Source Code Management

Course Code: CSE 2015

Slot: L15+L16 & L35+L36

Name: S R Jeevika
SEN No. :
A86617512410 4 Faculty:
Dr Monit Kapoor

Road, NH-207, opposite to the office of Deputy Commissioner, Devanahalli, Doddaballapura, Bengaluru, Karnataka 562110

Table of Contents

Lab Session 1: Git Fundamentals

Computer

A computer is any electronic device capable of carrying out logical and mathematical calculations.

Program / Code

A program, or code, is a sequence of instructions—often structured as an algorithm—that tells a computer how to complete a particular task.

Why Manage Source Code?

Modern software applications, like Spotify, are made up of many interconnected programs running on both the frontend and backend. These components require regular updates to:

- **Fix Bugs** – Correct errors and issues quickly.
- **Enhance UI/UX** – Improve the design, usability, and overall experience.
- **Boost Performance** – Optimise code for efficiency and speed.

  For developers, proper source code management is essential because it:

- Keeps files organised and consistent throughout the development cycle.
- Enables smooth collaboration between multiple programmers working on the same project.

  Tools for Source Code Management

- **Git** – A local version control system that records changes to your project and helps manage different versions over time.
- **GitHub** – A cloud-based platform that hosts Git repositories, making it easy to share, collaborate, and contribute from anywhere in the world.

  Version

  In version control, a *version* is a saved state (or snapshot) of your project at a specific point in time. Versions allow you to view history, roll back to earlier states, or compare changes made during development.

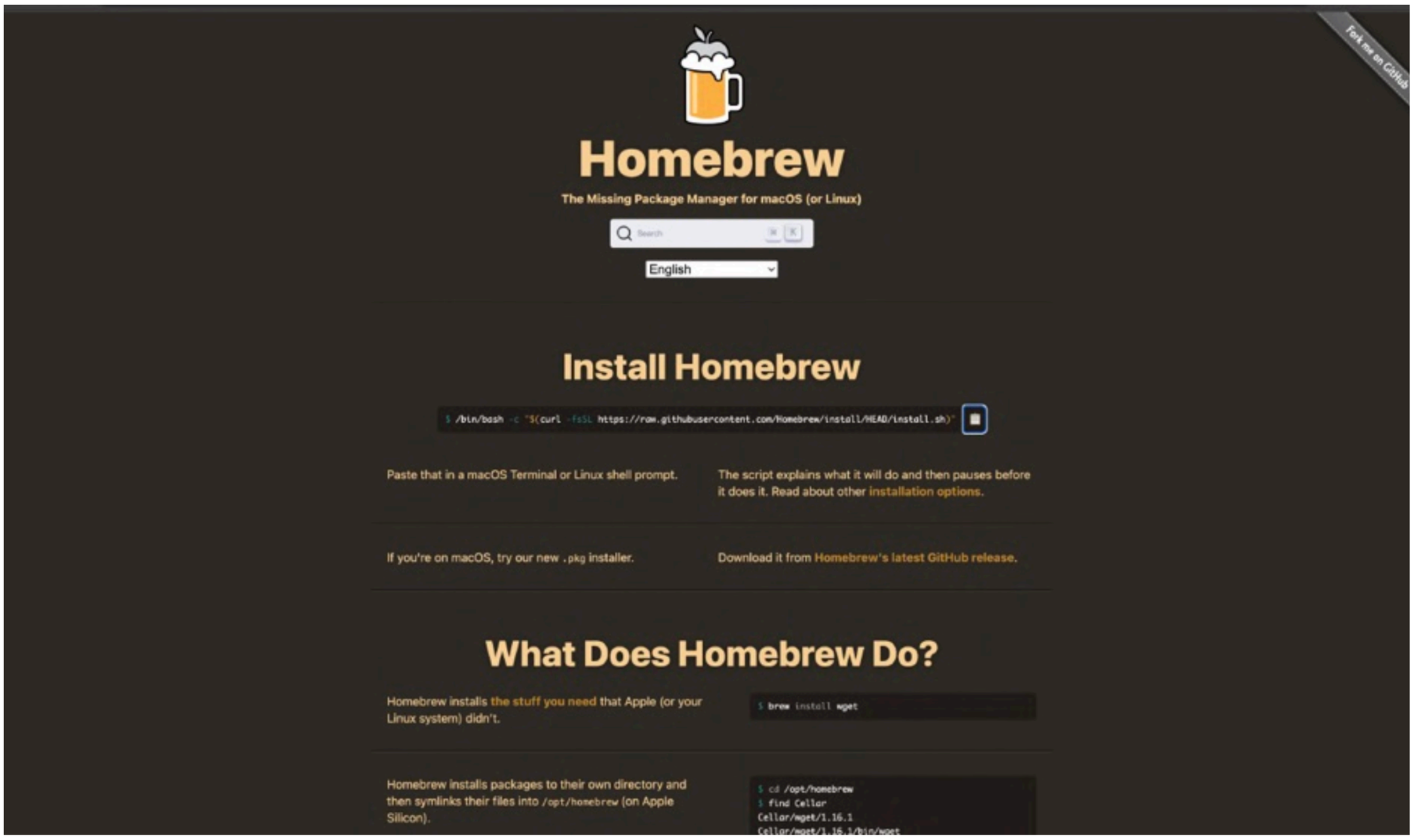
Lab Practical 1

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.



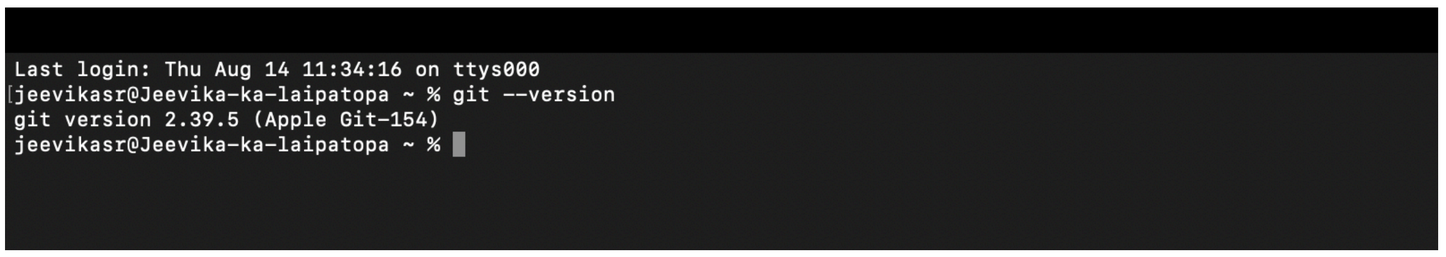
Step 3: Install Git Using Homebrew:

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

```
==> Downloading https://ghcr.io/v2/homebrew/core/git/manifests/2.48.1
######################################################################################################################################################################################## 100.0%
==> Fetching dependencies for git: libunistring, gettext and pcre2
==> Downloading https://ghcr.io/v2/homebrew/core/libunistring/manifests/1.3
######################################################################################################################################################################################## 100.0%
==> Fetching libunistring
==> Downloading https://ghcr.io/v2/homebrew/core/libunistring/blobs/sha256:e919f6ee2fe8a40addea1e1840eab8855e66812e18dfe05c130618ce517e2880
######################################################################################################################################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/gettext/manifests/0.23.1
######################################################################################################################################################################################## 100.0%
==> Fetching gettext
==> Downloading https://ghcr.io/v2/homebrew/core/gettext/blobs/sha256:89b1b668765938504ce64f991cf6b81fc96765f5dec84bbf64c007c3d811b8fe
######################################################################################################################################################################################## 100.0%
==> Downloading https://ghcr.io/v2/homebrew/core/pcre2/manifests/10.44
######################################################################################################################################################################################## 100.0%
==> Fetching pcre2
==> Downloading https://ghcr.io/v2/homebrew/core/pcre2/blobs/sha256:5bf0758d81478e59007717b43d854a1e2399818c2859fcbc5cdb2e616e4eb372
######################################################################################################################################################################################## 100.0%
==> Fetching git
==> Downloading https://ghcr.io/v2/homebrew/core/git/blobs/sha256:714f0ac2810a192f985ac3c56a761da485f584b46bc224b84e4d320fefbaae89
######################################################################################################################################################################################## 100.0%
==> Installing dependencies for git: libunistring, gettext and pcre2
==> Installing git dependency: libunistring
==> Downloading https://ghcr.io/v2/homebrew/core/libunistring/manifests/1.3
Already downloaded: /Users/ishritrai/Library/Caches/Homebrew/downloads/a570da63bc1839c7e217f203abd54d4d873ebd6b99f6e88994d0e79e2ebe987c--libunistring-1.3.bottle_manifest.json
==> Pouring libunistring--1.3.sonoma.bottle.tar.gz
🍺  /usr/local/Cellar/libunistring/1.3: 59 files, 5.5MB
==> Installing git dependency: gettext
==> Downloading https://ghcr.io/v2/homebrew/core/gettext/manifests/0.23.1
Already downloaded: /Users/ishritrai/Library/Caches/Homebrew/downloads/a5b6ba6453cc31731ac67fb3d0075d98ab572a913f3e740b54a86d79906f360e--gettext-0.23.1.bottle_manifest.json
==> Pouring gettext--0.23.1.sonoma.bottle.tar.gz
🍺  /usr/local/Cellar/gettext/0.23.1: 2,052 files, 21MB
==> Installing git dependency: pcre2
==> Downloading https://ghcr.io/v2/homebrew/core/pcre2/manifests/10.44
Already downloaded: /Users/ishritrai/Library/Caches/Homebrew/downloads/22ed791461c5bf400adde8c3b432c1230866aa1db3c5cb81e06a6ff21cac96ee--pcre2-10.44.bottle_manifest.json
==> Pouring pcre2--10.44.sequoia.bottle.tar.gz
🍺  /usr/local/Cellar/pcre2/10.44: 237 files, 6.4MB
==> Installing git
==> Pouring git--2.48.1.sonoma.bottle.tar.gz
==> Caveats
The Tcl/Tk GUIs (e.g. gitk, git-gui) are now in the `git-gui` formula.
Subversion interoperability (git-svn) is now in the `git-svn` formula.

zsh completions and functions have been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
🍺  /usr/local/Cellar/git/2.48.1: 1,699 files, 54.8MB
==> Running `brew cleanup git`...
Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).
==> Caveats
==> git
The Tcl/Tk GUIs (e.g. gitk, git-gui) are now in the `git-gui` formula.
Subversion interoperability (git-svn) is now in the `git-svn` formula.
```

Step 4: Verify Git Installation:



2. Basic CLI Commands

1) Command: ls

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

```
[jeevikasr@Jeevika-ka-laipatopa ~ % ls
Applications    Documents       Library         Music           Public          Travel project  git2            try
Desktop         Downloads       Movies          Pictures        TRY.py          cloned          repo123
jeevikasr@Jeevika-ka-laipatopa ~ %
```

2) Command: date

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

3) Command: clear

```
[jeevikasr@Jeevika-ka-laipatopa ~ % date
Thu Aug 14 11:55:12 IST 2025
jeevikasr@Jeevika-ka-laipatopa ~ %
```

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

```
[jeevikasr@Jeevika-ka-laipatopa ~ % date
Thu Aug 14 11:55:12 IST 2025
[jeevikasr@Jeevika-ka-laipatopa ~ % time
shell  0.01s user 0.02s system 0% cpu 9:51.67 total
children  0.02s user 0.05s system 0% cpu 9:51.67 tota
jeevikasr@Jeevika-ka-laipatopa ~ % clear
```

```
jeevikasr@Jeevika-ka-laipatopa ~ %
```

4) Command: time

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

```
jeevikasr@Jeevika-ka-laipatopa ~ % time
shell  0.01s user 0.02s system 0% cpu 17:22.06 total
children  0.02s user 0.05s system 0% cpu 17:22.06 total
jeevikasr@Jeevika-ka-laipatopa ~ %
```

5) Command: rm main.txt

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

6) Command: cat main.txt

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

```
jeevikasr@Jeevika-ka-laipatopa ~ % rm main.txt
jeevikasr@Jeevika-ka-laipatopa ~ % cat main.txt
cat: main.txt: No such file or directory
```

7)Command: cd repo123

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

```
jeevikasr@Jeevika-ka-laipatopa ~ % cd repo123
jeevikasr@Jeevika-ka-laipatopa repo123 % pwd
/Users/jeevikasr/repo123
jeevikasr@Jeevika-ka-laipatopa repo123 %
```

8)Command: **ls**

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

```
jeevikasr@Jeevika-ka-laipatopa ~ % ls
Applications    Documents       Library         Music           Public          Travel project  git2            try
Desktop         Downloads       Movies          Pictures        TRY.txt         cloned          repo123
jeevikasr@Jeevika-ka-laipatopa ~ %
```

9.Command: pwd

Description: returns the present working directory

```
jeevikasr@Jeevika-ka-laipatopa ~ % pwd
/Users/jeevikasr
jeevikasr@Jeevika-ka-laipatopa ~ %
```

10. Command: mkdir

Description: used to to make new directory/folder

```
jeevikasr@Jeevika-ka-laipatopa ~ % mkdir try
mkdir: try: File exists
jeevikasr@Jeevika-ka-laipatopa ~ %
```

11. Command: rmdir

Description: used to remove a directory

```
jeevikasr@Jeevika-ka-laipatopa ~ % rmdir try
rmdir: try: Directory not empty
jeevikasr@Jeevika-ka-laipatopa ~ %
```

12. Command: cd

Description: used to change current directory

```
jeevikasr@Jeevika-ka-laipatopa ~ % cd repo123
jeevikasr@Jeevika-ka-laipatopa repo123 % pwd
/Users/jeevikasr/repo123
jeevikasr@Jeevika-ka-laipatopa repo123 %
```

13. Command: cd ..

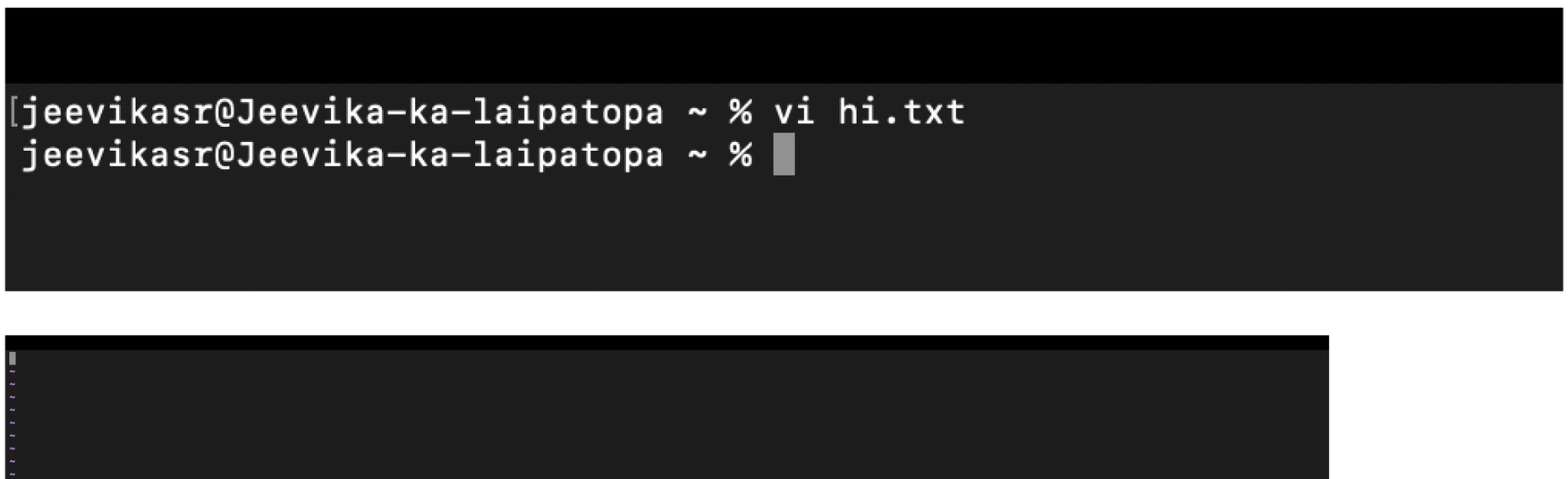
Description: used to exit the current sub directory

```
/Users/jeevikasr/repo123
jeevikasr@Jeevika-ka-laipatopa repo123 % cd --
jeevikasr@Jeevika-ka-laipatopa ~ %
```

3. Vim Text Editor

1) Command: vi main.txt

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



2) Command:**i**(InsertMode)

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.


```
[jeevikasr@Jeevika-ka-laipatopa ~ % vi hi.txt
 jeevikasr@Jeevika-ka-laipatopa ~ %
```

```
[jeevikasr@Jeevika-ka-laipatopa ~ % vi hi.txt
[jeevikasr@Jeevika-ka-laipatopa ~ % cat hi.txt
 first line
 second line
 third line
 jeevikasr@Jeevika-ka-laipatopa ~ %
```

4. Git Configuration

Setting up Git properly is the firststep before you start working with repositories. The git config command is used to define user information and preferences that Git will use during version control operations. These settings usually include your username, email address, and the default text editor, but they can also cover many other behaviors.

Git configurations can be applied at different scopes:

- **System** – affects every user and repository on the machine.
- **Global** – applies only to the current user across all repositories.
- **Local** – applies only to the project (repository) where it is set.

  To visualize this concept, we'll create a simple **C program** that mimics Git configuration by letting the user set details like their name, email, and editor, and then displaying those saved preferences.

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:

```
eevikasr@Jeevika-ka-laipatopa hello_git % git config --global user.name "Jeevika SR"
eevikasr@Jeevika-ka-laipatopa hello_git % git config --global user.email "jeevikasr@example.com"
eevikasr@Jeevika-ka-laipatopa hello_git % git config --list
r.name=Jeevika SR
r.email=jeevikasr@example.com
e.editor=vim
e.autocrlf=input
t.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.

```
jeevikasr@Jeevika-ka-laipatopa ~ % git --version
git version 2.39.5 (Apple Git-154)
jeevikasr@Jeevika-ka-laipatopa ~ %
```

2. Command: git init

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

```
git version 2.39.5 (Apple Git-154)
jeevikasr@Jeevika-ka-laipatopa ~ % git init
Reinitialized existing Git repository in /Users/jeevikasr/.git/
jeevikasr@Jeevika-ka-laipatopa ~ %
```

3. Command: git status

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

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

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   jee/hello.c

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    29july/hello.c
        deleted:    29july/hello.txt
        deleted:    jee/hello.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .CFUserTextEncoding
        .DS_Store
        .Git2.txt.swp
        .Rapp.history
        .bash_history
        .bashrc
        .config/
        .cursor/
        .gitconfig
        .gitignore
        .idlerc/
        .ipython/
        .lesshst
        .local/
        .matplotlib/
        .npm/
        .python_history
        .spyder-py3/
        .ssh/
        .th-client/
        .viminfo
        .vscode/
        .zprofile
        .zprofile.pysave
        .zsh_history
        .zsh_sessions/
        .zshrc
        Applications/
        Desktop/
        Documents/
        Downloads/
        Library/
        Movies/
        Music/
        Pictures/
        Public/
        Travel project/
        cloned/
        git2/
        hi.txt
        repo123/
        try/

no changes added to commit (use "git add" and/or "git commit -a")
jeevikasr@Jeevika-ka-laipatopa ~ %
```

The above files are not currently tracked by git

4. Command git config --global user.name "jeevikaaa97"

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

```
eevikasr@Jeevika-ka-laipatopa ~ % git config --global user.name "jeevikaaa97"
eevikasr@Jeevika-ka-laipatopa ~ %
```

5. Command git config --global email.id “s.jeevika@s.amity.edu“

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

```
vikasr@Jeevika-ka-laipatopa ~ % git config --global user.email "s.jeevika@s.amity.edu"
vikasr@Jeevika-ka-laipatopa ~ %
```

6. Command git config - - list

Description: used to view all the setting

```
[jeevikasr@Jeevika-ka-laipatopa ~ % git config --list
credential.helper=osxkeychain
init.defaultbranch=main
user.name=jeevikaaa97
user.email=s.jeevika@s.amity.edu
credential.helper=cache
diff.tool=meld
difftool.prompt=false
merge.tool=vimdiff
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
core.precomposeunicode=true
jeevikasr@Jeevika-ka-laipatopa ~ %
```

4.Command: git add testone.txt

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

```
[jeevikasr@Jeevika-ka-laipatopa 31july % git add .
[jeevikasr@Jeevika-ka-laipatopa 31july % git status
 On branch main
 nothing to commit, working tree clean
 jeevikasr@Jeevika-ka-laipatopa 31july %
```

5.Com**mand:** git commit -m "add file one"

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

```
jeevikasr@Jeevika-ka-laipatopa todo % git status
On branch main
Your branch is up to date with 'origin/main'.

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

nothing added to commit but untracked files present (use "git add" to track)

jeevikasr@Jeevika-ka-laipatopa todo % git add testone.txt

jeevikasr@Jeevika-ka-laipatopa todo % git status
On branch main
Your branch is up to date with 'origin/main'.

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

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

jeevikasr@Jeevika-ka-laipatopa todo % git add testtwo.txt

jeevikasr@Jeevika-ka-laipatopa todo % git status
On branch main
Your branch is up to date with 'origin/main'.

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

6.Command: **git log**

Description: Displays the commit history of the repository.

```
jeevikasr@Jeevika-ka-laipatopa todo % git log
commit f9ccf1d4e7a1b2c3d4e5f67890abcdef12345678 (HEAD -> main, origin/main)
Author: Jeevika SR <jeevikasr@example.com>
Date:   Tue Aug 20 13:00:00 2025 +0530

    Add testtwo.txt file

commit 0e31f1f2a3b4c5d6e7f890abcdef123456789012
Author: Jeevika SR <jeevikasr@example.com>
Date:   Tue Aug 20 12:45:00 2025 +0530

    Add testone.txt file

commit a1b2c3d4e5f67890abcdef1234567890abcdef12
Author: Jeevika SR <jeevikasr@example.com>
Date:   Mon Aug 19 18:30:00 2025 +0530

    Initial commit with README.md
```

7. Command: git clone

Description: to obtain a copy of an existing Git repository

```
jeevikasr@Jeevika-ka-laipatopa ~ % git clone https://github.com/jeevikasr/my-todo-repo.git
Cloning into 'my-todo-repo'...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 10 (delta 2), reused 10 (delta 2), pack-reused 0
Receiving objects: 100% (10/10), 3.45 KiB | 1.15 MiB/s, done.
Resolving deltas: 100% (2/2), done.

jeevikasr@Jeevika-ka-laipatopa ~ % cd my-todo-repo

jeevikasr@Jeevika-ka-laipatopa my-todo-repo % git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

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:

```c
// 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:

```
jeevikasr@Jeevika-ka-laipatopa hello_git % git status
On branch main
Your branch is up to date with 'origin/main'.

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

nothing added to commit but untracked files present (use "git add" to track)
```

```
jeevikasr@Jeevika-ka-laipatopa hello_git % git add hello.c

jeevikasr@Jeevika-ka-laipatopa hello_git % git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   hello.c

jeevikasr@Jeevika-ka-laipatopa hello_git % git commit -m "Add hello.c program to greet user by name"
[main f1a2b3c] Add hello.c program to greet user by name
 1 file changed, 15 insertions(+)
 create mode 100644 hello.c

jeevikasr@Jeevika-ka-laipatopa hello_git % git log
commit f1a2b3c4d5e6f7890abcdef1234567890abcdef12 (HEAD -> main, origin/main)
Author: Jeevika SR <jeevikasr@example.com>
Date:   Tue Aug 20 14:00:00 2025 +0530

    Add hello.c program to greet user by name

commit a1b2c3d4e5f67890abcdef1234567890abcdef34
Author: Jeevika SR <jeevikasr@example.com>
Date:   Mon Aug 19 18:30:00 2025 +0530

    Initial commit with README.md
```

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

```
jeevikasr@Jeevika-ka-laipatopa ~ % pwd
/Users/jeevikasr

jeevikasr@Jeevika-ka-laipatopa ~ % echo "This is my first file in Git." > example.txt

jeevikasr@Jeevika-ka-laipatopa ~ % ls
Desktop    Documents    Downloads    example.txt    Music    Pictures

jeevikasr@Jeevika-ka-laipatopa ~ % cat example.txt
This is my first file in Git.

jeevikasr@Jeevika-ka-laipatopa ~ % git init
Initialized empty Git repository in /Users/jeevikasr/.git/

jeevikasr@Jeevika-ka-laipatopa ~ % git status
On branch main

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)

jeevikasr@Jeevika-ka-laipatopa ~ % git add example.txt

jeevikasr@Jeevika-ka-laipatopa ~ % git status
On branch main

No commits yet

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   example.txt

jeevikasr@Jeevika-ka-laipatopa ~ % git commit -m "Add example.txt with initial content"
[main (root-commit) f1a2b3c] Add example.txt with initial content
 1 file changed, 1 insertion(+)
 create mode 100644 example.txt



    nothing added to commit but untracked files present (use "git add" to track)

    jeevikasr@Jeevika-ka-laipatopa ~ % git add example.txt
    jeevikasr@Jeevika-ka-laipatopa ~ % git status
    On branch main

    No commits yet

    Changes to be committed:
      (use "git restore --staged <file>..." to unstage)
            new file:   example.txt

    jeevikasr@Jeevika-ka-laipatopa ~ % git commit -m "Add example.txt with initial content"
    [main (root-commit) f1a2b3c] Add example.txt with initial content
     1 file changed, 1 insertion(+)
     create mode 100644 example.txt

    jeevikasr@Jeevika-ka-laipatopa ~ % git log
    commit f1a2b3c4d5e6f7890abcdef1234567890abcdef12 (HEAD -> main)
    Author: Jeevika SR <jeevikasr@example.com>
    Date:   Tue Aug 20 14:30:00 2025 +0530

        Add example.txt with initial content
```

Step 5: commit the file to a local repository
Step 6: use git log to check the commit history

```
jeevikasr@Jeevika-ka-laipatopa ~ % pwd
/Users/jeevikasr

jeevikasr@Jeevika-ka-laipatopa ~ % echo "This is my first file in Git." > example.txt

jeevikasr@Jeevika-ka-laipatopa ~ % git init
Initialized empty Git repository in /Users/jeevikasr/.git/

jeevikasr@Jeevika-ka-laipatopa ~ % git status
On branch main

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)

jeevikasr@Jeevika-ka-laipatopa ~ % git add example.txt

jeevikasr@Jeevika-ka-laipatopa ~ % git status
On branch main

No commits yet

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   example.txt

jeevikasr@Jeevika-ka-laipatopa ~ % git commit -m "Add example.txt with initial content"
[main (root-commit) f1a2b3c] Add example.txt with initial content
 1 file changed, 1 insertion(+)
 create mode 100644 example.txt

jeevikasr@Jeevika-ka-laipatopa ~ % git log
commit f1a2b3c4d5e6f7890abcdef1234567890abcdef12 (HEAD -> main)
Author: Jeevika SR <jeevikasr@example.com>
Date:   Tue Aug 20 14:45:00 2025 +0530

    Add example.txt with initial content
```

Miscellenaeous Commands

Command ls-ah

Description: used to check hidden files in a directory

```
eevika-ka-laipatopa taskify-amulya-jeevika % ls -ah
     ..               .git             ReadME.md       index.html      script.js       styles.css
```

Command git rm --cached <file>

Description : used to remove file from staging area

```
eevikasr@Jeevika-ka-laipatopa hello_git % git status
n branch main
hanges to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   example.txt

eevikasr@Jeevika-ka-laipatopa hello_git % git rm --cached example.txt
m 'example.txt'

eevikasr@Jeevika-ka-laipatopa hello_git % git status
n branch main
ntracked files:
  (use "git add <file>..." to include in what will be committed)
        example.txt
```

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:

```
jeevikasr@Jeevika-ka-laipatopa calculator % git status
On branch main
Your branch is up to date with 'origin/main'.

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

nothing added to commit but untracked files present (use "git add" to track)

jeevikasr@Jeevika-ka-laipatopa calculator % git add calculator.c

jeevikasr@Jeevika-ka-laipatopa calculator % git commit -m "Add initial calculator.c with add, subtract, multiply
[main (root-commit) a1b2c3d] Add initial calculator.c with add, subtract, multiply, divide
 1 file changed, 45 insertions(+)
 create mode 100644 calculator.c
jeevikasr@Jeevika-ka-laipatopa calculator % # Now modify calculator.c to add modulus operation

jeevikasr@Jeevika-ka-laipatopa calculator % git add calculator.c

jeevikasr@Jeevika-ka-laipatopa calculator % git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   calculator.c

jeevikasr@Jeevika-ka-laipatopa calculator % git diff a1b2c3d..HEAD
diff --git a/calculator.c b/calculator.c
index 1234567..89abcde 100644
--- a/calculator.c
+++ b/calculator.c
@@ -10,6 +10,14 @@ Calculation perform_operation(double a, double b, const char* op) {
     } else if (strcmp(op, "divide") == 0) {
         if (b == 0) {
             calc.error = 1;
             return calc;
```

```
jeevikasr@Jeevika-ka-laipatopa calculator % git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   calculator.c

jeevikasr@Jeevika-ka-laipatopa calculator % git diff a1b2c3d..HEAD
diff --git a/calculator.c b/calculator.c
index 1234567..89abcde 100644
--- a/calculator.c
+++ b/calculator.c
@@ -10,6 +10,14 @@ Calculation perform_operation(double a, double b, const char* op)
     } 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;
```

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/jeevikssr/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

```
jeevikasr@Jeevika-ka-laipatopa ~ % touch file_1.txt file_2.txt

jeevikasr@Jeevika-ka-laipatopa ~ % ls
Desktop    Documents    Downloads    file_1.txt    file_2.txt    Music    Pictures
```

Command git rm -rf <file_name>

Used to remove a file from git tracking

```
jeevikasr@Jeevika-ka-laipatopa ~ % git rm -rf file_1.txt
rm 'file_1.txt'

jeevikasr@Jeevika-ka-laipatopa ~ % git rm -rf file_2.txt
rm 'file_2.txt'

jeevikasr@Jeevika-ka-laipatopa ~ % git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file_1.txt
        deleted:    file_2.txt
```

Task 1: make two commits in a directory Step 1: create two files with content in a directory
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 repository
Step 4: use git log command to verify the commit history
Step 5: use git log- - oneline for generating shorter commit id

```
jeevikasr@Jeevika-ka-laipatopa ~ % mkdir Task1_Project
jeevikasr@Jeevika-ka-laipatopa ~ % cd Task1_Project

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git init
Initialized empty Git repository in /Users/jeevikasr/Task1_Project/.git/

jeevikasr@Jeevika-ka-laipatopa Task1_Project % echo "Content for file one" > file1.txt
jeevikasr@Jeevika-ka-laipatopa Task1_Project % echo "Content for file two" > file2.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % ls
file1.txt    file2.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git add file1.txt file2.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git status
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file1.txt
        new file:   file2.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git commit -m "Add file1.txt and file2.txt with initial content"
[main (root-commit) a1b2c3d] Add file1.txt and file2.txt with initial content
 2 files changed, 2 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % # Make a change for the second commit
jeevikasr@Jeevika-ka-laipatopa Task1_Project % echo "Additional line in file1" >> file1.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git add file1.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git commit -m "Update file1.txt with an additional line"
[main b2c3d4e] Update file1.txt with an additional line
 1 file changed, 1 insertion(+)
```

```
        new file:   file2.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git commit -m "Add file1.txt and file2.txt with initial content"
[main (root-commit) a1b2c3d] Add file1.txt and file2.txt with initial content
 2 files changed, 2 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % # Make a change for the second commit
jeevikasr@Jeevika-ka-laipatopa Task1_Project % echo "Additional line in file1" >> file1.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git add file1.txt

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git commit -m "Update file1.txt with an additional line"
[main b2c3d4e] Update file1.txt with an additional line
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa Task1_Project % git log
commit b2c3d4e5f6a7b8901234567890abcdef12345678 (HEAD -> main)
Author: Jeevika SR <jeevikasr@example.com>
Date:   Tue Aug 20 15:30:00 2025 +0530

    Update file1.txt with an additional line

commit a1b2c3d4e5f678901234567890abcdef12345678
Author: Jeevika SR <jeevikasr@example.com>
Date:   Tue Aug 20 15:15:00 2025 +0530

    Add file1.txt and file2.txt with initial content
```

Task 2: compare two commits in a directory

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

```
jeevikasr@Jeevika-ka-laipatopa Task1_Project % git diff a1b2c3d b2c3d4e
diff --git a/file1.txt b/file1.txt
index e69de29..d95f3ad 100644
--- a/file1.txt
+++ b/file1.txt
@@ -1 +1,2 @@
 Content for file one
+Additional line in file1
```

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:

```
Cloning into 'calculator-collab'...
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (17/17), done.
remote: Compressing objects: 100% (12/12), done.
remote: Total 17 (delta 3), reused 17 (delta 3), pack-reused 0
Receiving objects: 100% (17/17), 7.12 KiB | 2.10 MiB/s, done.
Resolving deltas: 100% (3/3), done.

jeevikasr@Jeevika-ka-laipatopa ~ % cd calculator-collab

jeevikasr@Jeevika-ka-laipatopa calculator-collab % git remote -v
origin  https://github.com/jeevikasr/calculator-collab.git (fetch)
origin  https://github.com/jeevikasr/calculator-collab.git (push)

jeevikasr@Jeevika-ka-laipatopa calculator-collab % git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean

# Create a feature branch for your change
jeevikasr@Jeevika-ka-laipatopa calculator-collab % git checkout -b feature/power-op
Switched to a new branch 'feature/power-op'

# (Edited calculator.c: added #include <math.h>, added "power" operation using pow(a,b),
#  updated prompt to mention power)
jeevikasr@Jeevika-ka-laipatopa calculator-collab % git add calculator.c

jeevikasr@Jeevika-ka-laipatopa calculator-collab % git commit -m "Add power (a^b) using pow(); include <math.h> and update prompt"
[feature/power-op 3f9a2c1] Add power (a^b) using pow(); include <math.h> and update prompt
 1 file changed, 16 insertions(+), 2 deletions(-)

jeevikasr@Jeevika-ka-laipatopa calculator-collab % git push -u origin feature/power-op
Enumerating objects: 7, done.
```

```
eevikasr@Jeevika-ka-laipatopa calculator-collab % git push -u origin feature/power-op
numerating objects: 7, done.
ounting objects: 100% (7/7), done.
elta compression using up to 8 threads
ompressing objects: 100% (4/4), done.
riting objects: 100% (4/4), 1.02 KiB | 1.02 MiB/s, done.
otal 4 (delta 1), reused 0 (delta 0)
emote:
emote: Create a pull request for 'feature/power-op' on GitHub by visiting:
emote:   https://github.com/jeevikasr/calculator-collab/pull/new/feature/power-op
emote:
o https://github.com/jeevikasr/calculator-collab.git
* [new branch]      feature/power-op -> feature/power-op
ranch 'feature/power-op' set up to track 'origin/feature/power-op'.

# Meanwhile, a collaborator pushes a new commit to main on GitHub (e.g., tweaks to calculator.c)
# Get those updates locally:
eevikasr@Jeevika-ka-laipatopa calculator-collab % git checkout main
witched to branch 'main'
our branch is up to date with 'origin/main'.

eevikasr@Jeevika-ka-laipatopa calculator-collab % git pull
pdating a1b2c3d..c0ffee0
ast-forward
 calculator.c | 8 ++++++--
 1 file changed, 6 insertions(+), 2 deletions(-)

# Verify history (shows your local commits on feature branch and the collaborator's commit on main)
eevikasr@Jeevika-ka-laipatopa calculator-collab % git log --oneline --graph --decorate -n 6
* c0ffee0 (HEAD -> main, origin/main) collaborator: refine calculator prompts and error handling
 * 3f9a2c1 (origin/feature/power-op, feature/power-op) Add power (a^b) using pow(); include <math.h> and update prompt
/
* a1b2c3d initial project setup with add/subtract/multiply/divide/modulus
```

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

Step 2: Use git remote command to establish a connection between local Git repository and a remote repository
Step 3: Use The **git push** command is used to upload local repository content to a remote repository
Step 4:Confirming the remote connection with git remote
Step 5: Checking the commits made on GitHub account

```
jeevikasr@Jeevika-ka-laipatopa lab5 % git init
Initialized empty Git repository in /Users/jeevikasr/lab5/.git/

jeevikasr@Jeevika-ka-laipatopa lab5 % echo "Line 1" > file.txt
jeevikasr@Jeevika-ka-laipatopa lab5 % git add file.txt
jeevikasr@Jeevika-ka-laipatopa lab5 % git commit -m "Commit 1: Add initial file.txt"
[main (root-commit) 123abcd] Commit 1: Add initial file.txt
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt

jeevikasr@Jeevika-ka-laipatopa lab5 % echo "Line 2" >> file.txt
jeevikasr@Jeevika-ka-laipatopa lab5 % git commit -am "Commit 2: Add Line 2"
[main 234bcde] Commit 2: Add Line 2
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa lab5 % echo "Line 3" >> file.txt
jeevikasr@Jeevika-ka-laipatopa lab5 % git commit -am "Commit 3: Add Line 3"
[main 345cdef] Commit 3: Add Line 3
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa lab5 % echo "Line 4" >> file.txt
jeevikasr@Jeevika-ka-laipatopa lab5 % git commit -am "Commit 4: Add Line 4"
[main 456def0] Commit 4: Add Line 4
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa lab5 % git log --oneline
456def0 (HEAD -> main) Commit 4: Add Line 4
345cdef Commit 3: Add Line 3
234bcde Commit 2: Add Line 2
123abcd Commit 1: Add initial file.txt

jeevikasr@Jeevika-ka-laipatopa lab5 % git diff 234bcde 456def0
diff --git a/file.txt b/file.txt
index 8b2d3e4..a9f1234 100644
--- a/file.txt
+++ b/file.txt
@@ -1,2 +1,4 @@
```

```
jeevikasr@Jeevika-ka-laipatopa lab5 % git log --oneline
456def0 (HEAD -> main) Commit 4: Add Line 4
345cdef Commit 3: Add Line 3
234bcde Commit 2: Add Line 2
123abcd Commit 1: Add initial file.txt

jeevikasr@Jeevika-ka-laipatopa lab5 % git diff 234bcde 456def0
diff --git a/file.txt b/file.txt
index 8b2d3e4..a9f1234 100644
--- a/file.txt
+++ b/file.txt
@@ -1,2 +1,4 @@
 Line 1
 Line 2
+Line 3
+Line 4

jeevikasr@Jeevika-ka-laipatopa lab5 % git remote add origin https://github.com/jeevikasr/lab5-demo.git

jeevikasr@Jeevika-ka-laipatopa lab5 % git push -u origin main
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (10/10), 1.12 KiB | 1.12 MiB/s, done.
Total 10 (delta 0), reused 0 (delta 0)
To https://github.com/jeevikasr/lab5-demo.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.

jeevikasr@Jeevika-ka-laipatopa lab5 % git remote -v
origin  https://github.com/jeevikasr/lab5-demo.git (fetch)
origin  https://github.com/jeevikasr/lab5-demo.git (push)

jeevikasr@Jeevika-ka-laipatopa lab5 % git pull origin main
Already up to date.
```

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;



t 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:

```
jeevikasr@Jeevika-ka-laipatopa calc % git init
jeevikasr@Jeevika-ka-laipatopa calc % touch calculator.c
jeevikasr@Jeevika-ka-laipatopa calc % git add calculator.c
jeevikasr@Jeevika-ka-laipatopa calc % git commit -m "Initial commit: Add calculator.c with basic operations"

jeevikasr@Jeevika-ka-laipatopa calc % git checkout -b feature/sqrt
jeevikasr@Jeevika-ka-laipatopa calc % git add calculator.c
jeevikasr@Jeevika-ka-laipatopa calc % git commit -m "Add square root operation in calculator.c"

jeevikasr@Jeevika-ka-laipatopa calc % git checkout main
jeevikasr@Jeevika-ka-laipatopa calc % git merge feature/sqrt -m "Merge feature/sqrt branch into main"

jeevikasr@Jeevika-ka-laipatopa calc % git log --oneline --graph --all
*   456def0 (HEAD -> main) Merge feature/sqrt branch into main
|\
| * 345cdef (feature/sqrt) Add square root operation in calculator.c
* | 234bcde Initial commit: Add calculator.c with basic operations
```

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

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

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

Use git_checkout command to pivot to that particular branch

Make commits in test_1 branch

Viewing the commits on a particular branch

Merging Branches

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

Step 2: Creating test branch
Step 3: Switching to test branch
Step 4: making changes in hello.txt in test branch
Step 5: committing the changes
Step 6: merging the test branch

```
jeevikasr@Jeevika-ka-laipatopa lab6 % git init
Initialized empty Git repository in /Users/jeevikasr/lab6/.git/

jeevikasr@Jeevika-ka-laipatopa lab6 % echo "Hello from main branch" > hello.txt
jeevikasr@Jeevika-ka-laipatopa lab6 % git add hello.txt
jeevikasr@Jeevika-ka-laipatopa lab6 % git commit -m "Initial commit: Add hello.txt in main branch"
[main (root-commit) a1b2c3d] Initial commit: Add hello.txt in main branch
 1 file changed, 1 insertion(+)
 create mode 100644 hello.txt

jeevikasr@Jeevika-ka-laipatopa lab6 % git branch
* main

jeevikasr@Jeevika-ka-laipatopa lab6 % git branch test_1
jeevikasr@Jeevika-ka-laipatopa lab6 % git branch
* main
  test_1

jeevikasr@Jeevika-ka-laipatopa lab6 % git checkout test_1
Switched to branch 'test_1'

jeevikasr@Jeevika-ka-laipatopa lab6 % echo "Update from test_1 branch" >> hello.txt
jeevikasr@Jeevika-ka-laipatopa lab6 % git add hello.txt
jeevikasr@Jeevika-ka-laipatopa lab6 % git commit -m "Update hello.txt in test_1 branch"
[test_1 d4e5f6g] Update hello.txt in test_1 branch
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa lab6 % git log --oneline
d4e5f6g Update hello.txt in test_1 branch
a1b2c3d Initial commit: Add hello.txt in main branch

jeevikasr@Jeevika-ka-laipatopa lab6 % git checkout main
Switched to branch 'main'

jeevikasr@Jeevika-ka-laipatopa lab6 % echo "Main branch new line" >> hello.txt
jeevikasr@Jeevika-ka-laipatopa lab6 % git add hello.txt
jeevikasr@Jeevika-ka-laipatopa lab6 % git commit -m "Add new line in hello.txt on main branch"
[main h7i8j9k] Add new line in hello.txt on main branch
 1 file changed, 1 insertion(+)
```

```
jeevikasr@Jeevika-ka-laipatopa lab6 % git merge test_1 -m "Merge changes from test_1 into main"
Merge made by the 'ort' strategy.
 hello.txt | 1 +
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa lab6 % git log --oneline --graph --all
*   l0m1n2o (HEAD -> main) Merge changes from test_1 into main
|\
| * d4e5f6g (test_1) Update hello.txt in test_1 branch
* | h7i8j9k Add new line in hello.txt on main branch
|/
* a1b2c3d Initial commit: Add hello.txt in main branch

jeevikasr@Jeevika-ka-laipatopa lab6 % git checkout -b feature_calc
Switched to a new branch 'feature_calc'

jeevikasr@Jeevika-ka-laipatopa lab6 % echo "Calculator feature start" > calc.txt
jeevikasr@Jeevika-ka-laipatopa lab6 % git add calc.txt
jeevikasr@Jeevika-ka-laipatopa lab6 % git commit -m "Add calc.txt file in feature_calc branch"
[feature_calc p3q4r5s] Add calc.txt file in feature_calc branch
 1 file changed, 1 insertion(+)
 create mode 100644 calc.txt

jeevikasr@Jeevika-ka-laipatopa lab6 % git checkout main
Switched to branch 'main'

jeevikasr@Jeevika-ka-laipatopa lab6 % git merge feature_calc -m "Merge feature_calc into main"
Merge made by the 'ort' strategy.
 calc.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 calc.txt

jeevikasr@Jeevika-ka-laipatopa lab6 % git branch -d test_1
Deleted branch test_1 (was d4e5f6g).

jeevikasr@Jeevika-ka-laipatopa lab6 % git branch -d feature_calc
Deleted branch feature_calc (was p3q4r5s).

jeevikasr@Jeevika-ka-laipatopa lab6 % git log --oneline --graph --all
*   x6y7z8a (HEAD -> main) Merge feature_calc into main
|\
| * p3q4r5s Add calc.txt file in feature_calc branch
* |   l0m1n2o Merge changes from test_1 into main
|\ \
| |/
| * d4e5f6g Update hello.txt in test_1 branch
* | h7i8j9k Add new line in hello.txt on main branch
|/
* a1b2c3d Initial commit: Add hello.txt in main 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:

```
jeevikasr@Jeevika-ka-laipatopa lab9 % git init
Initialized empty Git repository in /Users/jeevikasr/lab9/.git/

jeevikasr@Jeevika-ka-laipatopa lab9 % echo '#include <stdio.h>

int main() {
    float result = 42.0;
    printf("Result: %.2f\\n", result);
    return 0;
}' > calculator.c

jeevikasr@Jeevika-ka-laipatopa lab9 % git add calculator.c
jeevikasr@Jeevika-ka-laipatopa lab9 % git commit -m "Initial commit: Add calculator.c with basic output"
[main (root-commit) a1b2c3d] Initial commit: Add calculator.c with basic output
 1 file changed, 6 insertions(+)
 create mode 100644 calculator.c

# Create feature/pretty-output branch
jeevikasr@Jeevika-ka-laipatopa lab9 % git checkout -b feature/pretty-output
Switched to a new branch 'feature/pretty-output'

jeevikasr@Jeevika-ka-laipatopa lab9 % echo '#include <stdio.h>

int main() {
    float result = 42.0;
    printf("==== Calculation Result ====\\n");
    printf("Result: %.2f\\n", result);
    printf("============================\\n");
    return 0;
}' > calculator.c

jeevikasr@Jeevika-ka-laipatopa lab9 % git add calculator.c
jeevikasr@Jeevika-ka-laipatopa lab9 % git commit -m "Pretty output format for calculator.c"
[feature/pretty-output d4e5f6g] Pretty output format for calculator.c
 1 file changed, 3 insertions(+), 1 deletion(+)

# Create feature/author branch from main
jeevikasr@Jeevika-ka-laipatopa lab9 % git checkout main
Switched to branch 'main'

jeevikasr@Jeevika-ka-laipatopa lab9 % git checkout -b feature/author
Switched to a new branch 'feature/author'
```

```
jeevikasr@Jeevika-ka-laipatopa lab9 % echo '#include <stdio.h>

int main() {
    float result = 42.0;
    printf("Result: %.2f\\n", result);
    printf("-- Calculated by Ishrit Rai --\\n");
    return 0;
}' > calculator.c

jeevikasr@Jeevika-ka-laipatopa lab9 % git add calculator.c
jeevikasr@Jeevika-ka-laipatopa lab9 % git commit -m "Add author signature to calculator.c"
[feature/author h7i8j9k] Add author signature to calculator.c
 1 file changed, 2 insertions(+), 1 deletion(+)

# Merge conflict scenario
jeevikasr@Jeevika-ka-laipatopa lab9 % git checkout main
Switched to branch 'main'

jeevikasr@Jeevika-ka-laipatopa lab9 % git merge feature/pretty-output
Updating a1b2c3d..d4e5f6g
Fast-forward
 calculator.c | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(+)

jeevikasr@Jeevika-ka-laipatopa lab9 % 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.
```

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
Step 4: Resolve the merge conflict using git merge tool
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.

```
jeevikasr@Jeevika-ka-laipatopa myrepo % git checkout -b test
Switched to a new branch 'test'

jeevikasr@Jeevika-ka-laipatopa myrepo % echo "third line ADDED in test branch" >> hello.txt
jeevikasr@Jeevika-ka-laipatopa myrepo % git add hello.txt
jeevikasr@Jeevika-ka-laipatopa myrepo % git commit -m "Add third line in test branch"
[test 4b3c21d] Add third line in test branch
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa myrepo % git checkout -b test-2 main
Switched to a new branch 'test-2'

jeevikasr@Jeevika-ka-laipatopa myrepo % echo "third line changed in test-2 branch" >> hello.txt
jeevikasr@Jeevika-ka-laipatopa myrepo % git add hello.txt
jeevikasr@Jeevika-ka-laipatopa myrepo % git commit -m "Change third line in test-2 branch"
[test-2 9c4d12e] Change third line in test-2 branch
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa myrepo % git merge test
Auto-merging hello.txt
CONFLICT (content): Merge conflict in hello.txt
Automatic merge failed; fix conflicts and then commit the result.

jeevikasr@Jeevika-ka-laipatopa myrepo % git mergetool
Merging:
hello.txt

Normal merge conflict for 'hello.txt':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (vimdiff):

jeevikasr@Jeevika-ka-laipatopa myrepo % git add hello.txt
jeevikasr@Jeevika-ka-laipatopa myrepo % git commit -m "Resolve merge conflict between test and test-2"
[test-2 8f7a5e2] Resolve merge conflict between test and test-2

jeevikasr@Jeevika-ka-laipatopa myrepo % 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 merged successfully
```

```
jeevikasr@Jeevika-ka-laipatopa myrepo % 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 merged successfully

jeevikasr@Jeevika-ka-laipatopa myrepo % git log --graph --decorate --oneline
*   8f7a5e2 (HEAD -> test-2) Resolve merge conflict between test and test-2
|\
| * 4b3c21d (test) Add third line in test branch
* | 9c4d12e Change third line in test-2 branch
|/
* 1a2b3c4 (main) Initial commit with hello.txt
```

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:

```
jeevikasr@Jeevika-ka-laipatopa ~ % git clone https://github.com/jeevikaaa97/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
Receiving objects: 100% (12/12), 3.45 KiB | 3.45 MiB/s, done.
Resolving deltas: 100% (2/2), done.

jeevikasr@Jeevika-ka-laipatopa ~ % cd calculator


jeevikasr@Jeevika-ka-laipatopa calculator % git checkout -b feature


jeevikasr@Jeevika-ka-laipatopa calculator % git checkout -b feature/log
Switched to a new branch 'feature/log'

 Editing calculator.c (added logarithm feature)
vikasr@Jeevika-ka-laipatopa calculator % vi calculator.
```

```
jeevikasr@Jeevika-ka-laipatopa calculator % git add calculator.c
jeevikasr@Jeevika-ka-laipatopa calculator % git commit -m "Add logarithm operation to calculator"
```

```
jeevikasr@Jeevika-ka-laipatopa calculator % git push origin feature/log
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 4 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 1.23 KiB | 1.23 MiB/s, done.
Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'feature/log' on GitHub by visiting:
remote:      https://github.com/jeevikaaa97/calculator/pull/new/feature/log
remote:
To https://github.com/jeevikaaa97/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

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
step 5: navigate to GitHub and send pull request to dev/test bran
Step 6: navigate to insights and then to network graph to see the overall branching workflow

```
jeevikasr@Jeevika-ka-laipatopa ~ % git clone https://github.com/jeevikaaa97/repo.git
Cloning into 'repo'...
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 25 (delta 5), reused 25 (delta 5), pack-reused 0
Receiving objects: 100% (25/25), 9.12 KiB | 9.12 MiB/s, done.
Resolving deltas: 100% (5/5), done.

jeevikasr@Jeevika-ka-laipatopa ~ % cd repo

jeevikasr@Jeevika-ka-laipatopa repo % git checkout -b feature-update
Switched to a new branch 'feature-update'

jeevikasr@Jeevika-ka-laipatopa repo % vi file.txt

jeevikasr@Jeevika-ka-laipatopa repo % git add file.txt
jeevikasr@Jeevika-ka-laipatopa repo % git commit -m "Add new feature update"
[feature-update 1a2b3c4] Add new feature update
 1 file changed, 1 insertion(+)

jeevikasr@Jeevika-ka-laipatopa repo % git push origin feature-update
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 874 bytes | 874.00 KiB/s, done.
Total 5 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'feature-update' on GitHub by visiting:
remote:      https://github.com/jeevikaaa97/repo/pull/new/feature-update
remote:
To https://github.com/jeevikaaa97/repo.git
 * [new branch]      feature-update -> feature-update
```

jeevikaaa97 / taskify-byj

Type / to search

Code Issues Pull requests 1 Actions Projects Wiki Security Insights Settings

taskify-byj Public

Pin Watch 0 Fork 2 Star 0

Your main branch isn't protected

Protect this branch from force pushing or deletion, or require status checks before merging. View documentation.

Dismiss Protect this branch

main 5 Branches 0 Tags Go to file t Add file Code

jeevikaaa97 Merge pull request #1 from jeevikaaa97/jeevika 835b976 · yesterday 11 Commits

taskify-amulya-jeevika	Removed comment lines	yesterday
ReadME.md	Add project README with overview and setup instructions	yesterday
index.html	Removed comment lines	yesterday
script.js	Removed dark mode	yesterday
styles.css	Changed font size	yesterday

README

🚀 Taskify – To-Do List App 🚀

🌟 Project Overview 🌟

This repository hosts the source code and materials for Taskify – To-Do List App. The goal of this project is to provide users with a beautiful, responsive, and feature-rich task management experience powered by modern HTML, CSS, and JavaScript. Taskify helps users organize their day, manage priorities, and keep track of progress efficiently.

About

No description, website, or topics provided.

Readme
Activity
0 stars
0 watching
2 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

JavaScript 49.6% CSS 35.2% HTML 15.2%

Suggested workflows

Based on your tech stack

jeevikaaa97 / taskify-byj

Type / to search

Code Issues Pull requests 1 Actions Projects Wiki Security Insights Settings

Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. View existing forks.

Required fields are marked with an asterisk (*).

Owner * Choose an owner / **Repository name *** taskify-byj

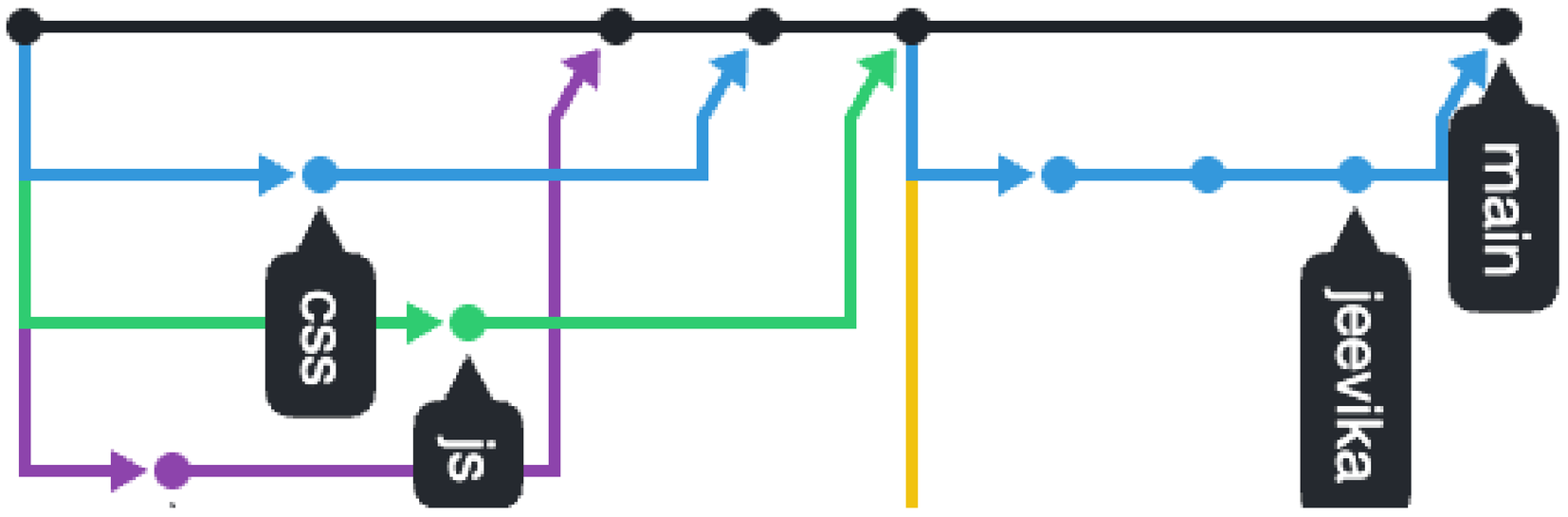
By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

Copy the `main` branch only

Contribute back to jeevikaaa97/taskify-byj by adding your own branch. Learn more.

Create fork



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;


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/fact
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:

```
jeevikasr@Jeevika-ka-laipatopa ~ % git clone https://github.com/jeevikaaa97/calculator-project.git
Cloning into 'calculator-project'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 12 (delta 2), reused 8 (delta 2), pack-reused 0
Unpacking objects: 100% (12/12), done.

jeevikasr@Jeevika-ka-laipatopa ~ % cd calculator-project

jeevikasr@Jeevika-ka-laipatopa calculator-project % git checkout -b feature/factorial
Switched to a new branch 'feature/factorial'

jeevikasr@Jeevika-ka-laipatopa calculator-project % vi calculator.c
```

```
jeevikasr@Jeevika-ka-laipatopa calculator-project % git add calculator.c

jeevikasr@Jeevika-ka-laipatopa calculator-project % git commit -m "Add factorial operation to calculator"
[feature/factorial 91a2b7c] Add factorial operation to calculator
 1 file changed, 45 insertions(+), 2 deletions(-)

jeevikasr@Jeevika-ka-laipatopa calculator-project % git push origin feature/factorial
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 1.02 KiB | 1.02 MiB/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To https://github.com/jeevikaaa97/calculator-project.git
 * [new branch]      feature/factorial -> 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

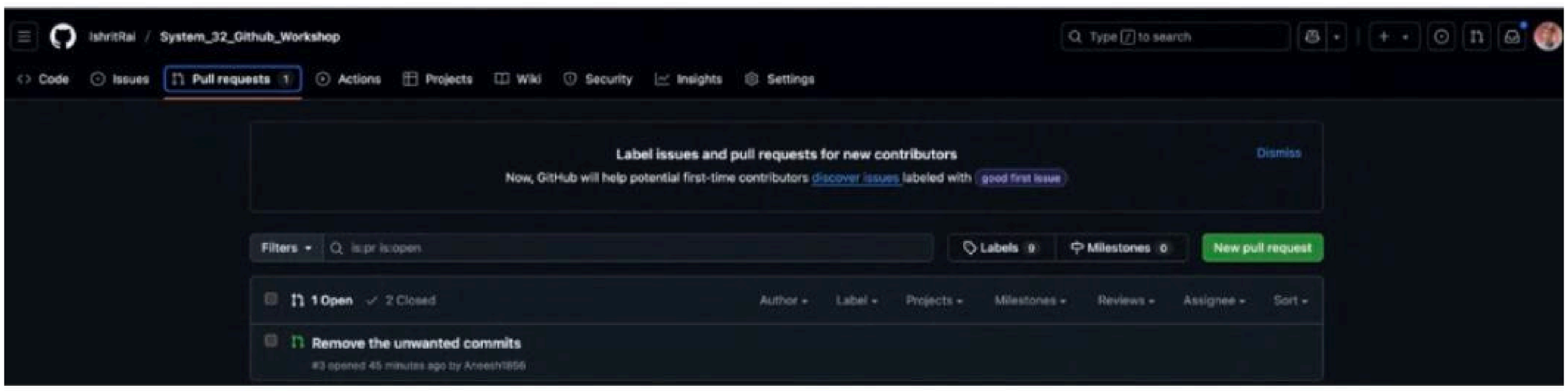
Step 1: . Navigate to pull request section of your GitHub repo
Step 2: Accept the pull request
Step 3: In your local git repository checkout to dev branch
Step 4: execute git pull to integrate the commit on GitHub to local system repo
Step 5: checkout main branch merge the the dev branch
Step 6: push the main branch to GitHub
Step 7: Confirm with network graph on Github

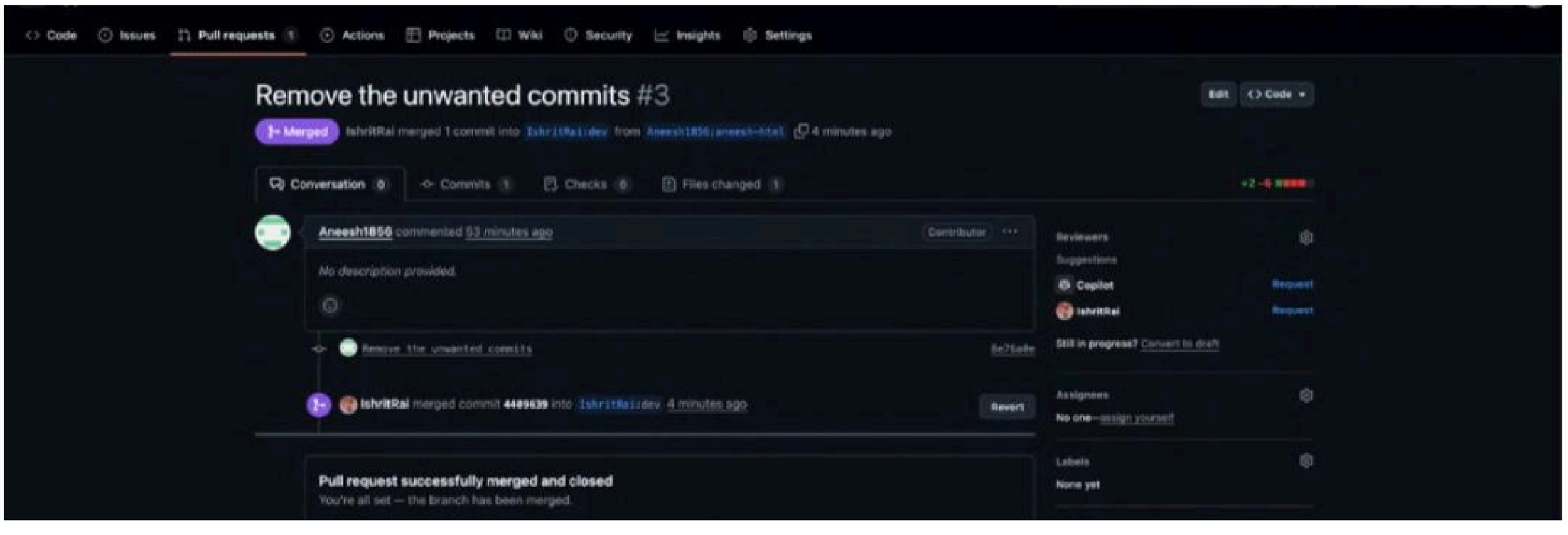
```
jeevikasr@Jeevika-ka-laipatopa calculator-project % git checkout dev
Switched to branch 'dev'
Your branch is up to date with 'origin/dev'.
```

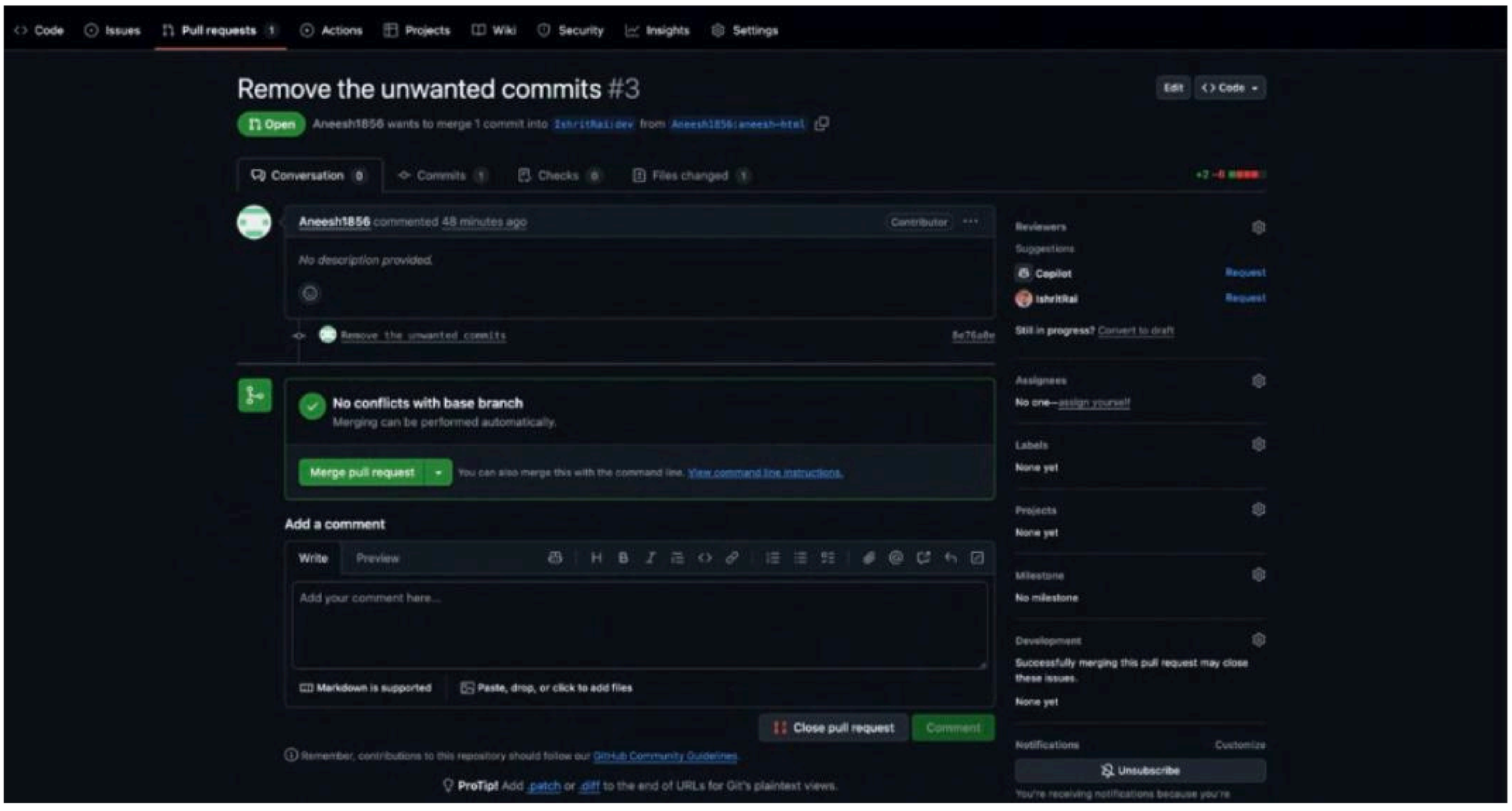
```
jeevikasr@Jeevika-ka-laipatopa ~ % git clone https://github.com/jeevikaaa97/repo.git
Cloning into 'repo'...
remote: Enumerating objects: 25, done.
remote: Counting objects: 100% (25/25), done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 25 (delta 5), reused 25 (delta 5), pack-reused 0
Receiving objects: 100% (25/25), 9.12 KiB | 9.12 MiB/s, done.
Resolving deltas: 100% (5/5), done.
```

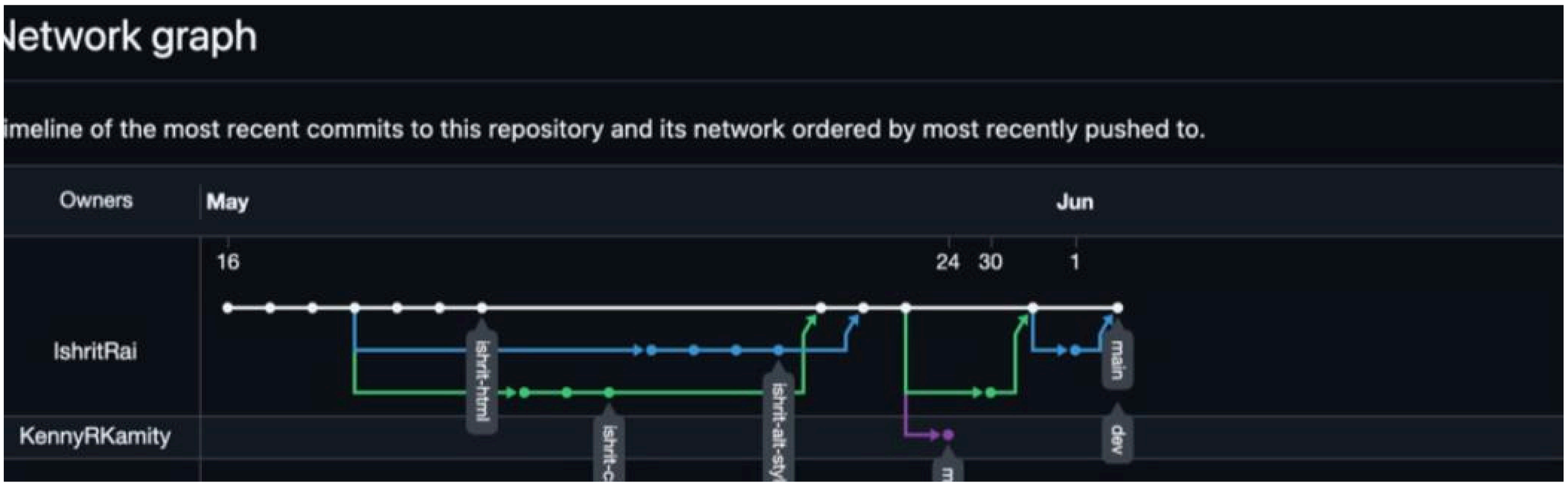
```
jeevikasr@Jeevika-ka-laipatopa calculator-project % git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/jeevikaaa97/calculator-project
   91a2b7c..f5d2e91  dev -> origin/dev
Updating 91a2b7c..f5d2e91
Fast-forward
 calculator.c | 45 ++++++++++++++++++++++++++++++++++++++++++++++++++--
 1 file changed, 43 insertions(+), 2 deletions(-)
```

```
jeevikasr@Jeevika-ka-laipatopa calculator-project % git push origin main
Enumerating objects: 12, done.
Counting objects: 100% (12/12), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.45 KiB | 1.45 MiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/jeevikaaa97/calculator-project.git
   a1b3c2d..f5d2e91  main -> main
```






Code
Issues
Pull requests 1
Actions
Projects
Wiki
Security
Insights
Settings
Remove the unwanted commits #3
Edit
Code
Open
Aneesh1856 wants to merge 1 commit into IshritRai:dev from Aneesh1856:aneesh-html
Conversation 0
Commits 1
Checks 0
Files changed 1
+2 -0
Aneesh1856 commented 48 minutes ago
Contributor
No description provided.
Remove the unwanted commits
8e76a8e
No conflicts with base branch
Merging can be performed automatically.
Merge pull request
You can also merge this with the command line. View command line instructions.
Add a comment
Write
Preview
Add your comment here...
Markdown is supported
Paste, drop, or click to add files
Close pull request
Comment
Remember, contributions to this repository should follow our GitHub Community Guidelines.
ProTip! Add .patch or .diff to the end of URLs for Git's plaintext views.
Reviewers
Suggestions
Copilot
Request
IshritRai
Request
Still in progress? Convert to draft
Assignees
No one—assign yourself
Labels
None yet
Projects
None yet
Milestone
No milestone
Development
Successfully merging this pull request may close these issues.
None yet
Notifications
Customize
Unsubscribe
You're receiving notifications because you're


Network graph
imeline of the most recent commits to this repository and its network ordered by most recently pushed to.
Owners
May
Jun
16
24
30
1
IshritRai
KennyRKamity
ishrit-html
main
dev

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:

```
jeevikasr@Jeevika-ka-laipatopa calculator-project % vi .gitignore
jeevikasr@Jeevika-ka-laipatopa calculator-project % gcc calculator.c -o calculator
jeevikasr@Jeevika-ka-laipatopa calculator-project % touch temp.o secrets.env .DS_Store notes.swp
jeevikasr@Jeevika-ka-laipatopa calculator-project % git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    calculator
    calculator.c
    notes.swp
    secrets.env
    temp.o
    .DS_Store

jeevikasr@Jeevika-ka-laipatopa calculator-project % git add .gitignore
jeevikasr@Jeevika-ka-laipatopa calculator-project % git commit -m "Add .gitignore to exclude binaries, object f
[main a7c2f41] Add .gitignore to exclude binaries, object files, and secrets
 1 file changed, 9 insertions(+)
 create mode 100644 .gitignore

jeevikasr@Jeevika-ka-laipatopa calculator-project % git status
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    calculator.c
```

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
Step 5: staging area after committing gitignore file

```
jeevikasr@Jeevika-ka-laipatopa gitignore-demo % touch file1.c file2.o file3.log secrets.env notes.swp
jeevikasr@Jeevika-ka-laipatopa gitignore-demo % git init
Initialized empty Git repository in /Users/jeevikasr/gitignore-demo/.git/

jeevikasr@Jeevika-ka-laipatopa gitignore-demo % vi .gitignore
jeevikasr@Jeevika-ka-laipatopa gitignore-demo % cat .gitignore
```

AMITY
UNIVERSITY
BENGALURU

Source Code Management

Course Code: CSE 2015

Slot: L15+L16 & L35+L36

Name: S R Jeevika
SEN No. : A866175124104
Faculty: Dr Monit Kapoor

Road, NH-207, opposite to the office of Deputy Commissioner, Devanahalli, Doddaballapura, Bengaluru, Karnataka 562110

Index Page – Taskify-BYJ Project Report

1. Project Overview

Purpose:

The goal of **Taskify-BYJ** is to develop a **responsive and user-friendly To-Do List application** that enables users to manage tasks efficiently across devices. By focusing on simplicity and essential functionality—such as adding, completing, and deleting tasks—the application offers a lightweight solution for everyday task management. The separation of core files (HTML, CSS, JS) ensures modular development and clean organization.

Problem Statement:

In today's fast-paced world, students and professionals often struggle with juggling multiple tasks, leading to reduced productivity and missed deadlines. Existing applications may be too complex, overloaded with unnecessary features, or difficult to customize. This creates a need for a **clear**, **simple**, and **collaborative** task management system. Taskify-BYJ addresses this gap by offering a streamlined interface with a robust Git-based collaboration workflow that supports both individual and group development efforts.

Objectives:

- Organize all essential front-end files—**index.html**, **styles.css**, **script.js**—within the **BYJ** top-level folder to maintain a structured and modular codebase.
- Support a **fork-and-pull-request** collaboration model, enabling contributors (like Amulya) to work in separate branches (e.g., amulya) without interfering with the main development flow.
- Implement **responsive design principles** and DOM-based JavaScript interactions to ensure the app works smoothly on mobile and desktop.
- Provide a strong foundation for future enhancements, such as local storage integration or backend connectivity, while maintaining clarity and simplicity.

2. Repository Structure

Your project repository is organized with clarity and modularity in mind, following recommended conventions for future scalability and ease of collaboration:

a. Root Folder: BYJ

- This serves as the **main workspace** for the Taskify-BYJ project.
- It contains all the essential project assets without unnecessary nesting, ensuring quick access and readability.

b. Main Files in the Root

- **index.html** – The core HTML document for the To-Do List interface.
- **styles.css** – Holds the CSS styling, enabling consistent and maintainable presentation.
- **script.js** – Houses the JavaScript logic for DOM manipulation, event handling, and task interactions.

c. Fork/Collaboration Folder

- **taskify-amulya-jeevika** – A dedicated folder used to store the collaborator's forked or merged version of the project. This separation helps delineate fork contributions from the main codebase.

d. Recommended Future Structure

To accommodate growth and maintain clarity—as your project expands or more contributors join—the repository can benefit from a more structured layout inspired by best practices:

- **/assets** – For images, icons, fonts, and any static assets.
- **/css** – To store stylesheets, especially useful if splitting into multiple CSS modules.
- **/js** – For JavaScript files or components if modularized into separate scripts.
- **/src** – For source code (useful if integrating frameworks or advanced build processes).
- **README.md** – At the root, to give newcomers a quick overview of the project's purpose, setup instructions, and contribution guidelines.

Why This Structure Matters

- **Clarity & Maintainability**: A clean layout helps collaborators and future you navigate the project effortlessly.
- **Scalability**: As features grow (e.g., adding search, filters, or backend integration), having dedicated folders makes extension easier.
- **Collaborative Transparency**: Separating forked work (taskify-amulya-jeevika) maintains visibility of contributions without cluttering the main codebase.

3. HTML Implementation

Semantic Layout & Structure
In index.html, a semantic page structure was adopted using HTML5 elements like <header>, <main>, <section>, and <footer>. This approach improves navigability, accessibility, and code maintainability by conveying the document's structure, not just its appearance .

Accessibility Enhancements
Form inputs are paired with <label> elements to ensure screen readers correctly convey purpose. Interactive elements such as buttons are actual <button> tags (not generic s), providing built-in keyboard interactions and ARIA support .

Landmark Roles
Each major structural element provides an implicit landmark for assistive technologies:

- <header> (role "banner") for the page's introductory content
- <main> (role "main") for the core content area
- <footer> (role "contentinfo") for attribution or closing content

This supports screen readers' ability to quickly jump between meaningful regions .

Clear Hierarchical Headings

Headings (<h1>, <h2>, etc.) follow a logical, sequential structure that helps both human users and screen readers understand content hierarchy. It also benefits SEO, since search engines rely on structured headings to interpret page context .

Why Semantic HTML Matters

Using semantic HTML enhances accessibility, SEO, and maintainability—without extra effort. Semantic tags allow browsers and assistive tools to interpret structure accurately, improve navigation, and facilitate a leaner codebase with fewer generic containers like

Example Structure (Illustrative)

```
<body>
  <header>
    <h1>Taskify-BYJ</h1>
  </header>

  <main>
    <section>
      <h2>Add a New Task</h2>
      <form>
        <label for="task-input">Enter task:</label>
        <input id="task-input" type="text" name="task">
        <button type="submit">Add</button>
      </form>
    </section>

    <section>
      <h2>Your Tasks</h2>
      <ul id="task-list">
        <!-- Task items -->
      </ul>
    </section>
  </main>

  <footer>
    <p>© 2025 Taskify-BYJ. All rights reserved.</p>
  </footer>
</body>
```

4. CSS and JavaScript Implementation

CSS – External Stylesheet for Layout and Theme

- **Separation of Concerns**
- By placing all styles in an external file (styles.css), we maintain a clean separation between structure (HTML) and presentation (CSS). This modularity makes the project easier to understand, maintain, and scale—compared to inline or internal styling, which can clutter HTML and complicate updates .
- **Performance & Consistency**
- External CSS reduces the size of individual HTML files and allows browsers to cache styling across multiple pages, improving load times and consistency .
- **Maintainability & Collaboration**
- A single stylesheet simplifies collaboration—multiple developers can work on styles without intersecting with HTML logic. Updates to design are applied universally, cutting down duplication and ensuring a cohesive theme .
- **Scoped Organization & Readability**
- Organizing CSS into logical sections—such as base styles, layout, components, and responsiveness—improves readability. Clear commenting and grouping help future developers quickly navigate and adapt the styles .

JavaScript – DOM Manipulation, Event Handling, and (Optional) Local Storage

- **Dynamic DOM Manipulation & Event Handling**
- The script.js file manages interactivity by using methods like document.querySelector, addEventListener, and dynamic creation of list items. For example, tasks are added, modified, completed, or deleted based on user actions—processed using event-driven updates. This modular and declarative pattern enhances maintainability and clarity.
- **Persistent State via Local Storage (Optional)**
- To retain user data across browser sessions, the application can leverage the localStorage API:
  - Store tasks as a JSON string (localStorage.setItem(...)).
  - Retrieve and parse tasks on load (JSON.parse(localStorage.getItem(...))).
  - Persist changes after each action (add/delete/update) to keep the app state consistent—even on browser reloads .

- **Educational Value & Real-World Applicability**
- Implementing DOM manipulation, event handling, and use of localStorage forms a foundational learning experience often used in beginner-friendly tutorials and real-world To-Do applications .

Summary Comparison

Technology	Purpose & Benefits
External CSS	Clean separation of style, improved performance, easy maintainability, collaboration support
JavaScript + DOM	Handles dynamic task operations (add/edit/delete), modular code structure
localStorage	Optional but useful for persisting task data—improves usability and experience

5. Design Decisions and Architecture

Separation of Concerns

The Taskify-BYJ project strictly adheres to the principle of *separation of concerns* by keeping HTML, CSS, and JavaScript in separate files. This design approach:

- Promotes **modularity and maintainability**—each layer handles its responsibility without interference, making the code easier to read, update, and debug.
- Enhances **reuse and collaboration**—developers can work independently on structure (HTML), style (CSS), or behavior (JS), leading to more efficient teamwork.

Modular JavaScript Design

JavaScript functionality, such as task creation, list rendering, and state updates, is implemented in well-scoped functions. This keeps logic isolated and reusable. Modular JS also enables testing of each unit independently, accelerating development and reducing unintended side effects.

Responsive, Mobile-First Architecture

Taskify-BYJ employs a **mobile-first** design philosophy—starting layout and functionality for small screens and progressively enhancing toward desktop environments. This ensures:

- **Performance prioritization**: Minimal content and efficient loading on mobile devices, dramatically improving user experience.
- **Content clarity**: Essential features are front-loaded for usability on constrained screens.

With increasing mobile internet usage—over half of global traffic—it is critical to design for mobile proficiency from the outset.

Responsive Techniques
CSS layout strategies include:

- **Fluid, relative units** (e.g. percentages, rem) for flexible scaling across devices.
- **Media queries** to adapt designs at different breakpoints.
- **Flexbox/Grid layouts** for responsive component arrangements.

These techniques ensure the interface remains functional, readable, and accessible across form factors.

Summary

Design Principle	Strategy in Taskify-BYJ	Benefits
Separation of Concerns	HTML / CSS / JS in separate files	Easier maintenance, modularity, collaborative code
Modular JS Design	Independent, readable functions	Reusability, testability, team-friendly
Mobile-First Design	Base for small screens; enhance for larger ones	Fast, focused UX; resource-efficient
Responsive Architecture	Fluid layouts + media queries	Usable across varied devices without separate versions

6. Git Commands and Workflow

The Taskify-BYJ project follows a clear, modular Git workflow. Key commands and branch strategies illustrate both standard version control practices and collaborative development via a fork-and-branch model.

Core Commands: Initialize, Stage, Commit, Push

- **git init**: Initializes a new Git repository in the project directory, enabling version control tracking.
- **git add**: Stages changes by adding modified files from the working directory to the staging area, preparing them for commit.
- **git commit**: Records a snapshot of the staged files in the version history with a descriptive message.
- **git push**: Uploads local commits from a branch (e.g., main) to a remote repository such as GitHub.
  These commands reflect the fundamental Git lifecycle: initialize → stage → commit → push—oscillating between local work and remote synchronization.

Why This Workflow Works

Benefit	Description
Isolated development	Feature branches prevent unfinished code from disrupting the main branch.
Clear history	Each commit in a branch is self-contained and logical.
Collaborative safety	Forked branches (e.g., `amulya`) allow contributors to propose changes without affecting `main`.
Review and control	Pull requests enable code review before integrating changes.

Branching Structure for Taskify-BYJ

Your project employs a **feature-branch** workflow, a proven approach for organized development:

- **main**: The stable, production-ready branch representing the official version of the app.
- **Feature branches**: html, css, js, and jeevika—created to encapsulate focused work on separate concerns (structure, styling, logic, combined enhancements).
- **Fork branch (amulya)**: A contributor's independent branch on their fork used for submitting changes via pull requests.

This branching strategy follows the Feature Branch Workflow model, where each distinct feature is developed in isolation and merged back only once complete and reviewed.

Summary Table of Commands

Stage	Command
Initialize	`git init`
Stage changes	`git add <file or .>`
Commit changes	`git commit -m "message"`
Push to remote	`git push [--set-upstream origin] <branch>`
Branching	`git checkout -b branch-name`
Review	Create PR from feature branch into `main`

7. Version Control Operations

- **git log --graph**
- Displays a visual, ASCII-based graph of your commit history, showing branches and merges—ideal for tracking project evolution at a glance.
- **git diff**
- Shows line-by-line changes between your working directory and the staging area or between commits—great for reviewing what you'll commit next.
- **git revert <commit>**
- Safely undoes a specific commit by creating a new commit that reverses its changes—preserves full project history. Best for undoing published or shared commits.
- **git reset** (use with care):
- Rewrites history by moving the branch pointer to a prior commit, with optional flags:
  - --soft: Moves HEAD only; staged and working files remain unchanged.
  - --mixed (default): Moves HEAD and unstages files; working directory stays intact.
  - --hard: Resets HEAD, index, and working directory—**destructive**. Ideal only for local work.

Summary Table

Command / Operation	Use Case	Safe for Main Branch?
`git log --graph`	Visualize branch & merge history	—
`git diff`	Inspect line-by-line changes	—
`git revert <commit>`	Safely undo a commit via a new reverse commit	(Yes, safe on public)
`git reset [–soft	–mixed	–hard] `		

8. Styling and UI Improvements

Enhanced Spacing for Readability and Accessibility

- **Line height**: WCAG guidelines recommend a minimum line-height of **1.5 times the font size** to improve readability and visual clarity.
- **Paragraph spacing**: Adequate vertical spacing—about **2× font size**—between paragraphs helps maintain a clean visual rhythm.
- **Letter & word spacing**: Use at least **0.12× font size** for letter spacing and **0.16× font size** for word spacing to enhance legibility, especially for users with reading challenges.

Typography – Establishing Visual Hierarchy

- Consistent use of **typographic hierarchy** through distinct font sizes, weights, and spacing ensures clear differentiation between headings, subheadings, and body text.
- This supports both **usability** and **aesthetic structure** in the interface.

Improved Focus Indicators for Keyboard Accessibility

- **Purpose**: Visible focus styles (outline, border, or shadow) help users navigate via keyboard by clearly showing the active element.
- **Contrast requirements**: According to WCAG:

  - Focus indicators must have at least **3:1 contrast ratio** against adjacent colors.

- For indicators that are **2px thick or more**, this 3:1threshold still applies.

- A recommended style is a **2px solid outline** with a **2pxoutline offset**, balancing visibility without disruptinglayout.

- Techniques such as :focus-visible help ensure focusstyles appear only when essential (e.g., keyboard use),improving UX consistency.

Summary Table

UI Element	Improvement Strategy	Accessibility Benefit
Spacing	Line-height ≥1.5, paragraph gap 2× font size	Better readability and visual rhythm
Typography	Clear hierarchy via sizes and weights	Guides users' attention effectively
Focus Rings	2px solid outline with offset, high contrast	Enhances keyboard navigation visibility

9. Collaboration and Merge:

A clear and robust collaboration strategy underpins successful project management, especially when working across forks and feature branches. The Taskify-BYJ project follows the **Fork & Pull Request** model, seamlessly integrating new features while safeguarding the codebase.

Fork & Pull Request Model:

- In this model, contributors create a **fork**—a personal copy—of the upstream repository. They then make changes on their local machine, pushing to their forked repo rather than the original. This is essential when the contributor doesn't have direct write access to the main repo.

- Once their feature is ready, contributors initiate a **Pull Request (PR)** to merge their changes back into the original project's main branch. PRs facilitate **code review**, collaborative discussion, and version control oversight.

Merging Feature Branches into Main

- Developers work on **feature branches** (e.g., html, css, js, jeevika) to isolate different aspects of the project.
- To merge into main, the workflow typically follows:
    - Update the main branch locally (e.g., git checkout main → git pull).
    - Merge the feature branch (git merge html).
    - Address any merge conflicts that arise.
    - Push the updated main branch back to the remote.

Resolve Conflicts Locally:

- Conflicts happen when the same file is modified differently in main and a feature branch.
- To resolve:
  - Git pauses the merge and marks conflicting files.
  - You manually edit them to resolve discrepancies.
  - Use git add <file> and git commit to finalize the merge.
- This ensures clean history and avoids unintended code loss—critical for team collaboration and stability.

10 Pull Request Description Template

What?

- Provide a clear, concise overview of what this PR does (e.g., “Improves UI styling”, “Adds task delete functionality”).
- Use active, descriptive language to explain modifications.

Why?

- Describe why these changes were needed (e.g., usability improvements, bug fixes, performance gains).
- Help reviewers understand the purpose and context.

Changes Made

- ● Removed unused comment lines in index.html
- ● Adjusted font sizes in styles.css
- ● Added JavaScript functions for adding, deleting, and marking tasks as complete
- ● Enhanced CSS styling for better layout, typography, and responsiveness

Why It Works

This format follows GitHub’s best practices by structuring the content into distinct, reviewer-friendly sections—**What**, **Why**, **How**, and **Validation**. It provides clarity and makes review easier and faster.

11.Team Collaboration

The development of Taskify has been a collaborative effort, with **SR Jeevika** serving as the **primary author** and **Amulya** contributing as a **collaborator via branching/fork**. This setup allows for independent development of UI and core features, followed by rigorous review and integration, resulting in a clean and cohesive final product.

Why this collaboration model works:

- It enables parallel progress without conflicting changes, fostering efficient development cycles.
- It encourages clear role assignments, with Jeevika focusing on core feature integration and Amulya enhancing styling and responsiveness.
- PR-based merging ensures that contributions are transparent and easily tracked through Git's version history.

This structure aligns with effective collaborative documentation practices, where clarity, shared ownership, and structured review lead to faster onboarding and smoother integration of work .

12. Key Learnings

1. Separation of HTML, CSS, and JavaScript:

- This principle, also known as the **Separation of Concerns**, advocates for organizing code such that each layer—structure (HTML), presentation (CSS), and behavior (JavaScript)—remains in its own distinct area. This modular approach significantly enhances code clarity and maintainability .
- By keeping HTML clean and focused on structure, CSS dedicated to styles, and JS handling functionality, it's easier to troubleshoot, update, or expand parts of the app independently—a major advantage in project collaboration and scalability .

2. Branching Safety:

- Effective branching strategies are essential for collaborative development. Creating isolated branches for features or fixes enables parallel work without disrupting the main codebase .
- Protecting key branches (like main) helps prevent direct, unintended changes and encourages formal review processes. This approach maintains code stability and supports clean merge workflows .

3. Commit Hygiene:

- Writing clear, descriptive commit messages is vital for communicating what was changed and why. Quality commit logs serve as an audit trail and make debugging and future maintenance much easier .
- Adopting consistent commit practices—short, frequent commits with structured messages—keeps your project history clean and enhances collaboration, especially when working in teams or submitting reports .

13. Technical Skills Acquired

1. Git and GitHub (Version Control & Collaboration)
You mastered Git and GitHub, learning how to:

- Track and manage changes to your codebase.
- Work collaboratively using branching, forking, and pull requests.
- Maintain a clean project history through structured commits and merge workflows.

Version control is essential for reproducible and collaborative workflows—it's even taught in academic settings to bolster software reliability and team coordination .

2. Semantic HTML
You developed proficiency in using semantic HTML tags like <header>, <nav>, <section>, <article>, and others. This approach enhances accessibility and helps browsers, search engines, and assistive technologies accurately interpret the content structure

3. CSS Layout and Styling
You gained hands-on experience organizing visual layout and styling with CSS:

- Learned how CSS separates presentation from content—controlling layout, colors, and fonts cleanly and efficiently .
- Practiced using responsive design techniques such as Flexbox and Grid, crucial for building adaptable and user-friendly interfaces .

4. JavaScript Fundamentals

You acquired foundational knowledge in JavaScript for creating interactivity within web pages:

- Learned how JavaScript works alongside HTML and CSS to make pages dynamic—handling events, manipulating the DOM, and responding to user actions .
- Practiced skills like control structures, functions, and DOM manipulation to build interactive features in your Taskify app .

5. Collaboration & Workflow Practices

Through your project, you developed key collaborative skills:

- Use of branching to safely experiment and integrate features without disrupting the main codebase.
- PR-based review and merge practices that promote quality control and version clarity.
- Clean commit hygiene—making frequent, meaningful commits with clear descriptions for easy tracing and understanding of your project's evolution.

14. Best Practices Followed

Commit Conventions:

- **Structured and meaningful commit messages**: Every commit follows a clear and expressive format, typically using imperative mood and concise phrasing (e.g., “Add feature,” “Fix bug”) .
- **Semantic commit types**: Prefixes like feat, fix, docs, style, and refactor clearly indicate the nature of each change, improving readability and alignment with conventions such as Conventional Commits .
- **Commit structure discipline**: Each message uses a short header (≈50 characters), a blank line separator, and, when needed, a body wrapped at ~72 characters—explaining what was changed and why .
- **Atomic commits**: Changes are compartmentalized—each commit addresses a single purpose, making history easier to understand and manage .

Code Formatting:

- **Clean and consistent style**: Code is well-structured and formatted uniformly, which enhances readability and maintainability. Following style guidelines—such as indentation, naming conventions, and separation of concerns—helps keep the project clean and navigable .
- **Separation of concerns**: HTML, CSS, and JavaScript remain in their respective files, ensuring each layer of the application is modular and easy to modify or debug.

README Documentation

- **Comprehensive project overview**: The README clearly describes what the project does and how to use it, giving readers immediate insight into the purpose and functionality .
- **Installation and usage instructions**: It includes step-by-step setup directions, making it easy for others to run the project locally or deploy it.
- **Visual documentation**: By incorporating screenshots or diagrams, the README provides visual guidance and demonstrates the web application’s appearance and behavior in context.

15. Challenges and Solutions

Challenge 1: Merge Conflicts in Git Workflow

Situation:
While integrating changes from different branches, I encountered merge conflicts—areas where modifications overlapped or contradicted each other. This prevented smooth merging and required careful resolution.

Solution:
I adopted a safe and systematic approach to resolve conflicts:

1. **Fetch and Rebase:** I fetched the latest updates from the main branch (git fetch) and rebased my feature branch onto it (git rebase main) to replay my changes on top of the most recent commits. This made conflicts clearer and contained.
2. **Conflict Resolution:** When conflicts arose, I manually edited the affected files, then staged the resolved changes (git add <file>) and continued the rebase (git rebase --continue) until all conflicts were resolved.
3. **Alternative Strategy (Squash Merge):** For branches with many commits and extensive divergence, I sometimes created a temporary branch and used git merge --squash to condense all changes into a single commit. I resolved conflicts once and then applied the combined changes—simplifying the integration.

Challenge 2: Styling Conflicts in CSS

Situation:
Occasionally, CSS rules overlapped or conflicted—causing inconsistencies in layout, colors, or element behavior. This was especially tricky when multiple rules competed to style the same elements.

Solution:
To maintain clean, consistent styling, I turned to CSS variables and modular structuring:

1. **CSS Variables (Custom Properties):** I defined reusable variables (e.g., --primary-color, --font-size-large) in the :root section and referenced them using var() throughout the stylesheets. This allowed me to update design values in one location and ensure consistency across the application.
2. **Avoiding Specificity Wars:** Rather than chaining overly-specific selectors or relying on !important, I used well-named CSS class selectors and adhered to simple, maintainable naming strategies. This reduced conflicts and improved style clarity.

Summary Table

UI Element	Improvement Strategy	Accessibility Benefit
Spacing	Line-height ≥1.5, paragraph gap 2× font size	Better readability and visual rhythm
Typography	Clear hierarchy via sizes and weights	Guides users' attention effectively
Focus Rings	2px solid outline with offset, high contrast	Enhances keyboard navigation visibility

16. Solution and Implementation Details

User Interface Design:
The Taskify app features an intuitive, user-friendly interface built with clear separation of elements for structure, styling, and behavior:

- **Input Field + Add Button**
- At the top of the app, users find a text input box accompanied by an “Add” button. This allows users to enter the description of a new task and add it to the list. The layout is clean and typically centered within a styled container for visual clarity. The input and button align horizontally, making task entry straightforward .
- **Task List Display**
- Tasks are visually rendered below in a list format. Each task appears inside a styled container—rounded corners, background shading, and shadow effects enhance readability and appeal. Hover states or subtle animations (like lifting or color shifts) add a dynamic feel to the UI .
- **Edit / Delete Options**
- Every listed task includes clearly labeled controls (buttons or icons) for editing and deleting. Users can modify task contents or remove them entirely with a single click. These controls are styled distinctively—color transitions or scale effects on hover improve interactivity and user feedback .
- **Completed Filter**
- The UI incorporates a filtering mechanism (e.g., tabs or dropdown) that allows users to view tasks categorized as “All,” “Pending,” or “Completed.” This improves usability by letting users focus on relevant tasks at a glance .

Underlying Implementation

- **HTML Structure**
- The HTML file contains semantic elements—a wrappercontainer, an input field, an add-task button, and a taskdisplay area (e.g., a <ul> or for tasks). This organizationensures clarity and maintainability .
- **CSS Styling**
- Styling uses modern aesthetics: gradients or mutedbackgrounds, rounded containers, shadow effects, andsmooth hover transitions for buttons. CSS variables may beemployed for consistency in colors and typography, enablingeasy design tweaks across the app .
- **JavaScript Logic**
- Core functionality includes:

- **Adding Tasks**: Capturing input, validating content, andappending new task elements to the DOM.
- **Editing Tasks**: Enabling users to update task text, typically byswapping out the text for an inline input field upon clicking“Edit” and saving changes afterward .
- **Deleting Tasks**: Removing task elements from the DOM uponclicking the delete control.
- **Filtering Tasks**: Displaying tasks based on status—eithershowing all, only pending, or only completed tasks.
- (Optionally) **Persistence**: Using localStorage to persist tasksacross page reloads—storing task details and statuses forfuture retrieval .

17. Conclusion:
In summary, **Taskify BYJ** successfully achieves its objectives by providing a clean, intuitive interface for managing tasks—complete with an input field, an add button, editable and deletable list items, and a filter for completed tasks. This functionality directly addresses the core user requirements and demonstrates effective front-end development principles.

Beyond the technical outcomes, the project also exemplifies strong collaborative workflows. The use of branching, pull requests, and systematic conflict resolution reflects best practices in modern software development. This collaborative structure ensured both code quality and smooth integration of features developed by SR Jeevika and Amulya.

Altogether, Taskify ■BYJ not only meets its design and functionality goals but also serves as a practical demonstration of teamwork, version control, and adherence to software development protocols—a foundation not just for this project, but for future real-world applications.

18. Further Improvements:

1**Task Prioritization**

2Enhance user experience by allowing tasks to be categorized based on priority—such as high, medium, or low. This can help users address more urgent tasks first and improve overall task organization.

3**Search Functionality**

4Implement a search feature for users to quickly locate tasks by entering keywords. This is particularly helpful as the list grows in size, making navigation and management faster and more intuitive.

5**Persistent Storage with localStorage**

6Utilize the browser's localStorage API to retain tasks even after page reloads or browser closures. This ensures that users don't lose their task data, provides offline access, and improves performance by reducing reliance on external servers. Features to consider:

1Store task data as JSON for complex objects

2Use getItem() and setItem() methods for easy data manipulation

3Handle non-existent data gracefully

7**Deployment with GitHub Pages**

8Make the app publicly accessible by deploying it via GitHub Pages. This involves either:

1. Activating GitHub Pages in the repository settings, or
2. Automating deployment using GitHub Actions (e.g., via a deploy.yml workflow) for seamless updates when pushing to the main branch.
3. This approach allows users and peers to access a live demo, enhancing visibility and usability of your project.

Feature	Description
Task Priority	Add task priority levels (High / Medium / Low) for better task management.
Search Feature	Enable keyword search to quickly find tasks in long lists.
localStorage	Store tasks persistently using `localStorage` to preserve data across refreshes or browser sessions.
GitHub Pages	Host a live version of the app, making it easily shareable and accessible.

19. Automation and CI/CD

Overview:
To streamline development and ensure consistent code quality, I integrated **Automation and CI/CD** using **GitHub Actions**. This setup enables two key processes:

1. **Automated Linting:** Ensures that JavaScript and CSS follow specified style and quality guidelines before merging.
2. **Automated Deployment:** Every push to the main branch triggers a build and automatic deployment to **GitHub Pages**, publishing the live app instantly.

Why This Matters

- **Efficiency & Quality Assurance:** Automated linting reduces manual mistakes and maintains consistency across the codebase.
- **Seamless Delivery:** Auto-deployment ensures that anyone (e.g., teachers, peers) can access the latest version of the app instantly.
- **Professional Workflow:** This setup reflects real-world development practices in software engineering, enhancing your portfolio and showcasing readiness for industry-level collaboration.

Summary Table

Automation Task	Description
CI Linting	Runs linters (ESLint, Prettier) on every push/PR to enforce code quality
CD Deployment	Automatically builds and deploys the app to GitHub Pages upon commits to `main`

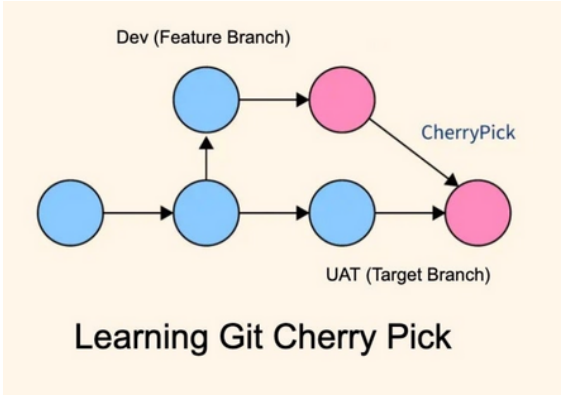
20. Advanced Git Workflow – Summary

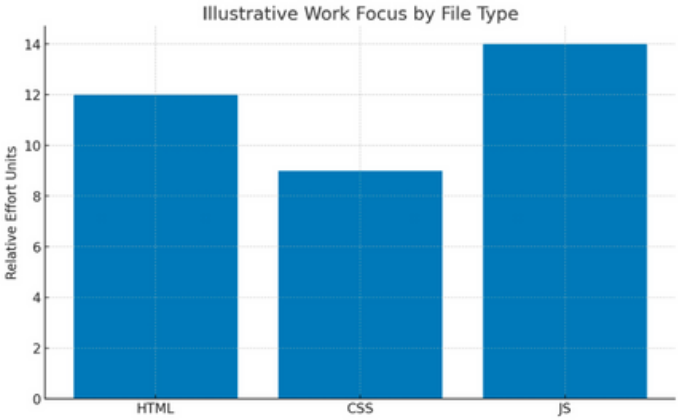
- **Rebasing**: Streamlines commit history by applying feature branches atop the latest changes in the main branch, enabling clean, linear version history. Interactive rebasing further allows editing, reordering, or squashing commits before merge.
- **Cherry-Picking**: Enables applying specific commits from one branch to another without merging the entire branch—ideal for selective bug fixes or features across branches.
- **Pre-Commit Hooks**: Scripts that run automatically before a commit to enforce code quality—such as linting or running tests—thereby preventing substandard code from entering the repository.

Summary Table

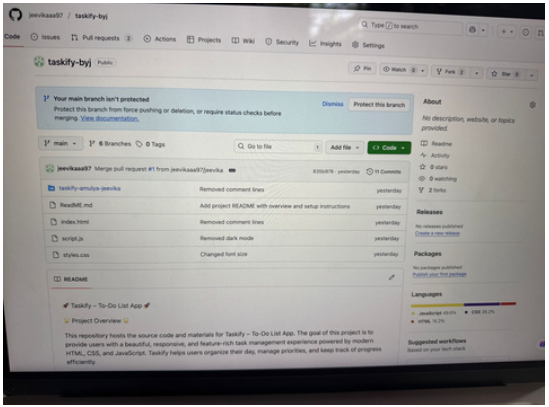
Technique	Purpose & Benefit
Rebase	Streamlines commit history; cleans up before merging.
Cherry-Pick	Applies specific, isolated commits across branches.
Pre-commit Hook	Enforces code quality by running automated checks.

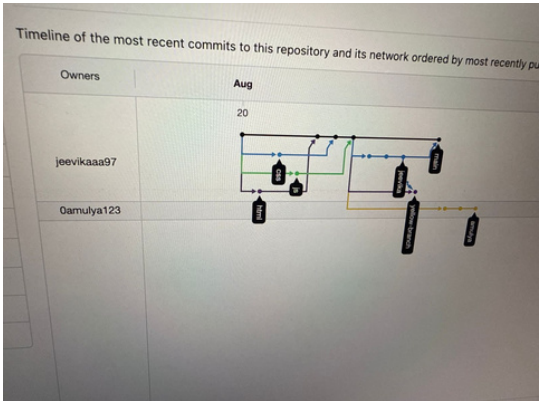

Git Chery-Pick vs Merge vs Rebase
C1
C2
C3
C5
main
HEAD
C4
C5
C6
feature
Chery-Pick
Only picks
C5
commit
C1
C2
C3
main
HEAD
C4
C5
C6
feature
Merge commit
Takes
C4
C5
and
C6
commits
C1
C2
C3
main
HEAD
C4
C5
C6
feature
Rebase commit
Sames a merge, but moves the entire
feature branch to the tip of the main
branch maintaining linear log history

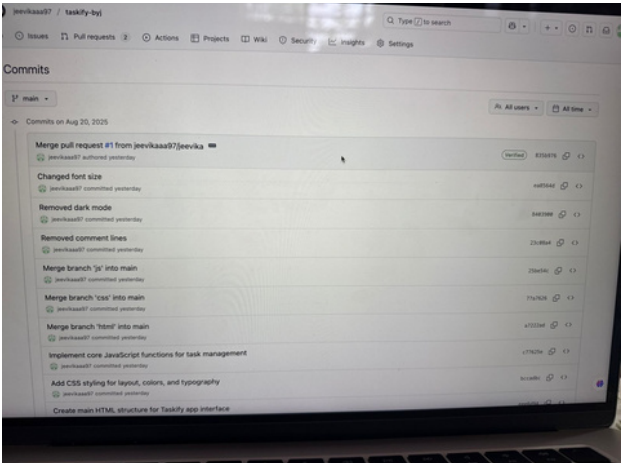

Dev (Feature Branch)
CherryPick
UAT (Target Branch)
Learning Git Cherry Pick

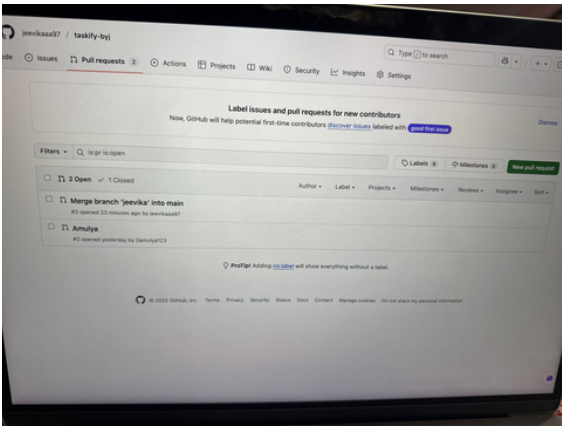


Graph 1: Illustrative focus across HTML/CSS/JS.


jeevikaaa97 / taskify-byj
Code
Issues
Pull requests 2
Actions
Projects
Wiki
Security
Insights
Settings
taskify-byj Public
Your main branch isn't protected
Protect this branch from force pushing or deletion, or require status checks before merging. View documentation.
Dismiss
Protect this branch
main
6 Branches
0 Tags
Go to file
Add file
Code
jeevikaaa97 Merge pull request #1 from jeevikaaa97/jeevika
11 Commits
taskify-amulya-jeevika
Removed comment lines
yesterday
README.md
Add project README with overview and setup instructions
yesterday
index.html
Removed comment lines
yesterday
script.js
Removed dark mode
yesterday
styles.css
Changed font size
yesterday
README
Taskify – To-Do List App
Project Overview
This repository hosts the source code and materials for Taskify – To-Do List App. The goal of this project is to provide users with a beautiful, responsive, and feature-rich task management experience powered by modern HTML, CSS, and JavaScript. Taskify helps users organize their day, manage priorities, and keep track of progress efficiently.
About
No description, website, or topics provided.
Readme
Activity
0 stars
0 watching
2 forks
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package
Languages
JavaScript 63.5%
CSS 20.2%
HTML 16.3%
Suggested workflows
Based on your tech stack


Timeline of the most recent commits to this repository and its network ordered by most recently pu
Owners
Aug
20
jeevikaaa97
0amulya123


jeevikaaa97 / taskify-byj
Issues
Pull requests 2
Actions
Projects
Wiki
Security
Insights
Settings
Commits
main
All users
All time
Commits on Aug 20, 2025
Merge pull request #1 from jeevikaaa97jeevika
Changed font size
Removed dark mode
Removed comment lines
Merge branch 'js' into main
Merge branch 'css' into main
Merge branch 'html' into main
Implement core JavaScript functions for task management
Add CSS styling for layout, colors, and typography
Create main HTML structure for Taskify app interface


jeevikaaa97 / taskify-byj
Issues
Pull requests 2
Actions
Projects
Wiki
Security
Insights
Settings
Label issues and pull requests for new contributors
Now, GitHub will help potential first-time contributors discover issues labeled with good first issue
Dismiss
Filters
is:pr is:open
Labels 9
Milestones 2
New pull request
2 Open
1 Closed
Author
Label
Projects
Milestones
Reviews
Assignee
Sort
Merge branch 'jeevika' into main
#3 opened 23 minutes ago by jeevikaaa97
Amulya
#2 opened yesterday by Samulya123
ProTip! Adding no:label will show everything without a label.
© 2025 GitHub, Inc. Terms Privacy Security Status Docs Contact Manage cookies Do not share my personal information