

1. Open the shared repository in your browser, e.g.:
   `https://github.com/mandli/RESCUER_workshop`
2. Make sure you are logged in to your own GitHub account.
3. Click the **"Fork"** button (top right).
4. Choose your own account as the destination.
5. After a moment, you should see a new repository under your account: for example
   `https://github.com/<your-username>/rescuer_workshop`.

1.2 Clone your fork locally

For each person, clone *your own* fork.

Option 1: Terminal

1. In a terminal, navigate to a suitable directory (e.g. Desktop or your home).
2. Run:

   ```
   git clone https://github.com/<your-username>/rescuer\_workshop.git
   cd rescuer\_workshop
   git status
   ```

3. You should see a clean working tree on the default branch (likely `main`).

Option 2: VS Code

1. Open VS Code.
2. Click the **Source Control** icon, then **Clone Repository**.
3. Paste your fork URL, e.g.: `https://github.com/<your-username>/rescuer_workshop.git`
4. Choose a local folder.
5. When asked, open the cloned repository.

1.3 Configure upstream remote (optional but recommended)

To keep your fork in sync with the original repo, you can add an `upstream` remote pointing to the original repository.

```
git remote -v
```

You should see `origin` pointing to
`https://github.com/<your-username>/rescuer_workshop.git`.
Add `upstream`:

```
git remote add upstream https://github.com/mandli/RESCUER\_workshop
git remote -v
```

Now `origin` should be your fork, and `upstream` should be the instructor's repo.

2 Step 2 – Local setup: environment and quick run

2.1 Create and activate a virtual environment

In the repository folder, navigate to
`Collaborative_Software_Development/hands-on/session2/code/`:

macOS / Linux

```
python3 -m venv .venv
source .venv/bin/activate
```

Windows (PowerShell)

```
python -m venv .venv
.\.venv\Scripts\Activate.ps1
```

2.2 Install dependencies

```
python -m pip install -U pip
pip install -r requirements.txt
```

2.3 Quick test run

Check that you can run the advection–diffusion code end-to-end:

```
python configs/make_ics.py
python -m computation.run --config configs/base_config.csv --ic configs/base_ic.
```

This should produce at least:

- a set of CSV files under `outputs/`, and
- a set of PNG plots (unless disabled).

If something fails, pair up and troubleshoot.

3 Step 3 – feature branch and small change

Each person now creates a small feature branch on their own fork.

3.1 Sync main (Optional but Safe)

```
git checkout main
git fetch upstream
git merge upstream/main
git push origin main
```

If you did not add `upstream`, you can skip this; just ensure your `main` matches your fork's default branch.

3.2 Create a feature branch

Choose a small feature idea, for example:

- add a command-line flag `-vx` or `-vy` to control advection velocity,
- add a plot title with key parameters (D, vx, vy, dt, nsteps),
- add a helper function to compute total mass and log it.

Now create a branch, e.g. `feature/plot-title`:

Terminal

```
git checkout -b feature/short-feature-name
git branch
```

VS Code

1. Click the branch name in the bottom-left corner.
2. Choose **Create new branch...**.
3. Name it `feature/short-feature-name`.

3.3 Implement the feature with 2–3 small commits

1. Edit the appropriate file(s), for example:
   - `computation/run.py` for new CLI options,
   - `computation/plotting.py` for plot titles,
   - `computation/solver.py` for numerical logic.
2. After each small change:

   Terminal:
   ```
   git status
   git add -p        # or: git add <file>
   git commit -m "feat: short description of change"
   ```
   VS Code:
   - Stage changed files in the Source Control panel.
   - Commit with a short, descriptive message.
3. Aim for **2–3 small commits** that each do one logical thing.

3.4 Run tests and the script

If the repository contains tests:

```
pytest -q
```

Then run the main script again, e.g.:

```
python -m computation.run --config configs/base_config.csv --ic configs/base_ic.
```

Ensure it still works and your feature behaves as expected.

4 Step 4 – Push branch and open a pull request

4.1 Push your feature branch to your fork

```
git push -u origin feature/short-feature-name
```

4.2 Open a Pull Request (PR) on GitHub

On GitHub, in your fork:

1. Go to the **“Pull requests”** tab.
2. Click **“New pull request”**.
3. Set:
   - **base**: your `main` branch,
   - **compare**: your feature branch, e.g. `feature/plot-title`.

4. Write:
   - a short **title**, e.g. `feat: add plot title with parameters`,
   - a brief **description**, explaining what you changed and why it matters for the advection–diffusion example.
5. Click **“Create pull request”**.

Checkpoint: Both Person A and Person B should now have an open PR in their own fork.

5 Step 5 – Swap PRs and perform code reviews

Now you will review each other’s work.

5.1 Swap links and assign reviewers

1. Person A: copy the URL of your PR and send it to Person B.
2. Person B: do the same and send your PR URL to Person A.
3. On GitHub, each of you can optionally assign your partner as a reviewer.

5.2 5.2 Review checklist

For each PR you review, check:

- **Correctness**
  - Does the code do what the PR claims?
  - Are parameter choices (dt, D, velocities) sensible?
- **Clarity**
  - Is the change small and focused?
  - Are function and variable names understandable?
  - Are docstrings or comments clear where needed?
- **Scope**
  - Does the PR avoid unrelated refactors?
  - Do the commits form a coherent story?
- **Documentation**
  - If user-facing behaviour changed, is `README` or help text updated?
- **Tests and Runs**
  - Are tests passing? (`pytest -q`)
  - Does the script run with a sample configuration?

5.3 Write constructive comments

In the GitHub PR view:

1. Click on **"Files changed"**.
2. Add inline comments where you have suggestions or questions.
3. Aim for at least **3–5 meaningful comments**, for example:
   - "Could we log the total mass here to monitor conservation?"
   - "Maybe rename `vx0` to `vx` for consistency."
   - "Consider checking that dt does not exceed the stability bound."
4. Avoid unhelpful comments such as only "Looks good" or "This is wrong" without explanation.

5.4 Respond to feedback and update the PR

After you receive comments on your own PR:

1. Discuss briefly with your partner if needed (2–3 minutes).
2. Apply changes locally:

   ```
   # Edit files according to the review comments
   git add .
   git commit -m "chore: address review comments"
   git push
   ```

3. The PR will update automatically with new commits.

If there are no more concerns, your partner can use **Approve** on GitHub, or simply state in a comment that the changes look good.

6 Step 6 – Simple merge conflict exercise

If time allows, you can practice a small, controlled merge conflict within your own fork.

6.1 Create two branches that modify the same line

1. Start from `main`:

   ```
   git checkout main
   ```

2. Create and switch to branch `conflict/variant-a`:

   ```
   git checkout -b conflict/variant-a
   ```

3. In a simple location (e.g. a log message or title in `computation/run.py`), change a string to version A, commit, and push.
4. Switch back to `main`, then create `conflict/variant-b`:

   ```
   git checkout main
   git checkout -b conflict/variant-b
   ```

5. Change the *same line* to a different text (version B), commit, and push.

6.2 Merge and resolve conflict with VS Code

1. Switch to `conflict/variant-a`:

   ```
   git checkout conflict/variant-a
   git merge conflict/variant-b
   ```

2. You should see a merge conflict in the file you edited.
3. Open that file in VS Code. It will show options such as: *Accept Current Change*, *Accept Incoming Change*, *Accept Both*.
4. Decide how to combine the two versions (for example, a clearer message).
5. Once done, stage and commit the resolved file:

   ```
   git add <file>
   git commit -m "merge: resolve conflict in message"
   ```

Optional: push this branch and open a PR if you want to see the merge history on GitHub.

7 Wrap-up

By the end of this session, you should have:

- Forked a shared advection–diffusion repository.
- Cloned your fork and created a local environment.
- Implemented a small feature on a dedicated branch with several commits.
- Opened a PR on your own fork and reviewed your partner's PR.
- Optionally, experienced and resolved a simple merge conflict.

These steps mirror typical collaborative workflows in research software projects: feature branches, forks, reviews, and careful merging on a shared code base.