**GitHub Enterprise** is a version of GitHub designed for organizations that need a more secure, scalable, and customizable Git hosting solution. It provides the core features of GitHub, such as code repositories, pull requests, and collaboration tools, but with enhanced options to meet the needs of enterprises. GitHub Enterprise can be deployed on-premises or used in a cloud-based setup, allowing flexibility in how organizations manage their development environments.

**Key Features of GitHub Enterprise:**

1. **Self-Hosting or Cloud Hosting**:
   * **GitHub Enterprise Server**: The on-premises version of GitHub Enterprise, which organizations can host on their own infrastructure or private cloud.
   * **GitHub Enterprise Cloud**: The cloud-hosted version, which is fully managed by GitHub and hosted on GitHub’s infrastructure (using the same architecture as GitHub.com).
2. **Security**:
   * **SAML Authentication & Single Sign-On (SSO)**: GitHub Enterprise integrates with enterprise identity providers for single sign-on (SSO), allowing easier management of user access across systems.
   * **Advanced Security Features**: Includes two-factor authentication (2FA), security advisories, and audit logs for compliance purposes.
   * **Branch Protection Rules**: Enforces rules on branches, such as requiring reviews or CI checks before merging.
   * **Security Alerts and Vulnerability Scanning**: Automated alerts for vulnerable dependencies within repositories.
3. **Scalability and Performance**:
   * GitHub Enterprise is optimized for large organizations with extensive teams and codebases.
   * High availability setups and disaster recovery options are available to ensure reliability in enterprise environments.
4. **Customization**:
   * **Webhooks & API Integrations**: GitHub Enterprise supports custom integrations using webhooks, GitHub Apps, and its REST API, enabling organizations to tailor the platform to their workflows.
   * **Custom Branding and User Interface**: Allows enterprises to customize the look and feel of their GitHub instance to match their corporate identity.
5. **Collaboration and Code Management**:
   * **GitHub Actions**: Integrated CI/CD pipelines directly into the GitHub workflow.
   * **Pull Requests, Issues, Discussions**: Facilitates collaboration with built-in tools for code reviews, bug tracking, and feature discussions.
   * **Code Scanning**: Analyzes code for security vulnerabilities directly within the repository.
6. **Compliance and Auditing**:
   * **Audit Logs**: Detailed logs of user actions, repository changes, and system access to meet compliance needs.
   * **Role-Based Access Control (RBAC)**: Granular access control to ensure the right users have the appropriate permissions for various repositories and actions.
7. **Enterprise Support**:
   * GitHub Enterprise comes with enterprise-grade support from GitHub’s team, including options for 24/7 support, training, and more.
   * **Managed Services**: For users who prefer a fully managed experience, GitHub offers managed services to help with setup, maintenance, and troubleshooting.

**Deployment Options:**

* **GitHub Enterprise Server**: Installed on-premises or within a private cloud. The organization has full control over hardware, networking, and maintenance.
* **GitHub Enterprise Cloud**: A fully managed, cloud-hosted solution where GitHub handles scaling, infrastructure, and updates.

**Use Cases for GitHub Enterprise:**

1. **Large-Scale Organizations**: Enterprises with hundreds or thousands of developers who need a centralized platform for managing code, collaboration, and CI/CD.
2. **Regulatory Compliance**: Companies in industries like finance, healthcare, or government that require strict security, privacy, and audit capabilities.
3. **Collaboration at Scale**: Teams that work on large projects across multiple locations and require advanced version control, security, and integration capabilities.
4. **Custom Workflows**: Organizations needing tailored GitHub features, integrations, and configurations specific to their development environment.

**Pricing:**

GitHub Enterprise has a subscription-based pricing model that depends on the number of users and whether the organization opts for cloud or self-hosted solutions. Pricing for GitHub Enterprise Server generally involves a licensing fee, along with an option for paid support services, while GitHub Enterprise Cloud typically involves a per-user or per-seat fee.

**Conclusion:**

GitHub Enterprise is designed to provide a seamless and secure Git experience for large organizations with complex needs. Whether you're managing a large-scale project with extensive security and compliance requirements or simply looking for advanced collaboration tools, GitHub Enterprise offers a robust and flexible solution.

**🚀 Key Differences: GitHub Enterprise vs GitHub.com**

| **Feature** | **GitHub.com (Free/Pro/Team)** | **GitHub Enterprise** |
| --- | --- | --- |
| **Authentication** | Standard OAuth, personal access tokens, and 2FA. | **SAML single sign-on (SSO)**, **SCIM provisioning**, and **Enterprise Managed Users (EMU)**. |
| **Access Control** | Limited to repository-level permissions. | **Organization-wide policies**, **IP allow lists**, and **role-based access controls (RBAC)**. |
| **Compliance** | Basic security features for public repos. | **Audit logs**, **SOC 2**, **GDPR**, **HIPAA** compliance, and **data residency controls**. |
| **Repository Management** | Basic settings for branches, merges, and webhooks. | **Required reviews**, **protected branches**, and **custom repository roles**. |
| **Deployment Controls** | Basic CI/CD via GitHub Actions. | Enhanced deployment security with **environment protection rules**, **deploy keys**, and **secrets scanning**. |
| **Administration** | Limited org-level settings. | **Enterprise Account**, centralized billing, and **Admin Center** with advanced user management. |
| **Support** | Community support and basic help. | **24/7 support**, **SLA guarantees**, and dedicated account managers. |
| **Integration** | Limited integrations. | **Webhooks**, REST/GraphQL APIs, and advanced integrations with Jira, Azure, and more. |
| **Security** | Dependabot alerts and basic security features. | **Advanced Security** with **code scanning**, **secret scanning**, and **dependency reviews**. |
| **Backup & Disaster Recovery** | Limited data recovery options. | Automated backups, **business continuity solutions**, and **high availability**. |
| **Compliance & Auditing** | Basic audit history. | **Audit logs**, **compliance reports**, and **custom security policies**. |

INITIALIZING UR DIRECTORY :

git config --global user.email “raman@example.com”

git config --global user.name “raman.khanna”

286 mkdir raman

287 ls

288 cd raman/

289 ls

290 git status

291 git init

292 git status

ADDING A FILE :

ls -la

295 cd .git/

296 ls

297 cd ..

298 ls

299 pwd

300 touch file1

301 ls

302 git status

303 git add .

304 git status

COMMITING TO REPO :

git status

308 git commit -m "first file commited"

309 git status

310 git log

vi file1

317 ls

318 cat file1

319 git status

320 git add file1

321 git status

322 git commit -m " added content in file1"

323 git status

git log

REMOTE REPO :

86 git remote add origin https://github.com/ramannkhanna2/myrepo.git

87 git push origin master

88 git status

--add a file on github manually and try to pull it

331 git diff 4950ad7174b02ffd8cb158ecd0b12bc327e61890 526591fe5cc497fbbef2e19c70f775539190df64

341 git pull origin master

342 ls

343 cat file-remote

git clone https://github.com/admingagan/test.git

**What is a Branch?**

In GitHub (and Git in general), a **branch** is like a separate "workspace" where you can work on a specific task or feature without affecting the main code. Imagine you're writing a book, and you want to try a new chapter idea. Instead of messing up the original manuscript, you make a copy of the page and write the new chapter there. If you like it, you can add it back to the original manuscript later.

* **Main Branch**: The main or default branch in Git is often called main (or master). This is like your book’s main content that everyone is reading.
* **Feature Branch**: A branch you create for working on new features, bug fixes, or experiments.

**How to Create a Branch?**

You create a branch when you want to work on something new, like adding a new feature or fixing a bug.

1. **Using GitHub Web Interface:**
   * Go to the repository (the project) you’re working on.
   * In the top-left corner, click the dropdown that says **Branch: main**.
   * Type the name of your new branch (e.g., feature/add-login-page).
   * Press **Enter**, and your new branch is ready!
2. **Using Git on Your Computer (Git CLI):**
   * If you’re working with Git on your computer, run this command:

bash

CopyEdit

git checkout -b feature/add-login-page

* + This creates and switches to your new branch.

BRANCHING :

git branch

348 git branch 2ndbranch

349 git branch

350 ls

351 cat file1

352 git status

353 git switch 2ndbranch

354 git --help

355 clear

356 git checkout 2ndbranch

357 git branch

358 ls

359 cat file

360 cat file1

361 vi file1

362 cat file1

363 git status

364 git add .

365 git status

366 git commit -m "changed file1 contents on 2ndbranch to test"

367 ls

368 git status

369 git checkout master

370 ls

371 cat file1

372 git checkout 2ndbranch

373 cat file1

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

DAY-2

MERGING :

git checkout 2ndbranch

373 cat file1

374 ls

375 history

376 git branch

377 touch file3

vi file3 : "my name is raman"

378 ls

379 git checkout master

380 ls

381 git checkout 2ndbranch

382 ls

383 vi file3 "my name is raman khanna "

384 ls

385 git branch

386 git add .

387 git status

388 git commit -m "added file3 in testing branch"

389 ls

393 git checkout 2ndbranch

394 git checkout master

395 git merge 2ndbranch

396 ls

397 cat file3 u will see : "my name is raman khanna " after merge

382 ls

383 vi file3

384 ls

385 git branch

386 git add .

387 git status

388 git commit -m "added file3 in testing branch"

389 ls

391 git checkout master

392 ls

395 git merge 2ndbranch

396 ls

397 cat file3

WORKING ON REMOTE REPO (PUSH/PULL/BRANCH CREATION) :

git checkout 2ndbranch

398 history

399 clear

400 git status

401 ls

402 clear

403 ls

404 git status

405 git push origin main

406 git status

407 git log

408 clear

409 git checkout 2ndbranch

410 ls

411 rm -rf file1 file3 file-remote

412 ls

413 git status

433 git add --all

434 git status

435 git commit -m " 2ndbranch files deleted "

436 git status

437 ls

441 vi file4-2ndbranch

443 git status

444 git add .

445 git commit -m " added4th file in 2ndbranch"

446 git status

447 git log

448 git push origin 2ndbranch

--- add a file in 2nd branch manually on github and pull it from there to cli :

git pull origin 2ndbranch

454 ls

455 cat file5-2ndbranch

\*\*\*\*

if u have pushed something to remote repo , and then again trying to pull from that particular

remote repo ,, it wll give a head error : ise below before pulling ::

git fetch --all

git reset --hard origin/master

or if you want a branch

git checkout your\_branch\_name\_here

git reset --hard origin/your\_branch\_name\_here

git branch -d 2ndbranch

487 git checkout master

488 git branch -d 2ndbranch

489 git branch -D 2ndbranch

to delete remote branch :

git push --delete origin 2ndbranch

**Hostname and Subdomain isolation** are ways to manage and protect your GitHub instance by customizing how it behaves with the internet domain. Here's what they mean in simpler terms:

**1. Hostname Isolation**

* This refers to the ability to assign a **unique domain name** (hostname) to your GitHub Enterprise Cloud instance.
* For example, instead of accessing your GitHub instance through a generic domain like github.com, you can access it through a custom hostname like github.yourcompany.com.
* **Why it's useful**: This makes your instance feel like it's entirely separate from GitHub's general cloud platform. It adds a layer of security and branding, ensuring that the URLs for your internal resources look distinct and professional.

**2. Subdomain Isolation**

* Subdomain isolation lets you divide your GitHub Enterprise Cloud into multiple **subdomains**, each with its own access and settings.
* For example, you could have dev.github.yourcompany.com for your development team, ops.github.yourcompany.com for the operations team, and so on.
* **Why it's useful**: It makes it easier to manage different parts of your organization or different environments (like dev, staging, and production) under the same GitHub Enterprise Cloud instance. Each subdomain can be tailored for specific use cases or departments while still using the same central GitHub platform.

**How it helps:**

* **Isolation** gives better control over traffic, security, and user access. You can limit access to specific teams or environments based on the subdomain they use.
* **Flexibility** allows you to scale and customize the GitHub instance without worrying about overlapping or conflicting settings.

In short, hostname and subdomain isolation help in managing and securing your organization's GitHub environment by creating separate, customizable spaces for different teams or use cases, all under one GitHub Enterprise Cloud instance.

**. Change the Domain (Hostname)**

* Go to GitHub Enterprise settings.
* In the **"Enterprise settings"**, navigate to **"Domains"**.
* You will see the option to **add a custom domain**.
* Update the domain settings with your new custom domain (e.g., github.yourcompany.com).
* Verify the domain using DNS records as instructed by GitHub.

**2. Set Up Subdomain Isolation**

* Once the custom domain is configured, you can create **subdomains** (e.g., dev.github.yourcompany.com, staging.github.yourcompany.com).
* Go to your DNS provider and set up **CNAME records** for each subdomain to point to GitHub's servers.
* Ensure each subdomain is properly configured in GitHub Enterprise to separate the environments or teams.

**Lab: Demonstrating SSH Access and Command Line Utilities on GitHub Enterprise Cloud**

This lab is designed to provide a hands-on experience in configuring SSH access and using Git command-line utilities on GitHub Enterprise Cloud (GHEC). By the end of the lab, you will understand how to securely access repositories and perform basic Git operations via the terminal.

**Prerequisites:**

* A **GitHub Enterprise Cloud** account.
* Basic knowledge of Git and command-line operations.
* Git and OpenSSH installed on your local machine.

**Lab Objectives:**

1. **Set up SSH access to GitHub Enterprise Cloud**.
2. **Configure Git with user credentials**.
3. **Clone a repository using SSH**.
4. **Perform basic Git operations**:
   * Add changes.
   * Commit changes.
   * Push changes.
   * Pull updates.
5. **Troubleshoot common SSH access issues**.

**Lab Environment Setup**

**Step 1: Installing Git and OpenSSH (if not already installed)**

**Linux** (Ubuntu/Debian-based):

bash

CopyEdit

sudo apt update

sudo apt install git

**Windows**:

* Install Git from [Git for Windows](https://gitforwindows.org/).
* Install OpenSSH from the [Windows Features](https://docs.microsoft.com/en-us/windows-server/administration/openssh/openssh_install_firstuse) if not already installed.

**Step 2: Create or Use an Existing GitHub Enterprise Cloud Account**

* Ensure you have a GitHub Enterprise Cloud account with access to a repository. If not, create one or request access to a repository.

**Part 1: Setting Up SSH Access**

**Step 3: Generate an SSH Key Pair**

1. Open your terminal.
2. Run the following command to generate an SSH key pair:

bash

CopyEdit

ssh-keygen -t rsa -b 4096 -C "your\_email@example.com"

Replace "your\_email@example.com" with the email address associated with your GitHub Enterprise Cloud account.

1. When prompted for the file to save the key, press **Enter** to accept the default location (~/.ssh/id\_rsa).
2. Optionally, set a passphrase for additional security. You can leave it blank for no passphrase.

**Step 5: Add SSH Key to GitHub Enterprise Cloud**

1. Copy the contents of your public key (id\_rsa.pub):

bash

CopyEdit

cat ~/.ssh/id\_rsa.pub

Copy the entire output starting with ssh-rsa.

1. Log in to your GitHub Enterprise Cloud account and navigate to **Settings** > **SSH and GPG keys**.
2. Click **New SSH key**, paste the copied key, and give it a descriptive title (e.g., "Work Laptop").
3. Click **Add SSH key**.

**Part 2: Configuring Git**

**Step 6: Set Up Global Git Configuration**

1. Set up your Git name and email (use the same email as in GitHub):

bash

CopyEdit

git config --global user.name "Your Name"

git config --global user.email "your\_email@example.com"

1. You can verify the configuration by running:

bash

CopyEdit

git config --list

**Part 3: Cloning a Repository Using SSH**

**Step 7: Clone a GitHub Repository**

1. Navigate to your GitHub Enterprise Cloud repository page.
2. Click the **Code** button and select **SSH** to get the SSH URL.
3. In the terminal, run:

bash

CopyEdit

git clone git@github.com:username/repository.git

Replace username/repository.git with the actual repository URL.

1. Change into the cloned directory:

bash

CopyEdit

cd repository

GitHub Pages is a feature provided by GitHub that allows you to host static websites directly from a repository. These sites are usually HTML, CSS, and JavaScript files, and they can be used for things like personal blogs, project documentation, portfolios, and more.

**GitHub Pages in GitHub Enterprise**

When it comes to **GitHub Enterprise**, you can use GitHub Pages in a similar way as you would on GitHub.com. The main difference is that GitHub Enterprise is a private, on-premise solution that is often used by organizations, so the hosting environment is private and customizable based on the organization's needs.

Here’s how GitHub Pages works in the context of **GitHub Enterprise**:

1. **Repository-based Hosting**:
   * You create a repository (public or private) on GitHub Enterprise.
   * In that repository, you can add files for a static website (HTML, CSS, JavaScript, images, etc.).
2. **GitHub Pages Settings**:
   * You can enable GitHub Pages for a repository by going to the **Settings** of the repository.
   * Under the **Pages** section, you can select which branch (usually main or gh-pages) to serve the website from. You can also configure which folder (like /docs) to use.
3. **Custom Domain**:
   * If your organization or project has a custom domain, you can configure the website to be accessible via that domain instead of using the default GitHub Pages domain (username.github.io or orgname.github.io).
4. **Access Control**:
   * Since GitHub Enterprise is for private repositories within an organization, the GitHub Pages site can be public or restricted to specific teams within the organization depending on the repository’s visibility (public or private).
   * For private repositories, GitHub Enterprise can also ensure that the pages are only accessible by authenticated users within the organization, though public-facing websites can be made entirely accessible to anyone.
5. **Security & Compliance**:
   * GitHub Enterprise allows organizations to have better control over security and compliance settings for their GitHub Pages sites, such as custom SSL certificates, access control policies, and ensuring that content is being served according to internal security guidelines.

**Key Points:**

* **GitHub Pages** lets you host static websites directly from your GitHub repositories.
* In **GitHub Enterprise**, this feature works the same way, but it's hosted within your organization’s private infrastructure.
* It’s often used for documentation, blogs, or simple websites.
* You can configure custom domains and control access permissions based on repository settings.

**========================================**

**DAY-3**

**API Overview: REST API vs. GraphQL**

APIs (Application Programming Interfaces) allow different software applications to communicate with each other. Two popular types of APIs are **REST API** and **GraphQL**. Let's dive into what they are, how they work, and use real-world examples to explain them.

**1. REST API (Representational State Transfer)**

**REST** is an architectural style for designing networked applications. It uses HTTP requests to perform CRUD operations (Create, Read, Update, Delete). REST APIs are based on a client-server model where the client requests data from the server, and the server responds with the requested data.

**Key Features of REST:**

* **HTTP Methods**: Uses standard HTTP methods like GET, POST, PUT, DELETE, etc.
* **Resources**: The data is represented as resources (e.g., users, products, etc.).
* **Stateless**: Each API request from the client contains all the information the server needs, so no session is stored on the server.
* **Cacheable**: Responses can be cached for efficiency.

**Real-world Example of REST API:**

Imagine you're using an online store website that shows product details. Here's how REST would work:

* **GET** /products: Fetches a list of all products.
* **GET** /products/123: Fetches details for the product with ID 123.
* **POST** /products: Creates a new product.
* **PUT** /products/123: Updates the details of product 123.
* **DELETE** /products/123: Deletes product 123.

Each URL (like /products or /products/123) represents a "resource," and the HTTP methods represent the actions you can perform on them.

**2. GraphQL API**

**GraphQL** is a query language for APIs developed by Facebook. It allows clients to request exactly the data they need, and nothing more. GraphQL APIs are more flexible and efficient compared to REST APIs.

**Key Features of GraphQL:**

* **Single Endpoint**: Unlike REST, which has multiple endpoints for different resources, GraphQL uses a single endpoint (e.g., /graphql).
* **Client-Specified Queries**: Clients can specify exactly what data they need in a query. This avoids over-fetching or under-fetching data.
* **Real-Time Updates**: With GraphQL, you can also subscribe to real-time updates on data changes (using subscriptions).

**Real-world Example of GraphQL:**

Imagine you're using a mobile app that shows information about a user, their posts, and comments on those posts. In a REST API, you might need multiple requests to get this data:

* **GET** /users/123: Fetches user details.
* **GET** /users/123/posts: Fetches posts made by user 123.
* **GET** /posts/456/comments: Fetches comments for post 456.

In GraphQL, you can send a single request to get all the information in one go:

graphql

CopyEdit

{

user(id: 123) {

name

email

posts {

title

comments {

text

}

}

}

}

This query requests:

* The name and email of the user with ID 123.
* The title of each of their posts.
* The text of each comment on those posts.

In a REST API, you would need separate requests for each of those pieces of data. With GraphQL, you can specify exactly what you need in one query.

**Key Differences Between REST and GraphQL:**

| **Feature** | **REST API** | **GraphQL API** |
| --- | --- | --- |
| **Data Fetching** | Fetches predefined data from fixed endpoints. | Fetches custom data based on queries. |
| **Multiple Endpoints** | Multiple endpoints for different resources. | Single endpoint for all operations. |
| **Over-fetching** | Can fetch unnecessary data (e.g., extra fields). | Avoids over-fetching by fetching only required data. |
| **Flexibility** | Less flexible, needs multiple requests. | More flexible, allows specifying exact data needed. |
| **Performance** | Can be inefficient (multiple requests for related data). | More efficient (single request for related data). |
| **Real-time Updates** | Not built-in, can be done via other protocols (e.g., WebSockets). | Can handle real-time updates with subscriptions. |

**When to Use REST API vs. GraphQL?**

* **Use REST** when:
  + You have fixed, predefined resources.
  + Your API clients don't need complex queries.
  + Simplicity and standardization are more important than flexibility.
* **Use GraphQL** when:
  + You need more flexible data retrieval.
  + Clients need to request specific data to avoid over-fetching.
  + You have complex relationships between data (e.g., a user, their posts, and comments).

**Conclusion**

* **REST** is a traditional, simple approach to APIs, great for straightforward use cases.
* **GraphQL** is a more modern, flexible alternative, suited for complex applications where precise control over the data is needed.

**Lab: API Testing with Postman for GitHub REST API**

This lab will guide you through the steps to perform API testing on GitHub's REST API using **Postman**. By the end of this lab, you'll be able to send requests to GitHub's API, validate responses, and automate API testing using Postman.

**Lab Overview**

GitHub provides a RESTful API that allows users to interact with their repositories, issues, pull requests, commits, and other resources available in GitHub. For this lab, we'll focus on the following:

1. **Fetching user information.**
2. **Listing repositories.**
3. **Creating an issue.**

You'll need a GitHub account and a personal access token to authenticate your requests.

**Prerequisites**

* **Postman Installed**: Ensure Postman is installed on your machine. If not, download it from here.
* **GitHub Account**: You need a GitHub account to generate an access token.
* **Personal Access Token**: GitHub's API requires authentication for most operations, so you will need a personal access token for authentication.

**Step 1: Generate GitHub Personal Access Token**

1. Log in to your **GitHub** account.
2. Go to **Settings** > **Developer settings** > **Personal access tokens**.
3. Click on **Generate new token**.
4. Choose the necessary scopes for the token. For this lab, you can use the following basic permissions:
   * repo (full control of private repositories)
   * user (access user information)
   * public\_repo (access public repositories)
5. Click **Generate token**.
6. **Copy the generated token** as you will use it for authentication in Postman.

**Step 2: Set Up Postman for API Testing**

**2.1 Create a New Request in Postman**

1. **Open Postman** and create a new request:
   * Click on **New** > **Request**.
   * Name the request (e.g., "GitHub User Information").
   * Add it to a collection or create a new collection for organizing your requests.
   * Create a blank collection named “GithubApiRequest” and add the requests inside it whatever u wl create

**2.2 Set Up Authentication**

1. For most requests to GitHub's REST API, you'll need authentication using your GitHub personal access token.
2. Edit the collection “GithubApiRequest”
3. Go to the **Authorization** tab in Postman.
4. Select **Bearer Token** as the type.
5. Paste the GitHub personal access token in the **Token** field.
6. and ur bearer token inside it.
7. Save it
8. and ur bearer token inside it.
9. Save it

**GITHUB API ENDPOINT PAGE :**

<https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#list-repositories-for-a-user>

**Step 3: Test GitHub API Endpoints**

**-- Create a get request under collection “GithubApirequest”**

**-- in the url >> add >>** [**https://api.github.com/users/USERNAME/repos**](https://api.github.com/users/USERNAME/repos)

**Replace username with ramannkhanna2**

* + - * **In authorization keep parent authorization**
  + **Send the api request and get the response**
  + **Test with curl as well :**

**curl -L \ -H "Accept: application/vnd.github+json" \ -H "Authorization: Bearer <YOUR-TOKEN>" \ -H "X-GitHub-Api-Version: 2022-11-28" \ https://api.github.com/users/USERNAME/repos**

**3.1 Get User Information**

GitHub's API allows you to fetch user information. Let's start with that:

* **Endpoint**: https://api.github.com/user
* **Method**: GET
* **Description**: Fetches the authenticated user's details.

1. In Postman, enter the following URL:

arduino

CopyEdit

https://api.github.com/user

1. Select the **GET** method from the dropdown next to the URL.
2. Click **Send**.
3. **Check the response**:
   * You should receive a **200 OK** response with user details (name, username, email, etc.).

**Validate the Response:**

* Status Code: 200 OK
* Check the body of the response for fields like login, id, name, etc.

**3.3 Create a Repository for the authenticated user :**

**https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#create-a-repository-for-the-authenticated-user**

Now, let's try creating GitHub repository using the API. We’ll need to use the **POST** method to create a new issue.

* **Endpoint**: https://api.github.com/user/repos
* **Method**: POST
* **Description**: Create a new issue in a specific repository.

1. Go to the **Body** tab in Postman and select **raw**. Choose **JSON** from the dropdown.
2. Add the following JSON body:

{

"name": "ramanrepofrompostman",

"description": "This is your first repo!",

"private": false

}

1. Click **Send**.
2. **Check the response**:

**Validate the Response:**

**3.3 Now delete the Repository for the authenticated user :**

**-- https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#delete-a-repository**

<https://api.github.com/repos/OWNER/REPO>

* + - * Replace owner and repo in above url like below :
    - <https://api.github.com/repos/ramannkhanna2/ramanrepofrompostman>

1. You must get a 204 response

**Integrating Tests, Deployment, and Notifications with GitHub Actions and Webhooks**

This unified process involves:

* **Running tests and reporting results** using the GitHub Status API.
* **Deploying the application** automatically with GitHub Actions.
* **Notifying stakeholders** using Slack Webhooks based on the deployment status.

Here’s how all of the above components come together in one cohesive GitHub Actions workflow:

name: CI/CD Pipeline with Deployment and Notifications

on:

push:

branches:

- main

pull\_request:

branches:

- main

jobs:

test:

runs-on: ubuntu-latest

steps:

- name: Checkout the repository

uses: actions/checkout@v2

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.9'

- name: Install dependencies

run: |

pip install -r requirements.txt

- name: Run tests

run: |

pytest tests/

- name: Report Test Results to GitHub Status API

if: success()

run: |

curl -X POST \

-H "Authorization: token ${{ secrets.GITHUB\_TOKEN }}" \

-d '{"state": "success", "description": "Tests passed successfully."}' \

https://api.github.com/repos/${{ github.repository }}/statuses/${{ github.sha }}

- name: Report Test Failure to GitHub Status API

if: failure()

run: |

curl -X POST \

-H "Authorization: token ${{ secrets.GITHUB\_TOKEN }}" \

-d '{"state": "failure", "description": "Tests failed."}' \

https://api.github.com/repos/${{ github.repository }}/statuses/${{ github.sha }}

deploy:

runs-on: ubuntu-latest

needs: test

steps:

- name: Checkout the repository

uses: actions/checkout@v2

- name: Set up AWS CLI

uses: aws-actions/configure-aws-credentials@v1

with:

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-region: 'us-west-2'

- name: Deploy to AWS EC2

run: |

scp -i ${{ secrets.EC2\_SSH\_PRIVATE\_KEY }} ./myapp.zip ubuntu@${{ secrets.EC2\_PUBLIC\_IP }}:/home/ubuntu/

ssh -i ${{ secrets.EC2\_SSH\_PRIVATE\_KEY }} ubuntu@${{ secrets.EC2\_PUBLIC\_IP }} 'unzip /home/ubuntu/myapp.zip -d /var/www/myapp'

- name: Report Deployment Status to GitHub

run: |

curl -X POST \

-H "Authorization: token ${{ secrets.GITHUB\_TOKEN }}" \

-d '{"state": "success", "description": "Deployment to AWS successful."}' \

https://api.github.com/repos/${{ github.repository }}/statuses/${{ github.sha }}

- name: Notify on Slack (Deployment Success)

run: |

curl -X POST -H 'Content-type: application/json' \

--data '{"text": "Deployment Successful! The tests passed and the app has been deployed to AWS EC2 successfully."}' \

https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX

**Workflow Breakdown:**

1. **Test Job**:
   * Runs tests on each push or pull request.
   * Updates GitHub commit status based on test results.
2. **Deploy Job**:
   * Deploys the application only if the tests pass.
   * After deployment, it updates the commit status in GitHub to reflect whether the deployment was successful.
3. **Slack Notification**:
   * Sends a message to Slack with the outcome of the deployment (success or failure).

**Organizations and Teams in GitHub Enterprise Cloud**

In **GitHub Enterprise Cloud (GHEC)**, **organizations** and **teams** are key structures used to manage groups of users, control access to repositories, and apply security policies efficiently.

**1. Organizations in GHEC**

An **organization** in GitHub Enterprise Cloud is a shared account that allows multiple users to collaborate across multiple repositories. Organizations are designed for businesses, open-source projects, or large-scale teams requiring centralized management.

**Key Features of Organizations in GHEC**

* **Centralized Management**: Admins control permissions, security policies, and billing.
* **Security & Compliance**: Supports advanced security features such as **SAML SSO**, **SCIM provisioning**, and **audit logs**.
* **Teams & Roles**: Users are organized into teams with defined roles.
* **Repository Management**: Efficient management of repositories with granular access control.
* **Billing Consolidation**: All organization members share a single billing plan.

**Organization Roles**

* **Owner**: Has full administrative access, can manage settings, billing, security, and repositories.
* **Member**: Has limited permissions based on team roles and repository access.
* **Billing Manager**: Can manage billing without access to repositories or code.
* **Security Manager** (Optional Role): Can manage security alerts, secrets scanning, and vulnerability details without full admin access.

**2. Teams in GHEC**

Teams are groups of organization members that simplify user management and permission control. They are hierarchical and support nested structures to align with organizational workflows.

**Key Features of Teams**

* **Hierarchical Structure**: Supports parent-child relationships (nested teams).
* **Role-based Access Control (RBAC)**: Assign permissions at the team level to streamline repository access.
* **Mentions & Notifications**: Teams can be mentioned directly (@team-name) for faster collaboration.
* **Team Discussions**: Dedicated discussion spaces for better communication.
* **Automatic Access Control**: Grant team-wide permissions to multiple repositories.

**Team Roles**

* **Maintainer**: Full administrative rights over the team (add/remove members, manage discussions, etc.).
* **Member**: Has access based on team permissions but cannot manage the team structure.

**3. Managing Organizations in GHEC**

**a) Creating an Organization**

1. Navigate to [GitHub](https://github.com).
2. In the top-right corner, click your profile photo and select **"Your organizations"**.
3. Click **New Organization**.
4. Select your plan (Enterprise or Free/Team plan).
5. Provide organization details, such as:
   * Organization name
   * Billing email
6. Follow the instructions to configure security settings, SAML SSO (if required), and invite users.

**b) Managing Organization Members**

* Navigate to your organization’s page.
* Go to **People** > **Invite Member**.
* Assign roles (e.g., **Owner**, **Member**, or **Security Manager**) and add the user to specific teams.

**Bulk Invitation:** For large teams, GitHub supports bulk invitations via CSV import.

**4. Managing Teams in GHEC**

**a) Creating a Team**

1. Navigate to your organization's **Teams** tab.
2. Click **New Team**.
3. Enter:
   * **Team Name** (e.g., devops-team)
   * **Description**
   * **Parent Team** (if applicable)
4. Configure repository permissions:
   * **Read** (View access)
   * **Write** (Contribute code)
   * **Admin** (Full control over repository settings)
5. Click **Create Team**.

**b) Adding Members to a Team**

1. Navigate to the created team.
2. Click **Add Member**.
3. Enter the usernames of the members.
4. Assign roles like **Maintainer** or **Member**.

**c) Managing Team Permissions**

Permissions can be applied at two levels:

* **Directly on Repositories**: Assign granular permissions per repository.
* **Inherited via Parent Teams**: Child teams inherit permissions from their parent teams.

| **Permission Level** | **Description** |
| --- | --- |
| **Read** | View code and discussions. |
| **Triage** | Manage issues and pull requests. |
| **Write** | Push commits and manage branches. |
| **Maintain** | Manage repository settings. |
| **Admin** | Full administrative control. |

**5. Best Practices for Managing Organizations and Teams in GHEC**

1. **Use Nested Teams** for large organizations to simplify role management.
2. **Enforce SAML SSO** for secure authentication.
3. **Enable Audit Logs** to track security events and ensure accountability.
4. **Apply IP Allowlisting** to restrict access to trusted networks only.
5. **Define Repository Rulesets** for branch protection, required reviews, and security protocols.
6. **Utilize CODEOWNERS** files to assign reviewers automatically based on code areas.

**6. Common Use Cases in GHEC**

✅ **Product Teams** — Create a dedicated team for each product line with write access to relevant repositories.  
✅ **Security Teams** — Assign the **Security Manager** role to handle vulnerability alerts and code scanning results.  
✅ **Outsourcing & Consultants** — Use granular repository permissions or deploy keys for limited code access.  
✅ **Compliance Teams** — Enable audit log exports to track policy adherence and user activities.

**Lab : Invite a user to organization and make them a member**

* **Login with the user and check if ur able to see that user In the organization**
* **than go to organizational settings to define organizational role assignment to that user what permission u want to give it (read permissions)**
* **go and check the permission if they are applied or not**
* **than go and give write permission to that user and confirm if now its able to to those operations**

**Step 2: Configuring Organization Settings**

* Navigate to your organization’s page via:
  + **https://github.com/organizations/<your-org>/settings**
* Under **Security** settings:  
  ✅ Enable **Two-Factor Authentication (2FA)** requirement for added security.  
  ✅ Enable **IP Allowlisting** (Optional for demo) to restrict access to specific IP addresses.

**Step 3: Creating Teams in the Organization**

1. Navigate to your organization's page.
2. Click the **Teams** tab → Select **New Team**.
3. Create the following teams:

| **Team Name** | **Description** | **Parent Team (Optional)** |
| --- | --- | --- |
| dev-team | Developers working on core projects | — |
| qa-team | Quality assurance team | — |
| backend-team | Backend developers | dev-team |
| frontend-team | Frontend developers | dev-team |

1. Assign **Team Maintainers** to manage each team.
2. Add members to each team to demonstrate team-wide access control.

**Step 4: Creating Repositories for Teams**

1. Navigate to the **Repositories** tab in your organization.
2. Create the following repositories:

| **Repository Name** | **Description** | **Visibility** |
| --- | --- | --- |
| project-alpha | Main project repository | Public/Private |
| project-beta | Secondary project repository | Private |
| internal-docs | Internal documentation | Private |

1. Assign permissions to teams as follows:

| **Repository** | **Team** | **Permission Level** |
| --- | --- | --- |
| project-alpha | dev-team | **Write** |
| project-alpha | qa-team | **Read** |
| project-beta | backend-team | **Write** |
| project-beta | frontend-team | **Read** |
| internal-docs | qa-team | **Write** |

1. Navigate to each repository's **Settings → Manage Access** section to assign these permissions.

**Managing Dormant Users in GitHub Enterprise**

In **GitHub Enterprise Cloud**, user accounts that are inactive for a certain period are considered **dormant users**. Managing these accounts is essential to maintaining security, license optimization, and ensuring active participation in your organization.

**What Are Dormant Users?**

A **dormant user** is a member of your GitHub Enterprise Cloud organization who has not performed any activity within a specific timeframe. Dormant users:

✅ Retain their membership in the organization.  
✅ Continue to consume a license.  
✅ Pose potential security risks if not actively managed.

**Identifying Dormant Users**

GHEC offers an **Activity Log** and **API** methods to identify inactive users. A user is considered dormant if:

* They have not logged in.
* They have not pushed to repositories.
* They have not commented on issues, pull requests, or discussions.
* They have not performed any other verifiable action.

**Dormancy Period:** By default, GitHub considers a user dormant if they have been inactive for **90 days**.

**Steps to Manage Dormant Users**

Here’s how you can effectively manage dormant users in GHEC:

**Step 1: Identify Dormant Users via the Web UI**

1. Navigate to your **organization's settings**.
2. In the left sidebar, select **People**.
3. Click the **"Members"** tab.
4. Apply the filter **dormant** to list users who haven’t been active for 90 days or longer.

**Step 2: Identify Dormant Users via API (Recommended for Automation)**

For bulk operations or automated checks, use the following API endpoint:

**API Endpoint:**

bash

CopyEdit

GET /orgs/{org}/members?filter=dormant

**Example:**

bash

CopyEdit

curl -H "Authorization: token YOUR\_GITHUB\_TOKEN" \

-H "Accept: application/vnd.github+json" \

https://api.github.com/orgs/YOUR\_ORG/members?filter=dormant

This returns a list of dormant users in JSON format.

**Step 3: Audit Dormant Users**

Once identified, analyze these users for:

* **Role** in the organization (e.g., Admin, Member, Billing Manager).
* **Access Levels** to repositories, projects, or organizational data.
* **Potential Security Risk**, especially if they have privileged access.

**Step 4: Take Appropriate Actions**

Depending on your organization's policy, you can take the following actions:

✅ **Contact Users:** Send reminders to encourage activity.  
✅ **Remove from the Organization:** Ideal if the user no longer requires access.  
✅ **Demote to Outside Collaborator:** Useful for retaining limited access.  
✅ **Suspend the Account (For SAML SSO):** If your organization uses SAML SSO, suspension may be the best option.

**Step 5: Automate Dormant User Management (Recommended for Large Organizations)**

For enterprises with hundreds of users, automation reduces manual effort. GitHub Actions or scheduled API calls can automate:

* Periodic checks for dormant users.
* Sending email reminders.
* Automatically revoking inactive accounts.

**Example GitHub Action Workflow:**

yaml

CopyEdit

name: Dormant User Audit

on:

schedule:

- cron: "0 0 1 \* \*" # Runs monthly

jobs:

check-dormant-users:

runs-on: ubuntu-latest

steps:

- name: Get Dormant Users

run: |

curl -H "Authorization: token ${{ secrets.GITHUB\_TOKEN }}" \

-H "Accept: application/vnd.github+json" \

https://api.github.com/orgs/YOUR\_ORG/members?filter=dormant

**Step 6: Implement Security Best Practices**

To reduce the risks associated with dormant accounts:

✅ Enable **2FA** for all organization members.  
✅ Set **automatic removal policies** for dormant users.  
✅ Regularly audit membership with tools like **GitHub Audit Log**.  
✅ Use **GitHub Actions** to automate notifications and removal.

**Key Considerations for GHEC**

* GitHub **does not automatically remove dormant users**; manual action or automation is required.
* Dormant users still consume seats in your GitHub Enterprise Cloud **license count**.
* Users marked dormant may still have access to **private repositories**, unless explicitly removed.

**Best Practices for Managing Dormant Users**

✅ Establish a **Dormant User Policy** within your organization’s security guidelines.  
✅ Regularly schedule audits (e.g., quarterly or biannually).  
✅ Automate reminders to ensure critical accounts don’t become inactive unintentionally.  
✅ Use **GitHub Insights** for broader reporting on user activity.

**============================================================**

**DAY-4**

**What is a Pull Request (PR)?**

A **Pull Request (PR)** is how you ask other people to review and merge your changes (in your branch) into the main project (typically the main branch). It's like saying, “Hey, I’ve worked on this new chapter (feature or bug fix), can you check it and, if it’s good, add it to the main manuscript (the main code)?”

**How to Create a Pull Request?**

1. **Once your work on a branch is done**, you create a Pull Request to let others know that you're ready to merge your changes into the main branch.
2. **Using GitHub Web Interface:**
   * Go to the **Pull Requests** tab in your repository.
   * Click **New Pull Request**.
   * Choose the **base branch** (usually main) and the **compare branch** (your feature branch like feature/add-login-page).
   * Write a short description explaining what you did.
   * Click **Create Pull Request** to send it for review.
3. **Using GitHub CLI:** If you prefer using the terminal, you can use GitHub CLI (a command line tool for GitHub) to create the pull request:

bash

CopyEdit

gh pr create --base main --head feature/add-login-page --title "Add login page" --body "This PR adds a login page."

**Why Use Branches and Pull Requests?**

* **Separation of Work**: You can work on new features without affecting the main project.
* **Team Collaboration**: Multiple people can work on different features or fixes simultaneously in separate branches.
* **Code Review**: Pull requests allow team members to review and discuss changes before merging them into the main branch.
* **Prevent Mistakes**: By reviewing code before merging, you reduce the risk of errors in the main project.
* **Lab Demo: Create Branches and Pull Requests**

This lab demo will guide you step-by-step through the process of creating branches and pull requests. You will learn how to use the GitHub web interface, Git CLI, and GitHub CLI to manage branches and collaborate through pull requests.

**Step 1: Prerequisites**

Before starting, ensure the following:

* You have access to a **GitHub Enterprise Cloud (GHEC)** organization and repository.
* You have Git installed on your local machine, and you have a basic understanding of Git commands.

**Step 2: Creating a New Branch**

In this step, you will create a new branch where you can work on changes independently.

**Create a Branch using GitHub Web Interface:**

1. **Go to Your Repository:**
   * Log in to your GitHub Enterprise Cloud instance.
   * Navigate to the repository where you want to create the branch.
2. **Access the Branch Dropdown:**
   * In the top left of the repository’s page, you will see a button labeled **Branch: main** (or the default branch name).
   * Click on the **Branch: main** dropdown to see a list of branches.
3. **Create a New Branch:**
   * In the **Branch** dropdown, type the name of your new branch in the search box.
   * If the branch name does not already exist, it will prompt you to create a new branch.
   * Type the name of your branch (e.g., feature/add-new-feature), and press **Enter**.

Now you have created the new branch and switched to it.

**Create a Branch using Git CLI:**

1. **Clone the Repository (if not done already):**
   * Open your terminal or command prompt and run:

bash

CopyEdit

git clone https://github.com/your-org/your-repository.git

cd your-repository

1. **Create a New Branch:**
   * Run the following command to create a new branch and switch to it:

bash

CopyEdit

git checkout -b feature/add-new-feature

1. **Push the Branch to GitHub:**
   * Push the branch to GitHub so it can be used for collaboration:

bash

CopyEdit

git push origin feature/add-new-feature

**Step 3: Making Changes and Committing Them**

Now that you've created a branch, let’s make some changes.

1. Open your repository folder in your code editor.
2. Modify or add some files (e.g., editing a README or adding a new feature).
3. Add and commit your changes:

bash

CopyEdit

git add .

git commit -m "Add new feature"

git push origin feature/add-new-feature

**Step 4: Creating a Pull Request (PR)**

After pushing the changes to GitHub, you can create a pull request to propose merging the changes from your feature branch into the main branch.

**Create a Pull Request using GitHub Web Interface:**

1. **Navigate to Your Repository:**
   * Go to the repository where you pushed your branch (feature/add-new-feature).
2. **Start a New Pull Request:**
   * GitHub will often display a banner prompting you to create a pull request for your newly pushed branch.
   * Alternatively, click on the **Pull Requests** tab at the top of the repository page and then click the **New Pull Request** button.
3. **Choose Base and Compare Branches:**
   * In the **base** dropdown, select the branch you want to merge into (typically main).
   * In the **compare** dropdown, select the branch you created (e.g., feature/add-new-feature).
4. **Review Changes:**
   * GitHub will show you the differences between the base branch (main) and your feature branch.
   * You can review the changes here, including any new commits, added files, or modified code.
5. **Create the Pull Request:**
   * Add a title for your pull request (e.g., “Add new feature”).
   * Provide a description explaining the purpose of the changes.
   * Optionally, assign reviewers and add labels, projects, or milestones.
   * Click **Create Pull Request**.

**Create a Pull Request using GitHub CLI (gh):**

1. **Install GitHub CLI:**
   * If you don’t have GitHub CLI (gh) installed, you can install it from [GitHub CLI Installation](https://cli.github.com/).
2. **Create a Pull Request:**
   * In your terminal, run the following command:

bash

CopyEdit

gh pr create --base main --head feature/add-new-feature --title "Add new feature" --body "This pull request adds a new feature to the application."

* + This will automatically create a pull request with the specified base branch (main) and the branch you created (feature/add-new-feature).

**Step 5: Reviewing and Merging the Pull Request**

Once the pull request is created, the reviewer (or you, if you have admin rights) will need to review the changes and merge them.

**Reviewing a Pull Request on GitHub:**

1. **Navigate to the Pull Request:**
   * Click on the **Pull Requests** tab in your repository.
   * Select the pull request you just created.
2. **Review the Changes:**
   * GitHub will show a comparison of the base branch (main) and the feature branch.
   * Review the code changes, and if everything looks good, click **Approve** to approve the pull request (or comment if there are suggestions or issues).

**Merge the Pull Request:**

1. If the pull request is ready to be merged, click the **Merge pull request** button.
2. GitHub will prompt you to confirm the merge. You can choose to **Merge** or **Squash and merge** if you want to combine all commits into one.
3. After merging, you can delete the feature branch by clicking **Delete branch**.

**Step 6: Clean Up Your Local Repository**

Once the PR is merged, it’s important to keep your local repository in sync with the remote repository.

1. **Switch to the main branch:**

bash

CopyEdit

git checkout main

1. **Pull the latest changes:**

bash

CopyEdit

git pull origin main

1. **Delete the local feature branch (optional):**

bash

CopyEdit

git branch -d feature/add-new-feature

**Conclusion**

You’ve now learned how to:

1. Create a new branch using both the GitHub Web interface and Git CLI.
2. Make changes and commit them to your branch.
3. Create a pull request (PR) to propose merging changes into the main branch.
4. Review and merge a PR.
5. Keep your local repository up to date.

**Auditing account activities in** **GitHub Enterprise** is crucial for tracking user actions, ensuring security, and maintaining monitoring compliance. GHEC provides several methods and tools to audit account activities effectively.

**Key Methods for Auditing Account Activities**

GitHub Enterprise Cloud offers multiple ways to track user activities:

**1. Audit Log (Primary Method)**

* The **audit log** captures detailed information about various events within your organization.
* It helps you track security events, user activities, and repository changes.

**Accessing the Audit Log**

* Go to your organization → **Settings** → **Audit Log**
* For REST API access:

bash

CopyEdit

curl -H "Authorization: token YOUR\_TOKEN" \

"https://api.github.com/orgs/ORG\_NAME/audit-log"

**Audit Log Filters**

You can filter the audit log based on:

* **Actor**: Specific user actions
* **Event**: Specific event types (e.g., team.create, repo.create)
* **Date Range**: Filter by specific dates
* **Action Types**: Filter by created, updated, deleted, etc.

**Example Search Query in UI:**

action:team.create actor:username

**2. GitHub API for Audit Events**

The REST API allows you to fetch audit log details programmatically.

**Example API Call:**

bash

CopyEdit

curl -H "Authorization: token YOUR\_TOKEN" \

"https://api.github.com/orgs/ORG\_NAME/audit-log?phrase=repo.create"

**Important API Parameters:**

* phrase: Filters events by keywords
* before and after: Time range
* order: Sorting order (ascending/descending)

**3. GitHub Advanced Security (GHAE)**

If you're using **GitHub Advanced Security**, you gain access to:

* **Code scanning**
* **Secret scanning**
* **Dependency review**

These security features add another layer of insight for tracking risky actions or vulnerabilities.

**4. Enterprise Account Audit Log (For Enterprise Admins)**

* Enterprise owners can access the audit log for all organizations within the enterprise.
* Provides comprehensive insights into user access, billing, and integrations.

**API Example for Enterprise Audit Log:**

bash

CopyEdit

curl -H "Authorization: token YOUR\_TOKEN" \

"https://api.github.com/enterprises/ENTERPRISE\_NAME/audit-log"

**6. External Tools for Auditing Integration**

For enhanced reporting, you can integrate GHEC with:

* **SIEM Solutions** (e.g., Splunk, Datadog, AWS CloudWatch)
* **Log Management Platforms** (e.g., Graylog, ELK Stack)

**Recommended Best Practices for Auditing in GHEC**

✅ Enable **SAML SSO** and **SCIM** for identity and access management.  
✅ Regularly review the audit log and webhook data for suspicious activities.  
✅ Integrate GHEC with your SIEM platform for centralized logging.  
✅ Enforce **2FA** (Two-Factor Authentication) to minimize compromised account risks.  
✅ Use **fine-grained personal access tokens** with minimal permissions.

**Example Use Cases for Audit Tracking**

1. **Tracking Repo Creation/Deletion:**
   * Use search filter: action:repo.create OR action:repo.delete
2. **Monitoring Permission Changes:**
   * Use search filter: action:team.add\_member
3. **Detecting Suspicious IP Activity:**
   * Filter by actor\_ip field.

| **Filter** | **Description** | **Example** |
| --- | --- | --- |
| action: | Tracks specific types of events | action:repo.create |
| actor: | Filters actions performed by a specific user | actor:john\_doe |
| actor\_ip: | Filters events originating from a specific IP | actor\_ip:203.0.113.1 |
| repo: | Filters events within a specific repository | repo:my-repo |
| org: | Tracks actions within a specific organization | org:my-organization |
| created: | Filters events created on a specific date | created:2024-03-01 |
| before: | Filters actions before a specific date | before:2024-01-01 |
| after: | Filters actions after a specific date | after:2024-02-01 |
| oauth\_app\_name: | Tracks actions involving a specific OAuth app | oauth\_app\_name:AzureAD |
| user: | Filters actions involving a particular user | user:developer01 |
| action:webhook. | Tracks webhook-related activities | action:webhook.create |
| action:ssh\_key. | Filters SSH key additions or deletions | action:ssh\_key.create |
| action:team. | Tracks team-related actions (creation, updates) | action:team.create |
| action:org. | Tracks organization-wide activities (e.g., invites) | action:org.invite\_member |
| action:public\_key. | Tracks deploy key creation or deletion | action:public\_key.create |
| action:billing. | Tracks billing updates or changes | action:billing.subscription\_updated |

**✅ Example Filter Combinations for Efficient Tracking**

1. **Monitor Repo Creation by a Specific User:**

action:repo.create actor:john\_doe

1. **Track Failed Login Attempts from a Specific IP:**

action:failed\_login actor\_ip:192.168.1.\*

1. **Audit OAuth App Authorizations After a Certain Date:**

action:oauth\_authorization.create after:2024-01-01

1. **Identify Deleted Repositories Within a Date Range:**

action:repo.delete created:>2024-01-01 created:<2024-03-01

1. **Monitor SSH Key Deletions Across the Organization:**

action:ssh\_key.delete org:my-org

**Protected branches :**

In GitHub Enterprise, **protected branches** are a way to enforce certain rules and safeguards on important branches in your repository (like main or master). These rules help maintain the integrity of the code, ensure quality control, and prevent unwanted changes that might cause issues in your project.

Here's a simple explanation of how protected branches work and how to set them up:

**What Are Protected Branches?**

A **protected branch** is a branch that has specific rules applied to it, such as:

* **Preventing force pushes**: Users can’t overwrite the history of the branch.
* **Requiring pull requests (PRs)**: Direct pushes to the branch are not allowed, and code changes need to be reviewed and merged through a pull request.
* **Enforcing status checks**: Before merging a PR, you can require that certain tests or checks (like CI/CD builds) must pass.
* **Requiring code reviews**: To ensure quality, at least one (or more ) team members must approve changes before they can be merged into the protected branch.

**Why Are They Important?**

* **Prevents accidental changes**: Prevents developers from accidentally pushing broken code or overwriting important changes.
* **Improves code quality**: Forces a process for reviewing and testing code before it’s merged into key branches.
* **Increases collaboration**: Encourages team collaboration by requiring approvals before changes are made.

**How to Set Up Protected Branches in GitHub Enterprise:**

1. **Go to Your Repository**: Navigate to your repository on GitHub Enterprise.
2. **Open Settings**: On the repository page, click on **Settings** (you need to have admin privileges to access this).
3. **Find Branches Settings**: In the left sidebar, under **Code and Automation**, click on **Branches**.
4. **Add Protection Rule**:
   * Under **Branch protection rules**, click on **Add rule**.
   * **Choose the Branch**: Specify which branch you want to protect (e.g., main).
   * Set the rules you want to enforce, like:
     + **Require pull request reviews before merging**.
     + **Require status checks to pass before merging** (such as CI tests).
     + **Restrict who can push to the branch** (limit this to certain team members or roles).
     + **Prevent force pushes** to avoid accidental overwriting of history.
5. **Save the Rule**: After setting the desired rules, click **Create** or **Save changes**.

**Common Protection Rules:**

* **Require pull request reviews**: Ensure changes are approved before being merged into the protected branch.
* **Require status checks**: Only allow merging if the code passes tests or other automated checks.
* **Restrict who can push to the branch**: Limit write access to specific users or teams.
* **Include administrators**: You can choose to apply these rules to admins as well (or exempt them).

**Example Scenario:**

Let’s say you have a repository with a main branch, which is where you want to keep the stable, production-ready code. By setting up the protected branch rule for main:

* Only pull requests (PRs) from team members will be allowed.
* The PRs must pass automated tests (CI/CD checks) before they can be merged.
* A team member has to review and approve the PR before merging.
* No one can force push or overwrite the history of the main branch.

**Conclusion:**

Protected branches are a simple but powerful feature in GitHub Enterprise to ensure that only validated and approved code makes it to critical branches like main. It helps maintain the stability and security of your codebase, and also encourages collaboration and code reviews.

Lab demo for **protected branches** in **GitHub Enterprise (GHE)**, we'll walk you through the necessary steps, including setting up a repository, configuring protected branches, and demonstrating how the rules work. Below is a step-by-step guide for setting up and demonstrating the use of protected branches in a GitHub Enterprise environment.

**Lab Setup Overview:**

1. **Create a GitHub Enterprise Organization & Repository**.
2. **Create a New Branch** (other than main) for the demo.
3. **Set up Branch Protection Rules** on the main branch.
4. **Demonstrate Pull Request Creation, Reviews, and Merging**.
5. **Show Forced Push Attempt on Protected Branch**.
6. **Demonstrate Successful and Unsuccessful Merges** based on protection rules.

**Step 1: Create a GitHub Enterprise Organization & Repository**

1. **Login to GitHub Enterprise** using your admin account.
2. **Create a New Organization**:
   * Go to your GitHub Enterprise home page.
   * Click on **Your Profile** (top-right) → **Your organizations**.
   * Click **New Organization**.
   * Provide the organization name (e.g., DemoOrg), description, and select visibility.
3. **Create a Repository** inside the organization:
   * From the **organization dashboard**, click **New** to create a new repository.
   * Name the repository demo-repo.
   * Set visibility as **Private** (you can also choose Public if needed).
   * Select **Initialize this repository with a README**.
   * Click **Create repository**.

**Step 2: Create a New Branch (e.g., feature-branch)**

1. **Clone the repository**:

bash

CopyEdit

git clone https://your-ghe-url/DemoOrg/demo-repo.git

cd demo-repo

1. **Create a new branch** (e.g., feature-branch):

bash

CopyEdit

git checkout -b feature-branch

1. **Make some changes to a file** (e.g., README.md):
   * Open the README.md file and add a new line at the end:  
     This is a demo of protected branches in GitHub Enterprise.
   * Save the file.
2. **Commit and push the changes**:

bash

CopyEdit

git add README.md

git commit -m "Updated README for demo"

git push origin feature-branch

* + - * Test before applying the protection rule

vi README.md

260 git add .

261 git commit -m "added raman2"

262 git push origin main

**Step 3: Set Up Branch Protection Rules on main Branch**

1. **Go to your repository settings**:
   * On the GitHub Enterprise web interface, navigate to your repository: demo-repo.
   * Click on **Settings** (near the top right of the page).
2. **Configure Branch Protection**:
   * In the left sidebar, under **Code and Automation**, click on **Branches**.
   * Under **Branch protection rules**, click on **Add rule**.
   * Mention main branch.
3. **Set Branch Protection Rules**:
   * Enable the following rules (you can add more based on your needs):
     + **Require pull request reviews before merging** (At least one approval).
     + **Do not allow bypassing the above settings**
4. **Now you will notice after giving a review still requester is able to merge by thems elves ; so add below rule in bp rules as well :**
   * + **Restrict who can push to matching branches : add ramankhanna2(owner ) in it.**
5. Than try it again …
6. **Save the Rule**: Once configured, click **Create** or **Save changes** to apply the protection.
   * + - Test it one more time now u will notice that rule will restrict the push to main 😊
7. vi README.md
8. 260 git add .
9. 261 git commit -m "added raman2"
10. 262 git push origin main

**Step 5: Show Forced Push Attempt on Protected Branch**

1. **Attempt a Force Push**:
   * Now that the main branch is protected, try force pushing to it. First, switch to the main branch:

bash

CopyEdit

git checkout main

* + Attempt a force push:

bash

CopyEdit

git push origin main --force

* + GitHub Enterprise will block the force push, and you'll see an error message like:

vbnet

CopyEdit

remote: error: refusing to update checked out branch: refs/heads/main

remote: error: by default, updating the current branch in this repository

remote: error: is forbidden, and you have no permission to force it.

* + - * Enable below protection rukes as well :
    - **Prevent force pushes** to avoid overwriting the branch history.
  + Now check by doing a force push again , it will work

**Step 4: Demonstrate Pull Request Creation, Reviews, and Merging (not tested)**

1. **Create a Pull Request (PR)**:
   * Go to the **Pull Requests** tab in your repository.
   * Click on **New Pull Request**.
   * Choose feature-branch as the source branch and main as the target.
   * Add a meaningful title and description for the PR.
   * Submit the pull request.
2. **Add Reviewers**:
   * Select one or more team members as reviewers for the PR.
   * If status checks are enabled, you’ll see them run (if you have CI/CD set up).
3. **Review and Approve**:
   * A reviewer will approve the PR once the changes look good.
   * If there are any conflicts or issues, the PR will be blocked until they are resolved.
4. **Merge the PR**:
   * Once approved, the PR can be merged by someone with write access (unless restricted by protection rules).
   * The option to merge will be enabled once all checks pass.

**Step 6: Demonstrate Successful and Unsuccessful Merges**

1. **Unsuccessful Merge Attempt (without PR)**:
   * Try pushing directly to the main branch without a PR:

bash

CopyEdit

git checkout main

git push origin main

* + GitHub will reject this push because the main branch is protected and doesn't allow direct pushes.

1. **Successful Merge with PR**:
   * After the pull request is reviewed and approved, the merge can happen successfully.
   * After the merge is completed, check that the README.md file is updated with your changes.

**Conclusion:**

In this demo, we walked through creating a repository in GitHub Enterprise, setting up a protected branch (main), and demonstrated how the protection rules (e.g., requiring pull request reviews, preventing force pushes) work. We also showed how direct pushes are blocked and how only approved pull requests can be merged. This ensures better collaboration and code quality control in your organization.

An **"Unhealthy repository**" refers to a repo that has issues or problems that make it difficult to work with, maintain, or scale. These issues can slow down development, cause errors, or make it hard for team members to collaborate effectively. Here are some common signs and causes of an unhealthy repository:

**Signs of an Unhealthy Repository**

1. **Large Repository Size**:
   * The repo has become too big, often because it contains unnecessary files (like binaries, logs, or old assets). This makes it slow to clone, pull, or push changes.

**Solutions:**

* **Use .gitignore:** Ensure unnecessary files (e.g., logs, binaries, build artifacts) are excluded from being tracked by Git.
* **Remove tracked unnecessary files:** Use git filter-repo or BFG Repo-Cleaner to permanently remove large files or directories from the repo's history.
* **Use Git LFS (Large File Storage):** For large assets like binaries, use Git LFS to store them outside the main repo.

1. **Messy Commit History**:
   * The commit history is cluttered with unclear, repetitive, or meaningless commit messages. This makes it hard to track changes or understand the project's evolution.

**Solutions:**

* **Squash commits:** Use git rebase -i to squash small, repetitive commits into larger, logical ones before merging into the main branch.

1. **Frequent Merge Conflicts**:
   * Team members often face conflicts when merging their changes. This usually happens when the repo isn't structured well or when people aren't following a consistent workflow.
2. **Lack of Branch Management**:
   * There are too many branches, or branches are not deleted after they're no longer needed. This creates confusion and makes it hard to know which branch is the "source of truth."
3. **Outdated or Unused Code**:
   * The repo contains old, unused, or deprecated code that hasn't been cleaned up. This adds unnecessary complexity and can confuse developers.
4. **Poor Documentation**:
   * The repo lacks clear documentation, such as a README file, contribution guidelines, or explanations of the codebase. This makes it hard for new developers to understand and contribute to the project.
5. **Security Issues**:
   * The repo might contain sensitive information (like passwords or API keys) that shouldn't be exposed, or it might use outdated dependencies with known vulnerabilities.

**Solutions:**

* **Remove sensitive data:** Use git filter-repo or BFG Repo-Cleaner to remove sensitive information (e.g., passwords, API keys) from the repo's history.
* **Use environment variables:** Store sensitive information in environment variables or secret management tools (e.g., AWS Secrets Manager, HashiCorp Vault).
* **Scan for secrets:** Use tools like truffleHog or git-secrets to detect and prevent sensitive data from being committed.

**Changing History with Git**

Changing the history of a Git repository is a powerful but dangerous operation. It's often done when you want to clean up commit history, remove sensitive data, or modify an earlier commit message. However, this can rewrite the repository's history, and if you're working with shared or remote branches, it can cause conflicts. So, it’s essential to understand the implications before performing such changes.

Here’s an overview of the primary methods used to change history in Git:

**1. Amend the Last Commit (git commit --amend)**

If you made a mistake in the last commit (like a wrong message, missing files, etc.), you can amend it using the following command:

**Example:**

bash

CopyEdit

git commit --amend

This opens your default text editor to modify the commit message. You can also add new files or changes to the commit.

If you only want to change the commit message without modifying the contents, you can use:

bash

CopyEdit

git commit --amend --no-edit

**Important Considerations:**

* This changes the commit hash because it’s creating a new commit.
* **Don't amend commits that have been shared** with others, as this will cause discrepancies.

**2. Rebase Interactive (git rebase -i)**

Rebasing allows you to rewrite commits in a branch, and interactive rebasing (-i) gives you more control. This is useful when you want to modify multiple commits, reword commit messages, squash commits, or reorder commits.

**Example:**

To rebase the last 3 commits:

bash

CopyEdit

git rebase -i HEAD~3

Git will open an editor with a list of recent commits, looking something like this:

sql

CopyEdit

pick abc123 Commit message 1

pick def456 Commit message 2

pick ghi789 Commit message 3

You can then change pick to other commands:

* pick: Keep the commit as is.
* reword: Change the commit message.
* edit: Stop at this commit to make changes.
* squash: Combine this commit with the previous one.
* fixup: Similar to squash, but discards the commit message.
* drop: Remove the commit.

**After making changes, save and close the editor. Git will replay the commits with the new changes. If you modified or reordered commits, you may need to resolve conflicts during the process.**

**Important Considerations:**

* **Rebase is destructive** and rewrites commit history, so you should **never rebase public/shared branches** unless you're sure no one else is working with the branch.
* After rebasing, if you've pushed the branch previously, you'll need to use git push --force to push your changes.

**3. Rewriting History with git filter-branch or git filter-repo**

Sometimes, you may need to rewrite history for things like removing sensitive data or large files. While git filter-branch can help rewrite the history, it is considered slow and inefficient for large repositories. A better alternative is git filter-repo, which is faster and more flexible.

**Example Using git filter-branch:**

To remove a file from all previous commits:

bash

CopyEdit

git filter-branch --tree-filter 'rm -f path/to/file' --prune-empty -- --all

This command removes the file path/to/file from all commits in the history. --prune-empty ensures empty commits are removed after the file is deleted.

**Example Using git filter-repo:**

First, install git-filter-repo (if not already installed):

bash

CopyEdit

pip install git-filter-repo

To remove a file from the entire history using git filter-repo:

bash

CopyEdit

git filter-repo --path path/to/file --invert-paths

This command will remove the file from all commits where it appears in the repository history.

**Important Considerations:**

* **History is rewritten**: Like other rewriting methods, filter-branch and filter-repo change the commit hashes and can affect collaboration if not done carefully.
* After using git filter-branch or git filter-repo, you will need to use git push --force to push the changes to a remote repository.

**4. Changing a Specific Commit in the Middle of History**

If you want to change a specific commit that's not the most recent one, you can use an interactive rebase to edit that commit.

**Example:**

To change a commit that is 3 commits behind:

bash

CopyEdit

git rebase -i HEAD~5

This command will show the last 5 commits. Find the commit you want to edit and change pick to edit for that commit:

sql

CopyEdit

pick abc123 Commit 1

edit def456 Commit 2

pick ghi789 Commit 3

After that, Git will pause at the selected commit. You can make your changes (e.g., modifying files or the commit message). Once done, continue the rebase:

bash

CopyEdit

git commit --amend # Edit the commit

git rebase --continue

**5. Rewriting History with git reflog and git reset**

If you made changes in your commit history (e.g., after a git rebase or git reset) and want to undo them, Git keeps a record of changes in the **reflog**.

**Example:**

If you want to go back to a previous state in the history:

bash

CopyEdit

git reflog

Find the commit you want to go back to (identified by its hash) and reset to it:

bash

CopyEdit

git reset --hard <commit-hash>

This command will discard any changes after that commit and reset the working directory.

**6. Removing Sensitive Data with BFG Repo-Cleaner**

BFG Repo-Cleaner is a faster alternative to git filter-branch for removing sensitive data (such as passwords or large files) from Git history.

**Example:**

To remove sensitive files from all history:

bash

CopyEdit

bfg --delete-files 'filename' repo.git

To remove specific sensitive data (e.g., passwords):

bash

CopyEdit

bfg --replace-text passwords.txt repo.git

After running BFG, you should follow it up with a forced push (git push --force) to update the remote repository.

* **Maintenance Mode**
* **Definition**: Maintenance mode is a state where a system is taken offline or restricted to perform updates, patches, or other administrative tasks without disrupting users.
* **GitHub Enterprise Cloud**: Since GitHub Enterprise Cloud is managed by GitHub (Microsoft), maintenance is handled by GitHub’s team. Users are typically notified in advance of scheduled maintenance, and downtime is minimized or avoided entirely due to redundant systems.
* **GitHub Enterprise Server**: In a self-hosted instance, the admin must manually enable maintenance mode to perform updates or fixes. During this time, users cannot access the instance.
* **Real-World Example**: If GitHub Enterprise Server is hosted on your company’s private data center, you might schedule maintenance during off-hours to apply a critical security patch. You’d enable maintenance mode, apply the patch, and then bring the system back online. In contrast, GitHub Enterprise Cloud users might not even notice the maintenance because it’s handled seamlessly by GitHub.
* **Version Upgrades**
* **Definition**: Version upgrades involve updating the software to a newer release to access new features, security patches, or performance improvements.
* **GitHub Enterprise Cloud**: Upgrades are automatic and handled by GitHub. Users always have access to the latest version without needing to intervene.
* **GitHub Enterprise Server**: Admins must manually plan and execute version upgrades. This involves downloading the new version, testing it in a staging environment, and then deploying it to the production instance.
* **Real-World Example**: Suppose GitHub releases a new version with improved CI/CD integration. For GitHub Enterprise Cloud, this feature becomes available immediately. For GitHub Enterprise Server, your IT team would need to schedule downtime, test the upgrade, and deploy it, which could take days or weeks depending on your processes.
* **High Availability Replica Instance**
* **Definition**: A high availability (HA) replica instance is a duplicate of the primary system that ensures minimal downtime in case of failure. If the primary instance goes down, the replica takes over.
* **GitHub Enterprise Cloud**: High availability is built into the cloud service. GitHub uses redundant servers and data centers to ensure uptime and reliability.
* **GitHub Enterprise Server**: If you want high availability, you must set up and manage replica instances yourself. This involves configuring multiple servers, load balancers, and failover mechanisms.
* **Real-World Example**: If your GitHub Enterprise Server instance is hosted on-premises and the primary server fails, the replica instance would automatically take over, ensuring no disruption to your developers. In GitHub Enterprise Cloud, this failover is handled automatically by GitHub’s infrastructure, so you don’t need to worry about it.

**Backup Utilities**

* **Definition**: Backup utilities are tools or processes used to create copies of data to prevent data loss in case of hardware failure, corruption, or accidental deletion.
* **GitHub Enterprise Cloud**: Backups are managed by GitHub. They use redundant storage and regular snapshots to ensure data integrity and availability.
* **GitHub Enterprise Server**: You are responsible for setting up and managing backups. GitHub provides tools like ghe-backup to create backups, but you must configure where and how often backups are stored.
* **Real-World Example**: If a developer accidentally deletes a critical repository, GitHub Enterprise Cloud users can rely on GitHub’s backup systems to restore the data. For GitHub Enterprise Server, your IT team would need to restore the repository from the latest backup, which could take time depending on your backup strategy.

**Summary Table: GitHub Enterprise Cloud vs. Server**

| **Feature** | **GitHub Enterprise Cloud (Cloud)** | **GitHub Enterprise Server (Self-Hosted)** |
| --- | --- | --- |
| **Maintenance Mode** | Handled by GitHub; minimal user involvement. | Admin must manually enable and manage maintenance. |
| **Version Upgrades** | Automatic and seamless. | Manual upgrades requiring planning and testing. |
| **High Availability** | Built-in redundancy and failover. | Requires manual setup of replica instances. |
| **Backup Utilities** | Managed by GitHub; no user action needed. | Admin must configure and manage backups. |

**Lab : to show Merge conflict :**

**Step 1: Create a New Repository**

1. Go to your **GitHub Enterprise Cloud** dashboard.
2. Click the **+** icon (top-right) → Select **New Repository**.
3. Enter:
   * **Repository Name:** merge-conflict-demo
   * **Visibility:** Private/Public
4. Initialize the repository with:
   * ✅ Add a README
   * ✅ Add a .gitignore (e.g., Node/Java/other project files)
5. Click **Create Repository**.

**Step 2: Clone the Repository Locally**

In your terminal:

bash

CopyEdit

git clone git@github.com:<your-org>/merge-conflict-demo.git

cd merge-conflict-demo

**Step 3: Create Two Branches**

We'll simulate two developers making conflicting changes.

bash

CopyEdit

# Create branch 'feature-A'

git checkout -b feature-A

echo "Initial content" > sample.txt

git add sample.txt

git commit -m "Add initial content"

git push origin feature-A

# Create branch 'feature-B'

git checkout main

git checkout -b feature-B

echo "Initial content" > sample.txt

git add sample.txt

git commit -m "Add initial content"

git push origin feature-B

**Step 4: Introduce Conflicting Changes**

In branch feature-A:

bash

CopyEdit

git checkout feature-A

echo "Change from Feature A" >> sample.txt

git commit -am "Update sample.txt from Feature A"

git push origin feature-A

In branch feature-B:

bash

CopyEdit

git checkout feature-B

echo "Change from Feature B" >> sample.txt

git commit -am "Update sample.txt from Feature B"

git push origin feature-B

**Step 5: Create a Pull Request (PR) for feature-A**

1. Go to your **GitHub Enterprise Cloud** repository.
2. Click the **"Pull requests"** tab.
3. Click **New pull request**.
4. Select **feature-A** as the source branch and **main** as the target branch.
5. Click **Create pull request**.

✅ **Merge feature-A into main.**  
✅ **main branch now contains the content from feature-A.**

**Step 6: Create a Pull Request (PR) for feature-B (Trigger Conflict)**

1. Repeat the above steps, but this time create a PR for **feature-B** into **main**.
2. GitHub will automatically detect a **merge conflict**.

**Step 7: Resolve the Conflict (Web UI)**

1. Click **Resolve conflicts** in the pull request.
2. Review the conflict markers:

markdown

CopyEdit

<<<<<<< feature-B

Change from Feature B

=======

Change from Feature A

>>>>>>> main

1. Manually modify the content to resolve the conflict:

sql

CopyEdit

Final Merged Content:

Change from Feature A

Change from Feature B

1. Click **Mark as resolved** → **Commit merge** → **Merge pull request**.

**Step 8: Resolve the Conflict (CLI)**

If you prefer CLI resolution:

1. Pull the latest changes from main.

bash

CopyEdit

git checkout feature-B

git fetch origin

git merge main

1. Git will highlight the conflict:

pgsql

CopyEdit

CONFLICT (content): Merge conflict in sample.txt

Automatic merge failed; fix conflicts and then commit the result.

1. Open sample.txt and resolve the conflict by editing the file:

css

CopyEdit

Change from Feature A

Change from Feature B

1. Mark the file as resolved:

bash

CopyEdit

git add sample.txt

git commit -m "Resolved conflict between feature-A and feature-B"

git push origin feature-B

1. Complete the PR on GitHub by merging the resolved branch.

**Step 9: Clean Up**

* Once both branches are merged, delete the branches from both the local repo and remote repository:

bash

CopyEdit

git branch -d feature-A

git branch -d feature-B

git push origin --delete feature-A

git push origin --delete feature-B