Technical Architecture

M.Srinivasan(IMT2021058), Ramsai Koushik Polisetti(IMT2021072)

Contents

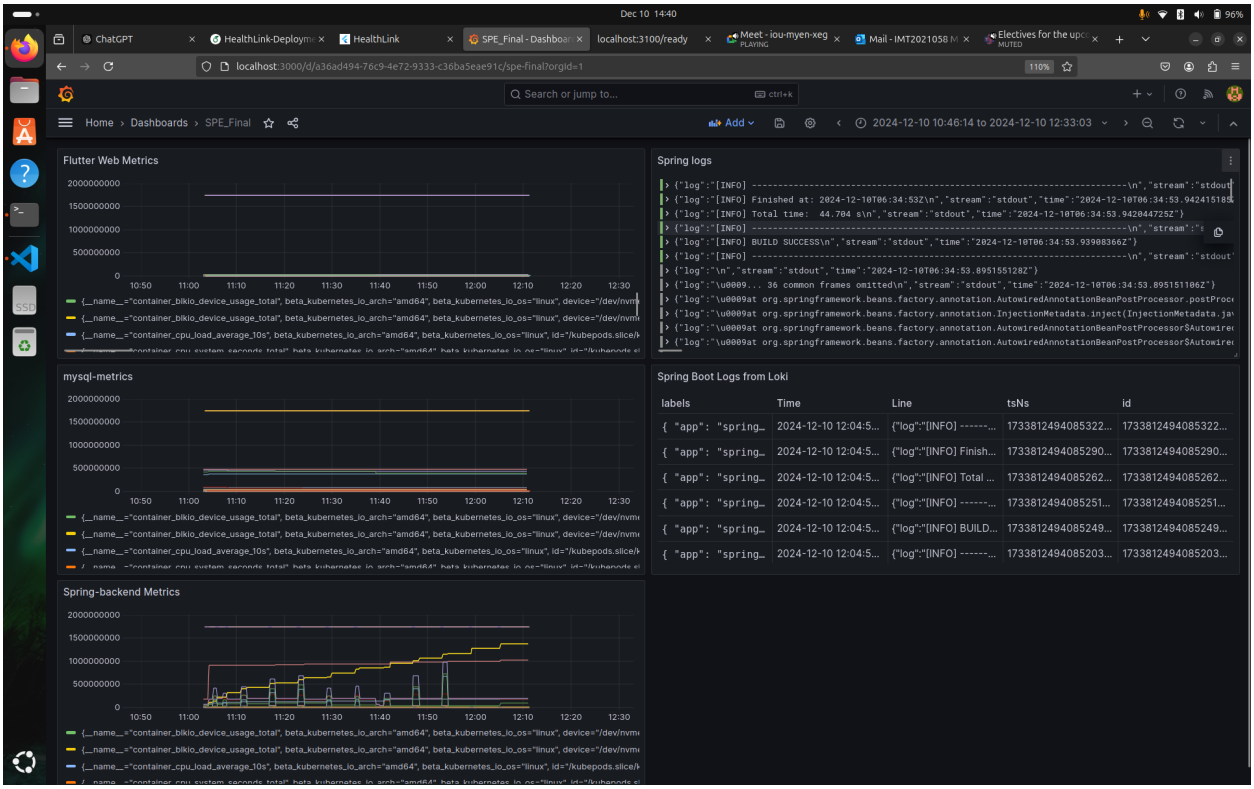

Figure 1: Overview of the Dashboard from the Grafana.

1 Project Description

Our project aimed to revolutionize healthcare by creating an AI/ML-powered medical chatbot application designed to assist patients in diagnosing health issues and connecting them with specialized doctors for consultations when necessary. Users can log in with basic authentication and provide their medical history, including allergies, medications, and demographics. The chatbot interacts with users, offering possible causes and remedies based on their symptoms. If a user needs a doctor's consultation, the app analyzes the chat context to recommend doctors specialized in the relevant medical field.

The app also supports consultations through text, phone calls, or video calls, ensuring convenience and accessibility and also allows doctors to review patients' medical histories during consultations. Post-consultation, a summary of the discussion and prescribed medications or guidelines is generated and saved for future reference. This comprehensive approach ensures continuity of care.

2 Project Tech Stack

Our healthcare chatbot application was built using the following tech stack:

- **Backend: Spring Boot**
- **Frontend: Flutter**
- **Database: MySQL**

Additionally, we integrated the following third-party APIs:

- **Cohere API:** Powered the AI/ML chatbot, offering personalized patient assistance based on medical queries.

- **Digital Samba API:** Enabled high-quality video consultations between patients and doctors, ensuring smooth, real-time communication.

3 Directory Structure of Codebase

The codebase is organized into a well-structured directory hierarchy as shown below:

```
flutter-frontend/
   healthlink/          # Frontend code
spring-backend/         # Backend code
deployment/
   frontend.yaml        # Deployment configuration for frontend
   backend.yaml         # Deployment configuration for backend
   mysql.yaml           # StatefulSet for persistent storage and PVC
deploy_stack.yaml       # Ansible deployment file
inventory.ini           # Ansible inventory file
Jenkinsfile             # Jenkins pipeline configuration file
```

The directory structure follows a modular design for ease of development and deployment. The frontend and backend are maintained separately, with deployment configurations housed in the `deployment/` folder. The Ansible deployment file `deploy_stack.yaml` automates the application deployment process, while `inventory.ini` defines server details. The CI/CD pipeline is managed using `Jenkinsfile`, ensuring streamlined automation and integration.

4 DevOps Overview

The DevOps pipeline for this project integrates several tools and processes to automate the Software Development Life Cycle (SDLC), from version control to deployment and monitoring. The process begins with version control using Git and GitHub, where developers commit code. This triggers an automated build in Jenkins, which serves as the Continuous Integration/Continuous Deployment (CI/CD) tool. Once the code is pushed, Jenkins runs automated tests to ensure the integrity of the code, builds Docker images for the frontend and backend, and pushes these images to Docker Hub.

The next step in the pipeline involves deploying the application using Kubernetes. After the Docker images are pushed to Docker Hub, an Ansible playbook is executed. The playbook contains instructions to pull the latest Docker images from Docker Hub and apply the necessary Kubernetes configuration files to the components of the application. The inventory.ini file specifies the IP addresses of the machines where the playbook should run, ensuring that the deployment occurs across the correct environments. This approach automates the process of deploying and managing the application within the Kubernetes cluster, while also ensuring that all services are orchestrated and configured properly.

For monitoring the application's performance and health, a monitoring stack consisting of Prometheus, Grafana, and Loki is utilized. Prometheus is used for collecting and storing metrics, while Loki handles the logs generated by the application. Grafana visualizes the data from Prometheus and Loki, providing a comprehensive view of the application's performance and activity. The Grafana dashboard is linked to both Prometheus and Loki, making it easy to monitor and visualize both metrics and logs in one interface.

5 DevOps Tech Stack

The project utilizes a variety of tools to automate and streamline the Software Development Life Cycle (SDLC). These tools are integrated to support version control, continuous integration, containerization, orchestration, configuration management, and monitoring.

5.1 Version Control: Git and GitHub

Git is used for version control, enabling collaborative development and maintaining a history of changes to the codebase. GitHub serves as the remote repository for the project, providing a platform for code storage, version management, and collaboration. It also integrates with Jenkins to trigger builds on code pushes.

5.2 Continuous Integration and Continuous Deployment: Jenkins

Jenkins is the central tool for Continuous Integration (CI) and Continuous Deployment (CD). It automates the process of code integration, testing, and deployment. Jenkins fetches the code from GitHub, runs automated tests, builds Docker images, pushes them to Docker Hub, and triggers deployment to Kubernetes through an Ansible playbook.

5.3 Containerization: Docker

Docker is used for containerizing the application, ensuring consistency across different environments. Dockerfiles are created for both the frontend and backend applications to define the environment in which they run. The application is then packaged into Docker images and stored in Docker Hub, from where they are pulled for deployment.

5.4 Orchestration: Kubernetes

Kubernetes (K8s) is used for orchestrating the deployment, scaling, and management of the application in a containerized environment. Kubernetes manages the lifecycle of Docker containers, ensuring that they are deployed, scaled, and maintained efficiently. The application is deployed using Kubernetes YAML configuration files that define the pods, services, deployments, and other components.

5.5 Configuration Management: Ansible

Ansible is used for configuration management and automating the deployment process. The Ansible playbook is responsible for pulling the latest Docker images from Docker Hub and applying the Kubernetes configuration files to the necessary components of the application. The playbook is executed on the machines specified in the inventory.ini file, ensuring that the deployment process is consistent across environments.

5.6 Monitoring: Prometheus, Grafana, and Loki

A stack consisting of Prometheus, Grafana, and Loki is used for monitoring. Prometheus collects and stores metrics from the application and Kubernetes cluster, while Loki handles the logs generated during the application's runtime. Grafana is used to visualize the metrics and logs, providing insights into the application's performance and health through dashboards that pull data from both Prometheus and Loki.

6 DevOps Implementation Details

6.1 Version Control Integration

Version control is essential for managing code changes. We used **Git** and **GitHub** for this project. The code is stored in a central GitHub repository: codebase url. The workflow follows:

- **Master branch**: Stable, production-ready code.
- **Feature branches**: For development and testing, merged into the main branch after completion.

Any push to the GitHub repository triggers the Jenkins CI/CD pipeline via **GitHub Webhooks**, automatically starting the build and deployment process.

6.2 Docker

Docker was used to containerize both the frontend and backend applications, ensuring consistent deployment across different environments. The following sections explain the contents of the Dockerfiles for each component in detail. A corresponding image of the Dockerfile will be placed above each explanation.

6.2.1 Frontend Dockerfile

- **Operating System and Dependencies**

```
FROM ubuntu:22.04

ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update
RUN apt-get install -y curl git wget unzip libgconf-2-4 gdb libstdc++6 \
    libglu1-mesa fonts-droid-fallback python3
RUN apt-get clean
```

- Explanation: The base image `ubuntu:22.04` is used. The environment variable `DEBIAN_FRONTEND` is set to `noninteractive` to suppress prompts during installation. Required packages are installed, and unnecessary files are removed afterward.

- **Flutter Setup**

```
ENV PUB_HOSTED_URL=https://pub.flutter-io.cn
ENV FLUTTER_STORAGE_BASE_URL=https://storage.flutter-io.cn

RUN git clone https://github.com/flutter/flutter.git /usr/local/flutter
ENV PATH="/usr/local/flutter/bin:/usr/local/flutter/bin/cache/dart-sdk/bin:${PATH
    ↪ }"
```

- Explanation: Environment variables for Flutter's services are defined to speed up downloads. The Flutter SDK is cloned and added to the system `PATH`.

- **Web Development Setup**

```
RUN flutter channel master
RUN flutter upgrade
RUN flutter config --enable-web
```

- Explanation: Flutter is switched to the `master` channel, updated, and configured for web development.

- **Build and Serve Application**

```
RUN mkdir /app/
COPY . /app/
WORKDIR /app/
RUN flutter build web

EXPOSE 9000
RUN ["chmod", "+x", "/app/server/server.sh"]
ENTRYPOINT [ "/app/server/server.sh" ]
```

- Explanation: The source code is copied to the container, built using `flutter build web`, and served on port `9000` using the startup script `server.sh`.

6.2.2 Backend Dockerfile

- **Base Image and Working Directory**

```
FROM maven:3.9.4-eclipse-temurin-17
WORKDIR /app
```

- Explanation: The base image `maven:3.9.4-eclipse-temurin-17`, equipped with OpenJDK 17 and Maven, is used. The working directory is set to `/app`.
- **Copy and Build Application**

```
COPY . .
EXPOSE 8080
ENTRYPOINT ["mvn", "spring-boot:run"]
```

- Explanation: The entire project is copied to the container. Port `8080` is exposed, and the application is started using the `mvn spring-boot:run` command.

6.2.3 Docker Workflow

After writing the Dockerfiles, the following commands were used to build and deploy the Docker images:

```
# Build Docker images
docker build -t <image_name> .

# Push images to Docker Hub
docker push <image_name>
```

The Docker images for both the frontend and backend were successfully built, tagged, and pushed to Docker Hub, enabling seamless deployment through Kubernetes.

6.3 Kubernetes Configuration

6.3.1 Introduction to Kubernetes

Kubernetes is an open-source platform for automating the deployment, scaling, and management of containerized applications. In the context of our project, Kubernetes plays a crucial role in managing the containerized services (Spring Boot backend, Flutter frontend, and MySQL database). It provides several advantages:

- **Scalability:** Kubernetes allows us to scale applications horizontally by adding or removing containers based on traffic or load.
- **Resilience:** It automatically manages failover in case a container crashes, ensuring the application remains available.
- **Automated Rollouts and Rollbacks:** Kubernetes can update containers in a controlled way, ensuring minimal disruption.
- **Service Discovery and Load Balancing:** Kubernetes automatically discovers services and load balances traffic to containers.

In this section, we will explain the Kubernetes configuration files that define the deployment of the Spring Boot backend, Flutter frontend, and MySQL database services. These configurations allow for the smooth management and scaling of our application.

6.3.2 Backend Configuration

- **Configuration file:**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-boot-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-boot-app
```

```
    template:
      metadata:
        labels:
          app: spring-boot-app
      spec:
        containers:
          - name: spring-boot-app
            image: srinivasan2404/spring-backend
            imagePullPolicy: Always
            ports:
              - containerPort: 5000
            env:
              - name: SPRING_DATASOURCE_URL
                value: "jdbc:mysql://mysql-service:3306/bytesynergy"
              - name: SPRING_DATASOURCE_USERNAME
                value: "ramsai"
              - name: SPRING_DATASOURCE_PASSWORD
                value: "ramsai@1"
---
apiVersion: v1
kind: Service
metadata:
  name: spring-boot-service
spec:
  selector:
    app: spring-boot-app
  ports:
    - protocol: TCP
      port: 5000
      targetPort: 5000
      nodePort: 30009
  type: NodePort
```

- **Explanation:** This configuration defines the deployment and service for the backend Spring Boot application. The backend service communicates with the MySQL database and serves the API to the frontend.
  - **Deployment:** We create a single replica of the Spring Boot application. The deployment uses the Docker image 'srinivasan2404/spring-backend' and sets the necessary environment variables for MySQL connectivity (username, password, and database URL).
  - **Service:** The service exposes the Spring Boot application on port 5000, and it's made available externally using a NodePort at port 30009. This allows access to the backend from outside the Kubernetes cluster.
- **Command to apply:**

```
kubectl apply -f backend.yaml
```

6.3.3 Frontend Configuration

- **Configuration File:**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: flutter-web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: flutter-web
  template:
    metadata:
      labels:
        app: flutter-web
```

```
    spec:
      containers:
        - name: flutter-web
          imagePullPolicy: Always
          image: srinivasan2404/flutter-web
          ports:
            - containerPort: 9000
---
apiVersion: v1
kind: Service
metadata:
  name: flutter-web-service
spec:
  selector:
    app: flutter-web
  ports:
    - protocol: TCP
      port: 9000
      targetPort: 9000
      nodePort: 30006
  type: NodePort
```

- **Explanation:** This configuration is responsible for deploying the Flutter web frontend. The frontend application provides the user interface for interacting with the backend.
  - **Deployment:** A single replica of the Flutter web application is deployed using the Docker image 'srinivasan2404/flutter-web'. It is exposed on port 9000.
  - **Service:** The service exposes the Flutter frontend on port 9000 and makes it accessible via a NodePort at port 30006. This ensures that users can access the frontend application from outside the Kubernetes cluster.
- **Command to apply:**

```
kubectl apply -f frontend.yaml
```

6.3.4 MySQL Configuration

- **Configuration File:**

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-db
spec:
  selector:
    matchLabels:
      app: mysql-db
  serviceName: mysql-headless
  replicas: 1
  template:
    metadata:
      labels:
        app: mysql-db
    spec:
      containers:
        - name: mysql-db
          image: mysql:8.0
          ports:
            - containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD
              value: "root_password"
            - name: MYSQL_DATABASE
              value: "bytesynergy"
            - name: MYSQL_USER
              value: "ramsai"
```

```
            - name: MYSQL_PASSWORD
              value: "ramsai@1"
          volumeMounts:
            - name: mysql-persistent-storage
              mountPath: /var/lib/mysql
  volumeClaimTemplates:
    - metadata:
        name: mysql-persistent-storage
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 1Gi
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-service
spec:
  selector:
    app: mysql-db
  ports:
    - protocol: TCP
      port: 3306
  type: ClusterIP
---
apiVersion: v1
kind: Service
metadata:
  name: mysql-headless
spec:
  clusterIP: None
  selector:
    app: mysql-db
  ports:
    - protocol: TCP
      port: 3306
```

- **Explanation:** The MySQL configuration is more complex as it involves setting up a stateful database, which requires persistent storage. MySQL is used as the database for the backend Spring Boot application. The database service allows the backend to store and retrieve user data.
  - **StatefulSet:** This is used for deploying the MySQL database, which requires persistent storage. We specify the MySQL image ('mysql:8.0') and configure environment variables like the root password, database name, and user credentials.
  - **Service:** The MySQL service is exposed internally within the Kubernetes cluster using a 'ClusterIP' service, which is only accessible to other components within the same cluster.
  - **Headless Service:** The headless service allows the StatefulSet to have direct communication between the pods, which is essential for maintaining the state of the database.
- **Command to apply:**

```
kubectl apply -f mysql.yaml
```

6.3.5 Horizontal Pod Autoscaler (HPA) Configuration

Horizontal Pod Autoscaler (HPA) in Kubernetes automatically adjusts the number of pods in a deployment based on resource utilization metrics such as CPU and memory usage. This ensures optimal resource management by scaling pods up or down according to workload demands.

The HPA configuration involves defining key parameters such as the minimum and maximum number of replicas, the target deployment to be scaled, and the metric thresholds that trigger scaling. In our project, we configured HPAs for both the backend and frontend services. The following are the Kubernetes HPA configuration files used in the project to enable automatic scaling based on resource utilization:

- **Configuration files:**
  - **backend_hpa.yaml**

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: spring-backend-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: spring-backend
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 70
```

  - **frontend_hpa.yaml**

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: flutter-web-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: flutter-web
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 50
```

- **Explanation**: For the backend, we monitor memory utilization, with a target average utilization of 70%. When memory consumption exceeds this threshold, additional pods are created until the usage normalizes or the maximum replica count of 10 is reached. Similarly, for the frontend, scaling is based on CPU utilization, targeting an average utilization of 50%. The `scaleTargetRef` field specifies the target deployment to be scaled by referencing its name, while selectors ensure the HPA monitors the correct pods based on matching labels.

This helps maintain responsive performance during periods of high user traffic while optimizing resource use during lighter workloads.

6.4 Continuous Deployment (CD) Pipeline

The Continuous Deployment (CD) process automates the deployment of the application stack using Ansible. This setup ensures that Kubernetes resources are deployed consistently across environments with minimal manual intervention. The key configuration files are detailed below:

- **Inventory File:** `inventory.ini`
  The inventory file specifies the target servers where Ansible will execute tasks. In this setup, the localhost `127.0.0.1` is used, with authentication managed securely via an Ansible vault file.

```
[servers]
127.0.0.1 ansible_connection=local ansible_user=srinidell ansible_password=@my_vault.
    ↪ yml
```

- **Ansible Playbook:** `deploy_stack.yaml`
  The playbook defines tasks to be performed on the specified hosts. It includes two roles: `common` for essential setup and `main_role` for application deployment.

```
---
- name: Deploy Full Stack
  hosts: servers
  roles:
    - common
    - main_role
```

- **Common Role Tasks:** `common/tasks/main.yaml`
  These tasks ensure Docker is installed, started, and enabled on the target system.

```
---
# Tasks file for common

- name: Ensure Docker is installed
  apt:
    name: docker.io
    state: present
  become: false

- name: Ensure Docker service is running and enabled
  service:
    name: docker
    state: started
    enabled: true
  become: false
```

- **Main Role Tasks:** `main_role/tasks/main.yaml`
  The main role handles pulling Docker images and applying Kubernetes configurations for services, deployments, and autoscalers.

```
---
# Tasks file for main_role

- name: Pull Docker images manually
  command: docker pull {{ item }}
  loop:
    - srinivasan2404/spring-backend
    - srinivasan2404/flutter-web
    - mysql:8.0

- name: Apply MySQL StatefulSet and Services
  command: kubectl apply -f mysql.yaml
  args:
    chdir: deployment

- name: Apply Backend Deployment and Service
  command: kubectl apply -f backend.yaml
  args:
    chdir: deployment

- name: Apply Frontend Deployment and Service
  command: kubectl apply -f frontend.yaml
  args:
    chdir: deployment

- name: Apply HPA for Frontend
  command: kubectl apply -f frontend_HPA.yaml
```

```
  args:
    chdir: deployment

- name: Apply HPA for Backend
  command: kubectl apply -f backend_HPA.yaml
  args:
    chdir: deployment
```

This Ansible-based deployment pipeline simplifies the process by automating the installation of Docker, pulling necessary container images, and deploying the entire stack using Kubernetes configurations. The configuration ensures the backend, frontend, database, and autoscaling are consistently deployed.

6.5 Continuous Integration (CI) Pipeline with Ngrok Integration

The CI pipeline automates the entire build, push, and deployment process using Jenkins. The pipeline consists of several stages defined in the `Jenkinsfile`. The process begins with building Docker images for the backend and frontend services using the appropriate Dockerfiles in their respective directories. These images are then pushed to Docker Hub using securely stored credentials managed by Jenkins. Afterward, local Docker images are cleaned up to free space on the build server. Finally, the application is deployed using Ansible and Kubernetes. Ansible Vault securely stores sensitive credentials, and the deployment process runs Ansible playbooks to set up Kubernetes services, deployments, and autoscalers in a fully automated manner.

6.5.1 Ngrok Integration

To enable the Jenkins server to trigger builds based on code commits in a private network setup, Ngrok is used to expose Jenkins to the public internet securely. Ngrok creates a tunnel that maps a publicly accessible URL to the local Jenkins server, allowing GitHub webhooks to notify Jenkins of repository changes.

Setting Up Ngrok for Jenkins:

1. **Install Ngrok:** Download and install Ngrok from `https://ngrok.com/`.
2. **Authenticate Ngrok:** Use your authentication token from Ngrok to secure your tunnel:

   ```
   ngrok authtoken <YOUR_AUTH_TOKEN>
   ```

3. **Run Ngrok:** Start Ngrok to expose the local Jenkins server. If Jenkins is running on port 8080, use the following command:

   ```
   ngrok http 8080
   ```

   This will generate a public URL, such as `https://abcd1234.ngrok.io`, which forwards traffic to the Jenkins server.

4. **Configure GitHub Webhook:**
   - Go to your GitHub repository settings and navigate to the `Webhooks` section.
   - Add a new webhook and set the `Payload URL` to the Ngrok URL (e.g., `https://abcd1234.ngrok.io/github-webhook/`).
   - Select `application/json` as the content type and choose events to trigger the webhook (e.g., push events).
   - Save the webhook configuration.
5. **Trigger Builds:** Any code push or pull request in the GitHub repository will now send a request to the Jenkins server through the Ngrok URL, triggering the pipeline defined in the `Jenkinsfile`.

The following `Jenkinsfile` defines the CI pipeline:

```
pipeline {

    environment {
        DOCKERHUB_CRED = credentials("Dockerhub-Credentials-ID") // Jenkins
            ↪ credentials ID for Docker Hub
        DOCKER_HUB_REPO = 'srinivasan2404' // Docker Hub username or repo name
        MINIKUBE_HOME = '/home/jenkins/.minikube'
        VAULT_PASS = credentials("ansible_vault_pass")
    }

    agent any
    stages {

        stage('Build and Tag Images') {
            steps {

                dir('spring-backend') {
                    sh "docker build -t ${DOCKER_HUB_REPO}/spring-backend:latest ."
                }

                dir('flutter-frontend/healthlink') {
                    sh "docker build -t ${DOCKER_HUB_REPO}/flutter-web:latest ."
                }
            }
        }

        stage('Push to Docker Hub') {
            steps {
                sh 'echo $DOCKERHUB_CRED_PSW | docker login -u $DOCKERHUB_CRED_USR --
                    ↪ password-stdin'
                sh "docker push ${DOCKER_HUB_REPO}/spring-backend:latest"
                sh "docker push ${DOCKER_HUB_REPO}/flutter-web:latest"
            }
        }

        stage('Clean Local Docker Images') {
            steps {
                sh "docker rmi ${DOCKER_HUB_REPO}/spring-bakend:latest || true"
                sh "docker rmi ${DOCKER_HUB_REPO}/flutter-web:latest || true"
            }
        }

        stage("Deploy Ansible Vault with Kubernetes"){
            steps {
               sh '''
                echo "$VAULT_PASS" > /tmp/vault_pass.txt
                chmod 600 /tmp/vault_pass.txt
                ansible-playbook -i inventory --vault-password-file /tmp/vault_pass.
                    ↪ txt deploy_stack.yaml
                rm -f /tmp/vault_pass.txt
                '''
            }

        }
    }
}
```

7 Grafana, Prometheus, and Loki Stack: Overview and Use Cases

The Grafana, Prometheus, and Loki stack is a powerful suite of open-source tools used for monitoring, logging, and visualization. It is primarily used for collecting, storing, analyzing, and visualizing time-series data, logs, and metrics, especially in cloud-native environments. This stack is designed to enable real-time insights into infrastructure and application performance.

7.1 Components of the Grafana, Prometheus, and Loki Stack

- **Prometheus:** Prometheus is an open-source system monitoring and alerting toolkit designed for collecting and storing time-series data. It pulls data from a variety of sources and stores it in a highly efficient, queryable time-series database. Prometheus is particularly well-suited for monitoring systems, services, and applications in dynamic environments like containers and Kubernetes.
- **Loki:** Loki is a log aggregation system designed to store and query log data. It is optimized for integration with Prometheus and provides an easy way to collect logs from a wide range of sources, including applications, services, and infrastructure. Loki enables searching and exploring log data in conjunction with metrics data from Prometheus for better observability.
- **Grafana:** Grafana is an open-source visualization and monitoring tool that integrates with both Prometheus and Loki. It allows users to create interactive dashboards that combine metrics and logs to visualize system performance, troubleshoot issues, and monitor applications and infrastructure in real time.

7.2 Why Use the Grafana, Prometheus, and Loki Stack?

The Grafana, Prometheus, and Loki stack is used extensively for monitoring, logging, and observability due to its scalability, flexibility, and deep integration across the three components. Some key reasons for its widespread use include:

- **Centralized Monitoring and Logging:** This stack centralizes both metrics and log data from various sources, making it easier to monitor and troubleshoot applications, services, and infrastructure.
- **Real-Time Data Analysis:** Prometheus offers real-time monitoring and alerting capabilities, while Loki enables real-time log collection. Grafana's visualization layer combines both metrics and logs to provide a comprehensive view of the system's health.
- **Scalability:** Prometheus is designed to scale horizontally, meaning it can handle large volumes of time-series data, especially in dynamic environments like Kubernetes. Loki is designed to scale with log data, while Grafana can visualize and monitor large datasets across various sources.
- **Flexibility and Customization:** Prometheus supports custom metrics collection from virtually any system or service. Loki can be used to collect logs from a variety of formats, and Grafana allows users to create highly customizable dashboards for various use cases.
- **Powerful Query and Alerting Capabilities:** Prometheus's query language (PromQL) and Loki's LogQL enable users to query time-series data and logs for detailed insights. The stack's alerting functionality can trigger notifications based on specific conditions, helping teams respond to issues proactively.
- **Unified Visualization:** Grafana's dashboards allow users to integrate both metrics from Prometheus and logs from Loki, providing a holistic view of system performance, health, and troubleshooting information in a single interface.
- **Comprehensive Observability:** This stack provides complete observability, covering metrics, logs, and traces, enabling users to detect issues, understand system performance, and gain deep insights into infrastructure and application behavior.

7.3 Common Use Cases for the Grafana, Prometheus, and Loki Stack

The Grafana, Prometheus, and Loki stack is commonly used in various monitoring, logging, and observability scenarios:

- **System and Application Monitoring:** Prometheus is widely used to monitor infrastructure and application performance, providing visibility into system metrics such as CPU usage, memory consumption, and response times.

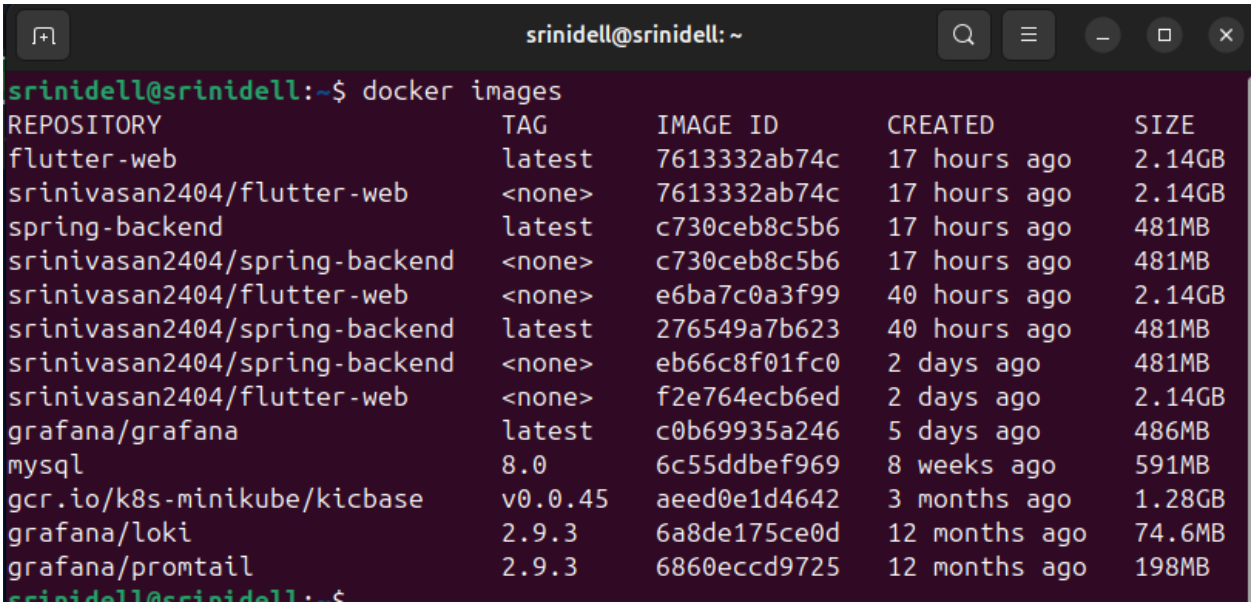

Figure 2: Image containing all the docker images present.

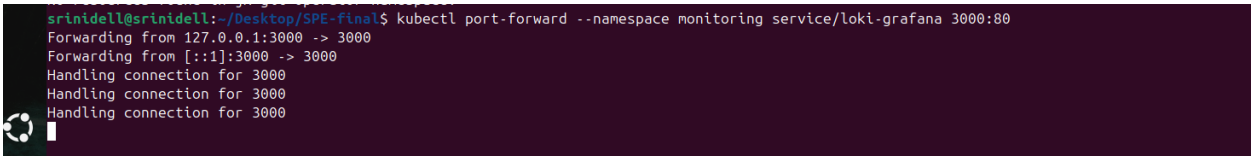

Figure 3: Image for the Loki- Grafana port forwarding.

- **Log Aggregation and Analysis:** Loki is used for log aggregation from different services and applications, allowing teams to explore logs, correlate them with metrics, and perform troubleshooting.
- **Infrastructure Monitoring:** This stack is widely used for monitoring servers, databases, network devices, and cloud infrastructure, allowing teams to track the health and performance of critical systems and services.
- **Alerting and Proactive Monitoring:** Prometheus's alerting capabilities, combined with Grafana's visualization, allow teams to set up custom alerting rules based on specific metrics or log conditions, enabling quick responses to incidents.
- **Cloud-Native and Kubernetes Monitoring:** This stack is particularly useful in Kubernetes environments, where Prometheus monitors containerized applications, and Loki aggregates logs from these containers to provide full-stack observability.
- **Business and Operations Analytics:** Grafana and Prometheus can be used for business intelligence, providing metrics on system performance, usage, and behavior that can inform decision-making, especially in dynamic environments.

Figure 1 shows the creative dashboard made from Grafana. It contains visualization for the logs of spring-backend, flutter frontend, and mysql pods in the application.

The Grafana visualization displayed in the figure 6 showcases monitoring of pods (spring), specifically for a Kubernetes environment. It includes:

- **Metrics tracked:** CPU usage, filesystem writes, and block I/O.
- **Instance:** Monitored on a local Kubernetes cluster (e.g., `minikube`).

Figure 4: SVC Local Monitoring

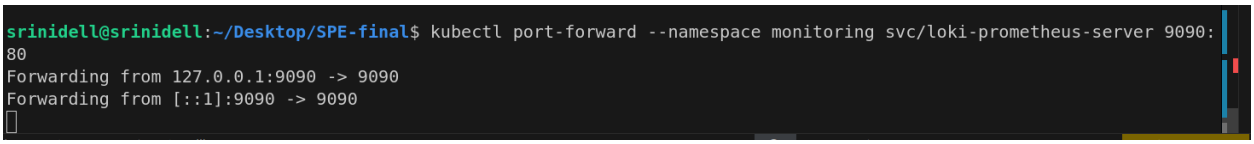

Figure 5: Port forwarding for loki-promethus

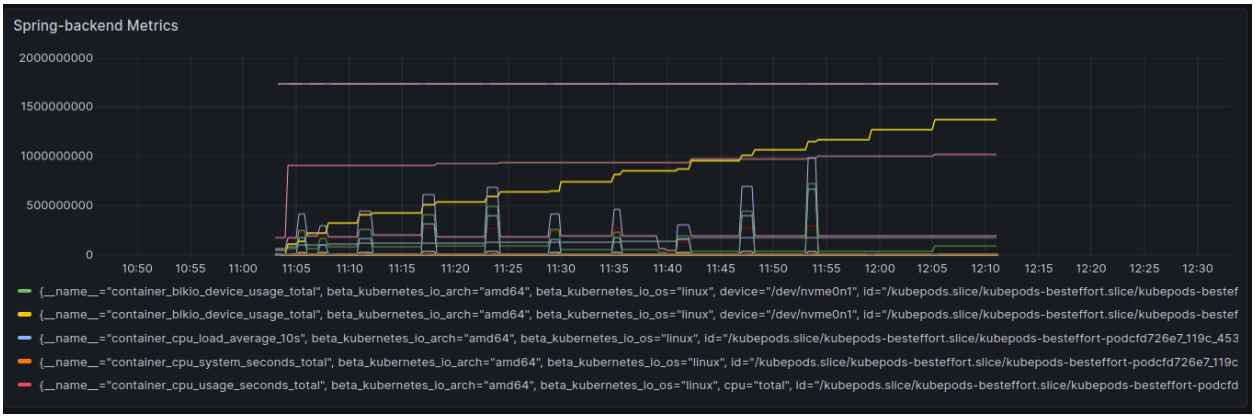

Figure 6: Visualization of metrics for spring pods

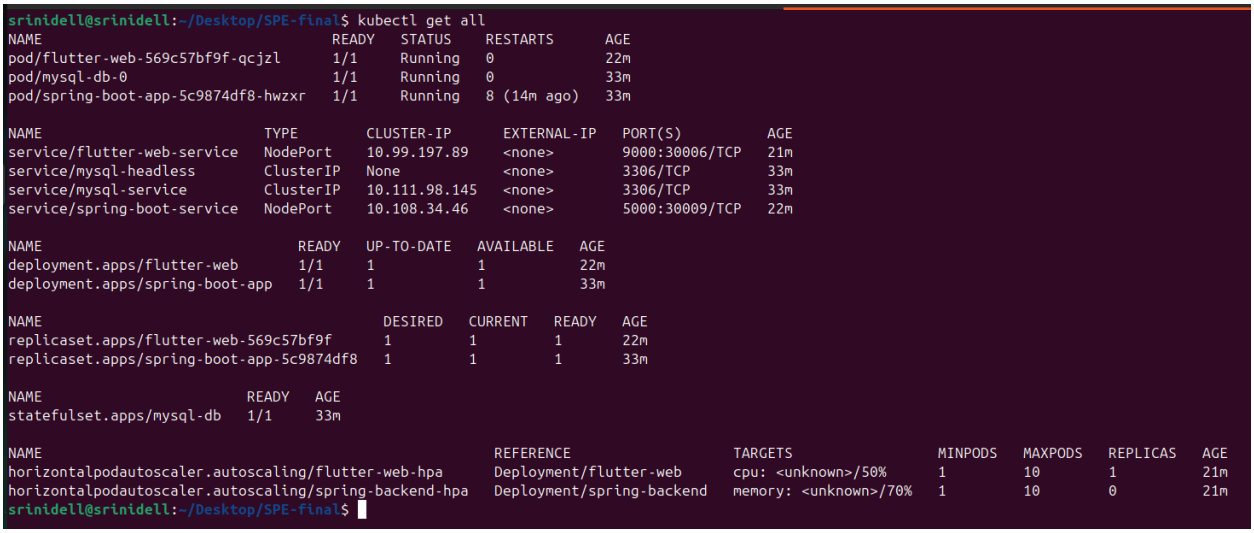

Figure 7: Snapshot of Kubectl get all output from the terminal.

- **Purpose:** To analyze the application's performance and detect resource bottlenecks.

Figure 2 shows the snapshot of all the docker images used in the project. Figure 7 shows the result of all pods, services, deployment, replicaset, statefulsets and HPAs used in the system.

8 Setup Steps from Scratch

In this section, we provide the high-level steps required to set up the environment for this project. Follow these steps to get the application up and running:

1. **Clone the GitHub Repository**
   - If you already have a local project, integrate it with Git. First, initialize your local directory as a Git repository (if not done already):

     ```
     git init
     ```

   - Add the remote repository:

     ```
     git remote add origin https://github.com/your-repo-url.git
     ```

   - Push your changes to GitHub:

     ```
     git push -u origin main
     ```

2. **Install Prerequisites**
   - Ensure you have the following installed on your system:
     - Docker
     - Kubernetes (Minikube or any other Kubernetes setup)
     - Ansible
     - Jenkins (for CI/CD)
   - Follow the installation guides for each tool from their official documentation.
3. **Build Docker Images**

- Navigate to the respective directories (`spring-backend`, `flutter-frontend`) and build the Docker images using:

```
docker build -t your-repo-name/spring-backend .
docker build -t your-repo-name/flutter-web .
```

4. **Deploy Kubernetes Resources**

- Apply the Kubernetes deployment files to set up the services and deployments:

```
kubectl apply -f deployment/mysql.yaml
kubectl apply -f deployment/backend.yaml
kubectl apply -f deployment/frontend.yaml
```

- After the resources are deployed, you can view the status of all pods and services using:

```
kubectl get pods
kubectl get services
```

- You can access the frontend and backend services via their NodePorts by opening a browser and navigating to `http://<your-node-ip>:<node-port>` (for example, `http://127.0.0.1:30006` for the frontend).

5. **Set Up Horizontal Pod Autoscaling (HPA)**

- Deploy the Horizontal Pod Autoscaler (HPA) for both backend and frontend:

```
kubectl apply -f deployment/backend_HPA.yaml
kubectl apply -f deployment/frontend_HPA.yaml
```

6. **Set Up Ansible for Deployment**

- First, create the required Ansible roles and organize your playbook files:
  - Create a role structure:

```
ansible-galaxy init main_role
```

```
ansible-galaxy init common
```

  - Create the `deploy_stack.yaml` playbook with the necessary tasks for deploying your services.
- Create an Ansible Vault file for securely storing sensitive information (like passwords and secrets):

```
ansible-vault create ansible_vault.yml
```

- When prompted, enter a password for the vault, which you'll later use for running the playbook.
- Run the Ansible playbook to deploy the full stack:

```
ansible-playbook -i inventory.ini --vault-password-file ansible_vault.yml deploy_stack
    ↪ .yaml
```

7. **Set Up Continuous Integration (CI)**

- Make sure your Jenkins instance is set up and running.
- Integrate your GitHub repository with Jenkins to enable automatic build triggers upon code changes.
- Configure the Jenkins pipeline (`Jenkinsfile`) to automate the build and deployment processes.

8. **Set Up CI/CD Pipeline with Jenkins**

- In Jenkins, configure the pipeline using the `Jenkinsfile` to handle the CI/CD process.

- Set up the necessary environment variables, credentials, and Docker Hub credentials in Jenkins to allow it to pull and push Docker images.
- Ensure Jenkins has access to your Kubernetes cluster to apply the necessary configurations.

9. **Setting Up ELK Stack with Docker and Helm**

- **Step 1: Pull Required Docker Images**
  To get started, pull the necessary Docker images for Loki, Promtail, and Grafana. Run the following commands:

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

- **Step 2: Install Helm**
  Install Helm using either `snap` or by running a script. First, run the following command to install `helm` via `snap`:

```
sudo snap install helm
```

  Alternatively, you can use the following script to install Helm:

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

- **Step 3: Add the Grafana Helm Repository**
  Add the Grafana Helm repository to access the necessary charts for installing the ELK stack components:

```
helm repo add grafana https://grafana.github.io/helm-charts
```

- **Step 4: Install the Loki Stack**
  Now, install the Loki stack, which includes Loki, Promtail, and Grafana. This will create the necessary services and components in the `monitoring` namespace. Run the following command to install the stack:

```
helm install loki grafana/loki-stack \
  --namespace monitoring \
  --create-namespace \
  --set loki.image.tag=2.9.3 \
  --set promtail.enabled=true \
  --set promtail.config.server.http_listen_port=3101 \
  --set promtail.config.clients[0].url=http://loki:3100/loki/api/v1/push \
  --set promtail.config.positions.filename=/run/promtail/positions.yaml \
  --set promtail.config.scrape_configs[0].job_name="kubernetes-pods" \
  --set promtail.config.scrape_configs[0].kubernetes_sd_configs[0].role="pod"
      ↪ \
  --set promtail.config.scrape_configs[0].relabel_configs[0].action="keep" \
  --set promtail.config.scrape_configs[0].relabel_configs[0].source_labels="[
      ↪ _meta_kubernetes_namespace]" \
  --set promtail.config.scrape_configs[0].relabel_configs[0].regex=".*" \
  --set promtail.config.scrape_configs[0].relabel_configs[1].source_labels="[
      ↪ meta_kubernetes_pod_name]" \
  --set promtail.config.scrape_configs[0].relabel_configs[1].target_label="job
      ↪ " \
  --set promtail.config.scrape_configs[0].relabel_configs[2].source_labels="[
      ↪ meta_kubernetes_namespace]" \
  --set promtail.config.scrape_configs[0].relabel_configs[2].target_label="
      ↪ namespace" \
  --set promtail.config.scrape_configs[0].relabel_configs[3].source_labels="[
      ↪ _meta_kubernetes_pod_name]" \
  --set promtail.config.scrape_configs[0].relabel_configs[3].target_label="pod
      ↪ " \
  --set grafana.enabled=true \
  --set prometheus.enabled=true
```

- **Step 5: Verify the Installation**
  After installation, you can verify the status of the deployed services by using the following command:

```
kubectl get all -n monitoring
```

- **Step 6: Retrieve Grafana Admin Password**
  To log into Grafana, you will need to retrieve the default admin password. Use the following command to decode the password:

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

- **Step 7: Port Forwarding to Access the Services**
  Set up port forwarding to access the services locally from your browser. Run the following commands to port-forward the necessary services:
  - Access the Prometheus server (used by Loki for scraping metrics):

```
kubectl port-forward --namespace monitoring svc/loki-prometheus-server
    ↪ 9090:80
```

  - Access the Grafana dashboard:

```
kubectl port-forward --namespace monitoring service/loki-grafana 3000:80
```

  - Access the Loki HTTP API:

```
kubectl port-forward svc/loki -n monitoring 3100:3100
```

- **Step 8: Accessing the Local Websites**
  After port forwarding, you can access the following services locally on your browser:
  - Grafana: Open `http://localhost:3000` to access the Grafana dashboard. Use the admin username and the password retrieved earlier to log in.
  - Prometheus: Open `http://localhost:9090` to access the Prometheus web UI.
  - Loki: Open `http://localhost:3100` to access the Loki API.

For further details on each step and the specific file contents, please refer to the **DevOps Implementation Details** section and the following GitHub repository: `GitHub Repository`