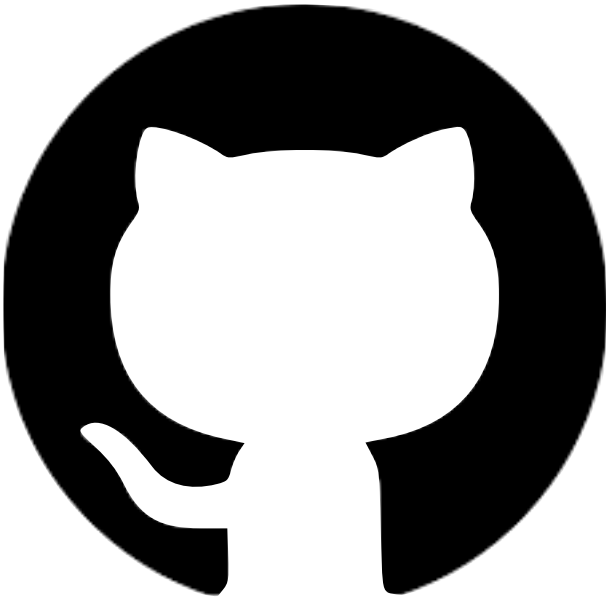
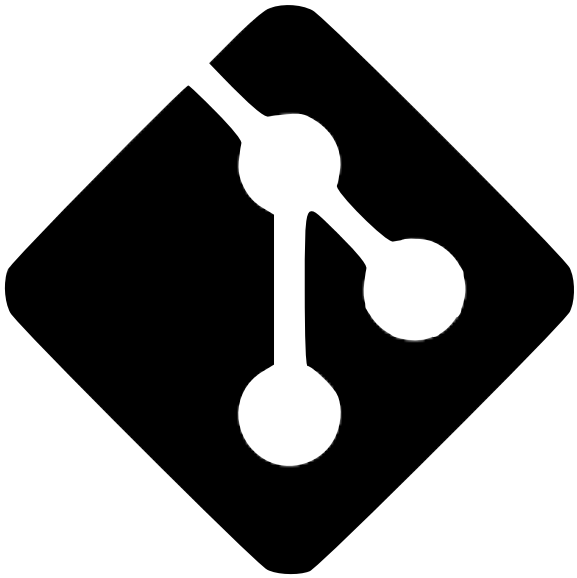
Source Code Management

Course Code: CSE 2015

Slot: L15 L16





Name: Ishrit Rai

SEN No.: A86605224057

Faculty: Dr Monit Kapoor

Lab Session 1: Git Fundamentals

Computer

A **computer** is any device capable of performing calculations, whether they are logical or mathematical.

Program/Code

A **program** (or **code**) is a set of instructions, often organised as an algorithm, that directs a computer to perform a specific task.

Need for Managing Source Code

Modern applications, such as Spotify, consist of multiple programs working together on both the frontend and backend to deliver a smooth user experience. Regular updates are essential for:

- **Fixing Bugs:** Quickly resolving errors that may occur.
- **Improving UI/UX:** Enhancing the user interface and overall experience.
- **Optimising Performance:** Addressing and refining issues for better performance.

For programmers, effective management of source code is crucial because:

- It ensures that all files remain in context throughout the lifecycle of the program.
- It facilitates collaboration, allowing multiple developers to work together on a shared codebase.

Tools for Source Code Management

1. **Git:**
   A version control system that runs locally on your computer. Git helps track changes and manage versions of your project.
2. **GitHub:**
   A global, cloud-based platform that hosts Git repositories, enabling developers to share, collaborate, and contribute to projects from anywhere in the world.

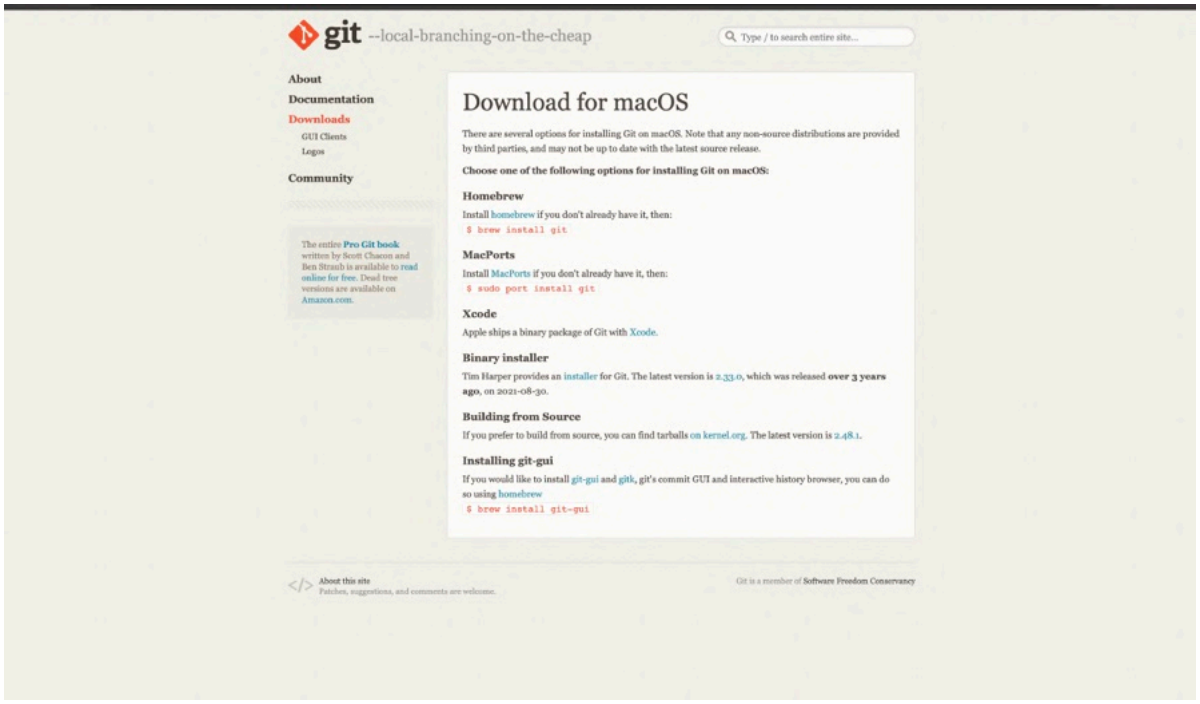
Version

A **version** in version control represents a snapshot of your project at a specific moment in time. This snapshot allows you to review, revert, or compare changes made throughout the development process.

Lab Practical 1

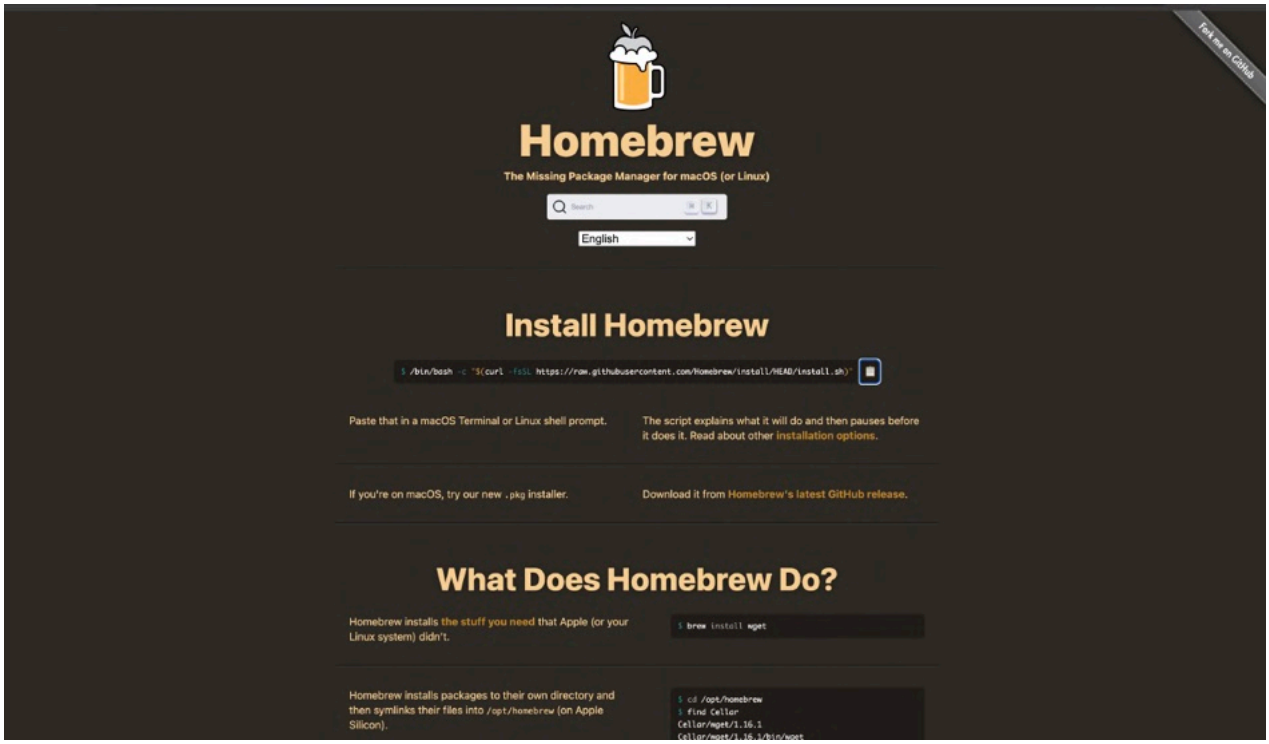
1. Installing Git Using Homebrew

Step 1: Visit section 1.5 of pro git document and navigate to macOS section



Step 2: Install Homebrew (if not already installed):

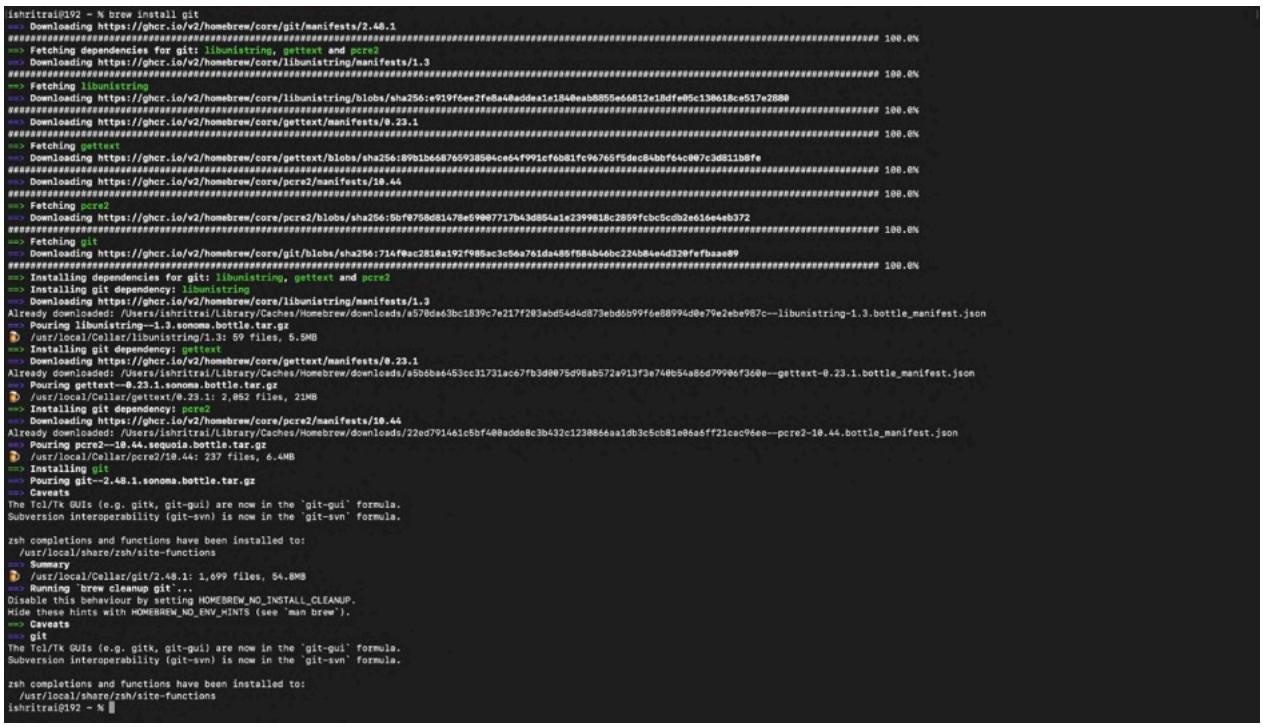
- **Command:**
- /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
- **Description:** Installs Homebrew, a package manager for macOS.



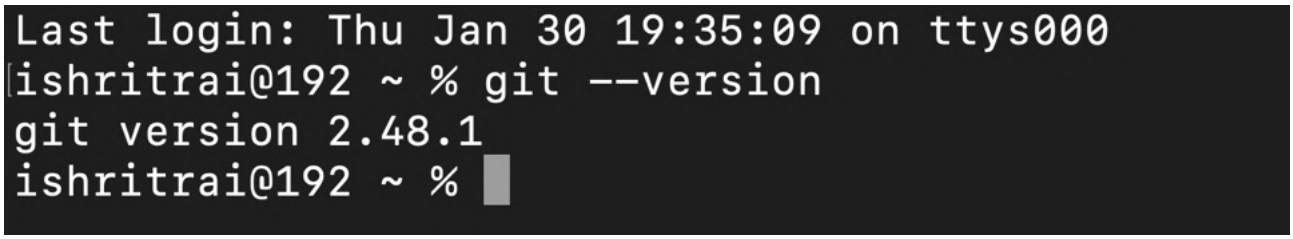
Step 3: Install Git Using Homebrew:

- **Command**: brew install git
- **Description:** Installs Git via Homebrew

```
ishritrai@192 ~ % $ brew install git
zsh: command not found: $
ishritrai@192 ~ % /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
==> Checking for `sudo` access (which may request your password)...
Password:
==> This script will install:
/usr/local/bin/brew
/usr/local/share/doc/homebrew
/usr/local/share/man/man1/brew.1
/usr/local/share/zsh/site-functions/_brew
/usr/local/etc/bash_completion.d/brew
/usr/local/Homebrew

Press RETURN/ENTER to continue or any other key to abort:
==> /usr/bin/sudo /usr/sbin/chown -R ishritrai:admin /usr/local/Homebrew
==> Downloading and installing Homebrew...
remote: Enumerating objects: 2768, done.
remote: Counting objects: 100% (627/627), done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 2768 (delta 616), reused 611 (delta 609), pack-reused 2141 (from 3)
==> Updating Homebrew...
==> Downloading https://ghcr.io/v2/homebrew/portable-ruby/portable-ruby/blobs/sha256:4ffc8607e08e9bd536f1df71643b2ecb4cea1a15be9226f297008bc34d0bc8e2
######################################################################################################################## 100.0%
==> Pouring portable-ruby-3.3.7.el_capitan.bottle.tar.gz
Updated 2 taps (homebrew/core and homebrew/cask).
==> Installation successful!

==> Homebrew has enabled anonymous aggregate formulae and cask analytics.
Read the analytics documentation (and how to opt-out) here:
  https://docs.brew.sh/Analytics
No analytics data has been sent yet (nor will any be during this install run).

==> Homebrew is run entirely by unpaid volunteers. Please consider donating:
  https://github.com/Homebrew/brew#donations

==> Next steps:
- Run these commands in your terminal to add Homebrew to your PATH:
    echo >> /Users/ishritrai/.zprofile
    echo 'eval "$(/usr/local/bin/brew shellenv)"' >> /Users/ishritrai/.zprofile
    eval "$(/usr/local/bin/brew shellenv)"
- Run brew help to get started
- Further documentation:
    https://docs.brew.sh
```



Step 4; Verify Git Installation:



2. Basic CLI Commands

1) Command: ls

Description: Lists all files and directories in the current directory.

```
ishritrai@192 ~ % ls
Applications            Downloads               Music                   Screen Studio Projects
Desktop                 Library                 Pictures                git
Documents               Movies                  Public                  hello.txt
ishritrai@192 ~ % ls DESKTOP
Installing_Git pdf.pdf
ishritrai@192 ~ %
```

2) Command: date

Description: shows the current date and time in a standard format

3) Command: clear

```
Last login: Thu Jan 30 19:37:21 on ttys000
ishritrai@192 ~ % date
Thu Jan 30 19:53:00 IST 2025
```

Description: The `clear` command in the CLI is used to clear all the current text and output displayed in the terminal window.

```
Last login: Thu Jan 30 19:37:21 on ttys000
ishritrai@192 ~ % date
Thu Jan 30 19:53:00 IST 2025
ishritrai@192 ~ % time
shell  0.01s user 0.02s system 0% cpu 36.693 total
children  0.01s user 0.01s system 0% cpu 36.693 total
ishritrai@192 ~ % clear
```

```
ishritrai@192 ~ %
```

4) Command: time

Description: The `time` command in the CLI is used to measure the execution time of a command or program.

```
ishritrai@192 ~ % time
shell  0.01s user 0.02s system 0% cpu 54.824 total
children  0.01s user 0.02s system 0% cpu 54.824 total
ishritrai@192 ~ %
```

5) Command: rm hello.txt

Description: Removes the file `hello.txt` from the current directory.

6) Command: cat hello.txt

Description: The `cat` command (short for **concatenate**) is used to display the contents of a file.

```
Last login: Thu Jan 30 19:55:39 on ttys000
ishritrai@192 ~ % rm hello.txt
ishritrai@192 ~ % cat hello.txt
cat: hello.txt: No such file or directory
ishritrai@192 ~ % cat hi.txt
hi this is the first line
hi this is the second line
bye this is the third line
ishritrai@192 ~ %
```

7)Command: cd Desktop

Description: Changes the current working directory to the `Desktop` directory.

```
ishritrai@192 ~ % cd Desktop
ishritrai@192 Desktop % pwd
/Users/ishritrai/Desktop
ishritrai@192 Desktop %
```

8)Command: `ls`

Description: Lists all files and directories in the current directory.

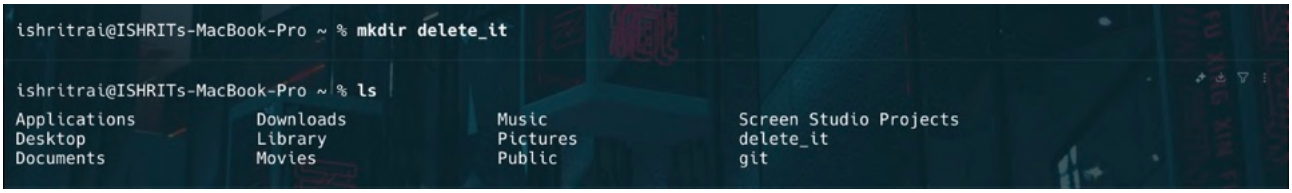
```
Last login: Thu Jan 30 21:19:53 on ttys000
ishritrai@192 ~ % cd /Users/ishritrai/Desktop/LEARNING_git
ishritrai@192 LEARNING_git % ls
git1    git2    git3
ishritrai@192 LEARNING_git % mkdir git4
ishritrai@192 LEARNING_git % ls
git1    git2    git3    git4
ishritrai@192 LEARNING_git %
```

9.Command: pwd

Description: returns the present working directory

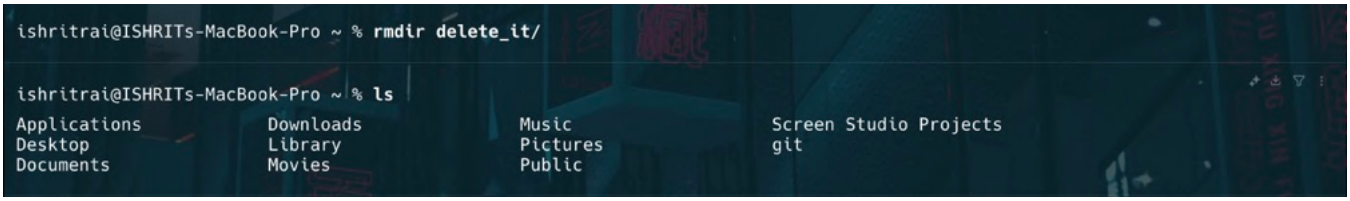
10. Command: mkdir

Description: used to to make new directory/folder



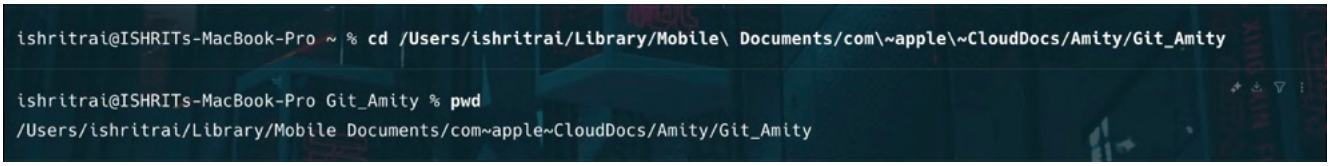
11. Command: rmdir

Description: used to remove a directory



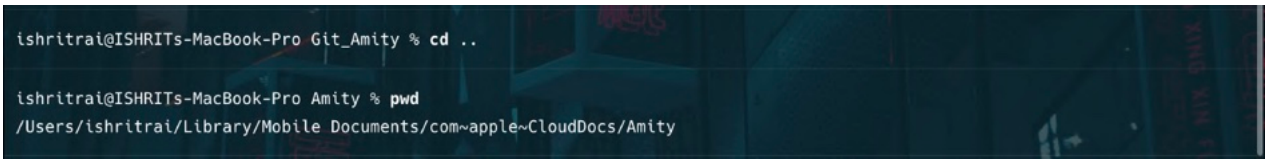
12. Command: cd

Description: used to change current directory



13. Command: cd ..

Description: used to exit the current sub directory



3. Vim Text Editor

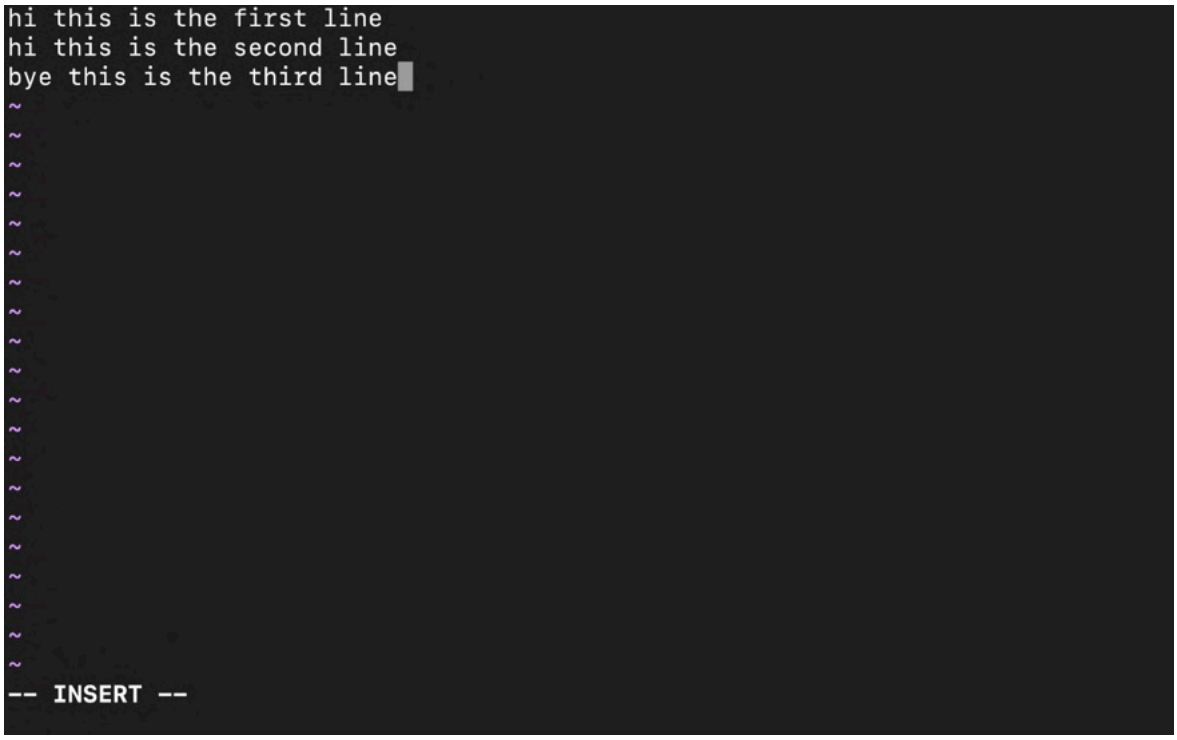
1) Command: vi hi.txt

Description: Opens (or creates) the file `hi.txt` in the Vim text editor.

```
Last login: Thu Jan 30 19:52:46 on ttys000
ishritrai@192 ~ % vi hi.txt
```

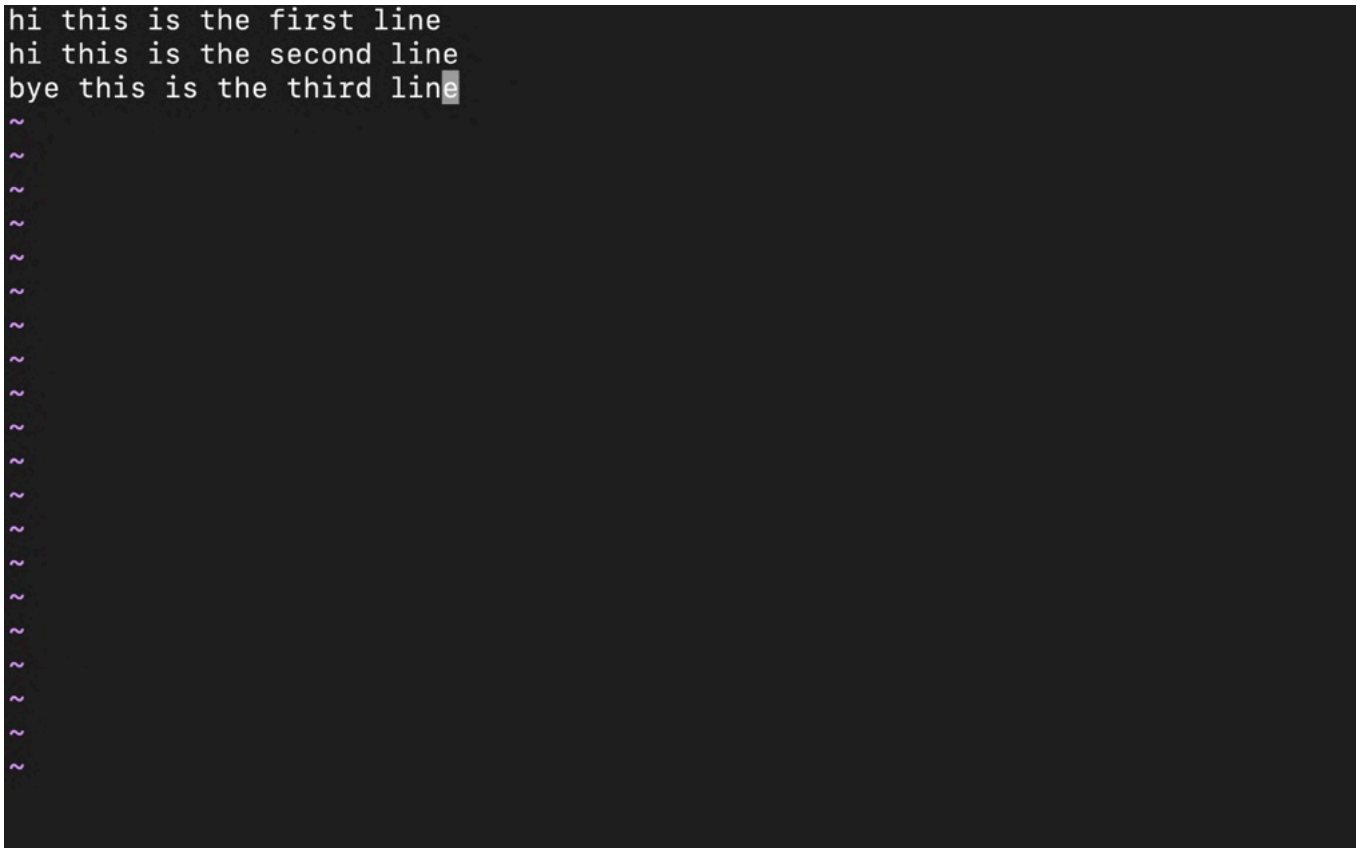
2) Command: `i` (Insert Mode)

Description: Enters insert mode in Vim to allow text input.



3) Command: esc

Description: Used to exit insert mode



4) Command: `:wq`

Description: Saves the changes and exits the Vim editor.

```
hi this is the first line
hi this is the second line
bye this is the third line
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

```
Last login: Thu Jan 30 19:52:46 on ttys000
ishritrai@192 ~ % vi hi.txt
ishritrai@192 ~ %
```

```
ishritrai@192 ~ % cat hi.txt
hi this is the first line
hi this is the second line
bye this is the third line
```

4. Git Configuration

Git configuration is a fundamental aspect of setting up your development environment. The `git config` command allows you to customize Git's behavior and set up your identity for version control. This includes configuring your name, email address, default editor, and various other settings that affect how Git operates. These configurations can be set at three levels: system-wide, user-specific (global), or repository-specific (local).

Let's create a simple C program to demonstrate Git configuration in action:

```
// hello.c
#include <stdio.h>

int main() {
    printf("Hello, Git Configuration!\n");
    return 0;
}
```

Now, let's see how to configure Git for this project:

```
ishritrai@192:~/git_lab$ git config --global user.name "Ishrit Rai"
ishritrai@192:~/git_lab$ git config --global user.email "ishritrai@example.com"
ishritrai@192:~/git_lab$ git config --list
user.name=Ishrit Rai user.email=ishritrai@example.com core.editor=vim
core.autocrlf=input init.defaultbranch=main
```

The above commands demonstrate:

- Setting your global Git username
- Setting your global Git email address
- Viewing all current Git configurations

4. Git setup Commands

1. Command: git - - version

Description:The `git --version` command is used to check the installed version of Git on your system.

```
ishritrai@192 LEARNING_git % git --version
git version 2.48.1
ishritrai@192 LEARNING_git %
```

2. Command: `git init`

Description: Initializes a new Git repository in the current directory.

```
ishritrai@192 git1 % git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/ishritrai/Desktop/LEARNING_git/git1/.git/
ishritrai@192 git1 % git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

3. Command: git status

Description: Displays the current status of the working directory and staging area.

```
ishritrai@192 LEARNING_git % ls
git1    git2    git3    git4
ishritrai@192 LEARNING_git % cd git1
ishritrai@192 git1 % ls
ishritrai@192 git1 % git status
fatal: not a git repository (or any of the parent directories): .git
ishritrai@192 git1 % ls la
ls: la: No such file or directory
```

The above files are not currently tracked by git

4. Command git config --global user.name “Ishrit Rai”

Description: used to set up user name which will be linked to future commits

5. Command git config --global email.id “raiishrit@gmail.com“

Description: used to set up email Id which will be linked to future commits.

6. Command git config - - list

Description: used to view all the setting

4.Command: git add testone.txt

Description: Adds `testone.txt` to the staging area in preparation for a commit.

```
nothing added to commit but untracked files present (use "git add" to track)
ishritrai@192 git1 % git add testone.txt
ishritrai@192 git1 % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   testone.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        testtwo.txt

ishritrai@192 git1 % git add testtwo.txt
ishritrai@192 git1 % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   testone.txt
        new file:   testtwo.txt

ishritrai@192 git1 %
```

5.Command: git commit -m "add file one"

Description: Commits the staged changes with the message "add file one"

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   testone.txt
        new file:   testtwo.txt

ishritrai@192 git1 % git commit -m "add file one"
[master (root-commit) a02c03d] add file one
 Committer: ISHRIT RAI <ishritrai@192.168.0.100>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 testone.txt
 create mode 100644 testtwo.txt
ishritrai@192 git1 %
```

6.Command: `git log`

Description: Displays the commit history of the repository.

```
ishritrai@192 git1 % git log
commit a02c03dd5351dbc497b02cc75519c8d9aedf7d4b (HEAD -> master)
Author: ISHRIT RAI <ishritrai@192.168.0.100>
Date:   Thu Jan 30 22:46:50 2025 +0530

    add file one
```

7. Command: git clone

Description: to obtain a copy of an existing Git repository

5. Git Commits

Git commits are the fundamental building blocks of version control, representing snapshots of your project at specific points in time. Each commit creates a permanent record of changes with a unique identifier (hash), author information, timestamp, and a descriptive message. Commits allow developers to track the evolution of their codebase, revert changes when needed, and collaborate effectively with team members.

Let's enhance our hello.c program to demonstrate the commit process:

```
// hello.c
#include <stdio.h>
#include <stdlib.h>

void print_greeting(const char* name) {
    printf("Hello, %s! Welcome to Git!\n", name);
}

int main() {
    char name[50];
    printf("Enter your name: ");
    scanf("%49s", name);
    print_greeting(name);
    return 0;
}
```

Now, let's see how to create and manage commits for this project:

```
ishritrai@192:~/git_lab$ git init
Initialized empty Git repository in /home/ishritrai/git_lab/.git/
ishritrai@192:~/git_lab$ git add hello.c
ishritrai@192:~/git_lab$ git status
On branch main Changes to be committed: (use "git restore --staged <file>" to
unstage) new file: hello.c
ishritrai@192:~/git_lab$ git commit -m "Initial commit: Add interactive hello
program"
[main (root-commit) a1b2c3d] Initial commit: Add interactive hello program 1 file
changed, 12 insertions(+) create mode 100644 hello.c
ishritrai@192:~/git_lab$ git log --oneline
a1b2c3d (HEAD -> main) Initial commit: Add interactive hello program
```

The above commands demonstrate:

- Initializing a new Git repository
- Staging files for commit using `git add`
- Checking the status of your working directory
- Creating a commit with a descriptive message
- Viewing the commit history

Best practices for commits:

- Make atomic commits (one logical change per commit)
- Write clear, descriptive commit messages
- Use the present tense in commit messages
- Keep commits focused and related to a single feature or fix

Git commit Lab Exercise

Step 1: create a file in the present working directory and add content

```
ishritrai@ISHRITs-MacBook-Pro Lab_2 % vi test.txt

ishritrai@ISHRITs-MacBook-Pro Lab_2 % cat test.txt
first line of test.txt
```

Step 2: using git init initialise a hidden git repository for tracking the files

```
ishritrai@ISHRITs-MacBook-Pro Lab_2 % git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/ishritrai/Library/Mobile Documents/com~apple~CloudDocs/Amity/Git_amity/Lab_2/.git/
```

Step 3: using git add move the file to staging area

demonstrated in screenshot below

Step 4: check git status for confirmation

demonstrated in screenshot below

```
ishritrai@ISHRITs-MacBook-Pro Lab_2 % git add test.txt

ishritrai@ISHRITs-MacBook-Pro Lab_2 % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt
```

Step 5: commit the file to a local repository

```
ishritrai@ISHRITs-MacBook-Pro Lab_2 % git commit -m "Add test.txt"
[master (root-commit) efa03a2] Add test.txt
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt
```

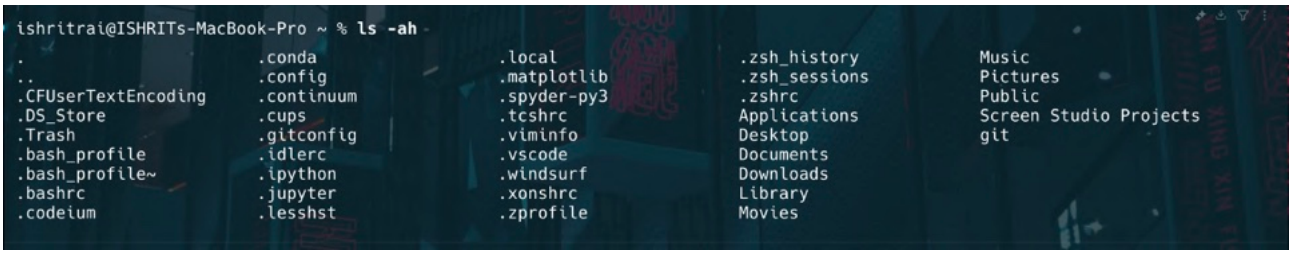
Step 6: use git log to check the commit history

```
ishritrai@192 git_commit % git log
commit f1643f47f450cf94518c6f009fc6ad92735471df (HEAD -> master)
Author: Ishrit Rai <ishrit.rai@s.amity.edu>
Date:   Sat May 31 22:38:19 2025 +0530
```

Miscellenaeous Commands

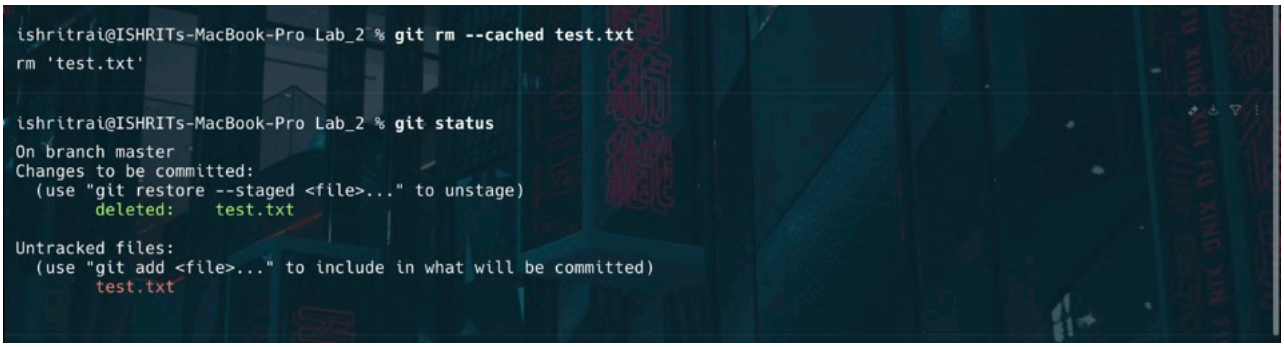
Command ls-ah

Description: used to check hidden files in a directory



Command git rm --cached <file>

Description : used to remove file from staging area



6. Git Diff

Git diff is a powerful tool for examining changes between different versions of your code. It helps developers understand what has changed, review code modifications, and resolve conflicts. Git diff can compare changes between the working directory and staging area, staged changes and the last commit, or between two specific commits.

Let's use our `calculator.c` program to demonstrate a real-world scenario. Suppose you want to add a new operation (modulus) to the calculator. We'll walk through making the change, committing it, and then using `git diff <commit1> <commit2>` to see exactly what changed in the code.

```
// calculator.c (before)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    double result;
    char operation[20];
    int error;
} Calculation;

Calculation perform_operation(double a, double b, const char* op) {
    Calculation calc = {0};
    strcpy(calc.operation, op);

    if (strcmp(op, "add") == 0) {
        calc.result = a + b;
    } else if (strcmp(op, "subtract") == 0) {
        calc.result = a - b;
    } else if (strcmp(op, "multiply") == 0) {
        calc.result = a * b;
    } else if (strcmp(op, "divide") == 0) {
        if (b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = a / b;
    } else {
        calc.error = 1;
        return calc;
    }

    return calc;
}

int main() {
    double num1, num2;
    char operation[20];
```

```
    printf("Enter first number: ");
    scanf("%lf", &num1);
    printf("Enter second number: ");
    scanf("%lf", &num2);
    printf("Enter operation (add/subtract/multiply/divide): ");
    scanf("%19s", operation);

    Calculation result = perform_operation(num1, num2, operation);

    if (result.error) {
        printf("Error: Invalid operation or division by zero\n");
        return 1;
    }

    printf("Result: %.2f\n", result.result);
    return 0;
}
```

Now, let's add modulus (%) support to the calculator and see the difference:

```
// calculator.c (after)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    double result;
    char operation[20];
    int error;
} Calculation;

Calculation perform_operation(double a, double b, const char* op) {
    Calculation calc = {0};
    strcpy(calc.operation, op);

    if (strcmp(op, "add") == 0) {
        calc.result = a + b;
    } else if (strcmp(op, "subtract") == 0) {
        calc.result = a - b;
    } else if (strcmp(op, "multiply") == 0) {
        calc.result = a * b;
    } else if (strcmp(op, "divide") == 0) {
        if (b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = a / b;
    } else if (strcmp(op, "modulus") == 0) {
        if ((int)b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = (int)a % (int)b;
```

```
    } else {
        calc.error = 1;
        return calc;
    }

    return calc;
}

int main() {
    double num1, num2;
    char operation[20];

    printf("Enter first number: ");
    scanf("%lf", &num1);
    printf("Enter second number: ");
    scanf("%lf", &num2);
    printf("Enter operation (add/subtract/multiply/divide/modulus): ");
    scanf("%19s", operation);

    Calculation result = perform_operation(num1, num2, operation);

    if (result.error) {
        printf("Error: Invalid operation or division by zero\n");
        return 1;
    }

    printf("Result: %.2f\n", result.result);
    return 0;
}
```

After making and committing the change, you can use `git diff <commit1> <commit2>` to see the exact code differences:

```
ishritrai@192:~/git_lab$ git add calculator.c
ishritrai@192:~/git_lab$ git commit -m "Initial commit: Add calculator program"
[main 1a2b3c4] Initial commit: Add calculator program
1 file changed, 45 insertions(+)
create mode 100644 calculator.c
ishritrai@192:~/git_lab$ # (edit calculator.c to add modulus support)
ishritrai@192:~/git_lab$ git add calculator.c
ishritrai@192:~/git_lab$ git commit -m "Add modulus operation to calculator"
[main 2b3c4d5] Add modulus operation to calculator
1 file changed, 8 insertions(+), 1 deletion(-)
ishritrai@192:~/git_lab$ git diff 1a2b3c4 2b3c4d5
diff --git a/calculator.c b/calculator.c
index 1234567..89abcde 100644
--- a/calculator.c
+++ b/calculator.c
@@ -13,10 +13,18 @@ Calculation perform_operation(double a, double b, const char*
op) {
if (strcmp(op, "add") == 0) {
calc.result = a + b;
} else if (strcmp(op, "subtract") == 0) {
```

```
calc.result = a - b;
} else if (strcmp(op, "multiply") == 0) {
calc.result = a * b;
} else if (strcmp(op, "divide") == 0) {
if (b == 0) {
calc.error = 1;
return calc;
}
calc.result = a / b;
+ } else if (strcmp(op, "modulus") == 0) {
+ if ((int)b == 0) {
+ calc.error = 1;
+ return calc;
+ }
+ calc.result = (int)a % (int)b;
} else {
calc.error = 1;
return calc;
}
return calc;
}
@@ -32,7 +40,8 @@ int main() {
printf("Enter first number: ");
scanf("%lf", &num1);
printf("Enter second number: ");
- printf("Enter operation (add/subtract/multiply/divide): ");
+ printf("Enter operation (add/subtract/multiply/divide/modulus): ");
scanf("%19s", operation);
Calculation result = perform_operation(num1, num2, operation);
if (result.error) {
printf("Error: Invalid operation or division by zero\n");
return 1;
}
printf("Result: %.2f\n", result.result);
return 0;
}
```

The above demonstration shows:

- How to use `git diff <commit1> <commit2>` to compare two versions of a real C program
- How code changes (like adding a new operation) are reflected in the diff output
- Line-by-line, organized output for clarity
- Understanding diff output format:
    - `---` and `+++` indicate the files being compared
    - `@@` lines show the location and size of changes
    - `-` lines show removed content
    - `+` lines show added content

This approach demonstrates a realistic workflow: making a meaningful code change, committing it, and using git diff to review exactly what was modified in your C project.

Lab Session 3: Git Diff

Mount Point

Point from where we can access the desired folder directly

/Users/ishritrai/Library/Mobile Documents/com~apple~CloudDocs/Amity/Git_Amity/Lab_2

In the above file path mount point of Lab_2 is Git_Amity

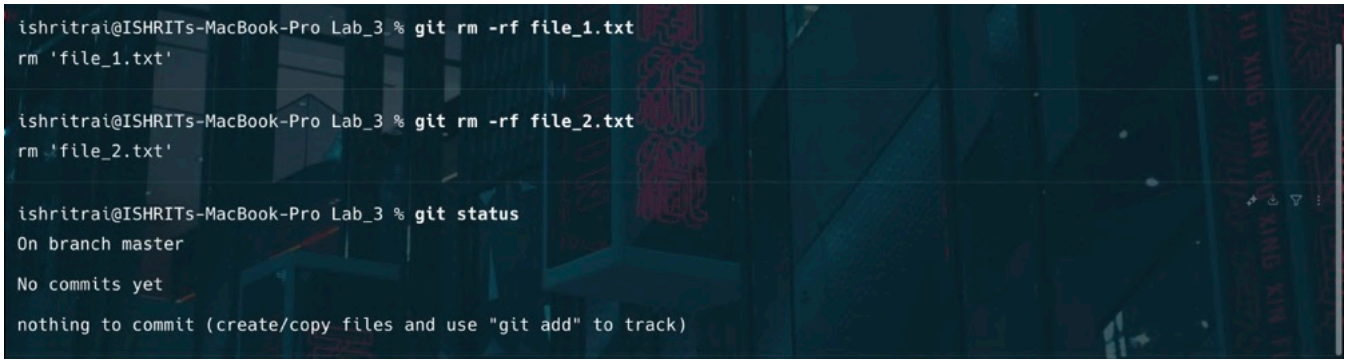
Lets make modifications in our repository before demonstrating git diff

Command touch <file_name>

Used to create a file without any content

Command git rm -rf <file_name>

Used to remove a file from git tracking



Task 1: make two commits in a directory

Step 1: create two files with content in a directory

```
ishritrai@ISHRITs-MacBook-Pro Lab_3 % ls
file_1.txt      file_2.txt

ishritrai@ISHRITs-MacBook-Pro Lab_3 % cat file_1.txt file_2.txt
first line of file_1.txt
second line of file_1.txt
first line of file_2.txt
second line of file_2.txt
third line of file_2.txt
fourth line of file_2.txt
```

Step 2: use git add. Command to add both files in the staging area
Step 3: use git commit -m to commit both files to local

```
ishritrai@ISHRITs-MacBook-Pro Lab_3 % git add .

ishritrai@ISHRITs-MacBook-Pro Lab_3 % git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file_1.txt
        new file:   file_2.txt
```

repository
Step 4: use git log command to verify the commit history

```
ishritrai@ISHRITs-MacBook-Pro Lab_3 % git commit -m "Add file_1.txt" file_1.txt
[master (root-commit) 25a7385] Add file_1.txt
 1 file changed, 2 insertions(+)
 create mode 100644 file_1.txt

ishritrai@ISHRITs-MacBook-Pro Lab_3 % git commit -m "Add file_2.txt" file_2.txt
[master 21ca055] Add file_2.txt
 1 file changed, 4 insertions(+)
 create mode 100644 file_2.txt
```

```
ishritrai@ISHRITs-MacBook-Pro Lab_3 % git log
commit 21ca055b213c44126de5e2e017dbfa738e731b96 (HEAD -> master)
Author: Ishrit Rai <raiishrit@gmail.com>
Date:   Tue Feb 18 13:15:44 2025 +0530

    Add file_2.txt

commit 25a7385f10d894a63f409f1257e3358d98cbdcdb
Author: Ishrit Rai <raiishrit@gmail.com>
Date:   Tue Feb 18 13:15:37 2025 +0530

    Add file_1.txt
```

Step 5: use git log- - oneline for generating shorter commit id

Task 2: compare two commits in a directory

Use git diff along with the commit id generated from git log - - online

7. Working with Remotes

Working with remotes in Git allows you to collaborate with others by synchronizing your local repository with repositories hosted on remote servers (such as GitHub, GitLab, or Bitbucket). The most common remote operations are `git clone` (to copy a remote repository), `git push` (to upload your changes), and `git pull` (to fetch and merge changes from the remote).

Let's demonstrate a realistic collaborative workflow using our `calculator.c` project. Suppose you want to contribute to a shared repository hosted on GitHub. You'll clone the repository, make a change, push it, and then pull updates made by a collaborator.

```
// calculator.c (collaborator adds a new feature)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    double result;
    char operation[20];
    int error;
} Calculation;

Calculation perform_operation(double a, double b, const char* op) {
    Calculation calc = {0};
    strcpy(calc.operation, op);
    if (strcmp(op, "add") == 0) {
        calc.result = a + b;
    } else if (strcmp(op, "subtract") == 0) {
        calc.result = a - b;
    } else if (strcmp(op, "multiply") == 0) {
        calc.result = a * b;
    } else if (strcmp(op, "divide") == 0) {
        if (b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = a / b;
    } else if (strcmp(op, "modulus") == 0) {
        if ((int)b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = (int)a % (int)b;
    } else if (strcmp(op, "power") == 0) {
        calc.result = pow(a, b);
    } else {
        calc.error = 1;
        return calc;
    }
    return calc;
```

```
}

int main() {
    double num1, num2;
    char operation[20];
    printf("Enter first number: ");
    scanf("%lf", &num1);
    printf("Enter second number: ");
    scanf("%lf", &num2);
    printf("Enter operation (add/subtract/multiply/divide/modulus/power): ");
    scanf("%19s", operation);
    Calculation result = perform_operation(num1, num2, operation);
    if (result.error) {
        printf("Error: Invalid operation or division by zero\n");
        return 1;
    }
    printf("Result: %.2f\n", result.result);
    return 0;
}
```

Here is a step-by-step terminal simulation of a collaborative workflow:

```
ishritrai@192:~$ git clone https://github.com/example/calculator.git
Cloning into 'calculator'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 12 (delta 2), reused 12 (delta 2), pack-reused 0
Unpacking objects: 100% (12/12), done.
ishritrai@192:~$ cd calculator
ishritrai@192:~/calculator$ # Make a change: add a comment to calculator.c
ishritrai@192:~/calculator$ echo "// Collaborative edit by Ishrit Rai" | cat -
calculator.c > temp && mv temp calculator.c
ishritrai@192:~/calculator$ git add calculator.c
ishritrai@192:~/calculator$ git commit -m "Add collaborative comment to
calculator.c"
[main 3e4f5g6] Add collaborative comment to calculator.c
1 file changed, 1 insertion(+)
ishritrai@192:~/calculator$ git push origin main
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 350 bytes | 350.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/example/calculator.git
2b3c4d5..3e4f5g6 main -> main
ishritrai@192:~/calculator$ # Collaborator pushes a new feature (power operation)
ishritrai@192:~/calculator$ git pull origin main
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 2), reused 6 (delta 2), pack-reused 0
```

```
Unpacking objects: 100% (6/6), done.
From https://github.com/example/calculator
3e4f5g6..4h5i6j7 main -> origin/main
Updating 3e4f5g6..4h5i6j7
Fast-forward
calculator.c | 10 ++++++++++
1 file changed, 10 insertions(+)
```

The above demonstration shows:

- Cloning a remote repository to start collaborating
- Making and committing a change to the C project
- Pushing your changes to the remote repository
- Pulling updates made by a collaborator (e.g., a new power operation in the calculator)
- How remote operations integrate with real C project development

This workflow is essential for team-based software development, ensuring everyone stays up-to-date and can contribute effectively.

Lab Session 5 : Working with remotes

Step1: Make 4 commits and compare them using git diff

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git log --oneline
c9175db (HEAD -> master) ds Store
6104b3c DSU-4
d97972d DSU-3
c2f4a59 DSU-2
becdbc1 DSU-1
```

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git diff c2f4a59 becdbc1
diff --git a/file_2.txt b/file_2.txt
deleted file mode 100644
index 15fef7c..0000000
--- a/file_2.txt
+++ /dev/null
@@ -1,67 +0,0 @@
-#include<bits/stdc++.h>
-using namespace std;
-
-const int N = 3e5 + 9;
-
-struct DSU {
-  vector<vector<pair<int, int>>> par;
-  int time = 0; //initial time
-  DSU(int n) : par(n + 1, {{-1, 0}}) {}
-  bool merge(int u, int v) {
-    ++time;
-    if ((u = root(u, time)) == (v = root(v, time))) return 0;
-    if (par[u].back().first > par[v].back().first) swap(u, v);
-    par[u].push_back({par[u].back().first + par[v].back().first, time});
-    par[v].push_back({u, time}); //par[v] = u
-    return 1;
-  }
-  bool same(int u, int v, int t) {
-    return root(u, t) == root(v, t);
-  }
-  int root(int u, int t) { //root of u at time t
-    if (par[u].back().first >= 0 && par[u].back().second <= t) return root(par[u].back().first, t);
-    return u;
-  }
-  int size(int u, int t) { //size of the component of u at time t
-    u = root(u, t);
-    int l = 0, r = (int) par[u].size() - 1, ans = 0;
-    while (l <= r) {
-      int mid = l + r >> 1;
-      if (par[u][mid].second <= t) ans = mid, l = mid + 1;
-      else r = mid - 1;
-    }
-    return -par[u][ans].first;
-  }
```

Step 2: Use git remote command to establish a connection between local Git repository and a remote repository

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git remote add git_amity https://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@github.com/IshritRai/Git_Amity
```

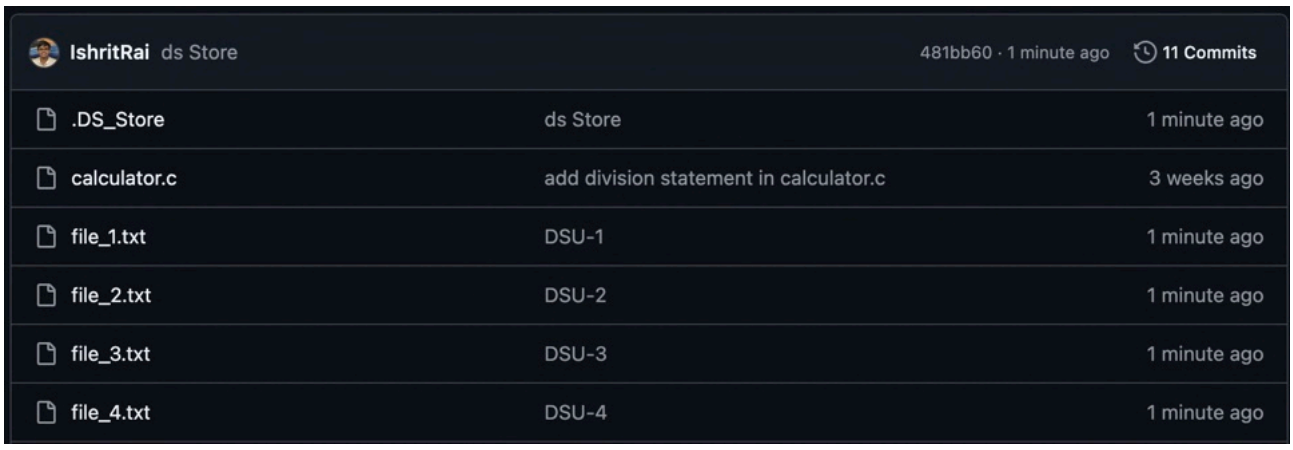
Step 3: Use The `git push` command is used to upload local repository content to a remote repository

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git push git_amity master
Enumerating objects: 16, done.
Counting objects: 100% (16/16), done.
Delta compression using up to 12 threads
Compressing objects: 100% (15/15), done.
Writing objects: 100% (15/15), 4.66 KiB | 2.33 MiB/s, done.
Total 15 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (4/4), done.
To https://github.com/IshritRai/Git_Amity
   84ca08d..481bb60  master -> master
```

Step 4:Confirming the remote connection with git remote

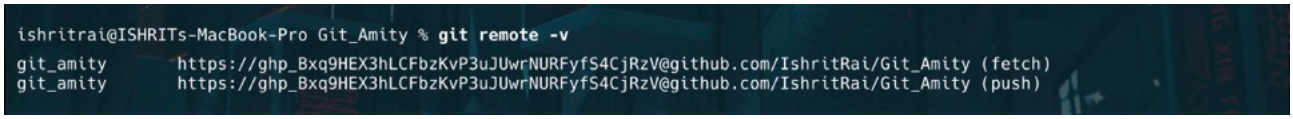
```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git remote
git_amity
```

Step 5: Checking the commits made on GitHub account



Git pull is used to fetch and integrate changes which are in the remote repository to local repository

Git remote-v: used to view all remote repositories in a directory lists all configured remote repositories along with their corresponding URL



8. Branching and Merging

Branching in Git allows you to diverge from the main line of development and work on features, bug fixes, or experiments in isolation. Merging brings these changes back together. This workflow is essential for collaborative and parallel development, enabling teams to work independently without interfering with each other's progress.

Let's demonstrate branching and merging with our `calculator.c` project. We'll create a feature branch to add a new scientific function (square root), then merge it back into the main branch.

```
// calculator.c (feature/sqrt branch adds square root)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef struct {
    double result;
    char operation[20];
    int error;
} Calculation;

Calculation perform_operation(double a, double b, const char* op) {
    Calculation calc = {0};
    strcpy(calc.operation, op);
    if (strcmp(op, "add") == 0) {
        calc.result = a + b;
    } else if (strcmp(op, "subtract") == 0) {
        calc.result = a - b;
    } else if (strcmp(op, "multiply") == 0) {
        calc.result = a * b;
    } else if (strcmp(op, "divide") == 0) {
        if (b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = a / b;
    } else if (strcmp(op, "modulus") == 0) {
        if ((int)b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = (int)a % (int)b;
    } else if (strcmp(op, "power") == 0) {
        calc.result = pow(a, b);
    } else if (strcmp(op, "sqrt") == 0) {
        if (a < 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = sqrt(a);
```

```
    } else {
        calc.error = 1;
        return calc;
    }
    return calc;
}

int main() {
    double num1 = 0, num2 = 0;
    char operation[20];
    printf("Enter first number: ");
    scanf("%lf", &num1);
    printf("Enter second number (or 0 for sqrt): ");
    scanf("%lf", &num2);
    printf("Enter operation (add/subtract/multiply/divide/modulus/power/sqrt): ");
    scanf("%19s", operation);
    Calculation result = perform_operation(num1, num2, operation);
    if (result.error) {
        printf("Error: Invalid operation or input\n");
        return 1;
    }
    printf("Result: %.2f\n", result.result);
    return 0;
}
```

Here is a step-by-step terminal simulation of branching and merging:

```
ishritrai@192:~/calculator$ git checkout -b feature/sqrt
Switched to a new branch 'feature/sqrt'
ishritrai@192:~/calculator$ # Edit calculator.c to add sqrt support
ishritrai@192:~/calculator$ git add calculator.c
ishritrai@192:~/calculator$ git commit -m "Add square root operation"
[feature/sqrt 5f6g7h8] Add square root operation
1 file changed, 10 insertions(+)
ishritrai@192:~/calculator$ git checkout main
Switched to branch 'main'
ishritrai@192:~/calculator$ git merge feature/sqrt
Updating 4h5i6j7..5f6g7h8
Fast-forward
calculator.c | 10 ++++++++++
1 file changed, 10 insertions(+)
ishritrai@192:~/calculator$ git branch -d feature/sqrt
Deleted branch feature/sqrt (was 5f6g7h8).
```

The above demonstration shows:

- Creating a feature branch for isolated development
- Making and committing a significant change to the C project
- Merging the feature branch back into the main branch
- Cleaning up by deleting the merged branch
- How branching and merging support parallel and collaborative development

This workflow is fundamental for managing new features, bug fixes, and experiments in a safe and organized way.

Lab Session 6 : git branching

Branch: pointer to a commit

Pointer: connects two memory address where at least one variable must have an active memory address

Head: branch on which the last commit is made

Git branch command used to view the existing branches in the git repository

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git branch
* master
```

Git checkout command used to switch the currently active branch to another branch. Here we want to create a new branch from a particular commit

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git checkout 6104b3c
Note: switching to '6104b3c'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 6104b3c DSU-4
```

Git branch test_1: used to create branch with name test_1
Confirm the created branch by using git branch command to view all the branches

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git branch
* (HEAD detached at 6104b3c)
  master
  test_1
```

Use git_checkout command to pivot to that particular branch

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git checkout test_1
Switched to branch 'test_1'
```

Make commits in test_1 branch

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git add file_5.txt

ishritrai@ISHRITs-MacBook-Pro Git_Amity % git commit -m "Add file_5.txt"
[test_1 f9c7445] Add file_5.txt
 1 file changed, 1 insertion(+)
 create mode 100644 file_5.txt
```

Viewing the commits on a particular branch

```
ishritrai@ISHRITs-MacBook-Pro Git_Amity % git log --oneline
f9c7445 (HEAD -> test_1) Add file_5.txt
6104b3c DSU-4
d97972d DSU-3
c2f4a59 DSU-2
becdbc1 DSU-1

ishritrai@ISHRITs-MacBook-Pro Git_Amity % git checkout test_1
Already on 'test_1'

ishritrai@ISHRITs-MacBook-Pro Git_Amity % git status
On branch test_1
nothing to commit, working tree clean
```

Merging Branches

Step 1: committing changes in hello .txt on main branch

```
ishritrai@192 git_commit % vi hello.txt

ishritrai@192 git_commit % git add .

ishritrai@192 git_commit % git commit -m "update hello.txt"
[master b54569c] update hello.txt
 1 file changed, 2 insertions(+), 1 deletion(-)

ishritrai@192 git_commit % git log
commit b54569c25147057f63bb7acca4205c8d0fd98783 (HEAD -> master)
Author: Ishrit Rai <ishrit.rai@s.amity.edu>
Date:   Sun Jun 1 07:31:51 2025 +0530

    update hello.txt

commit f1643f47f450cf94518c6f009fc6ad92735471df
Author: Ishrit Rai <ishrit.rai@s.amity.edu>
Date:   Sat May 31 22:38:19 2025 +0530

    add test.txt
```

Step 2: Creating test branch

```
ishritrai@192 git_commit % git branch test

ishritrai@192 git_commit % git branch
* master
  test
```

Step 3: Switching to test branch

```
ishritrai@192 git_commit % git checkout test
Switched to branch 'test'
```

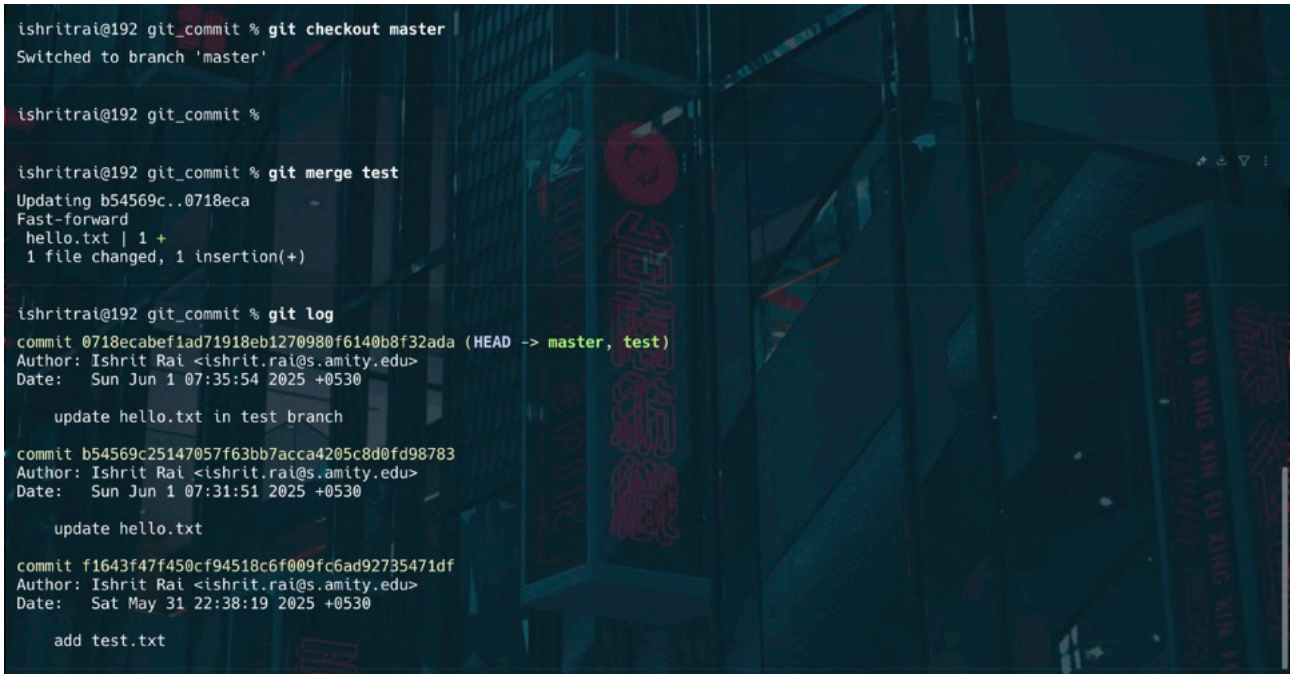
Step 4: making changes in hello.txt in test branch

```
ishritrai@192 git_commit % vi hello.txt

ishritrai@192 git_commit % cat hello.txt
hi this is the first line of hello.txt in main branch
second line of hello.txt in main branch
third line ADDED in test branch
```

Step 5: committing the changes

```
ishritrai@192 git_commit % git add .

ishritrai@192 git_commit % git commit -m "update hello.txt in test branch"
[test 0718eca] update hello.txt in test branch
 1 file changed, 1 insertion(+)

ishritrai@192 git_commit % git log
commit 0718ecabef1ad71918eb1270980f6140b8f32ada (HEAD -> test)
Author: Ishrit Rai <ishrit.rai@s.amity.edu>
Date:   Sun Jun 1 07:35:54 2025 +0530

    update hello.txt in test branch

commit b54569c25147057f63bb7acca4205c8d0fd98783 (master)
Author: Ishrit Rai <ishrit.rai@s.amity.edu>
Date:   Sun Jun 1 07:31:51 2025 +0530

    update hello.txt

commit f1643f47f450cf94518c6f009fc6ad92735471df
Author: Ishrit Rai <ishrit.rai@s.amity.edu>
Date:   Sat May 31 22:38:19 2025 +0530

    add test.txt
```

Step 6: merging the test branch



9. Merge Conflicts

Merge conflicts occur when changes from different branches cannot be automatically reconciled by Git. This typically happens when two branches modify the same lines in a file. Resolving merge conflicts is a critical skill for collaborative development.

Let's demonstrate a merge conflict scenario with `calculator.c`. Suppose two branches make different changes to the same function. We'll walk through creating the conflict, seeing Git's response, and resolving it.

```
// calculator.c (main branch)
printf("Result: %.2f\n", result.result);
return 0;
}
```

```
// calculator.c (feature/pretty-output branch)
printf("==== Calculation Result ====\n");
printf("Result: %.2f\n", result.result);
printf("============================\n");
return 0;
}
```

```
// calculator.c (feature/author branch)
printf("Result: %.2f\n", result.result);
printf("-- Calculated by Ishrit Rai --\n");
return 0;
}
```

Here is a step-by-step terminal simulation of a merge conflict and its resolution:

```
ishritrai@192:~/calculator$ git checkout -b feature/pretty-output
Switched to a new branch 'feature/pretty-output'
ishritrai@192:~/calculator$ # Edit calculator.c to add pretty output
ishritrai@192:~/calculator$ git add calculator.c
ishritrai@192:~/calculator$ git commit -m "Add pretty output formatting"
[feature/pretty-output 6a7b8c9] Add pretty output formatting
1 file changed, 3 insertions(+)
ishritrai@192:~/calculator$ git checkout main
Switched to branch 'main'
ishritrai@192:~/calculator$ git checkout -b feature/author
Switched to a new branch 'feature/author'
ishritrai@192:~/calculator$ # Edit calculator.c to add author line
ishritrai@192:~/calculator$ git add calculator.c
ishritrai@192:~/calculator$ git commit -m "Add author line to output"
[feature/author 7b8c9d0] Add author line to output
1 file changed, 1 insertion(+)
ishritrai@192:~/calculator$ git checkout main
Switched to branch 'main'
ishritrai@192:~/calculator$ git merge feature/pretty-output
Updating 5f6g7h8..6a7b8c9
```

```
Fast-forward
calculator.c | 3 +++
1 file changed, 3 insertions(+)
ishritrai@192:~/calculator$ git merge feature/author
Auto-merging calculator.c
CONFLICT (content): Merge conflict in calculator.c
Automatic merge failed; fix conflicts and then commit the result.
ishritrai@192:~/calculator$ # Open calculator.c and resolve the conflict
ishritrai@192:~/calculator$ git add calculator.c
ishritrai@192:~/calculator$ git commit -m "Resolve merge conflict: combine pretty
output and author line"
[main 8c9d0e1] Resolve merge conflict: combine pretty output and author line
```

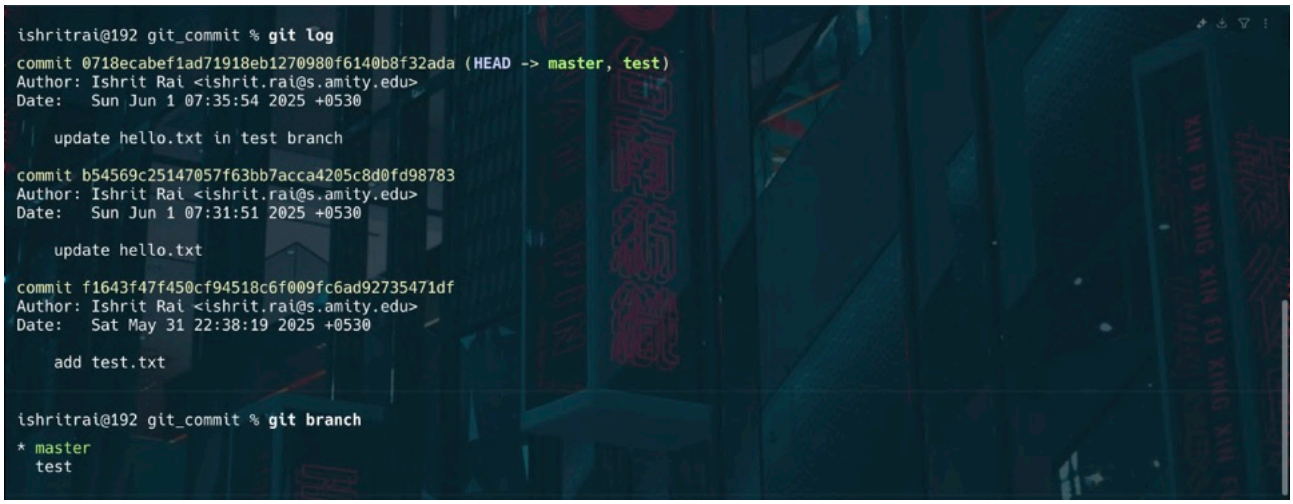
The above demonstration shows:

- How two branches can make conflicting changes to the same file
- How Git reports and marks a merge conflict
- How to resolve the conflict by editing the file and combining both changes
- How to complete the merge after resolving the conflict

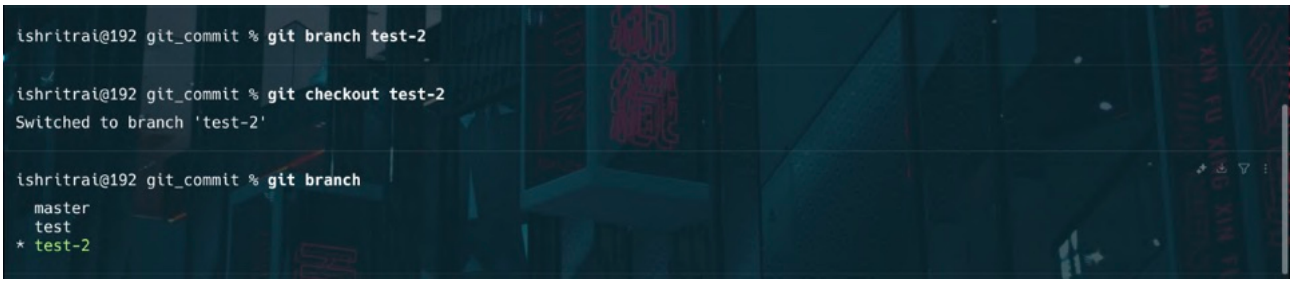
Merge conflicts are a normal part of collaborative development. Understanding how to resolve them ensures smooth teamwork and project progress.

Lab Exercise 7: Merge Conflicts

Below screenshot reflects the current configuration of our repository



Step 1: create another test branch and switch to that branch



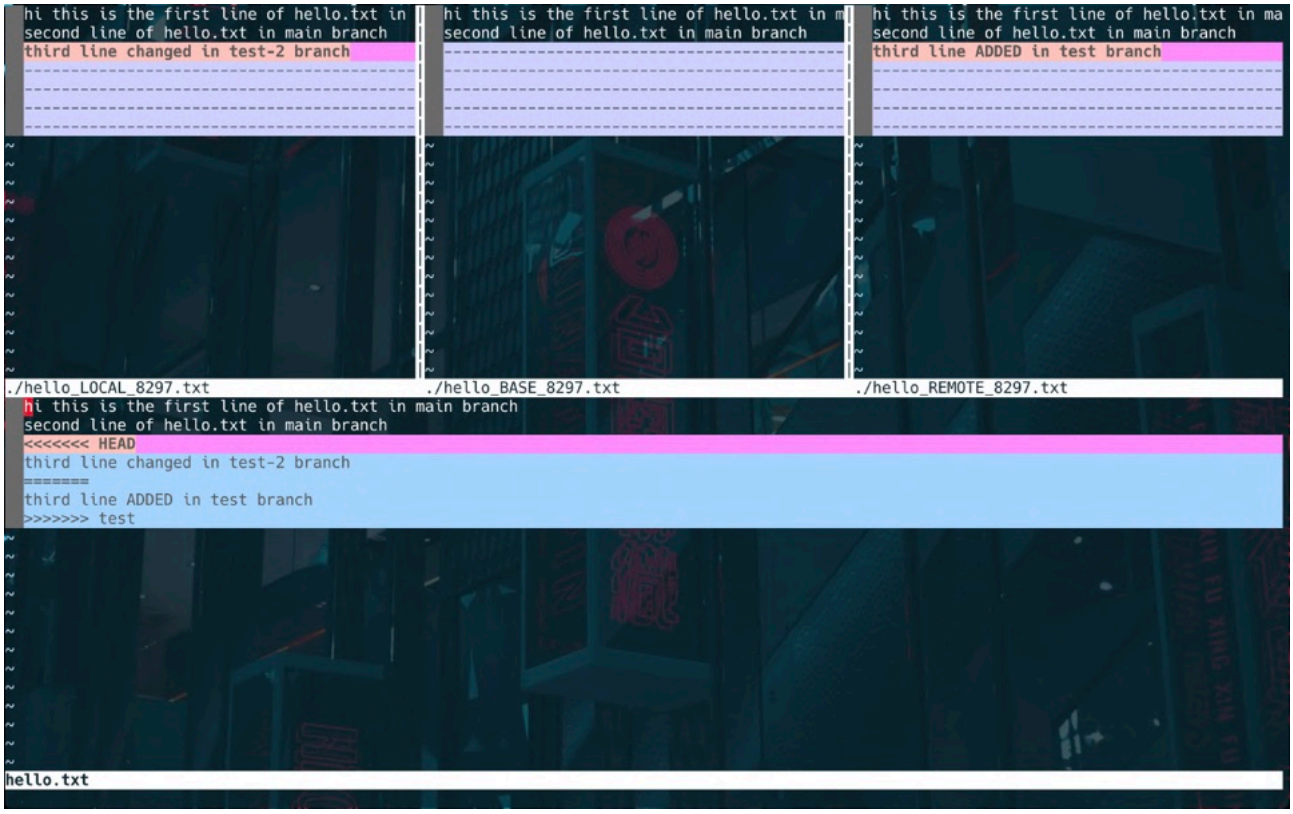
Step2: modify hello.txt and commit those changes

Step 3: merge test branch to test-2 branch. A merge conflict will appear

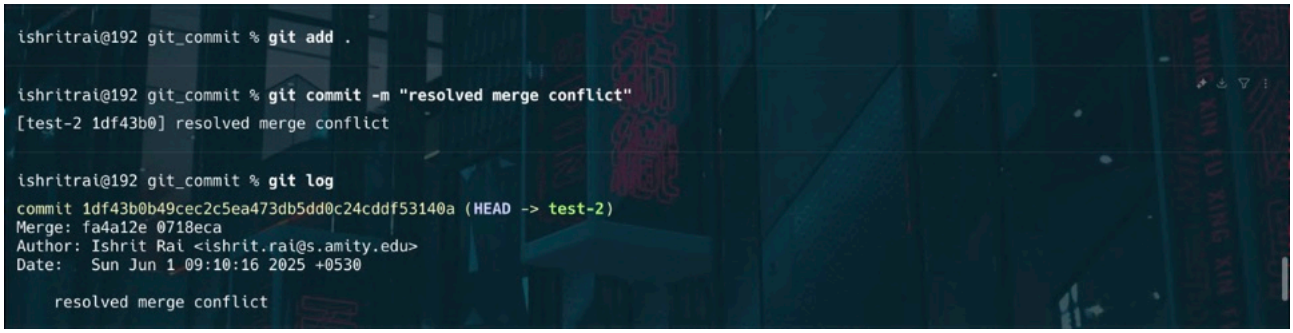
```
ishritrai@192 git_commit % git merge test
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Step 4: Resolve the merge conflict using git merge tool

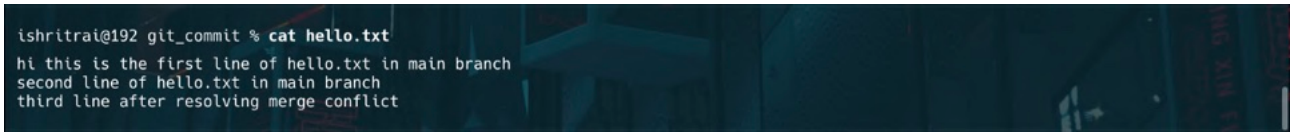
```
ishritrai@192 git_commit % git mergetool --tool=vimdiff
Merging:
hello.txt

Normal merge conflict for 'hello.txt':
  {local}: modified file
  {remote}: modified file
4 files to edit
```



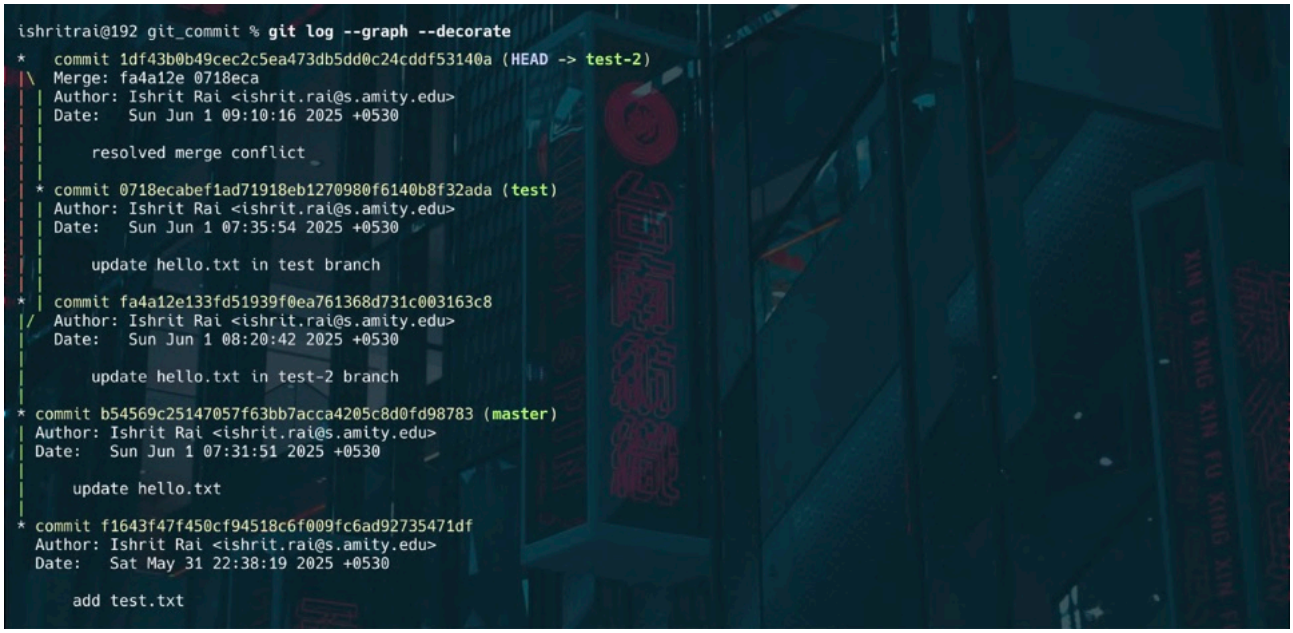
Step 5: commit the changes



Step 6: checking hello.txt for confirming the merge



Step 7: using git log - - graph - - decorate for visual representation of branches.



11. Forking and Cloning

Forking and cloning are essential for contributing to open source and collaborative projects. Forking creates a personal copy of a repository under your account, while cloning downloads that repository to your local machine. This workflow allows you to experiment, develop features, and propose changes without affecting the original project.

Let's demonstrate forking and cloning with the `calculator.c` project. Suppose you want to experiment with a new feature (logarithm operation) in your own fork before submitting a pull request.

```
// calculator.c (feature/log branch adds logarithm)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

typedef struct {
    double result;
    char operation[20];
    int error;
} Calculation;

Calculation perform_operation(double a, double b, const char* op) {
    Calculation calc = {0};
    strcpy(calc.operation, op);
    if (strcmp(op, "add") == 0) {
        calc.result = a + b;
    } else if (strcmp(op, "subtract") == 0) {
        calc.result = a - b;
    } else if (strcmp(op, "multiply") == 0) {
        calc.result = a * b;
    } else if (strcmp(op, "divide") == 0) {
        if (b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = a / b;
    } else if (strcmp(op, "log") == 0) {
        if (a <= 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = log(a);
    } else {
        calc.error = 1;
        return calc;
    }
    return calc;
}

int main() {
```

```
    double num1 = 0, num2 = 0;
    char operation[20];
    printf("Enter first number: ");
    scanf("%lf", &num1);
    printf("Enter second number (or 0 for unary ops): ");
    scanf("%lf", &num2);
    printf("Enter operation (add/subtract/multiply/divide/log): ");
    scanf("%19s", operation);
    Calculation result = perform_operation(num1, num2, operation);
    if (result.error) {
        printf("Error: Invalid operation or input\n");
        return 1;
    }
    printf("Result: %.2f\n", result.result);
    return 0;
}
```

Here is a step-by-step terminal simulation of forking and cloning:

```
ishritrai@192:~$ # On GitHub, click 'Fork' on the original calculator repository
ishritrai@192:~$ git clone https://github.com/ishritrai/calculator.git
Cloning into 'calculator'...
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 25 (delta 6), reused 25 (delta 6), pack-reused 0
Unpacking objects: 100% (25/25), done.
ishritrai@192:~$ cd calculator
ishritrai@192:~/calculator$ git checkout -b feature/log
Switched to a new branch 'feature/log'
ishritrai@192:~/calculator$ # Edit calculator.c to add logarithm support
ishritrai@192:~/calculator$ git add calculator.c
ishritrai@192:~/calculator$ git commit -m "Add logarithm operation"
[feature/log 1b2c3d4] Add logarithm operation
1 file changed, 8 insertions(+)
ishritrai@192:~/calculator$ git push origin feature/log
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 900 bytes | 900.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
To https://github.com/ishritrai/calculator.git
* [new branch] feature/log -> feature/log
```

The above demonstration shows:

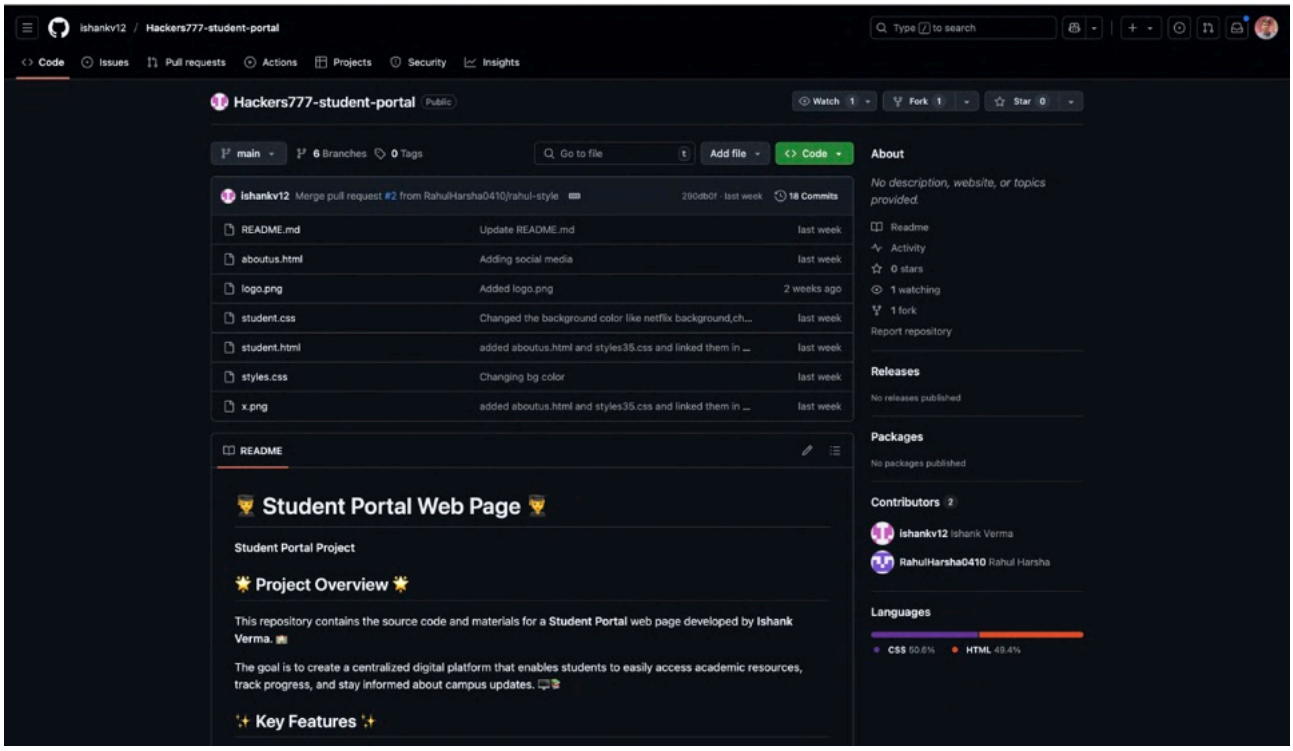
- Forking a repository to your own GitHub account
- Cloning your fork to your local machine
- Creating a feature branch for experimentation
- Making and committing a significant change to the C project
- Pushing your branch to your fork on GitHub

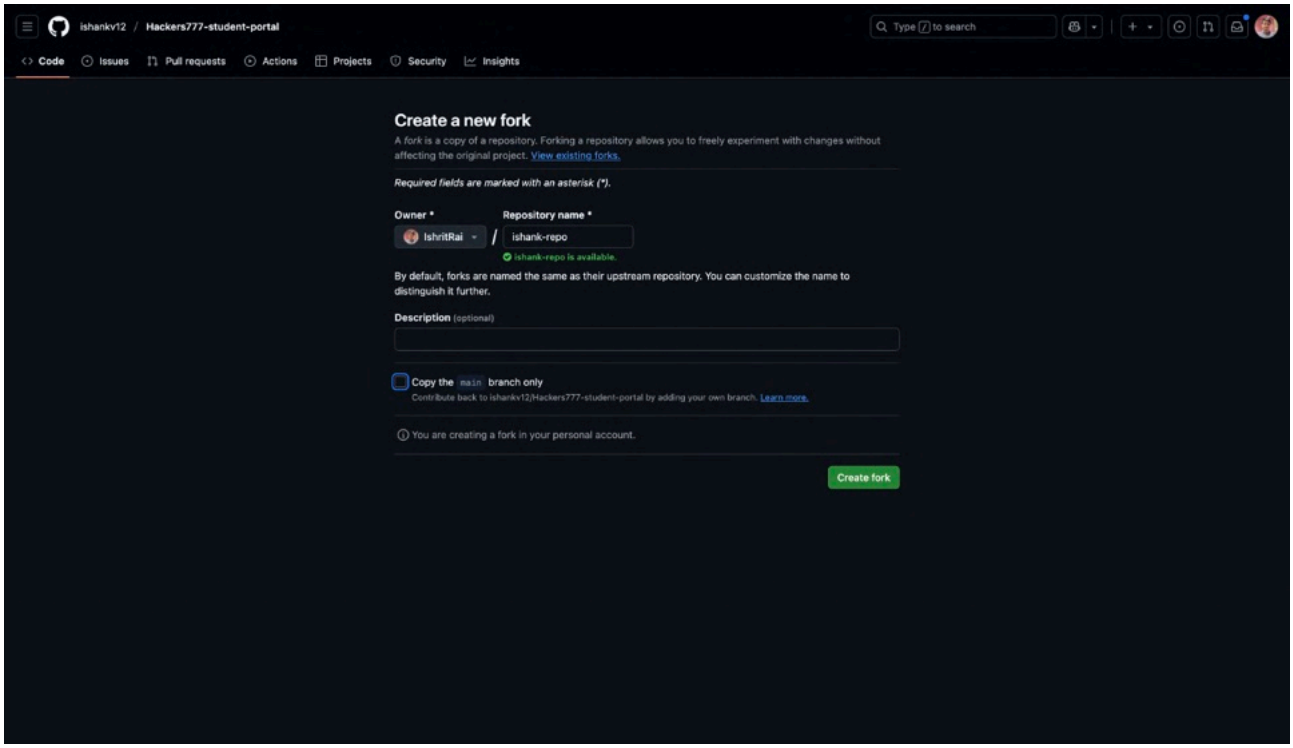
- How forking and cloning enable safe experimentation and contribution

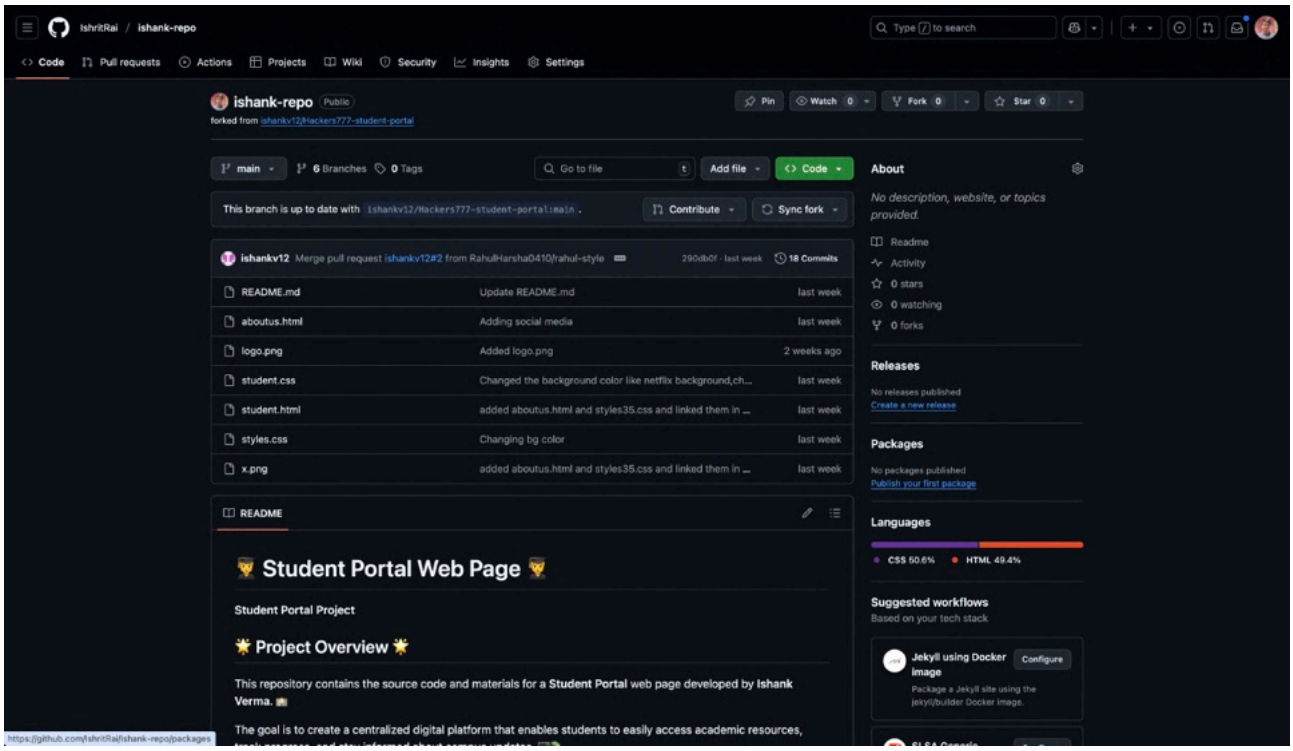
Forking and cloning are fundamental for open source collaboration, allowing you to innovate and contribute without risk to the original project.

Lab Exercise 8: Fork Clone Workflow and Sending Pull request

Step1: Navigate to a desired repository and fork it

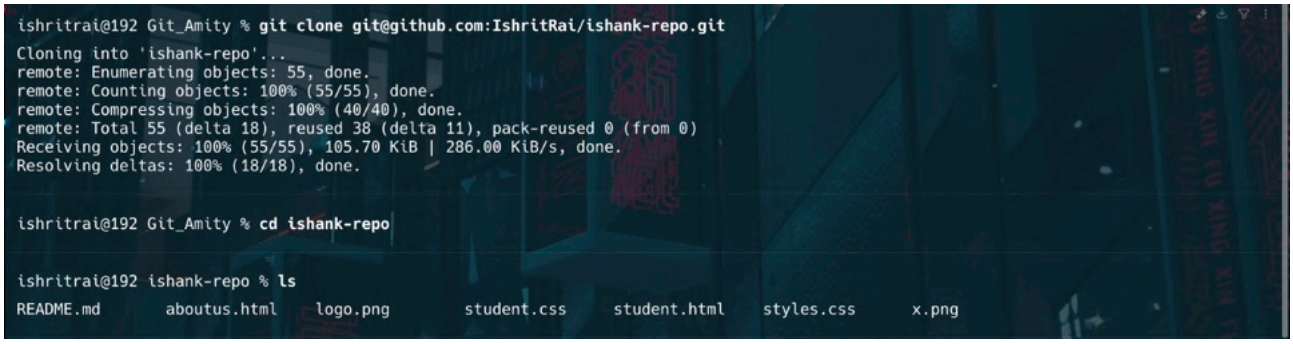






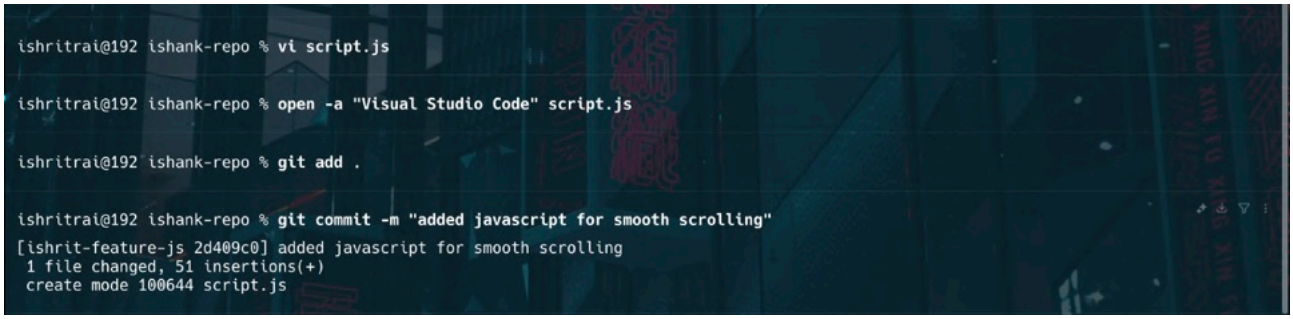
Forked repo

Step 2: using git clone command create a copy of the repository in the local system



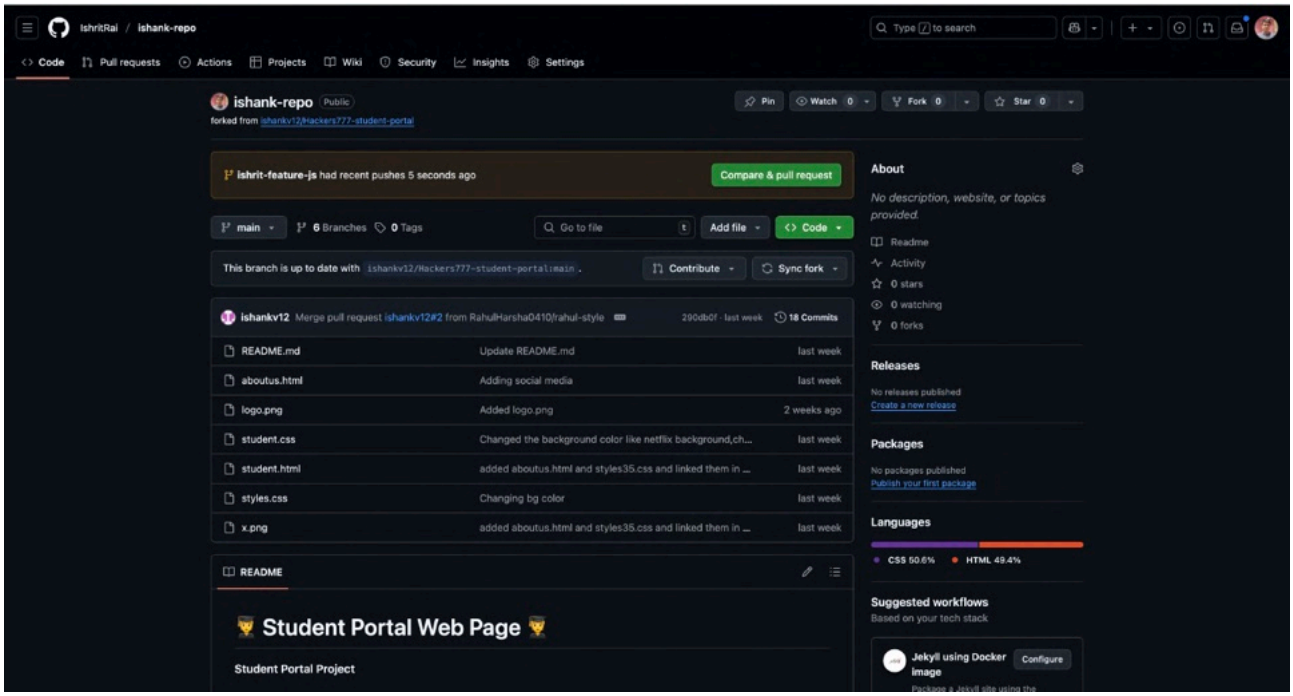
step 3: create your own feature branch for making the required changes

Step 4: commit those changes

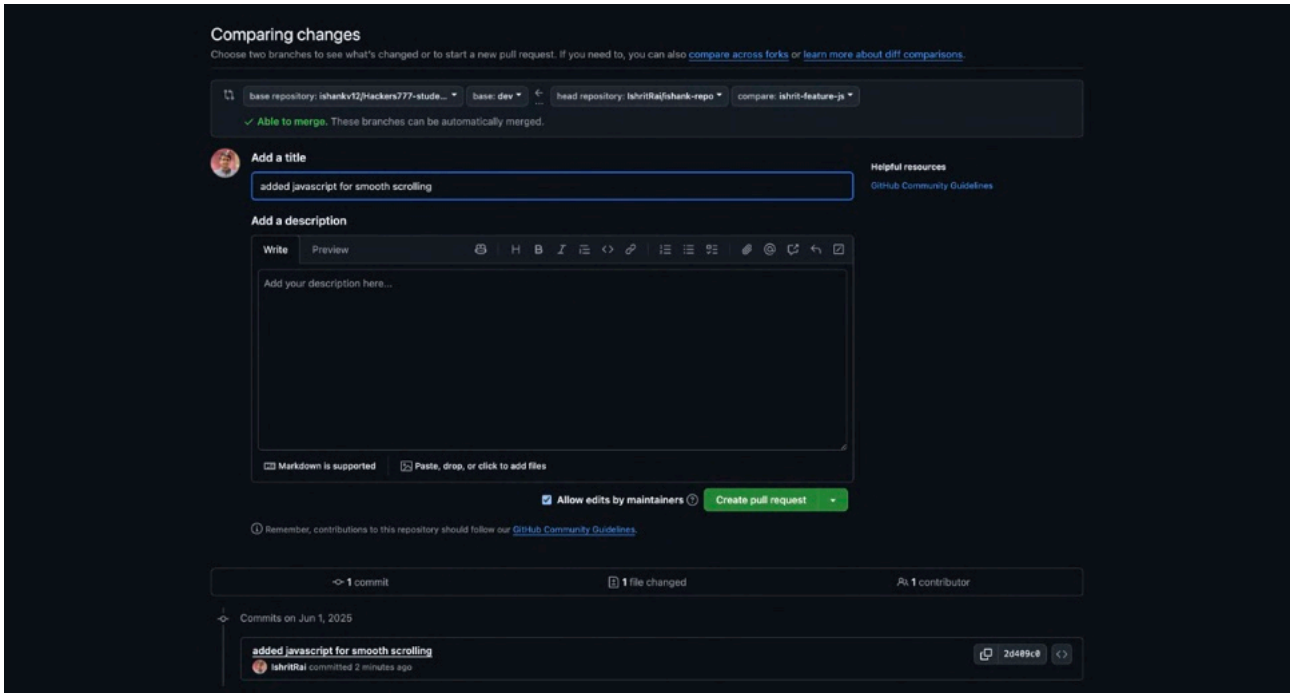


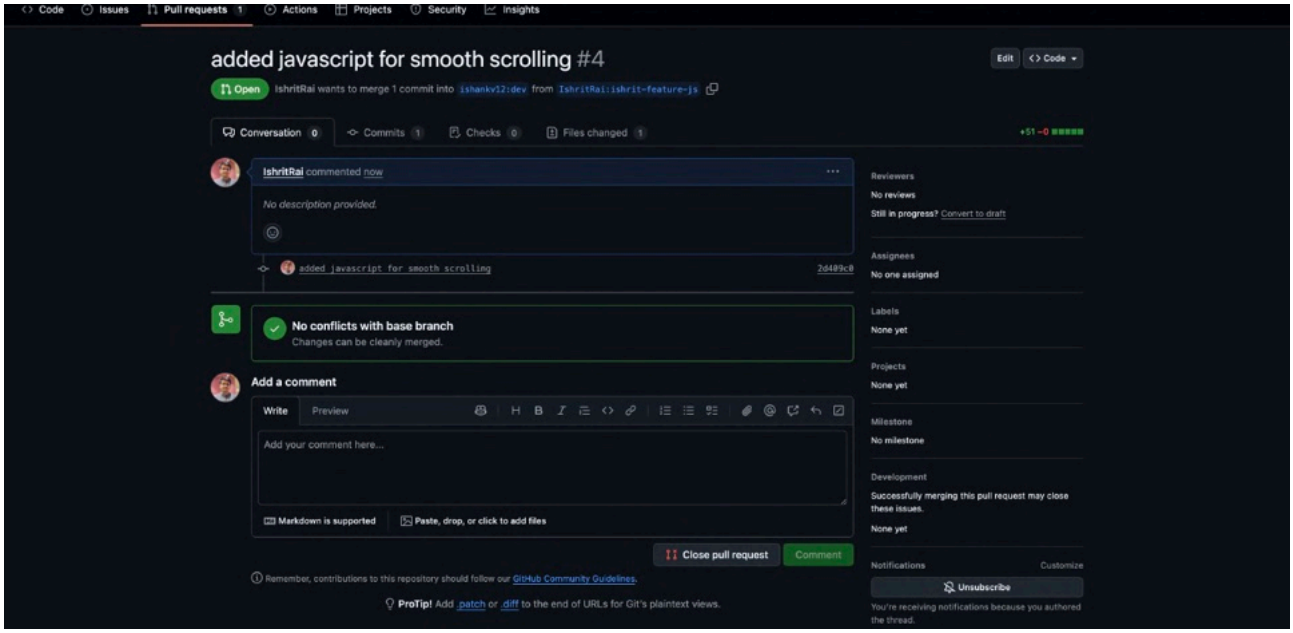
Step 5 : push the changes to forked repo

```
ishritrai@192 ishank-repo % git push origin ishrit-feature-js
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 988 bytes | 988.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'ishrit-feature-js' on GitHub by visiting:
remote:      https://github.com/IshritRai/ishank-repo/pull/new/ishrit-feature-js
remote:
To github.com:IshritRai/ishank-repo.git
 * [new branch]      ishrit-feature-js -> ishrit-feature-js
```



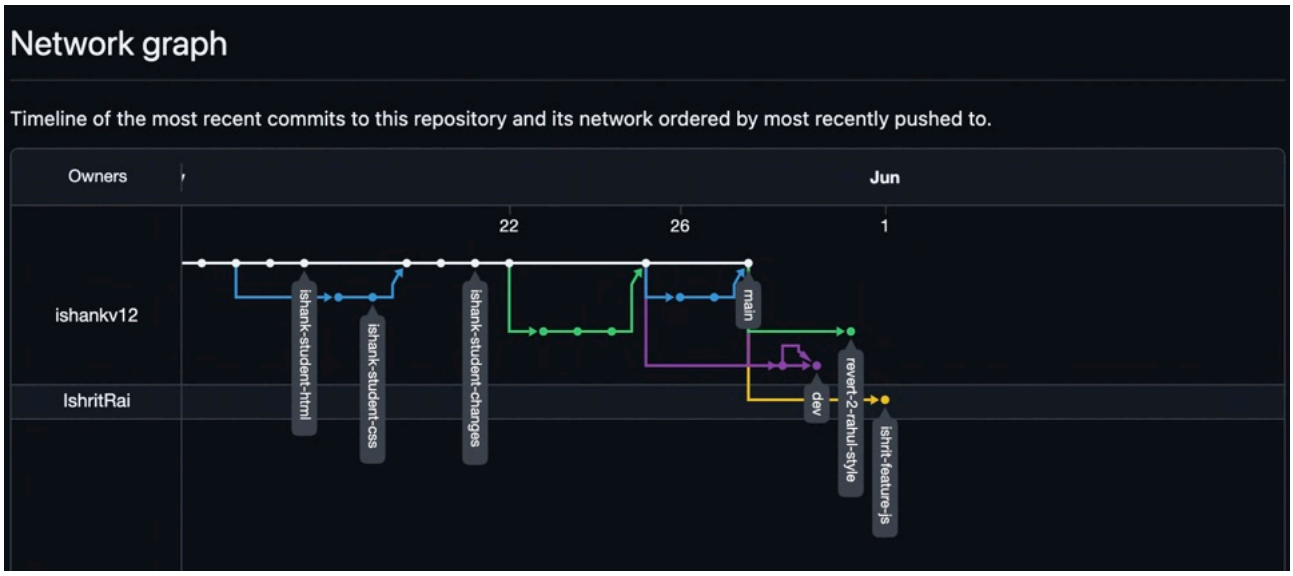
step 5: navigate to GitHub and send pull request to dev/test branch





Pull request sent

Step 6: navigate to insights and then to network graph to see the overall branching workflow



10. Pull Requests

Pull requests (PRs) are a core feature of collaborative development platforms like GitHub and GitLab. They allow contributors to propose changes, discuss them, and request that maintainers review and merge their work into the main project. PRs are essential for code review, quality control, and team collaboration.

Let's demonstrate a realistic pull request workflow using the `calculator.c` project. Suppose you want to contribute a new feature (factorial operation) to a public repository. You'll fork the repository, create a feature branch, make your changes, push them, and open a pull request.

```
// calculator.c (feature/factorial branch adds factorial)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

unsigned long long factorial(int n) {
    if (n < 0) return 0;
    if (n == 0) return 1;
    return n * factorial(n - 1);
}

typedef struct {
    double result;
    char operation[20];
    int error;
} Calculation;

Calculation perform_operation(double a, double b, const char* op) {
    Calculation calc = {0};
    strcpy(calc.operation, op);
    if (strcmp(op, "add") == 0) {
        calc.result = a + b;
    } else if (strcmp(op, "subtract") == 0) {
        calc.result = a - b;
    } else if (strcmp(op, "multiply") == 0) {
        calc.result = a * b;
    } else if (strcmp(op, "divide") == 0) {
        if (b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = a / b;
    } else if (strcmp(op, "modulus") == 0) {
        if ((int)b == 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = (int)a % (int)b;
```

```
    } else if (strcmp(op, "power") == 0) {
        calc.result = pow(a, b);
    } else if (strcmp(op, "sqrt") == 0) {
        if (a < 0) {
            calc.error = 1;
            return calc;
        }
        calc.result = sqrt(a);
    } else if (strcmp(op, "factorial") == 0) {
        if (a < 0 || (int)a != a) {
            calc.error = 1;
            return calc;
        }
        calc.result = (double)factorial((int)a);
    } else {
        calc.error = 1;
        return calc;
    }
    return calc;
}

int main() {
    double num1 = 0, num2 = 0;
    char operation[20];
    printf("Enter first number: ");
    scanf("%lf", &num1);
    printf("Enter second number (or 0 for unary ops): ");
    scanf("%lf", &num2);
    printf("Enter operation (add/subtract/multiply/divide/modulus/power/sqrt/factori
    scanf("%19s", operation);
    Calculation result = perform_operation(num1, num2, operation);
    if (result.error) {
        printf("Error: Invalid operation or input\n");
        return 1;
    }
    printf("Result: %.2f\n", result.result);
    return 0;
}
```

Here is a step-by-step terminal simulation of a pull request workflow:

```
ishritrai@192:~$ git clone https://github.com/original/calculator.git
Cloning into 'calculator'...
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 20 (delta 4), reused 20 (delta 4), pack-reused 0
Unpacking objects: 100% (20/20), done.
ishritrai@192:~$ cd calculator
ishritrai@192:~/calculator$ git remote rename origin upstream
ishritrai@192:~/calculator$ git remote add origin
https://github.com/ishritrai/calculator.git
ishritrai@192:~/calculator$ git checkout -b feature/factorial
```

```
Switched to a new branch 'feature/factorial'
ishritrai@192:~/calculator$ # Edit calculator.c to add factorial support
ishritrai@192:~/calculator$ git add calculator.c
ishritrai@192:~/calculator$ git commit -m "Add factorial operation"
[feature/factorial 9a0b1c2] Add factorial operation
1 file changed, 15 insertions(+)
ishritrai@192:~/calculator$ git push origin feature/factorial
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (5/5), 1.2 KiB | 1.20 MiB/s, done.
Total 5 (delta 2), reused 0 (delta 0)
To https://github.com/ishritrai/calculator.git
* [new branch] feature/factorial -> feature/factorial
ishritrai@192:~/calculator$ # Open GitHub and create a pull request from
feature/factorial
```

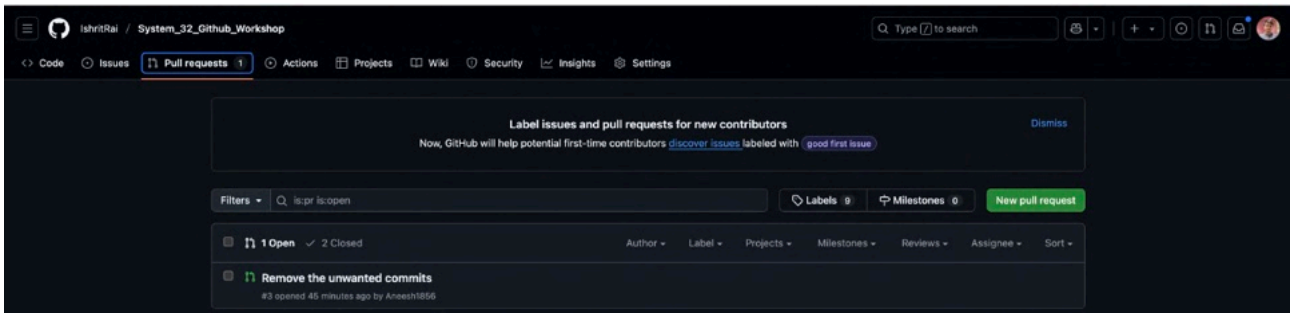
The above demonstration shows:

- Forking and cloning a repository to contribute
- Creating a feature branch for your work
- Making and committing a significant change to the C project
- Pushing your branch to your fork
- Opening a pull request for review and merging
- How pull requests support code review and collaborative development

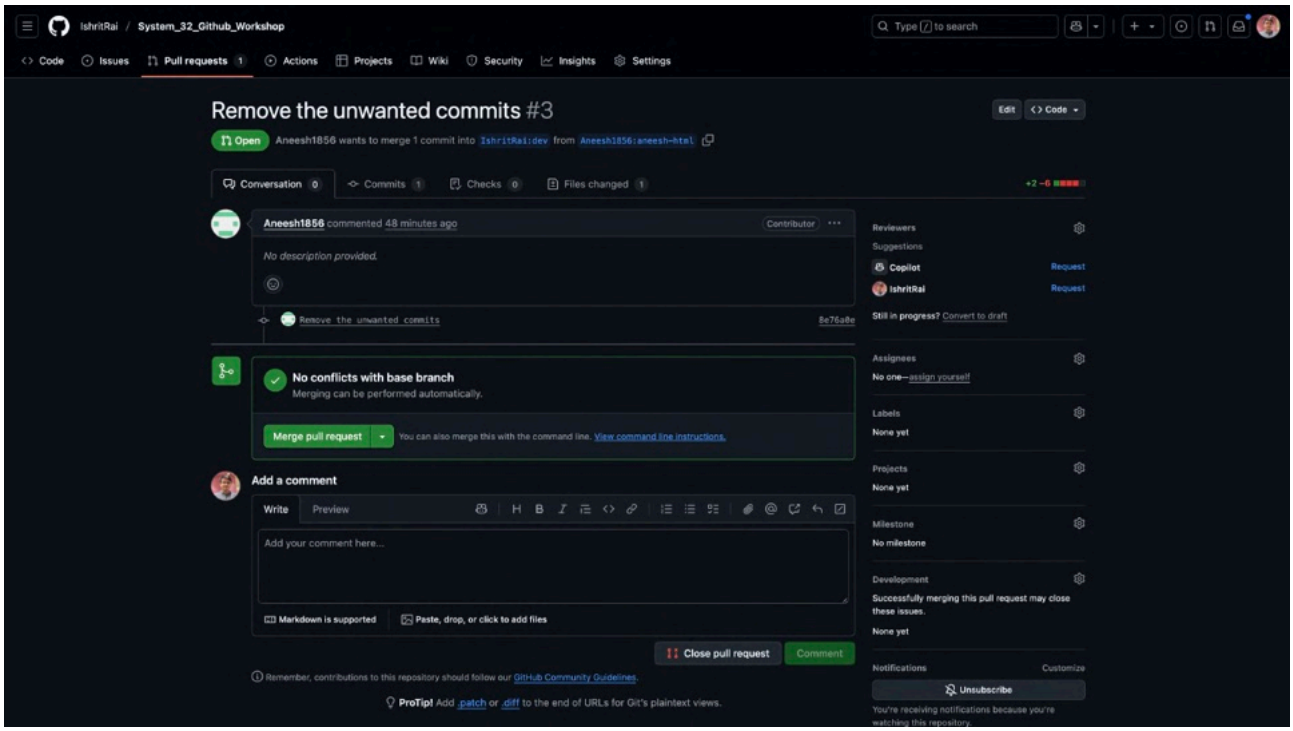
Pull requests are essential for maintaining code quality and enabling effective teamwork in open source and private projects alike.

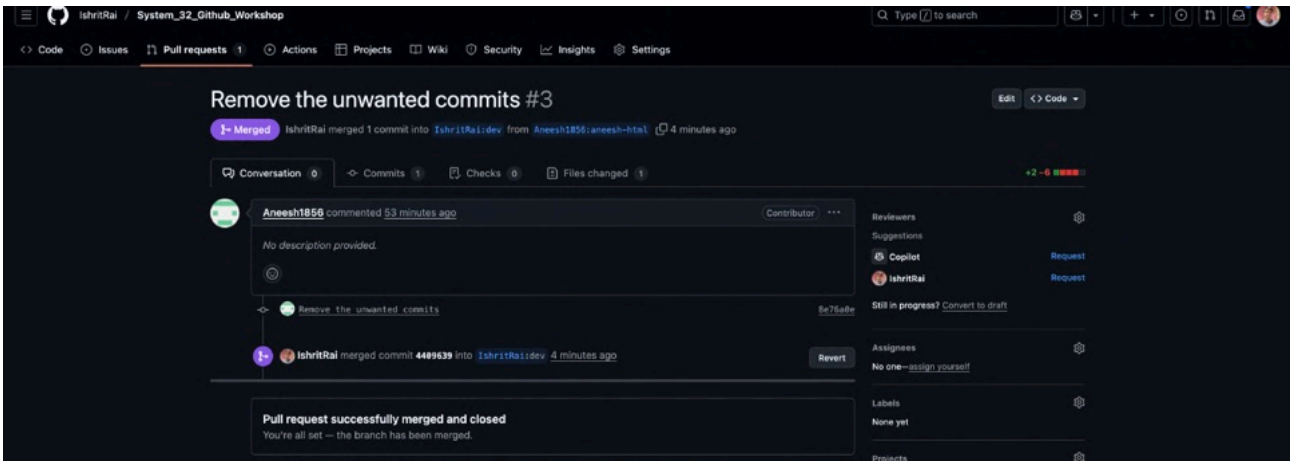
Lab Exercise 9: Accepting pull Requests

Step 1: . Navigate to pull request section of your GitHub repo



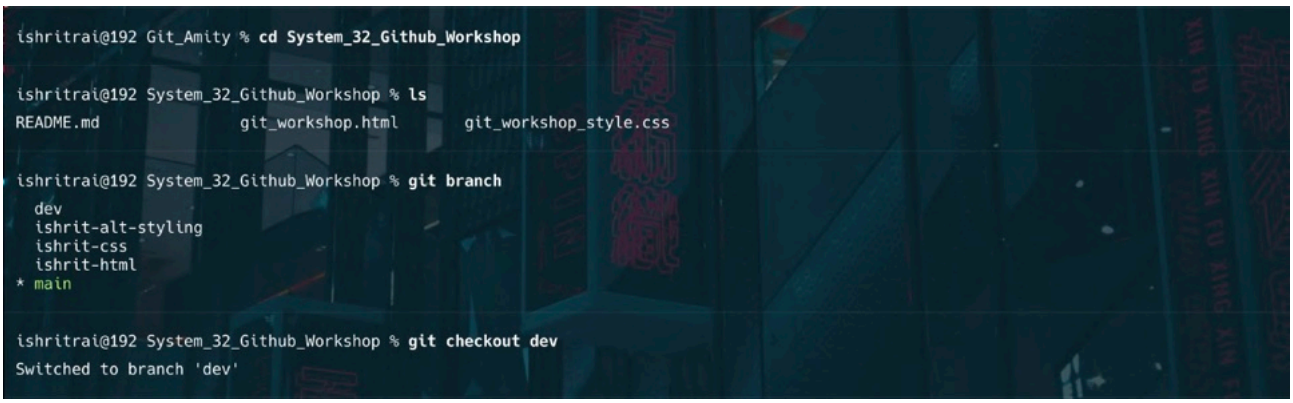
Step 2: Accept the pull request





Pull request merged

Step 3: In your local git repository checkout to dev branch



Step 4: execute git pull to integrate the commit on GitHub to local system repo

Step 5: checkout main branch merge the the dev branch

```
ishritrai@192 System_32_Github_Workshop % git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.

ishritrai@192 System_32_Github_Workshop % git pull origin main
From github.com:IshritRai/System_32_Github_Workshop
 * branch            main       -> FETCH_HEAD
Already up to date.

ishritrai@192 System_32_Github_Workshop % git merge dev
Updating 6216ef9..4409639
Fast-forward
 git_workshop.html | 8 ++------
 1 file changed, 2 insertions(+), 6 deletions(-)
```

Step 6: push the main branch to GitHub

```
ishritrai@192 System_32_Github_Workshop % git push origin main
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:IshritRai/System_32_Github_Workshop.git
   6216ef9..4409639  main -> main
```

Step 7: Confirm with network graph on Github

12. .gitignore

The `.gitignore` file tells Git which files or directories to ignore in a project. This is crucial for keeping your repository clean from build artifacts, temporary files, and sensitive information. A well-crafted `.gitignore` improves collaboration and prevents accidental commits of unwanted files.

Let's demonstrate `.gitignore` with the `calculator.c` project. We'll ignore compiled binaries, object files, and a sensitive file containing API keys.

```
# .gitignore for calculator project
# Ignore compiled binaries
calculator

# Ignore object files
*.o

# Ignore editor/OS files
*.swp
.DS_Store

# Ignore sensitive files
secrets.env
```

Here is a step-by-step terminal simulation showing the effect of `.gitignore`:

```
ishritrai@192:~/calculator$ echo "API_KEY=supersecret" > secrets.env
ishritrai@192:~/calculator$ gcc calculator.c -o calculator
ishritrai@192:~/calculator$ ls
calculator calculator.c secrets.env
ishritrai@192:~/calculator$ echo -e
'calculator\n*.o\n*.swp\n.DS_Store\nsecrets.env' > .gitignore
ishritrai@192:~/calculator$ git status
On branch main
Untracked files:
(use "git add <file>..." to include in what will be committed)
calculator.c
.gitignore
ishritrai@192:~/calculator$ git add .gitignore calculator.c
ishritrai@192:~/calculator$ git status
On branch main
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
new file: .gitignore
new file: calculator.c
Untracked files:
calculator
secrets.env
ishritrai@192:~/calculator$ git add calculator secrets.env
The following paths are ignored by one of your .gitignore files:
calculator
```

```
secrets.env
Use -f if you really want to add them.
```

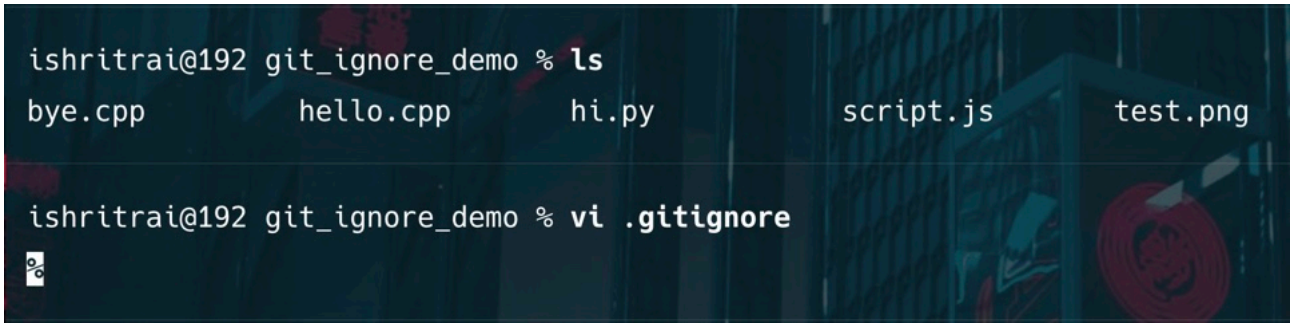
The above demonstration shows:

- How to create a `.gitignore` file for a C project
- How ignored files (binaries, object files, sensitive files) are excluded from version control
- How `git status` and `git add` behave with ignored files
- Best practices for keeping your repository clean and secure

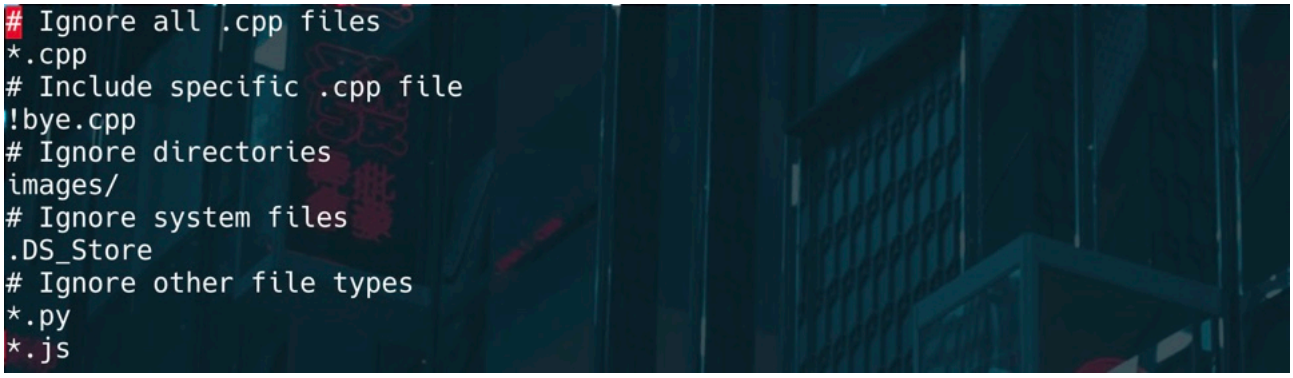
Using `.gitignore` is essential for all professional software projects to avoid clutter and protect sensitive data.

Lab Exercise 10: .gitignore

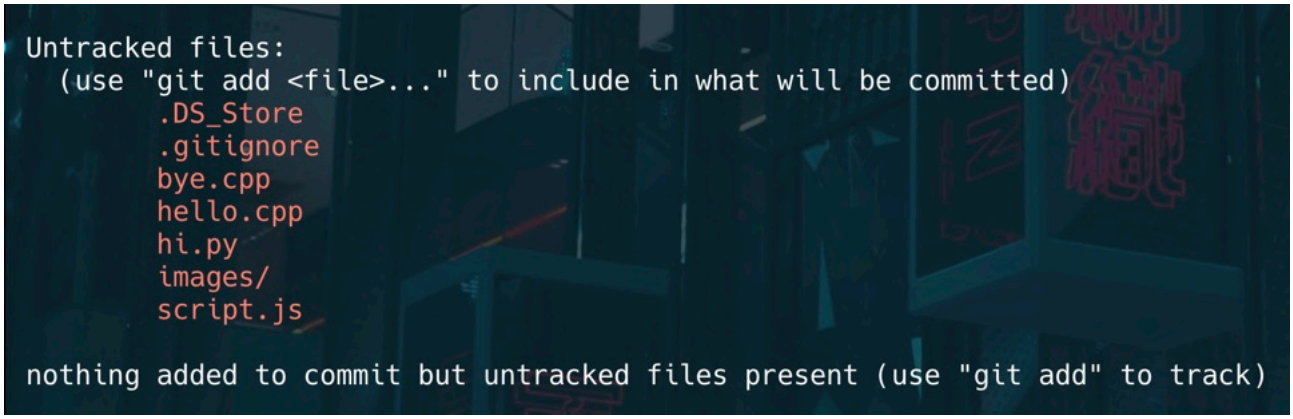
Step 1: added multiple files with different extensions



Step 2: initialised git repository and included.gitignore file



Step 3: staging area before committing gitingore file



Step 4: committing gitignore file

```
ishritrai@192 git_ignore_demo % git commit -m "added .gitignore file"
[master (root-commit) e883aef] added .gitignore file
 1 file changed, 3 insertions(+)
 create mode 100644 .gitignore
```

Step 5: staging area after committing gitignore file

```
ishritrai@192 git_ignore_demo % git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   bye.cpp
```

Source Code Management Project Report

Git and GitHub Workshop Implementation

Student: Ishrit Rai **Course:** Source Code Management **Date:** June 2025

Table of Contents

1. Project Overview

This report documents the implementation and learning outcomes from the Source Code Management course, focusing on Git and GitHub practices. The project involved creating and managing repositories, collaborating with team members, and implementing version control best practices. Through this project, we gained practical experience in distributed version control systems, collaborative development workflows, and modern software development practices.

Project Timeline

Week 1: Repository setup and initial documentation

Week 2: Core feature implementation

Week 3: Team collaboration and pull requests

Week 4: Final integration and documentation

2. Repository Structure

2.1 My Repository: System_32_Ishrit_git_workshop

My repository contains a web development project with a well-organized structure that follows modern development practices.

Frontend Architecture

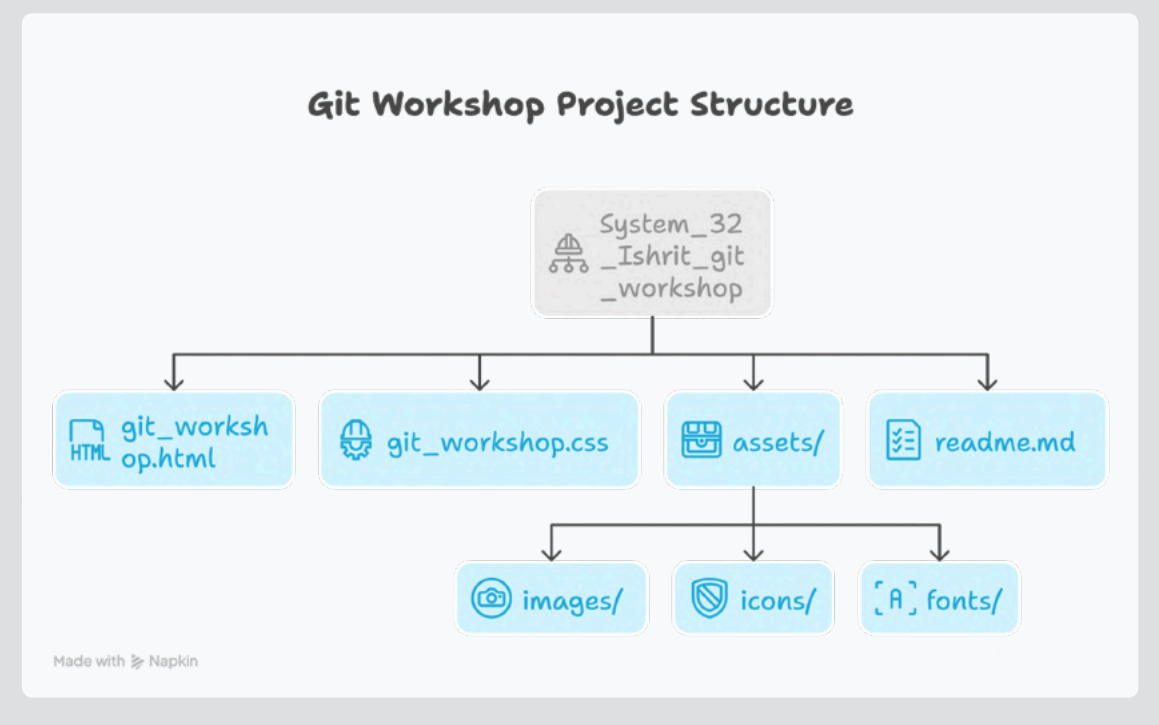


Figure 2.1: Project Structure Diagram showing the hierarchical organization of the System_32_Ishrit_git_workshop repository

HTML Structure Implementation

```
<!-- Navigation -->
<nav class="main-nav">
  <div class="logo">
    <img src="assets/images/logo.svg" alt="Event Logo">
  </div>
  <ul class="nav-links">
    <li><a href="#home">Home</a></li>
    <li><a href="#schedule">Schedule</a></li>
    <li><a href="#speakers">Speakers</a></li>
    <li><a href="#register">Register</a></li>
  </ul>
</nav>

<!-- Hero Section -->
<section class="hero">
  <h1>Git & GitHub Workshop</h1>
  <p>Learn version control and collaboration</p>
  <button class="cta-button">Register Now</button>
</section>
```

CSS Architecture

```
/* CSS Variables for Theming */
:root {
  --primary-color: #2c3e50;
  --secondary-color: #3498db;
  --accent-color: #e74c3c;
  --text-color: #333;
  --background-color: #fff;
}

/* Responsive Grid System */
.container {
  display: grid;
  grid-template-columns: repeat(12, 1fr);
  gap: 20px;
  max-width: 1200px;
  margin: 0 auto;
  padding: 0 20px;
}

/* Animation Keyframes */
@keyframes fadeIn {
  from { opacity: 0; }
  to { opacity: 1; }
}
```

Design Decisions and Architecture

Component	Approach	Benefits
Component-Based Architecture	Modular HTML structure with reusable CSS components	Maintainable codebase, separation of concerns
Responsive Design	Mobile-first approach with fluid typography	Optimal user experience across devices
Performance Optimization	Lazy loading, CSS minification, asset optimization	Faster load times and better user experience

3. Git Commands and Workflow

3.1 Repository Setup and Initialization

```
# Initialize new repository
git init

# Configure user identity
git config --global user.name "Ishrit Rai"
git config --global user.email "your.email@example.com"

# Configure default branch name
git config --global init.defaultBranch main

# Configure line ending behavior
git config --global core.autocrlf input
```

Initial Setup Commands

```
# Stage all files
git add .

# Create first commit
git commit -m "Initial commit: Project setup and basic structure"

# Rename default branch to main
git branch -M main

# Add remote repository
git remote add origin
https://github.com/username/System_32_Ishrit_git_workshop.git

# Push to remote repository
git push -u origin main

# Verify remote configuration
git remote -v
```

3.2 Version Control Operations

Feature Development Workflow

```
# Create feature branch with conventional naming
git checkout -b feature/hero-section

# Development work with atomic commits
git add git_workshop.html
git commit -m "feat(hero): Add responsive hero section

- Implement responsive navigation
- Add hero content with animations
- Include CTA button with hover effects
- Optimize for mobile devices"

# Push feature branch
git push origin feature/hero-section
```

> **Best Practice:** Always use descriptive commit messages following conventional commit standards. This helps in maintaining a clear project history and enables automated changelog generation.

Styling and UI Improvements

```
# Create styling branch
git checkout -b feature/styling

# Implement CSS architecture
git add git_workshop.css
git commit -m "style: Implement CSS architecture

- Add CSS variables for theming
- Create responsive grid system
- Implement mobile-first media queries
- Add animation keyframes"

# Add component styles
git commit -m "style: Add component styles

- Style navigation component
- Implement hero section design
- Add button and form styles
- Create card components"
```

Collaboration and Merging

```
# Fetch latest changes
git fetch origin

# Update local main branch
git checkout main
git pull origin main

# Merge feature branch
git merge feature/styling

# Resolve conflicts if any
git status
# Edit conflicting files
git add .
git commit –m "merge: Resolved conflicts in styling"

# Push changes
git push origin main
```

4. Collaboration and Contributions

4.1 Pull Request Process

Creating Comprehensive Pull Requests

```
# Create feature branch
git checkout -b feature/new-feature

# Implement changes with detailed commits
git add .
git commit -m "feat: Implement new feature

## Changes
- Added new component
- Updated styling
- Fixed bugs

## Technical Details
- Used CSS Grid for layout
- Implemented responsive design
- Added accessibility features"

# Push branch
git push origin feature/new-feature
```

Pull Request Template

A comprehensive PR should include:

- **Overview:** Brief description of changes
- **Technical Details:** Implementation specifics
- **Testing:** Test cases and verification steps
- **Screenshots:** Visual changes documentation
- **Related Issues:** Links to relevant issues

4.2 Team Contributions

Contributing to Team Repositories

```
# Fork repository
gh repo fork Aditya12705/Aditya-SCM-Project

# Clone fork
git clone https://github.com/your-username/Aditya-SCM-Project.git

# Create feature branch
git checkout -b feature/blockchain

# Implement changes
git add Blockchain.html
git commit -m "feat: Implement blockchain functionality"

# Push changes
git push origin feature/blockchain
```

Repository	Contribution Type	Description
Aditya-SCM-Project	Feature Implementation	Added blockchain functionality with responsive design
Team Repository 2	UI Improvements	Enhanced user interface with modern design patterns
Team Repository 3	Bug Fixes	Resolved critical issues and performance problems

5. Key Learnings

5.1 Technical Skills

Advanced Git Operations

```
# Advanced branch management
git branch -m old-name new-name          # Rename branch
git branch -u origin/main feature/branch # Set upstream
git branch --merged                      # List merged branches
git branch --no-merged                   # List unmerged branches

# Advanced commit management
git commit --amend                       # Modify last commit
git rebase -i HEAD~3                     # Interactive rebase
git cherry-pick commit-hash              # Apply specific commit
git revert commit-hash                   # Revert specific commit

# Advanced remote operations
git remote set-url origin new-url        # Change remote URL
git remote show origin                   # Show remote details
git fetch --prune                        # Remove stale branches
git push --force-with-lease              # Safe force push
```

GitHub Collaboration Commands

```
# Fork repository
gh repo fork owner/repo

# Create PR
gh pr create

# Review PR
gh pr review

# Merge PR
gh pr merge
```

5.2 Best Practices

Commit Management Standards

```
# Conventional commits
git commit -m "feat: Add new feature"
git commit -m "fix: Resolve bug"
git commit -m "docs: Update documentation"
git commit -m "style: Format code"
git commit -m "refactor: Restructure code"
```

Branching Strategy

- **Feature branches:** feature/feature-name
- **Bug fix branches:** fix/issue-description
- **Release branches:** release/version-number
- **Hotfix branches:** hotfix/critical-fix

Documentation Best Practices

```
# Update README
git add README.md
git commit -m "docs: Update project documentation

## Added
- Installation instructions
- Usage examples
- Contributing guidelines

## Updated
- API documentation
- Troubleshooting section"
```

6. Challenges and Solutions

6.1 Common Challenges Encountered

Merge Conflicts Resolution

```
# Identify conflicts
git status

# Resolve conflicts
# Edit conflicting files
git add .
git commit -m "merge: Resolve conflicts"

# Abort merge if needed
git merge --abort
```

> **Lesson Learned:** Regular communication with team members and frequent pulls from the main branch help minimize merge conflicts.

File Management Issues

```
# Create .gitignore
echo "*.log" >> .gitignore
echo "node_modules/" >> .gitignore
echo ".env" >> .gitignore

# Remove tracked files
git rm --cached file
git commit -m "chore: Remove tracked file"
```

6.2 Solutions Implemented

Challenge	Solution	Prevention Strategy
Merge Conflicts	Manual resolution and team communication	Regular pulls and smaller, focused commits
Accidental File Commits	Proper .gitignore configuration	Template-based .gitignore setup
Branch Management	Standardized naming conventions	Team guidelines and documentation

7. Conclusion

This project provided comprehensive hands-on experience with Git and GitHub, demonstrating the importance of version control in modern software development. The implementation of various features and collaboration with team members helped in understanding real-world development scenarios.

Key Achievements

- **Distributed Version Control:** Mastered Git fundamentals and advanced operations
- **Collaborative Development:** Successfully worked with team members on shared repositories
- **Code Review Processes:** Implemented thorough PR review workflows
- **Project Management:** Organized development workflow with proper branching strategies
- **Documentation Practices:** Maintained comprehensive project documentation

The project successfully demonstrated the practical application of source code management principles in a collaborative environment, providing valuable experience for future software development projects.

8. Future Improvements

8.1 Enhanced Documentation

```
# Generate API documentation
npm run docs:generate

# Update architecture diagrams
git add docs/architecture/
git commit -m "docs: Update architecture documentation

## Added
- System architecture diagram
- Component hierarchy
- Data flow diagrams

## Updated
- API documentation
- Deployment guide
- Development setup"
```

8.2 Automation and CI/CD

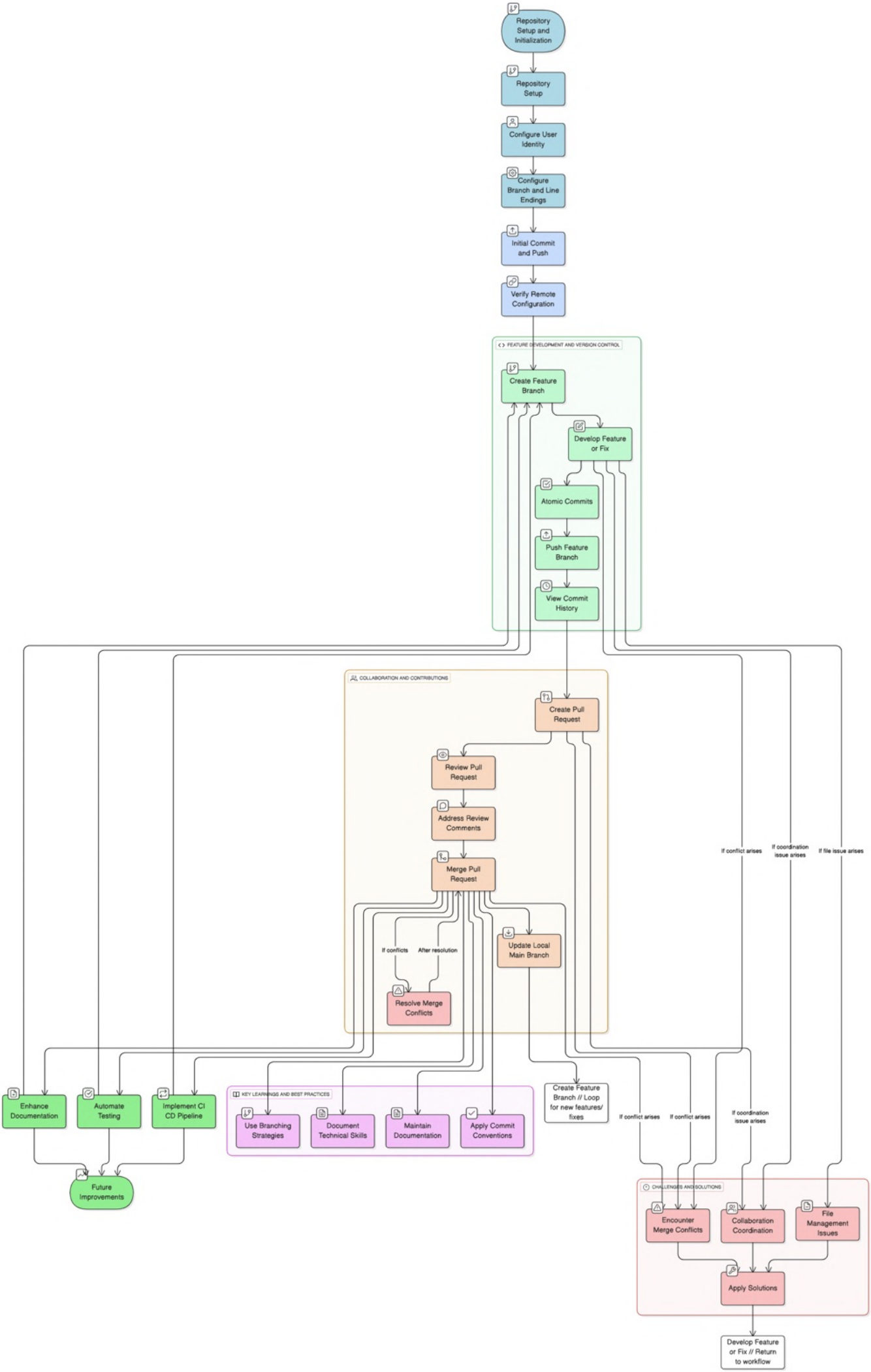
```
# Add GitHub Actions workflow
git add .github/workflows/
git commit -m "ci: Add automated testing workflow

## Added
- Automated testing on PR
- Code quality checks
- Deployment pipeline

## Features
- Multi-browser testing
- Performance monitoring
- Security scanning"
```

8.3 Advanced Git Workflows

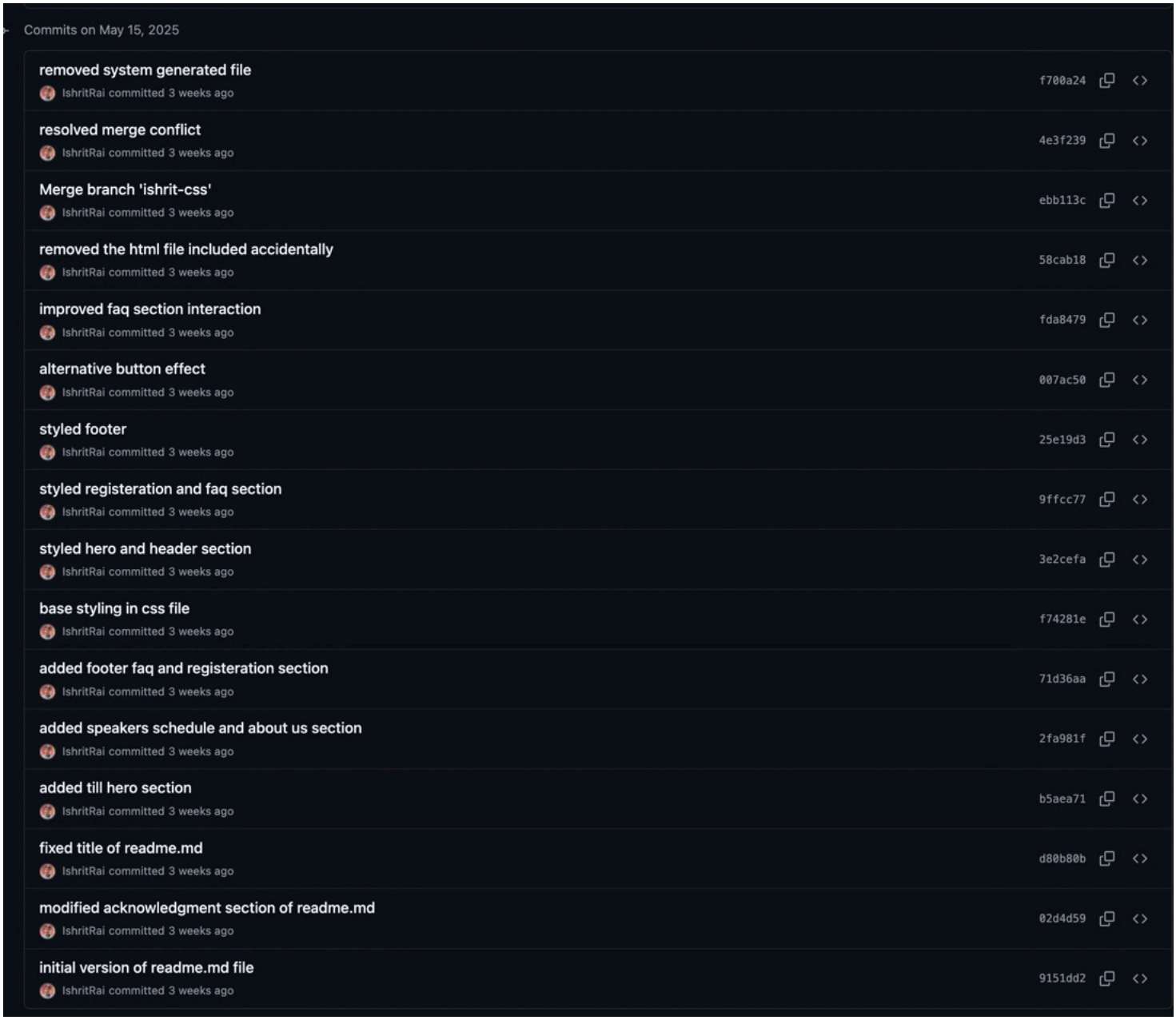
Planned Implementations

- **Git Hooks:** Pre-commit hooks for code quality
- **Semantic Versioning:** Automated version management
- **Release Management:** Structured release processes
- **Advanced Branching:** GitFlow implementation

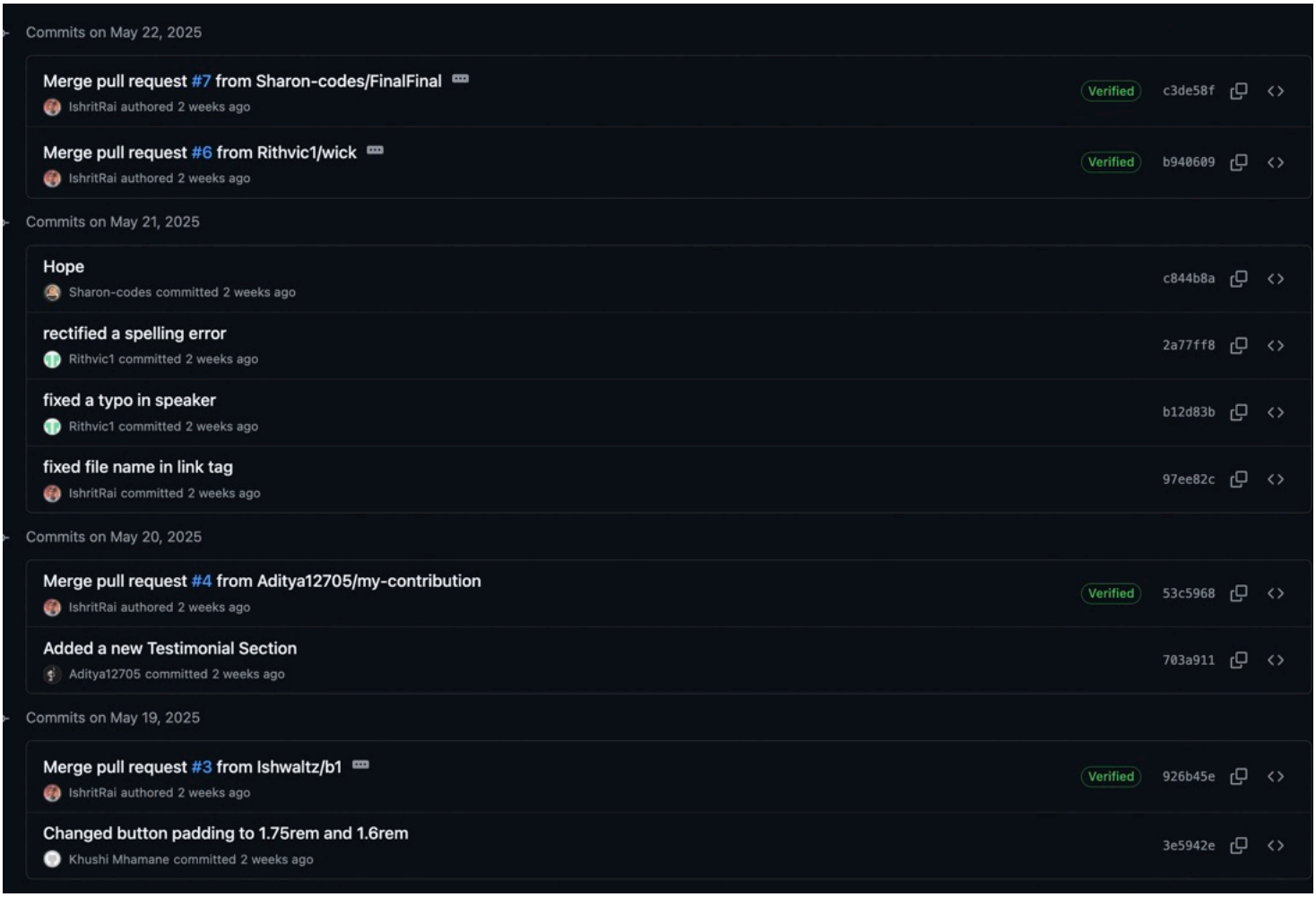
These improvements will further enhance the development workflow and prepare the team for larger, more complex projects requiring sophisticated version control strategies.


Repository Setup and Initialization
Repository Setup
Configure User Identity
Configure Branch and Line Endings
Initial Commit and Push
Verify Remote Configuration
FEATURE DEVELOPMENT AND VERSION CONTROL
Create Feature Branch
Develop Feature or Fix
Atomic Commits
Push Feature Branch
View Commit History
COLLABORATION AND CONTRIBUTIONS
Create Pull Request
Review Pull Request
Address Review Comments
Merge Pull Request
If conflicts
After resolution
Update Local Main Branch
Resolve Merge Conflicts
If conflict arises
If coordination issue arises
If file issue arises
Enhance Documentation
Automate Testing
Implement CI CD Pipeline
KEY LEARNINGS AND BEST PRACTICES
Use Branching Strategies
Document Technical Skills
Maintain Documentation
Apply Commit Conventions
Create Feature Branch // Loop for new features/ fixes
If conflict arises
If conflict arises
If coordination issue arises
Future Improvements
CHALLENGES AND SOLUTIONS
Encounter Merge Conflicts
Collaboration Coordination
File Management Issues
Apply Solutions
Develop Feature or Fix // Return to workflow

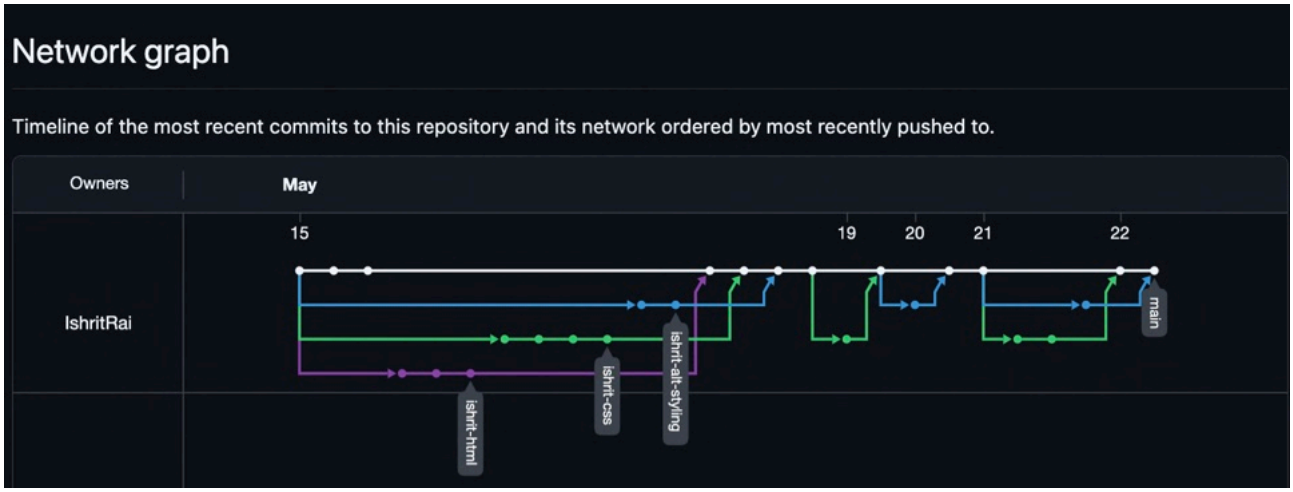
Personal Repo Commits



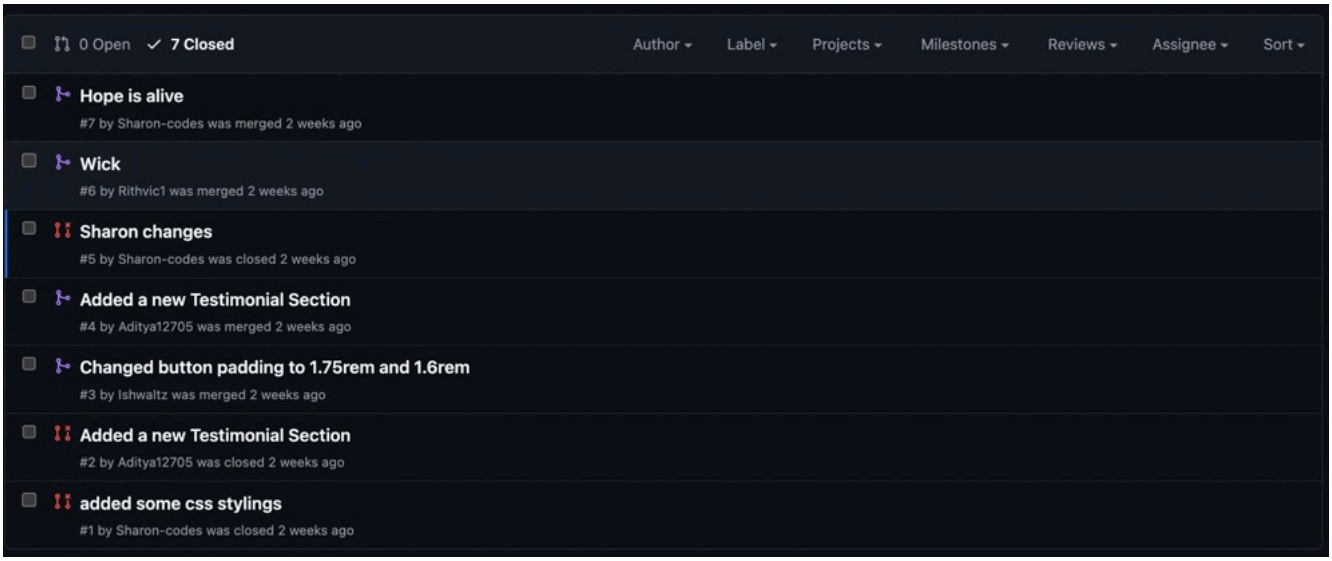
Merge Requests



Network Graph of my repo



Pull Requests received



Pull Requests Sent