Algorithmic Trading

Team members:

Saanvi Vishal (IMT2021043)

Siddharth Chauhan (IMT2021046)

Kushal Partani (IMT2021062)

Links:

GitHub: https://github.com/SiddharthChauhan303/SPE-Project-kub-hpa.git

DockerHub: https://hub.docker.com/u/siddharth303

Introduction

This project focuses on building a robust algorithmic trading platform that leverages modern web technologies and machine learning techniques like Reinforcement Learning to deliver high-performance trading insights and execution. The algorithms used are DQN. The application is structured with a React frontend for an interactive user interface, a Node.js backend , and a Flask API that hosts the machine learning model for decision-making and predictive analytics.

To ensure scalability, reliability, and continuous integration, a Jenkins-based SDLC pipeline was implemented to automate code testing, building, and deployment. The project also utilizes Kubernetes for container orchestration, enabling efficient management of application scaling and deployment across multiple environments. This architecture ensures that the platform is optimized for high availability and real-time performance in the dynamic domain of algorithmic trading.

The project demonstrates an RL agent who can buy, sell and hold stocks.For the demonstration purpose we used the initial capital of 100000.

Flow of the Project

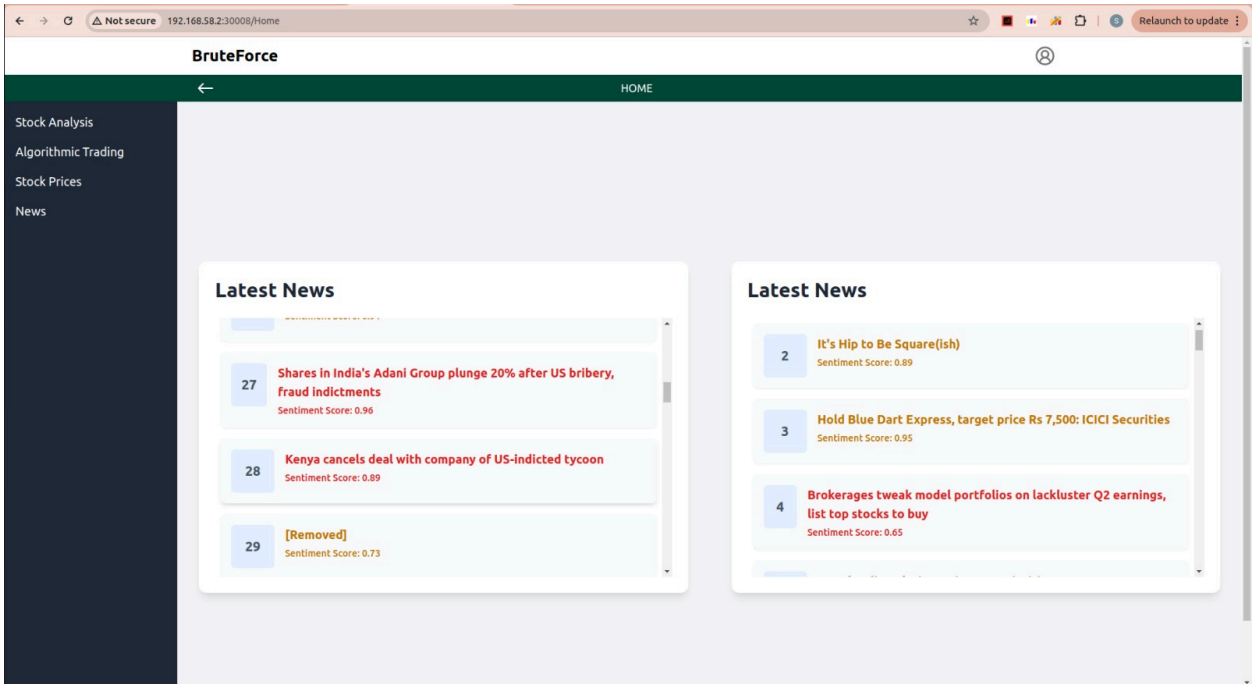
The algorithmic trading platform is designed to provide users with insights into financial markets through news analysis, candlestick charts, and model backtesting buy/sell signals, and portfolio value updates. The system architecture integrates a React frontend, a Node.js backend, and a machine learning model running in a Flask API. These components are containerized using Docker .The **frontend**, built with React, serves as the user interface.

- It includes a news section that displays sentiment-analyzed news from the last 15 days, leveraging FinBERT, a machine learning model for sentiment analysis.
- Another key feature of the frontend is the candlestick chart, which visualizes market trends, buy/sell signals, and portfolio values.
- The frontend interacts with the backend via API calls to fetch data and update the visualizations dynamically.

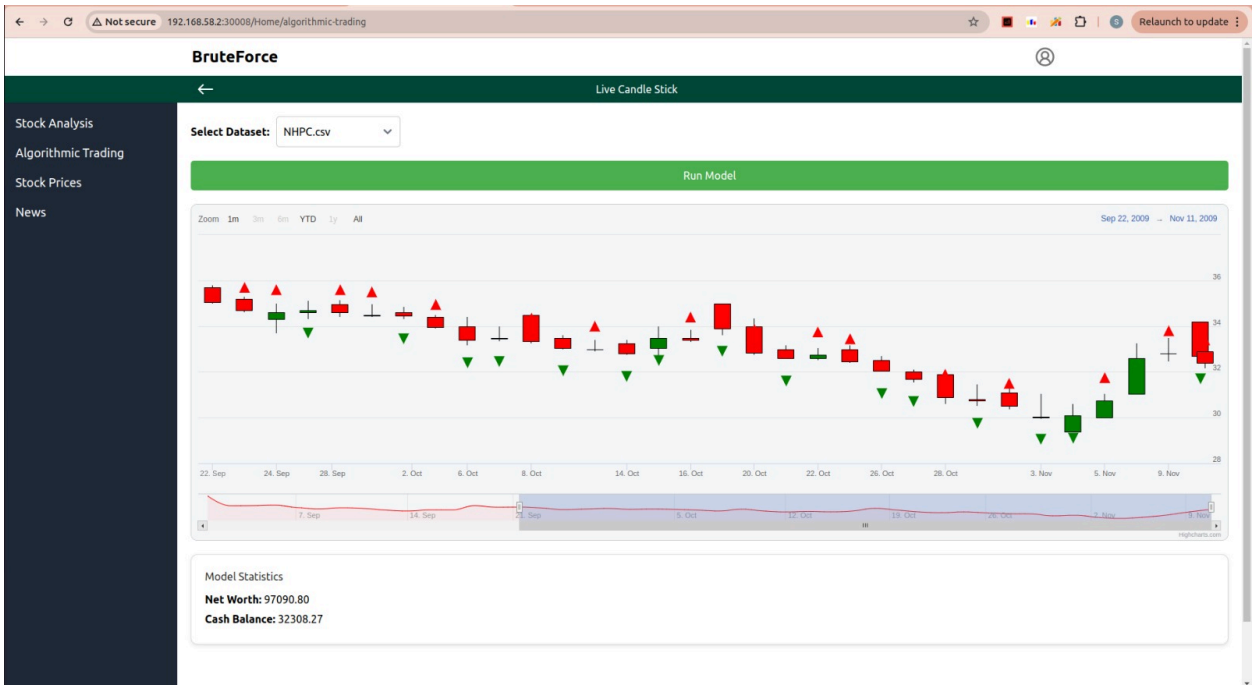
The **backend**, developed in Node.js, acts as a communication bridge between the frontend and the machine learning model. It handles requests from the frontend for sentiment scores, candlestick data, trading signals, and portfolio updates. Upon receiving a request, the backend processes the data or forwards the request to the model container, depending on the requirement. It also formats the processed data before sending it back to the frontend for display.

The **machine learning model** operates in a Flask API hosted within a dedicated Docker container. It comprises two main components: the FinBERT model for sentiment analysis and a reinforcement learning (RL) agent for generating trading signals and portfolio values. The FinBERT model uses an OpenAI API key to analyze news articles and derive sentiment scores. Meanwhile, the RL agent processes historical market data to produce buy/sell signals and calculate the final portfolio value.

When users interact with the **news section**, the backend sends a request which retrieves sentiment scores using the OpenAI API. The scores are then formatted by the backend and displayed in the frontend. For the **candlestick chart**, the frontend triggers a call to the backend, which, in turn, requests the RL agent in the model container to process market data. The agent outputs a CSV file containing buy/sell signals and portfolio metrics, which the backend processes and delivers to the frontend.



The entire system operates within a **containerized environment**. Three Docker images—`frontend`, `backend`, and `model`—are used for the React application, Node.js backend, and Flask API, respectively.



Docker Images

In the algorithmic trading platform, Docker was utilized to containerize the React frontend, Node.js backend, and Flask-based machine learning model. This approach provided several advantages. By isolating dependencies, we avoided conflicts between the environments required for the frontend, backend, and machine learning model. The modular design ensured that each component could be developed, debugged, and deployed independently, improving overall maintainability. Additionally, the separation allowed for scalability, enabling individual components to be scaled based on demand, such as increasing backend instances without impacting the frontend or model.

Docker image for model:

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install any necessary dependencies
RUN pip install -r requirements.txt

# Ensure the evaluation script is executable
RUN chmod +x inference.py

# Expose the input and output directories for the user to bind
VOLUME ["/input", "/output"]

EXPOSE 4000

# Define the command to run the script
ENTRYPOINT ["python", "evaluate_model.py"]
```

We use the Python:3.9 base image and install all dependencies with the RUN command in the Dockerfile. The Flask API model runs on port 4000, which is exposed for communication. The entry point for the container is set to the Flask application script, `evaluate_model.py`. The requirements consist of Tensorflow, numpy, pandas, flask etc.

evaluate_model.py

```python
from flask import Flask, request, jsonify
import os
import pandas as pd
import subprocess

app = Flask(__name__)

@app.route('/predict', methods=['POST','GET'])
def predict():
    data = request.json
    input_file = data.get('input_file')
    print(input_file)
    # if not input_file or not os.path.exists(input_file):
    #     return jsonify({'error': 'Invalid or missing input file'}), 400

    # Derive output file path
    output_file = input_file.replace('.csv', '_results.csv')

    try:
        # Run the inference script
        command = f"python inference.py app/{input_file}"
        result = subprocess.run(command, shell=True, capture_output=True)

        if result.returncode != 0:
            return jsonify({'error': result.stderr.decode()}), 500

        return jsonify({'output_file': output_file})

    except Exception as e:
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=4000)
```

This is the flask entry point in this docker image. It will run the inference model and then save the output in csv.

Docker image for Backend:

```
FROM node:18.12.1

# Install Docker CLI
RUN apt-get update && apt-get install -y \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg \
    lsb-release \
    && curl -fsSL https://download.docker.com/linux/debian/gpg | gpg --dearmor -o /usr/share/keyrings/do
    && echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring
    && apt-get update && apt-get install -y docker-ce-cli
RUN apt-get install -y docker.io

# Create App Directory
RUN mkdir -p /app
WORKDIR /app

# Install Dependencies
COPY package*.json ./
RUN npm install
RUN npm install -g nodemon

# Copy app source code
COPY . .

# Exports
EXPOSE 3000

CMD ["npm", "run", "dev"]
```

This is the dockerfile for the backend. Here we are installing all the npm libraries and server runs on port 3000, which is exposed for communication. To run the server we use the `CMD` option of the dockerfile.
We also tried installing the docker i.e using DIND (docker in docker).

Docker image for Frontend:

```
# Dockerfile for React frontend

# Build react frontend
FROM node:18

# Working directory be app
WORKDIR /app


COPY package*.json ./

###  Installing dependencies

RUN npm install

# copy local files to app folder
COPY . .

EXPOSE 5173

CMD ["npm","run","dev"]
```

This is the dockerfile for the frontend. Here we are installing all the npm libraries and server runs on port 5173, which is exposed for communication. To run the server we use the `CMD` option of the dockerfile.

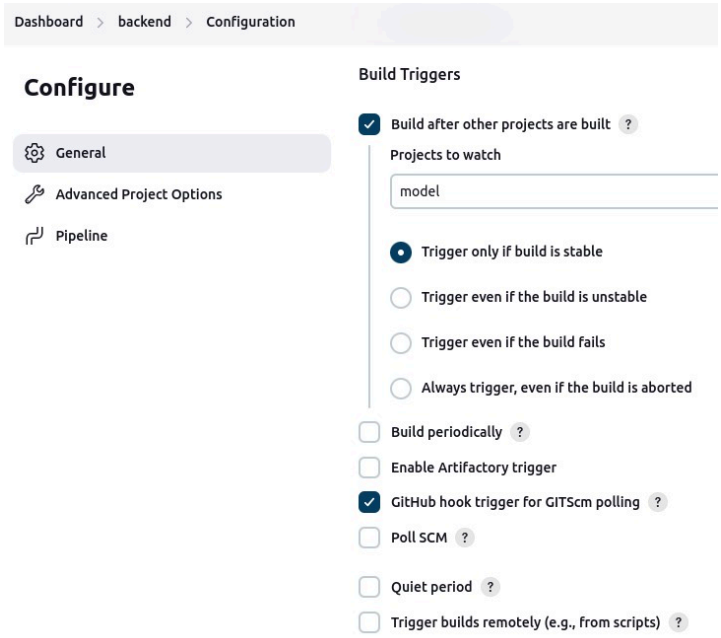
Jenkins Pipelines

We implemented four Jenkins pipelines to ensure modularity and maintain a clear dependency structure: model ➜ backend ➜ frontend ➜ final-project. Each pipeline is designed to build sequentially, with the next pipeline triggered only if the previous build is stable. This approach ensures a robust and error-free CI/CD process, streamlining integration and deployment across the project's components.

Model:

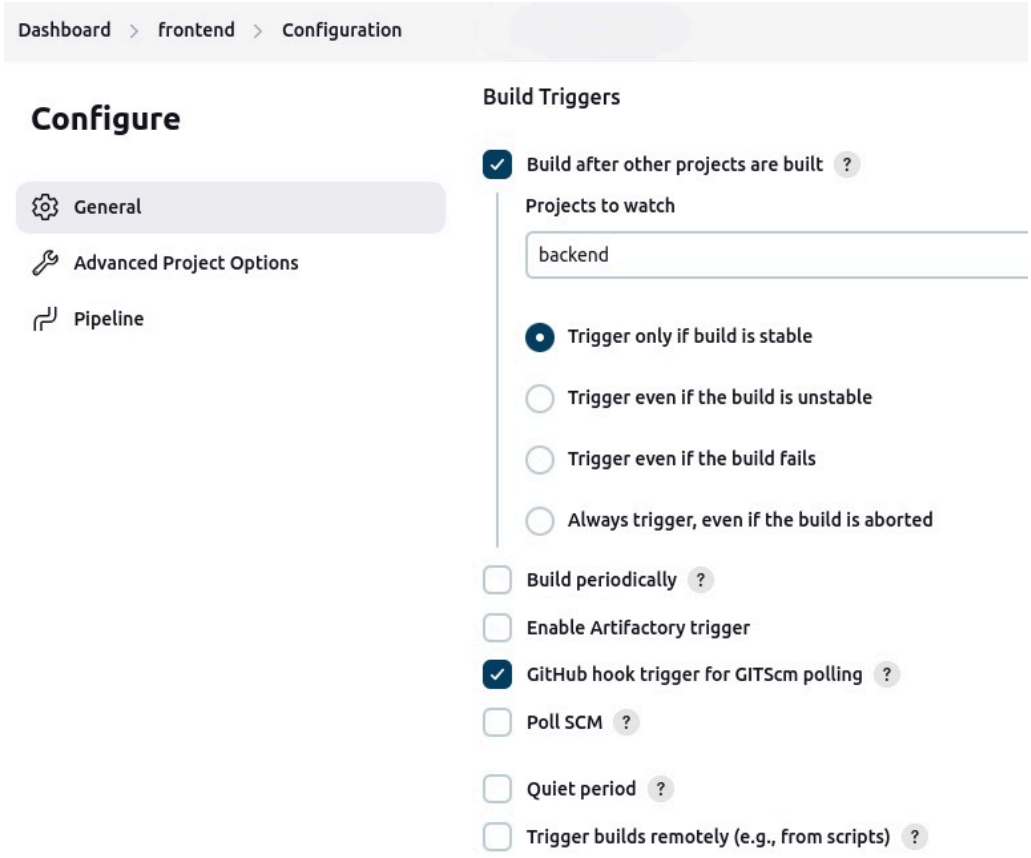
This Jenkins pipeline for the "model" stage includes multiple steps. The **Initialize** stage prepares the environment, and **Preprocess Data** ensures the data is ready for training. The model is trained and backtested, followed by **Unit Testing** for validation. Finally, the Docker image is built and pushed to DockerHub for deployment.

Backend:


Dashboard > backend > Configuration
Configure
General
Advanced Project Options
Pipeline
Build Triggers
Build after other projects are built
Projects to watch
model
Trigger only if build is stable
Trigger even if the build is unstable
Trigger even if the build fails
Always trigger, even if the build is aborted
Build periodically
Enable Artifactory trigger
GitHub hook trigger for GITScm polling
Poll SCM
Quiet period
Trigger builds remotely (e.g., from scripts)

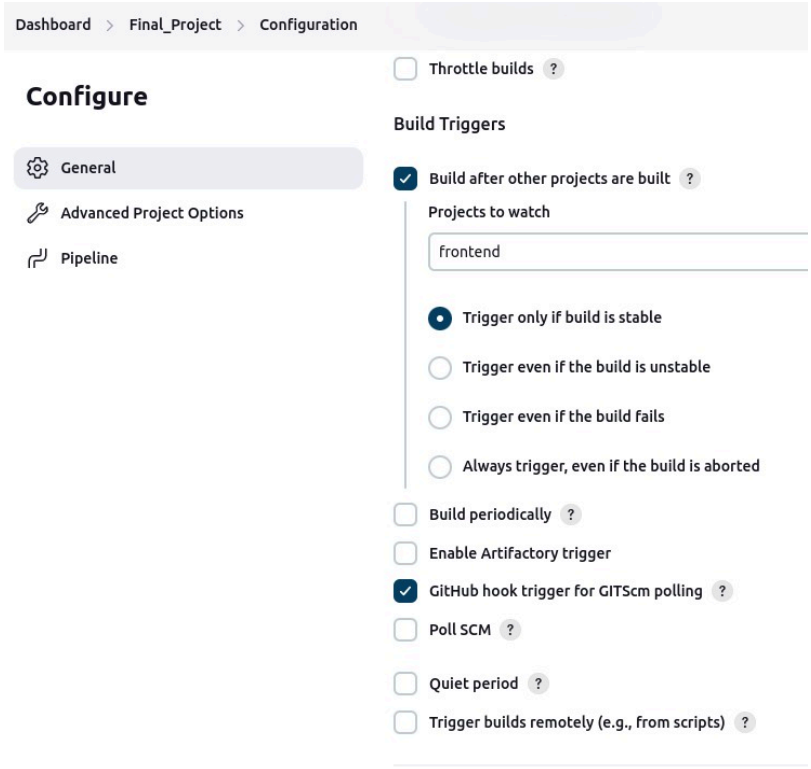

backend
Stage View
Declarative: Checkout SCM
Declarative: Tool Install
Git Clone
Backend Testing
Creating Docker Image for backend
Push Backend Docker Image
Average stage times: (Average full run time: ~2min 43s)
1s
161ms
1s
45s
23s
15s
#4 Dec 08 15:50 1 commit
1s
102ms
996ms
4s
1min 33s
1min 0s
#3 Dec 08 15:48 1 commit
1s
203ms
4s
10s failed
107ms failed
103ms failed
#2 Dec 08 15:46 1 commit
1s
107ms
908ms
8s failed
100ms failed
147ms failed
#1 Dec 08 15:41 No Changes
1s
235ms
1s
2min 36s aborted
71ms aborted
88ms aborted

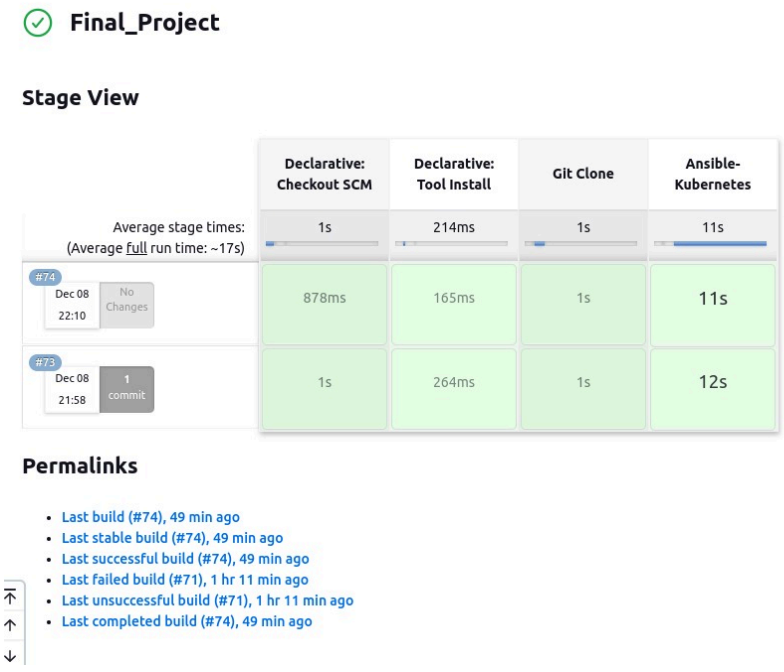
Frontend:


Dashboard > frontend > Configuration
Configure
General
Advanced Project Options
Pipeline
Build Triggers
Build after other projects are built
Projects to watch
backend
Trigger only if build is stable
Trigger even if the build is unstable
Trigger even if the build fails
Always trigger, even if the build is aborted
Build periodically
Enable Artifactory trigger
GitHub hook trigger for GITScm polling
Poll SCM
Quiet period
Trigger builds remotely (e.g., from scripts)


frontend
Stage View
Declarative: Checkout SCM
Declarative: Tool Install
Git Clone
Frontend Testing
Creating Docker Image for frontend
Push Frontend Docker Image
Average stage times:
(Average full run time: ~3min 28s)
1s
182ms
1s
28s
1min 36s
1min 16s
#1
Dec 08 15:56
No Changes
1s
182ms
1s
28s
1min 36s
1min 16s
Permalinks
Last build (#1), 7 hr 5 min ago
Last stable build (#1), 7 hr 5 min ago
Last successful build (#1), 7 hr 5 min ago
Last completed build (#1), 7 hr 5 min ago

Final Project:





Build History of Jenkins

S	Build	Time Since ↑	Status	
✓	Final_Project #75	18 sec	stable	
✓	frontend #2	2 min 58 sec	stable	
✓	backend #5	6 min 13 sec	stable	
✓	model #15	15 min	stable	

Docker-Compose

```yaml
services:
  server:
    image: siddharth303/backend-elk:latest
    container_name: myapp-node-server
    # command: npm install morgan --save && npm run dev
    ports:
      - "3000:3000"
    environment:
      - NODE_ENV=development
      - LOGSTASH_HOST=logstash:5044
    networks:
      - app-network
    depends_on:
      - model
      - client
      - logstash
    volumes:
      - /app/node_modules
      - /home/siddharth/Desktop/SPE-FInal-Project-main/app.log:/app/app.log
      - shared-dataset:/app/dataset
    dns:
      - 8.8.8.8

  client:
    image: siddharth303/frontend:latest
    container_name: myapp-react-client
    ports:
      - "5173:5173"
    environment:
      - NODE_ENV=development
    networks:
      - app-network
    volumes:
      - /app/node_modules

  model:
    image: siddharth303/model:latest
    container_name: myapp-model
    ports:
      - "4000:4000"
    networks:
      - app-network
    volumes:
      - shared-dataset:/app/dataset
```

This Docker Compose file defines a multi-service application with three main services: **server**, **client**, and **model**, each configured with specific roles and dependencies.

1. **Server Service**:
    - **Image**: Uses the `siddharth303/backend-elk:latest` image for the backend service.
    - **Container Name**: The container is named `myapp-node-server`.
    - **Ports**: Maps port `3000` inside the container to port `3000` on the host machine.
    - **Environment Variables**: Sets `NODE_ENV=development` and `LOGSTASH_HOST=logstash:5044` for the backend environment.
    - **Networks**: Connects to a custom network `app-network`.
    - **Dependencies**: Depends on the `model`, `client`, and `logstash` services, ensuring they start before the server.
    - **Volumes**: Maps local paths and shared datasets to paths inside the container for persistent data handling and log management.
    - **DNS Configuration**: Specifies an external DNS server (`8.8.8.8`).
2. **Client Service**:
    - **Image**: Uses the `siddharth303/frontend:latest` image for the React frontend.
    - **Container Name**: The container is named `myapp-react-client`.
    - **Ports**: Maps port `5173` inside the container to port `5173` on the host machine.

- **Environment Variables**: Sets `NODE_ENV=development` for the development environment.
- **Networks**: Connects to the shared `app-network`.
- **Volumes**: Mounts a local directory for `node_modules`.

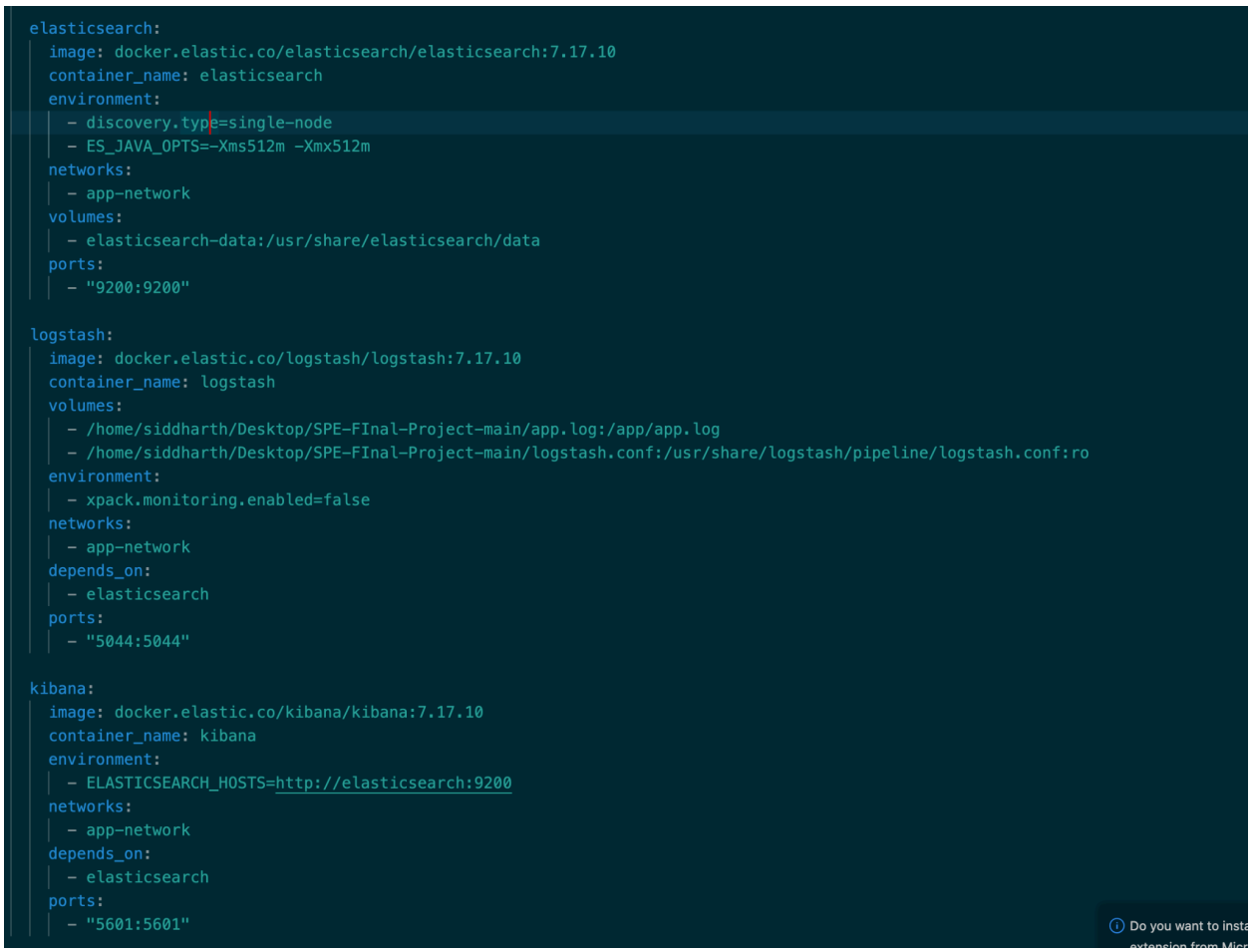
3. **Model Service**:
   - **Image**: Uses the `siddharth303/model:latest` image for the Flask-based model service.
   - **Container Name**: The container is named `myapp-model`.
   - **Ports**: Maps port `4000` inside the container to port `4000` on the host machine for API communication.
   - **Networks**: Connects to the shared `app-network`.
   - **Volumes**: Shares datasets between the container and the host using `shared-dataset`.

Networks: All services are connected to the `app-network` for inter-service communication.

Volumes: Persistent storage is configured to share data and logs between containers and the host.

Dependencies: The `depends_on` directive ensures proper startup order, with the backend waiting for the model and client services.

Monitoring using ELK stack in Docker-compose



```yaml
elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch:7.17.10
  container_name: elasticsearch
  environment:
    - discovery.type=single-node
    - ES_JAVA_OPTS=-Xms512m -Xmx512m
  networks:
    - app-network
  volumes:
    - elasticsearch-data:/usr/share/elasticsearch/data
  ports:
    - "9200:9200"

logstash:
  image: docker.elastic.co/logstash/logstash:7.17.10
  container_name: logstash
  volumes:
    - /home/siddharth/Desktop/SPE-FInal-Project-main/app.log:/app/app.log
    - /home/siddharth/Desktop/SPE-FInal-Project-main/logstash.conf:/usr/share/logstash/pipeline/logstash.conf:ro
  environment:
    - xpack.monitoring.enabled=false
  networks:
    - app-network
  depends_on:
    - elasticsearch
  ports:
    - "5044:5044"

kibana:
  image: docker.elastic.co/kibana/kibana:7.17.10
  container_name: kibana
  environment:
    - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
  networks:
    - app-network
  depends_on:
    - elasticsearch
  ports:
    - "5601:5601"
```

ELK is a powerful, open-source stack used for **log management**, **data processing**, and **visualization**. It consists of three primary components: **Elasticsearch**, **Logstash**, and **Kibana**, each playing a unique role in managing and analyzing large volumes of data.

1. Elasticsearch

Elasticsearch is a **search and analytics engine** designed for distributed data storage and querying. It indexes and stores structured, semi-structured, or unstructured data, making it searchable in near real-time. It is highly

scalable, capable of handling large amounts of data across multiple nodes, and offers advanced search features like full-text search, filtering, and aggregation.

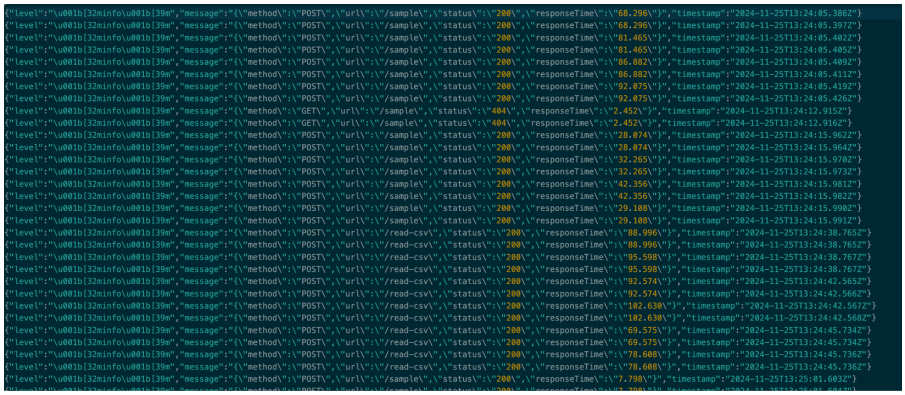
2. Logstash

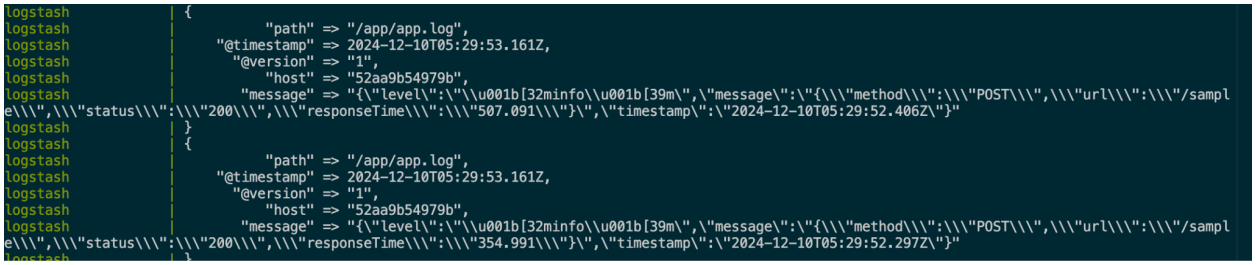
Logstash is a **data processing pipeline** that collects, parses, and transforms logs and other data streams. It acts as the central component for ingesting raw data, processing it into a structured format, and forwarding it to Elasticsearch or other destinations. Logstash supports a wide range of input sources, filters for transformation, and output destinations, making it highly flexible for processing logs, metrics, and application data.

3. Kibana

Kibana is a **visualization and analytics platform** designed to work with Elasticsearch. It provides an intuitive interface to explore data, create dashboards, and monitor logs and system performance in real-time. Kibana allows users to visualize data through charts, graphs, and maps, helping teams gain insights and make informed decisions.

Log file: This is the log file which was generated.



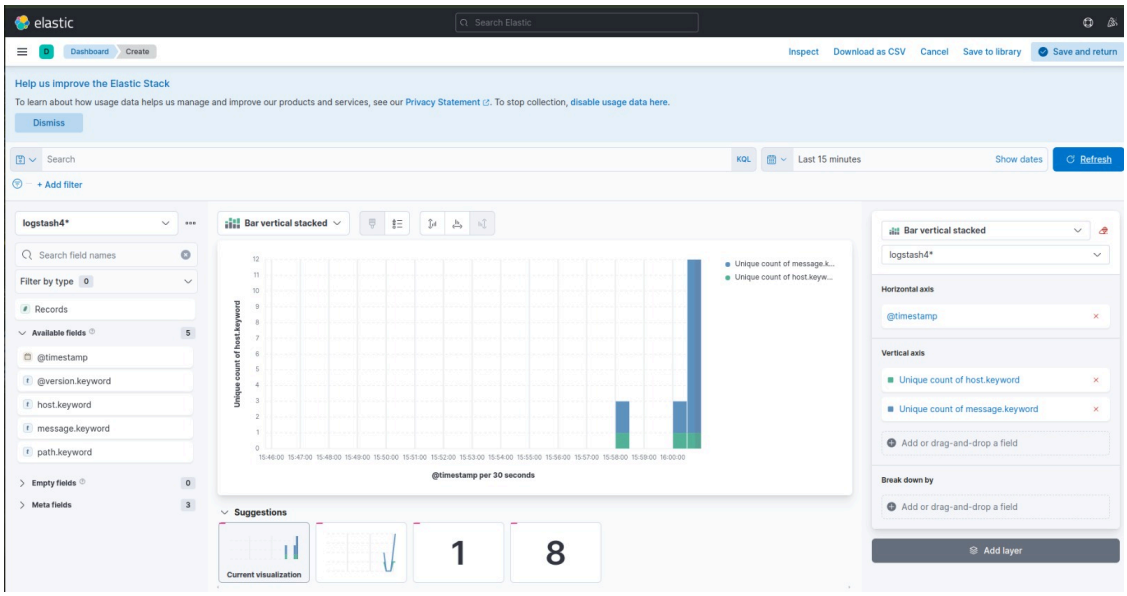


This log output appears to be from Logstash processing application logs. Each entry contains details such as:

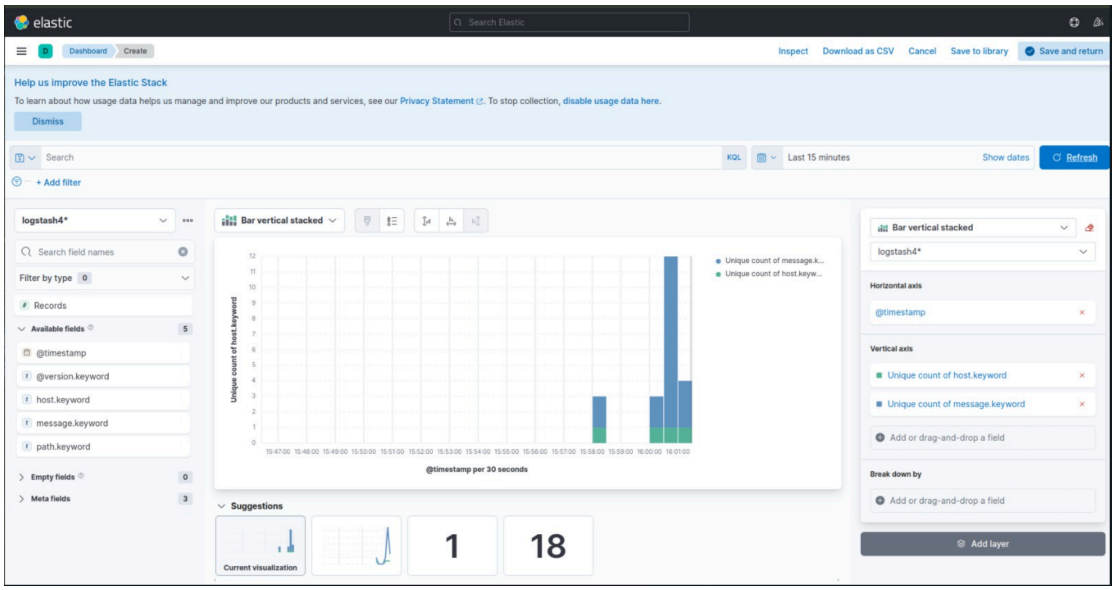
1. **Path**: The file source (`/app/app.log`) of the logs.
2. **Timestamp**: The precise time the log entry was processed (`2024-12-10T05:29:53.161Z`).
3. **Host**: The identifier of the host machine (`52aa9b54979b`).
4. **Message**: The actual log message, which is a JSON string including the HTTP request method (`POST`), URL (`/sample`), status code (`200`), and response time (e.g., `507.091ms`).

It shows structured data processed by Logstash for further analysis or visualization.

Visualisations using Kibana:



This Kibana interface visualizes log data indexed under `logstash4*`. The bar chart displays unique counts of `host.keyword` and `message.keyword` over time, with the `@timestamp` field on the horizontal axis. It provides real-time insights into log activity and host interactions.



This Kibana interface visualizes log data indexed under `logstash4*`. The bar chart represents the unique counts of `host.keyword` and `message.keyword` over time, using `@timestamp` as the horizontal axis. This visualization helps analyze log patterns and host interactions within a specific timeframe.

Dind (Docker-in-Docker):

For running the model in the backend we also tried Dind i.e Docker-in-Docker. Docker-in-Docker (DinD) is a setup where a Docker instance runs inside a Docker container. It is commonly used in CI/CD pipelines, allowing isolated environments for building, testing, and running containers without interfering with the host system. DinD provides flexibility for containerized workflows, especially for managing multiple Docker instances in a single system. However, it should be used cautiously in production due to potential security risks and resource overhead.

Kubernetes

1. `backend-deployment.yaml`

We used this file to define the deployment for our backend application, my-node-backend. The deployment ensures that there is always one instance (replica) of the backend running. The container runs with the image siddharth303/backendk:latest, which contains our backend application. We set the container to listen on port 3000. Additionally, we specified memory resource requirements, requesting 1GB and setting a limit of 1GB to prevent overuse of resources. This deployment makes sure that if the backend

container crashes or is deleted, Kubernetes will automatically recreate it to maintain stability.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-node-backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-node-backend
  template:
    metadata:
      labels:
        app: my-node-backend
    spec:
      containers:
        - name: my-node-backend
          image: siddharth303/backendk:latest
          ports:
            - containerPort: 3000
          resources:
            requests:
              memory: "1Gi" # Request 256 MB of memory
            limits:
              memory: "1Gi" # Limit to 512 MB of memory
                # Path inside the container
```

---

2. backend-hpa.yaml

This file defines the Horizontal Pod Autoscaler (HPA) for our backend service. The HPA monitors the my-node-backend deployment and automatically adjusts the number of pods (replicas) based on CPU usage. We configured it to scale between 1 and 5 replicas. If the average CPU utilization goes above 50%, the HPA will increase the number of pods to handle the load. Conversely, it will reduce the number of pods when the load decreases. This ensures our backend service remains efficient and responsive under varying workloads without manual intervention.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-node-backend-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-node-backend
  minReplicas: 1
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50  # Scale if average CP
```

---

3. backend-service.yaml

This file describes the Kubernetes service that exposes our backend application to the cluster and external clients. We used a NodePort service type, which allows the backend to be accessed via port 30007 on any node in the cluster. The service forwards requests to port 3000 of the backend container. The selector field ensures that traffic is directed only to the pods labeled my-node-backend. This configuration enables seamless communication with our backend application.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-node-backend-service
spec:
  type: NodePort
  selector:
    app: my-node-backend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
      nodePort: 30007
```

---

4. **frontend-deployment.yaml**

We defined the deployment for our frontend application, my-react-app, using this file. It ensures one instance of the frontend is always running. The container uses the image siddharth303/frontendk:latest and listens on port 5173. Just like the backend, we allocated 1GB of memory to the frontend container. This deployment ensures the frontend application is continuously available and that it restarts automatically if any issues occur.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-react-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-react-app
  template:
    metadata:
      labels:
        app: my-react-app
    spec:
      containers:
        - name: my-react-app
          image: siddharth303/frontendk:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 5173
          resources:
            requests:
              memory: "1Gi" # Request 256 megabytes of memory
            limits:
              memory: "1Gi" # Limit to 512 megabytes of memory
```

---

5. frontend-hpa.yaml

This file configures the Horizontal Pod Autoscaler (HPA) for our frontend service. It monitors the CPU usage of the my-react-app deployment and dynamically adjusts the number of pods between 1 and 5 replicas. If the CPU utilization exceeds 50%, additional pods are created to handle the increased load. This setup allows our frontend to scale automatically, ensuring a smooth user experience even under heavy traffic.

---

6. frontend-service.yaml

We used this file to define a Kubernetes service for our frontend application. The service is of type NodePort, making the frontend accessible through port 30008 on the cluster nodes. It forwards traffic to port 5173 of the frontend container. The selector field ensures that only pods with the label my-react-app receive the traffic. This setup allows external users to interact with our frontend interface.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-react-app-service
spec:
  type: NodePort
  selector:
    app: my-react-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5173
      nodePort: 30008
```

---

7. model-deployment.yaml

This file sets up the deployment for our machine learning model service, named my-model. Similar to the backend and frontend, we specified a single replica for the container running the siddharth303/modelk:latest image. The container listens on port 4000. Resource limits were also set, with 1GB allocated to ensure optimal performance. This deployment ensures the model service is always running and can restart automatically if necessary.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-model
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-model
  template:
    metadata:
      labels:
        app: my-model
    spec:
      containers:
        - name: my-model
          image: siddharth303/modelk:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 4000
          resources:
            requests:
              memory: "1Gi" # Request 256 MB of memory
            limits:
              memory: "1Gi" # Limit to 512 MB of memory
```

---

8. model-hpa.yaml

We configured this file to enable Horizontal Pod Autoscaling for the model service. The HPA monitors the my-model deployment and adjusts the number of replicas between 1 and 5 based on CPU utilization. If the CPU usage goes beyond 50%, additional replicas are created to distribute the workload. This ensures that the model service can handle increased requests without manual intervention.

---

9. model-service.yaml

This file defines the service for the machine learning model. Like the others, it uses a NodePort type, exposing the service on port 30009 for external access. It forwards incoming traffic to port 4000 of the container. The service is tied to pods labeled my-model, ensuring that traffic is routed correctly. This configuration allows users to interact with the model through a stable endpoint.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-model-service
spec:
  type: NodePort
  selector:
    app: my-model
  ports:
    - protocol: TCP
      port: 4000
      targetPort: 4000
      nodePort: 30009
```

These configurations were integral to deploying and managing our backend, frontend, and machine learning model services in a Kubernetes cluster. They ensure scalability, reliability, and efficient resource usage across all components of our system.

Ansible

```
[ansible_nodes]
localhost ansible_user=siddharth ansible_password=@my_vault.yml ansible_python_interpreter=/usr/bin/python3

[ansible_nodes:vars]
ansible_connection=local
```

This Ansible inventory configuration defines a group called `[ansible_nodes]` with a single localhost entry. The `ansible_user` is set to `siddharth`, and the `ansible_password` is retrieved from `@my_vault.yml` for secure storage. The Python interpreter is explicitly set to `/usr/bin/python3`. Under `[ansible_nodes:vars]`, the connection type is set to `local`, indicating that the playbooks will run directly on the localhost without SSH.

```yaml
- name: Deploying with Kubernetes
  hosts: all
  become: true
  tasks:
    - name: Show ansible_user
      debug:
        msg: "The ansible_user is {{ ansible_user }}"
    - name: Add directory to Git's safe list
      ansible.builtin.command:
        cmd: git config --global --add safe.directory /tmp/SPE-Project-kub-hpa
    - name: Clone the repository
      git:
        repo: https://github.com/SiddharthChauhan303/SPE-Project-kub-hpa.git
        dest: /tmp/SPE-Project-kub-hpa
    # - name: Delete
      # command: export KUBECONFIG=~/.kube/config
    - name: Apply Kubernetes Deployment
      ansible.builtin.command:
        cmd: kubectl apply -f /tmp/SPE-Project-kub-hpa/deployment --insecure-skip-tls-verify
    # - name:
      # command: sudo minikube status
```

This Ansible playbook automates the deployment of a Kubernetes application:

1. **Setup**: Targets all hosts and runs with elevated privileges.
2. **Show User**: Displays the current `ansible_user` for debugging purposes.
3. **Git Operations**: Adds a directory to Git's safe list and clones the repository to `/tmp/SPE-Project-kub-hpa`.
4. **Kubernetes Deployment**: Applies the Kubernetes deployment using `kubectl apply` with the `--insecure-skip-tls-verify` option to deploy resources from the cloned repository.

This ensures a streamlined deployment process for Kubernetes applications.

Vault

```
$ANSIBLE_VAULT;1.1;AES256
62643039636538323139666534343263376138383765346662333736356664643865643163383735
38623333346330323336613262353463616535306566336 40a3564323230613931613830633539 38
35366466643966313362376135306637663534323465636262333565336130623335343035663562
37346465623432646 60a303235313266626631313666326532613031363364633539623537643062
3738
```

This file contains the Ansible Vault-encrypted password for the localhost, which is referenced in the inventory file. The password is securely stored using AES256 encryption, as indicated by the `$ANSIBLE_VAULT` header. During runtime, Ansible decrypts this file to authenticate the localhost. By integrating it into the inventory, sensitive credentials are kept secure while enabling automated tasks. This approach ensures that the password is protected yet accessible when needed for Ansible playbook execution.

Monitoring in K8 using Grafana, Loki, Prometheus

1. Install Helm

Download and install Helm 3, a package manager for Kubernetes, using the following command:

```
curl
https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
bash
```

Purpose: Helm simplifies the deployment and management of Kubernetes applications.

---

2. Pull Required Docker Images Fetch the necessary Docker images for Loki, Promtail, and Grafana:

- `docker pull grafana/loki:2.9.3`
- `docker pull grafana/promtail:2.9.3`
- `docker pull grafana/grafana:latest`

Details:

- **Loki (grafana/loki:2.9.3):** A log aggregation system by Grafana Labs.
- **Promtail (grafana/promtail:2.9.3):** An agent for shipping logs to Loki.
- **Grafana (grafana/grafana:latest):** A web-based visualization platform.

---

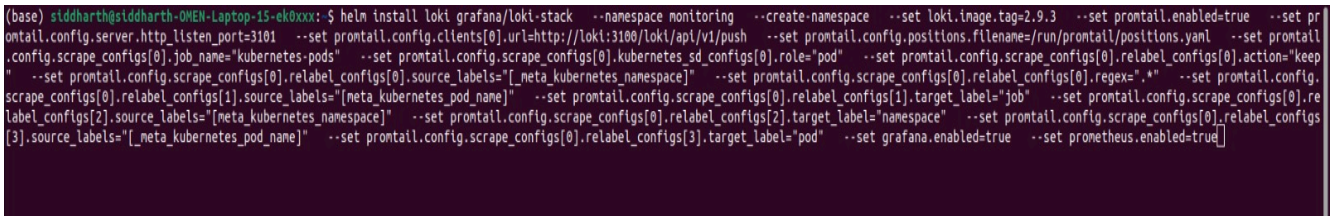
3. Add the Grafana Helm Chart Repository

Add the repository containing Helm charts for Grafana and its tools:
`helm repo add grafana https://grafana.github.io/helm-charts`

Purpose: Enables the use of Grafana's official Helm charts for deploying Loki, Promtail, and Grafana.

---

4. Deploy the Loki Stack



Use Helm to install the Loki stack in the `monitoring` namespace with customized configurations:

Configurations:

- Loki and Promtail are set up with specific configurations to scrape logs from Kubernetes pods.
- Grafana and Prometheus are enabled for visualization and metrics collection.

---

5. Set Up Port Forwarding

Forward ports to access Loki, Grafana, and Prometheus locally:

- Forward Prometheus to local port 9090:
  `kubectl port-forward --namespace monitoring svc/loki-prometheus-server 9090:80`
- Forward Grafana to local port 3020:
  `kubectl port-forward --namespace monitoring service/loki-grafana 3020:80`
- Forward Loki to local port 3100:
  `kubectl port-forward svc/loki -n monitoring 3100:3100`

  Purpose: Makes the services accessible locally for monitoring and visualization.

---

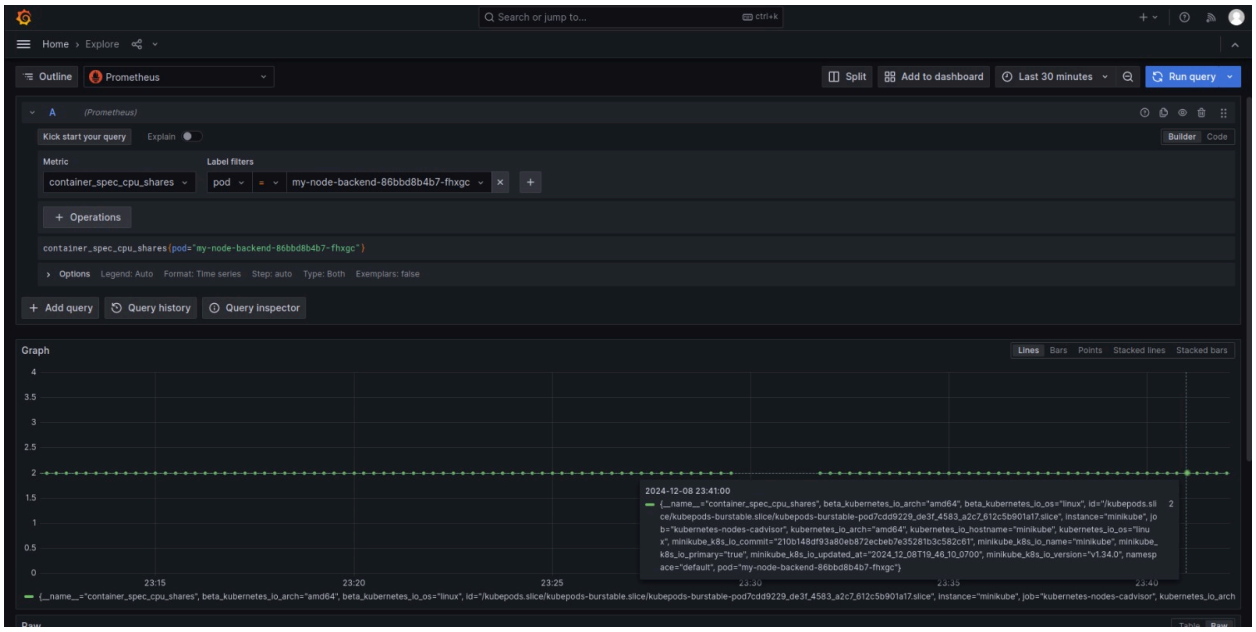
6. Retrieve Grafana Admin Password

Extract and decode the Grafana admin password:

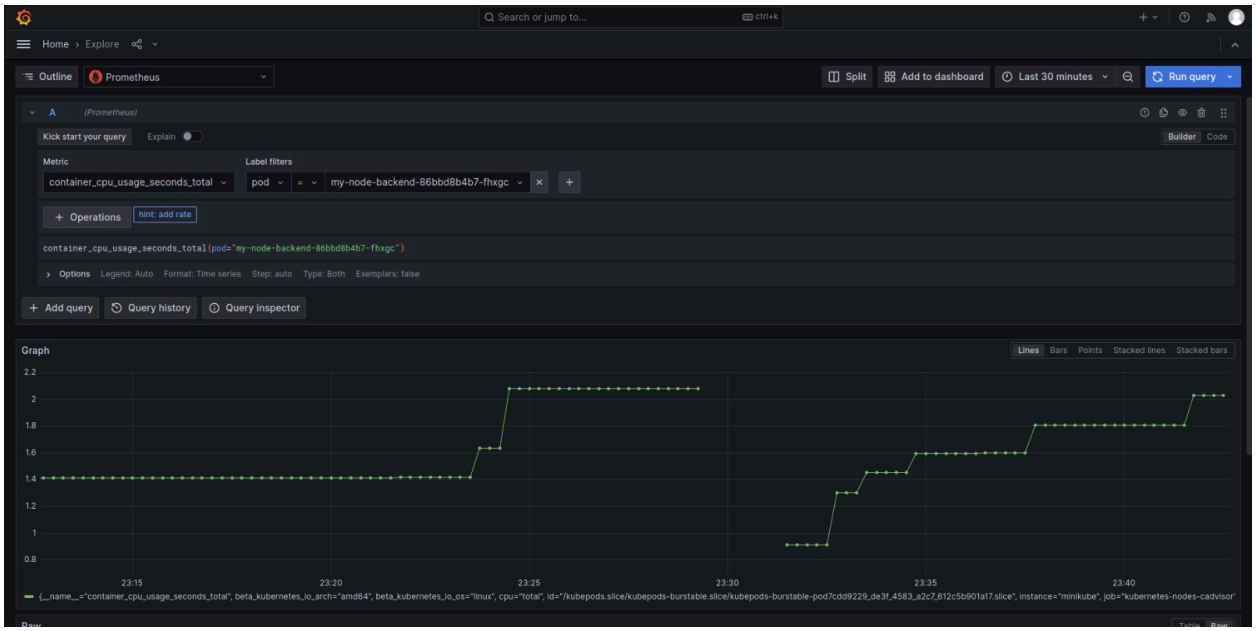
```
kubectl get secret loki-grafana -n monitoring -o jsonpath="{.data.admin-password}" | base64 --decode
```

Details: The password is stored in a Kubernetes secret named `loki-grafana`. This command decodes and displays it for use in logging into Grafana.

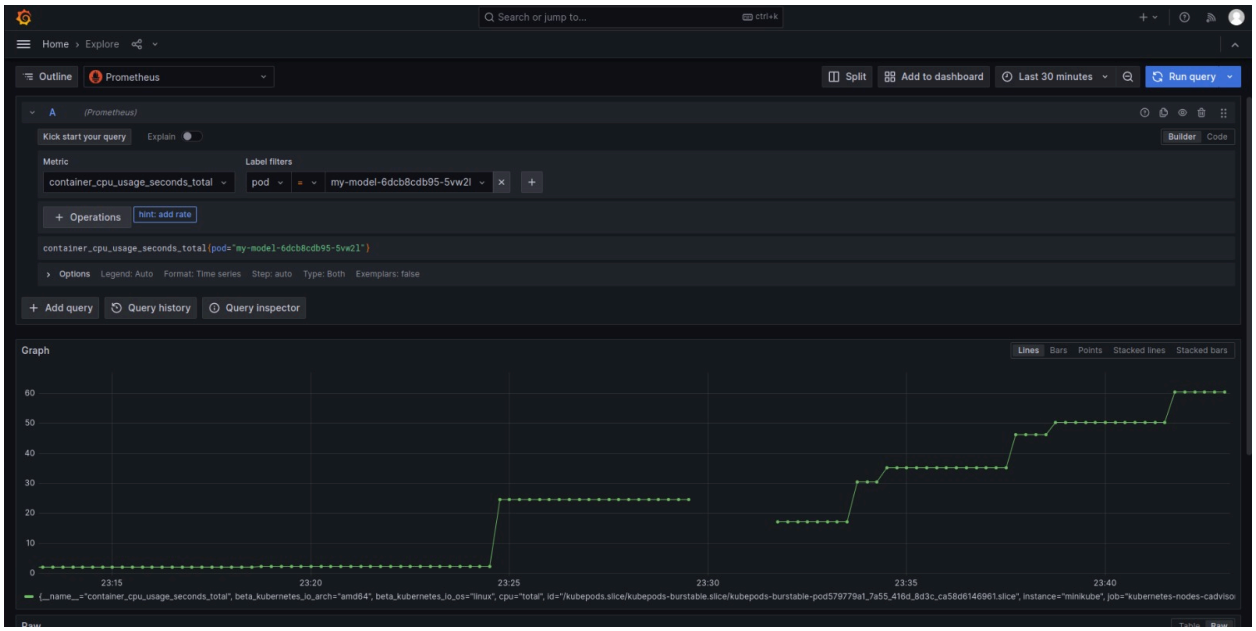
Visualisations using Grafana



This is a Grafana dashboard showing a Prometheus query for `container_spec_cpu_shares` of a specific Kubernetes pod (`my-node-backend`). The graph visualizes CPU resource allocation over time, allowing monitoring of pod resource usage.

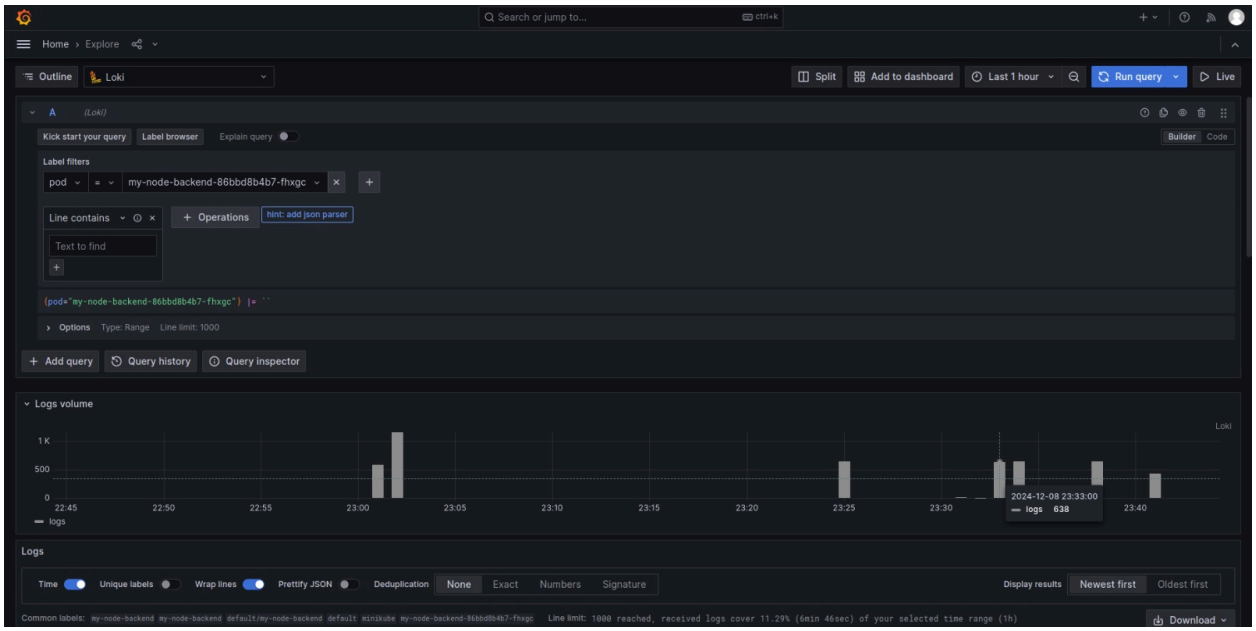


This Grafana dashboard displays a Prometheus query tracking `container_cpu_usage_seconds_total` for a specific Kubernetes pod (`my-node-backend`). The graph shows the cumulative CPU usage over time, helping monitor the pod's resource consumption trends.

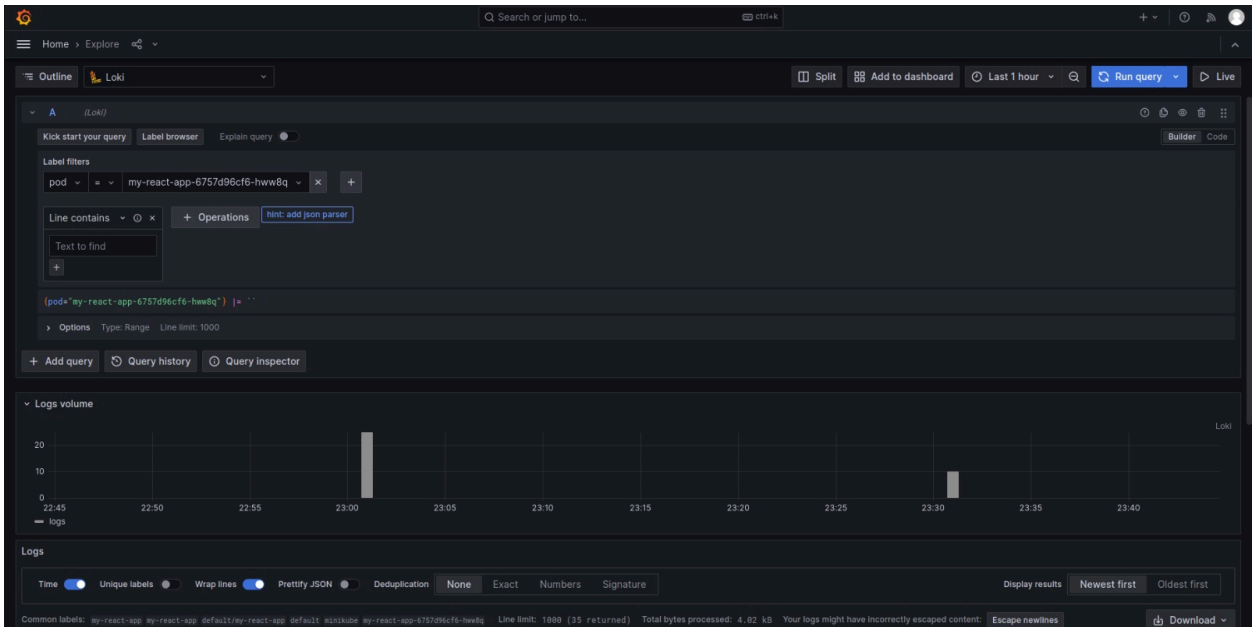


This Grafana dashboard shows a Prometheus query tracking `container_cpu_usage_seconds_total` for the `my-model` pod. The graph illustrates cumulative CPU usage over time, enabling monitoring of resource consumption for the machine learning model container.

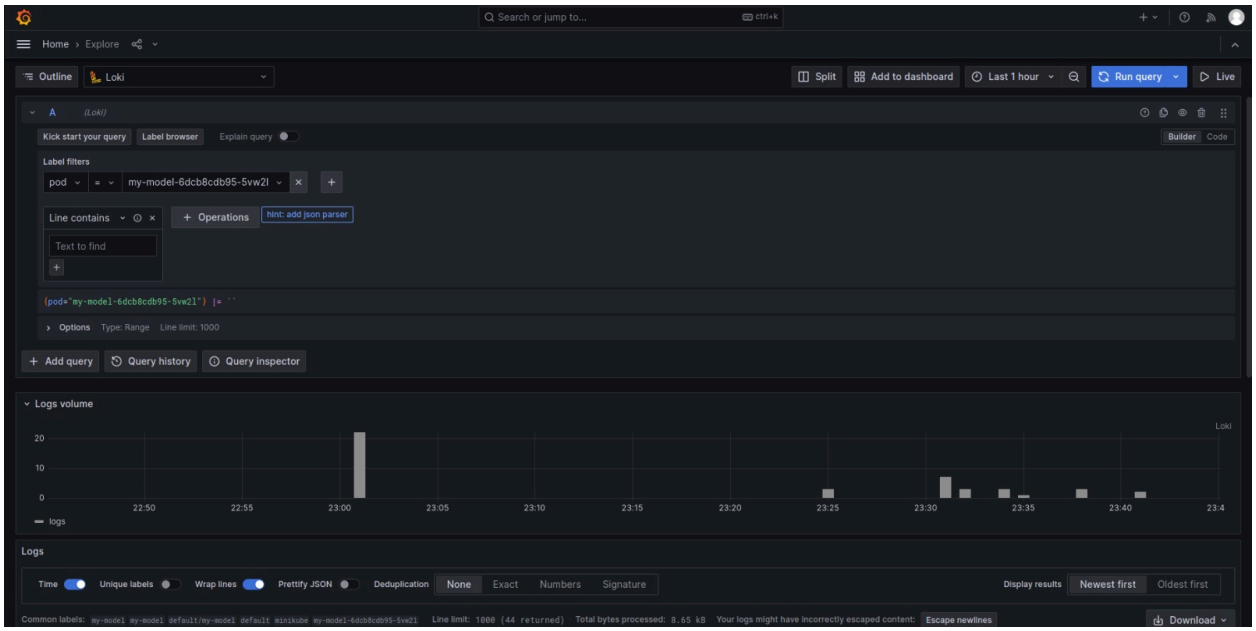
This Grafana dashboard tracks `container_cpu_usage_seconds_total` for the `my-react-app` pod. The graph displays the cumulative CPU usage, helping monitor the resource consumption of the React frontend container over time.



This Grafana dashboard, integrated with Loki, monitors logs for the `my-node-backend` pod. It visualizes log volume over time, enabling real-time analysis of application behavior and debugging through detailed log entries.



This Grafana dashboard, integrated with Loki, monitors logs for the `my-react-app` pod. It visualizes log volume over time, allowing developers to analyze and debug the React frontend's behavior through detailed log entries.



This Grafana dashboard, using Loki, monitors logs for the `my-model` pod. It visualizes log activity over time, providing insights into the model's runtime behavior and enabling easier debugging.