**Cloud computing**

**Lab 03**

**Name:** Abiha Nadeem

**Roll no:** 2023-BSE-001

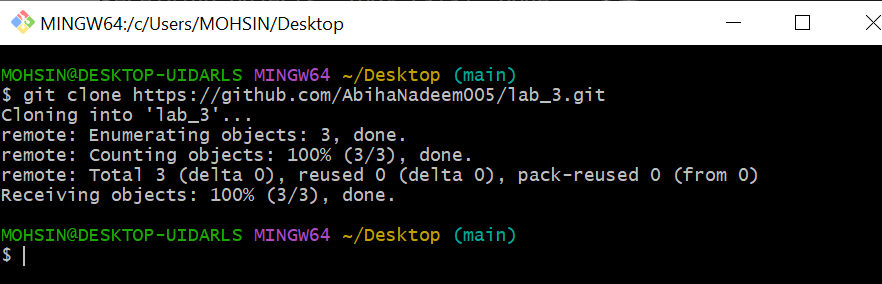
**Submitted to:** Engr. Muhammad Shoaib

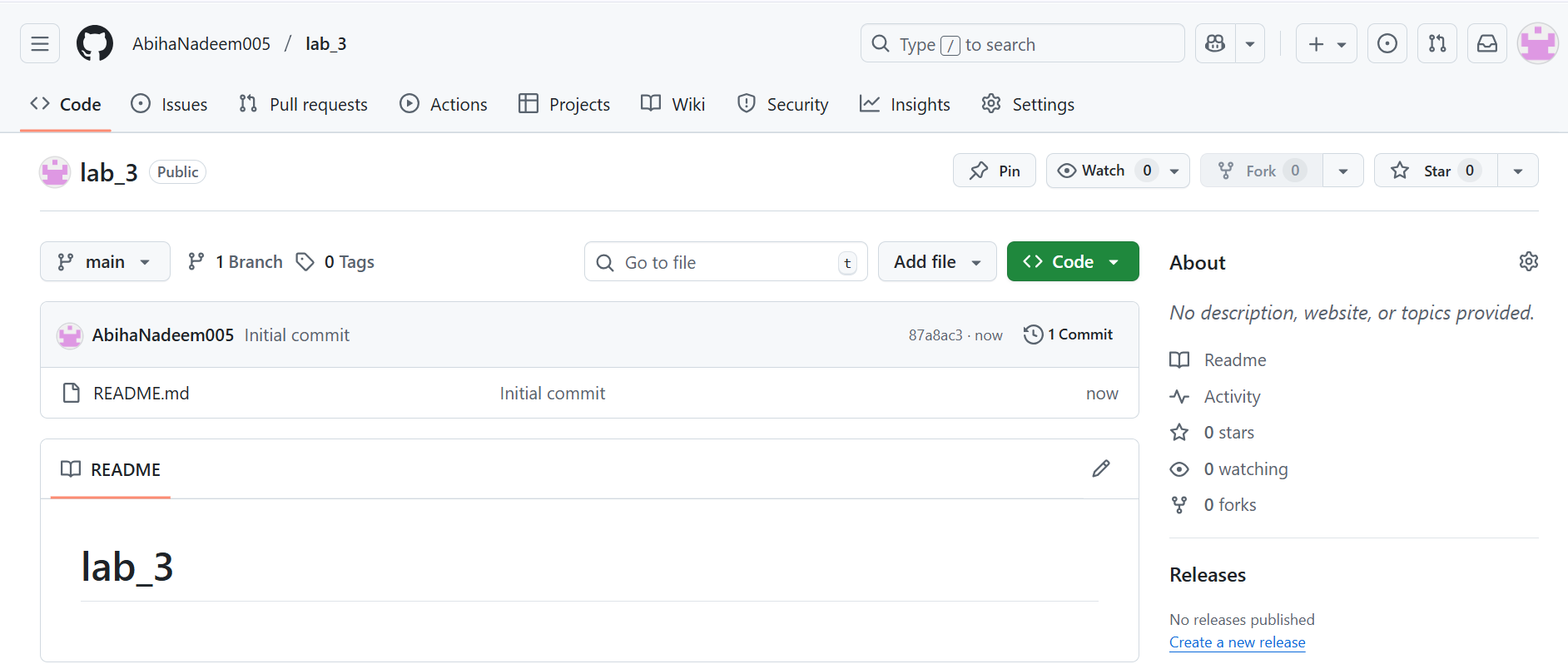
**Task 1 – Handling Local and Remote Commit Conflicts (Pull vs Pull --rebase)**

This task demonstrates what happens when your local repository and remote repository both have new commits — and how git pull and git pull --rebase behave differently.

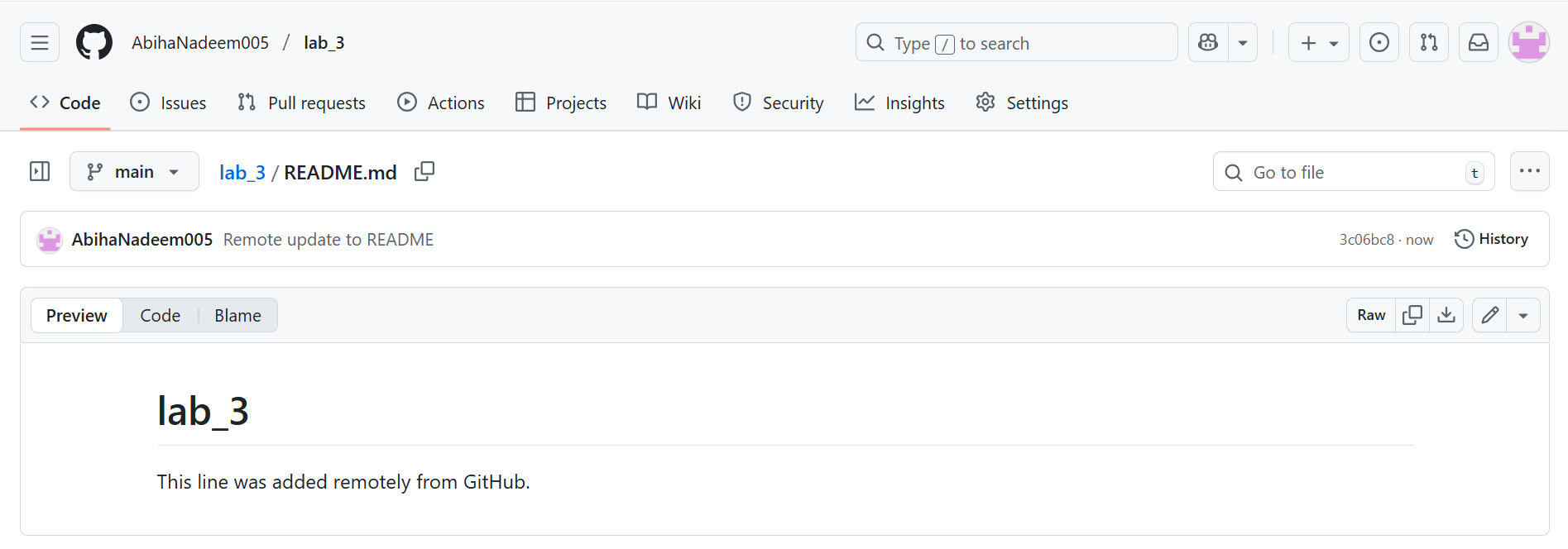
**Steps**

1. Open your repository on **GitHub** and edit the README.md file directly in the browser.





* + Add a new line such as:
  + This line was added remotely from GitHub.
  + Commit your change on GitHub (e.g., message: *Remote update to README*).
  + Save a screenshot of your browser showing the change as remote\_edit.png.

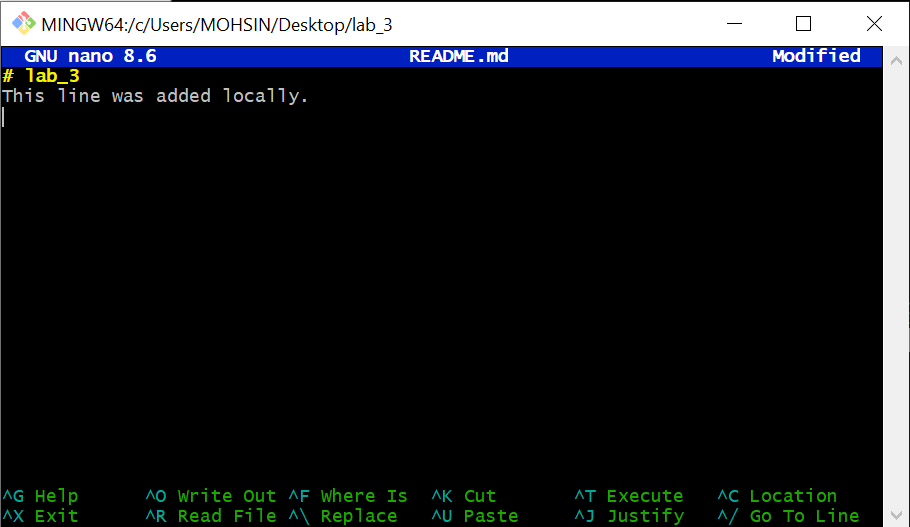


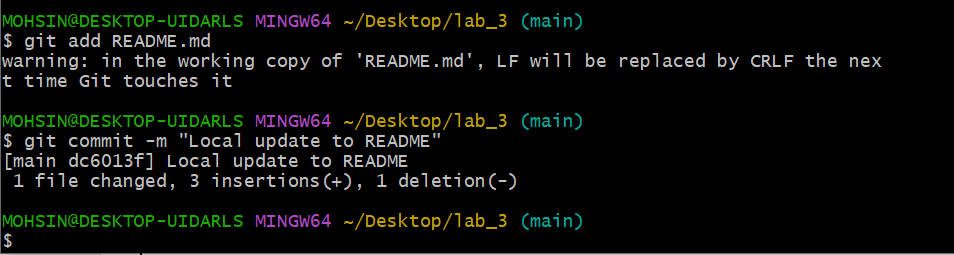
1. On your **local machine**, open the same repository folder.
   * Edit the same README.md file locally, for example:
   * This line was added locally.
   * Stage and commit your change:
   * git add README.md

git commit -m "Local update to README"

* + Save a screenshot of your terminal as local\_commit.png.







1. Try to push:

git push origin main

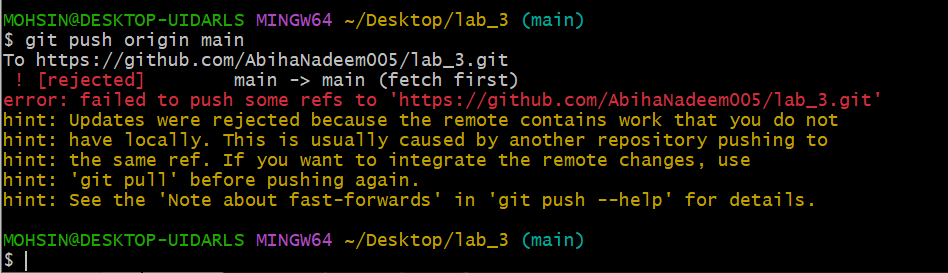
You’ll see an error message similar to:

! [rejected] main -> main (fetch first)

error: failed to push some refs to 'github.com:username/repo.git'

hint: Updates were rejected because the remote contains work that you do not have locally.

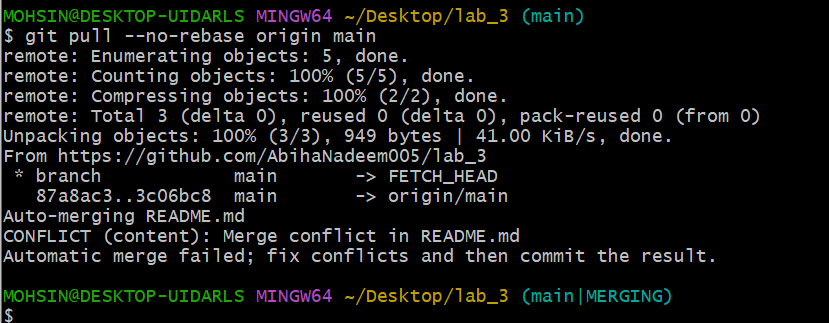
* + Save a screenshot of this error as push\_error.png.

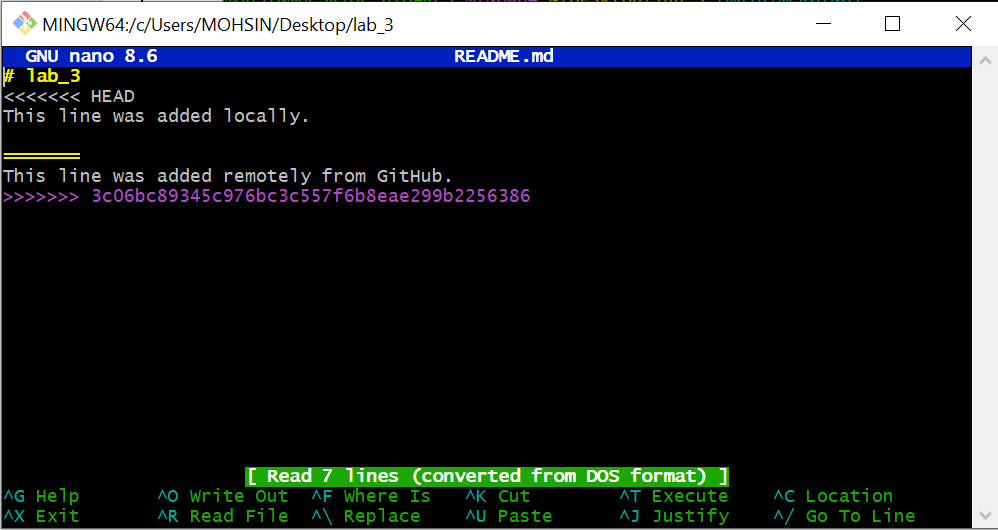


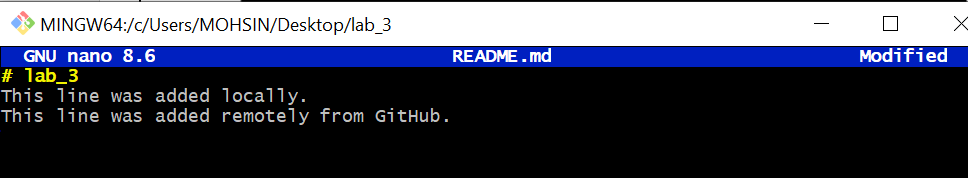
1. To fix this, pull the latest changes from remote:

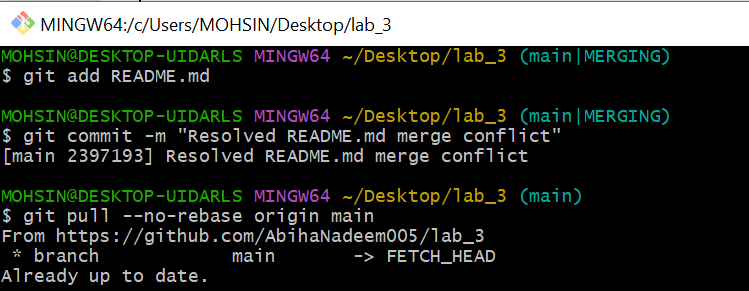
git pull --no-rebase origin main

* + Git will **merge** your local and remote commits, creating a *merge commit*.
  + Save a screenshot as merge\_commit.png.





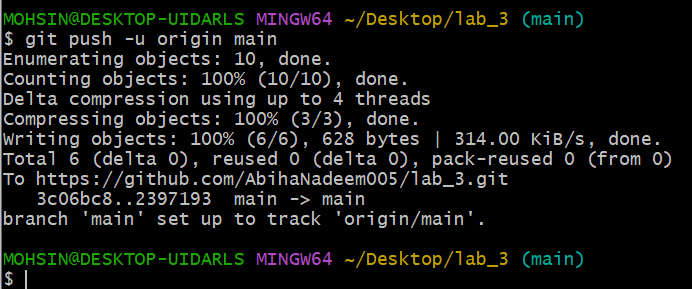




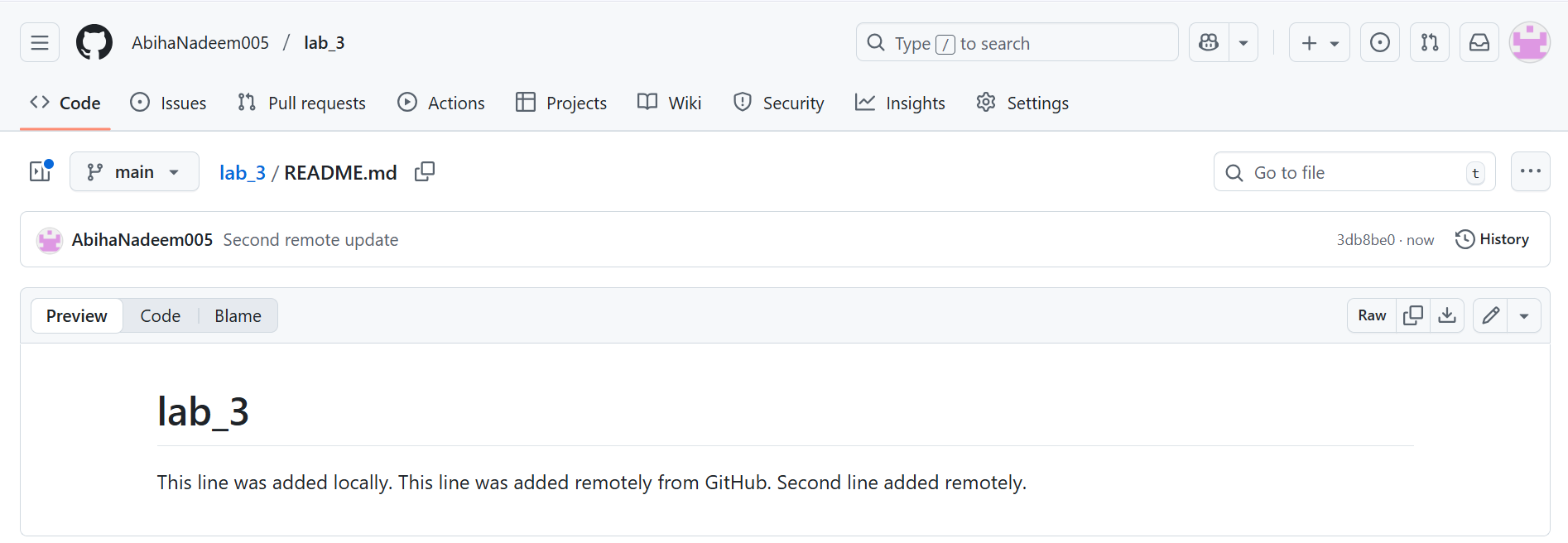
1. Push again:

git push -u origin main

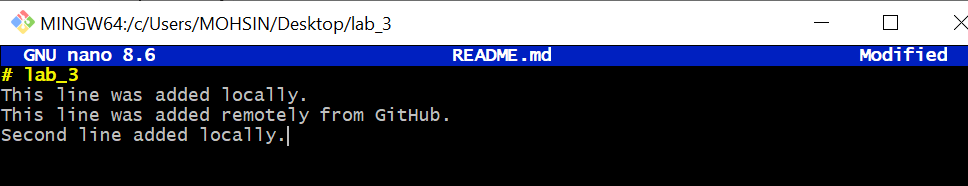
* + Save a screenshot as push\_after\_merge.png.

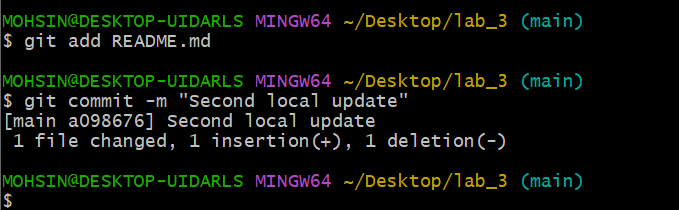


1. Now repeat the same process, but this time use **rebase instead of merge**:
   * Make another remote edit on GitHub in README.md.



* + Then make another local edit and commit it.

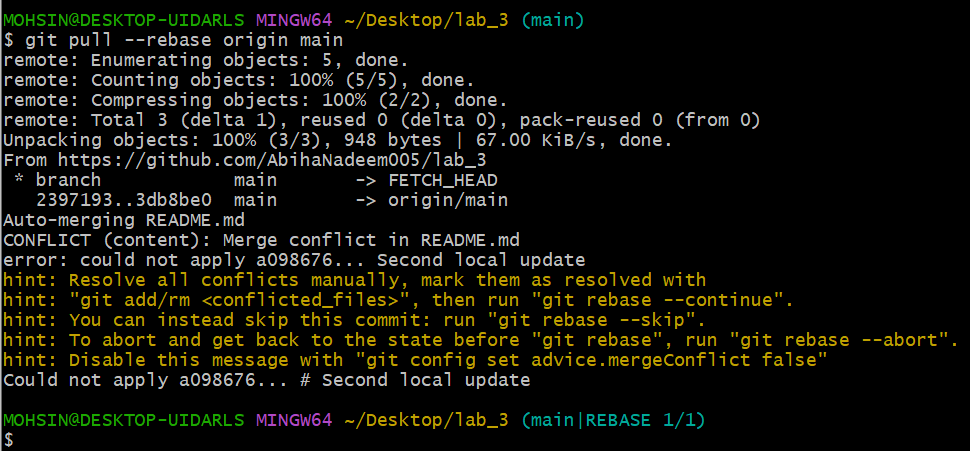


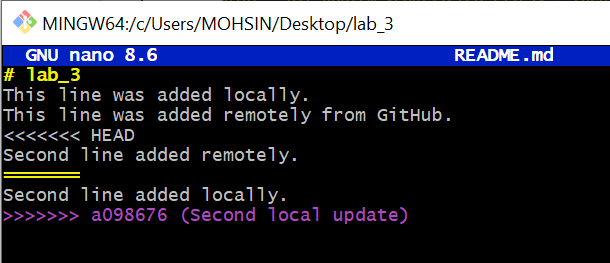


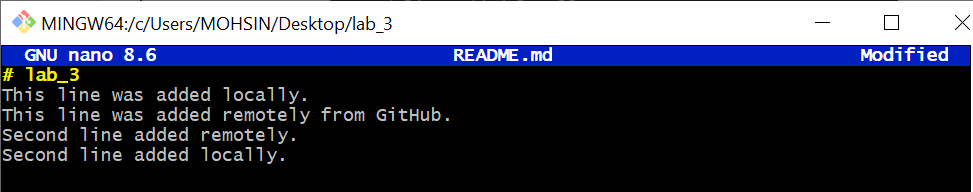
* + Instead of using git pull, run:

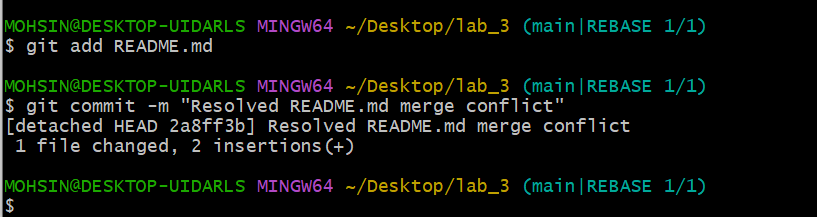
git pull --rebase origin main

* + - Git will **replay your local commits on top** of the pulled commits, keeping history linear.
    - Save a screenshot as rebase\_pull.png.





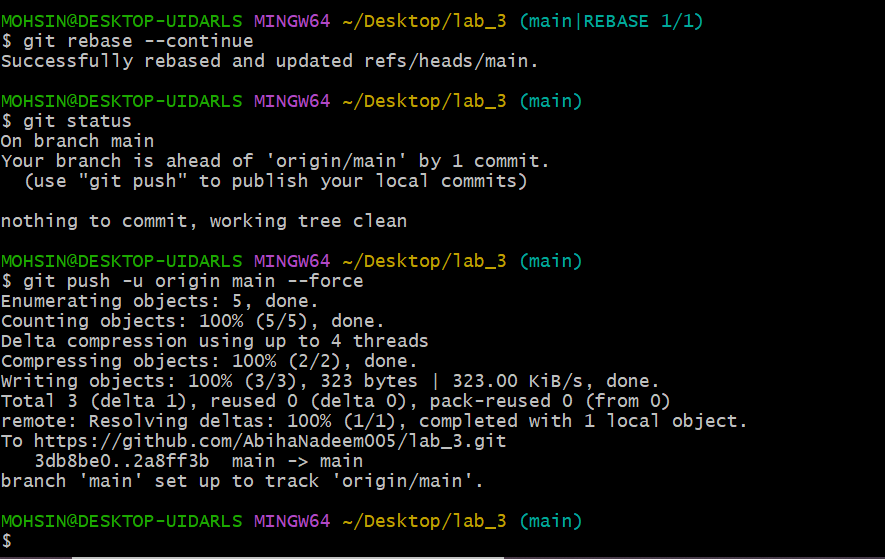




* + Push again:

git push -u origin main

* + - Save a screenshot as push\_after\_rebase.png.



**Task 2 – Creating and Resolving Merge Conflicts Manually**

This task will help you understand how Git handles file conflicts when two people modify the same line of code differently.

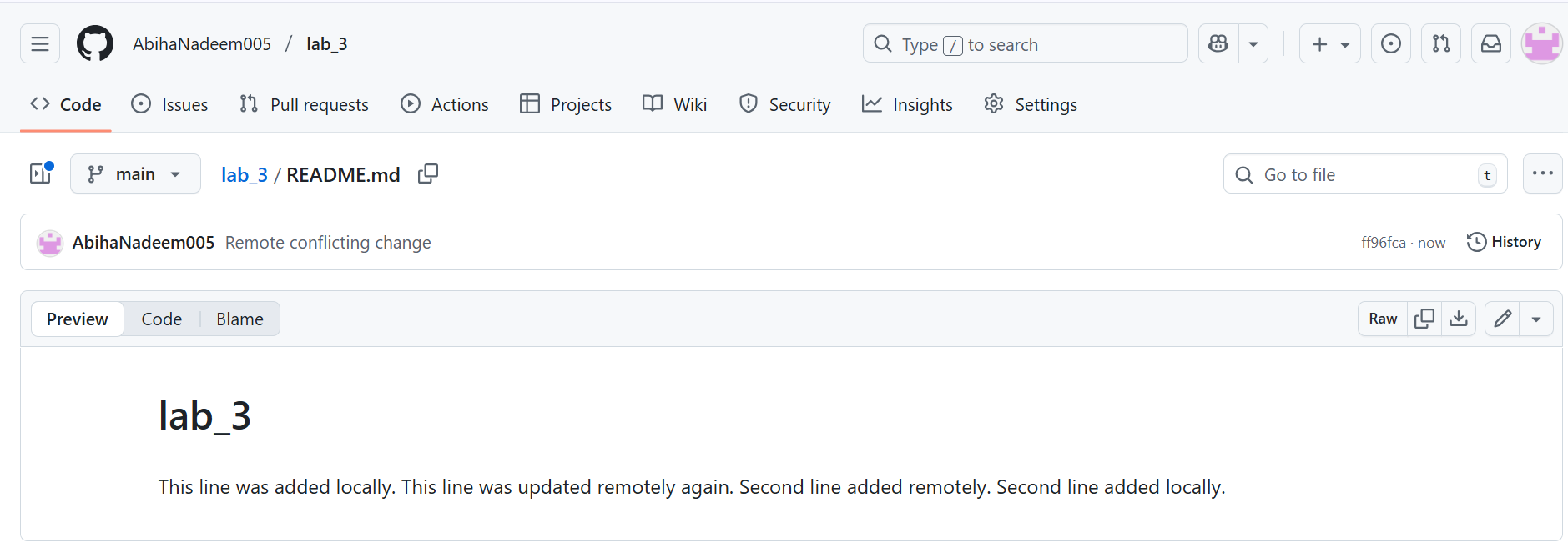
**Steps**

1. On **GitHub (remote)**, open your README.md file and change an existing line.  
   For example, if it currently says:
2. This line was added remotely from GitHub.

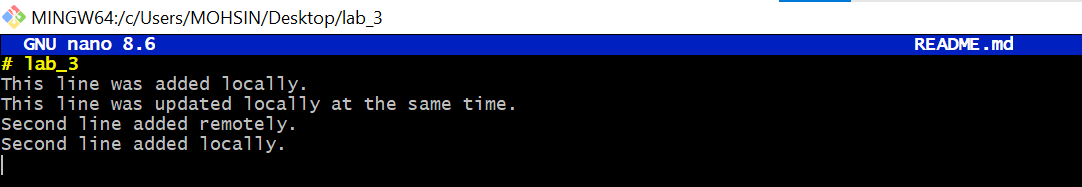
Change it to:

This line was updated remotely again.

* + Commit your change with the message: *Remote conflicting change*
  + Save a screenshot as remote\_conflict\_edit.png.



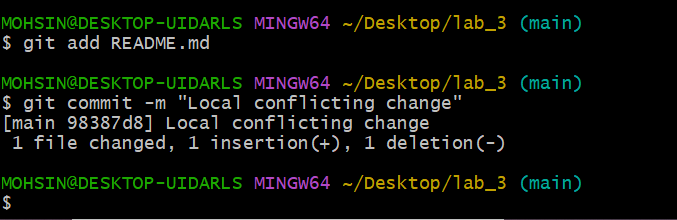
1. On your **local machine**, edit the **same line** in the same file but make a **different change**:
2. This line was updated locally at the same time.
   * Save a screenshot of your edit in VS Code as local\_conflict\_edit.png.



1. Stage and commit your local change:
2. git add README.md

git commit -m "Local conflicting change"

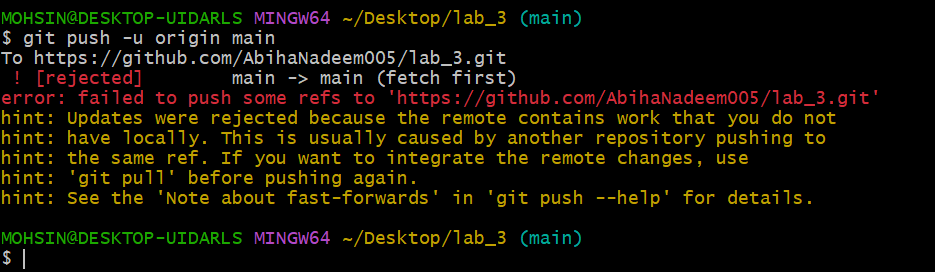
* + Save a screenshot as local\_conflict\_commit.png.



1. Try to push:

git push -u origin main

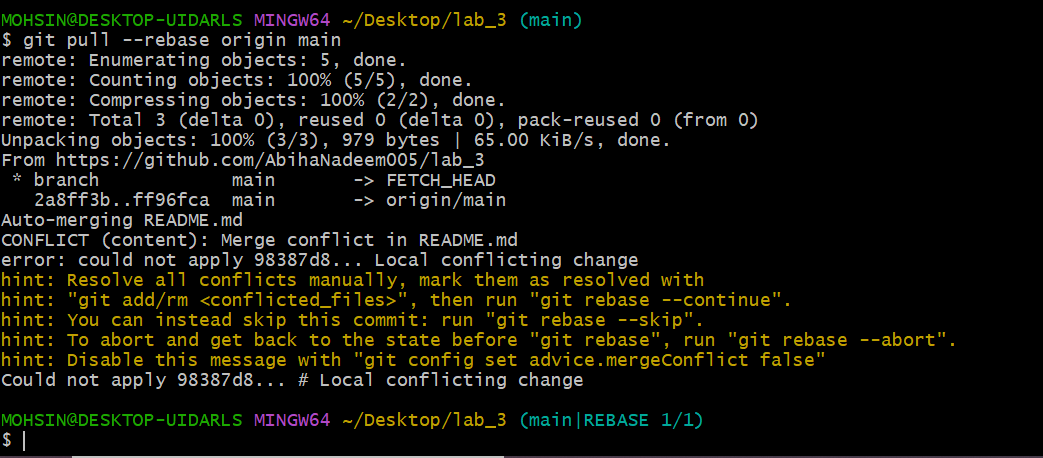
* + The push will be **rejected**, since the remote version has conflicting changes.
  + Save a screenshot of this error as conflict\_push\_error.png.



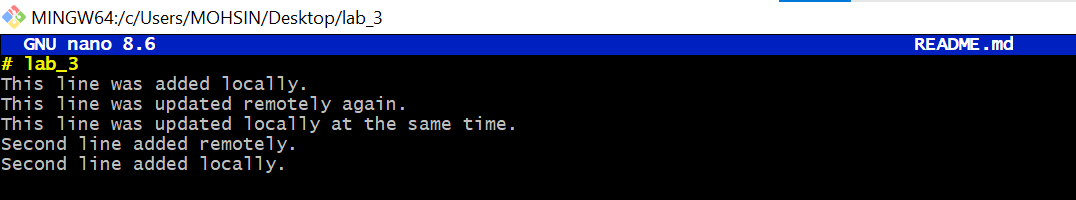
1. Pull with rebase to bring in remote changes:

git pull --rebase origin main

* + Git will **pause** the rebase and notify you about a conflict.
  + Save a screenshot of the conflict message as conflict\_message.png.



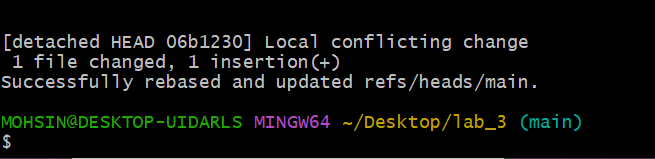
1. Open the README.md file in your editor — you’ll see conflict markers like this:
2. <<<<<<< HEAD
3. This line was updated locally at the same time.
4. =======
5. This line was updated remotely again.
6. >>>>>>> origin/main
   * Manually edit the file to keep the correct line (for example, merge both ideas).
   * Save a screenshot of the resolved file as resolved\_readme.png.



1. After fixing the file, mark the conflict as resolved and continue the rebase:
2. git add README.md

git rebase --continue

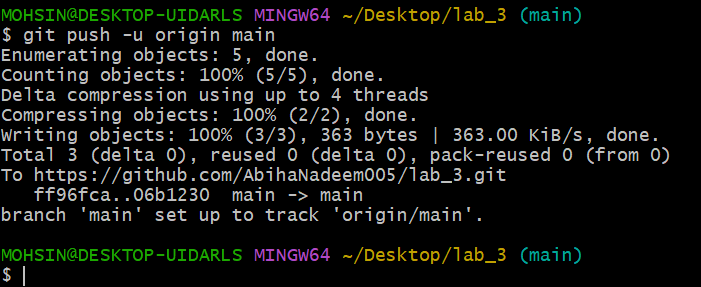
* + Save a screenshot of this step as rebase\_continue.png.



1. Finally, push your changes:

git push -u origin main

* + Save a screenshot as push\_after\_resolve.png.



**Task 3 – Managing Ignored Files with .gitignore and Removing Tracked Files**

In this task, you will learn how to use a .gitignore file to exclude files or directories from version control and how to remove them from Git’s tracking while keeping them locally.

**Steps**

1. Create a new folder named textfiles inside your repository:

mkdir textfiles

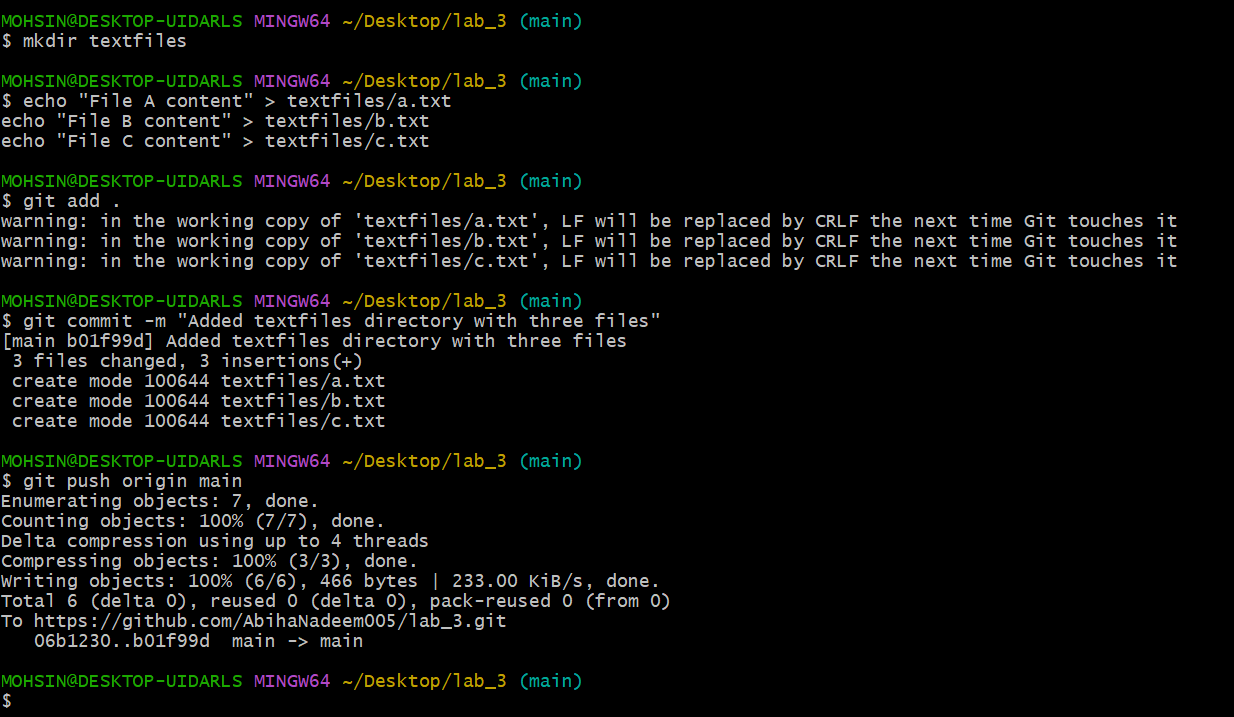
1. Inside the textfiles folder, create three text files:
2. echo "File A content" > textfiles/a.txt
3. echo "File B content" > textfiles/b.txt

echo "File C content" > textfiles/c.txt

1. Add and commit the new directory:
2. git add .
3. git commit -m "Added textfiles directory with three files"

git push origin main

* + Save a screenshot as push\_textfiles.png.



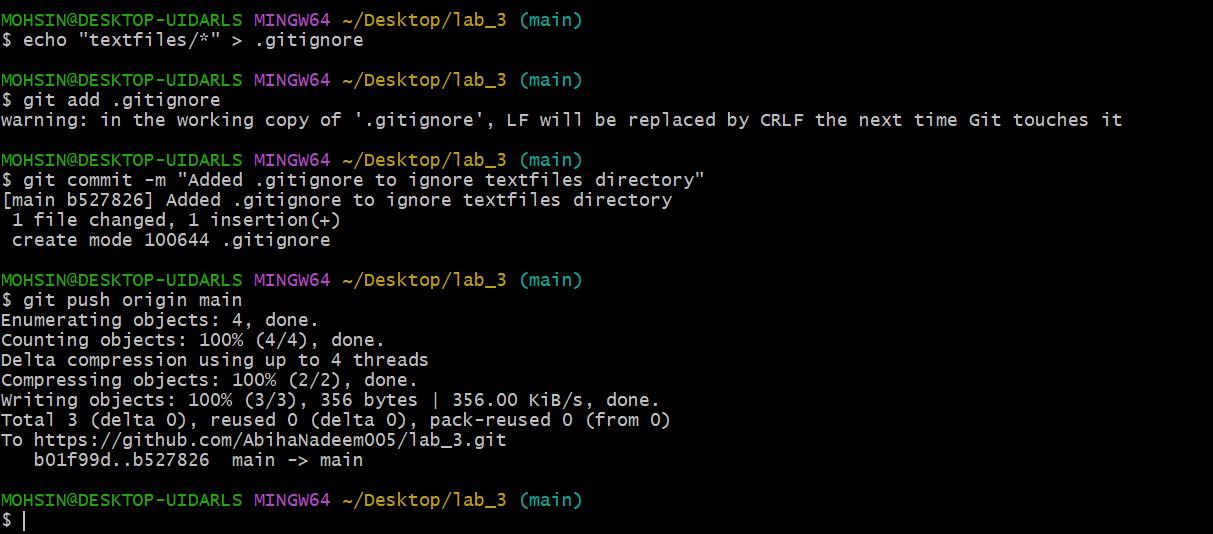
1. Now, create a .gitignore file in the root of your repository:

echo "textfiles/\*" > .gitignore

1. Add and commit the .gitignore file:
2. git add .gitignore
3. git commit -m "Added .gitignore to ignore textfiles directory"

git push origin main

* + Save a screenshot as gitignore\_push.png.



1. Go to **GitHub** and check your repository — notice that the textfiles directory is **still visible** on the remote.
   * Save a screenshot as repo\_still\_has\_textfiles.png.



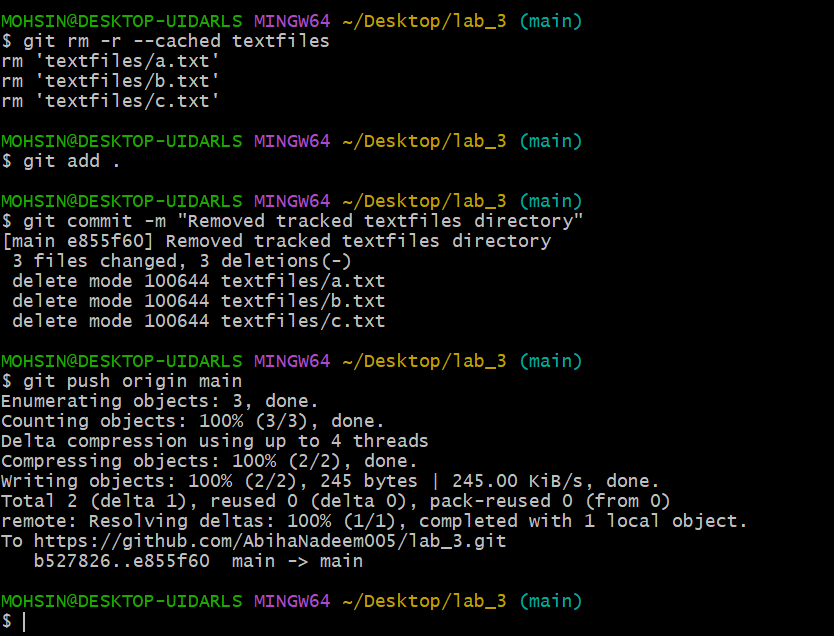
1. To remove already tracked files from Git (but not from your local system), run:

git rm -r --cached textfiles

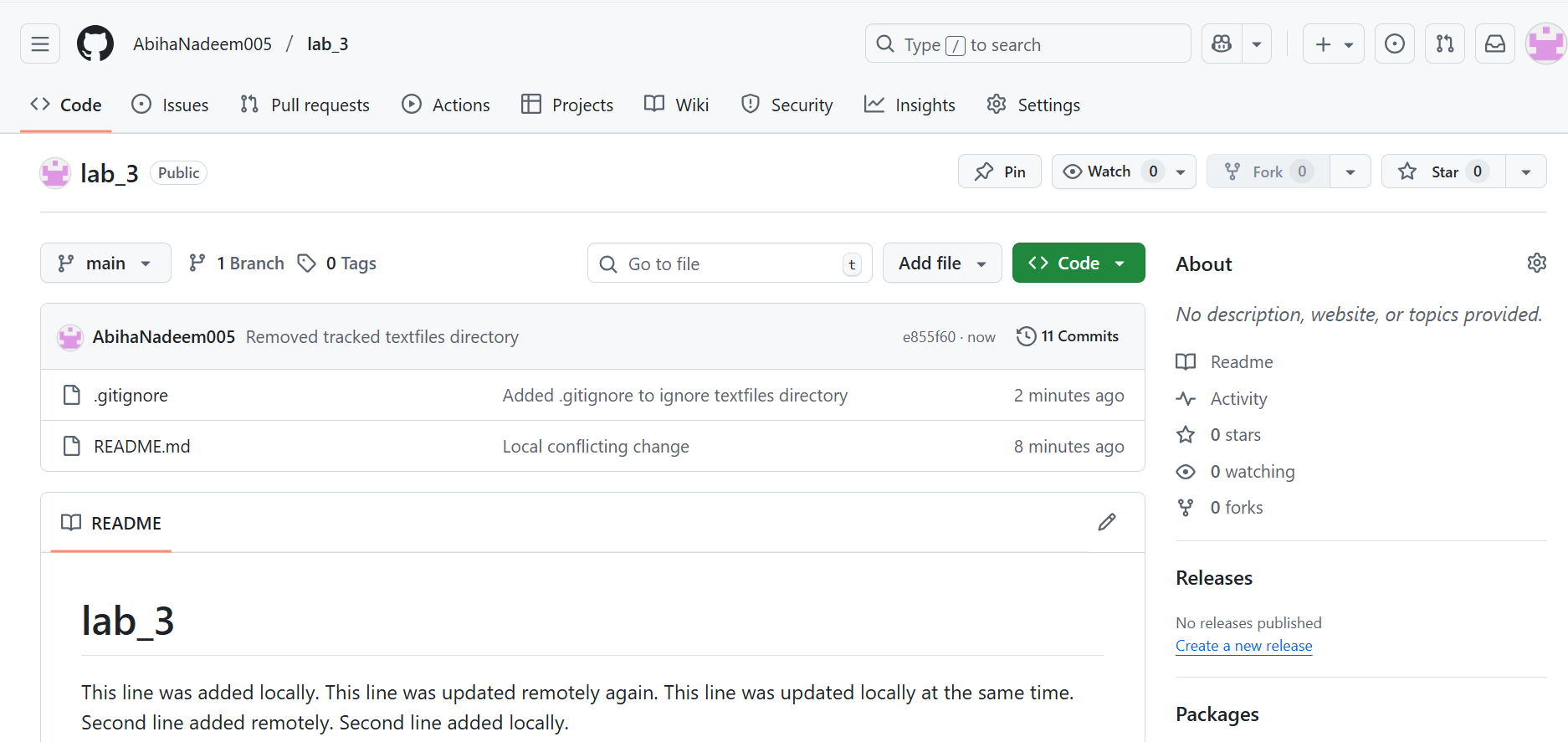
1. Commit and push again:
2. git add .
3. git commit -m "Removed tracked textfiles directory"

git push origin main

* + Save a screenshot as rm\_cached\_push.png.



1. Check your GitHub repository again — the textfiles folder should now be **deleted remotely**.
   * Save a screenshot as repo\_textfiles\_removed.png.



**Task 4 – Create Temporary Changes and Use git stash**

This task demonstrates how git stash allows you to temporarily save uncommitted changes so that you can safely switch branches without losing work.

**Steps**

1. Create a feature-branch.

git checkout -b feature-branch

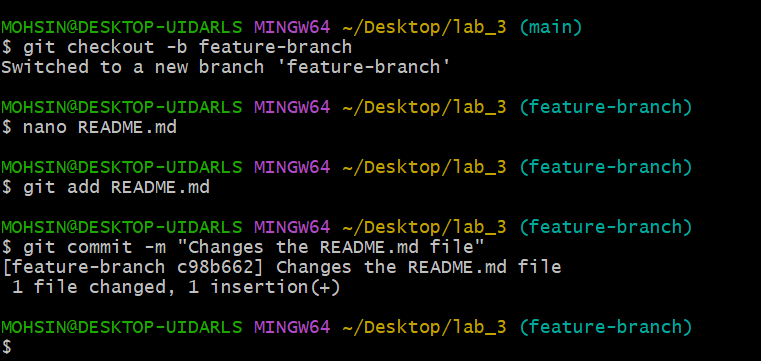
1. Modify any file (for example, README.md) by adding a few test lines.



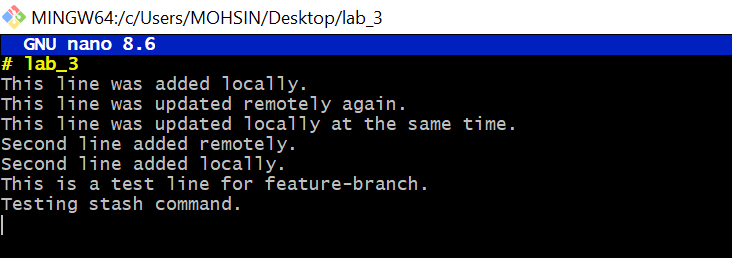
1. git add README.md

git commit -m "Changes the README.md file"

* + Save a screenshot as modified\_readme.png.



1. Modify any file (for example, README.md) by adding a few new lines.



1. Without committing or stashing the changes, try to switch to another branch:

git checkout main

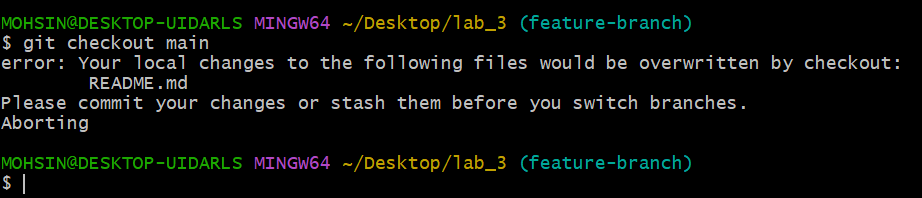
You’ll see an error message similar to:

error: Your local changes to the following files would be overwritten by checkout:

README.md

Please commit your changes or stash them before you switch branches.

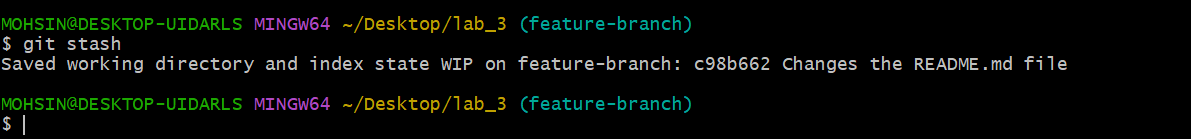
* + Save a screenshot as checkout\_error.png.



1. To fix this, **stash your changes**:

git stash

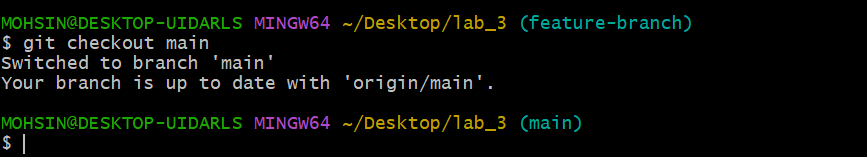
* + Save a screenshot as stash\_command.png.



1. Try switching branches again — it should now work:

git checkout main

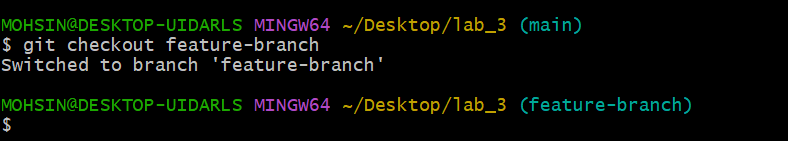
* + Save a screenshot as branch\_switched.png.



1. Return to your previous branch (for example, feature-branch):

git checkout feature-branch

* + Save a screenshot as back\_to\_feature.png.



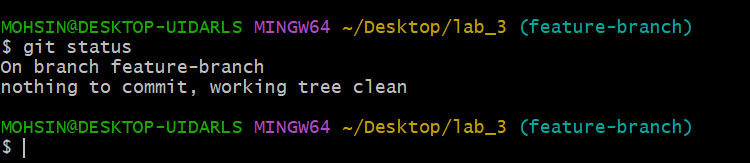
1. Check your working directory status:

git status

It should show:

nothing to commit, working tree clean

* + Save a screenshot as status\_clean.png.



1. Now restore your stashed changes:

git stash pop

You’ll see a message indicating that changes were reapplied:

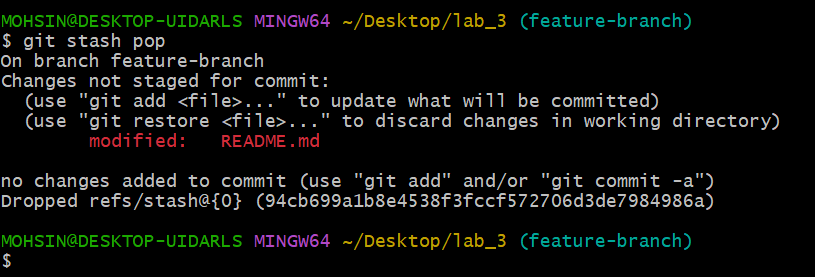
On branch feature-branch

Changes not staged for commit:

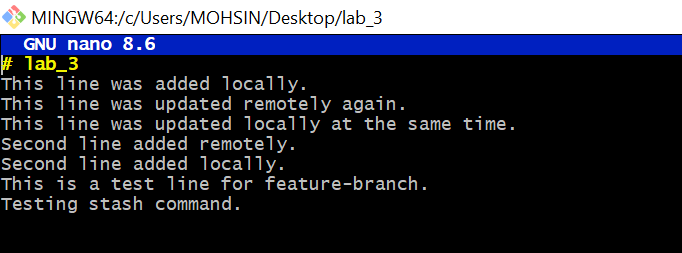
(use "git add <file>..." to update what will be committed)

(use "git restore <file>..." to discard changes in working directory)

* + Save a screenshot as stash\_pop.png.



1. Confirm that your previous edits are back in the file.

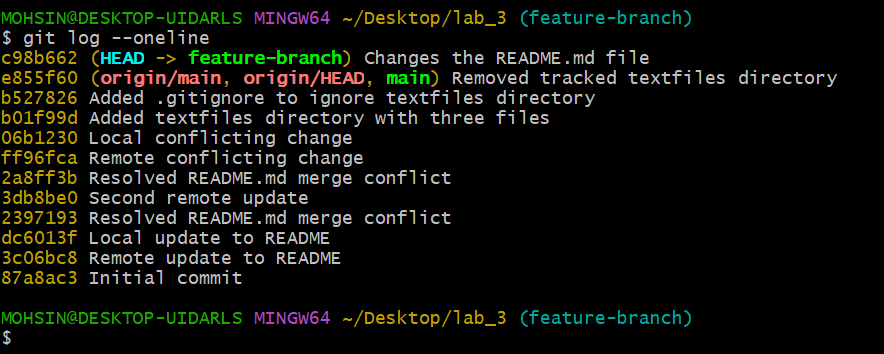


**Task 5 – Checkout a Specific Commit Using git log**

1. View commit history:

git log --oneline

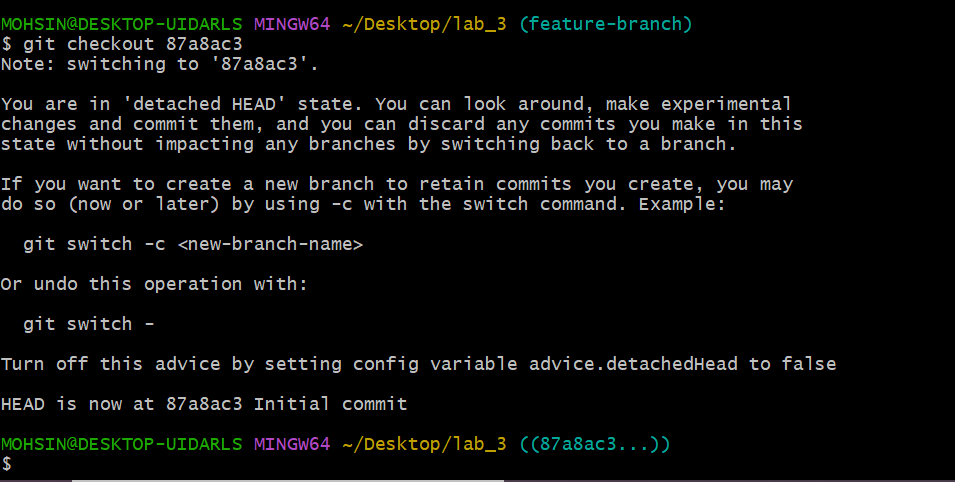
* + Save a screenshot as log\_before\_checkout.png.



1. Copy any previous commit hash (e.g., a1b2c3d).
2. Checkout that commit:

git checkout <commit-hash>

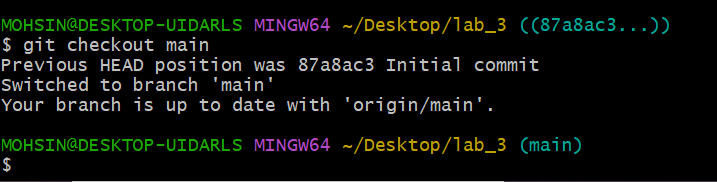
* + Save a screenshot as detached\_head.png.



1. To return to your main branch:

git checkout main

* + Save a screenshot as back\_to\_main.png.

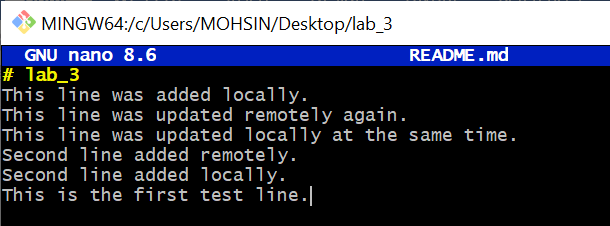


**Task 6 – Resetting Commits (Soft vs Hard Reset) (With Verification Steps)**

This task demonstrates the difference between a soft and hard reset in Git. In this version, you will **verify the presence of changes in files and confirm the existence of commits after each reset**.

**Steps**

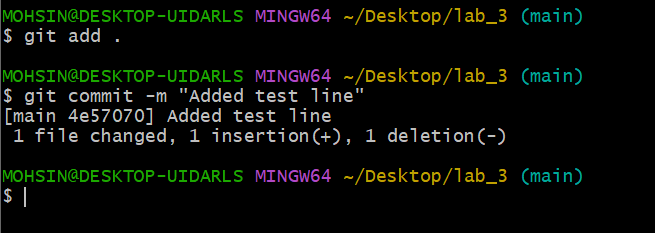
1. **Add a new line in any file and commit it:**
2. # Edit any file (e.g., README.md), add a line



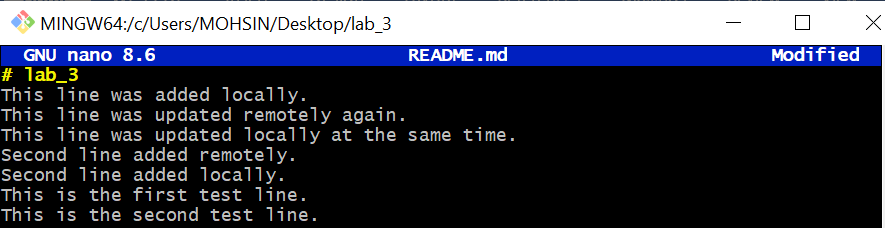
1. git add .

git commit -m "Added test line"

* + Save a screenshot as first\_commit.png.



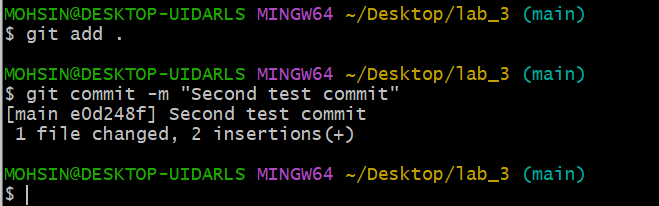
1. **Add another change and commit again:**
2. # Edit the file again, add a different line



1. git add .

git commit -m "Second test commit"

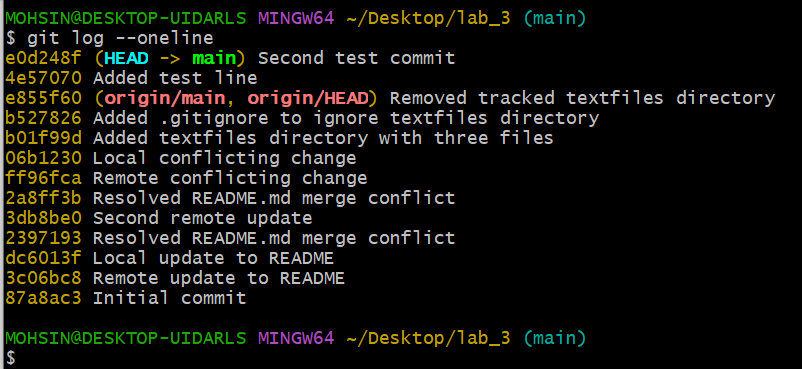
* + Save a screenshot as second\_commit.png.



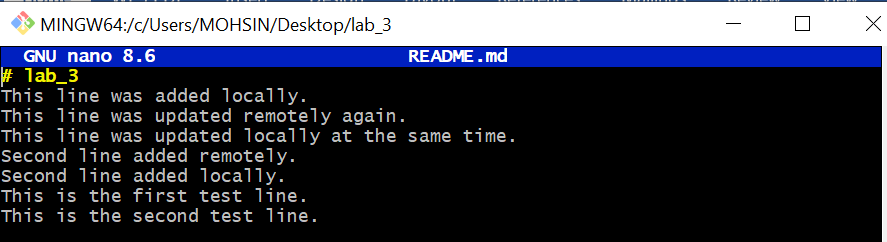
1. **View the history before reset:**

git log --oneline

* + Save a screenshot as log\_before\_reset.png.



1. **Check file contents for both changes:**
   * Open the edited file (e.g., README.md) and visually confirm BOTH added lines exist.
   * Save a screenshot as file\_before\_reset.png.



1. **Perform a soft reset (keeps changes in working directory):**

git reset --soft HEAD~1

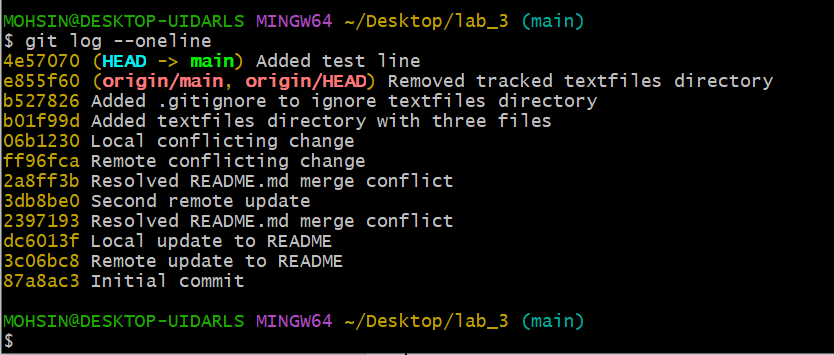
* + Save a screenshot as soft\_reset.png.



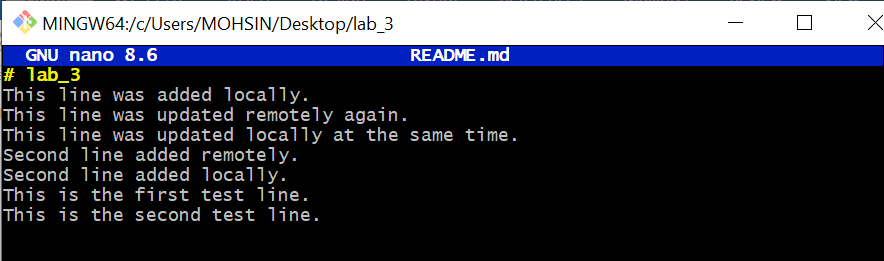
1. **Check commit history after soft reset:**

git log --oneline

* + Save a screenshot as log\_after\_soft\_reset.png.



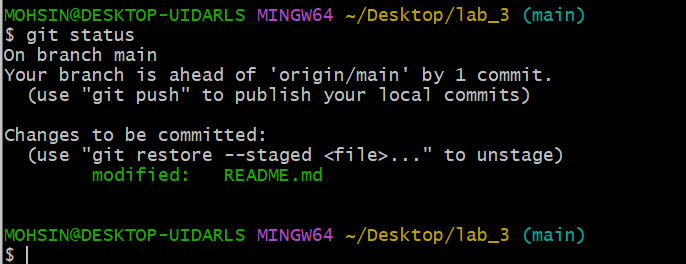
1. **Verify changes in the file:**
   * Open the file again and confirm both edits are still present (nothing lost).
   * Save a screenshot as file\_after\_soft\_reset.png.



1. **Check git status:**

git status

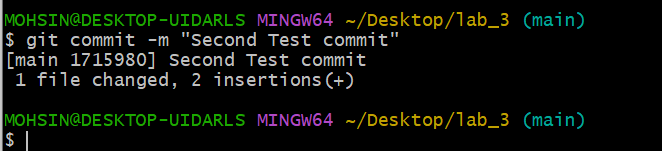
* + You should see your changes are staged (ready to commit again).



1. **Perform commit**

git commit -m "Second Test commit"

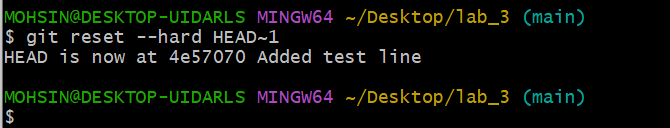
* + Save a screenshot as file\_after\_hard\_reset.png.



1. **Perform a hard reset (discards changes completely):**

git reset --hard HEAD~1

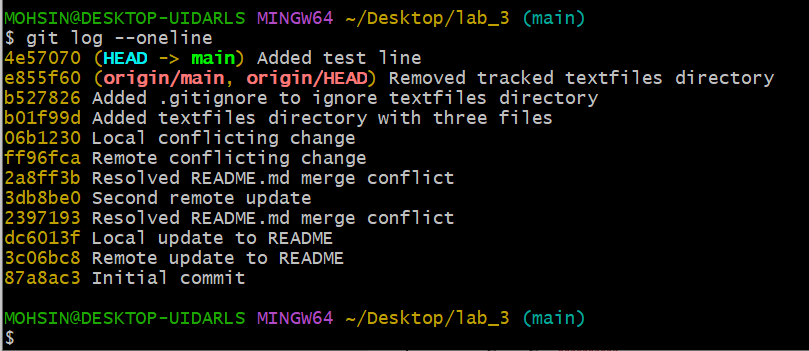
* Save a screenshot as hard\_reset.png.



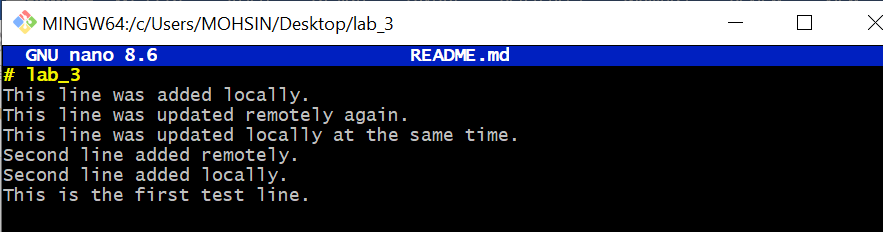
1. **Check commit history after hard reset:**

git log --oneline

* + Save a screenshot as log\_after\_hard\_reset.png.



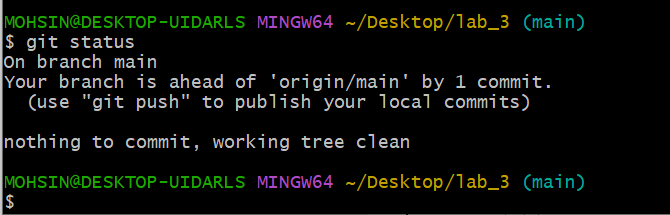
1. **Verify changes in the file:**
   * Open the file and confirm the **second edit is gone** (reverted to previous commit).
   * Save a screenshot as file\_after\_hard\_reset.png.



1. **Check git status:**

git status

* + Should report "nothing to commit, working tree clean".



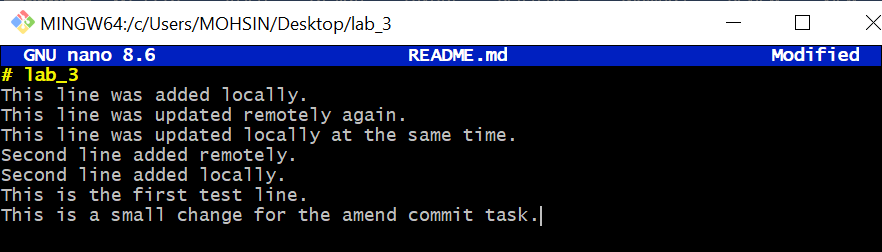
**Task 7 – Amending the Last Commit**

1. Make a small change in any file.
2. Stage it and commit:

git add .

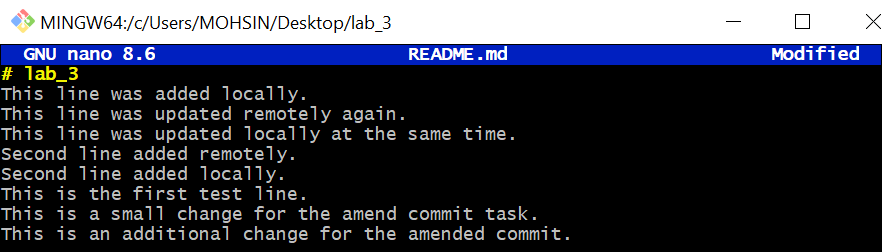
git commit -m "Fix log message"

* + Save a screenshot as first\_amend\_commit.png.





1. Realize you forgot to change another file. Update it now.

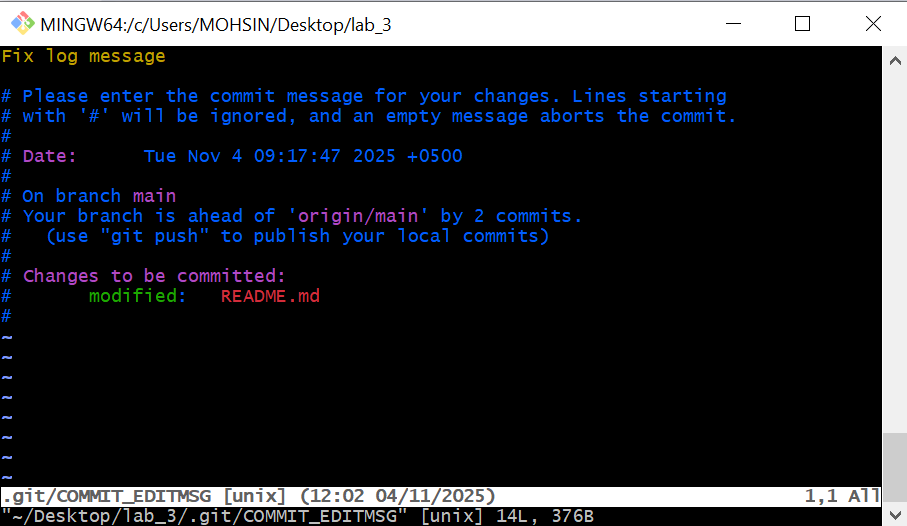


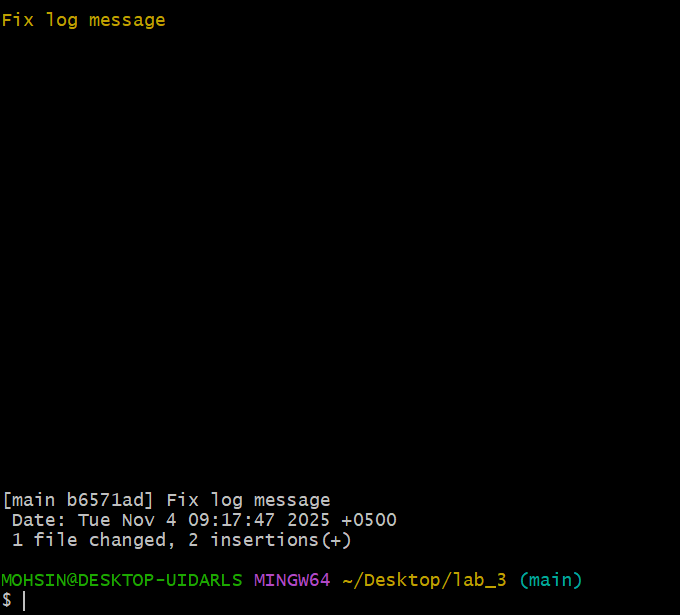
1. Add it and amend the last commit:

git add .

git commit --amend

* + Save a screenshot as amend\_commit.png.





**Task 8 – Reverting a Commit (Safe Undo on Remote Branch)**

1. Make a change and commit it:

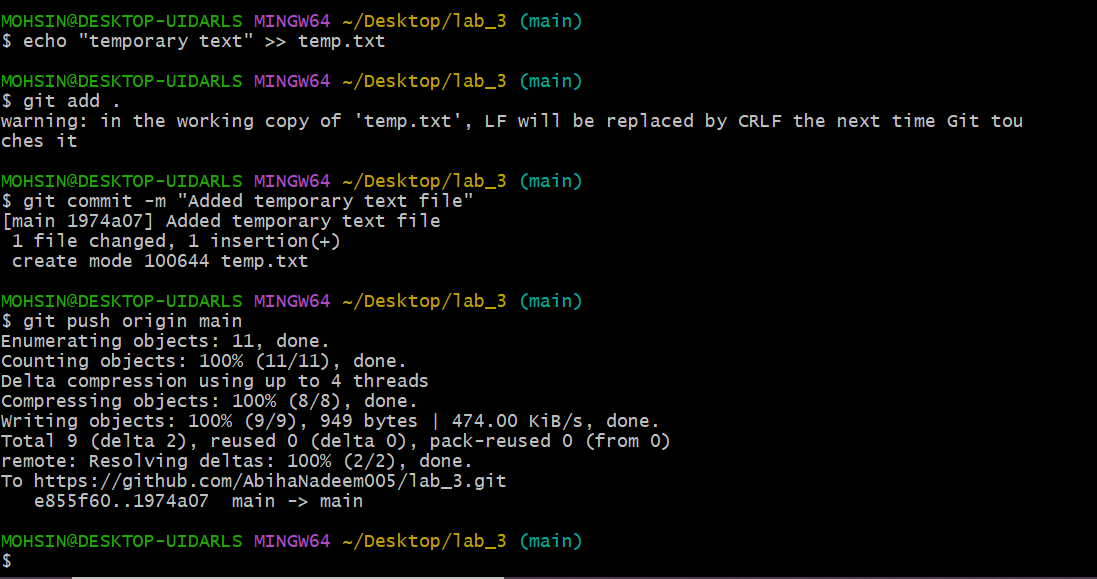
echo "temporary text" >> temp.txt

git add .

git commit -m "Added temporary text file"

git push origin main

* + Save a screenshot as commit\_temp\_file.png.

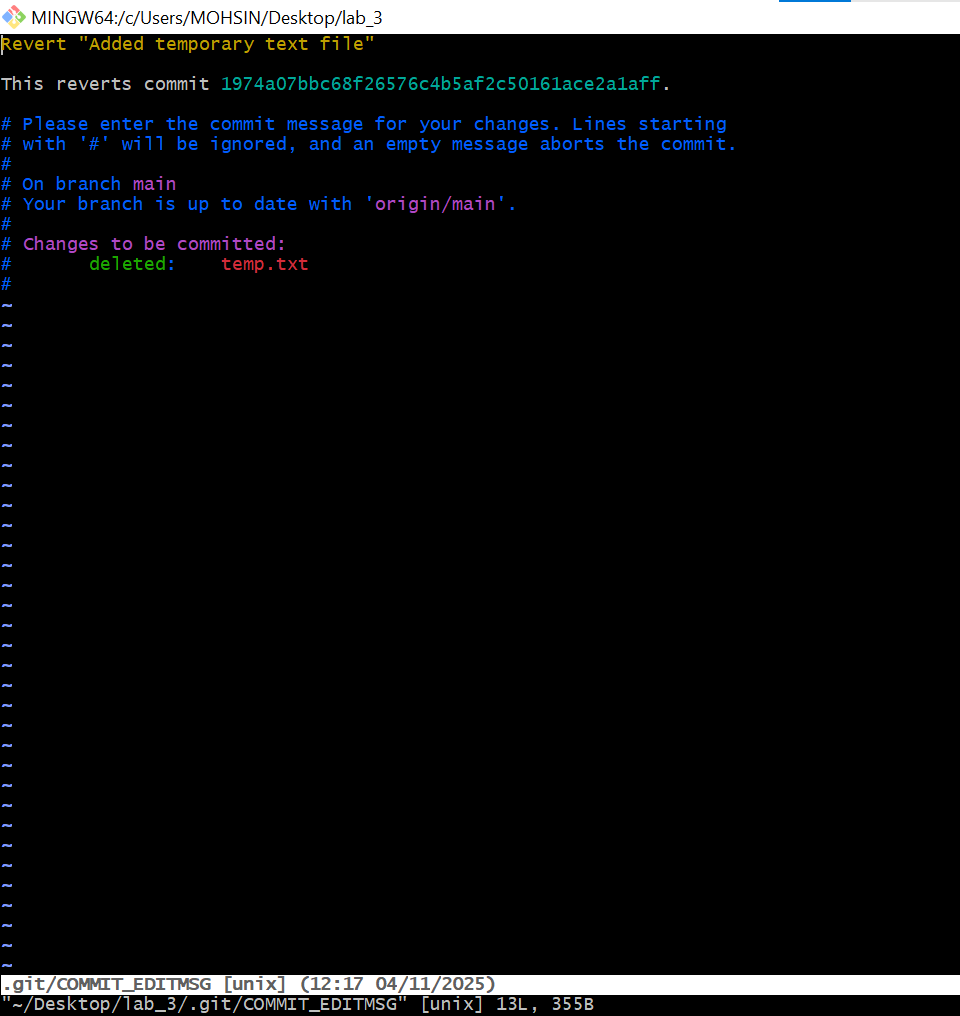


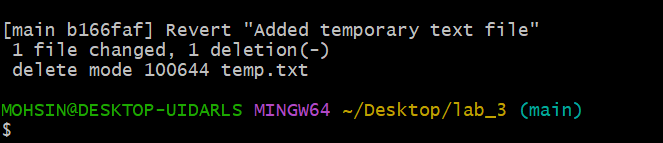
1. Now revert that commit safely (do not delete it):

git log --oneline

git revert <commit-hash>

* + Save a screenshot as revert\_commit.png.

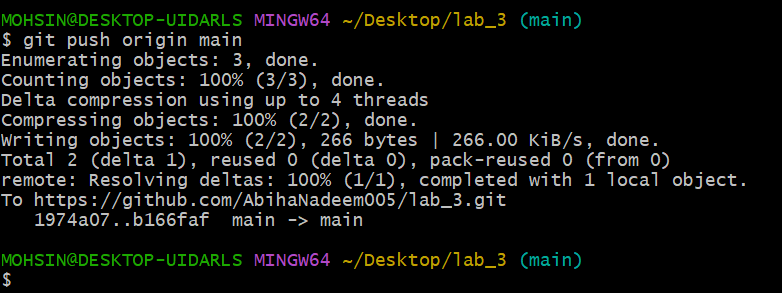




1. Push the revert commit:

git push origin main

* + Save a screenshot as revert\_push.png.



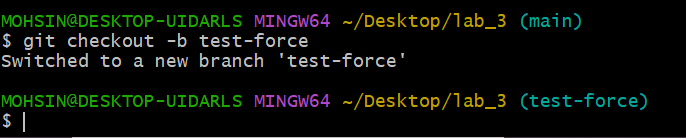
**Task 9 – Force Push (With Caution)**

⚠️ Use this only in your own branch (never on main or develop).

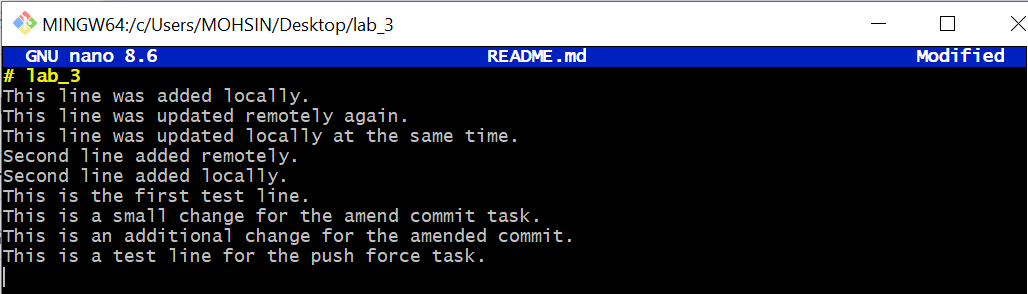
1. Create a new branch:

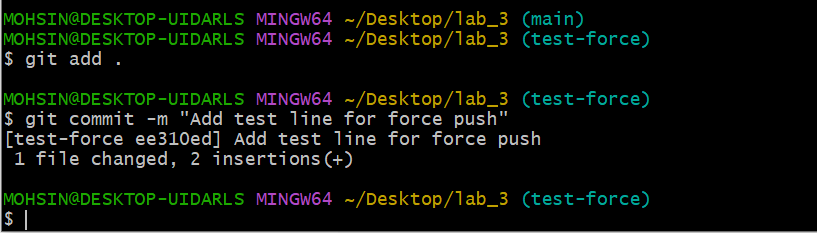
git checkout -b test-force

* + Save a screenshot as new\_branch.png.



1. Make and commit a small change.
   * Save a screenshot as force\_commit.png.

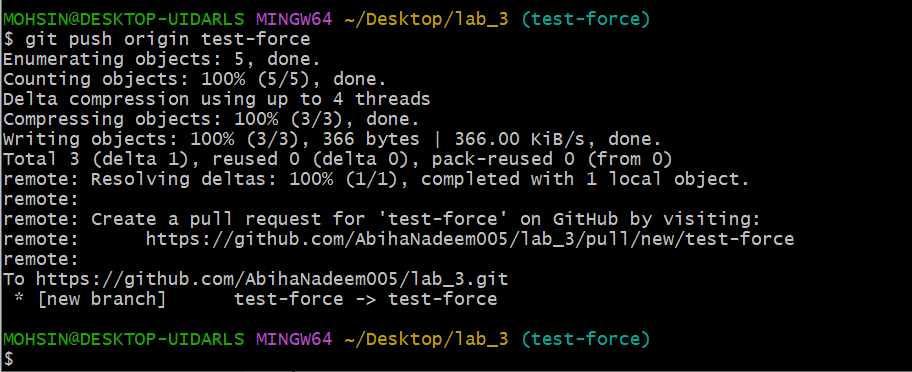




1. Push it:

git push origin test-force

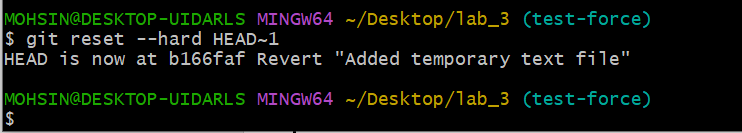
* + Save a screenshot as push\_force\_branch.png.



1. Perform a hard reset:

git reset --hard HEAD~1

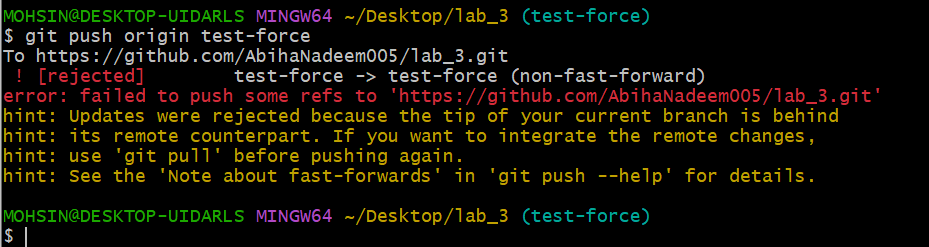
* + Save a screenshot as hard\_reset\_force.png.



1. Push again

git push origin test-force

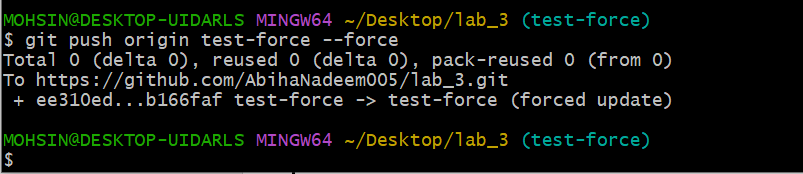
* + Rejected by remote repository
  + Save a screenshot as normal\_push.png.

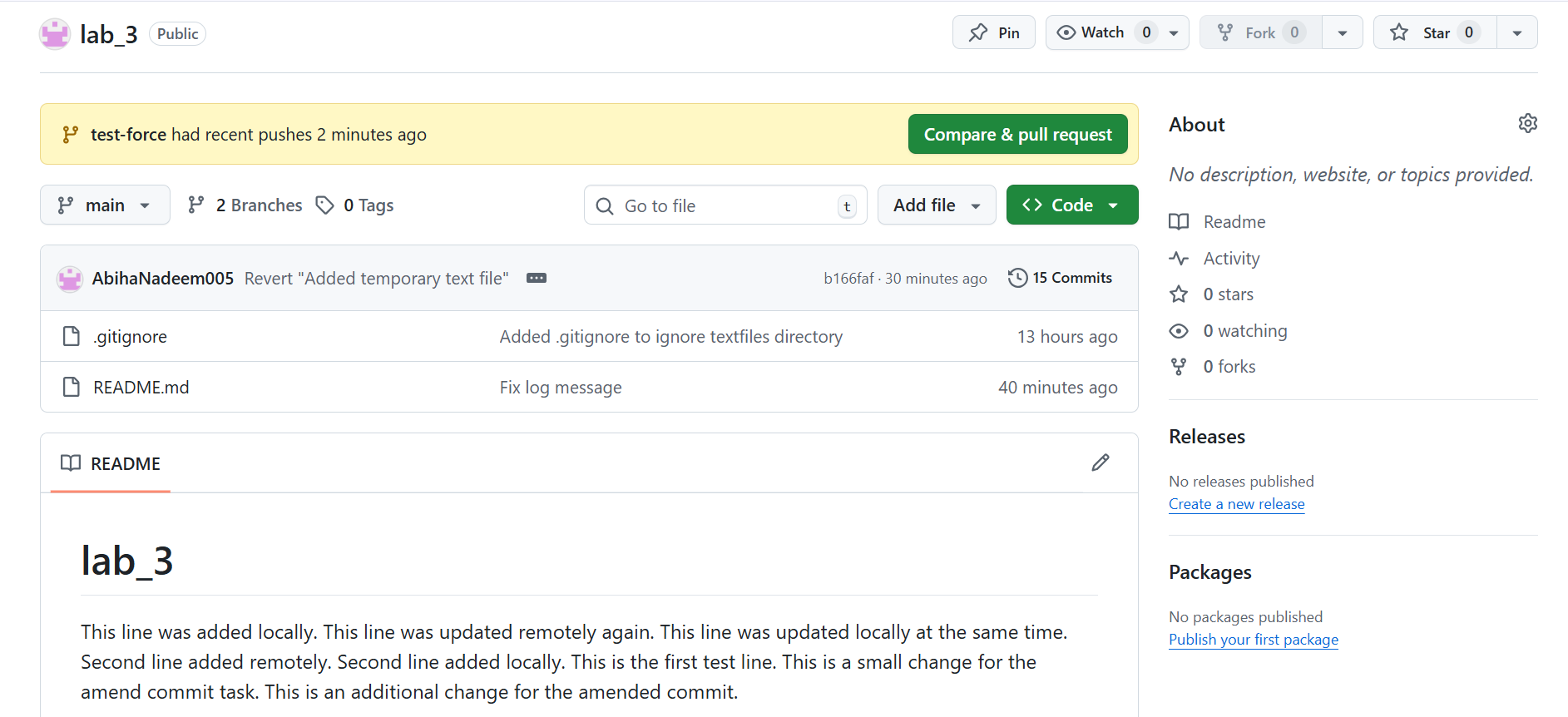


1. Now force-push to remote:

git push origin test-force --force

* + Save a screenshot as force\_push.png.





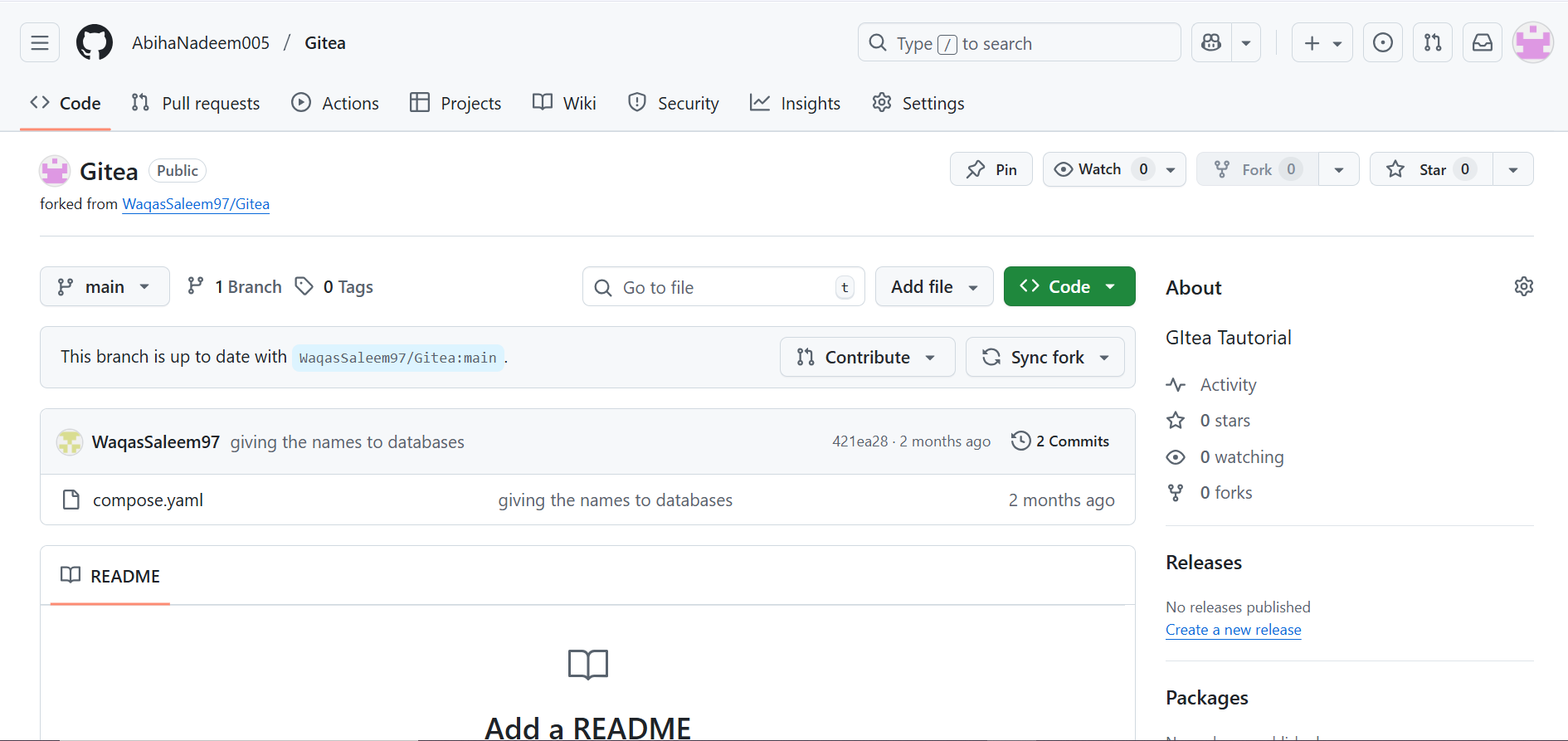
Link: <https://github.com/AbihaNadeem005/lab_3>

**Task 10 – Running Gitea in GitHub Codespaces via Docker Compose**

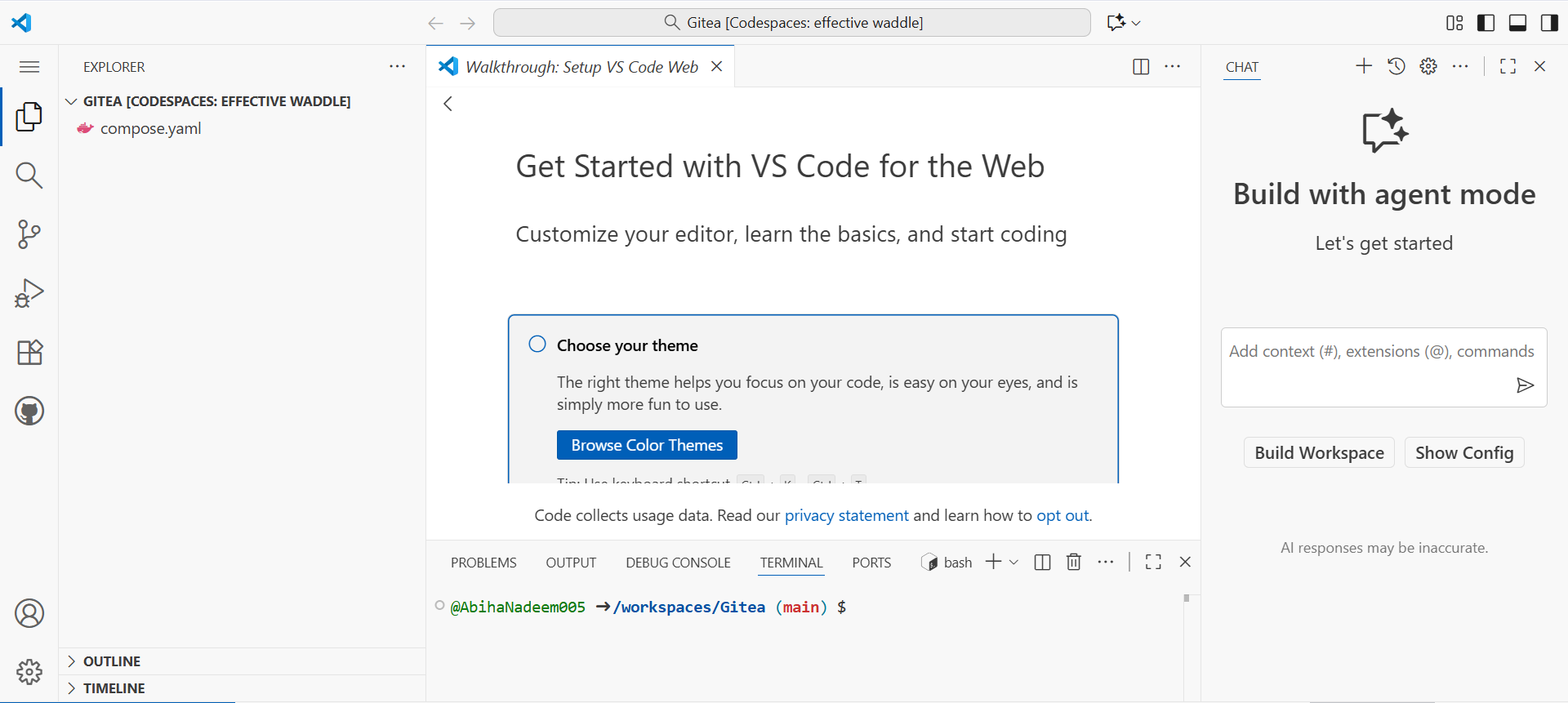
This task will guide you through forking a repository, running a full web application in a GitHub Codespace, and interacting with Gitea (a self-hosted Git service) inside Codespaces.

**Steps**

1. **Fork the Gitea Repository**
   * Go to [Gitea](https://github.com/WaqasSaleem97/Gitea).
   * Click the **Fork** button at the top right and fork it into your own GitHub account.
   * Save a screenshot of your forked repository page as forked\_gitea.png.



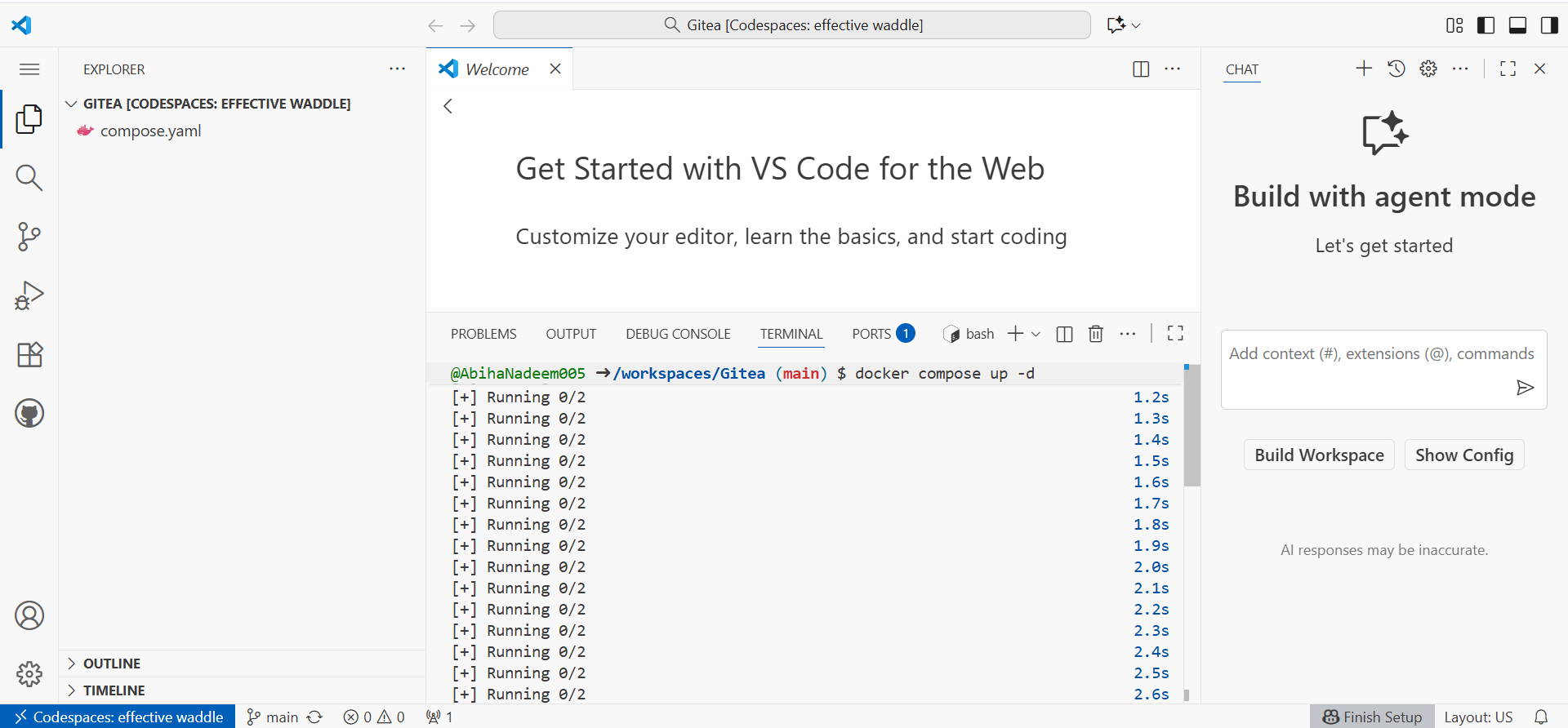
1. **Open the Forked Repo in GitHub Codespaces**
   * On your forked repo, click **Code** > **Codespaces** > **Create codespace on main**.
   * Save a screenshot of the Codespace loading as codespace\_loading.png.

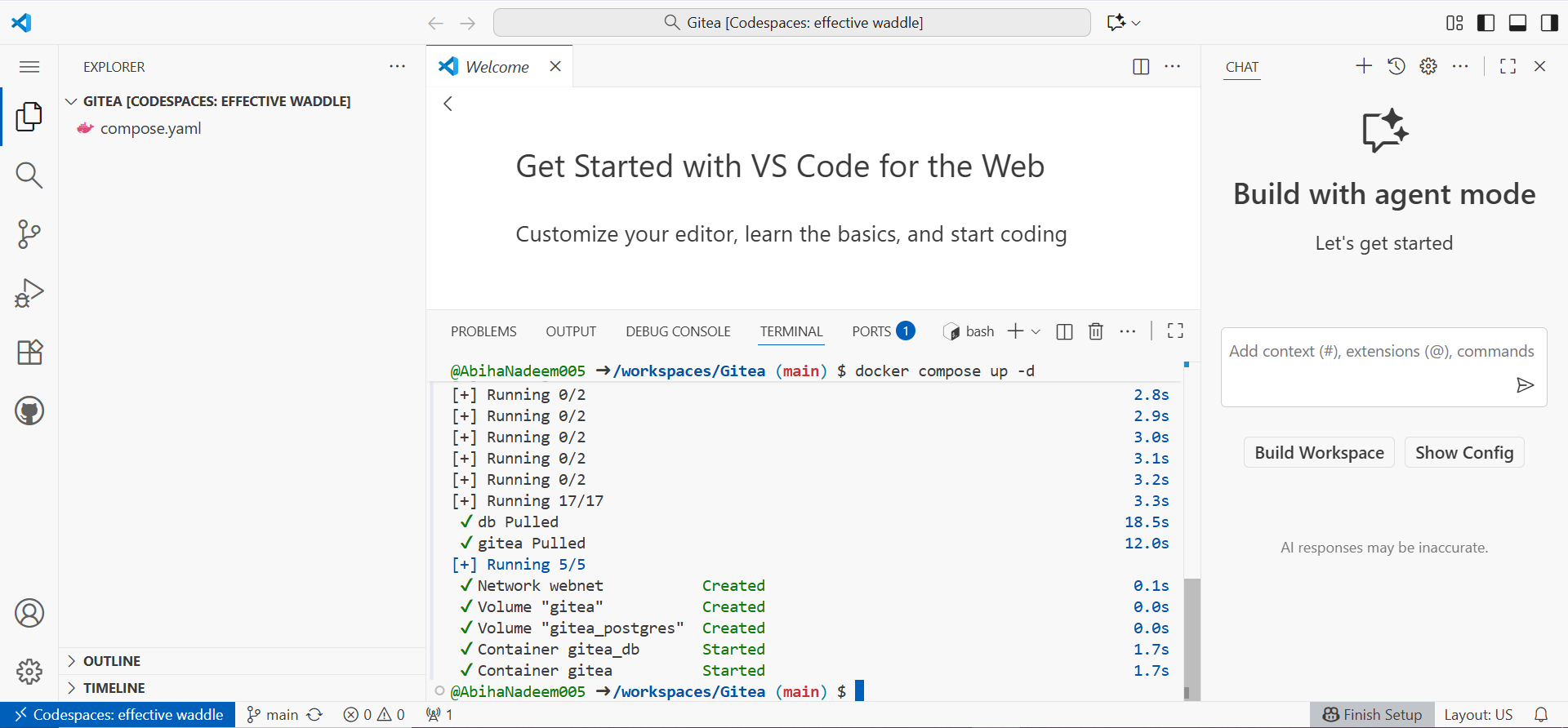


1. **Start Gitea with Docker Compose**
   * In the Codespace terminal, run:

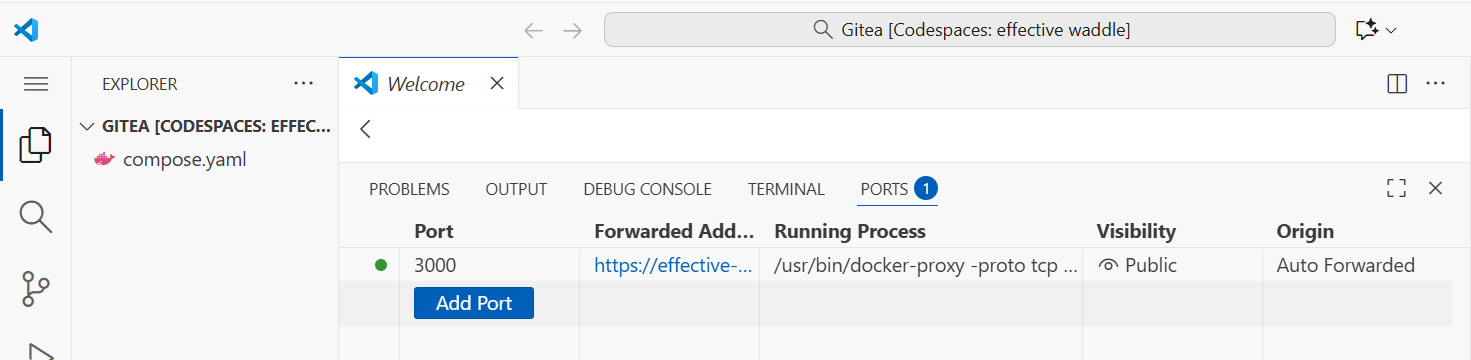
docker compose up -d

* + Wait for the containers to start.
  + Save a screenshot of the terminal showing the containers running as docker\_up.png.

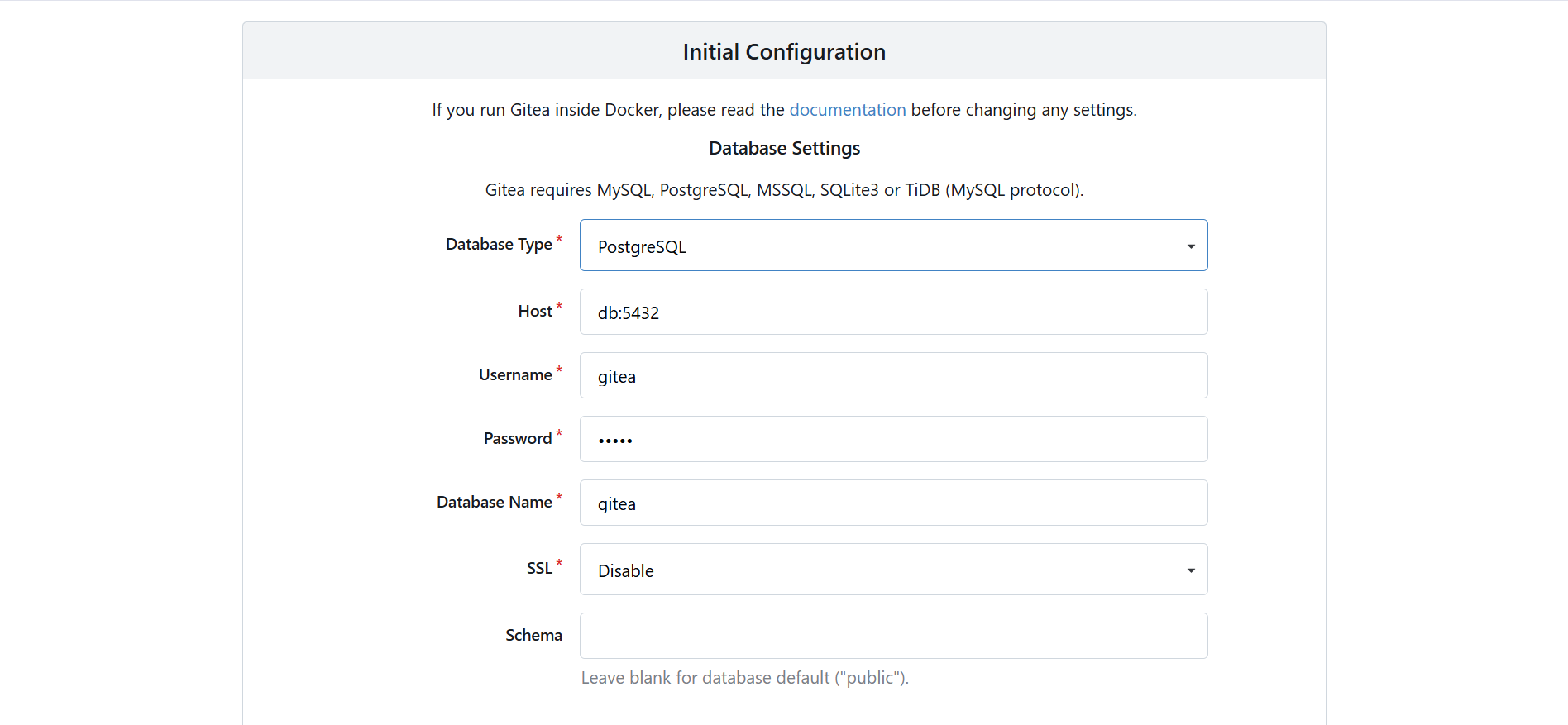




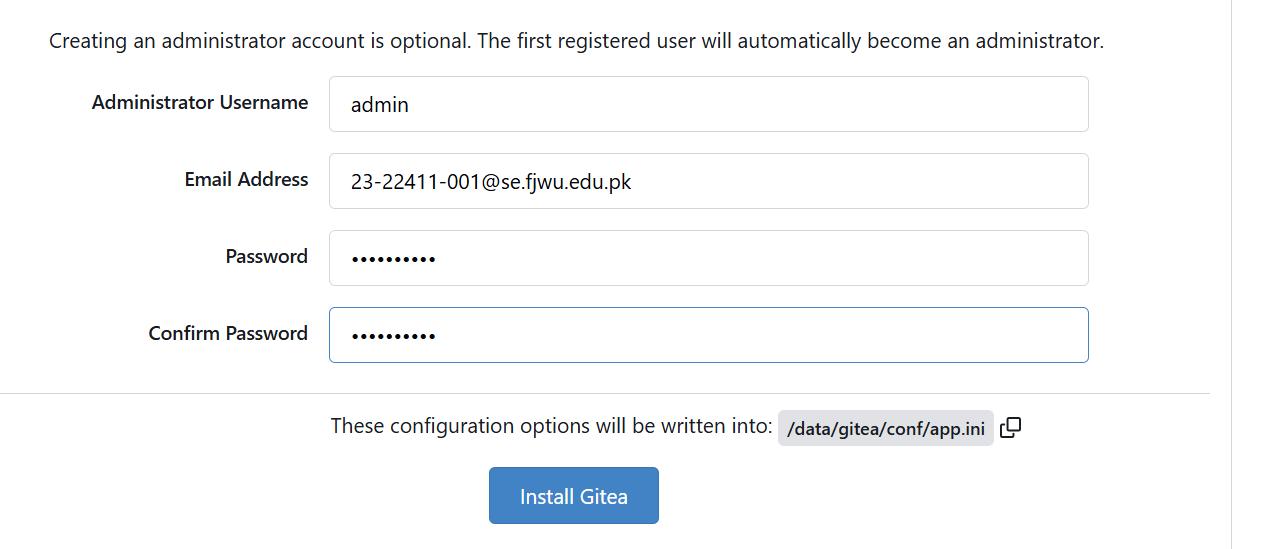
1. **Access Gitea Web Interface**
   * In Codespaces, forward **port 3000** (see Codespaces port forwarding UI).
   * Change the Visibility of **port 3000** from **private** to **public**.



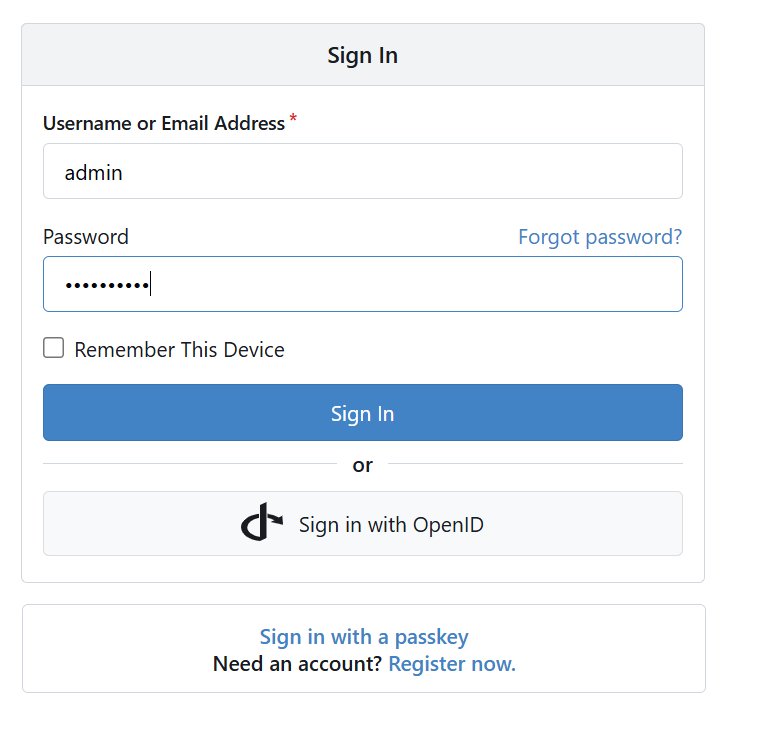
* + Click the forwarded port 3000 link to open Gitea in your browser tab.
  + Save a screenshot of the Gitea install page as gitea\_install\_page.png.



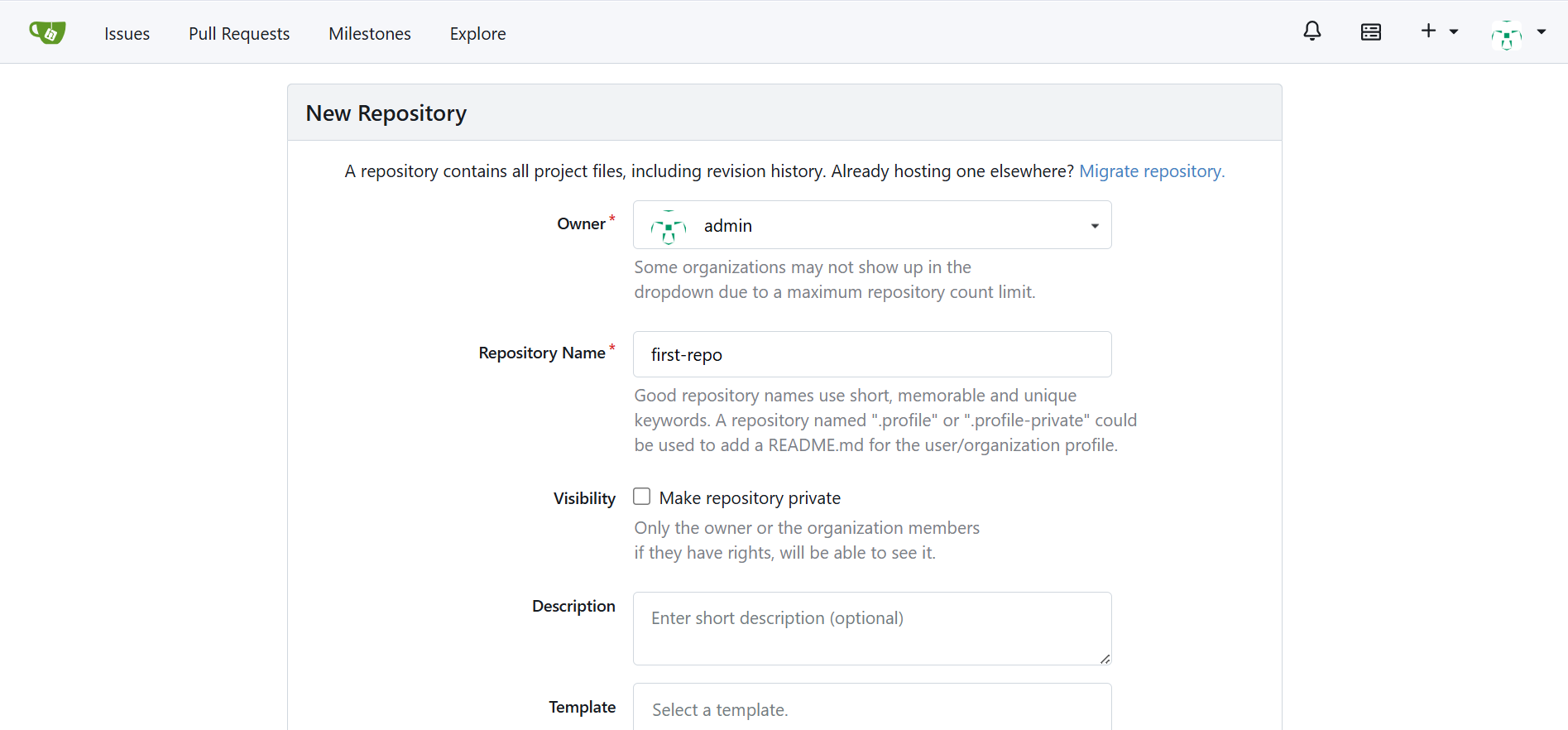
1. **Install Gitea**
   * Fill out the installation form, providing an **admin username** and **password** (e.g., admin / yourpassword).
   * Save a screenshot of the completed setup (show the admin account setup) as admin\_setup.png.

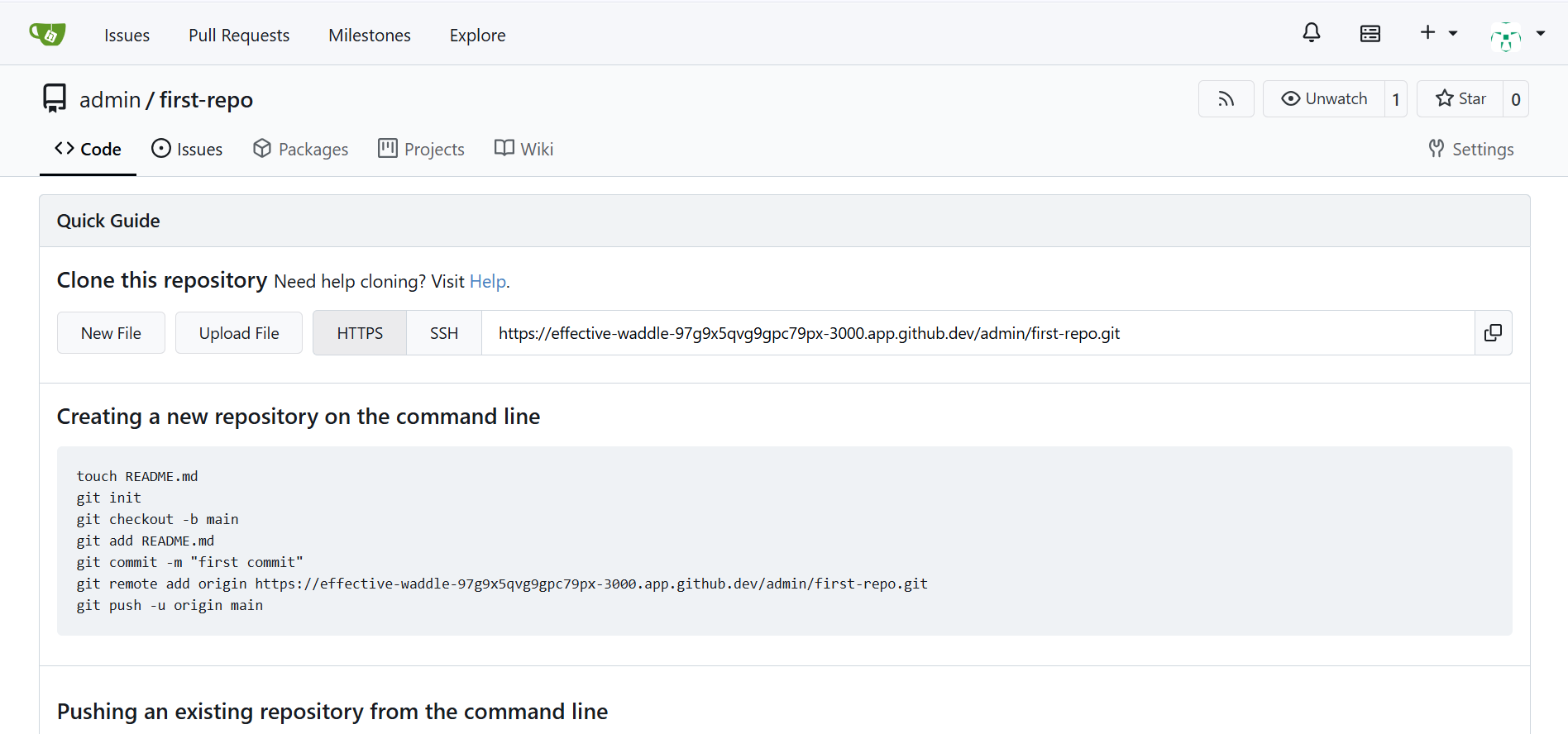


1. **Log In to Gitea**
   * Use your admin credentials to log in.
   * Save a screenshot of the Gitea dashboard after login as gitea\_dashboard.png.



1. **Create a New Repository in Gitea**
   * Click **New Repository** or **+** > **New Repository** in Gitea.
   * Fill out the repository details and create it.
   * Save a screenshot of the new repository page as gitea\_new\_repo.png.





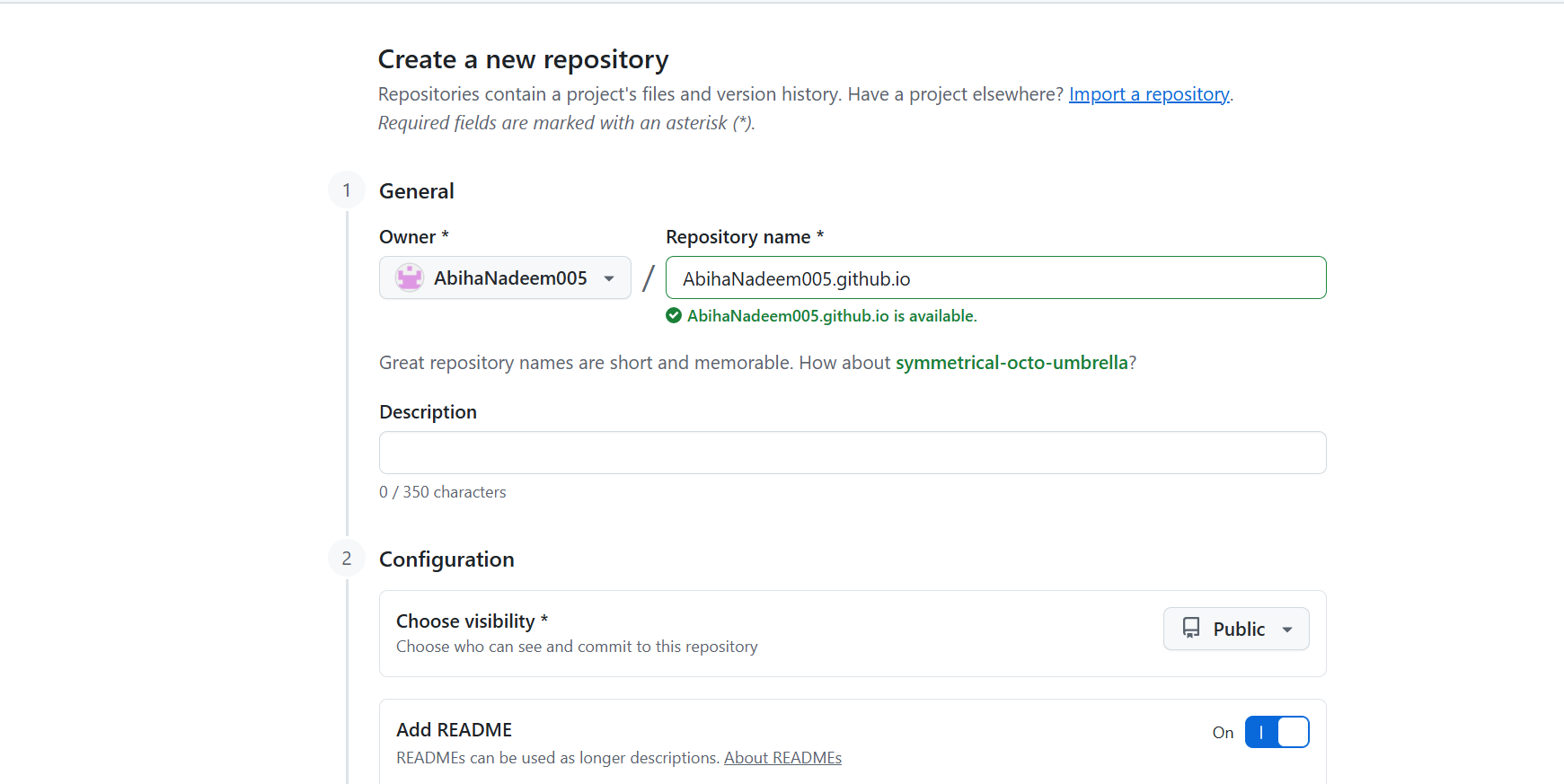
Link: <https://effective-waddle-97g9x5qvg9gpc79px-3000.app.github.dev/admin/first-repo>

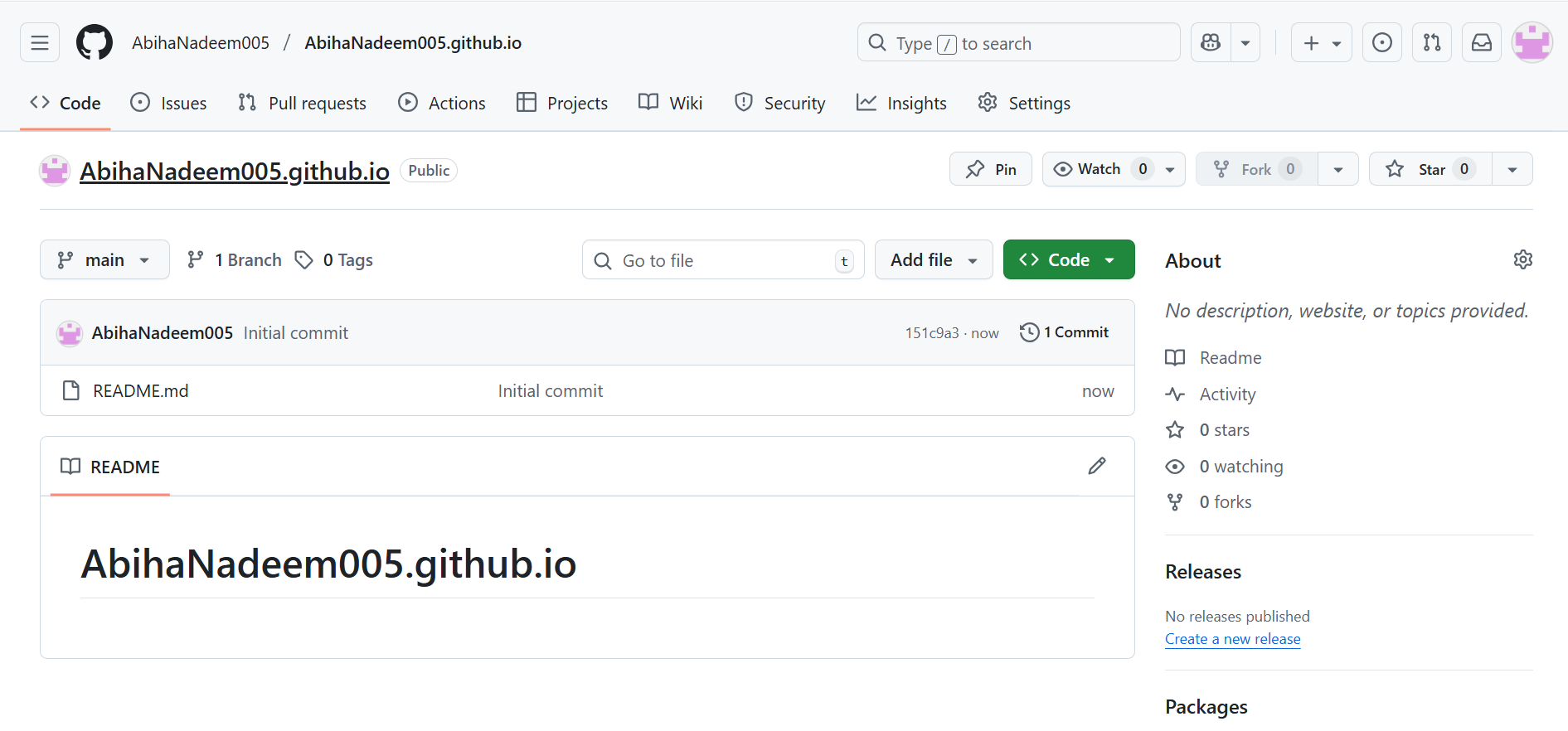
**Task 11 – Creating a GitHub Pages Portfolio Site**

This task will guide you through creating a personal portfolio/static site using GitHub Pages.

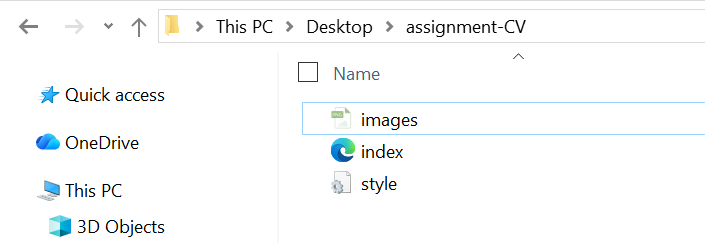
**Steps**

1. **Create a GitHub Pages Repository**
   * Go to GitHub and create a new repository named <your-username>.github.io  
     (e.g., if your username is WaqasSaleem97, the repo should be WaqasSaleem97.github.io).
   * Make sure the repository is **public**.
   * Save a screenshot of the newly created repository as github\_pages\_repo.png.





1. **Add Static Website Code**
   * Prepare a simple static website (e.g., your CV or portfolio) in HTML/CSS/JS by using the sample code provided in the repository [Iqra Bashir CV](https://github.com/WaqasSaleem97/assignment-CV).
   * Put your files in a local folder.
   * Save a screenshot of your local files as local\_static\_site.png.



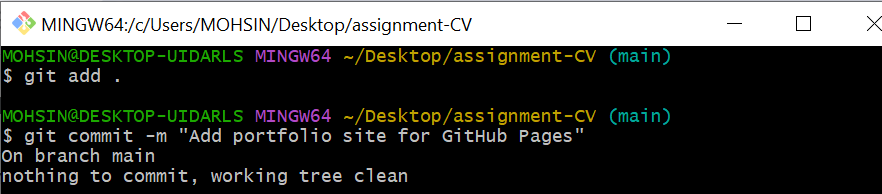
1. **Push the Files to GitHub**
   * Initialize git in your folder (if not already):
   * git init

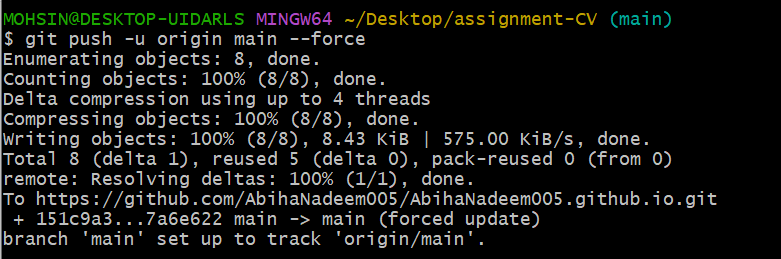
git remote add origin git@github.com:<your-username>/<your-username>.github.io.git

* + Add and commit your files:
  + git add .
  + git commit -m "Add portfolio site for GitHub Pages"

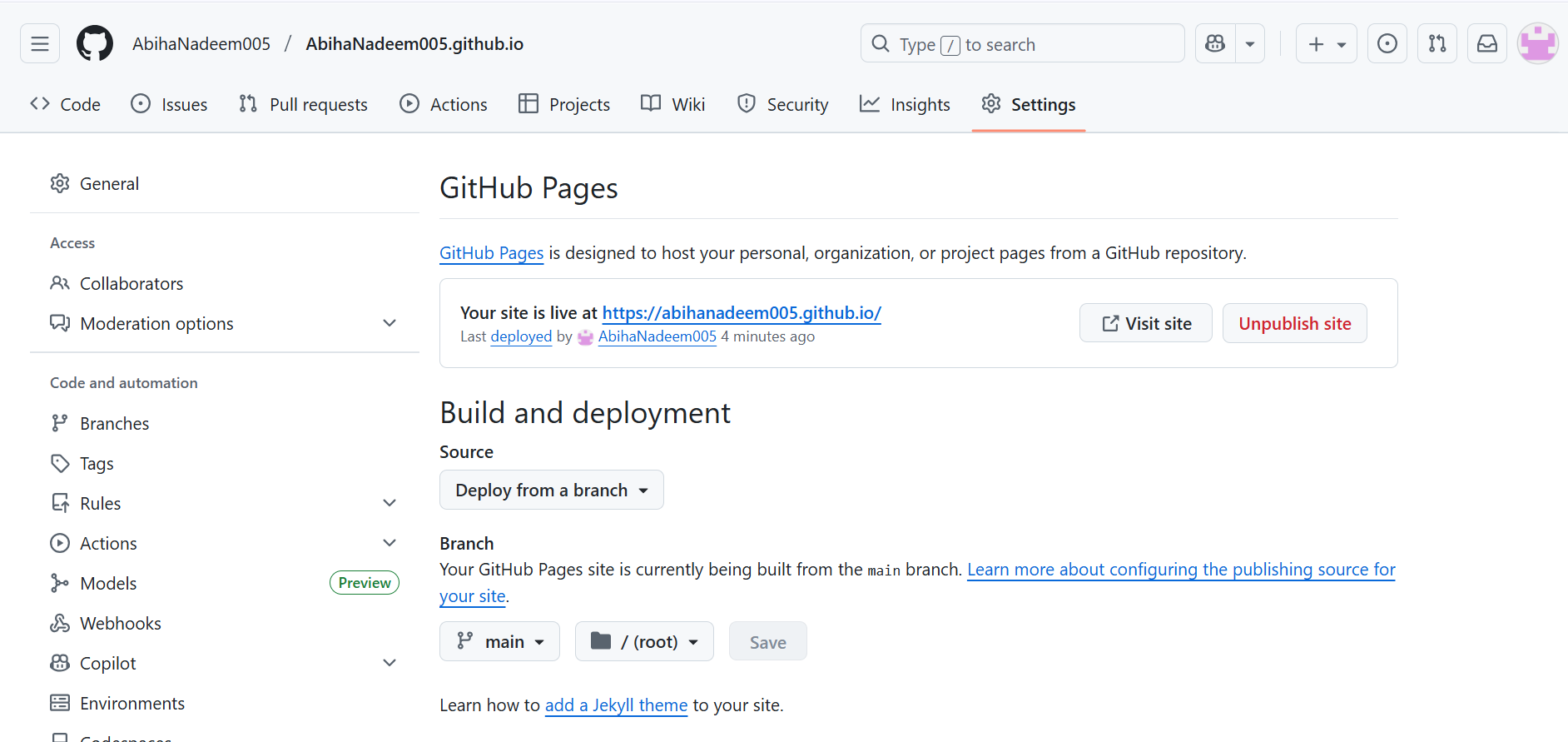
git push -u origin main

* + Save a screenshot of your terminal showing a successful push as push\_static\_site.png.

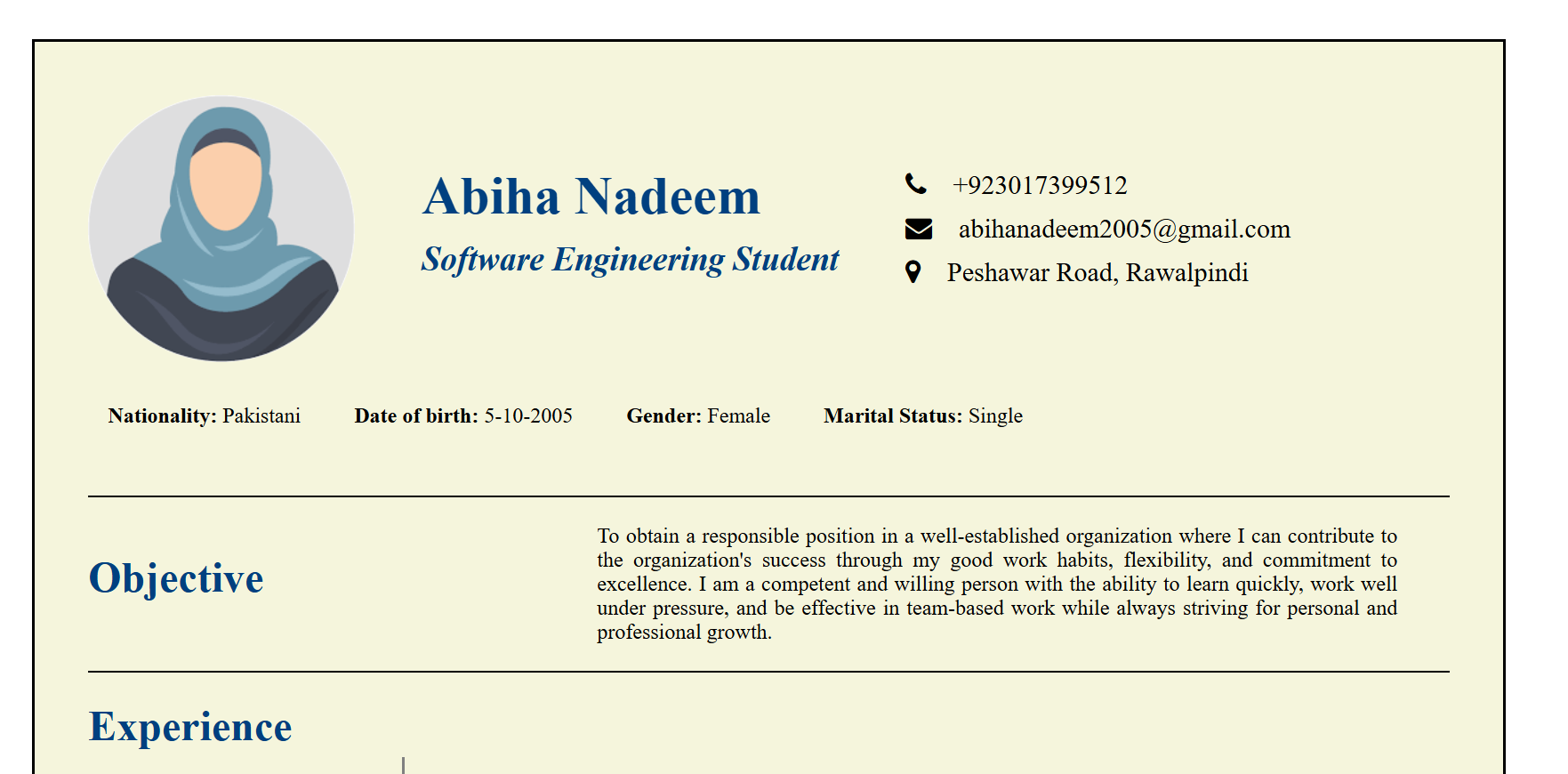




1. **Check GitHub Pages Settings**
   * Go to your repository on GitHub.
   * Click **Settings** > **Pages**.
   * Confirm that your site is published and see the link to your live site (e.g., [https://your-username.github.io](https://your-username.github.io/)).
   * Save a screenshot of the Pages settings as github\_pages\_settings.png.



1. **Visit Your Live Site**
   * Open your GitHub Pages site in the browser.
   * Save a screenshot of your live site as live\_site.png.



Link: <https://github.com/AbihaNadeem005/AbihaNadeem005.github.io/settings/pages>

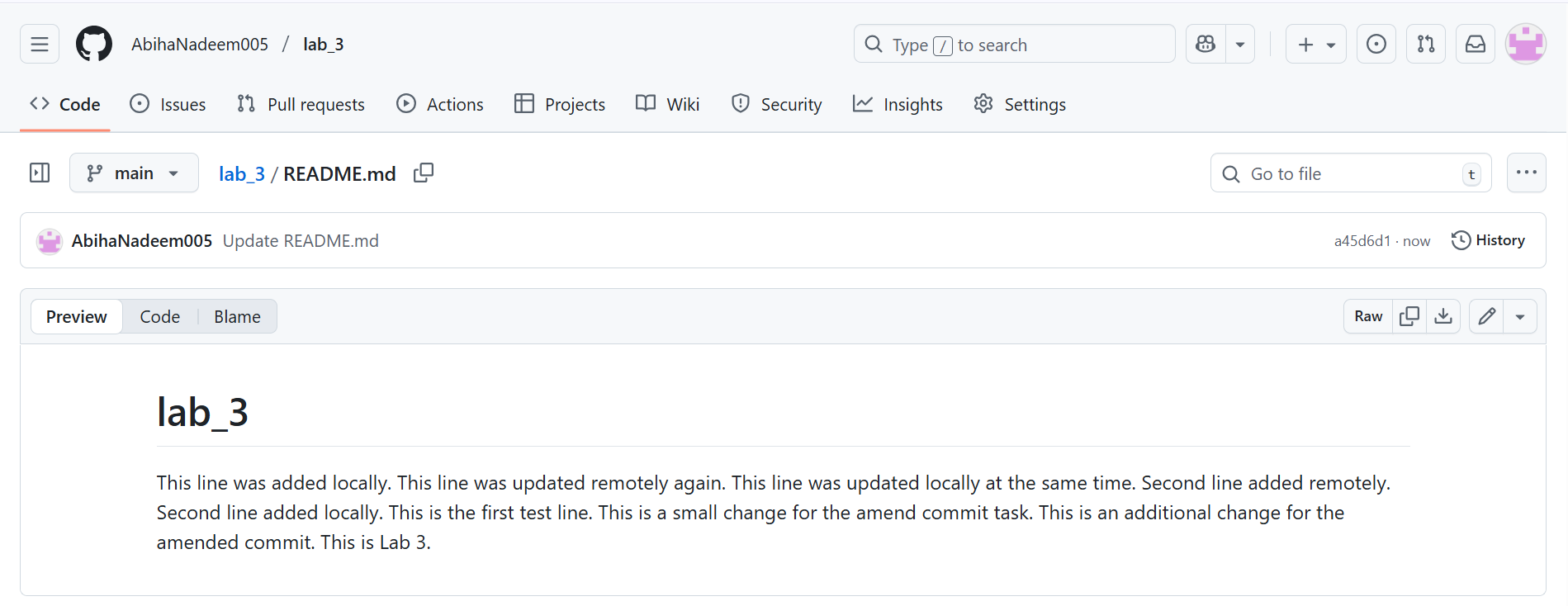
<https://abihanadeem005.github.io/>

**Exam Evaluation Questions**

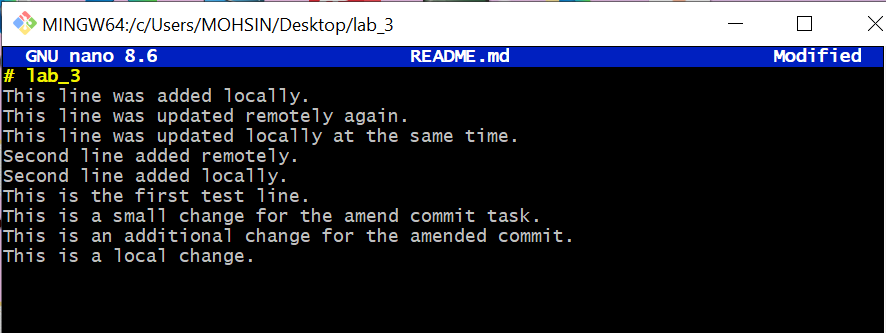
**1. Local vs Remote Conflict Resolution**

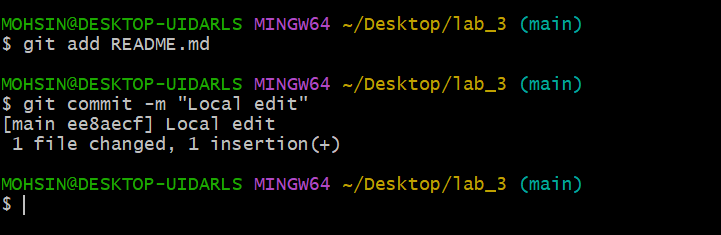
**Steps:**

1. On GitHub, edit a file (e.g., README.md) and commit the change.
   * Screenshot as Q1\_remote\_edit.png

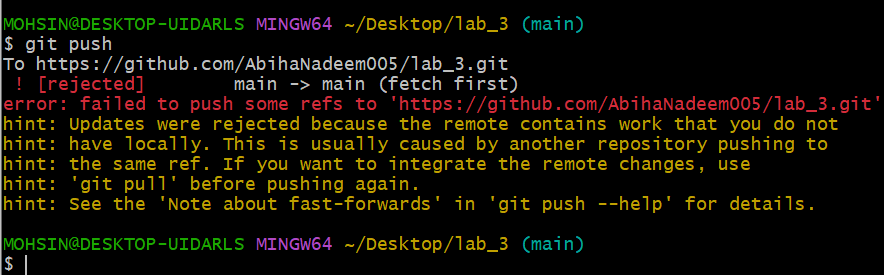


1. On your local machine, edit the same file differently (Avoid Conflict) and commit.
   * Screenshot as Q1\_local\_edit.png

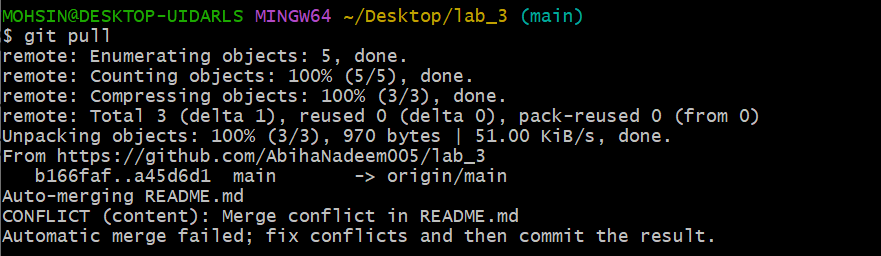


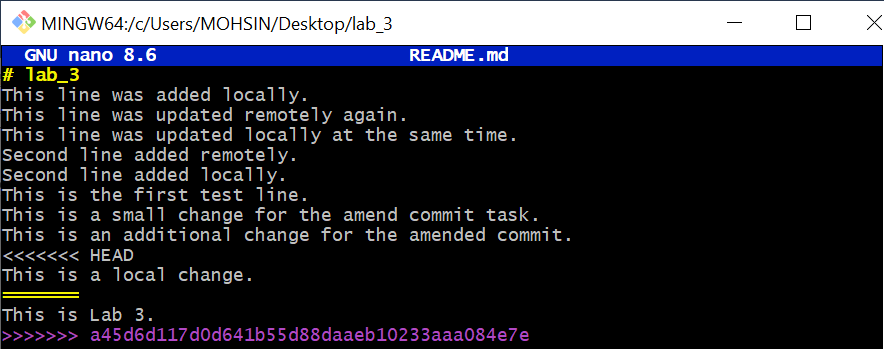


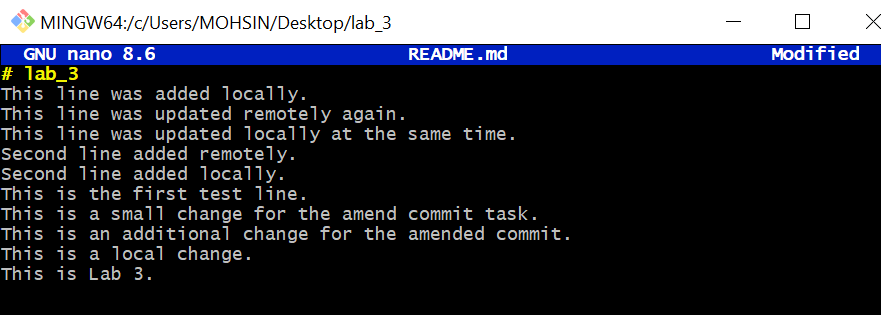
1. Try to push your local commit and observe the error.
   * Screenshot as Q1\_push\_error.png

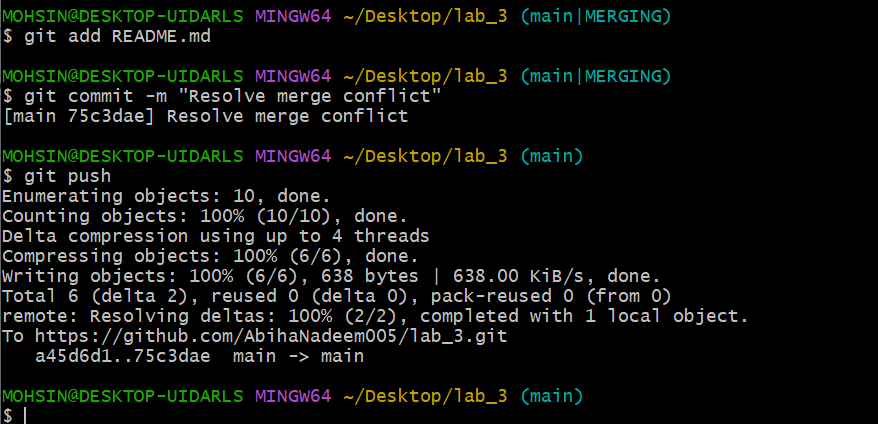


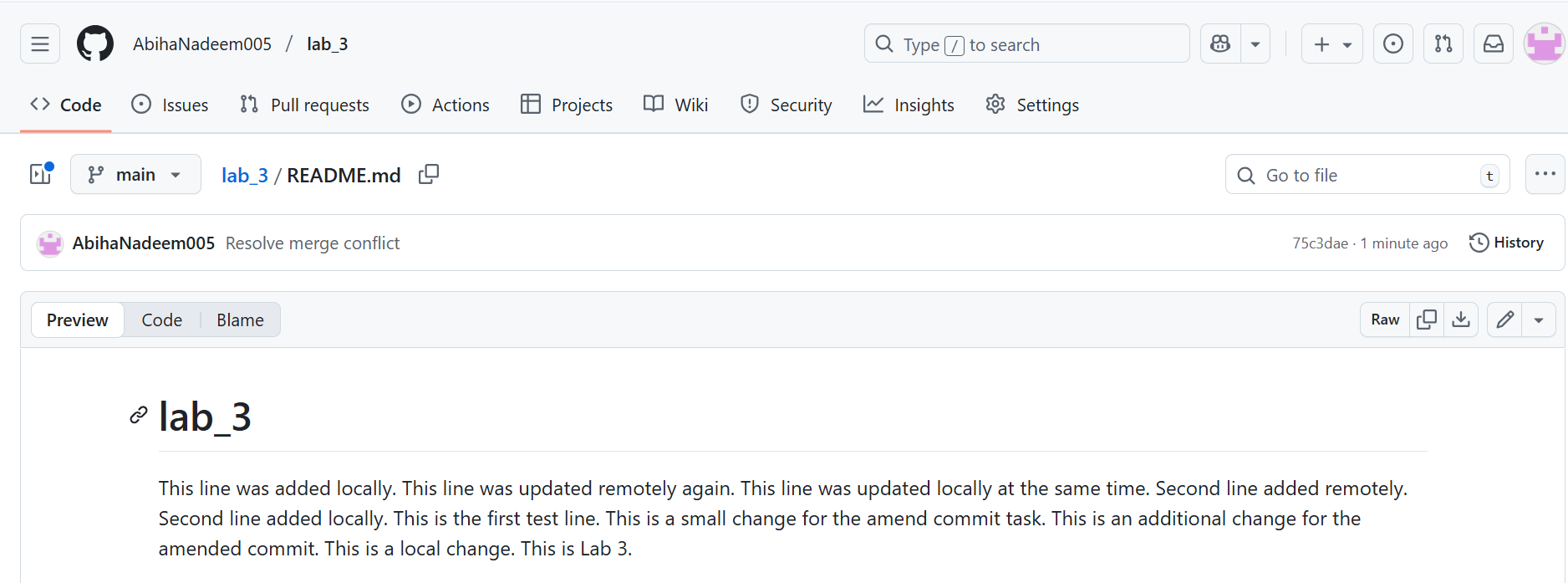
1. Resolve the conflict by running git pull (merge), then push.
   * Screenshot as Q1\_merge\_resolution.png



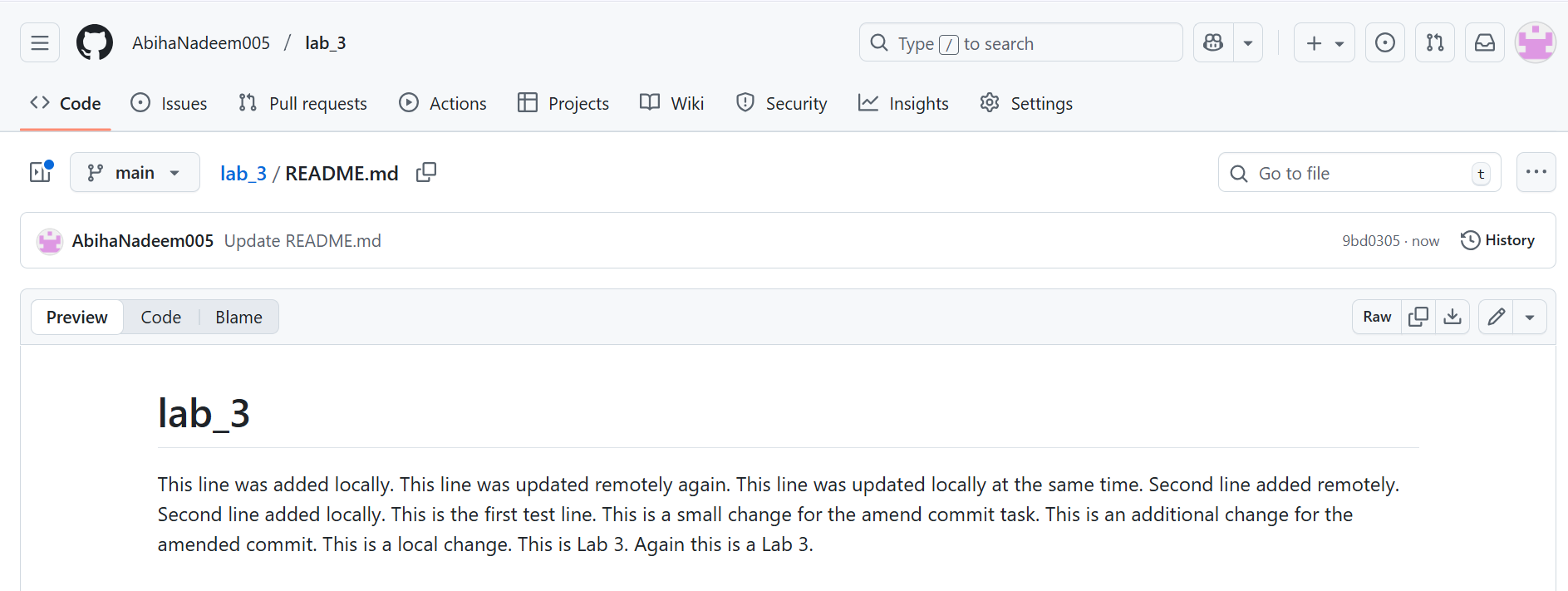


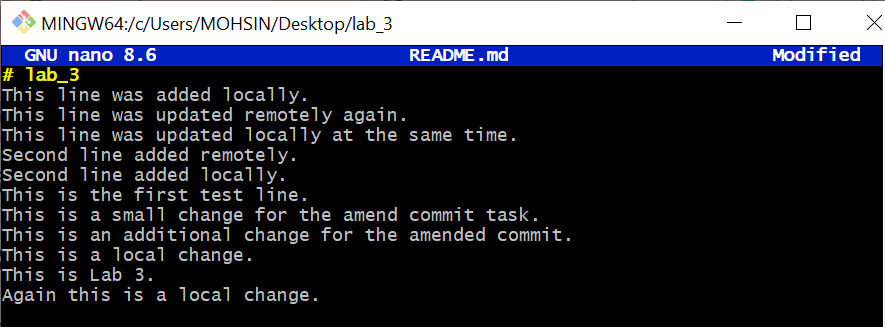




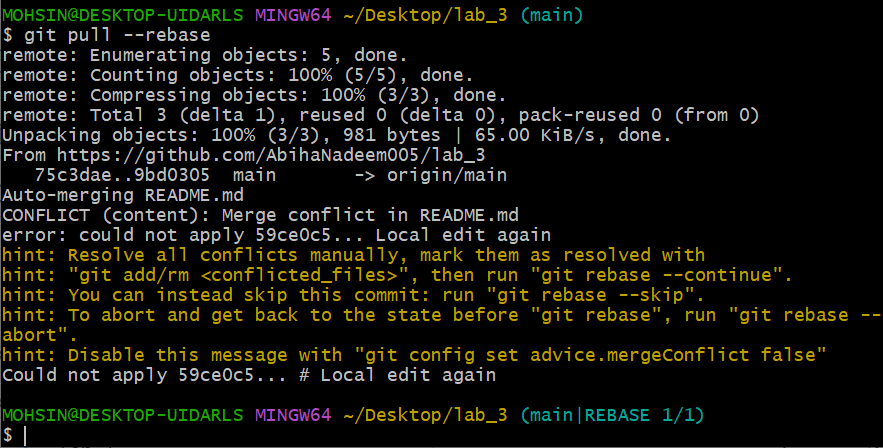


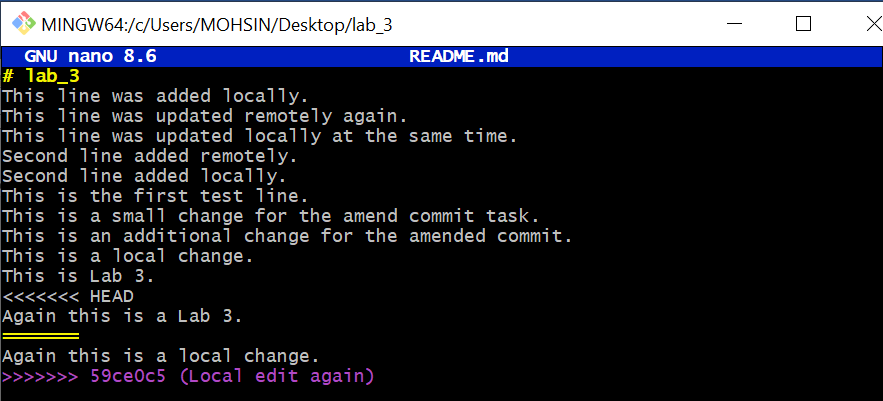
1. Repeat with another remote/local change, but resolve using git pull --rebase, then push.
   * Screenshot as Q1\_rebase\_resolution.png

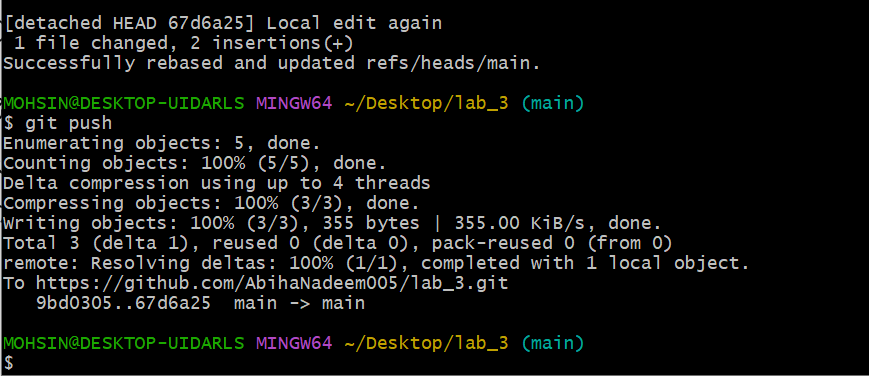


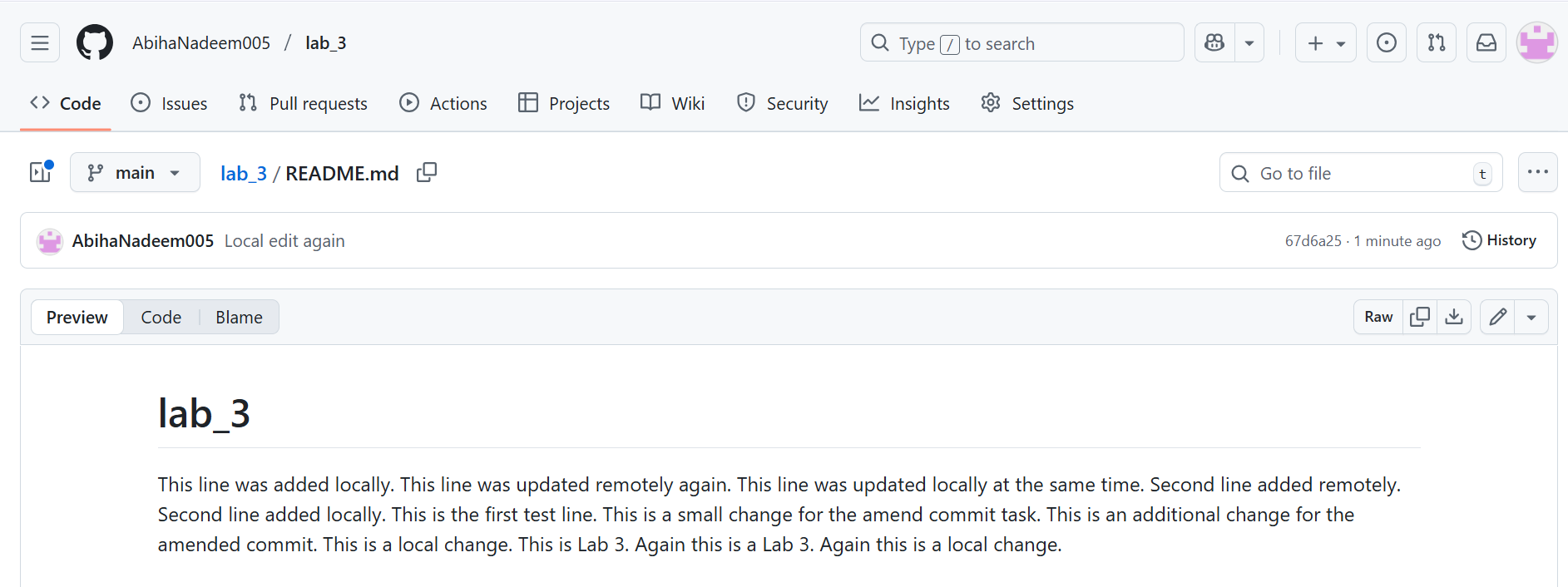








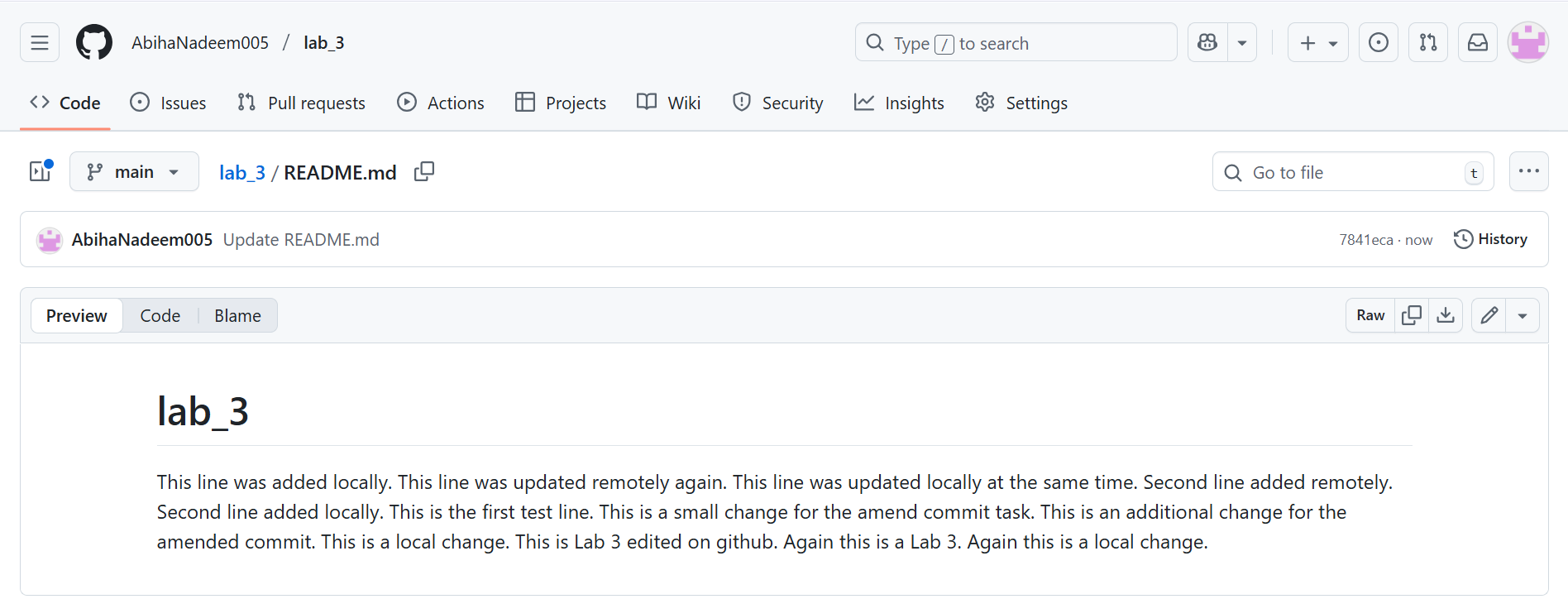




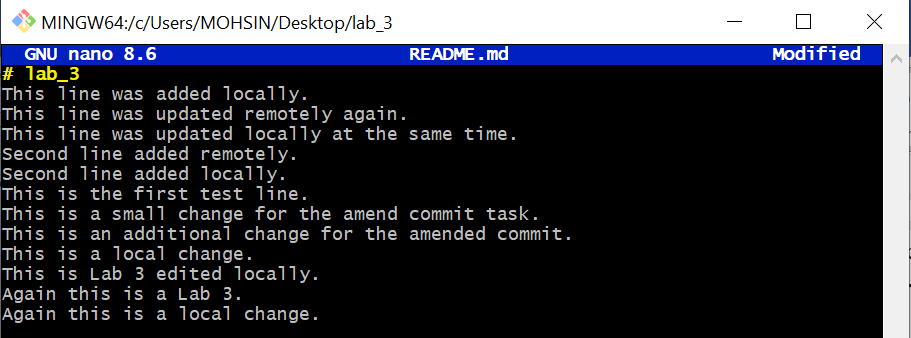
**2. Manual Merge Conflict Handling**

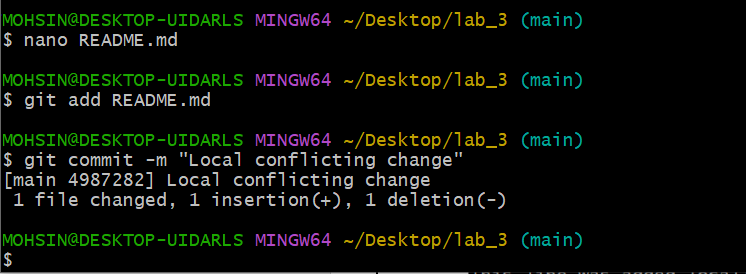
**Steps:**

1. On GitHub, change a specific line in a file and commit.
   * Screenshot as Q2\_remote\_conflict\_edit.png



1. Locally, change the same line differently and commit.
   * Screenshot as Q2\_local\_conflict\_edit.png

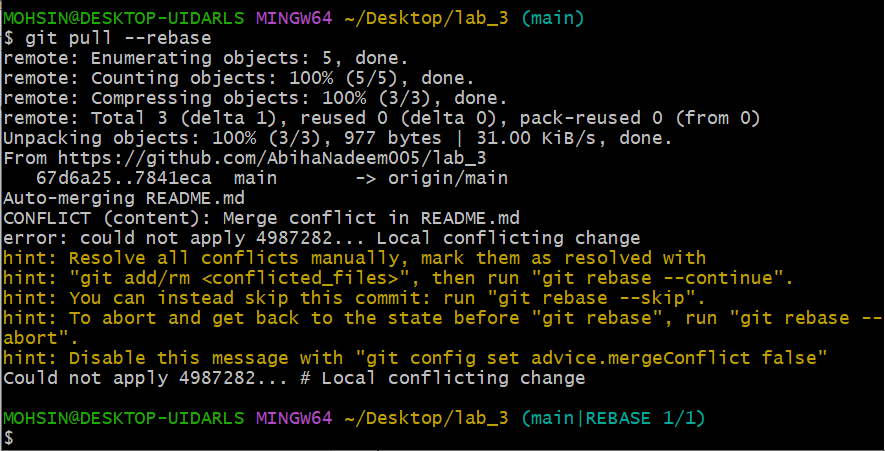




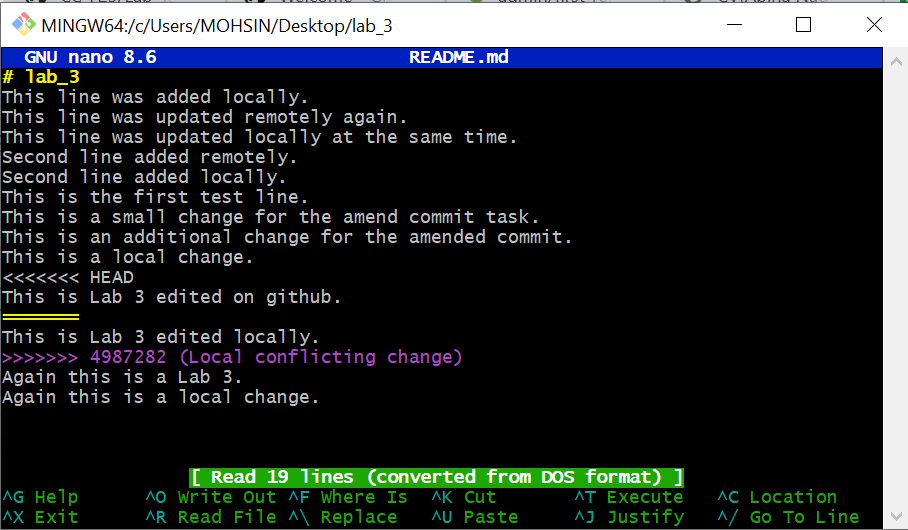
1. Try to push your local change and observe the conflict error.
   * Screenshot as Q2\_conflict\_push\_error.png

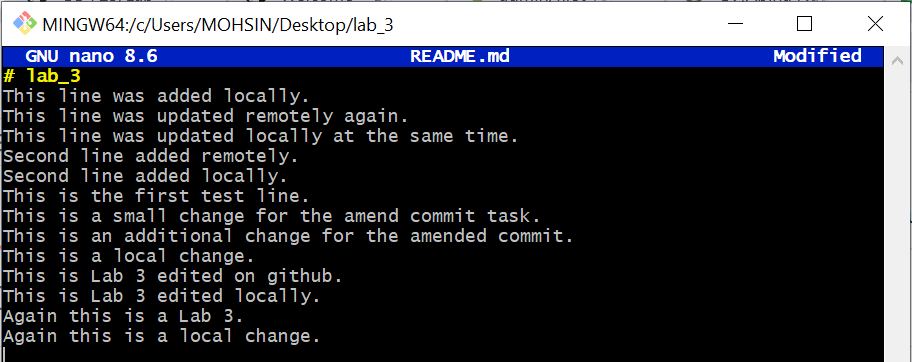


1. Use git pull --rebase to fetch changes and trigger the conflict.
   * Screenshot as Q2\_rebase\_conflict.png

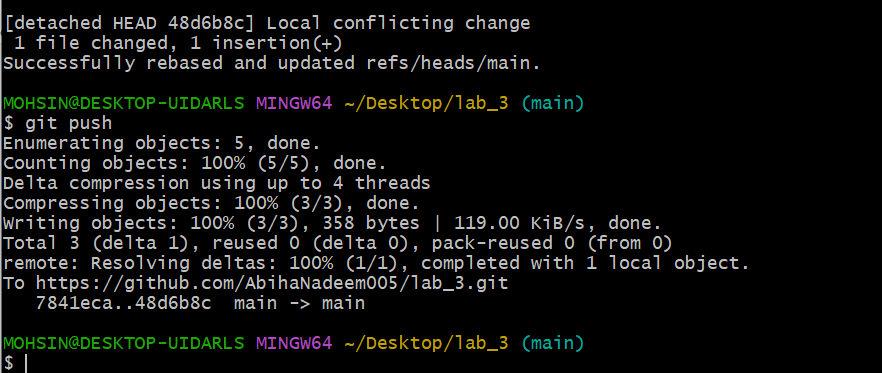


1. Edit the conflicted file to resolve the conflict manually.
   * Screenshot as Q2\_resolved\_file.png





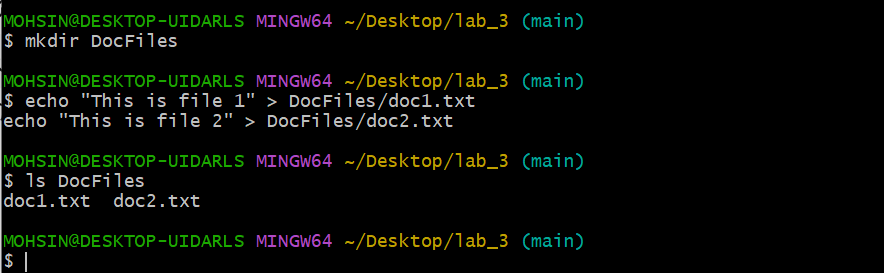
1. Mark the conflict resolved (git add <file>, git rebase --continue) and push.
   * Screenshot as Q2\_resolution\_complete.png



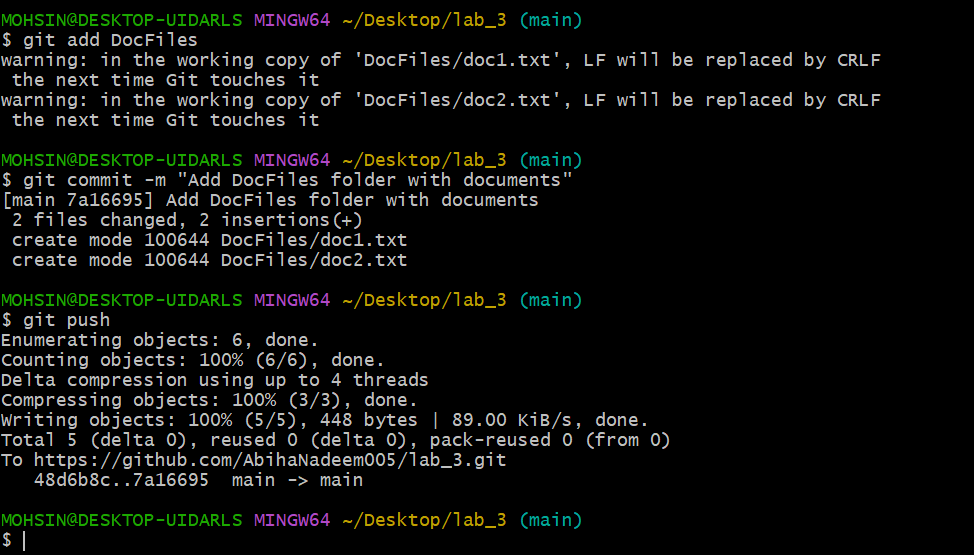
**3. Managing Ignored and Tracked Files**

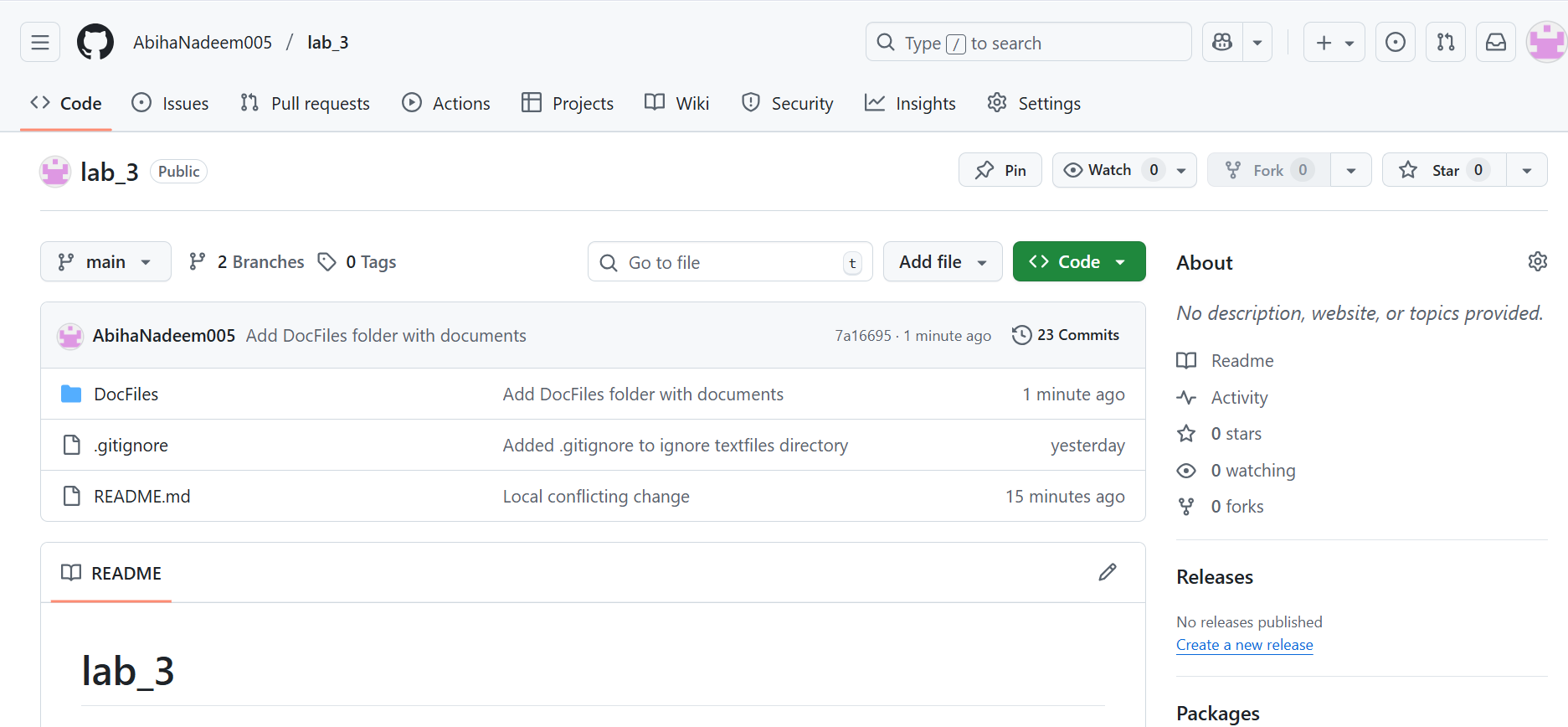
**Steps:**

1. Create a new folder (e.g., DocFiles) and add several files inside.
   * Screenshot as Q3\_folder\_created.png

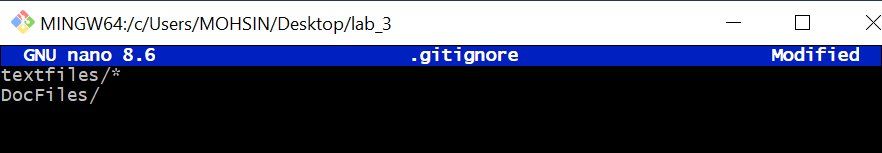


1. Commit and push the folder/files to GitHub.
   * Screenshot as Q3\_files\_pushed.png

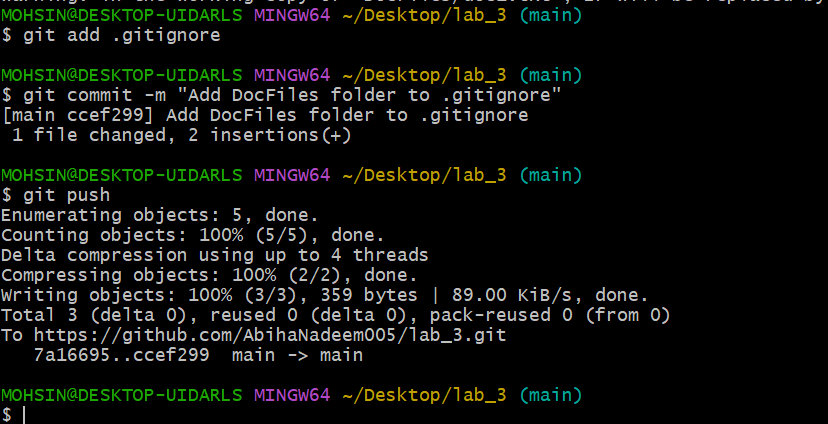


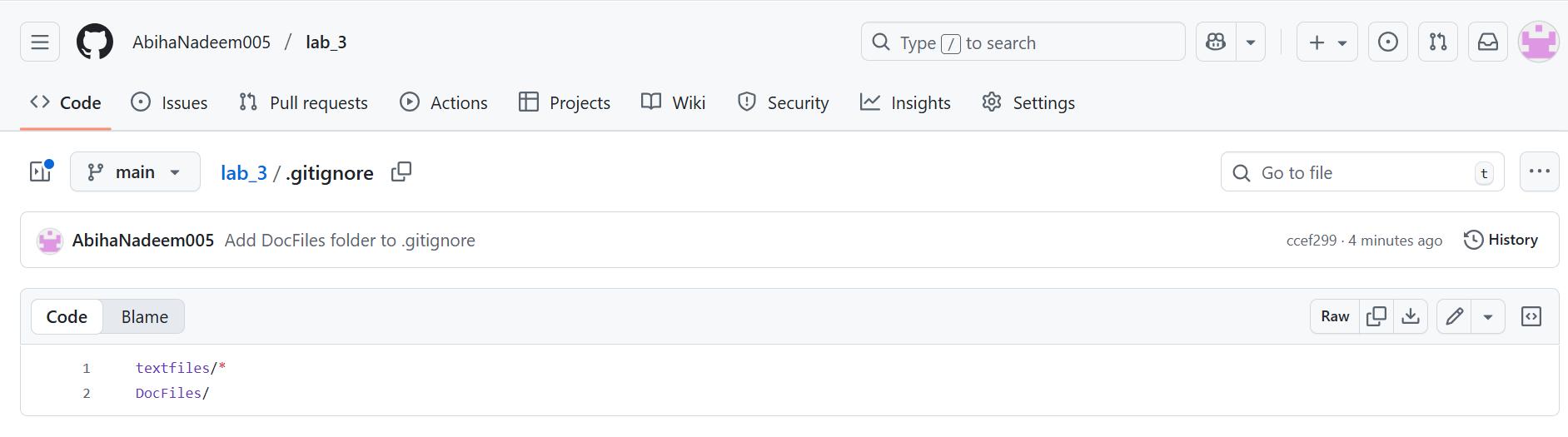


1. Add the folder to your .gitignore file.
   * Screenshot as Q3\_gitignore\_added.png

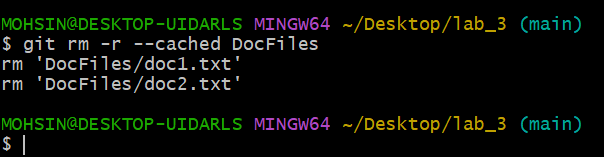


1. Commit and push the .gitignore update.
   * Screenshot as Q3\_gitignore\_pushed.png

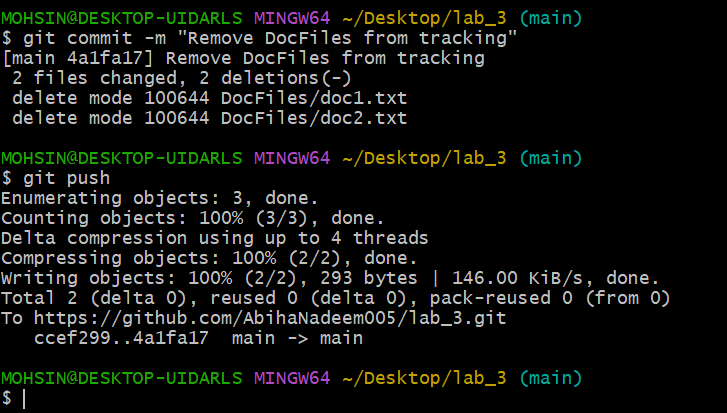




1. Remove the folder from tracking using git rm -r --cached <folder>.
   * Screenshot as Q3\_folder\_untracked.png



1. Commit and push the change, then verify the folder is no longer tracked on GitHub.
   * Screenshot as Q3\_folder\_removed\_github.png

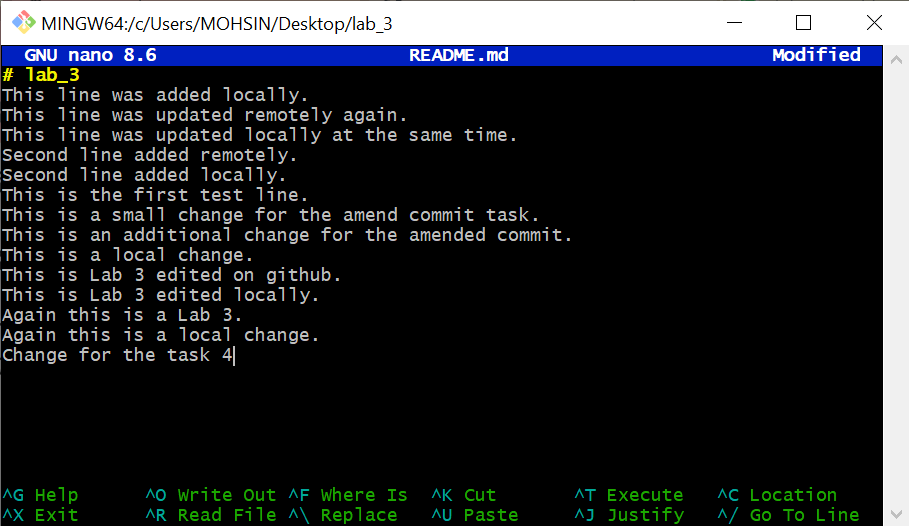


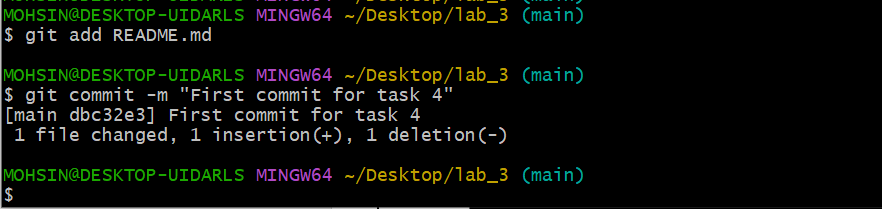


**4. Commit History Manipulation and Recovery**

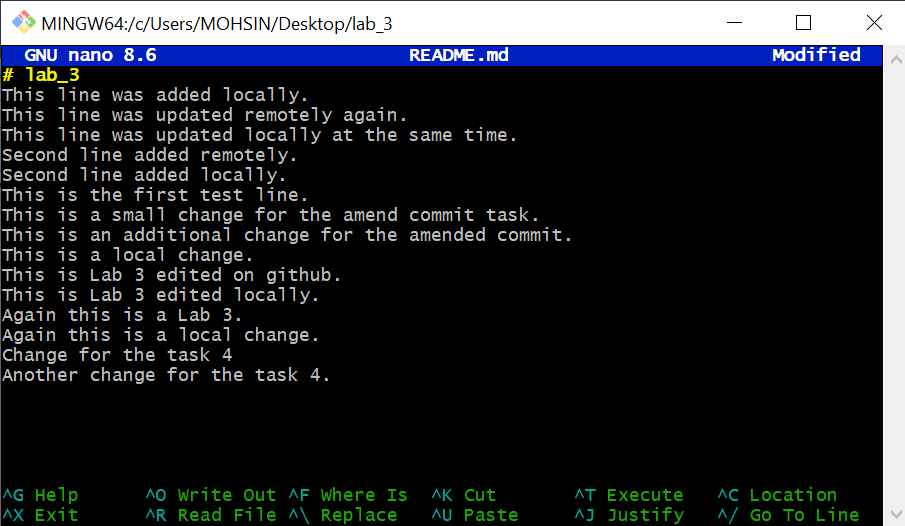
**Steps:**

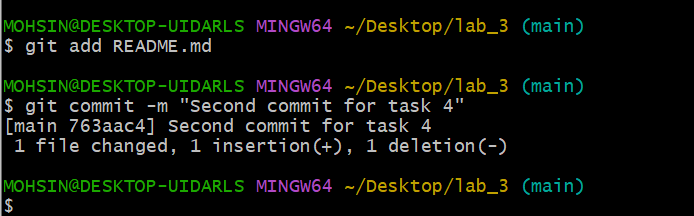
1. Make a change and commit it.
   * Screenshot as Q4\_first\_commit.png



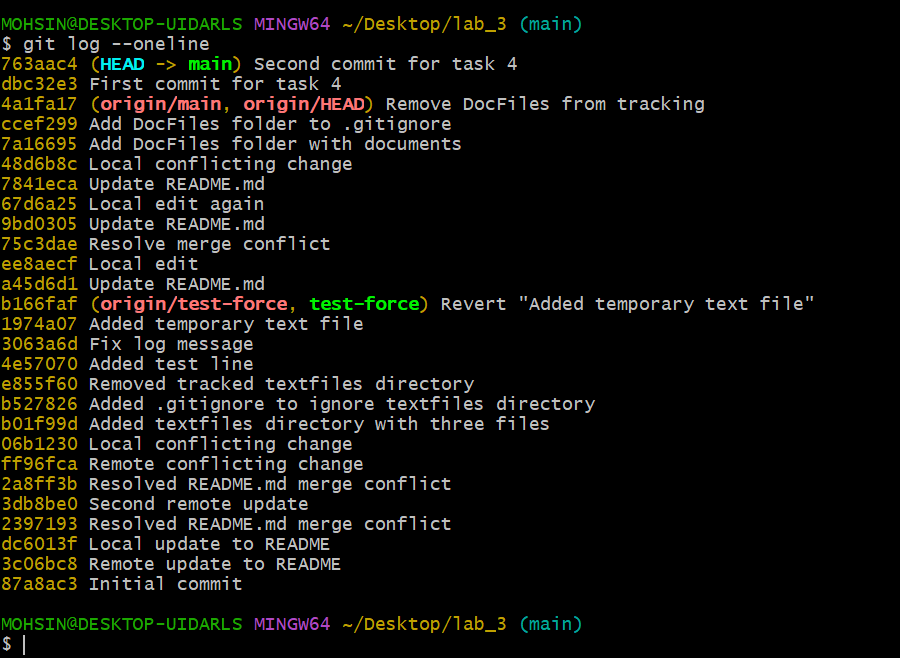


1. Make another change and commit again.
   * Screenshot as Q4\_second\_commit.png

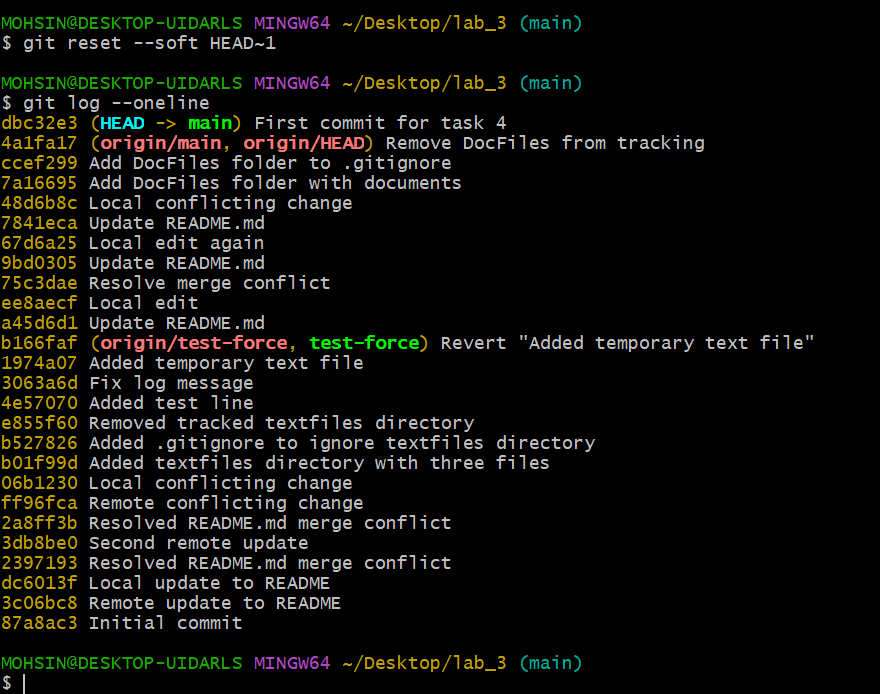




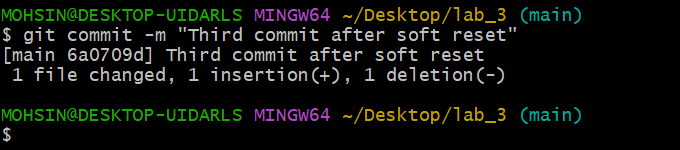
1. View your commit history.
   * Screenshot as Q4\_commit\_history.png



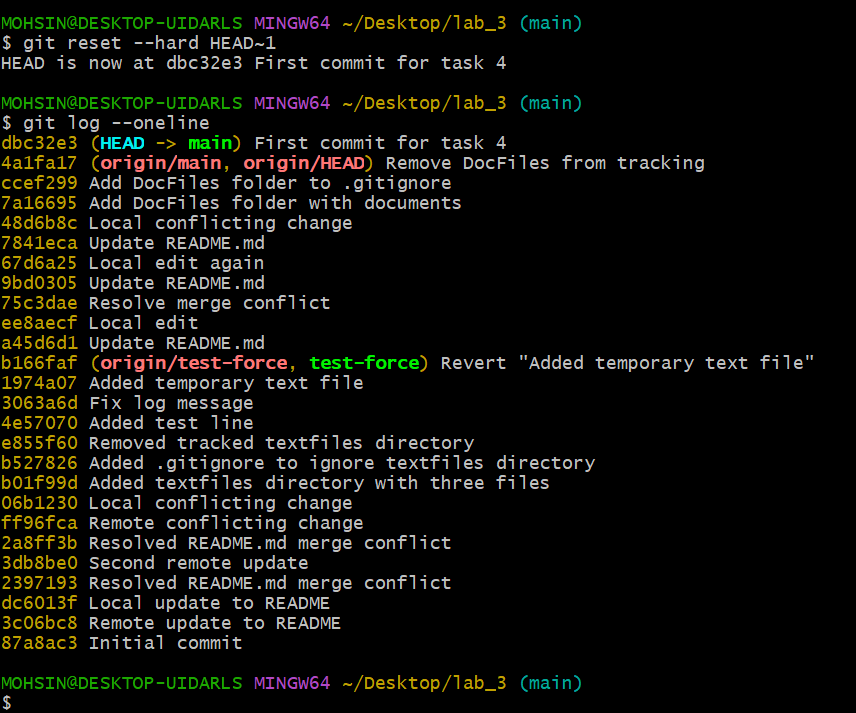
1. Perform a soft reset (git reset --soft HEAD~1) and observe your file and history.
   * Screenshot as Q4\_soft\_reset.png



1. Make commit again.
   * Screenshot as Q4\_third\_commit.png



1. Perform a hard reset (git reset --hard HEAD~1) and observe the changes.
   * Screenshot as Q4\_hard\_reset.png



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*