Q1.

**Explanation to the Team on Kubernetes Usage for Website Management:**

Kubernetes is a powerful system for managing containerized applications at scale. It provides an environment to deploy, manage, and scale applications in a way that abstracts away much of the complexity of the underlying infrastructure. For a company website, Kubernetes can help by automating deployment, scaling, and management, even during high load or heavy incoming requests.

Let’s break down how Kubernetes can help with our company website:

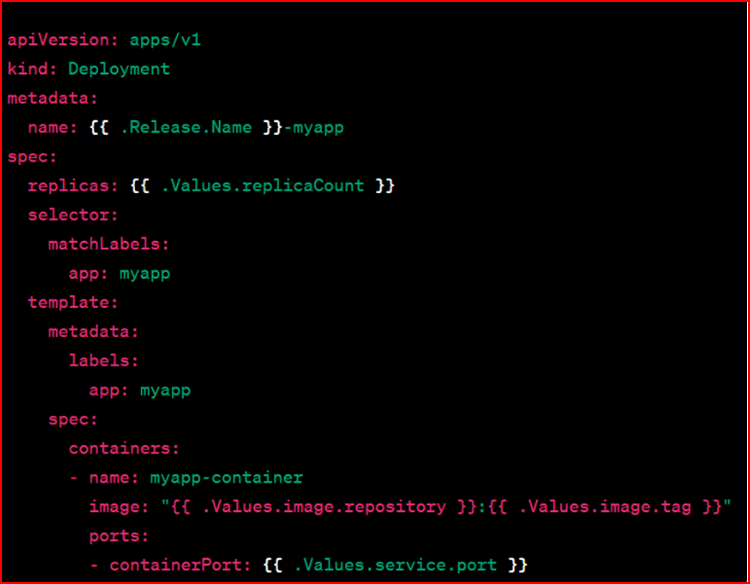
1. **Managing Traffic with High Load**: Kubernetes can automatically scale the number of instances (replicas) of your website application based on incoming traffic. If there’s a surge in traffic, Kubernetes can scale out the application to handle the increased load. If the traffic decreases, Kubernetes can scale it back down to save resources.
2. **High Availability**: Kubernetes runs multiple replicas of your application across different nodes in the cluster. This ensures that if one server (node) goes down, the application is still available on another node.
3. **Zero Downtime Deployments**: Kubernetes supports rolling updates, which means that you can deploy new versions of your website without taking it offline. It will update one instance at a time, ensuring there is no downtime for users.
4. **Automatic Failover**: Kubernetes monitors the health of the containers and automatically restarts any container that fails. This ensures the website remains up and running without manual intervention.

### Step 1: Create a Kubernetes Deployment

A **Deployment** in Kubernetes manages the desired state of an application, specifying how many replicas (copies) of the application should run, and which container image to use.

Here, we will create a Deployment for an NGINX web server.

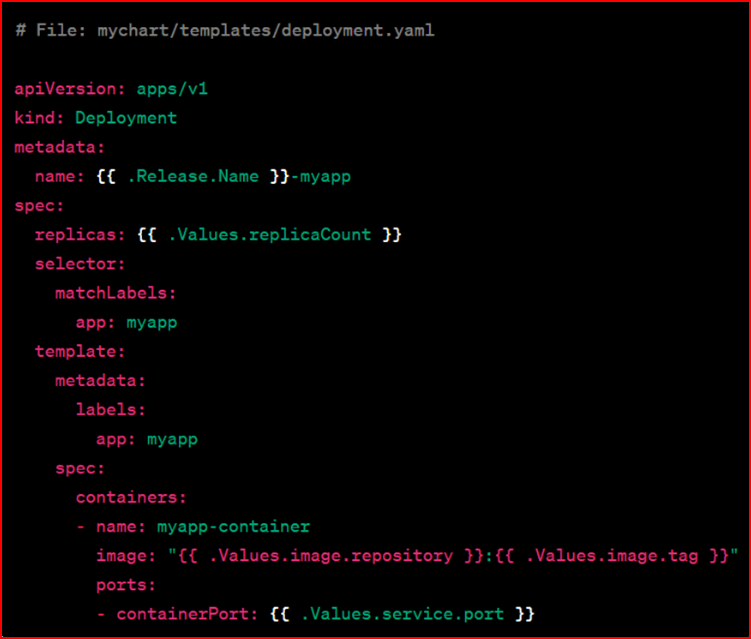
#### Kubernetes Deployment YAML file:



### Step 2: Create a Kubernetes Service

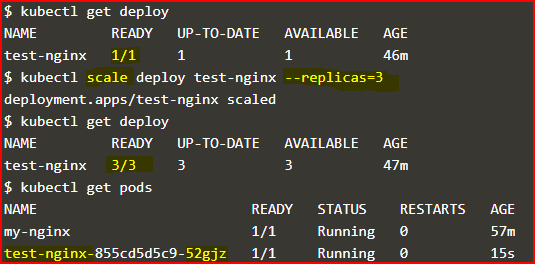
Once the deployment is created, we need a **Service** to expose the NGINX web application to external users. A Kubernetes **Service** allows communication between the Pods and the outside world.

Here, we will create a Service that will expose our NGINX deployment to external traffic.



### Step 3: Verify the Deployment and Service

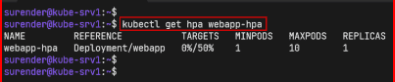
At this point, Kubernetes will have created both the Deployment and the Service. You can check the status of the deployment and ensure everything is running correctly.



### Step 4: Auto-Scaling the Application

Now, to ensure that your NGINX application can handle varying amounts of traffic, we can set up **Horizontal Pod Autoscaler (HPA)**. The HPA automatically adjusts the number of Pods based on CPU usage or other metrics.

For example, let's set up an autoscaler to increase the number of replicas if the CPU usage exceeds 50%.



### Step 5: Verify Auto-Scaling

You can verify the autoscaling by checking the status of the HPA:



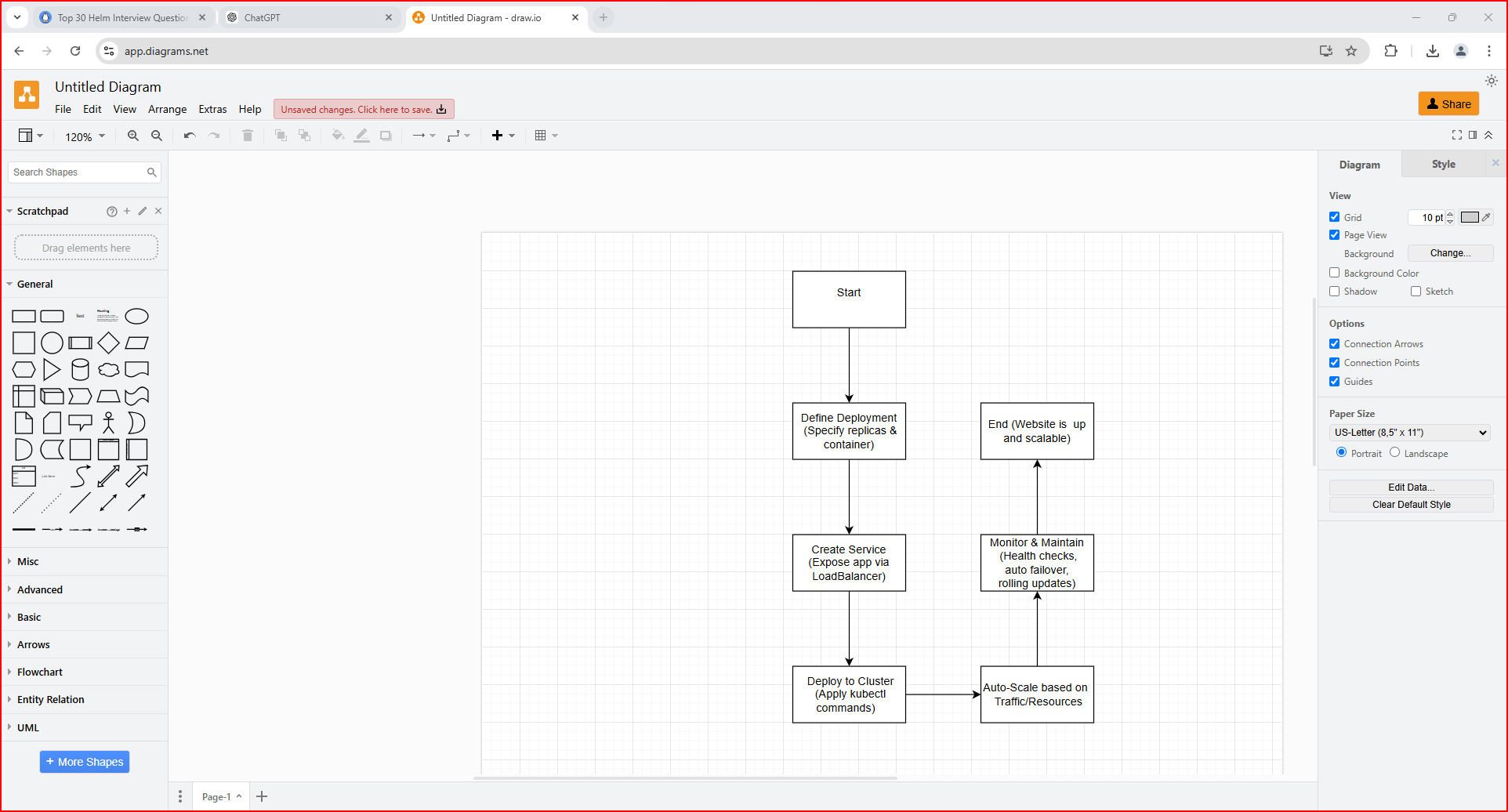
This will show the current state of the Horizontal Pod Autoscaler and how it’s adjusting the number of replicas based on CPU usage.

### Summary

* **Step 1**: We created a Kubernetes **Deployment** to define how the NGINX web server should run (3 replicas).
* **Step 2**: We exposed the NGINX service externally using a **Service** (type LoadBalancer).
* **Step 3**: We verified the deployment and service by checking the status of Pods and services.
* **Step 4**: We set up **Horizontal Pod Autoscaling** to automatically adjust the number of Pods based on CPU usage.

With this setup, Kubernetes handles deployment, scaling, and high availability for your company’s website, allowing it to efficiently manage high traffic and provide uninterrupted service.

**FLOWCHART**



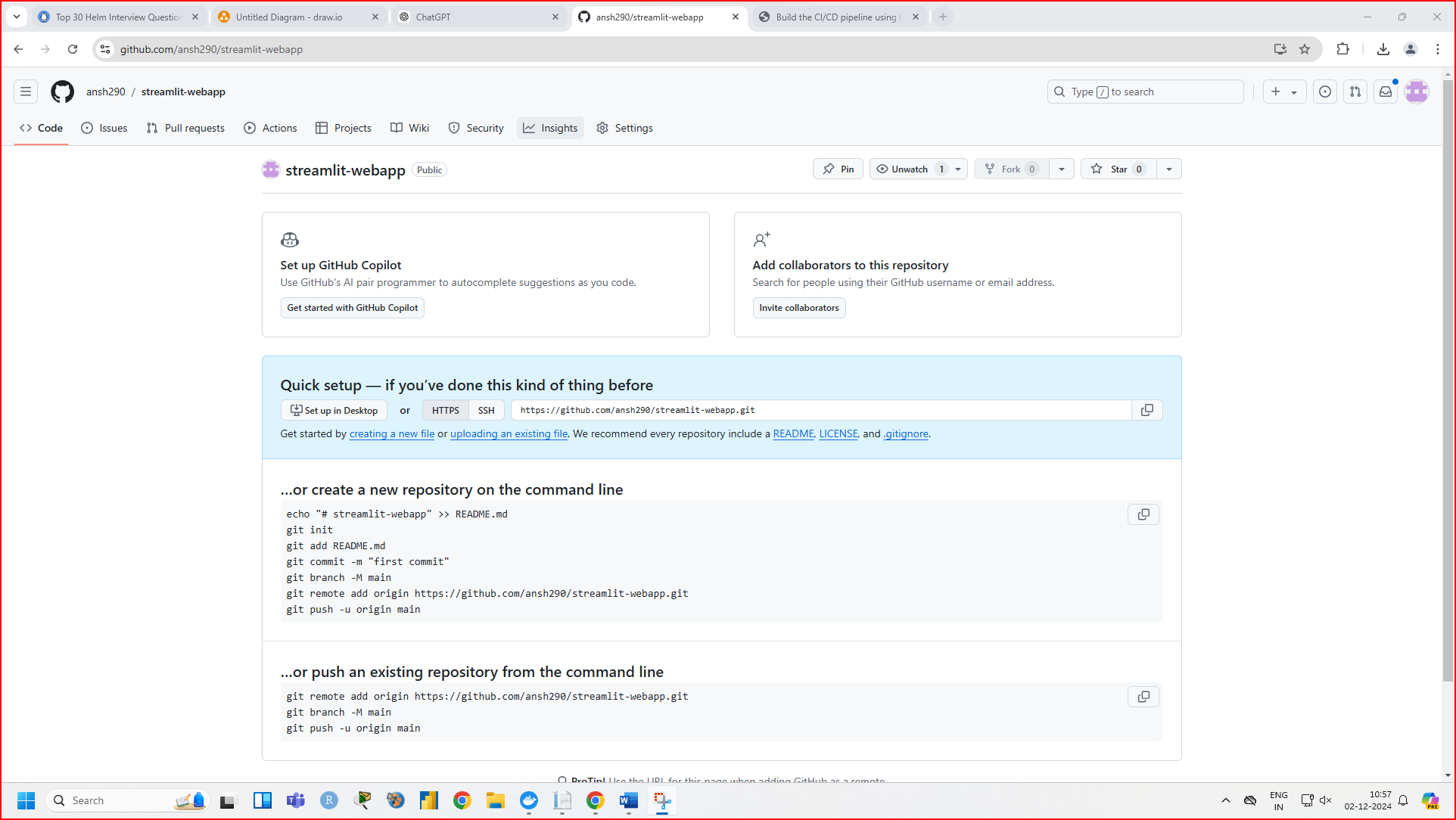
### Explanation of Key Kubernetes Terms:

1. **Pod**: A **Pod** is the smallest and simplest unit in Kubernetes. A pod represents a single instance of a running process in the cluster. It can host one or more containers.
2. **Deployment**: A **Deployment** is a Kubernetes object that manages a set of Pods, ensuring that the specified number of replicas are running at any given time. It also allows easy rolling updates of your application.
3. **Service**: A **Service** exposes a set of Pods to the outside world or within the cluster. It provides a stable IP address and DNS name, ensuring that applications can communicate with each other reliably, even if Pods are recreated.
4. **ReplicaSet**: A **ReplicaSet** ensures that a specified number of identical Pods are running at all times. It is used by a Deployment to manage the Pods.
5. **Horizontal Pod Autoscaler (HPA)**: The **HPA** automatically scales the number of Pods in a Deployment or ReplicaSet based on observed CPU utilization (or other metrics).
6. **LoadBalancer**: A **LoadBalancer** is a type of Kubernetes Service that exposes the application to external traffic. It’s typically used in cloud environments and automatically distributes incoming traffic across multiple Pods.
7. **Rolling Updates**: A **Rolling Update** in Kubernetes is a deployment strategy where new versions of an application are deployed incrementally, minimizing downtime.

B.

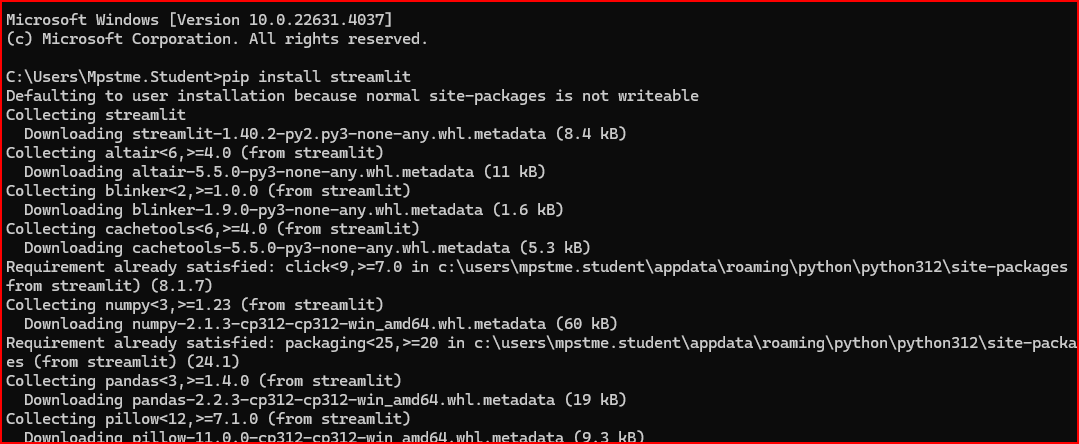
### Step 1: ****Create a GitHub Repository****

1. **Go to GitHub**: Open [GitHub](https://github.com) and log in to your account (or create one if you don't have one yet).
2. **Create a New Repository**:
   * Click on the **+** icon in the top-right corner and select **New repository**.
   * Give it a name (e.g., streamlit-webapp), choose a **Public** or **Private** repository (for a team, you may want to make it private), and add a description if desired.
   * Initialize the repository with a **README** and choose a **.gitignore** for Python (this will ignore unnecessary files like \_\_pycache\_\_).

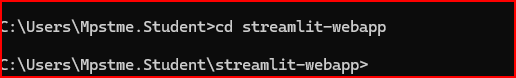


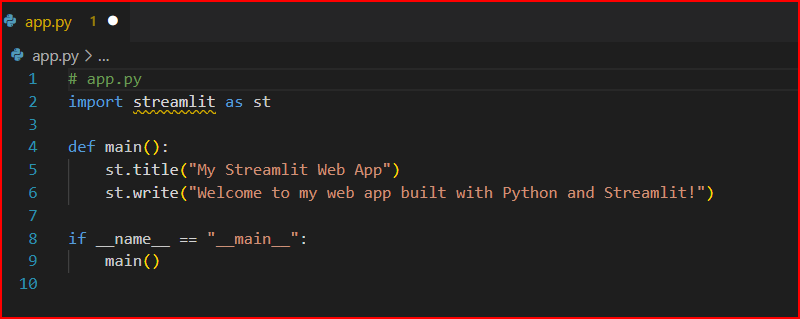
### Step 2: ****Create a Simple Streamlit App****

1. **Create the Streamlit app**: In your project directory, create a Python file called app.py. Here’s a simple example:



**Create a new directory** (folder) for your Streamlit app:





 **import streamlit as st**: This imports the Streamlit library, allowing you to use its functions in your app.

 **st.title("My Streamlit Web App")**: This displays a title at the top of the page.

 **st.write("Welcome to my web app...")**: This writes text to the page (you can display text, tables, charts, and more).

 **main()**: Defines a function to keep the code organized.

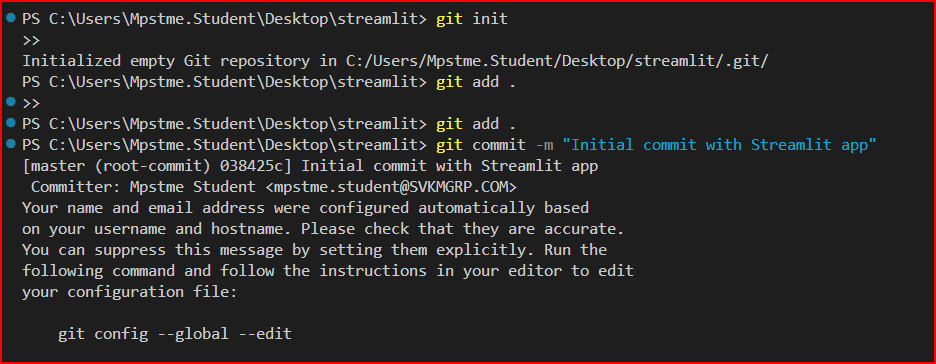
 **if \_\_name\_\_ == "\_\_main\_\_":**: This makes sure the main() function is executed when running the script.



1. Streamlit will start a local development server and give you a link to open the app in your web browser (typically http://localhost:8501).
2. When you open the link, you should see your Streamlit app running with the title "My Streamlit Web App" and the message "Welcome to my web app built with Python and Streamlit!"

### Step 3: ****Push to GitHub****

Once you've created the app.py and requirements.txt files, and confirmed the app is working locally, you're ready to push the project to GitHub.





Now your code is available on GitHub!

### Summary of the Process:

1. **Install Streamlit**: Use pip install streamlit to install the necessary dependencies.
2. **Create a Python file (app.py)**: This is where you write your Streamlit app code.
3. **Run the app locally**: Use streamlit run app.py to launch the app in your browser.
4. **Create a requirements.txt file**: This lists your dependencies for easy installation.
5. **Push the app to GitHub**: Use Git to version control your app and push it to GitHub.

This simple Streamlit app can now be expanded with more features, such as adding widgets, data visualizations, or connecting to databases. Let me know if you need more help with any specific part of the process!

C.

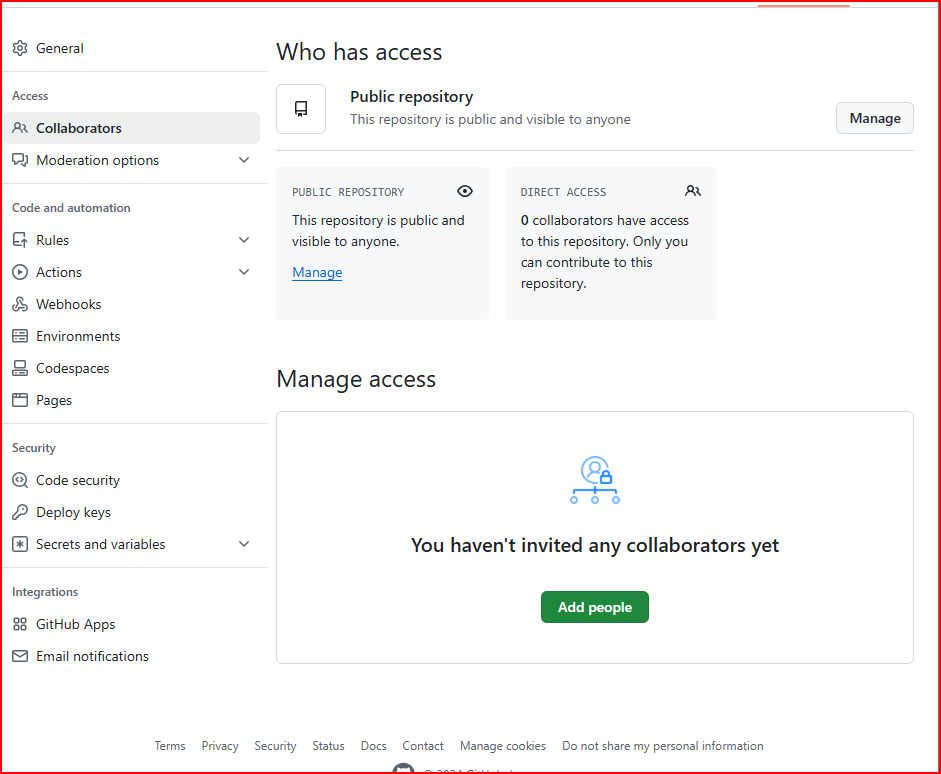
**1. Add Collaborators to Your Repository (Manage Access)**

**Purpose**: You want to ensure only your team members have access to the repository and avoid unauthorized contributors.

**How to Implement**:

1. Go to your **GitHub repository**.
2. Click on the **Settings** tab (at the top right of the repository).
3. In the left sidebar, click on **Manage access**.
4. Click **Invite teams or collaborators**.
5. Enter the **GitHub username** of your team member and click **Add collaborator**.
6. Choose their role (Admin, Write, or Read access).
7. Click **Send invitation**.

Once the collaborator accepts, they can contribute based on their assigned access level.



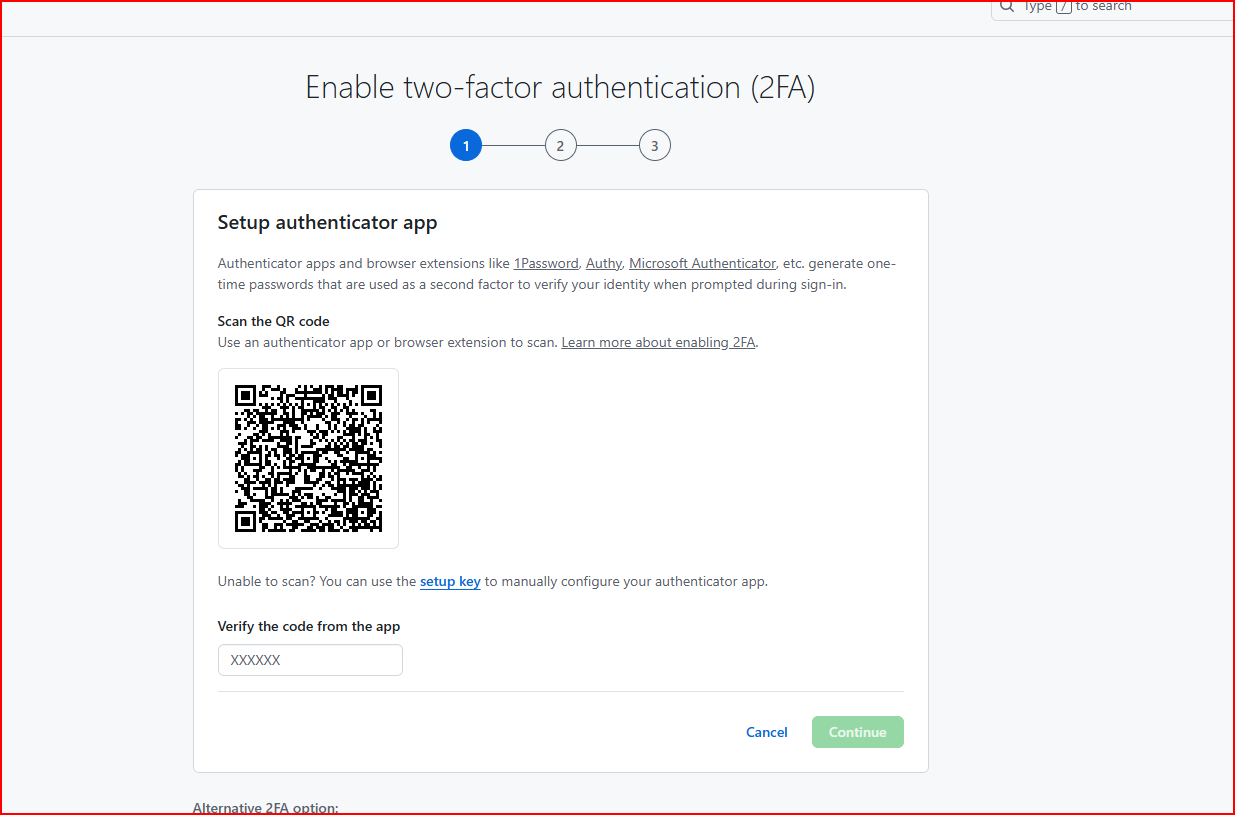
**2. Enable Two-Factor Authentication (2FA)**

**Purpose**: This adds an extra layer of security to your GitHub account to prevent unauthorized access even if a password is compromised.

**How to Implement**:

1. Go to **GitHub** and log in to your account.
2. In the top-right corner, click on your **profile picture** and select **Settings**.
3. In the left sidebar, click on **Security**.
4. Under **Two-factor authentication**, click **Enable two-factor authentication**.
5. You will be asked to choose between **SMS** (text messages) or an **Authentication app** (like Google Authenticator or Authy).
6. Follow the prompts to complete the setup.

Once enabled, each time you log into GitHub, you’ll need to enter a code sent to your phone or generated by your app.

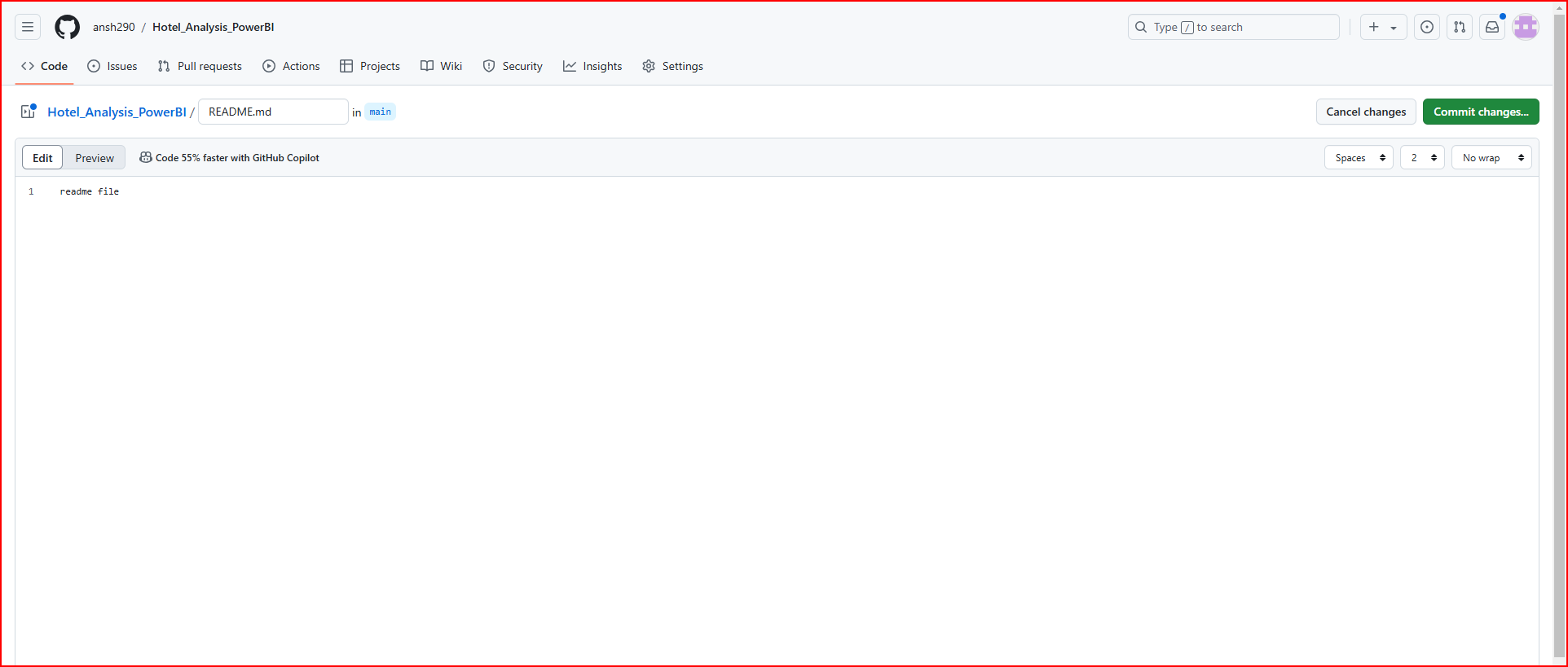


### ****3. Set Up a README File for Your Repository****

**Purpose**: The README file is the first thing people see when they visit your repository. It’s a simple way to explain what your project is about and how others can collaborate. This helps new team members or contributors understand the project quickly.

#### How to Implement:

1. **Go to your GitHub repository**.
2. In the **root directory** of the repository, create a new file called **README.md**.
3. Inside the README.md file, add basic information like:
   * **Project title** and description.
   * How to **install** and **run** the project (e.g., with pip install -r requirements.txt).
   * **How to contribute** if other team members want to help.
   * **Link to the deployed app** or **live demo** (if available).



### ****4. Create and Use Labels for Issues and Pull Requests****

**Purpose**: Labels help you categorize and prioritize issues and pull requests (PRs) in a project. For example, you can tag issues as bug, enhancement, or help wanted. This makes it easier for team members to know which tasks are critical or need attention.

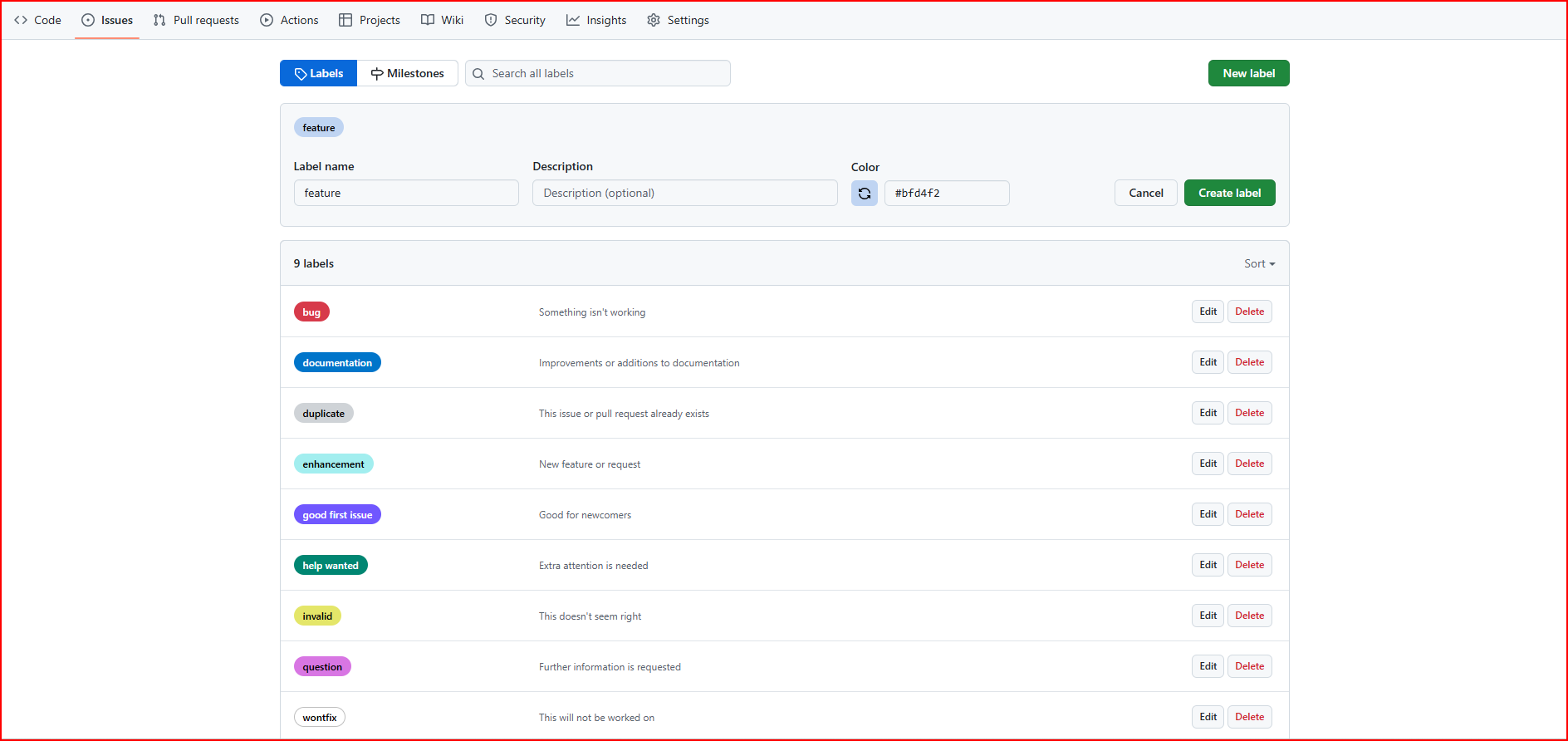
#### How to Implement:

1. Go to your **GitHub repository**.
2. Click on the **Issues** tab at the top of the page.
3. Click on the **Labels** button on the right sidebar.
4. Click **New label** to create a new one.
5. Give the label a name, like bug, feature request, or in-progress.
   * **Color**: Choose any color that helps you easily spot the label.
   * Click **Create label**.

Once the labels are created, you can assign them to issues or pull requests:

1. Open an **Issue** or **Pull Request**.
2. On the right side, find the **Labels** section.
3. Click **Edit labels** and select the appropriate label (e.g., bug, enhancement, etc.).

This will help your team quickly understand the status and type of the issue or PR.



### 5. ****Use the "Projects" Tab for Basic Project Management****

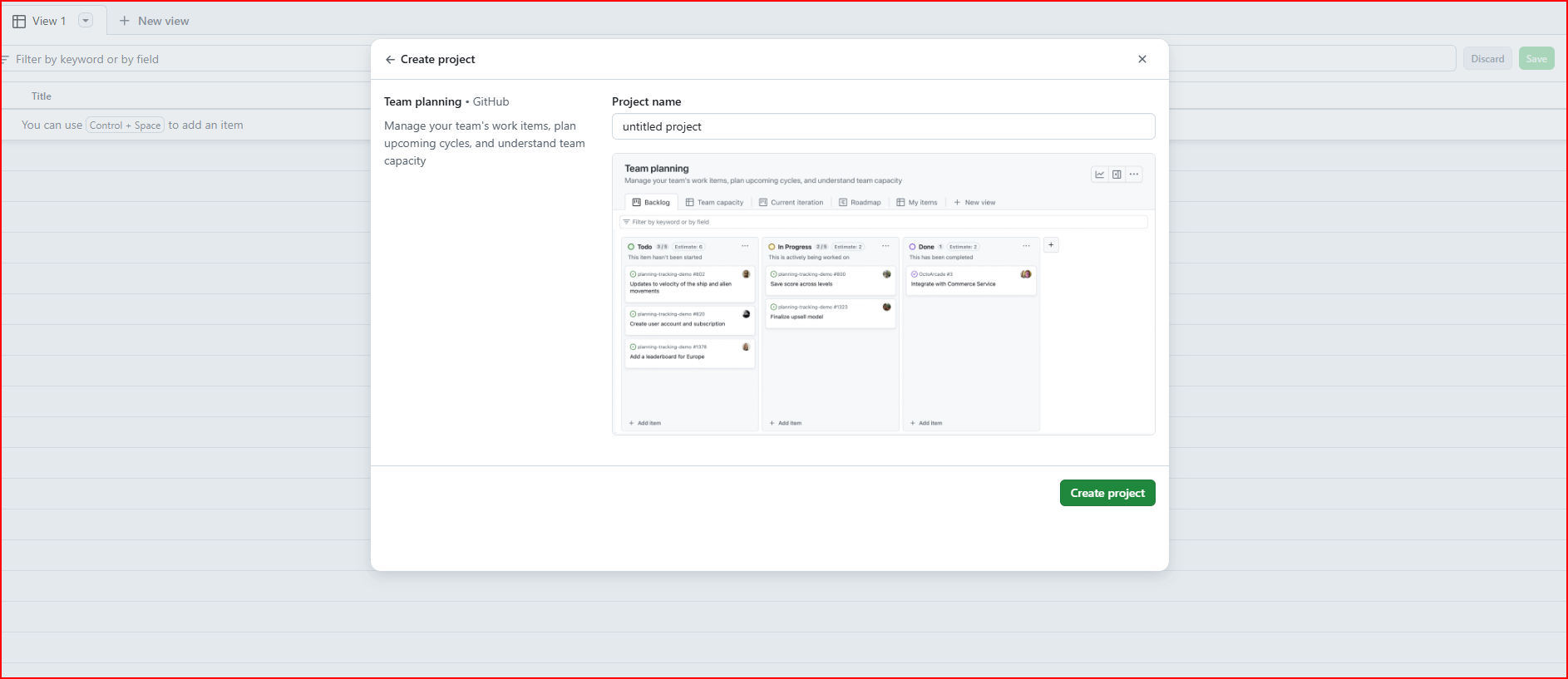
**Purpose**: The **Projects** tab allows you to organize your tasks using a **Kanban-style board**. It's a great way to track the progress of different features or bugs. It’s simple and doesn’t require complex configurations, making it perfect for small teams or solo developers.

#### How to Implement:

1. Go to your **GitHub repository**.
2. Click on the **Projects** tab at the top of the page.
3. Click **Create a project** and choose **Basic Kanban**.
4. Name your project (e.g., Website Features or Bug Tracking).
5. Click **Create project**.

You’ll now see columns like **To do**, **In Progress**, and **Done**. You can create tasks by adding issues directly to the project or creating new tasks.

* **Add an Issue** to the board by clicking **+ Add cards** under the columns.
* Drag tasks between columns to track progress as they move from To Do to In Progress and finally to Done.



d.

To showcase **Docker Desktop** and **Docker Hub** functionality to your project team, I’ll demonstrate how to containerize a **Streamlit-based Python website** with a basic **Hello World** example. I’ll walk you through all the steps, including Docker commands, building the Docker image, pushing it to Docker Hub, and using **Streamlit app secrets** for managing sensitive information securely.

**Overview of Steps:**

1. **Create a Python and Streamlit "Hello World" Application.**
2. **Dockerize the Streamlit Application.**
3. **Build and Run the Docker Image Locally using Docker Desktop.**
4. **Push the Docker Image to Docker Hub.**
5. **Use App Secrets in Streamlit for Managing Sensitive Information.**

Hello\_world.py

import streamlit as st

st.title("Using App Secrets")

secret\_key = st.secrets["api\_keys"]["my\_secret\_key"]

st.write(f"The secret key is: {secret\_key}")

DockerFile

# Use the official Python image as the base

FROM python:3.10-slim

# Set the working directory

WORKDIR /app

# Copy the application files

COPY hello\_world.py /app/

COPY requirements.txt /app/

COPY secrets.toml /app/.streamlit/

# Install dependencies

RUN pip install --no-cache-dir -r requirements.txt

# Expose Streamlit's default port

EXPOSE 8501

# Command to run the Streamlit app

CMD ["streamlit", "run", "hello\_world.py", "--server.port=8501", "--server.enableCORS=false"]

Secret.toml

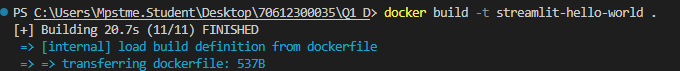
**[api\_keys]**

**my\_secret\_key = "supersecretvalue"**

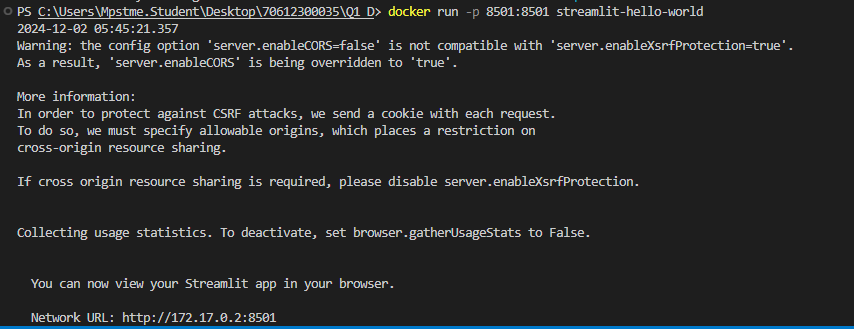
**Commands**

* **docker build -t streamlit-hello-world .**

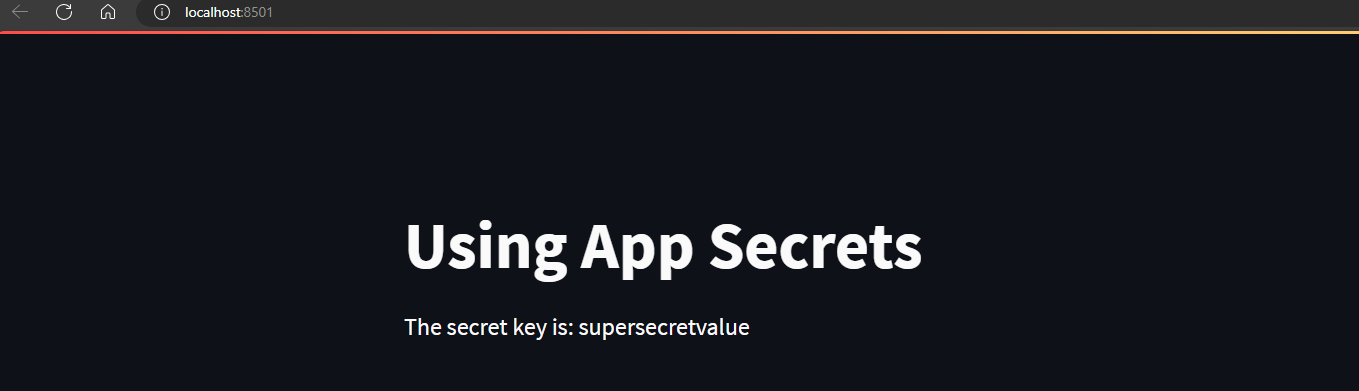
**Output**

****

* **docker run -p 8501:8501 streamlit-hello-world**

****

Output

****