

# **Automated Deep-Learning based Image Captioning System with Dev-Ops Driven Deployment**

**Software Production Engineering - Final Project**  
**IIITB M.Tech 2nd Sem, 2025**

**Aman Bahuguna**  
**(MT2024018)**

**Soumik Pal**  
**(MT2024153)**

DevOps is invaluable in today's IT topology, because it bridges the gap between software development and IT operations, enabling faster and more reliable delivery of applications and services. By fostering collaboration, automating repetitive tasks like testing, deployment, and infrastructure management, DevOps reduces errors and accelerates release cycles. This leads to improved product quality, quicker feedback loops, and greater agility in responding to changing business needs, ultimately driving better customer satisfaction and competitive advantage.

In this project, we design and implement a comprehensive DevOps framework to fully automate the Software Development Life Cycle (SDLC)

using cutting-edge DevOps tools. Focusing on the specialized domain of Machine Learning, we seamlessly integrate DevOps principles to unlock its benefits—thus pioneering a robust and efficient MLOps pipeline.

## 1. ML Component :

Problem Statement :

Given an image, generate a caption for it which would serve as the textual description of the image. This task combines techniques from both Computer Vision and Natural Language Processing, making it a multidisciplinary challenge.

Model Architecture chosen :

Typically, image captioning models follow an encoder-decoder architecture—where the image is first transformed into a meaningful feature representation by the encoder, and then the decoder translates this representation into a coherent descriptive sentence.

In our project, we have used a DenseNet CNN to extract image features and an LSTM to generate text by combining image and word embeddings.

Dataset Used : <https://www.kaggle.com/datasets/adityajn105/flickr8k>

We used a subset of the standard Flickr dataset, called Flickr8K that has ~8K images and 5 captions for each image, creating around 40K image, caption pairs. This was split into a training dataset, validation dataset and testing dataset (80% - 10% - 10% split).

## Preprocessing :

To make the training, validation and testing process quicker, the feature extraction of all 8k images, using DenseNet201 model from tensorflow library, were done beforehand and stored as .pkl artifact in models folder. This was later loaded and used during training, validation and testing as a simple lookup.

The caption text preprocessing involved cleaning and normalizing sentences by lowercasing, removing special characters, extra spaces, and single letters, and adding start and end tags for model training.

## Training the Model :

The model was trained over 12 epochs using a carefully selected learning rate and batch size to optimize performance. To ensure efficient training and prevent overfitting, model checkpointing was employed to save the best-performing model based on validation loss, allowing for recovery and reuse of the most effective weights. Additionally, early stopping was implemented with a patience of 3, meaning training would halt if the validation loss did not improve for three consecutive epochs—thereby saving time and computational resources while maintaining model generalization.

The model and tokenizer used were saved in the local models directory to be later reused during testing and inference.

## Testing the Model :

The saved model was loaded and tested on the test dataset to evaluate BLEU score. The BLEU score evaluates the quality of generated captions by measuring their similarity to reference captions based on overlapping n-grams.

## Inference and Frontend App :

The inference script has no main method and instead exposes its method as an API which is called from app.py, a streamlit based frontend where a user can submit their image and the api call will return the generated caption which is then displayed to the user.

## Project Structure based on ML Components :

### Scripts :

- extract\_features.py
- train.py
- test.py
- infer.py

### Dataset :

- data/flickr/captions.txt
- data/flickr/images

### Model and Artifacts :

- Features.pkl
- Tokenizer.pkl
- model.h5

```
(ml-devops-env) soumik@soumik-VirtualBox:~/SPE_Final_Project$ ls extract_features.py train.py test.py infer.py app.py data/ models/
app.py  extract_features.py  infer.py  test.py  train.py

data/:
flickr

models/:
features.pkl  model.h5  tokenizer.pkl
```

The entire Machine Learning project was developed inside a Virtual Environment to ensure all dependencies could be replicated consistently.

## 2. Code versioning - Git, Github

Git is a distributed version control system for tracking code changes, while GitHub is a cloud-based platform for hosting and collaborating on Git repositories.

In our project we are versioning all the scripts used in the ML pipeline such as extract\_features.py, train.py, test.py, infer.py & app.py along with the certain artifacts that are saved during training which will be later needed during testing and inference such as the tokenizer fit on training data captions for word to embedding mappings.

Along with these, other scripts used later in the pipeline such as Jenkinsfile, Dockerfile, Ansible playbooks, inventory file, Ansible Roles, Kubernetes Manifests to deploy the application, enable logging and monitoring and HPA are also versioned in this repository.

Final project repository :

The screenshot shows a GitHub repository page for 'Soumik1410/SPE\_Final\_Project'. The main area displays a list of 58 commits over the past 2 hours, with the most recent update being 'Update fluent-bit-configmap.yaml' 3 hours ago. The commits include updates to various files like 'files', 'models', 'roles/k8s\_deploy', '.dockerignore', '.gitignore', 'Dockerfile', 'HPAplaybook.yaml', 'Jenkinsfile', 'app.py', 'deployment.yaml', 'extract\_features.py', 'infer.py', 'inventory.ini', 'logging-playbook.yaml', 'playbook.yaml', 'service.yaml', 'test.py', and 'train.py'. The repository has 0 stars, 1 watching, and 0 forks. It also shows sections for 'provided.', 'Activity', 'Releases', 'Packages', and 'Languages' (Python 96.3%, Dockerfile 3.7%).

Other large files present in the local development virtual environment such as the dataset and the trained models were not pushed to the Github repository by maintaining a .gitignore file

```
(ml-devops-env) soumik@soumik-VirtualBox:~/SPE_Final_Project$ cat .gitignore
__pycache__/
*.pyc
*.log
*.h5
data/flickr/Images/
data/flickr/captions.txt
mlruns/
.env

(ml-devops-env) soumik@soumik-VirtualBox:~/SPE_Final_Project$
```

Personal Access Tokens were created in Github for pushing commits from the local repository to the remote Github repository.

The screenshot shows the GitHub developer settings page at [github.com/settings/tokens](https://github.com/settings/tokens). The left sidebar has sections for GitHub Apps, OAuth Apps, and Personal access tokens (the latter is selected). The main area displays a table of personal access tokens:

Token Name	Scope	Last Used	Action
Second git push token	repo, workflow	Within the last week	Delete
Git push and checkout token	admin:repo_hook, repo, workflow	Within the last week	Delete

A note at the bottom explains that personal access tokens function like OAuth tokens and can be used for HTTPS or API authentication.

### 3. MLFlow - Experiment Tracking and Hyperparameter Versioning

When experimenting with different model architectures and different hyperparameters, it helps to track every experiment, the architecture & the different hyperparameters used for training, the metrics evaluated etc. so that the same model and results can be reproduced consistently and different metrics of different experiments can be easily compared.

MLFlow provides an easy way to do this, saving all these to a central server and provides a UI to view these experiments and corresponding saved parameters & metrics later. MLFlow also provides a registry to save and version models, but we opted to do this manually with checkpointing.

To efficiently track training experiments, the training loop was encapsulated within a code block that initiated an MLflow run each time a model was trained. During this process, key parameters such as batch size, optimizer, embedding dimension, LSTM units, dropout rate, and the number of training epochs were logged. At the end of each epoch, both the training loss and validation loss were recorded. Once the training was completed, the training loss versus validation loss curve was saved as an MLflow artifact, providing a visual representation of the model's performance throughout the training process.

```
# MLflow logging
with mlflow.start_run():
    # Log hyperparameters
    mlflow.log_param("batch_size", 64)
    mlflow.log_param("optimizer", "adam")
    mlflow.log_param("embedding_dim", 256)
    mlflow.log_param("lstm_units", 256)
    mlflow.log_param("dropout", 0.5)
    mlflow.log_param("epochs", 10)

    # Train and capture history
    history = caption_model.fit(
        train_gen,
        validation_data=val_gen,
        epochs=10,
        callbacks=[checkpoint, earlystop, reduce_lr],
        verbose=1
    )

    # Log metrics per epoch
    for epoch in range(len(history.history['loss'])):
        mlflow.log_metric("loss", history.history['loss'][epoch], step=epoch)
        mlflow.log_metric("val_loss", history.history['val_loss'][epoch], step=epoch)
```

```

# Log metrics per epoch
for epoch in range(len(history.history['loss'])):
    mlflow.log_metric("loss", history.history['loss'][epoch], step=epoch)
    mlflow.log_metric("val_loss", history.history['val_loss'][epoch], step=epoch)

# Save training curve
plt.figure()
plt.plot(history.history['loss'], label='train_loss')
plt.plot(history.history['val_loss'], label='val_loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Training Curve')
plot_path = os.path.join(model_dir, 'training_curve.png')
plt.savefig(plot_path)
mlflow.log_artifact(plot_path)

```

Similarly, in the test script, the BLEU score evaluated on the trained model loaded in the script is logged along with the length of the test dataset & the complete set of predictions made on the test set.

To view these saved metrics and parameters, we activate the UI with the command : mlflow ui

```

(ml-devops-env) soumik@soumik-VirtualBox:~/SPE_Final_Project$ mlflow ui
[2025-05-21 16:36:43 +0530] [2847600] [INFO] Starting gunicorn 23.0.0
[2025-05-21 16:36:43 +0530] [2847600] [INFO] Listening at: http://127.0.0.1:5000 (2847600)
[2025-05-21 16:36:43 +0530] [2847600] [INFO] Using worker: sync
[2025-05-21 16:36:43 +0530] [2847625] [INFO] Booting worker with pid: 2847625
[2025-05-21 16:36:43 +0530] [2847626] [INFO] Booting worker with pid: 2847626
[2025-05-21 16:36:44 +0530] [2847627] [INFO] Booting worker with pid: 2847627
[2025-05-21 16:36:44 +0530] [2847642] [INFO] Booting worker with pid: 2847642

```

This boots up the UI on port 5000 on localhost. We can open this in a browser to view the different saved runs and their corresponding parameters and metrics.

← → ⌂ ⌂ http://localhost:5000/#/experiments/283014949511983756?searchFilter=&orderByKey=attribute ☆

**mlflow** 2.22.0 Experiments Models Prompts GitHub Docs

### Experiments

Search experiments

- Default
- Image Captioning Test Eval...
- Image Captioning

### Image Captioning

Provide Feedback Add Description Share

Runs Evaluation Experimental Traces

Metrics: metrics.rmse < 1 and params.model = "tree"

Time created

New run

State: Active Datasets Sort: Created Columns Group by

Run Name	Created	Dataset	Duration	Source
casual-shrimp-178	3 hours ago	-	2.4h	train.py
resilient-bat-106	16 hours ago	-	23.7min	train.py
bemused-rook-822	17 hours ago	-	25.5min	train.py
orderly-fish-886	1 day ago	-	3.5h	train.py
victorious-flinch-453	1 day ago	-		train.py
auspicious-zebra-655	1 day ago	-	15.4min	train.py
adaptable-lamb-540	1 day ago	-	14.3min	train.py
useful-toad-302	1 day ago	-	2.2h	train.py
polite-flea-521	1 day ago	-	13.2min	train.py
loud-colt-35	1 day ago	-	11.4min	train.py

14 matching runs

← → ⌂ ⌂ http://localhost:5000/#/experiments/283014949511983756/runs/99c2938a569c4ece9565ecc0fb0 ☆

**mlflow** 2.22.0 Experiments Models Prompts GitHub Docs

Image Captioning >

### orderly-fish-886

⋮

Overview Model metrics System metrics Traces Artifacts

Registered models	—
Registered prompts	—

Parameters (6)

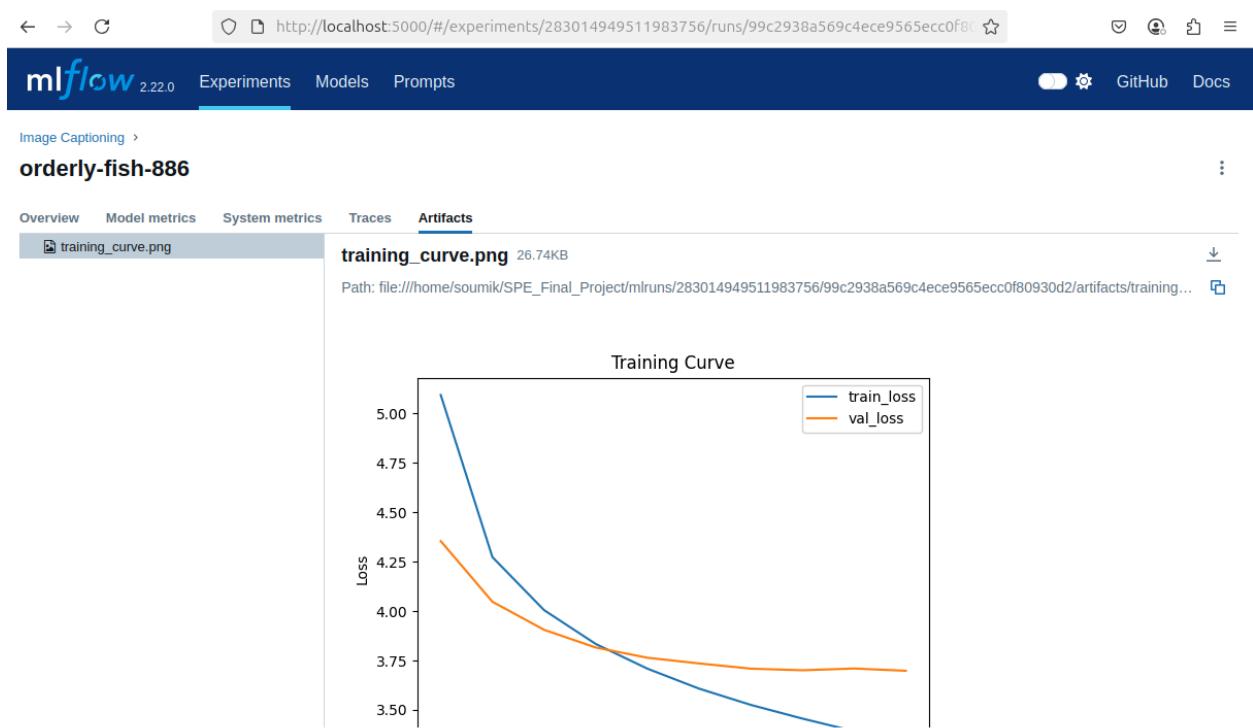
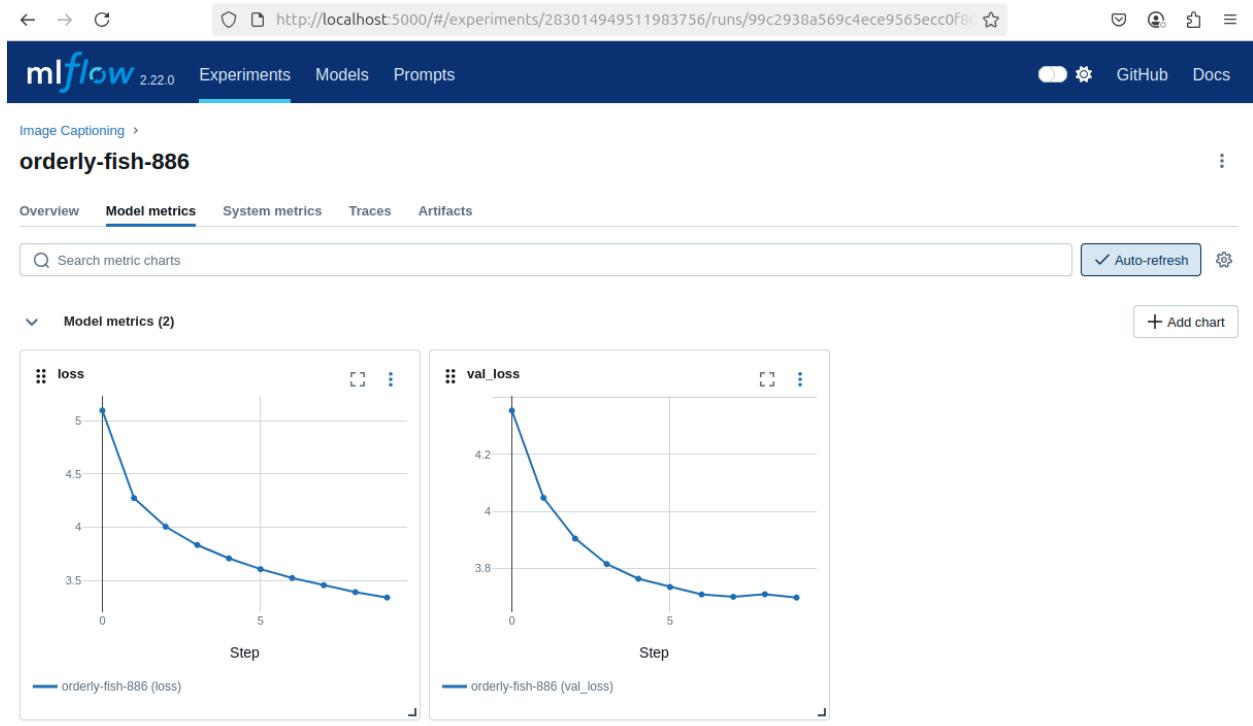
Search parameters

Parameter	Value
dropout	0.5
lstm_units	256
epochs	10
batch_size	64
embedding_dim	256
optimizer	adam

Metrics (2)

Search metrics

Metric	Value
val_loss	3.6981589794158936
loss	3.34120774269104



← → ⌂ http://localhost:5000/#/experiments/740161503892734476?searchFilter=&orderByKey=attribute ☆

mlflow 2.22.0 Experiments Models Prompts GitHub Docs

### Experiments

Search experiments

- Default
- Image Captioning Test Evaluation
- Image Captioning

### Image Captioning Test Evaluation

Provide Feedback ⓘ Add Description Share

Runs Evaluation Experimental Traces

Runs:  metrics.rmse < 1 and params.model = "tree" ⏷ Time created ⏷ ⏷ + New run

State: Active Datasets Sort: Created Columns Group by

Run Name	Created	Dataset	Duration	Source
test_evaluation	1 hour ago	-		test.py
test_evaluation	22 hours ago	-	1.3h	test.py
test_evaluation	1 day ago	-	1.4h	test.py
test_evaluation	1 day ago	-		test.py
test_evaluation	1 day ago	-	1.1h	test.py

5 matching runs

← → ⌂ http://localhost:5000/#/experiments/740161503892734476/runs/ad90bb4dd0004975a0e72c1df1☆

mlflow 2.22.0 Experiments Models Prompts GitHub Docs

Image Captioning Test Evaluation >

### test\_evaluation

⋮

Overview Model metrics System metrics Traces Artifacts

Registered models	—
Registered prompts	—

Parameters (1)

Search parameters

Parameter	Value
test_samples	4050

Metrics (1)

Search metrics

Metric	Value
average_bleu	0.041442931007944314

Image Captioning Test Evaluation >  
**test\_evaluation**

Overview Model metrics System metrics Traces Artifacts

**test\_predictions.csv** 584.94KB

Path: file:///home/soumik/SPE\_Final\_Project/mlruns/740161503892734476/ad90bb4dd0004975a0e72c1df1558517/artifacts/test\_pr...

Previewing the first 500 rows

image	caption	predicted_caption
436015762_8d0bae90c3.jpg	startseq man prepares to enter the red buil...	startseq man is standing on the street end...
436015762_8d0bae90c3.jpg	startseq man walking around the corner of...	startseq man is standing on the street end...
436015762_8d0bae90c3.jpg	startseq man walks past red building with ...	startseq man is standing on the street end...
436015762_8d0bae90c3.jpg	startseq man walks under building with lar...	startseq man is standing on the street end...
436015762_8d0bae90c3.jpg	startseq person walking by red building wi...	startseq man is standing on the street end...
436393371_822ee70952.jpg	startseq brown doberman is outside with s...	startseq dog is running through the grass ...

## 4. Jenkins - Automation CI/CD server

Jenkins is an open-source automation server widely used in DevOps for continuous integration and continuous delivery (CI/CD). It facilitates the automation of software development processes by allowing developers to automatically build, test, and deploy code changes.

In our project, we developed a Jenkins pipeline that is triggered by webhooks sent by a SCM service (Github), and it performs the following actions automatically :

1. Checkout the latest code from the repository
2. Train the model

3. Execute the automated test script to evaluate the trained model
4. Generate a list of the environment's dependencies
5. Prune the list to include only the essential dependencies
6. Dockerize the application
7. Push the built image to Docker Hub
8. Trigger an Ansible playbook to pull the image and deploy it on the Kubernetes cluster

For this, we will go over the different stages of the pipeline as and when we discuss the other DevOps tools used. For now, we will show how the pipeline was set up to trigger from Github webhooks and the first 3 stages till automated testing.

The screenshot shows a Jenkins configuration page for a project named "SPE\_Final\_Project". The top navigation bar includes links for "Dashboard", "SPE\_Final\_Project", and "Configuration". The main area is titled "Configure" and contains several sections:

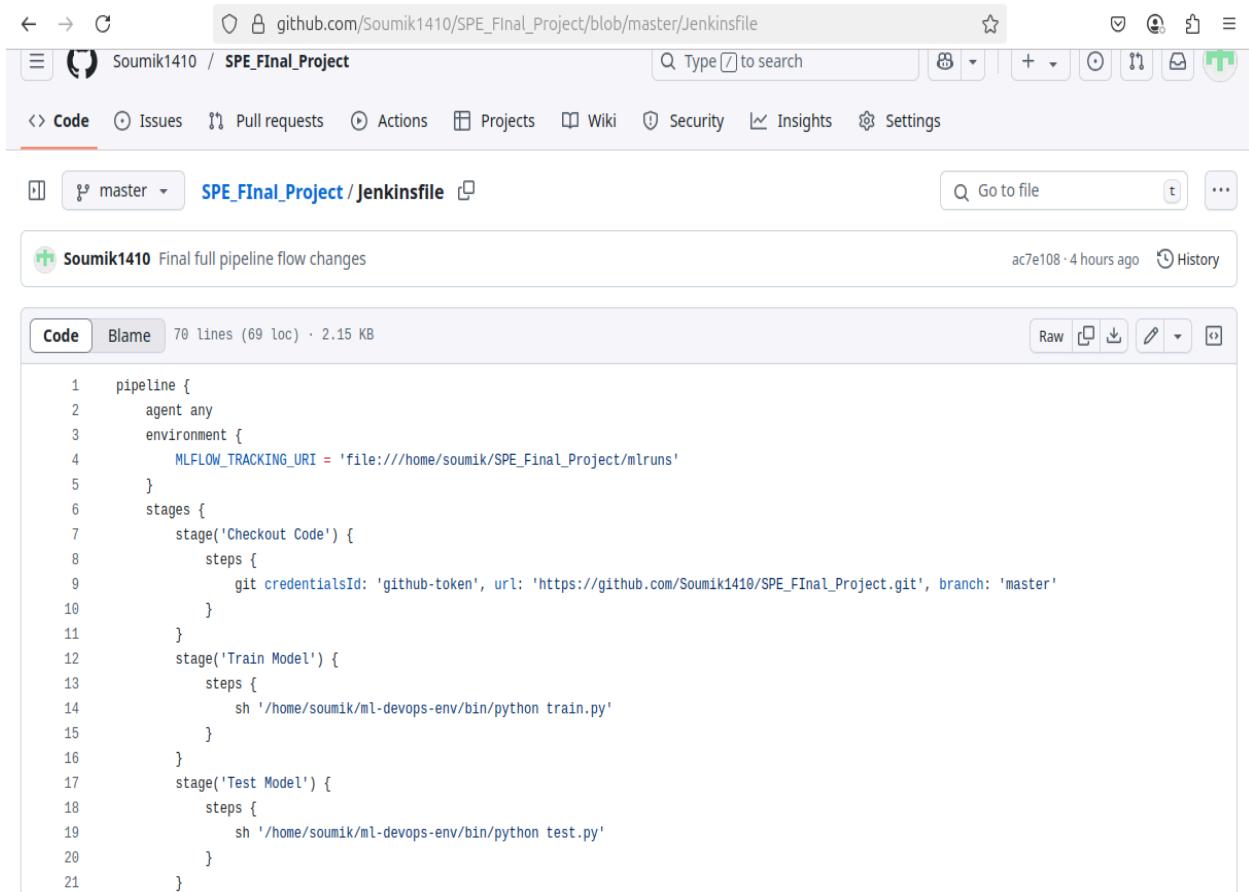
- General**: Contains checkboxes for "Preserve stashes from completed builds", "This project is parameterized", and "Throttle builds".
- Triggers**: A section titled "Set up automated actions that start your build based on specific events, like code changes or scheduled times." It includes checkboxes for "Build after other projects are built", "Build periodically", "GitHub hook trigger for GITScm polling" (which is checked), "Poll SCM", and "Trigger builds remotely (e.g., from scripts)".

The screenshot shows the Jenkins Pipeline configuration page for a job named 'SPE\_Final\_Project'. The left sidebar has tabs for General, Triggers, Pipeline (which is selected), and Advanced. The main area is titled 'Definition' and shows 'Pipeline script from SCM'. Under 'SCM', it is set to 'Git'. The 'Repositories' section contains a single repository with the URL `https://github.com/Soumik1410/SPE_Flnal_Project/` and credentials 'Soumik1410/\*\*\*\*\*\*\*\*'. There is also an 'Advanced' dropdown.

The screenshot shows the Jenkins Pipeline configuration page for the same job. The left sidebar has tabs for General, Triggers, Pipeline (selected), and Advanced. The main area is titled 'Branches to build' and shows a branch specifier `*/master`. It includes an 'Add Branch' button, a 'Repository browser' set to '(Auto)', and an 'Additional Behaviours' section with an 'Add' button. Below these are sections for 'Script Path' (set to 'Jenkinsfile') and 'Lightweight checkout' (checkbox checked).

This creates a new Jenkins pipeline job that is triggered by webhooks sent from GitHub. It specifies that the source of the pipeline script resides in a Git repository, with details such as the repository URL, credentials for access, the branch where the pipeline script is located, and the filename of

the script—in our case, Jenkinsfile. The Jenkinsfile itself defines the pipeline using declarative syntax, outlining each stage and the steps required within those stages to execute the tasks mentioned earlier.



The screenshot shows a GitHub repository page for 'SPE\_FInal\_Project'. The 'Code' tab is selected, displaying the Jenkinsfile. The code defines a pipeline with three stages: 'Checkout Code', 'Train Model', and 'Test Model'. Each stage has specific steps defined, such as cloning the repository and running Python scripts. The Jenkinsfile also includes environment variables and a tracking URI.

```
1 pipeline {
2     agent any
3     environment {
4         MLFLOW_TRACKING_URI = 'file:///home/soumik/SPE_Final_Project/mlruns'
5     }
6     stages {
7         stage('Checkout Code') {
8             steps {
9                 git credentialsId: 'github-token', url: 'https://github.com/Soumik1410/SPE_FInal_Project.git', branch: 'master'
10            }
11        }
12        stage('Train Model') {
13            steps {
14                sh '/home/soumik/ml-devops-env/bin/python train.py'
15            }
16        }
17        stage('Test Model') {
18            steps {
19                sh '/home/soumik/ml-devops-env/bin/python test.py'
20            }
21        }
}
```

For the first step i.e. checking out the latest code from a Github repository, the credentials needed were added to Jenkins credential manager. A Personal Access Token was created in Github, and that was stored in Jenkins as a username with password passing the PAT in place of the user's actual password.

The screenshot shows the Jenkins Global credentials (unrestricted) page. At the top, there is a navigation bar with links to Dashboard, Manage Jenkins, Credentials, System, and Global credentials (unrestricted). Below the navigation bar, the title "Global credentials (unrestricted)" is displayed, along with a "Add Credentials" button. A table lists two credentials:

ID	Name	Kind	Description
dockerhub-creds	soumik1410/*****	Username with password	
github-token	Soumik1410/*****	Username with password	

At the bottom left, there is a "Icon:" label followed by three size options: S, M, and L, with M selected.

For the training and test stages, we needed all the libraries that were installed in the virtual environment used for development and as such first that environment is activated by using that specific environment's python interpreter instead of the system wise python interpreter located at /usr/bin/python.

MLFlow also normally stores all metrics and parameters and artifacts in a local folder 'mlruns' relative to the execution context of the train and test scripts. However, for these runs to show in the UI, they need to store them in a single location specified by the MLFLOW\_TRACKING\_URI environment variable.

Thus, the same is set in jenkins to ensure that different mlruns folders are not used to store the artifacts for different builds of the job, but instead they are all stored in the single specified location. Jenkins user needs to be given write permissions to the folder specified by MLFLOW\_TRACKING\_URI and all its parent folders.

Now to trigger the Jenkins pipeline whenever a push was made to the Github repository, a webhook was set up in Github that is delivered to Jenkins everytime a push is made.

The screenshot shows the GitHub settings page for a repository named 'SPE\_FInal\_Project'. The 'Webhooks' tab is selected. On the left, there's a sidebar with options like General, Access, Collaborators, Moderation options, Code and automation, Branches, and Tags. A single webhook is listed with the URL 'https://a7d7-49-207-60-154.ngrok... (push)'. Below it, a message says 'Last delivery was successful.' with 'Edit' and 'Delete' buttons.

The screenshot shows the 'Webhooks / Manage webhook' page. The sidebar has a 'Webhooks' section highlighted with a green preview button. The main area shows a table of recent deliveries under the 'Recent Deliveries' tab. Each row contains a green checkmark, a small icon, a unique ID, the event type ('push'), and the timestamp ('2025-05-21 18:15:34').

Delivery	Event	Timestamp
7c5fa128-3641-11f0-958a-18a090d96c2a	push	2025-05-21 18:15:34
62f6bc26-3641-11f0-9296-d1f4c411eddf	push	2025-05-21 18:14:52
f25a4c36-3617-11f0-8ec9-8a7299c44bca	push	2025-05-21 13:18:13
63bfccba-3614-11f0-9e36-e5bd97b8427a	push	2025-05-21 12:52:46
194ffa2a-3613-11f0-9c97-5094f53f0426	push	2025-05-21 12:43:31
097b626a-360e-11f0-8fe9-d8125ad2f405	push	2025-05-21 12:07:17
03dc5ab4-360c-11f0-8256-b9506a203c9c	push	2025-05-21 11:52:49
49276836-360a-11f0-896b-f2604d1ea0dd	push	2025-05-21 11:40:26

To enable the delivery of the webhook, the Jenkins instance running on my localhost needed to be exposed to the internet through a secure HTTP tunnel. This was achieved by using ngrok, which exposes port 8080 and generates a public URL. This <https://<ngrok URL>/github-webhook>, serves as the endpoint where the webhooks are delivered.

## 4. Dockerizing the Application :

The next step in the project is to build an image that includes my inference script and the Streamlit-based frontend (defined in [app.py](#)). This image will be built on top of a base Python image, incorporating all necessary dependencies, the trained model, and any other saved artifacts required for deployment.

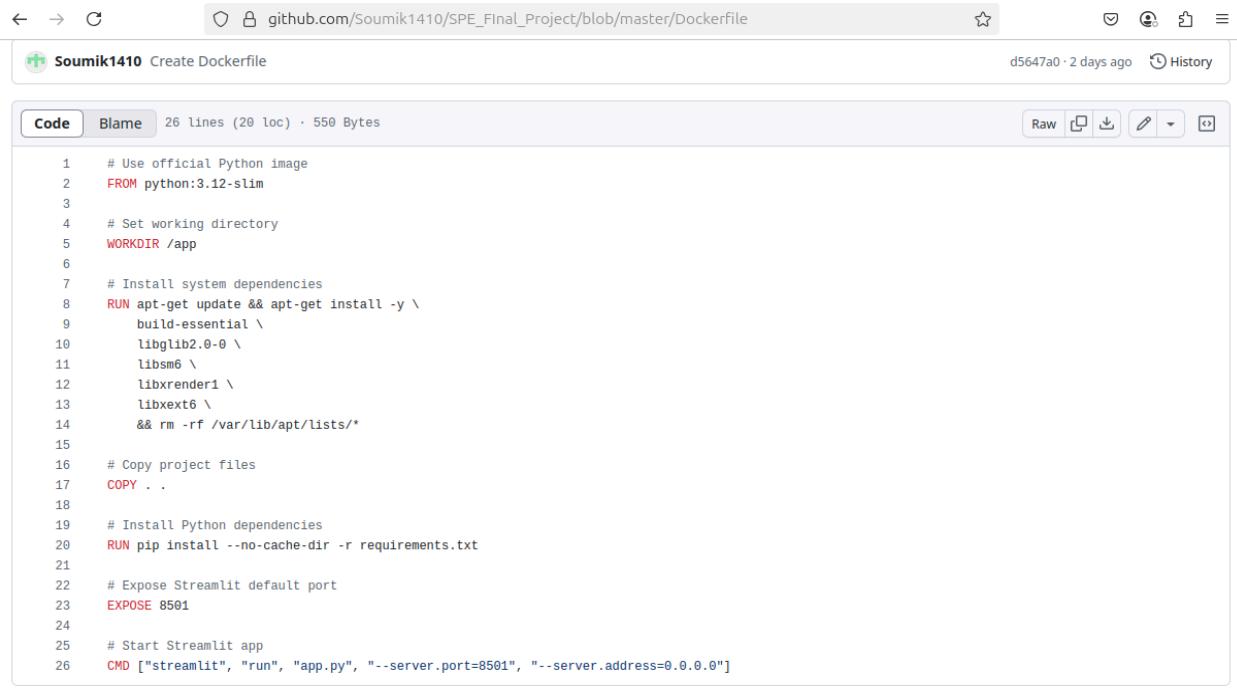
The next stage added to the Jenkins pipeline focuses on generating a requirements.txt file that lists all the environment's dependencies. However, since the environment is quite bloated due to the presence of training and testing scripts in the same repository, we need to prune the dependencies to include only the essential ones required to run app.py and infer.py. To achieve this, we added an additional stage in Jenkins to filter out all unnecessary dependencies from requirements.txt, leaving only Streamlit, TensorFlow, NumPy, Pandas, and their transitive dependencies. This helps reduce the overall size of the image.



A screenshot of a GitHub code editor showing the Jenkinsfile for the SPE\_Final\_Project. The file contains Jenkins pipeline code with two stages: 'Export Requirements' and 'Clean Requirements'. The 'Export Requirements' stage runs a shell command to activate a virtual environment and pip freeze into requirements.txt. The 'Clean Requirements' stage uses sed to remove specific dependency versions (TensorFlow 2.19.0, Streamlit 1.45.1, pillow 11.0.0, numpy 1.1.28) from requirements.txt, keeping only the essential packages.

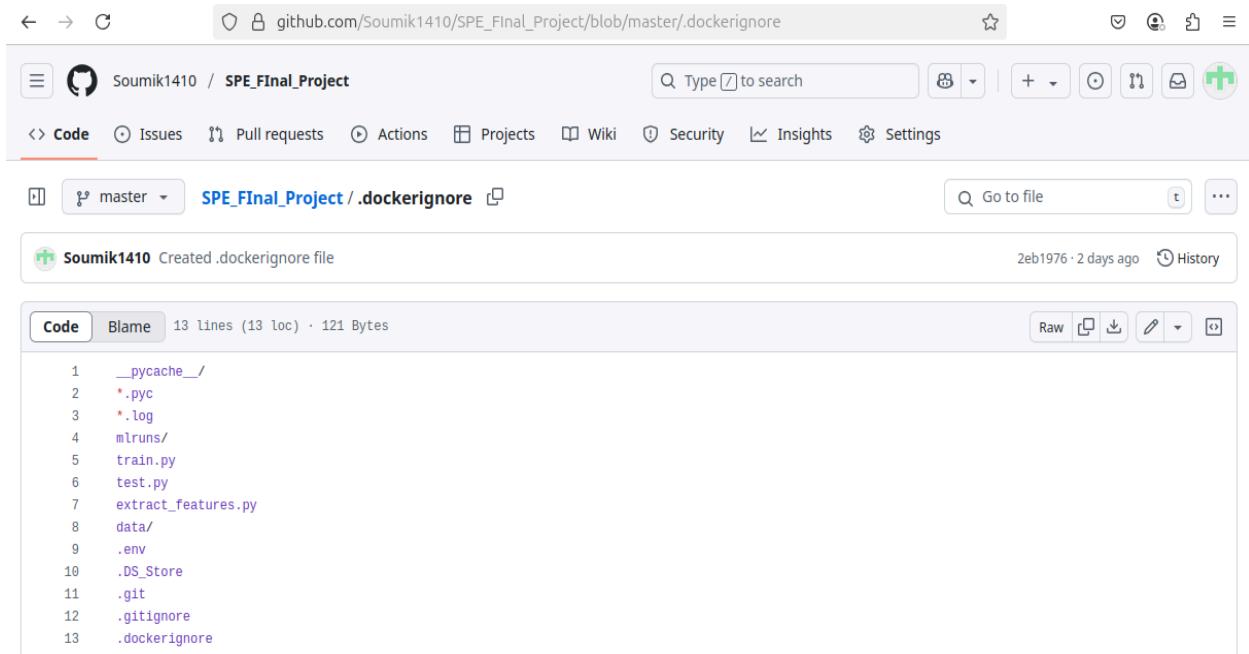
```
19      sh '/home/soumik/ml-devops-env/bin/python test.py'
20  }
21  }*/
22 stage('Export Requirements') {
23   steps {
24     sh '''
25       bash -c "source /home/soumik/ml-devops-env/bin/activate && pip freeze > requirements.txt"
26     '''
27   }
28 }
29 stage('Clean Requirements') {
30   steps {
31     script {
32       sh '''
33         echo "[INFO] Cleaning requirements.txt to keep only essential packages..."
34         sed -i -e '/^tensorflow==2\.19\.0$/p' \\
35             -e '/^streamlit==1\.45\.1$/p' \\
36             -e '/^pillow==11\.0\.0$/p' \\
37             -e '/^numpy==1\.1\.28$/p' \\
38             requirements.txt
39         echo "[INFO] Cleaned requirements.txt content:"
40         cat requirements.txt
41     '''
42   }
43 }
44 }
```

Then finally to create our docker image, we design our Dockerfile and .dockerignore files that Docker will use to build the image and also a new stage added to the Jenkins pipeline to trigger the docker build command.



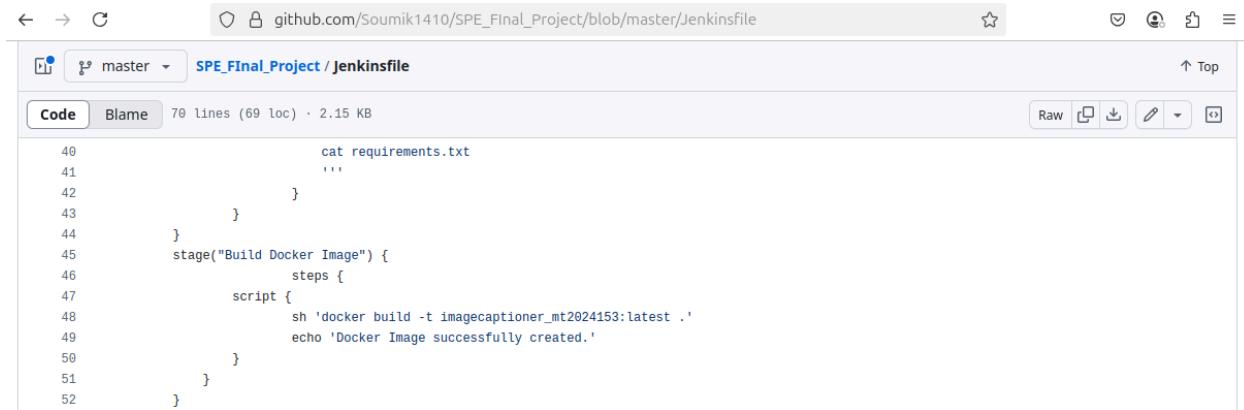
A screenshot of a GitHub repository page showing the Dockerfile for the project. The Dockerfile contains the following code:

```
1 # Use official Python image
2 FROM python:3.12-slim
3
4 # Set working directory
5 WORKDIR /app
6
7 # Install system dependencies
8 RUN apt-get update && apt-get install -y \
9     build-essential \
10    libglib2.0-0 \
11    libsm6 \
12    libxrender1 \
13    libxext6 \
14    && rm -rf /var/lib/apt/lists/*
15
16 # Copy project files
17 COPY .
18
19 # Install Python dependencies
20 RUN pip install --no-cache-dir -r requirements.txt
21
22 # Expose Streamlit default port
23 EXPOSE 8501
24
25 # Start Streamlit app
26 CMD ["streamlit", "run", "app.py", "--server.port=8501", "--server.address=0.0.0.0"]
```



A screenshot of a GitHub repository page showing the .dockerignore file for the project. The .dockerignore file contains the following entries:

```
1 __pycache__/
2 *.pyc
3 *.log
4 mruns/
5 train.py
6 test.py
7 extract_features.py
8 data/
9 .env
10 .DS_Store
11 .git
12 .gitignore
13 .dockerignore
```



The screenshot shows a GitHub code editor interface. The URL in the address bar is [github.com/Soumik1410/SPE\\_Final\\_Project/blob/master/Jenkinsfile](https://github.com/Soumik1410/SPE_Final_Project/blob/master/Jenkinsfile). The tab bar shows 'master' and 'SPE\_Final\_Project / Jenkinsfile'. The main area displays the Jenkinsfile code, which includes stages for building Docker images and running Streamlit.

```
40         cat requirements.txt
41         ...
42     }
43 }
44 stage("Build Docker Image") {
45     steps {
46         script {
47             sh 'docker build -t imagecaptioner_mt2024153:latest .'
48             echo 'Docker Image successfully created.'
49         }
50     }
51 }
52 }
```

Docker will build the image according to the instructions given in the Dockerfile i.e. using python:3.12-slim as the base image, installing system dependencies, copying everything from jenkins workspace current directory to docker build (for the saved model and artifacts), then installing required python dependencies, exposing streamlit's default port, then finally running the streamlit point as entry command.

.dockerfile lists the files and folders to not be copied when executing the COPY . . command in Dockerfile

Once the build is complete, it is pushed to the Docker Hub repository. To facilitate this, a new stage is added to the Jenkins pipeline, and Docker Hub credentials are securely stored in Jenkins' credential manager for seamless authentication during the push process.

```
53     stage('Pushing Docker Image to Hub') {
54         steps {
55             script {
56                 withDockerRegistry([credentialsId: 'dockerhub-creds', url: 'https://index.docker.io/v1/']) {
57                     sh 'docker tag imagecaptioner_mt2024153:latest soumik1410/imagecaptioner_mt2024153:latest'
58                     sh 'docker push soumik1410/imagecaptioner_mt2024153:latest'
59                 }
60             }
61         }
62     }
```

← → C [a7d7-49-207-60-154.ngrok-free.app/manage/credentials/store/system/domain/\\_/](https://a7d7-49-207-60-154.ngrok-free.app/manage/credentials/store/system/domain/_/) ☆

 Jenkins

Dashboard > Manage Jenkins > Credentials > System > Global credentials (unrestricted) >

## Global credentials (unrestricted)

Credentials that should be available irrespective of domain specification to requirements matching.

ID	Name	Kind	Description
 dockerhub-creds	soumik1410/*********	Username with password	

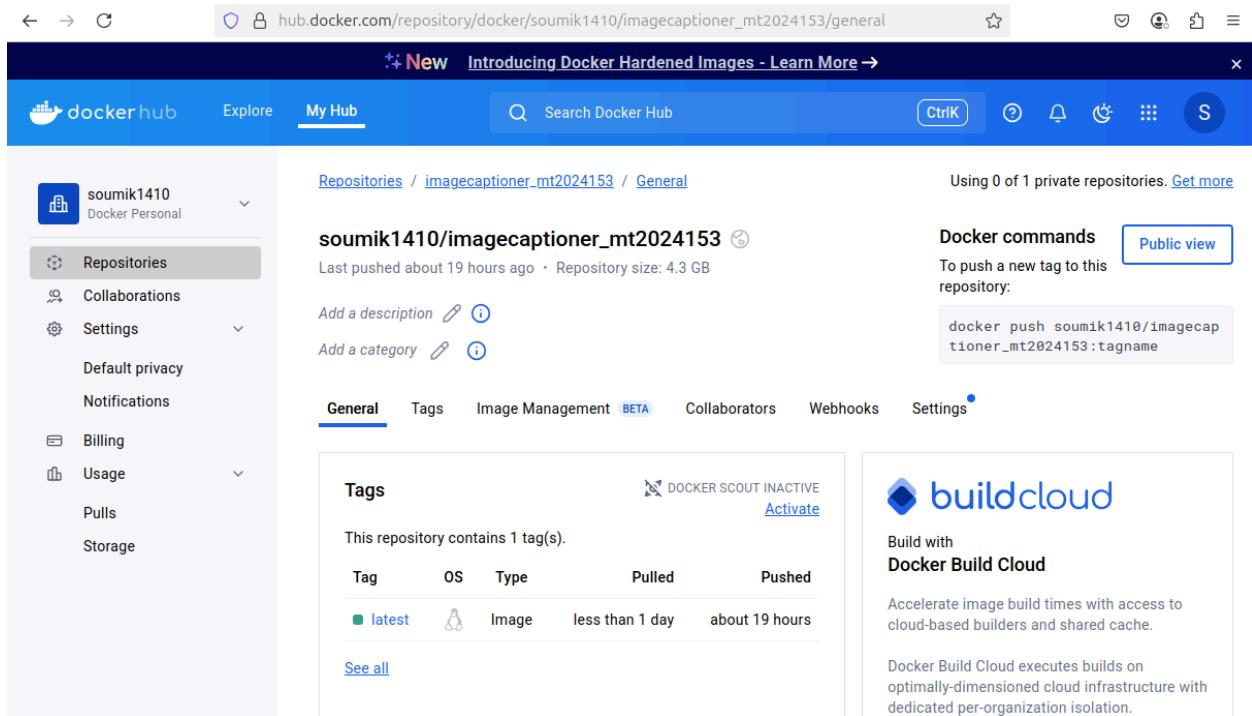
← → C [hub.docker.com/repositories/soumik1410](https://hub.docker.com/repositories/soumik1410) ☆

 dockerhub Explore My Hub  CtrlK ? ⚡ 🔍

**Repositories**  
All repositories within the soumik1410 namespace.

Name	Last Pushed	Contains	Visibility	Scout
soumik1410/imagecaptioner_mt2024153	about 18 hours ago	IMAGE	Public	Inactive
soumik1410/calculator_mt2024153	about 2 months ago	IMAGE	Public	Inactive

1–2 of 2 < >



## 5. Ansible - Deploying the application to a Kubernetes cluster (with Ansible Roles for modularity) :

Once the build has been successfully pushed to Docker Hub, the next step is to deploy it to the target environment. The deployment will be carried out on a Kubernetes cluster running locally using Minikube. The deployment tasks involve several key actions: pulling the latest build from Docker Hub, gracefully stopping the existing Minikube instance, starting up Minikube, setting the kubectl context to Minikube, creating the necessary namespace, and applying the Kubernetes manifests for both the application deployment and service.

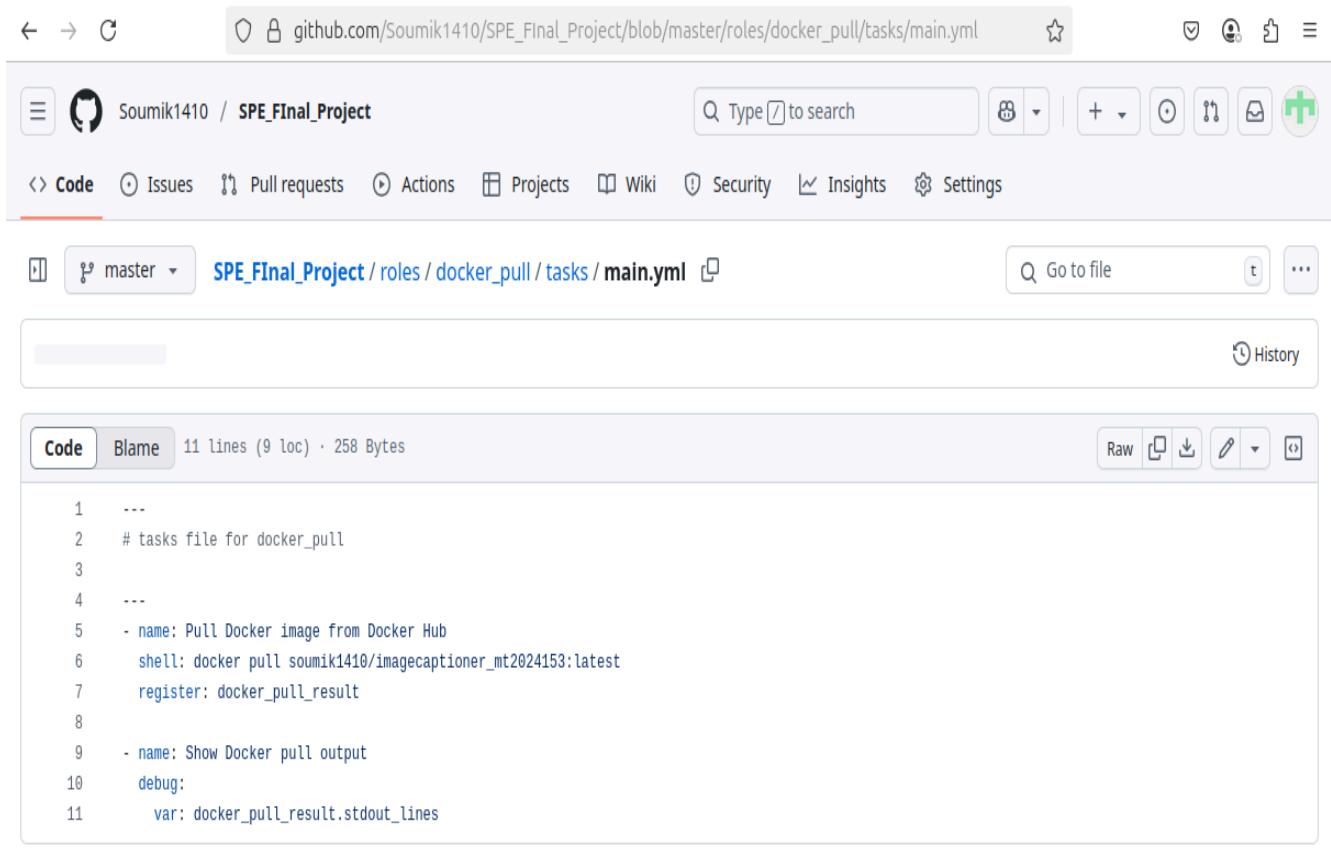
To automate these steps, we utilize an Ansible playbook that defines each task using the appropriate Ansible modules and specifies the desired states or actions. This ensures a smooth, repeatable deployment process on the local Minikube cluster.

We utilize Ansible roles to break up all the deployment tasks into 2 modules, 1 for pulling the latest docker image from Docker Hub and another for handling deployment of the application to the Kubernetes cluster.

The 2 roles created were docker\_pull and k8s\_deploy.

Folder structure for this included a roles folder with these 2 subdirectories and both followed a common structure with subfolders for tasks, handlers, tests etc. We have only used tasks in our project, modifying the main.yaml files for the 2 roles to perform the tasks mentioned earlier.

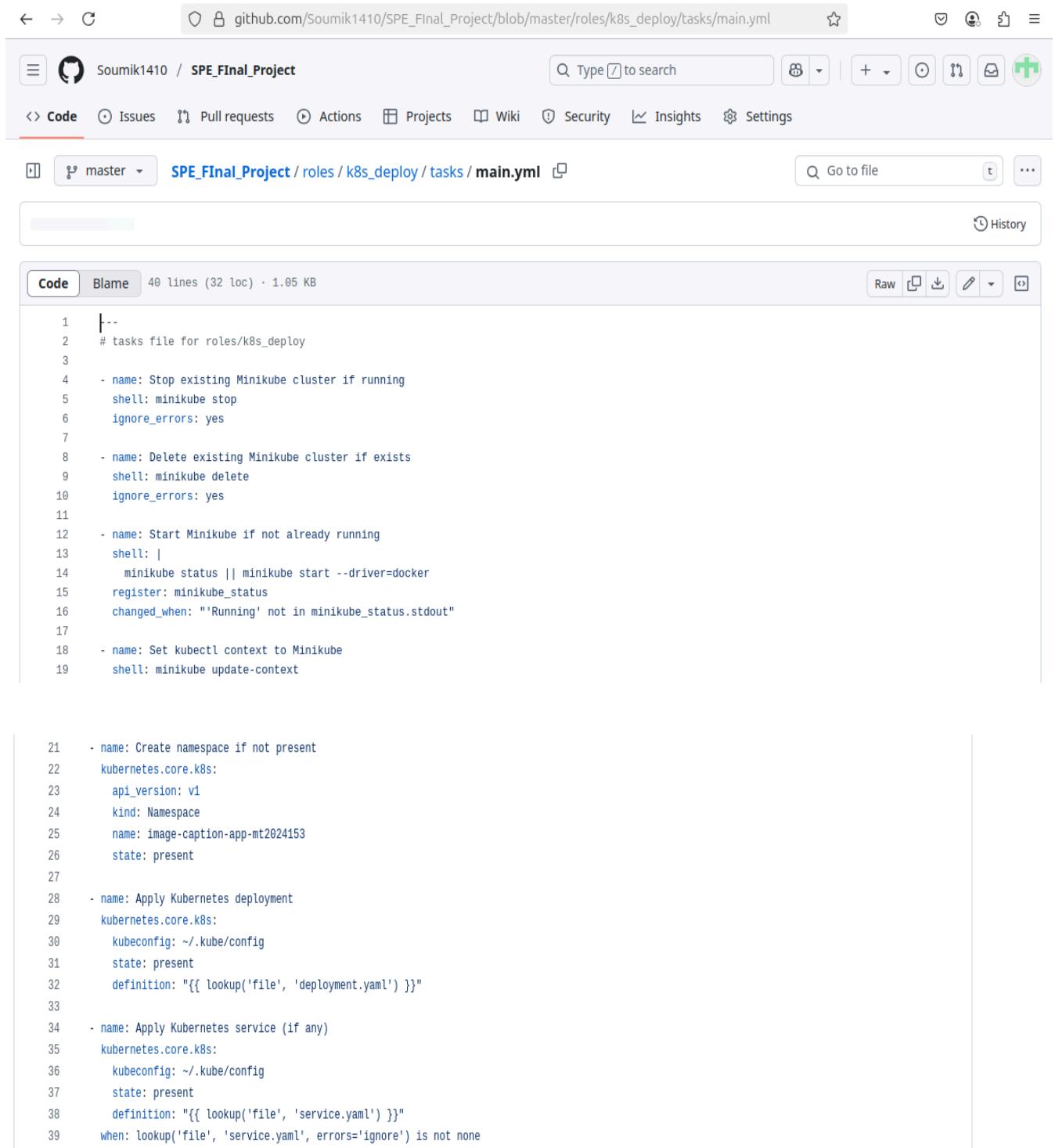
roles/docker\_pull/tasks/main.yaml :



The screenshot shows a GitHub code editor interface. The URL in the address bar is [github.com/Soumik1410/SPE\\_Final\\_Project/blob/master/roles/docker\\_pull/tasks/main.yaml](https://github.com/Soumik1410/SPE_Final_Project/blob/master/roles/docker_pull/tasks/main.yaml). The repository name is SPE\_Final\_Project. The code editor has tabs for Code, Blame, and 11 lines (9 loc) · 258 Bytes. The Code tab is selected. The history button is visible at the top right. The code content is as follows:

```
1  ---
2  # tasks file for docker_pull
3
4  ---
5  - name: Pull Docker image from Docker Hub
6    shell: docker pull soumik1410/imagecaptioner_mt2024153:latest
7    register: docker_pull_result
8
9  - name: Show Docker pull output
10   debug:
11     var: docker_pull_result.stdout_lines
```

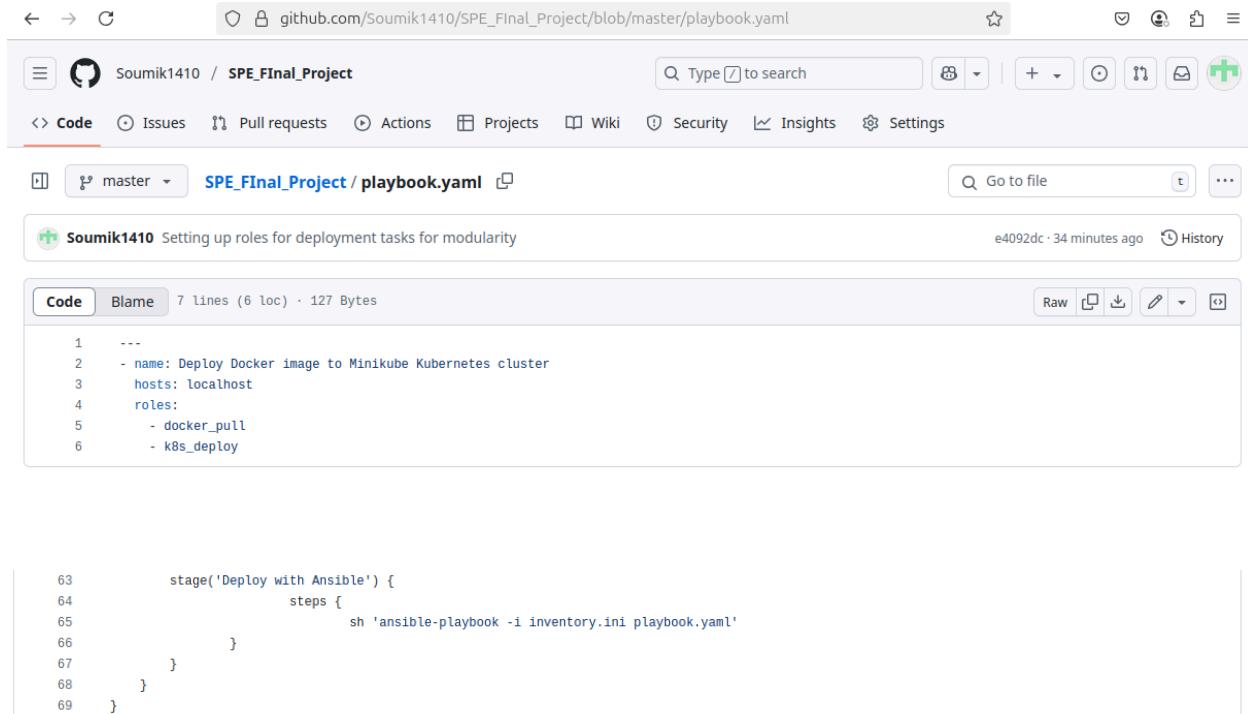
## roles/k8s\_deploy/tasks/main.yaml :



The screenshot shows a GitHub repository page for 'SPE\_Final\_Project'. The file 'main.yaml' is displayed under the 'Code' tab. The code content is as follows:

```
1  #--  
2  # tasks file for roles/k8s_deploy  
3  
4  - name: Stop existing Minikube cluster if running  
5    shell: minikube stop  
6    ignore_errors: yes  
7  
8  - name: Delete existing Minikube cluster if exists  
9    shell: minikube delete  
10   ignore_errors: yes  
11  
12 - name: Start Minikube if not already running  
13   shell: |  
14     minikube status || minikube start --driver=docker  
15   register: minikube_status  
16   changed_when: "'Running' not in minikube_status.stdout"  
17  
18 - name: Set kubectl context to Minikube  
19   shell: minikube update-context  
  
21 - name: Create namespace if not present  
22   kubernetes.core.k8s:  
23     api_version: v1  
24     kind: Namespace  
25     name: image-caption-app-mt2024153  
26     state: present  
27  
28 - name: Apply Kubernetes deployment  
29   kubernetes.core.k8s:  
30     kubeconfig: ~/.kube/config  
31     state: present  
32     definition: "{{ lookup('file', 'deployment.yaml') }}"  
33  
34 - name: Apply Kubernetes service (if any)  
35   kubernetes.core.k8s:  
36     kubeconfig: ~/.kube/config  
37     state: present  
38     definition: "{{ lookup('file', 'service.yaml') }}"  
39     when: lookup('file', 'service.yaml', errors='ignore') is not none
```

The main playbook file simply lists these roles and a new stage is added to the Jenkins pipeline that executes this main playbook file



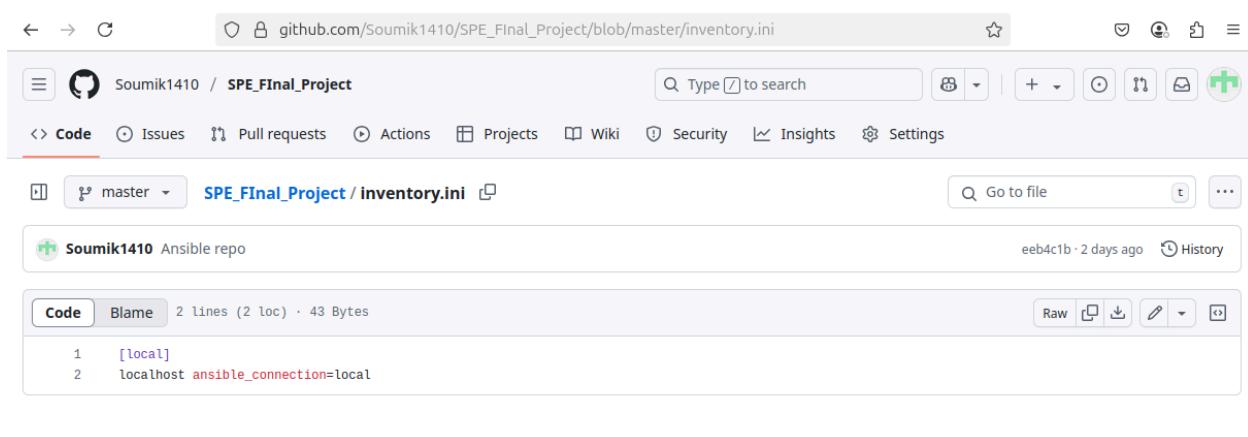
The screenshot shows a GitHub repository page for 'SPE\_Final\_Project'. The 'Code' tab is selected, showing the 'playbook.yaml' file. The code content is as follows:

```
1  ---
2  - name: Deploy Docker image to Minikube Kubernetes cluster
3    hosts: localhost
4    roles:
5      - docker_pull
6      - k8s_deploy
```

Below this, another section of the file is shown:

```
63      stage('Deploy with Ansible') {
64        steps {
65          sh 'ansible-playbook -i inventory.ini playbook.yaml'
66        }
67      }
68    }
69 }
```

Ansible inventory file used for this deployment is simple as deployment target is localhost

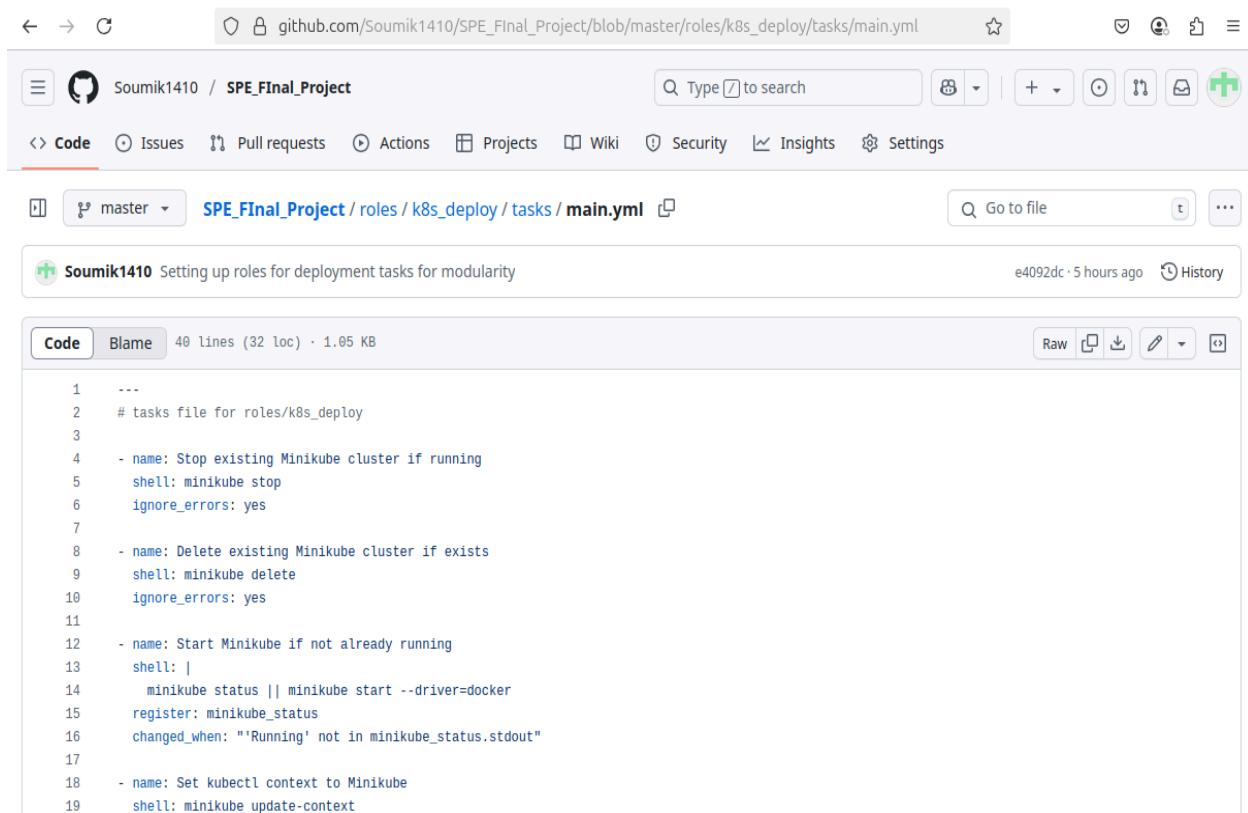


The screenshot shows a GitHub repository page for 'SPE\_Final\_Project'. The 'Code' tab is selected, showing the 'inventory.ini' file. The code content is as follows:

```
[local]
localhost ansible_connection=local
```

Our deployment target is a kubernetes cluster deployed on my localhost node using minikube. The Ansible role k8s\_deploy is used to set up the Kubernetes cluster and deploy our image captioning application.

First, it gracefully shuts down any existing minikube instance that might be running and starts up a new minikube instance, setting kubectl context to the kubernetes cluster deployed by the fresh minikube instance so that the cluster can be managed easily with kubectl commands without needing to specify cluster everytime.



The screenshot shows a GitHub repository page for 'SPE\_Final\_Project'. The 'Code' tab is selected, displaying the 'main.yml' file. The file contains Ansible tasks for managing a Minikube cluster. The code is as follows:

```
1  ---
2  # tasks file for roles/k8s_deploy
3
4  - name: Stop existing Minikube cluster if running
5    shell: minikube stop
6    ignore_errors: yes
7
8  - name: Delete existing Minikube cluster if exists
9    shell: minikube delete
10   ignore_errors: yes
11
12 - name: Start Minikube if not already running
13   shell: |
14     minikube status || minikube start --driver=docker
15   register: minikube_status
16   changed_when: "'Running' not in minikube_status.stdout"
17
18 - name: Set kubectl context to Minikube
19   shell: minikube update-context
```

Then, we use kubectl commands to create a custom namespace for our application called image-caption-app-mt2024153 and apply our deployment and service Kubernetes manifests which are used to specify the number of replicas, the build deployed and network access to our application.

Ansible module used for these is kubernetes.core.k8s which can be installed with ansible-galaxy collection install kubernetes.core

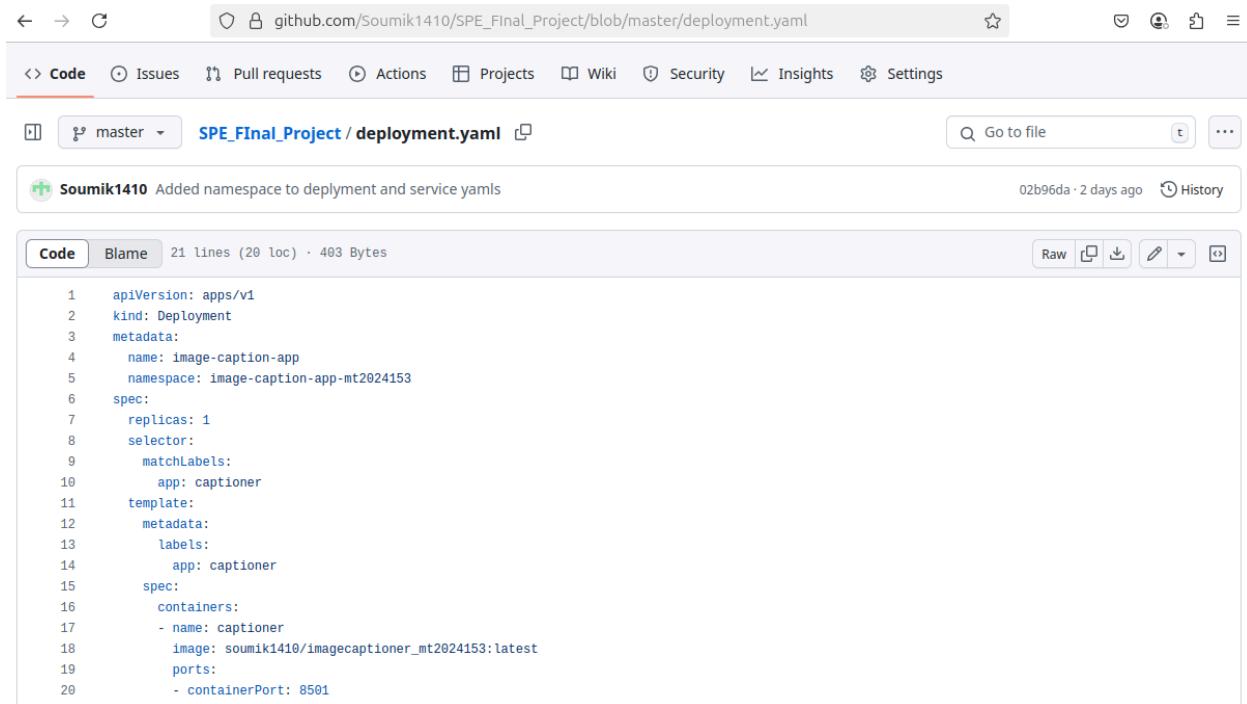
First, a separate namespace is created for our application called image-caption-app-mt2024153.

```
21 - name: Create namespace if not present
22   kubernetes.core.k8s:
23     api_version: v1
24     kind: Namespace
25     name: image-caption-app-mt2024153
26     state: present
```

Then, we apply a deployment.yaml and service.yaml in that namespace that deploys our application's docker image in a container inside a Kubernetes pod and sets up the network access to it. State:present with kubernetes.core.k8s module is equivalent to kubectl apply.

```
28 - name: Apply Kubernetes deployment
29   kubernetes.core.k8s:
30     kubeconfig: ~/.kube/config
31     state: present
32     definition: "{{ lookup('file', 'deployment.yaml') }}"
33
34 - name: Apply Kubernetes service (if any)
35   kubernetes.core.k8s:
36     kubeconfig: ~/.kube/config
37     state: present
38     definition: "{{ lookup('file', 'service.yaml') }}"
39     when: lookup('file', 'service.yaml', errors='ignore') is not none
```

## deployment.yaml :



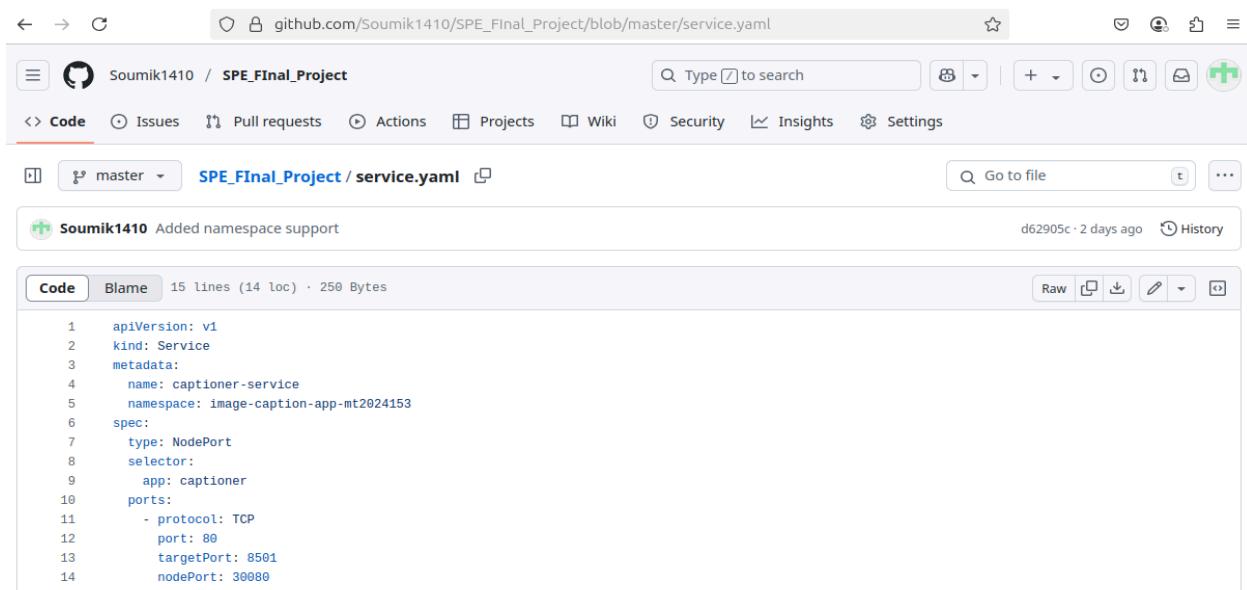
The screenshot shows a GitHub repository page for 'SPE\_Final\_Project'. The 'Code' tab is selected, displaying the 'deployment.yaml' file. The code content is as follows:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: image-caption-app
5    namespace: image-caption-app-mt2024153
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10        app: captioner
11    template:
12      metadata:
13        labels:
14          app: captioner
15    spec:
16      containers:
17        - name: captioner
18          image: soumik1410/imagecaptioner_mt2024153:latest
19        ports:
20          - containerPort: 8501
```

This configuration defines a Kubernetes Deployment for the Image Captioning App, called `image-caption-app`, intended to run within the `image-caption-app-mt2024153` namespace. It specifies a single replica (i.e., one Pod) for the application. Both the application and the Pod are labeled with `captioner` to facilitate identification, enabling users to easily determine which application is running on which Pod, as well as to manage the associated Pods within the Deployment.

The YAML further details that the Deployment will launch a single container within the Pod, which will run the Docker image for the Image Captioning App. This container will expose port 8501, the required port for accessing the Streamlit application hosted on the Pod.

### service.yaml



A screenshot of a GitHub repository page for `SPE_FInal_Project`. The `Code` tab is selected, showing the `master` branch. The file `service.yaml` is open. The code content is as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: captioner-service
  namespace: image-caption-app-mt2024153
spec:
  type: NodePort
  selector:
    app: captioner
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8501
      nodePort: 30080
```

This YAML specifies a Kubernetes Service called `captioner-service`, intended to run within the namespace `image-caption-app-mt2024153`. It is a NodePort service, which is used to allow external HTTP connections to containers deployed within this namespace. This YAML defines a Kubernetes Service named `captioner-service`, which is deployed within the `image-caption-app-mt2024153` namespace. The service is of type

NodePort, allowing external HTTP traffic to access containers running in the Kubernetes cluster. It uses the app: captioner label selector to identify and route traffic to the appropriate Pods.

The service maps node port 30080 to the container's port 8501 — the default port used by Streamlit. This setup allows external clients to connect to the Streamlit application running inside the Pods.

Once the deployment is complete we can verify the status using kubectl commands.

```
jenkins@soumik-VirtualBox:~$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured

jenkins@soumik-VirtualBox:~$
```

```
jenkins@soumik-VirtualBox:~$ kubectl get ns
NAME          STATUS   AGE
default       Active   22m
image-caption-app-mt2024153   Active   22m
kube-node-lease   Active   22m
kube-public     Active   22m
kube-system     Active   22m
jenkins@soumik-VirtualBox:~$
```

```
jenkins@soumik-VirtualBox:~$ kubectl get pods -n image-caption-app-mt2024153
NAME           READY   STATUS    RESTARTS   AGE
fluent-bit-c4zh6   1/1    Running   0          17m
grafana-8559bcb694-2xpdc   1/1    Running   0          17m
image-caption-app-756c979f57-zmjnh   1/1    Running   0          18m
loki-87d578f98-c22dr    1/1    Running   0          17m
jenkins@soumik-VirtualBox:~$
```

```
jenkins@soumik-VirtualBox:~$ kubectl describe pod image-caption-app-756c979f57-zmjnh -n image-caption-app-mt2024153
Name:           image-caption-app-756c979f57-zmjnh
Namespace:      image-caption-app-mt2024153
Priority:       0
Service Account: default
Node:           minikube/192.168.58.2
Start Time:     Thu, 22 May 2025 15:42:58 +0530
Labels:         app=captioner
                pod-template-hash=756c979f57
Annotations:    <none>
Status:         Running
IP:             10.244.0.5
IPs:
  IP:          10.244.0.5
Controlled By: ReplicaSet/image-caption-app-756c979f57
Containers:
  captioner:
    Container ID:  docker://4019ff149aada3b8bed5b265fd142a395df318b46541688933f7e1dbd75f1169
    Image:         soumik1410/imagecaptioner_mt2024153:latest
    Image ID:     docker-pullable://soumik1410/imagecaptioner_mt2024153@sha256:f24895847a4e7558a26f4d598765c7b9f38460b
    39b3a9b86261e72e32c6bb19b
    Port:         8501/TCP
    Host Port:   0/TCP
    State:        Running
      Started:   Thu, 22 May 2025 15:43:01 +0530
    Ready:        True
    Restart Count: 0
    Limits:
      cpu:  500m
    Requests:
      cpu:  200m
```

```
Environment:  <none>
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-2j4qd (ro)
Conditions:
  Type        Status
  PodReadyToStartContainers  True
  Initialized  True
  Ready        True
  ContainersReady  True
  PodScheduled  True
Volumes:
  kube-api-access-2j4qd:
    Type:           Projected (a volume that contains injected data from multiple sources)
    TokenExpirationSeconds: 3607
    ConfigMapName:   kube-root-ca.crt
    ConfigMapOptional: <nil>
    DownwardAPI:    true
  QoS Class:      Burstable
  Node-Selectors: <none>
  Tolerations:    node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                  node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type  Reason  Age  From            Message
  ----  -----  ---  ----
  Normal Scheduled 19m  default-scheduler  Successfully assigned image-caption-app-mt2024153/image-caption-app-756c979f57-zmjnh to minikube
  Normal Pulling   19m  kubelet        Pulling image "soumik1410/imagecaptioner_mt2024153:latest"
  Normal Pulled   19m  kubelet        Successfully pulled image "soumik1410/imagecaptioner_mt2024153:latest" in 2.137s (2.137s including waiting). Image size: 3039320995 bytes.
  Normal Created   19m  kubelet        Created container: captioner
  Normal Started   19m  kubelet        Started container captioner
jenkins@soumik-VirtualBox:~$
```

```

jenkins@soumik-VirtualBox:~$ kubectl get deployment -n image-caption-app-mt2024153
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
grafana       1/1     1           1           21m
image-caption-app 1/1     1           1           26m
loki          1/1     1           1           21m
jenkins@soumik-VirtualBox:~$ kubectl get svc -n image-caption-app-mt2024153
NAME            TYPE      CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
captioner-service  NodePort  10.104.28.183 <none>        80:30080/TCP  26m
grafana         NodePort  10.107.126.227 <none>        3000:30030/TCP 21m
loki            ClusterIP 10.99.136.160  <none>        3100/TCP      21m
jenkins@soumik-VirtualBox:~$ 

```

As we can see, all our deployment activities went successful. Here, we can see a couple more pods and deployments, these were brought up to introduce Horizontal Pod Autoscaling (HPA) and logging and monitoring in the pipeline.

## 6. Horizontal Pod Autoscaler :

HPA (Horizontal Pod Autoscaler) is a Kubernetes resource that automatically scales the number of Pod replicas in a Deployment based on observed resource usage, such as CPU or memory, or on custom metrics.

For this to work in our cluster, we need to first bring up the metrics-server which measures metrics such as CPU usage or memory usage of pods. We need to set requests of CPU and memory usage for our image-caption-app Deployment. Then in our HPA configuration, we need to set the resource that we want to monitor, the target utilization (computed as a percentage of the requested value) and min and max replicas that we desire. Then based on current utilization vs target utilization, HPA will automatically scale up and scale down the number of replicas to ensure average resource utilization is close to the target threshold.

We define another Ansible playbook called HPPlaybook.yaml that lists all these tasks using the kubernetes.core.k8s module. This playbook is executed automatically once the application deployment is completed, by a Jenkins pipeline which checks out the latest version of this playbook from our Github repository and then executes this playbook. This is a separate

pipeline from our first Jenkins pipeline that handled automated model training, testing, containerization and deployment of the application. This was done to modularize and independently test the HPA without needing to train the model time and time again. This second pipeline is automatically triggered on successful completion of the first pipeline.

The screenshot shows two Jenkins configuration pages for a pipeline job named "SPE\_Final\_Project\_HPA".

**Top Configuration Page (Triggers Tab):**

- General Tab:** Selected.
- Triggers Tab:** Enabled (checkbox checked).
- Projects to watch:** SPE\_Final\_Project.
- Trigger options:**
  - Trigger only if build is stable (selected)
  - Trigger even if the build is unstable
  - Trigger even if the build fails
  - Always trigger, even if the build is aborted
- Build periodically
- GitHub hook trigger for GITScm polling
- Poll SCM
- Trigger builds remotely (e.g., from scripts)

**Bottom Configuration Page (Pipeline Tab):**

- Pipeline Tab:** Selected.
- Definition:** Pipeline script (dropdown selected).
- Script:**

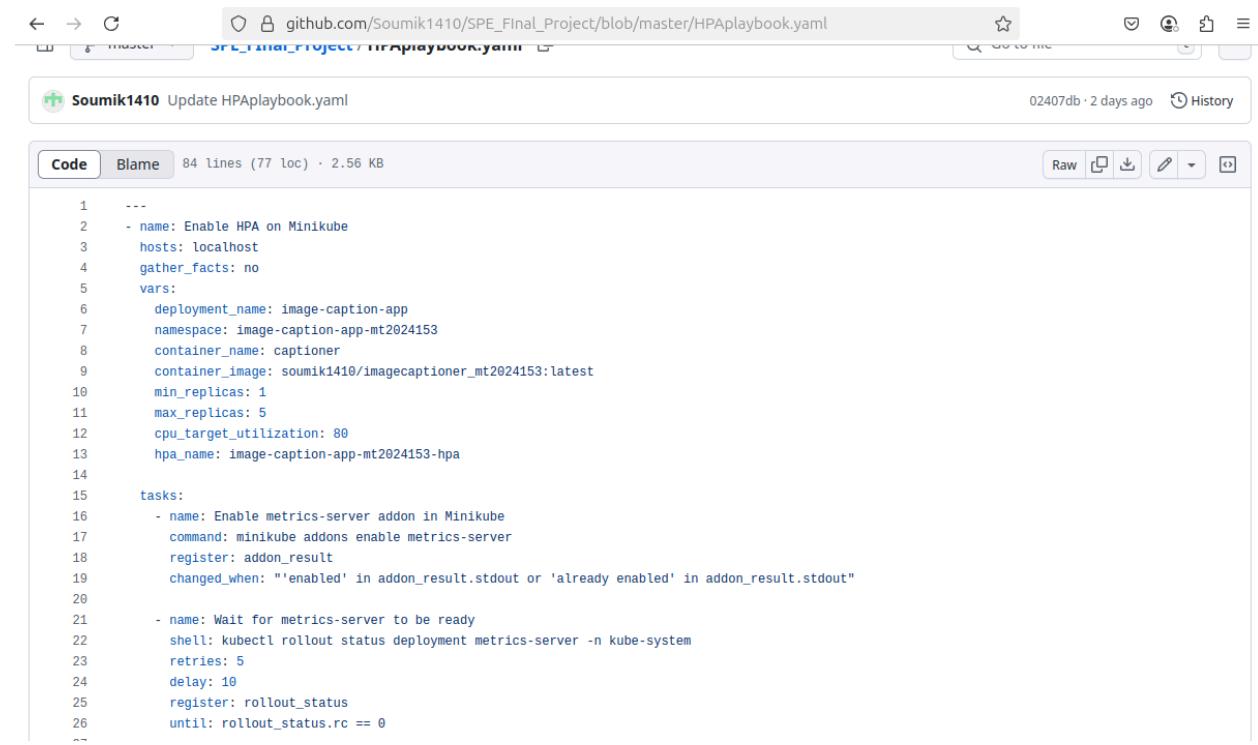
```

1 pipeline [
2   agent any
3   stages {
4     stage('Checkout Code') {
5       steps {
6         git credentialsId: 'github-token', url: 'https://github.com/Soumik1410/SPE_FInal_Project'
7       }
8     }
9     stage('Enable HPA with Ansible') {
10       steps {
11         sh 'ansible-playbook -i inventory.ini HPlaybook.yaml'
12       }
13     }
14   }
15 ]

```
- Use Groovy Sandbox:** Enabled (checkbox checked).

## HPAplaybook.yaml tasks :

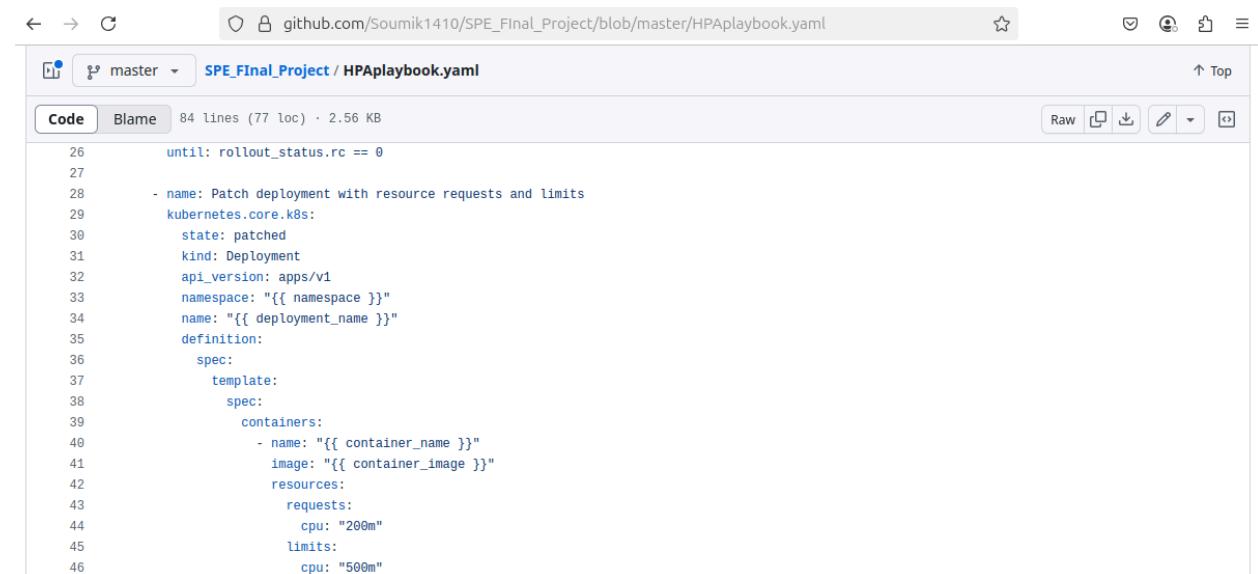
### Starting metrics server and waiting for it to be ready



The screenshot shows a GitHub code editor interface for the file `HPAplaybook.yaml`. The file contains YAML configuration for Kubernetes. It includes sections for enabling HPA on Minikube, defining a deployment for an image-caption app, and performing tasks to enable the metrics-server addon and wait for it to be ready. The code is annotated with line numbers from 1 to 26.

```
1  ---
2  - name: Enable HPA on Minikube
3    hosts: localhost
4    gather_facts: no
5    vars:
6      deployment_name: image-caption-app
7      namespace: image-caption-app-mt2024153
8      container_name: captioner
9      container_image: soumik1410/imagecaptioner_mt2024153:latest
10     min_replicas: 1
11     max_replicas: 5
12     cpu_target_utilization: 80
13     hpa_name: image-caption-app-mt2024153-hpa
14
15   tasks:
16     - name: Enable metrics-server addon in Minikube
17       command: minikube addons enable metrics-server
18       register: addon_result
19       changed_when: "'enabled' in addon_result.stdout or 'already enabled' in addon_result.stderr"
20
21     - name: Wait for metrics-server to be ready
22       shell: kubectl rollout status deployment metrics-server -n kube-system
23       retries: 5
24       delay: 10
25       register: rollout_status
26       until: rollout_status.rc == 0
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
```

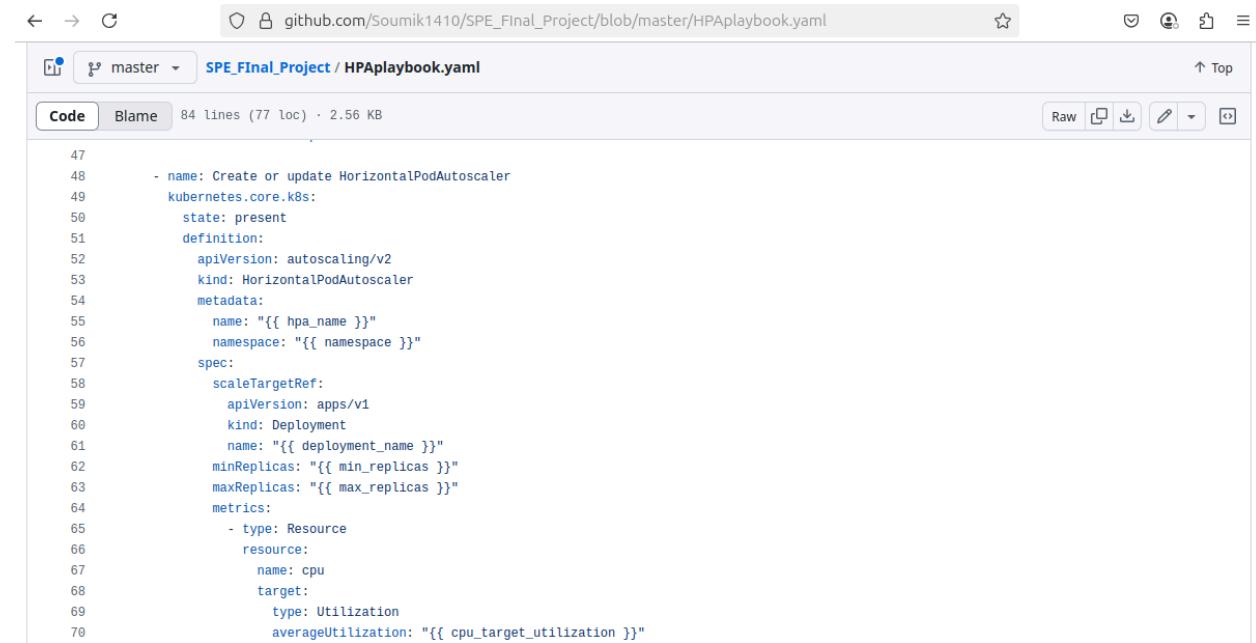
### Patch our application deployment to have resource requests and limits



The screenshot shows a GitHub code editor interface for the file `HPAplaybook.yaml`. The file now includes a new section for patching the deployment with resource requests and limits. The code is annotated with line numbers from 26 to 46.

```
26   until: rollout_status.rc == 0
27
28   - name: Patch deployment with resource requests and limits
29     kubernetes.core.k8s:
30       state: patched
31       kind: Deployment
32       api_version: apps/v1
33       namespace: "{{ namespace }}"
34       name: "{{ deployment_name }}"
35       definition:
36         spec:
37           template:
38             spec:
39               containers:
40                 - name: "{{ container_name }}"
41                   image: "{{ container_image }}"
42                   resources:
43                     requests:
44                       cpu: "200m"
45                     limits:
46                       cpu: "500m"
```

## Configure HPA



The screenshot shows a GitHub code editor interface. The URL in the address bar is `github.com/Soumik1410/SPE_Final_Project/blob/master/HPAplaybook.yaml`. The repository name is `SPE_Final_Project`, and the file name is `HPAplaybook.yaml`. The code editor displays a YAML configuration for a Horizontal Pod Autoscaler (HPA). The code includes variables like `hpa_name`, `namespace`, `deployment_name`, `min_replicas`, `max_replicas`, and `cpu_target_utilization`.

```
47
48     - name: Create or update HorizontalPodAutoscaler
49       kubernetes.core.k8s:
50         state: present
51         definition:
52           apiVersion: autoscaling/v2
53           kind: HorizontalPodAutoscaler
54           metadata:
55             name: "{{ hpa_name }}"
56             namespace: "{{ namespace }}"
57           spec:
58             scaleTargetRef:
59               apiVersion: apps/v1
60               kind: Deployment
61               name: "{{ deployment_name }}"
62             minReplicas: "{{ min_replicas }}"
63             maxReplicas: "{{ max_replicas }}"
64           metrics:
65             - type: Resource
66               resource:
67                 name: cpu
68                 target:
69                   type: Utilization
70                   averageUtilization: "{{ cpu_target_utilization }}"
```

## Wait for HPA to start up and ready to start monitoring

```
72     - name: Wait for HPA to be created
73       shell: kubectl get hpa "{{ hpa_name }}" -n "{{ namespace }}"
74       retries: 5
75       delay: 5
76       register: hpa_status
77       until: hpa_status.rc == 0
78
79     - name: Show HPA status
80       command: kubectl describe hpa "{{ hpa_name }}" -n "{{ namespace }}"
81       register: hpa_describe
82
83     - debug:
84       msg: "{{ hpa_describe.stdout }}"
```

Once this is executed, we can verify the status using `kubectl` commands.

```
jenkins@soumik-VirtualBox:~$ kubectl rollout status deployment metrics-server -n kube-system
deployment "metrics-server" successfully rolled out
jenkins@soumik-VirtualBox:~$ kubectl get pods -n kube-system
NAME                  READY   STATUS    RESTARTS   AGE
coredns-668d6bf9bc-zjgtz  1/1    Running   0          4h35m
etcd-minikube          1/1    Running   0          4h36m
kube-apiserver-minikube  1/1    Running   0          4h35m
kube-controller-manager-minikube  1/1    Running   0          4h36m
kube-proxy-kb4st        1/1    Running   0          4h35m
kube-scheduler-minikube  1/1    Running   0          4h36m
metrics-server-7ffb699795-5zxvg  1/1    Running   0          4h35m
storage-provisioner      1/1    Running   1 (4h35m ago) 4h35m
jenkins@soumik-VirtualBox:~$ kubectl top pods -n image-caption-app-mt2024153
NAME                           CPU(cores)   MEMORY(bytes)
fluent-bit-c4zh6                98m         12Mi
grafana-8559bcb694-2xpdc       2m          41Mi
image-caption-app-756c979f57-zmjnh  0m          7Mi
loki-87d578f98-c22dr          33m         280Mi
jenkins@soumik-VirtualBox:~$
```

```
jenkins@soumik-VirtualBox:~$ kubectl describe deployment image-caption-app -n image-caption-app-mt2024153
Name:           image-caption-app
Namespace:      image-caption-app-mt2024153
CreationTimestamp: Thu, 22 May 2025 15:38:24 +0530
Labels:         <none>
Annotations:   deployment.kubernetes.io/revision: 2
Selector:       app=captioner
Replicas:       1 desired | 1 updated | 1 total | 1 available | 0 unavailable
StrategyType:  RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=captioner
  Containers:
    captioner:
      Image:      soumik1410/imagecaptioner_mt2024153:latest
      Port:       8501/TCP
      Host Port: 0/TCP
      Limits:
        cpu: 500m
      Requests:
        cpu: 200m
      Environment: <none>
      Mounts:      <none>
      Volumes:     <none>
    Node-Selectors: <none>
    Tolerations:  <none>
  Conditions:
    Type     Status  Reason
    ----     -----  -----
    Available  True    MinimumReplicasAvailable
    Progressing  True    NewReplicaSetAvailable
```

```
jenkins@soumik-VirtualBox:~$ kubectl get hpa -n image-caption-app-mt2024153
NAME          REFERENCE  TARGETS  MINPODS  MAXPODS  REPLICAS  AGE
image-caption-app-mt2024153-hpa  Deployment/image-caption-app  cpu: 0%/80%  1        5          1          4h36m
jenkins@soumik-VirtualBox:~$
```

## 7. Logging and Monitoring :

Logging and monitoring with stacks like ELK provide real-time visibility into application performance and system behavior, helping quickly detect, troubleshoot, and respond to issues in a DevOps pipeline.

We saw that when running Elastic search it was consuming a lot of resources, especially in a resource constrained environment like a Minikube cluster on a Virtualbox VM and leading to crashes. As such, we decided to instead go for a different more lightweight stack using Fluent Bit, Loki and Grafana.

Fluent Bit, Loki, and Grafana are significantly lighter and more resource-efficient than ELK, making them ideal for a resource-constrained environment like a Minikube cluster on a VirtualBox VM. Additionally, Fluent

Bit integrates seamlessly with Kubernetes for log collection, Loki stores logs efficiently without heavy indexing (unlike Elasticsearch), and Grafana provides powerful visualizations—all with a much smaller footprint.

To set up this stack, we need to apply deployment and configuration YAMLS for each of the components inside our kubernetes cluster.

To do this, a third Jenkins pipeline was created which gets triggered automatically when the second pipeline that sets up HPA successfully executes.

The screenshots show the Jenkins Pipeline configuration interface for a job named "SPE\_Final\_Project\_FluentBit\_Loki\_Grafana".

**Top Screenshot (Triggers):**

- General:** Build after other projects are built (checked), watching "SPE\_Final\_Project\_HPA".
- Triggers:** Trigger only if build is stable (selected).
- Advanced:** Build periodically, GitHub hook trigger for GITScm polling, Poll SCM, Trigger builds remotely (e.g., from scripts).

**Bottom Screenshot (Pipeline):**

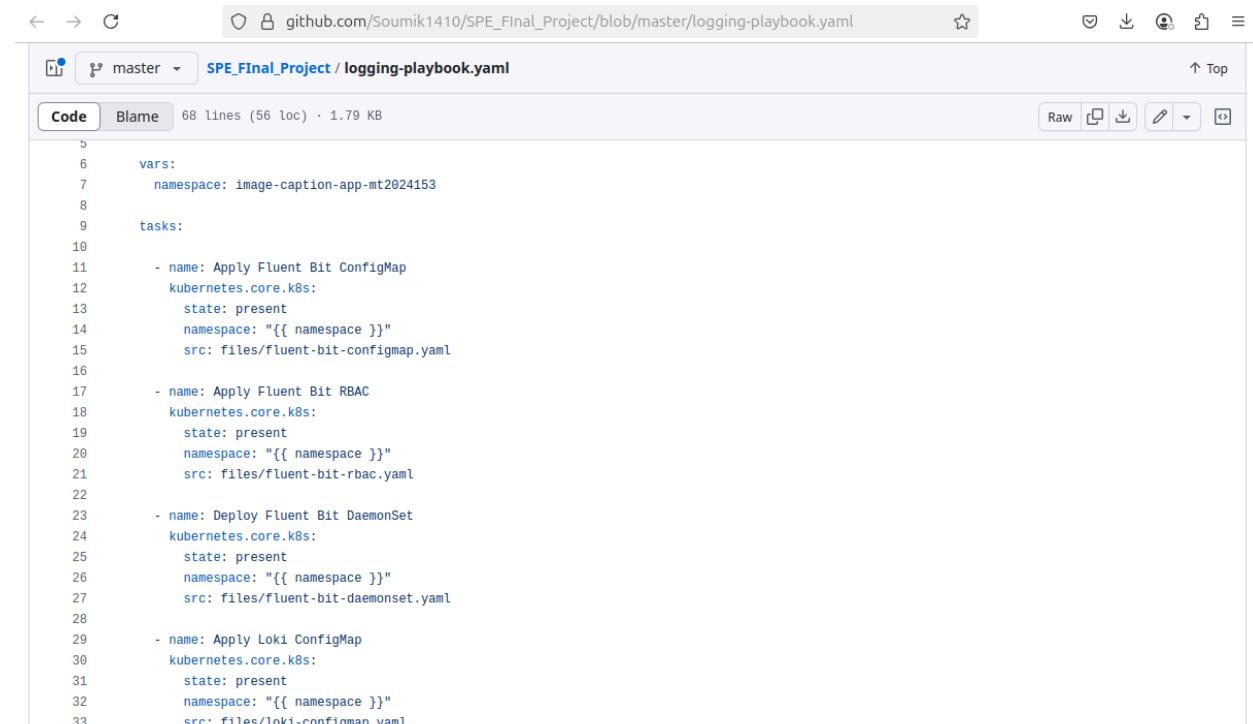
- Pipeline:** Define your Pipeline using Groovy directly or pull it from source control.
- Definition:** Pipeline script.
- Script:** Groovy code defining the pipeline stages.

```
1 pipeline {
2     agent any
3     stages {
4         stage('Checkout Code') {
5             steps {
6                 git credentialsId: 'github-token', url: 'https://github.com/Soumik1410/SPE_Final_Project'
7             }
8         }
9         stage('Enable Fluent Bit, Loki and Grafana Stack') {
10            steps {
11                sh 'ansible-playbook -i inventory.ini logging-playbook.yaml'
12            }
13        }
14    }
15 }
```

- Advanced:** Use Groovy Sandbox (checked).

This Jenkins pipeline checks out the latest version of the Github repository and executes the Ansible playbook that sets up Fluent Bit, LOKI and Grafana inside the kubernetes cluster.

### logging-playbook.yaml :



The screenshot shows the GitHub interface displaying the `logging-playbook.yaml` file from the `SPE_Final_Project` repository. The file contains Ansible playbooks for setting up Fluent Bit, LOKI, and Grafana in a Kubernetes cluster. The code is organized into sections for variables, tasks, and roles.

```
vars:
  namespace: image-caption-app-mt2024153

tasks:
  - name: Apply Fluent Bit ConfigMap
    kubernetes.core.k8s:
      state: present
      namespace: "{{ namespace }}"
      src: files/fluent-bit-configmap.yaml

  - name: Apply Fluent Bit RBAC
    kubernetes.core.k8s:
      state: present
      namespace: "{{ namespace }}"
      src: files/fluent-bit-rbac.yaml

  - name: Deploy Fluent Bit DaemonSet
    kubernetes.core.k8s:
      state: present
      namespace: "{{ namespace }}"
      src: files/fluent-bit-daemonset.yaml

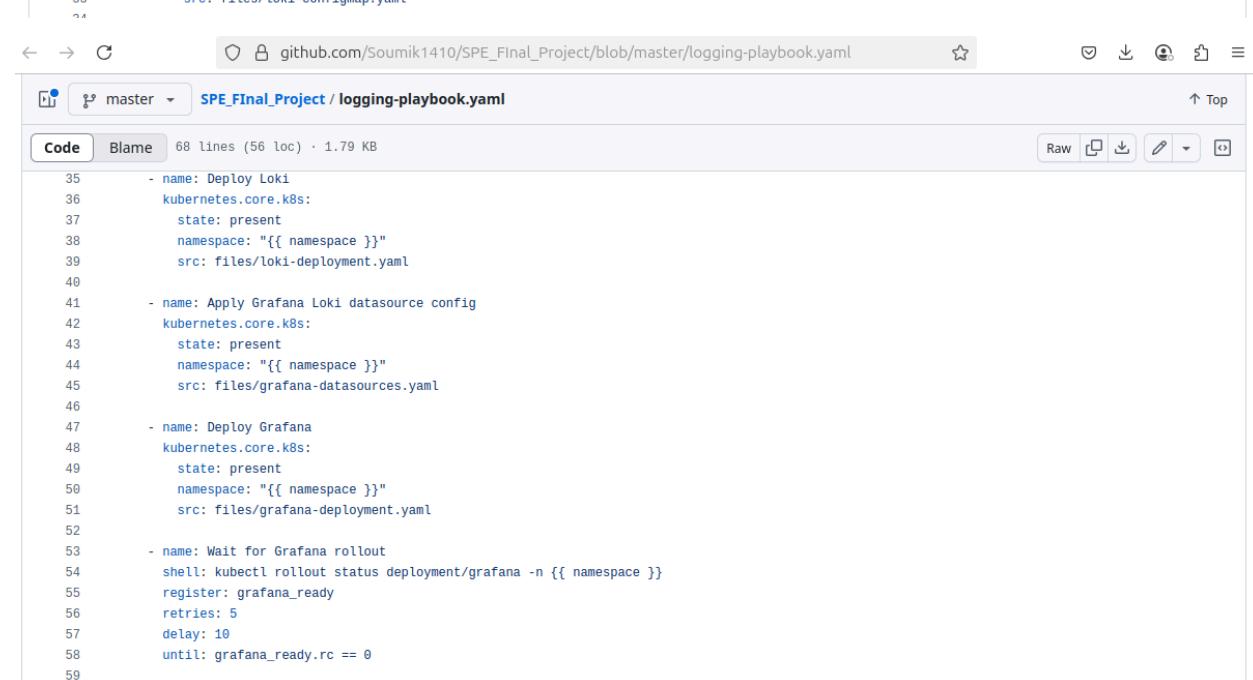
  - name: Apply Loki ConfigMap
    kubernetes.core.k8s:
      state: present
      namespace: "{{ namespace }}"
      src: files/loki-configmap.yaml

  - name: Deploy Loki
    kubernetes.core.k8s:
      state: present
      namespace: "{{ namespace }}"
      src: files/loki-deployment.yaml

  - name: Apply Grafana Loki datasource config
    kubernetes.core.k8s:
      state: present
      namespace: "{{ namespace }}"
      src: files/grafana-datasources.yaml

  - name: Deploy Grafana
    kubernetes.core.k8s:
      state: present
      namespace: "{{ namespace }}"
      src: files/grafana-deployment.yaml

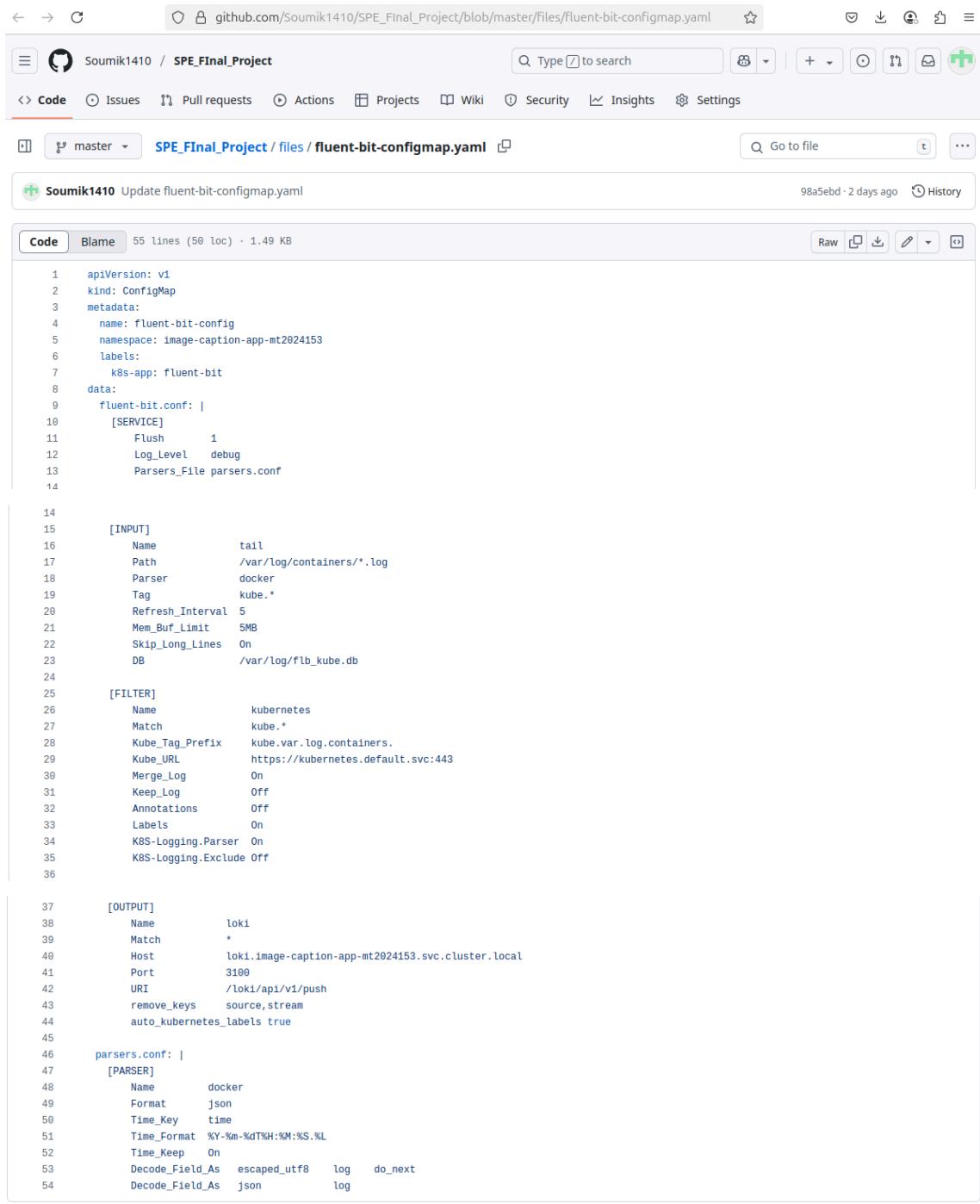
  - name: Wait for Grafana rollout
    shell: kubectl rollout status deployment/grafana -n {{ namespace }}
    register: grafana_ready
    retries: 5
    delay: 10
    until: grafana_ready.rc == 0
```



The second screenshot shows the same GitHub interface with the same file content, likely a later revision or a different part of the file. The code remains largely the same, with minor differences in line numbers and specific configuration details.

It applies a set of Kubernetes manifests in a particular order to configure and bring up the components needed for logging and monitoring and waiting till Grafana rollout and thus the full stack deployment is complete.

### fluent-bit-configmap.yaml :



The screenshot shows a GitHub code editor interface with the file `fluent-bit-configmap.yaml` open. The file content is a YAML configuration for Fluent Bit, defining inputs from log files, filters for Kubernetes logs, and an output to Loki.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluent-bit-config
  namespace: image-caption-app-mt2024153
  labels:
    k8s-app: fluent-bit
data:
  fluent-bit.conf: |
    [SERVICE]
      Flush 1
      Log_Level debug
      Parsers_File parsers.conf

  [INPUT]
    Name tail
    Path /var/log/containers/*.log
    Parser docker
    Tag kube.*
    Refresh_Interval 5
    Mem_Buf_Limit 5MB
    Skip_Long_Lines On
    DB /var/log/flb_kube.db

  [FILTER]
    Name kubernetes
    Match kube./*
    Kube_Tag_Prefix kube.var.log.containers.
    Kube_URL https://kubernetes.default.svc:443
    Merge_Log On
    Keep_Log Off
    Annotations Off
    Labels On
    K8S-Logging.Parser On
    K8S-Logging.Exclude Off

  [OUTPUT]
    Name loki
    Match *
    Host loki.image-caption-app-mt2024153.svc.cluster.local
    Port 3100
    URI /loki/api/v1/push
    remove_keys source,stream
    auto_kubernetes_labels true

  parsers.conf: |
    [PARSER]
      Name docker
      Format json
      Time_Key time
      Time_Format %Y-%m-%dT%H:%M:%S.%L
      Time_Keep On
      Decode_Field_As escaped_utf8 log do_next
      Decode_Field_As json log
```

This configuration specifies first the name of the configmap and the label of the logging app stack. Then it specifies the actual configuration data starting with service block which states the log flushing frequency as 1 second, log level is debug & specifies the specific parser file used to parse the collected logs according to a specific format.

Then it specifies the input block which states the plugin to use to collect logs (tail), path to container logs, specific parser defined in the parser file specified in service block, tagging the logs for processing and filtering later and other configurations, like refresh interval looking for new logs, memory buffer limit and DB to collect logs.

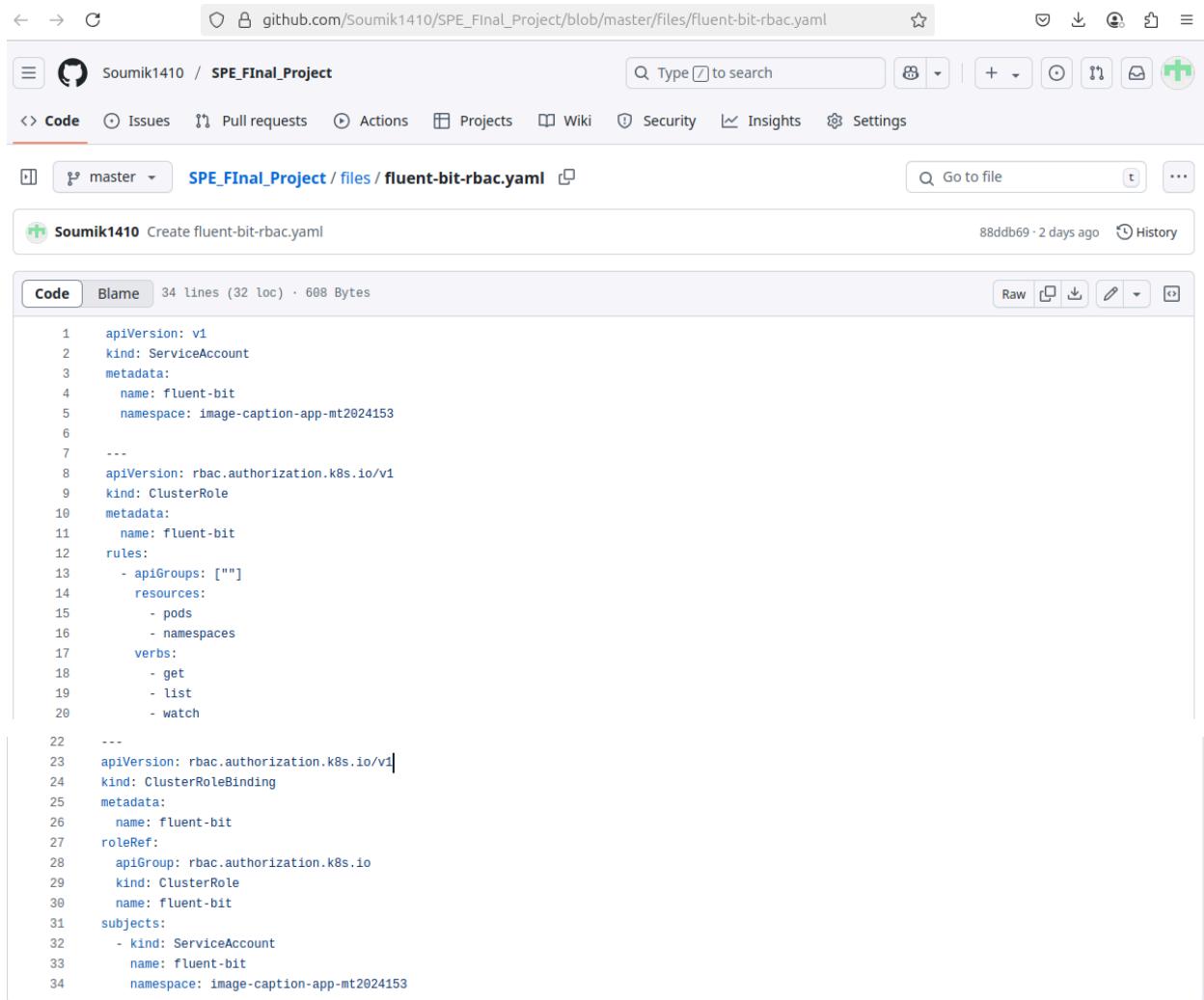
In the filter block, the kubernetes filter is applied to all the logs, applying the filter to logs tagged with the tag specified in the input block, stripping the prefix and merging logs when needed, applying kubernetes labels to the logs like job, namespace details etc.

In the output block, the output channel is specified, for our stack it is the Loki server we will bring up. Match \* indicates all logs collected will be forwarded to loki. Then the host, port and relative URI path to the loki server is specified. Finally, certain keys are removed and certain kubernetes labels are auto applied to the logs before they are sent to Loki.

The parser block specifies the details of the parser file specified in the service block such as the name of the parser, format to parse the logs according to, preferences for the time field, and decoding preferences for the actual logs message.

Once the configmap is applied, the next step is to apply RBAC for Fluent Bit to access Kubernetes resources like pods to collect & enrich logs.

## fluent-bit-rbac.yaml :



The screenshot shows a GitHub repository page for 'SPE\_Final\_Project'. The file 'fluent-bit-rbac.yaml' is being viewed. The code editor displays the YAML configuration for creating a ServiceAccount and a ClusterRole, and then binding the ClusterRole to the ServiceAccount.

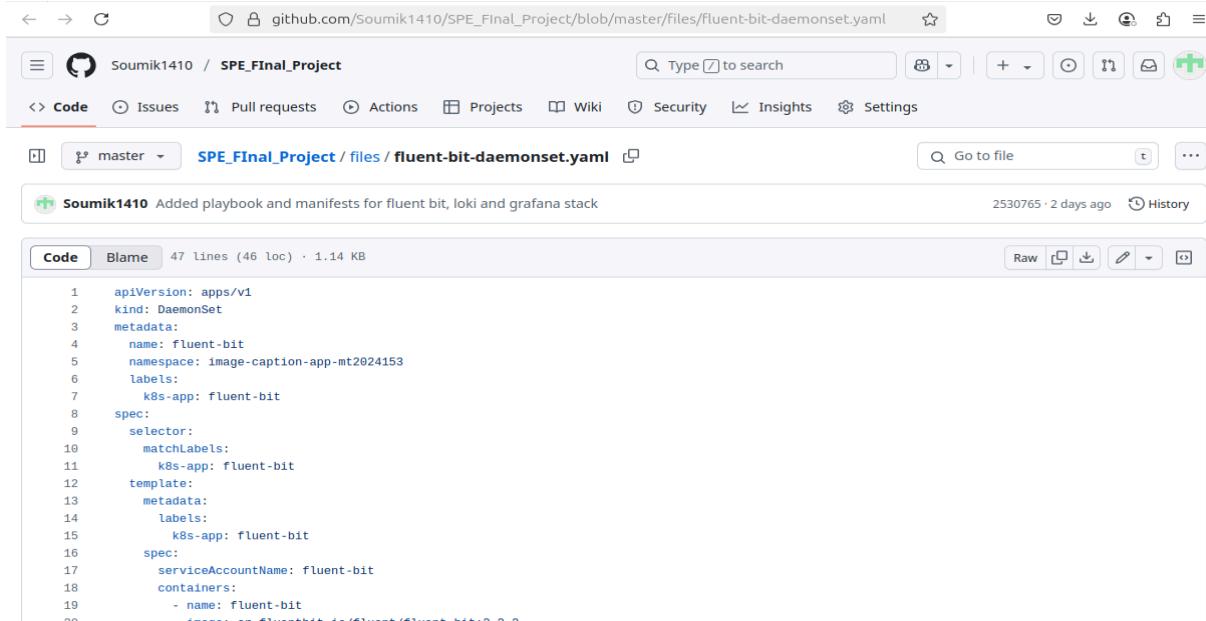
```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: fluent-bit
  namespace: image-caption-app-mt2024153
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: fluent-bit
rules:
- apiGroups: [""]
  resources:
    - pods
    - namespaces
  verbs:
    - get
    - list
    - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: fluent-bit
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: fluent-bit
subjects:
- kind: ServiceAccount
  name: fluent-bit
  namespace: image-caption-app-mt2024153
```

Here, first a ServiceAccount is created with the name fluent-bit in the namespace image-caption-app-mt2024153, then a role is created that is cluster-wide and has the same name fluent-bit with permissions to access core Kubernetes api groups, pods and namespaces and perform read only actions such as get, list and watch.

Then finally the role is bound to the service account created with a ClusterRoleBinding which is also named fluent-bit. The role fluent-bit is mapped to the subject service account fluent-bit.

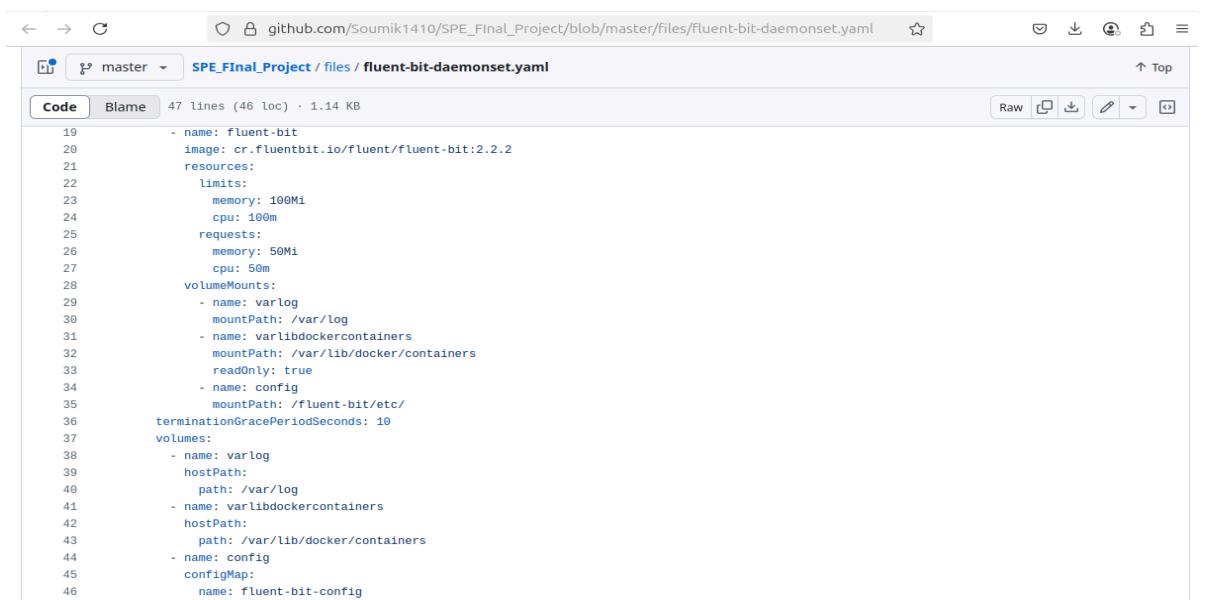
Once the configmap and RBAC is both set up, the fluent bit daemon set is applied to the cluster, which is a special Kubernetes resource that brings up 1 fluent bit pod for every node in the cluster, each pod collecting logs from its corresponding node. Since minikube is a 1 node cluster, it brings up 1 pod in this case.

### fluent-bit-daemonset.yaml :



The screenshot shows the GitHub interface for the file `fluent-bit-daemonset.yaml` in the `SPE_Final_Project` repository. The code is as follows:

```
1 apiVersion: apps/v1
2 kind: DaemonSet
3 metadata:
4   name: fluent-bit
5   namespace: image-caption-app-mt2024153
6   labels:
7     k8s-app: fluent-bit
8 spec:
9   selector:
10     matchLabels:
11       k8s-app: fluent-bit
12   template:
13     metadata:
14       labels:
15         k8s-app: fluent-bit
16     spec:
17       serviceAccountName: fluent-bit
18     containers:
19       - name: fluent-bit
20         image: cr.fluentbit.io/fluent-bit:2.2.2
```



The screenshot shows the GitHub interface for the same file, with the container definition expanded to show resources and volumes. The code is as follows:

```
19       - name: fluent-bit
20         image: cr.fluentbit.io/fluent/fluent-bit:2.2.2
21         resources:
22           limits:
23             memory: 100Mi
24             cpu: 100m
25           requests:
26             memory: 50Mi
27             cpu: 50m
28         volumeMounts:
29           - name: varlog
30             mountPath: /var/log
31           - name: varlibdockercontainers
32             mountPath: /var/lib/docker/containers
33             readOnly: true
34           - name: config
35             mountPath: /fluent-bit/etc/
36         terminationGracePeriodSeconds: 10
37     volumes:
38       - name: varlog
39         hostPath:
40           path: /var/log
41       - name: varlibdockercontainers
42         hostPath:
43           path: /var/lib/docker/containers
44       - name: config
45         configMap:
46           name: fluent-bit-config
```

The YAML specifies the name of the daemonset, the namespace to be applied in and labels used for identification. The label is applied to the pods brought up by the daemonset and is used to identify the pods that will be managed by the daemonset. The fluent bit pods are associated with the Service Account fluent-bit for RBAC access.

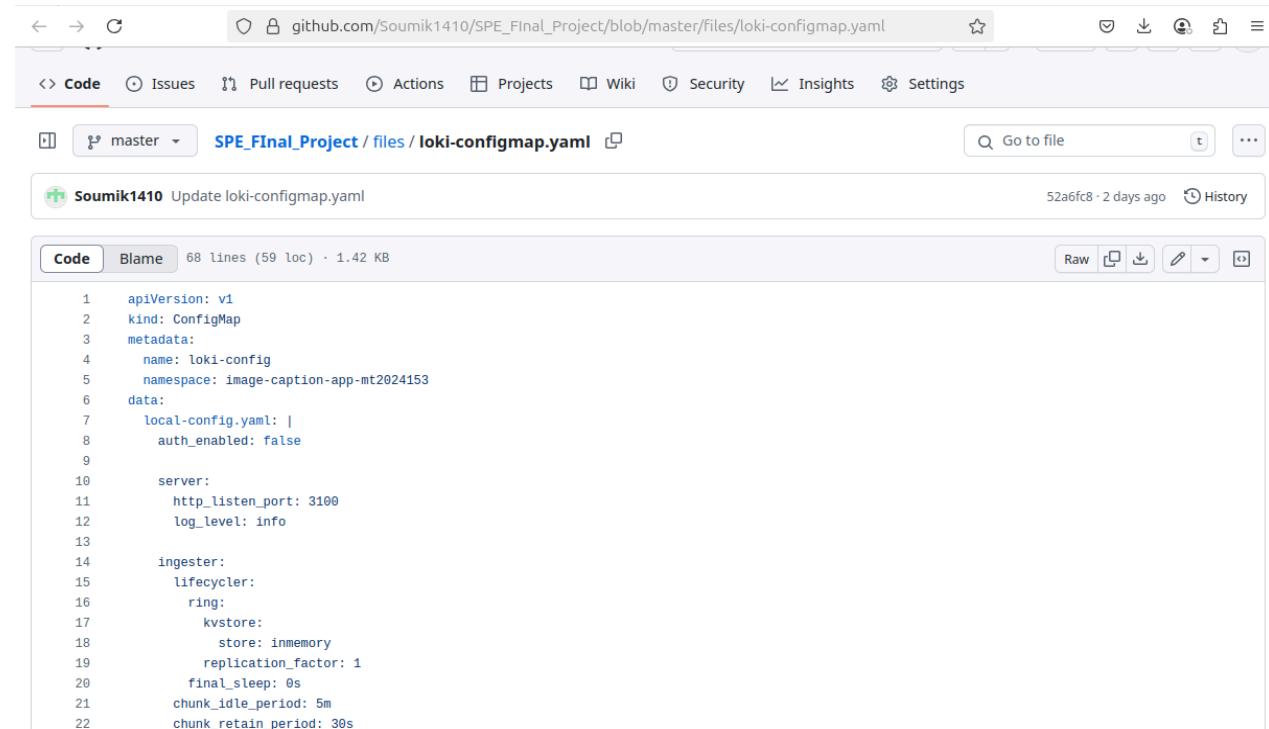
Then the container details are specified such as the fluent-bit image to deploy, resource limits and requests for the container, and host directories mounted to access the fluent bit config map and container logs, volumes specifying the data sources at those directories.

Once this is applied, a fluent bit daemonset and 1 fluent bit pod will be up and running in the namespace.

Next, the Loki server needs to be brought up and running.

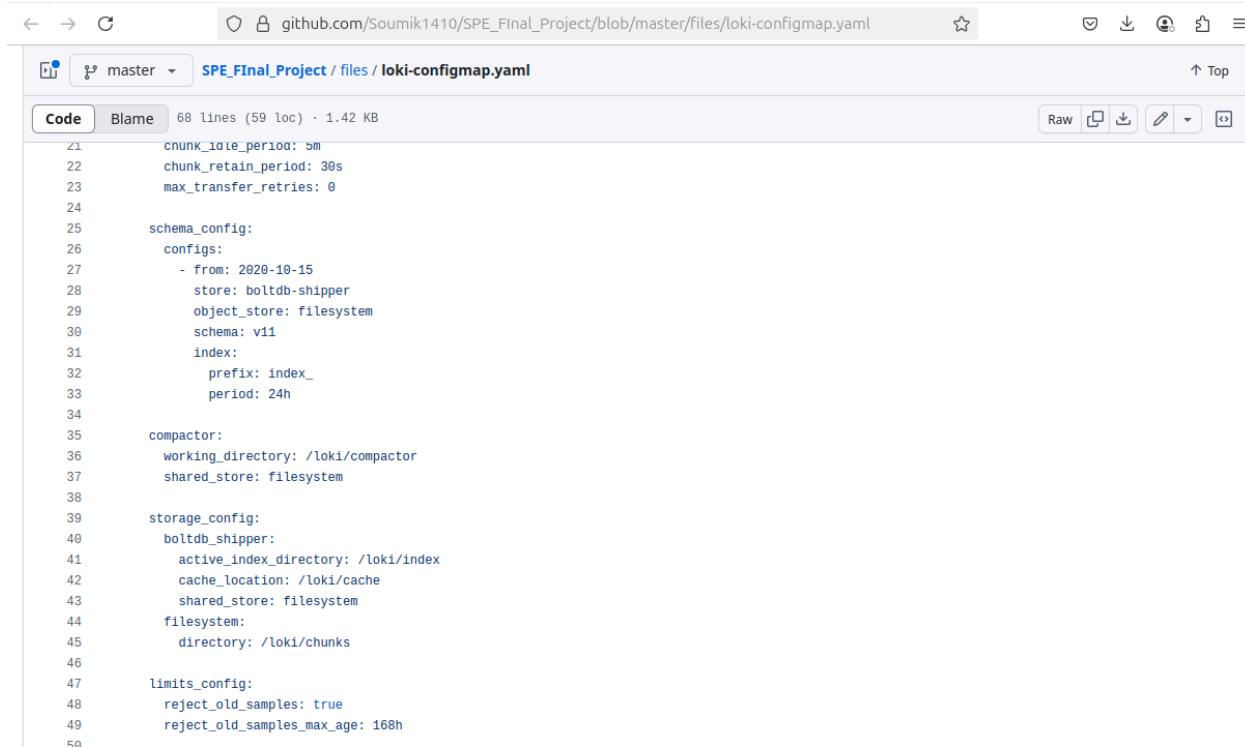
For this, first the configmap for the loki server is applied.

loki-configmap.yaml :



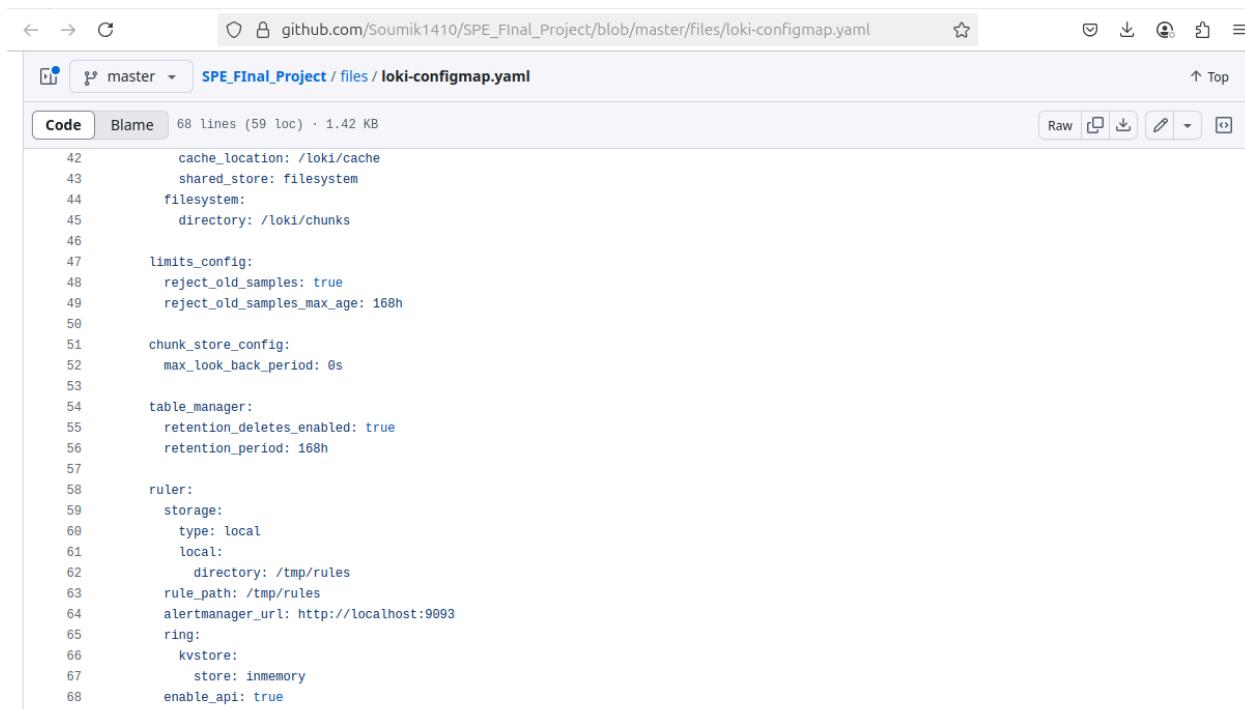
A screenshot of a GitHub code editor interface. The URL in the address bar is `github.com/Soumik1410/SPE_Final_Project/blob/master/files/loki-configmap.yaml`. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. Below the navigation bar, there's a search bar with the placeholder "Go to file" and a dropdown menu showing "master". A commit history card is visible, showing a commit from "Soumik1410" with the message "Update loki-configmap.yaml" made 2 days ago. The main content area shows the YAML configuration file:

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: loki-config
5    namespace: image-caption-app-mt2024153
6  data:
7    local-config.yaml: |
8      auth_enabled: false
9
10   server:
11     http_listen_port: 3100
12     log_level: info
13
14   ingester:
15     lifecycler:
16       ring:
17         kvstore:
18           store: inmemory
19           replication_factor: 1
20           final_sleep: 0s
21         chunk_idle_period: 5m
22         chunk_retain_period: 30s
```



A screenshot of a GitHub code viewer displaying the file `loki-configmap.yaml`. The file is 68 lines long and 1.42 KB in size. The code defines various configurations for Loki, including chunking, schema, compaction, storage, limits, and table management.

```
21     chunk_idle_period: 5m
22     chunk_retain_period: 30s
23     max_transfer_retries: 0
24
25   schema_config:
26     configs:
27       - from: 2020-10-15
28         store: boltdb-shipper
29         object_store: filesystem
30         schema: v11
31         index:
32           prefix: index_
33           period: 24h
34
35   compactor:
36     working_directory: /loki/compactor
37     shared_store: filesystem
38
39   storage_config:
40     boltdb_shipper:
41       active_index_directory: /loki/index
42       cache_location: /loki/cache
43       shared_store: filesystem
44     filesystem:
45       directory: /loki/chunks
46
47   limits_config:
48     reject_old_samples: true
49     reject_old_samples_max_age: 168h
50
```



A screenshot of a GitHub code viewer displaying the file `loki-configmap.yaml`. The file is 68 lines long and 1.42 KB in size. The code continues from the previous snippet, defining additional configurations such as chunk store, table manager, and ruler.

```
42     cache_location: /loki/cache
43     shared_store: filesystem
44   filesystem:
45     directory: /loki/chunks
46
47   limits_config:
48     reject_old_samples: true
49     reject_old_samples_max_age: 168h
50
51   chunk_store_config:
52     max_look_back_period: 0s
53
54   table_manager:
55     retention_deletes_enabled: true
56     retention_period: 168h
57
58   ruler:
59     storage:
60       type: local
61       local:
62         directory: /tmp/rules
63       rule_path: /tmp/rules
64       alertmanager_url: http://localhost:9093
65     ring:
66       kvstore:
67         store: inmemory
68       enable_api: true
```

The configmap YAML specifies that `auth_enabled` is set to false which disables authentication, sets Loki's HTTP port and sets the log level as info.

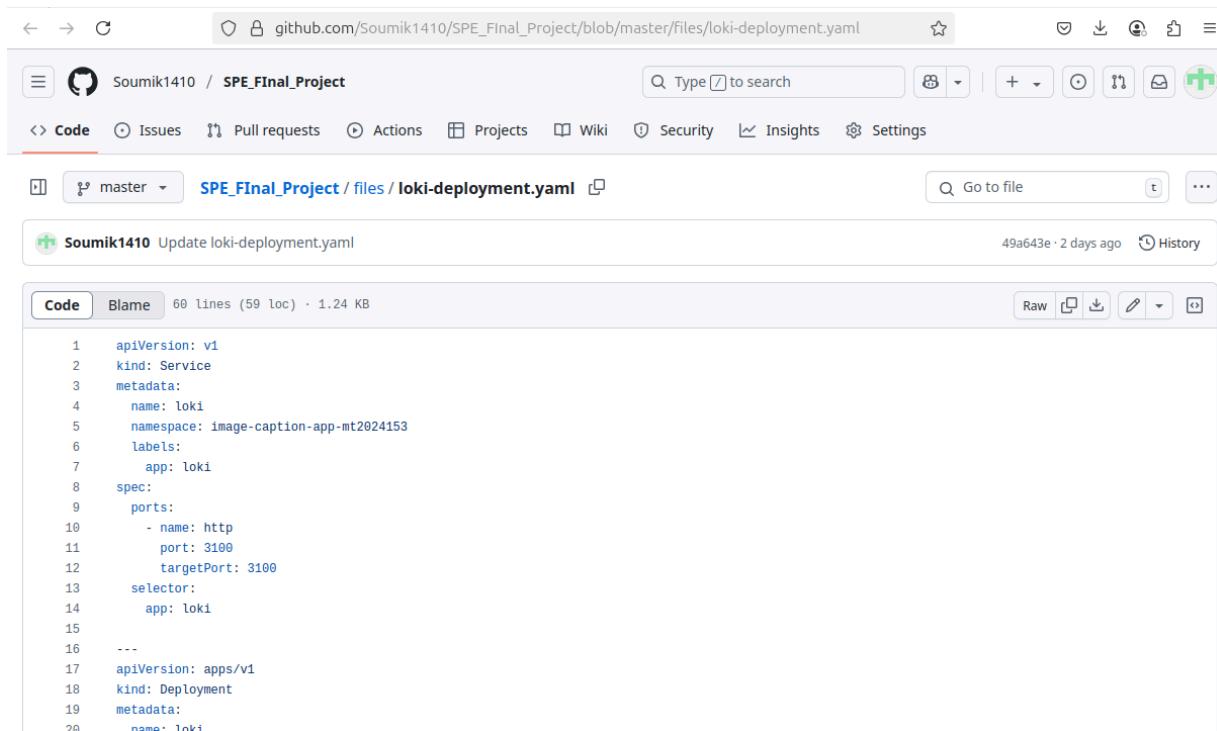
Then it specifies the details of the ingester, which handles incoming log data before it's written to storage, such as the lifecycle and ring participation of ingesters, time to wait before flushing an idle chunk, time to retain chunks after flushing.

Then it defines how logs are indexed and stored. In our case, we use BoltDB for storing indexes and storing log chunks on local disk. The compactor component compacts and manages index files, using the file system for compaction.

It also defines the storage paths for logs and directories for indices and cache. It also applies ingestion limits by rejecting old samples that are older than the max age (168h). It disables the look-back window for querying logs, sets retention period of log data to 7 days and points to Alertmanager for managing alerts & rule evaluations.

Then, the Loki deployment YAML is applied.

loki-deployment.yaml :



The screenshot shows a GitHub code editor interface. The top navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. Below the navigation bar, the repository name "Soumik1410 / SPE\_FInal\_Project" is displayed, along with a search bar and various file management icons. The current file is "loki-deployment.yaml". A commit history is visible, showing a recent update from user "Soumik1410" at "49a643e · 2 days ago". The main content area displays the YAML configuration for the Loki deployment:

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: loki
5   namespace: image-caption-app-mt2024153
6   labels:
7     app: loki
8 spec:
9   ports:
10    - name: http
11      port: 3100
12      targetPort: 3100
13   selector:
14     app: loki
15
16 ---
17 apiVersion: apps/v1
18 kind: Deployment
19 metadata:
20   name: loki
```

```

19   metadata:
20     name: loki
21     namespace: image-caption-app-mt2024153
22     labels:
23       app: loki
24   spec:
25     replicas: 1
26     selector:
27       matchLabels:
28         app: loki
29     template:
30       metadata:
31         labels:
32           app: loki
33     spec:
34       initContainers:
35         - name: init-loki-storage
36           image: busybox
37           command: ["sh", "-c", "mkdir -p /loki/index /loki/cache /loki/chunks /loki/compactor"]
38       volumeMounts:
39         - name: loki-storage
40           mountPath: /loki
41     containers:
42       - name: loki
43         image: grafana/loki:2.9.4
44         ports:
45           - containerPort: 3100
46         args:
47           - "-config.file=/etc/loki/local-config.yaml"
48       volumeMounts:
49         - name: config
50           mountPath: /etc/loki
51         - name: loki-storage
52           mountPath: /loki
53     securityContext:
54       runAsUser: 0
55     volumes:
56       - name: config
57         configMap:
58           name: loki-config
59       - name: loki-storage
60         emptyDir: {}

```

Here, first a Service named loki is defined within the image-caption-app-mt2024153 namespace, which is a ClusterIP service exposing it only within the cluster via a stable virtual IP. It specifies that loki will be listening on the port 3100, which is exposed within the cluster and this was also the port specified in the fluent bit configmap's output block.

Then the deployment manifest defines the loki deployment which is applied in the same namespace & has the label loki. Then in the spec field, the desired behaviour is defined, including the number of replicas which is set to 1, the label to identify loki pods and apply to those pods. initContainers runs before the loki container is brought up to prepare the storage folders,

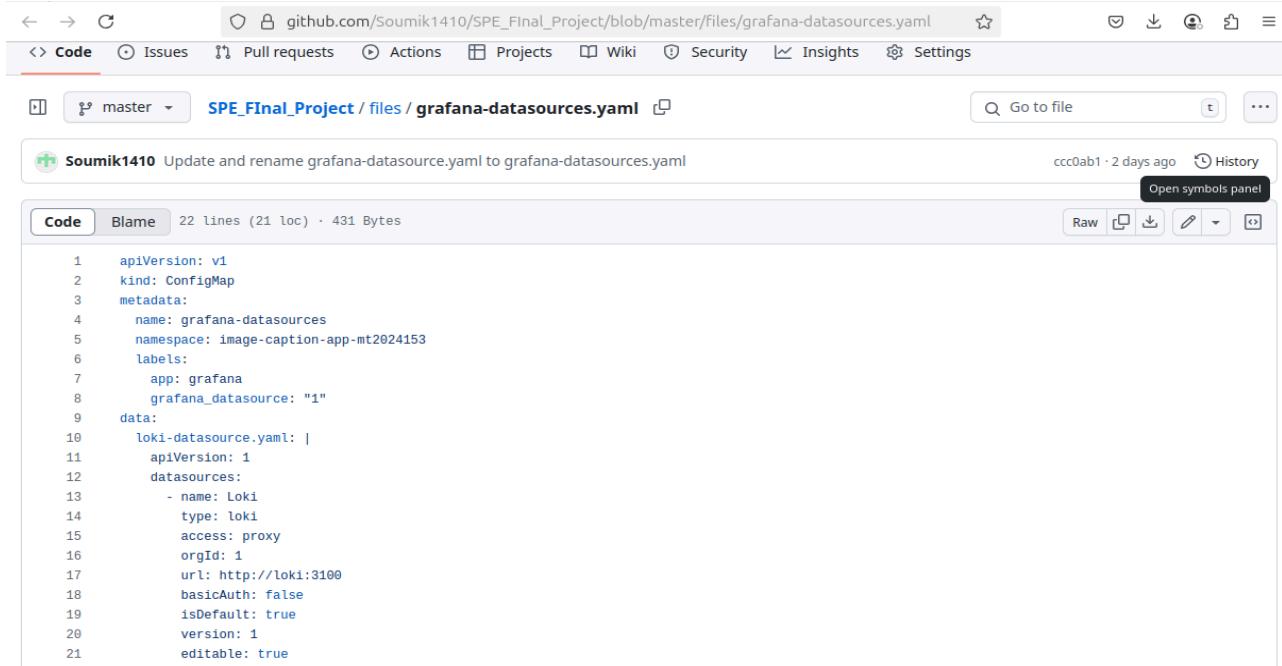
using a lightweight Busybox image. It executes a shell command to create the storage folders where loki will write to, then mounts these host directories at /loki. Then the container block specifies details such as the loki image to run, expose port 3100, the loki config yaml is passed as an arg and also present at one of the volumes mounted, also the loki storage folders volume mounted at /loki.

Finally it lists the sources used for the mounted volumes, the loki configmap for the config volume and an empty in-memory temporary directory for the loki storage folders volume.

Once these configurations are applied to the cluster, the Loki pod will be up and running, and log forwarding from Fluent Bit to Loki will be seamlessly established.

Finally, to deploy Grafana, we need to set its datasource as the Loki server we brought up and then apply its service and deployment manifests.

grafana-datasources.yaml :

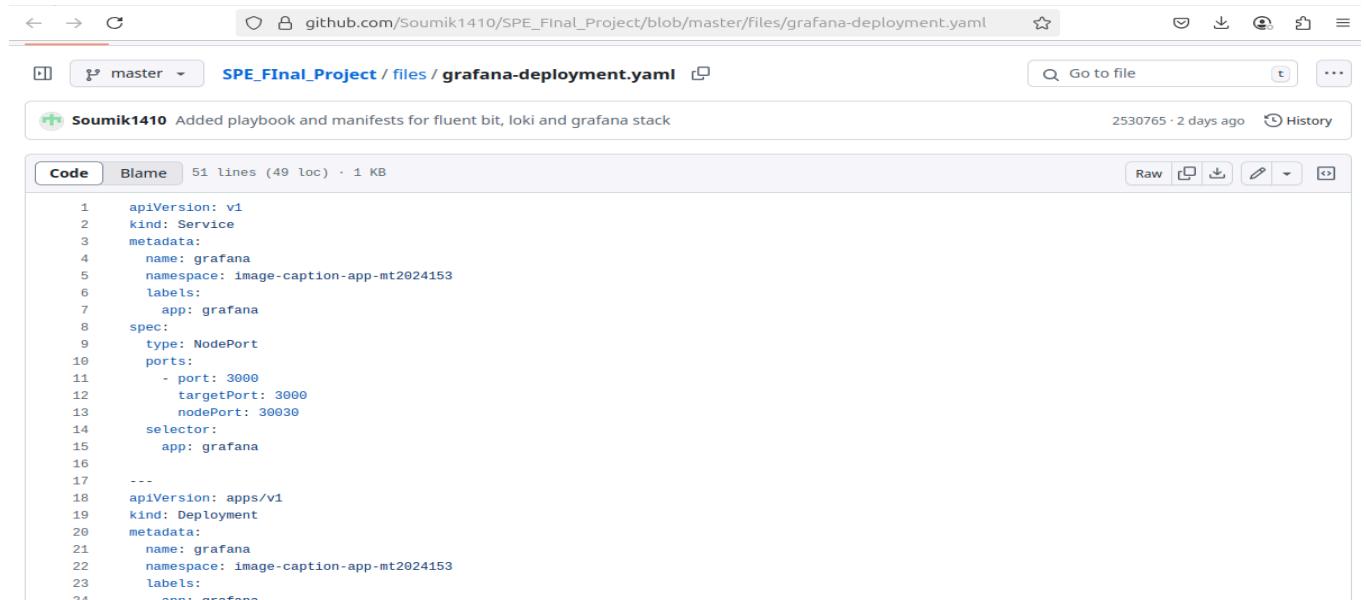


The screenshot shows a GitHub repository page for 'SPE\_Final\_Project' with the file 'grafana-datasources.yaml' open. The code editor displays the following YAML configuration:

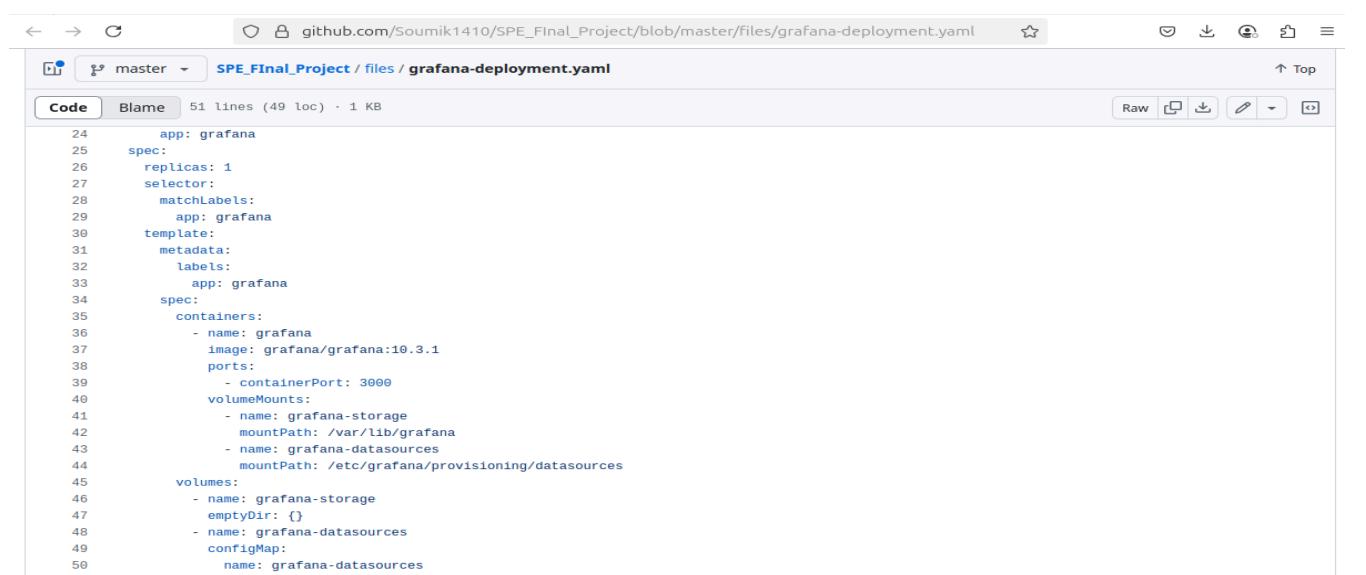
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-datasources
  namespace: image-caption-app-mt2024153
  labels:
    app: grafana
    grafana_datasource: "1"
data:
  loki-datasource.yaml: |
    apiVersion: 1
    datasources:
      - name: loki
        type: loki
        access: proxy
        orgId: 1
        url: http://loki:3100
        basicAuth: false
        isDefault: true
        version: 1
        editable: true
```

This defines a configMap named grafana-datasources in the namespace image-caption-app-mt2024153, which specifies the number of datasources as 1, and the datasource details as the loki server, its url, and other access details like authentication are provided. This allows Grafana to connect to the Loki server and query the stored data as required.

Finally the service and deployment manifests for Grafana are applied.  
grafana-deployment.yaml :



```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: grafana
5    namespace: image-caption-app-mt2024153
6    labels:
7      app: grafana
8  spec:
9    type: NodePort
10   ports:
11     - port: 3000
12       targetPort: 3000
13       nodePort: 30030
14   selector:
15     app: grafana
16
17 ---
18 apiVersion: apps/v1
19 kind: Deployment
20 metadata:
21   name: grafana
22   namespace: image-caption-app-mt2024153
23   labels:
24     app: grafana
```

```
24   app: grafana
25   spec:
26     replicas: 1
27     selector:
28       matchLabels:
29         app: grafana
30     template:
31       metadata:
32         labels:
33           app: grafana
34     spec:
35       containers:
36         - name: grafana
37           image: grafana/grafana:10.3.1
38         ports:
39           - containerPort: 3000
40         volumeMounts:
41           - name: grafana-storage
42             mountPath: /var/lib/grafana
43           - name: grafana-datasources
44             mountPath: /etc/grafana/provisioning/datasources
45       volumes:
46         - name: grafana-storage
47           emptyDir: {}
48         - name: grafana-datasources
49           configMap:
50             name: grafana-datasources
```

This YAML defines the deployment and service configuration for running Grafana inside the image-caption-app-**mt2024153** namespace.

The Service of type NodePort exposes Grafana outside the cluster. It maps port 3000 from the pod to the node's port 30030, making the Grafana dashboard accessible via <NodeIP>:30030 or <Minikube IP>:30080 in our case. The service uses a label selector (app: grafana) to route traffic to the correct pod.

The Deployment section creates a single replica of a Grafana pod using the specified Grafana image within the same namespace. The pod listens on port 3000, which is Grafana's default web UI port. Two volumes are mounted: one for Grafana's data storage using an ephemeral empty directory, and the other for accessing the configuration of data source.

Together, this setup ensures that Grafana starts up with datasource pre configured as the Loki server, and is reachable both from within the cluster and externally via the NodePort. Now the Fluent Bit to Loki to Grafana pipeline is fully established.

We can verify all the Kubernetes logging resources within the cluster with kubectl commands.

```
jenkins@soumik-VirtualBox:~$ minikube ip
192.168.49.2
jenkins@soumik-VirtualBox:~$ kubectl get pods -n image-caption-app-mt2024153
NAME                  READY   STATUS    RESTARTS   AGE
fluent-bit-8p44j      1/1     Running   0          76m
grafana-8559bcb694-mm224 1/1     Running   0          76m
image-caption-app-756c979f57-5966q 1/1     Running   0          76m
loki-87d578f98-rnt4l   1/1     Running   0          76m
jenkins@soumik-VirtualBox:~$ kubectl get configmaps -n image-caption-app-mt2024153
NAME        DATA   AGE
fluent-bit-config  2      77m
grafana-datasources 1      77m
kube-root-ca.crt  1      85m
loki-config      1      77m
jenkins@soumik-VirtualBox:~$ kubectl get svc -n image-caption-app-mt2024153
NAME           TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)        AGE
captioner-service  NodePort  10.101.120.69  <none>        80:30080/TCP  85m
grafana        NodePort  10.100.206.149  <none>        3000:30030/TCP 77m
loki           ClusterIP 10.110.107.180  <none>        3100/TCP    77m
jenkins@soumik-VirtualBox:~$ kubectl get daemonsets -n image-caption-app-mt2024153
NAME      DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
fluent-bit  1         1         1       1           1           <none>      77m
jenkins@soumik-VirtualBox:~$ █
```

We can see in fluent bit logs that chunks are being flushed to Loki and Loki responds with HTTP 204 success response code.

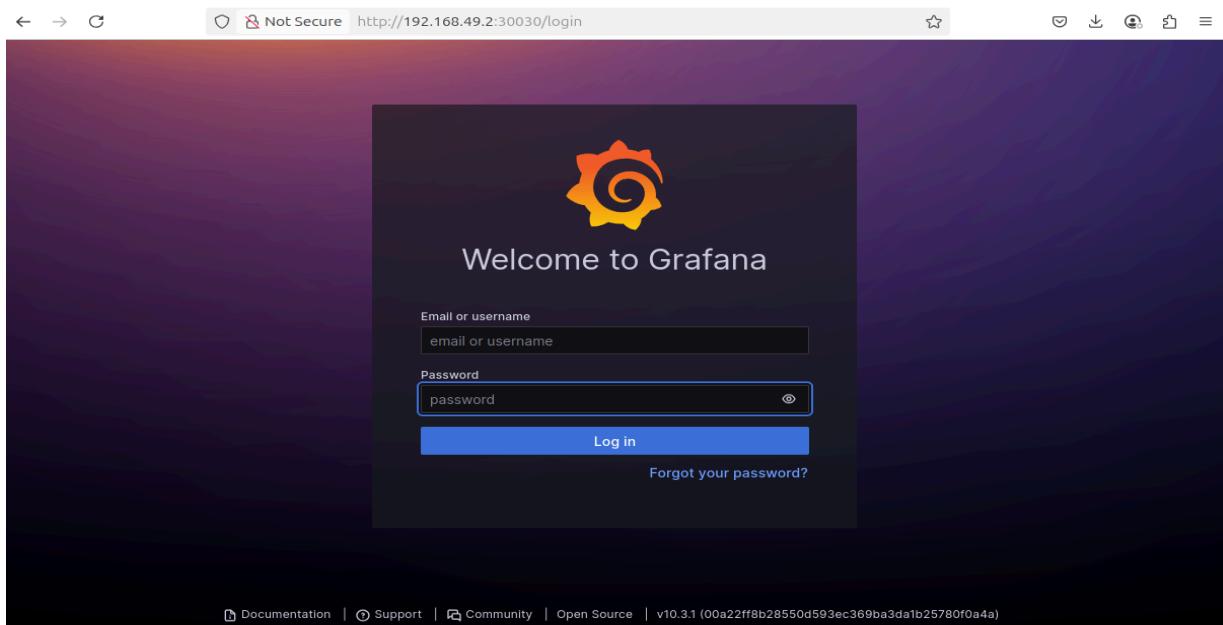
```
jenkins@soumik-VirtualBox:~$ kubectl logs fluent-bit-8p44j -n image-caption-app-mt2024153 | grep loki
[2025/05/22 23:17:23] [debug] [upstream] KA connection #53 to loki.image-caption-app-mt2024153.svc.cluster.local:3100 has been assigned (recycled)
[2025/05/22 23:17:23] [debug] [output:loki:loki.0] loki.image-caption-app-mt2024153.svc.cluster.local:3100, HTTP status=204
[2025/05/22 23:17:23] [debug] [upstream] KA connection #53 to loki.image-caption-app-mt2024153.svc.cluster.local:3100 is now available
```

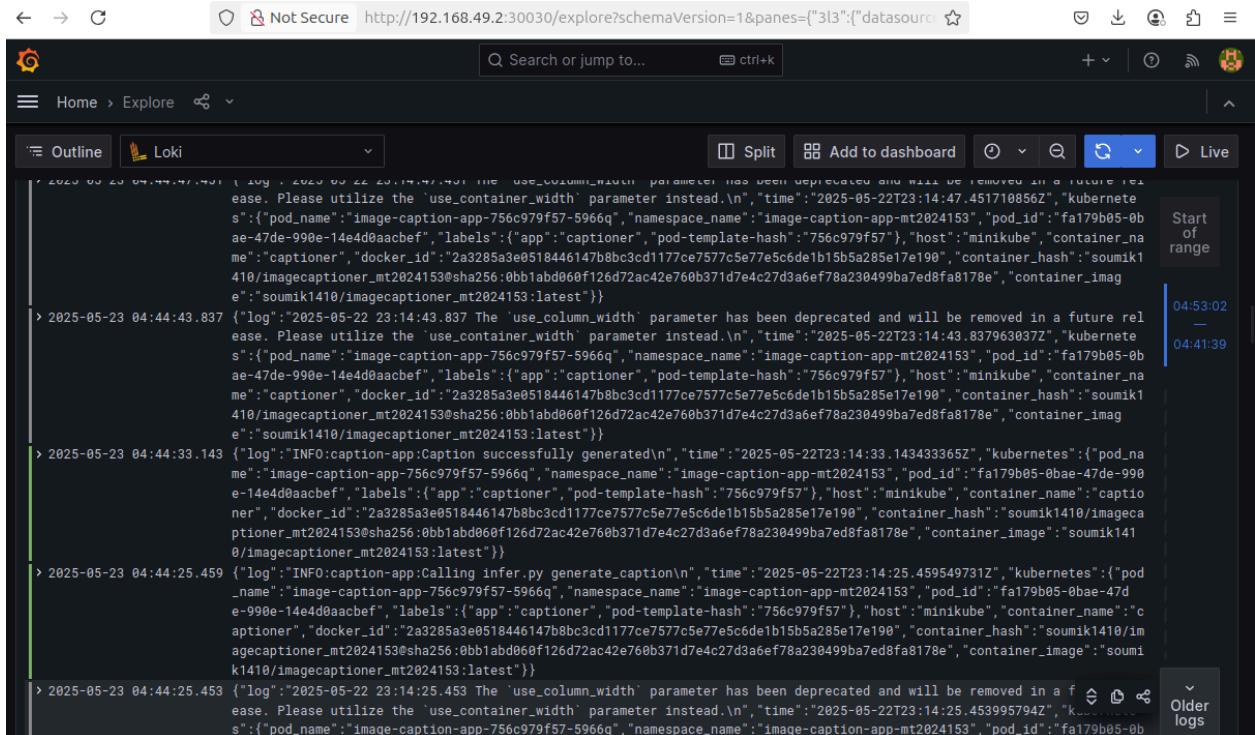
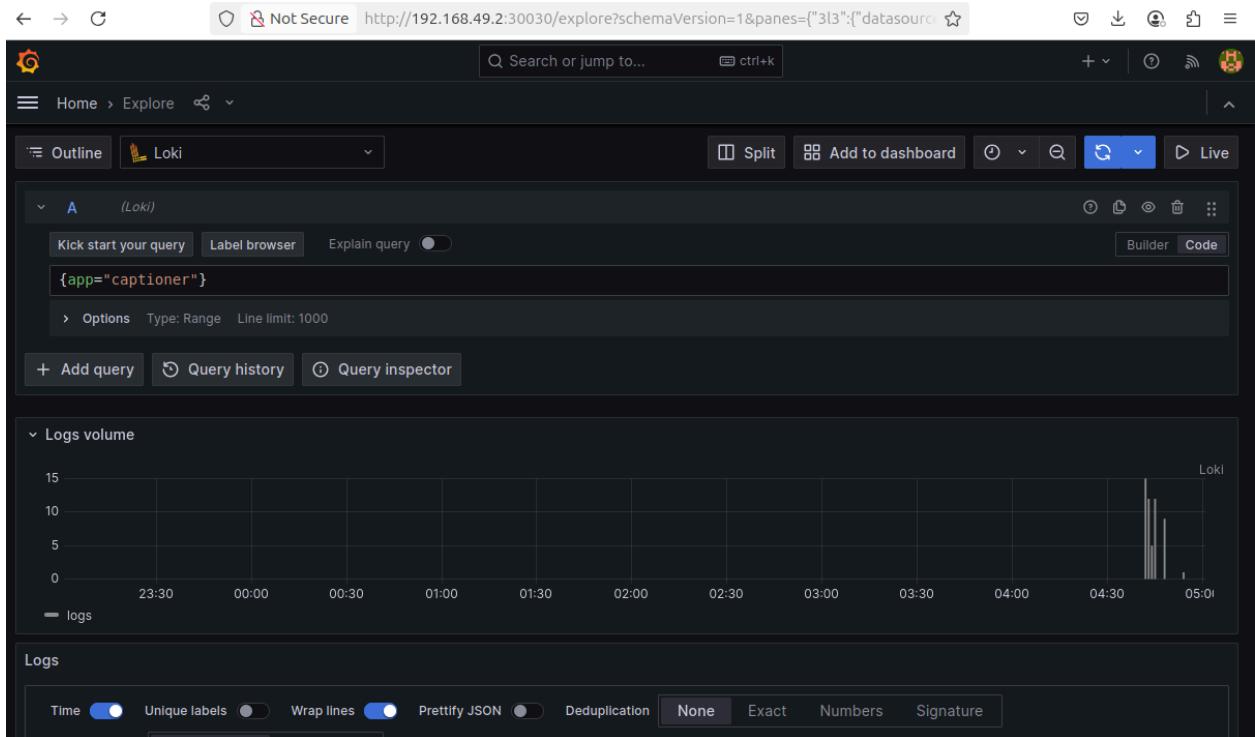
Similarly, we can see in Loki logs that it receives log chunks from Fluent-Bit.

```
jenkins@soumik-VirtualBox:~$ kubectl logs loki-87d57f98-rnt4l -n image-caption-app-mt2024153 | grep stream
Defaulted container "loki" out of: loki, init-loki-storage (init)
level=info ts=2025-05-22T22:37:16.528464395Z caller=flush.go:167 msg="flushing stream" user=fake fp=fd557ace0f1cf528 immediate=false num_chunks=1 labels={"controller_revision_hash="64876d45d7", k8s_app="fluent-bit", pod_template_generation="1\"}
level=info ts=2025-05-22T22:38:46.340265553Z caller=flush.go:167 msg="flushing stream" user=fake fp=fd557ace0f1cf528 immediate=false num_chunks=1 labels={"controller_revision_hash="64876d45d7", k8s_app="fluent-bit", pod_template_generation="1\"}"
```

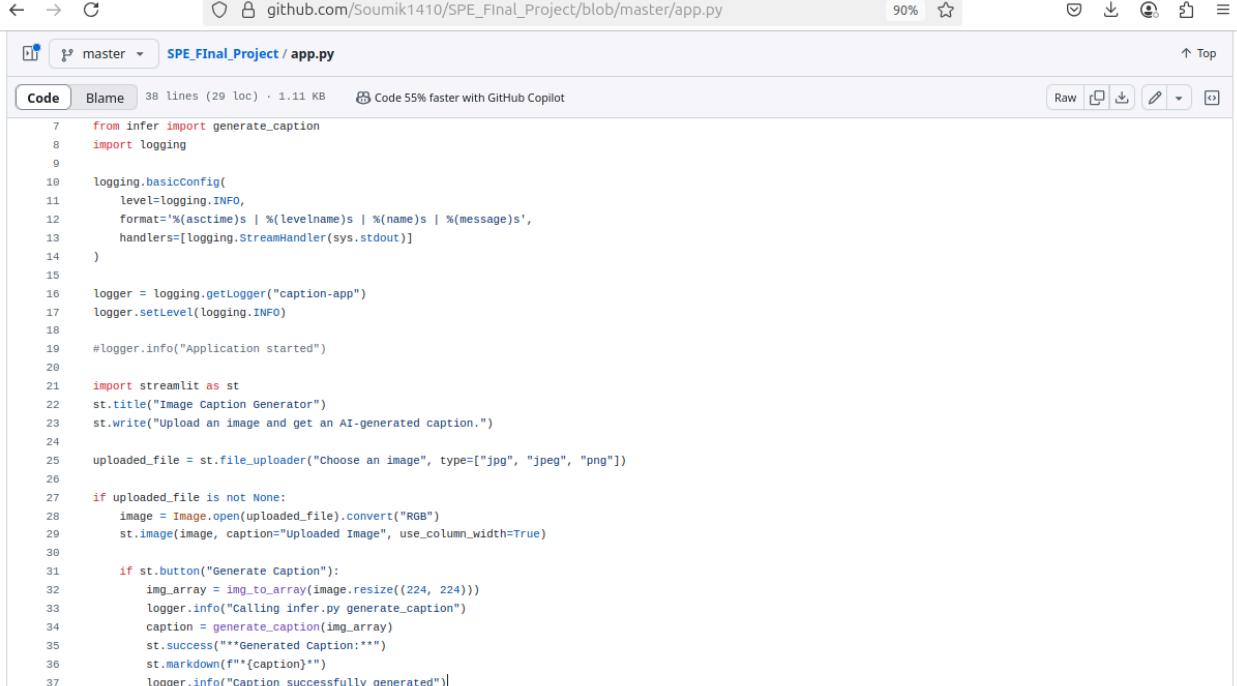
Finally, we can also verify in Grafana UI at <Minikube IP>:30030 , in the explore tab, when we set datasource as Loki and run a query, we can see the corresponding logs in the UI.

```
jenkins@soumik-VirtualBox:~$ minikube ip
192.168.49.2
jenkins@soumik-VirtualBox:~$ █
```





To ensure application logs are properly forwarded and not just system logs, we modified `app.py` to utilize Python's inbuilt module `logging` to log certain statements.



The screenshot shows a GitHub code editor interface for a file named 'app.py' in a repository titled 'SPE\_Final\_Project'. The code is written in Python and includes imports for 'infer' and 'logging', configuration for logging basic info with a specific format and handlers, and setup for a Streamlit application. The Streamlit app handles file uploads and generates captions using the 'infer' module's 'generate\_caption' function. It also logs success messages for generating captions.

```
from infer import generate_caption
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s | %(levelname)s | %(name)s | %(message)s',
    handlers=[logging.StreamHandler(sys.stdout)]
)

logger = logging.getLogger("caption-app")
logger.setLevel(logging.INFO)

#logger.info("Application started")

import streamlit as st
st.title("Image Caption Generator")
st.write("Upload an image and get an AI-generated caption.")

uploaded_file = st.file_uploader("Choose an image", type=["jpg", "jpeg", "png"])

if uploaded_file is not None:
    image = Image.open(uploaded_file).convert("RGB")
    st.image(image, caption="Uploaded Image", use_column_width=True)

if st.button("Generate Caption"):
    img_array = img_to_array(image.resize((224, 224)))
    logger.info("Calling infer.py generate_caption")
    caption = generate_caption(img_array)
    st.success(f"Generated Caption:{caption}")
    st.markdown(f"Caption:{caption}")
    logger.info("Caption successfully generated")
```

We can verify that in the above screenshot that our custom logging lines "Calling infer.py generate\_caption" and "Caption successfully generated" are also being picked up by Fluent Bit, forwarded to Loki and stored there, Grafana is able to query and retrieve it and show in the UI.

By leveraging a comprehensive suite of DevOps tools—including Git, GitHub, Jenkins, Docker, Docker Hub, Ansible with Roles, Kubernetes, HPA, and the Fluent Bit–Loki–Grafana stack—we have successfully built a robust CI/CD pipeline. This pipeline automates the entire workflow: from building, testing, and containerizing the application, to deploying it on a Kubernetes cluster. Post-deployment, the application metrics are continuously monitored through Horizontal Pod Autoscaling (HPA), while logs are collected, visualized, and analyzed to gain insights into performance and reliability.

Finally, the streamlit application is exposed on node port 30080 and is available to the external Internet at <Minikube IP>:30080 where given an

image, it utilizes the DenseNet201 to extract features and trained model to infer and generate a caption.

The screenshot shows a web browser window with the URL <http://192.168.49.2:30080>. The title bar reads "Image Caption Generator". Below the title bar, there is a message: "Upload an image and get an AI-generated caption." A "Choose an image" button is present. A file input field with a placeholder "Drag and drop file here" and a note "Limit 200MB per file • JPG, JPEG, PNG" is shown. A "Browse files" button is located to the right of the input field.

The screenshot shows the same web browser window after an image has been uploaded. The image of two dogs playing in the grass is displayed. Below the image, the text "Uploaded Image" is visible. A "Generate Caption" button is present below the image.

The screenshot shows the same web browser window after the caption has been generated. The image of the dogs is still displayed. Below the image, the text "Generated Caption:" is followed by the generated caption "two dogs are playing in the grass".