Kubernetes cluster for healt care services

Federico Berton
Computer and Robotics Engineering
Virtual Network & Cloud Computing
University of Perugia
Perugia, Italy
federicoberton.fb@gmail.com

In the digital era, the healthcare sector is experiencing a profound transformation fueled by technological advancements. This project harnesses the capabilities of Kubernetes to enhance both the accessibility and quality of healthcare services. By integrating state-of-the-art medical devices to capture clinical data, the system facilitates the precise and real-time monitoring of vital signs and critical health metrics, thereby improving patient care and enabling timely interventions.

I. Introduction

As the capabilities of digital medical devices advance, it becomes essential to have tools for real-time collection and analysis of data generated by these devices. This project aims to develop an advanced Kubernetes-based infrastructure designed to ensure efficient and scalable management of clinical data. Kubernetes, renowned for its ability to orchestrate containers dynamically and resiliently, serves as the hub for receiving, processing and securely storing information, ensuring its integrity and constant availability. Processed data is made accessible through user-friendly interfaces, enabling health care providers to continuously monitor patients' health status in detail. This approach not only enhances the speed and accuracy of clinical decisions, but also enables predictive analytics, identifying potential risks early and optimizing treatment strategies.

In this project, two primary services are implemented: one concerning the recording of heart rate and another related to cerebral electrical activity.

II. Infrastructure Design and Automation

First of all, it is critical to carefully examine the design architecture, which is designed to ensure, once the Kubernetes cluster is up and running, an automation of the testing and deployment processes in response to each source code change. This automation mechanism relies on the use of GitHub Actions, a framework that enables continuous integration and continuous delivery (CI/CD) in a smooth and efficient manner. For each service, GitHub Actions orchestrate the creation of Docker images, run a full suite of automated tests, and then manage the deployment of the final version to the Kubernetes cluster.

Let us now delve into the detailed process for achieving this setup: first, it is essential to determine the physical or virtual machines that will be used. Following this, we need to define the configuration for Kubernetes, including the number of nodes and other specifications, and proceed with its installation. Once Kubernetes is set up, we can manage integration with GitHub Actions and Docker Hub to automate workflows and deploy the desired services efficiently.

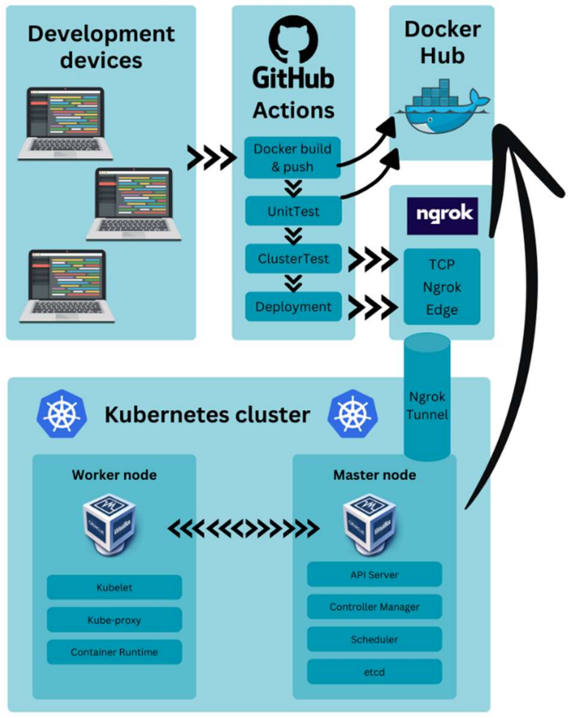


Fig. 1. Infrastructure design

A. Server simulation and Kubernetes cluster creation

A Kubernetes cluster can be tested using virtual machines on a standard laptop. Specifically, two Linux VMs can be created: one as the Master node and the other as the Worker node. The Master node oversees the entire cluster, coordinating the various services. Its key functions include:

- **API Server**: it handles incoming requests (via REST API) **and** serves as the primary interface for interacting with the cluster.
- **etcd**: it stores cluster configuration data in a distributed and **consistent** manner.
- **Controller manager**: it runs controllers that monitor the cluster's state and ensure it matches the desired state.
- **Scheduler**: it determines which node should run a new *Pod*, considering factors like available resources and more.

The Worker node is responsible for actually running the applications. Its main functions include:

- **Kubelet**: the node agent responsible for managing pods. It ensures containers are running as specified and communicates with the master’s API server.

- **Kube-proxy**: it manages network rules on the nodes, enabling communication between different services within the cluster.
- **Container runtime**: the component that actually runs the containers (e.g., Docker).

The virtual machines can be configured in different network modes based on the requirements:

- If communication is needed only between the Master and Worker nodes and not with external devices, configuring the VMs with a NAT network is sufficient. This setup isolates the cluster from external networks while allowing internal communication between the nodes.
- To enable communication between the Kubernetes cluster and external devices (such as the host laptop containing the VMs or other devices on the same Wi-Fi network), the VMs should be configured with a *Bridge adapter* network setting. This configuration allows the cluster to interact with both the host system and other devices on the network.

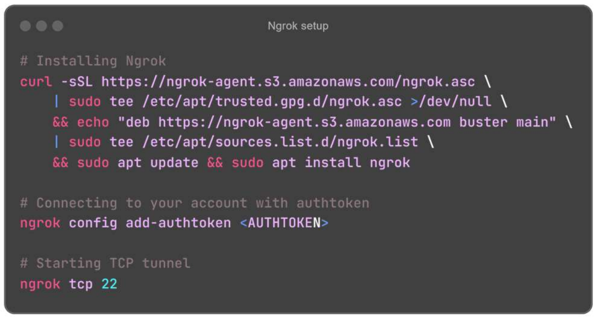
Once the VMs are created, they must be configured for IP address and hostname [1]:

- Modify the netplan configuration file to set the static IP address. The file is usually located in /etc/netplan/, and it might be named something *like 01-netcfg.yaml* or *01-network-manager-all.yaml*.
- To ensure proper hostname resolution within the cluster, edit */etc/hosts* and */etc/hostname* in both the Master and Worker.

At this stage, the installation of Kubernetes can proceed. A straightforward method for this process is to use Kubespray [1], which simplifies the deployment by automating the setup and configuration of Kubernetes clusters across multiple nodes.

B. Ngrok tunnel [4]

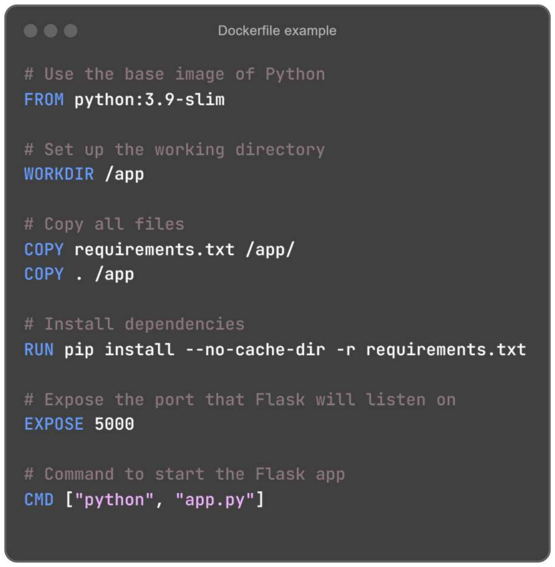
Ngrok was employed to expose the Kubernetes master node within a virtual machine to facilitate integration with GitHub Actions. Despite the use of a bridge adapter in the VM configuration, the connection to the external network is confined to the subnet of the Wi-Fi in use. Ngrok enabled the creation of a secure tunnel by providing a public URL for the Kubernetes master node. After creating your own token in the Ngrok account, you can create a tunnel connection with the Master node via a TCP link:



```
# Installing Ngrok
curl -sSL https://ngrok-agent.s3.amazonaws.com/ngrok.asc \
    | sudo tee /etc/apt/trusted.gpg.d/ngrok.asc >/dev/null \
    && echo "deb https://ngrok-agent.s3.amazonaws.com buster main" \
    | sudo tee /etc/apt/sources.list.d/ngrok.list \
    && sudo apt update && sudo apt install ngrok

# Connecting to your account with authtoken
ngrok config add-authtoken <AUTHTOKEN>

# Starting TCP tunnel
ngrok tcp 22
```

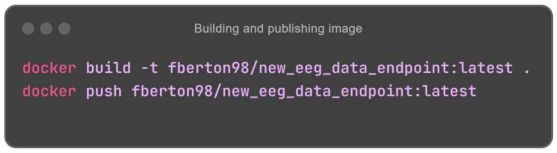
C. Docker and DockerHub

Docker was used to build, manage, and deploy services in isolated containers, developed in Python using the Flask framework. The following is the information needed to create the image:



```
# Use the base image of Python
FROM python:3.9-slim

# Set up the working directory
WORKDIR /app

# Copy all files
COPY requirements.txt /app/
COPY . /app

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Expose the port that Flask will listen on
EXPOSE 5000

# Command to start the Flask app
CMD ["python", "app.py"]
```

The file *requirements.txt* contains the dependencies that such a service needs. Finally, to deploy the developed services, it is necessary to create local Docker images and upload them to DockerHub:



```
docker build -t fberton98/new_eeg_data_endpoint:latest .
docker push fberton98/new_eeg_data_endpoint:latest
```

This approach ensures that services are isolated and easily reproducible, while simplifying the development and deployment cycle. It thus fosters a reliable and scalable release pipeline.

D. GitHub and GitHub Actions

GitHub Actions is used to automate the entire lifecycle of the development, testing, and deployment processes for the services. Its integration allowed predefined workflows, written in YAML files, to be automatically triggered with every push to the repository. These workflows included building Docker images, running unit tests, creating temporary test environments, and finally updating production deployments if all tests were successful. The use of GitHub Actions ensured a continuous, fast, and reliable process, minimizing manual intervention and enhancing the efficiency of the release pipeline.

III. PROJECT DEPLOYMENT

The project is structured to ensure efficient and secure management of clinical data through a services-based architecture. The core component of the system is a persistent database designed to securely store all incoming data. This database serves as the central repository, allowing for continuous storage and access to clinical data.

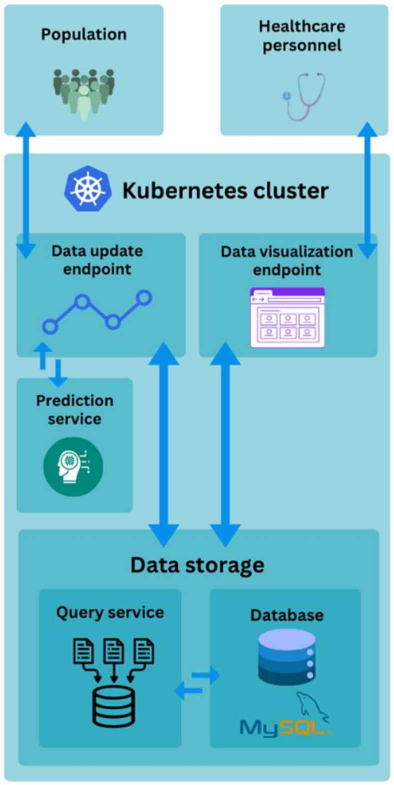


Fig. 2. Project structure

A. Services

The system is divided into several services, each with a specific function:

- **Electroencephalographic (EEG) data acquisition:** this service is responsible for listening to and receiving electroencephalographic data from connected subjects.
- **Heart rate monitoring:** this service handles the acquisition of heart rate data from connected subjects, receiving real-time data through compatible wearable devices.
- **Predictive EEG processing:** utilizing a neural network-based predictive algorithm, this service processes EEG data to determine the presence of epileptic episodes. The algorithm has been trained on a comprehensive dataset to ensure high accuracy and reliability.
- **Database query:** this service facilitates querying the persistent database, allowing for the insertion of new data and retrieval of previously recorded data.
- **Real-time data visualization:** designed to expose recorded data in real-time, this service provides a web page accessible to healthcare personnel. The web page is continuously updated to reflect the most recent data.

B. Test mobile apps

To support system development and validation, a test application has been created to simulate the transmission of electroencephalographic data. This application uses previously saved EEG data to emulate real-world scenarios of data acquisition and transmission.

Additionally, an existing iPhone application has been integrated to extract heart rate data recorded by the Apple Watch and transmit it in real-time to a dedicated REST API. This integration allows the use of widely available and reliable devices for heart rate data collection.

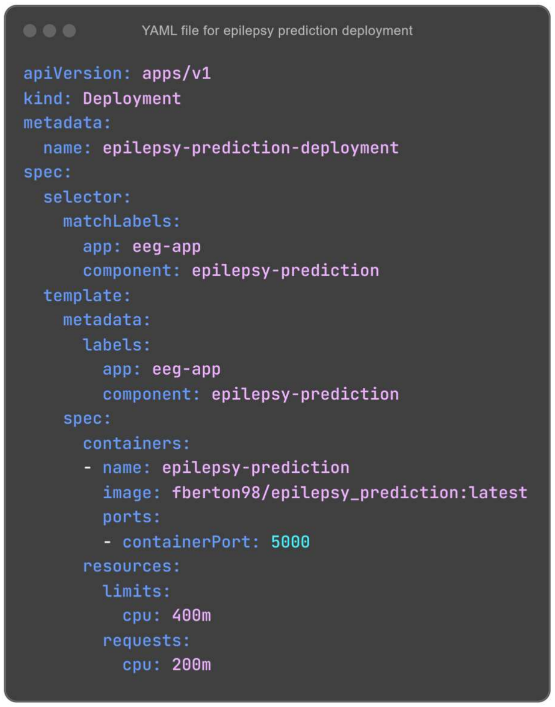
IV. KUBERNETES SERVICES DEVELOPEMENT

This section delves into the orchestration process of the application layer within the Kubernetes cluster, starting with the procedure for containerizing each service using Docker. Following that, the execution of unit tests to validate the behavior of the individual service will be outlined. If successful, the implementation of tests that verify the proper functioning of basic functionality of a service in the cluster itself will be addressed. Finally, the definitive deployment will be carried out, ensuring the service's integration into the production system.

A. Kubernetes deployments and services

After building a Docker service image, it is possible to deploy it in the Kubernetes cluster. To accomplish this efficiently, create a ***deployment_service.yaml*** file that includes both the deployment and relative service configurations.

A *Deployment* manages the creation and updating of *pods* (operating units), whereas a *Service*, exposes a group of *pods*, allowing them to be accessed permanently by other services or users, regardless of the individual pods that may be created or destroyed. For the deployment, the following key parameters are specified:



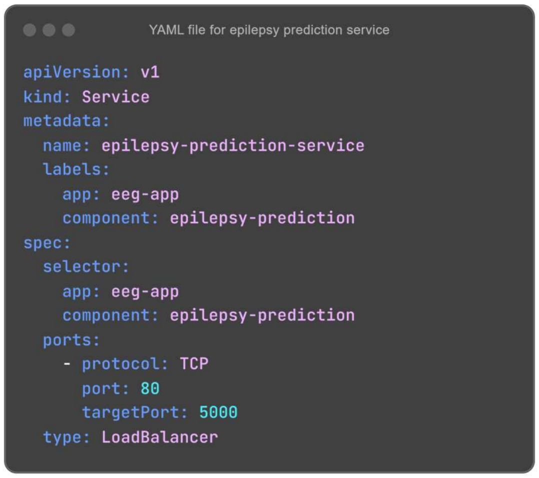
YAML file for epilepsy prediction deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: epilepsy-prediction-deployment
spec:
  selector:
    matchLabels:
      app: eeg-app
      component: epilepsy-prediction
  template:
    metadata:
      labels:
        app: eeg-app
        component: epilepsy-prediction
    spec:
      containers:
      - name: epilepsy-prediction
        image: fberton98/epilepsy_prediction:latest
        ports:
        - containerPort: 5000
      resources:
        limits:
          cpu: 400m
        requests:
          cpu: 200m
```

Deployment settings in YAML file	
Keys	***Description***

name	It uniquely identifies the *deployment* within the Kubernetes cluster
labels: app	It indicates the name of the application associated with the *deployment*
labels: component	It specifies the component of the application
image	It specifies the Docker image to be used for instantiating the deployment and its associated pods
containerPort	It defines the port within the container that the application listens on. This port should match the one configured in the Docker image
resources: requests	Minimum amount of resources the *pod* requires to start up
resources: limits	Maximum amount of resources the *pod* can use

Regarding *Service*, the following parameters are established:

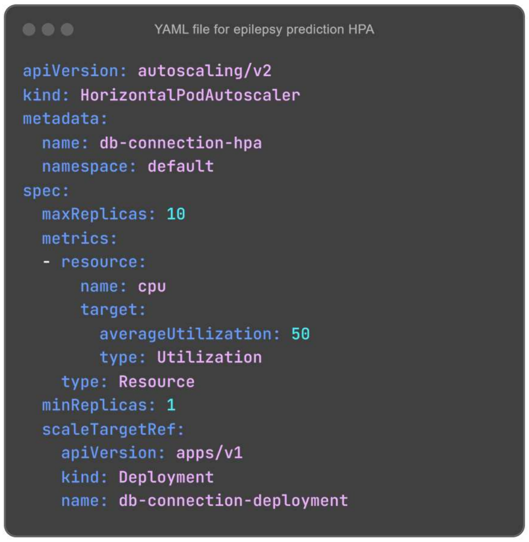


```yaml
apiVersion: v1
kind: Service
metadata:
  name: epilepsy-prediction-service
  labels:
    app: eeg-app
    component: epilepsy-prediction
spec:
  selector:
    app: eeg-app
    component: epilepsy-prediction
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
  type: LoadBalancer
```

Service settings in YAML file	
Keys	***Description***
name	It assigns a unique identity to the service
labels: app	It identifies the application that the service aims to expose
labels: component	It specifies the component of the application to which the service is directed
protocol	It determines the communication protocol used by the service
type	Defines how to balance traffic between *pods* and, in addition, also exposure outside the cluster via IP or DNS

For services accessible only within the cluster, it is sufficient to configure the *ClusterIP* type. However, if it is necessary to make services accessible from outside the cluster, the *LoadBalancer* type must be used. In addition, if you wish to manage the exposure of external services via *Ingress*, services intended to receive external traffic can also be configured as *ClusterIP*, since the *Ingress Controller* itself is responsible for the exposure of endpoints outside the cluster.

In relation to deployments, an *HorizontalPodAutoscaler* (HPA) can be configured to dynamically adjust the number of *pod* replicas based on resource utilization. The HPA in Kubernetes needs a metrics server to function properly. It collects and provides the HPA with data on pod resource utilization, such as CPU and memory. Without these metrics, the HPA would not be able to determine when to scale the number of pods based on the workload



```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: db-connection-hpa
  namespace: default
spec:
  maxReplicas: 10
  metrics:
  - resource:
      name: cpu
      target:
        averageUtilization: 50
        type: Utilization
    type: Resource
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: db-connection-deployment
```

HorizontalPodAutoscaler settings in YAML file	
Keys	***Description***
name	It specifies the unique name of the HorizontalPodAutoscaler resource
minReplicas	It defines the minimum number of pod replicas that the HPA will maintain
maxReplicas	It defines the maximum number of pod replicas that the HPA will maintain
averageUtilization	It indicates the target average CPU utilization across all pods, to determine when to scale the number of pods up or down.
scaleTargetRef: kind	It specifies the type of resource that the HPA is targeting for scaling (typically ***Deployment***)

scaleTargetRef: name	It refers to the name of the resource being scale

Together, these elements enable effective management and exposure of applications inside and outside the cluster, optimizing scalability and resilience.

B. MetalLB

To provide load balancing for Kubernetes services in environments such as virtual machines, which lack the native load balancer support offered by public cloud providers, MetalLB can be configured to handle this task [3]. MetalLB offers the following functionalities:

- **Exposing services with external IPs**: MetalLB assigns external IP addresses to Kubernetes services, making these services accessible outside the cluster. This allows users and external applications within the same subnet to connect to Kubernetes services through designated IP addresses.
- **Traffic management**: by utilizing ARP, MetalLB manages network traffic by assigning external IP addresses and directing it to the appropriate pods within the cluster. This emulates the behavior of a traditional load balancer by ensuring efficient traffic distribution.

To achieve this, we need to [1]:

- Install MetalLB in cluster.
- Configure the IP address pool, creating an *IPAddressPool* object that defines the range of IP addresses available to MetalLB.

C. Database

In the Kubernetes cluster, a persistent MySQL database has been configured to securely store and manage health-related data. This setup involves several key component [2]:

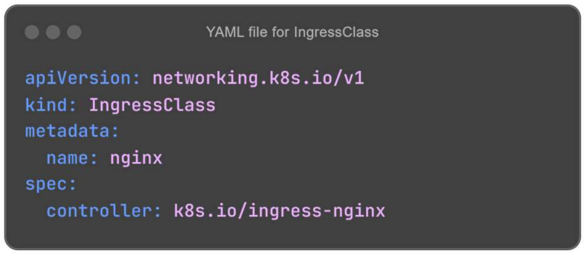
Cluster resources defined for database	
Component	***Description***
PersistentVolume	Storage resource in the Kubernetes cluster that provides persistent storage space
PersistentVolumeClaim	It specifies the amount of storage needed and the access mode. When the *PVC* is satisfied by a *PV*, the MySQL *pod* can mount this storage and use it to save database data
Secret	Contains the credentials needed to access the database
Deployment	A Kubernetes resource that automates the creation, updating, and management of MySQL pod
Service	Provides a stable network name and a fixed IP address to access the database

This comprehensive configuration ensures that the MySQL database is capable of maintaining continuous availability and integrity of health-related data within the Kubernetes environment.

D. Ingress

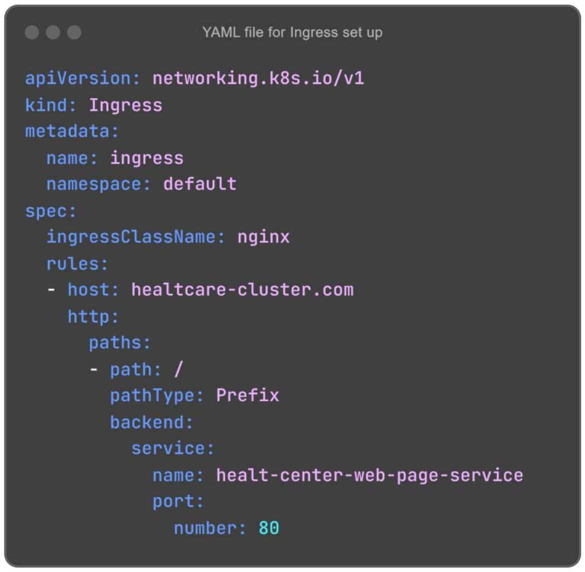
To manage external access to services within a Kubernetes cluster, *Ingress* efficiently routes HTTP/S traffic to the appropriate services according to predefined rules. It provides a centralized point for managing incoming traffic, avoiding the need to configure individual services to manage routing. To configure it, it is necessary to:

1. Apply the default configuration (from the official Kubernetes Ingress NGINX repository) for the *NGINX Ingress controller*, which will manage the routing rules for HTTP/S traffic.
2. Create a new *Ingress Class* that uses the *NGINX controller*.



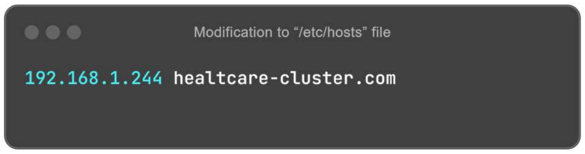
```yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: nginx
spec:
  controller: k8s.io/ingress-nginx
```

3. Create an Ingress resource that manages the routing of HTTP/S traffic based on defined rules. It directs traffic to specific services within the cluster depending on the requested hostname and URL path, thereby allowing external access to the services [1].



```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress
  namespace: default
spec:
  ingressClassName: nginx
  rules:
  - host: healtcare-cluster.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: healt-center-web-page-service
            port:
              number: 80
```

4. Edit the /etc/*hosts* file of your operating system to associate the domain with a specific IP address of *Ingress Controller*.


Modification to "/etc/hosts" file

```
192.168.1.244 healtcare-cluster.com
```

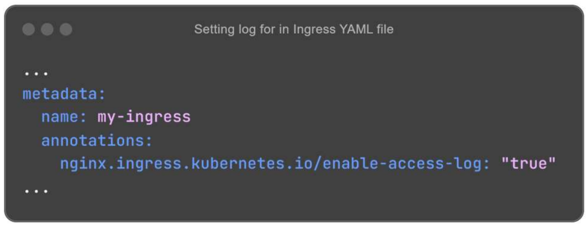
V. Markovian Analysis of Servants Number

The Markov queuing model can be considered an effective tool for determining the optimal number of *pods* in a Kubernetes cluster because of its ability to model complex systems operating under varying loads. The M/M/m/m model of queuing theory assumes that arrival processes follow a Poisson distribution, that service times are exponentially distributed, and that there is a fixed number of servers (m) with no possibility of accommodating excess requests beyond the system's capacity (m), i.e., no queuing. Since the Kubernetes cluster in question rejects or uses a very short time-out in case there is a request in case, it makes sense to approximate it as M/M/m/m.

Using that model, one can calculate the offered traffic, defined as the product of the request arrival rate and the average service time, and then apply Erlang's formula B to determine the ideal number of servers. Although the HPA automates *pod* management, it is important to set a maximum *pod* limit to avoid excessive requests and high costs, and this can be approximated through the Markovian approach.

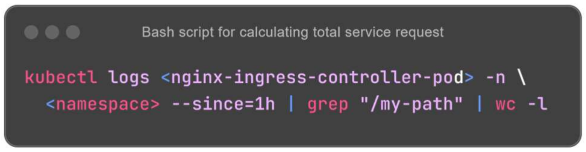
Because a feed-forward queueing system is used, in which requests move in a direct flow through the various services without feedback, Burke's theorem ensures that the outflow of arrivals from each servant follows a Poisson distribution, thus allowing each service to be treated as independent of the others. Consequently, for each service, it is essential to analyze the frequency of request arrival and service time separately, and then determine the blocking probability as a function of the number of available servants. Alternatively, a preliminary analysis can be conducted on request volumes and average response times to determine the optimal number of servants to ensure a desired blocking probability. This process can be accomplished as follows:

- establish, for such ti test phase, a fixed number of service *pods* to be analyzed.
- perform a preliminary analysis of the request arrival rate (λ), including failed or rejected attempts. This monitoring can be accomplished using tools such as Prometheus, which allows you to collect metrics in real time, or through an Ingress log analysis. To enable the latter mode, it is necessary to properly configure the annotation in the Ingress, enabling the recording of request details in the log file:


Setting log for in Ingress YAML file

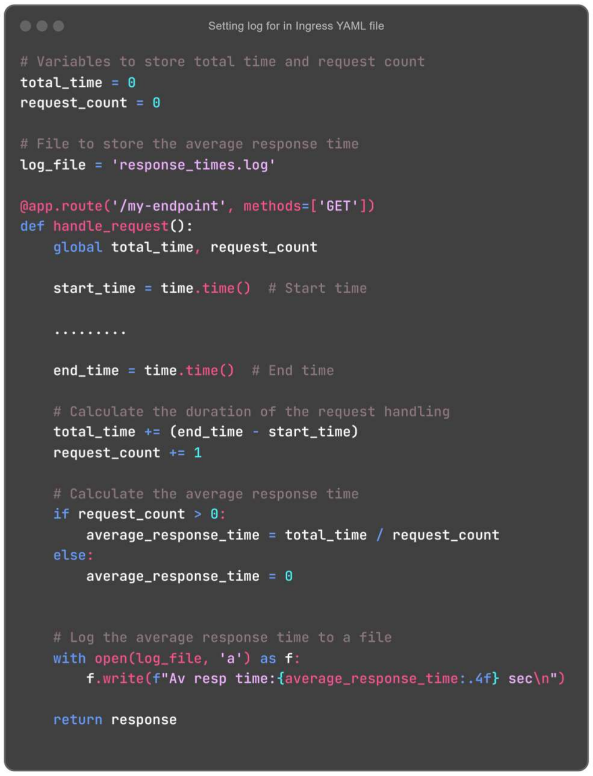
```
...
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/enable-access-log: "true"
...
```

From the analysis of the collected logs or metrics, it is possible to estimate the total number of requests that affected a particular path (in the example below, "*/my-path*"):


Bash script for calculating total service request

```
kubectl logs <nginx-ingress-controller-pod> -n \
  <namespace> --since=1h | grep "/my-path" | wc -l
```

- determine the service time (T_s) required by each servant to complete a single request. This value can be approximated by entering appropriate logs in the service code itself. For example, you can log the start and end time of each request and calculate the difference between the two times, as in the following example in Python:


Setting log for in Ingress YAML file

```
# Variables to store total time and request count
total_time = 0
request_count = 0

# File to store the average response time
log_file = 'response_times.log'

@app.route('/my-endpoint', methods=['GET'])
def handle_request():
    global total_time, request_count

    start_time = time.time()  # Start time

    .........

    end_time = time.time()  # End time

    # Calculate the duration of the request handling
    total_time += (end_time - start_time)
    request_count += 1

    # Calculate the average response time
    if request_count > 0:
        average_response_time = total_time / request_count
    else:
        average_response_time = 0


    # Log the average response time to a file
    with open(log_file, 'a') as f:
        f.write(f"Av resp time:{average_response_time:.4f} sec\n")

    return response
```

- define the out-of-service probability (P_B), which represents the percentage of requests that the system rejects once the maximum available servant capacity is reached. This blocking probability should be determined based on the quality requirements for the application.

- determine the optimal number of servants, apply Erlang's principle B, given below:

$$P_B = \frac{\frac{A^m}{m!}}{\sum_{k=0}^{m} \frac{A^k}{k!}}, \quad con \quad A = \lambda\, T_s$$

This principle allows the optimal number of servants to be calculated as a function of the desired blocking probability, frequency of request arrival, and service time. Direct calculation can be complex, so the Erlang B formula is often inverted using computational software or iterative attempts with different numbers of servers to obtain the desired blocking probability.

This methodology makes it possible to accurately estimate the optimal number of *pods* needed to ensure an adequate level of service, taking into account the average number of incoming requests and the desired blocking probability.

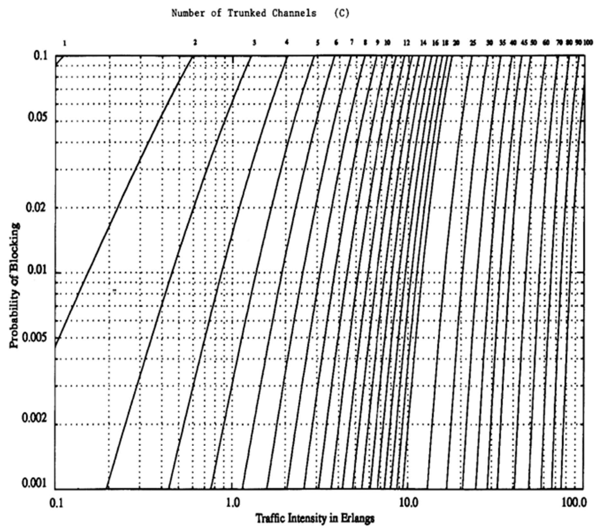


Fig. 3. Graph to determine the optimal number of pods as a function of the desired out-of-service probability and the traffic offered.

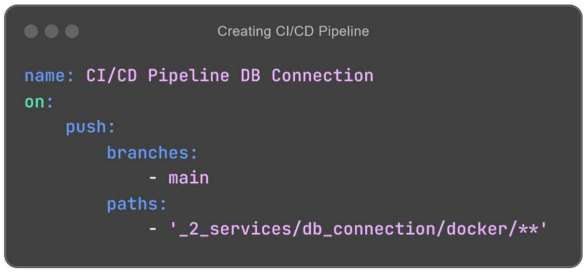
VI. DEVOPS AUTOMATION

Through the implementation of an automated pipeline based on GitHub Actions, a structured process for the continuous lifecycle management of software services has been implemented. This automated mechanism is triggered by commits or pushes made to the repository, ensuring a series of sequential operations that are critical to maintaining software quality. In detail, the automated process encompasses updating Docker images, running unit tests in locally, creating temporary test *Deployments* and *Services*, and finally updating the *Deployments* and *Services* in production. This continuous integration and deployment (CI/CD) mode not only optimizes workflow, but also ensures a safe and reliable software release cycle, minimizing the risk of introducing errors into production versions.

Let's analyze an example of CI/CD Pipeline, that is, how the automation proceeds after *commit* and *push* regarding the *db_connection* service.

A. Initialize CI/CD environment

First and foremost, within the *.github/workflows* directory of the repository, a dedicated YAML file must be created for each service for which an automated workflow is to be developed, intended to be executed following every commit or push.

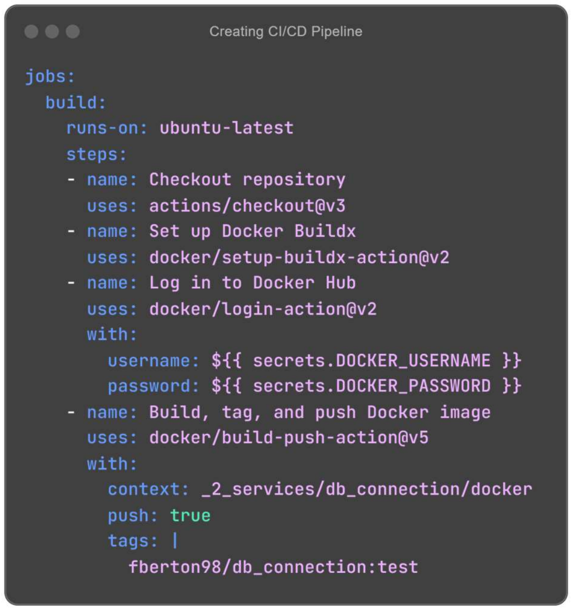


Creating CI/CD Pipeline

```yaml
name: CI/CD Pipeline DB Connection
on:
    push:
        branches:
            - main
        paths:
            - '_2_services/db_connection/docker/**'
```

B. Updating Docker images

Subsequently, the process for updating the Docker image of the service must be defined. At this stage, the image will be generated with a *test* tag, as it will initially be employed for testing purposes.



Creating CI/CD Pipeline

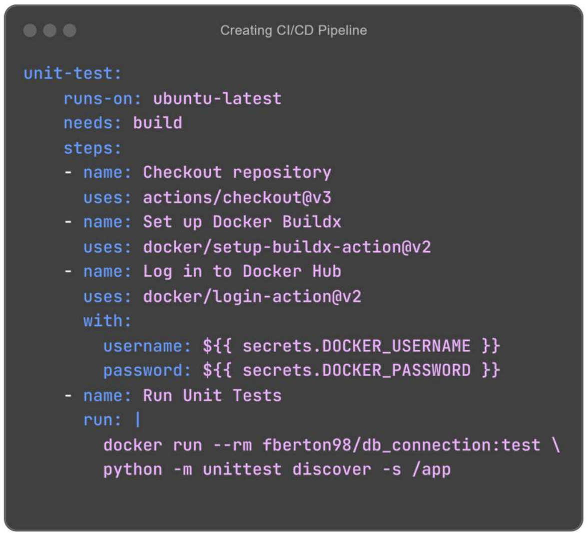
```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout repository
      uses: actions/checkout@v3
    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2
    - name: Log in to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}
    - name: Build, tag, and push Docker image
      uses: docker/build-push-action@v5
      with:
        context: _2_services/db_connection/docker
        push: true
        tags: |
          fberton98/db_connection:test
```

C. Running Unittest

After the release of the new version's image, it is essential to conduct a thorough verification of the service's functionality in a local environment. This preliminary testing phase is crucial to ensure that the changes introduced have not caused regressions or critical defects.

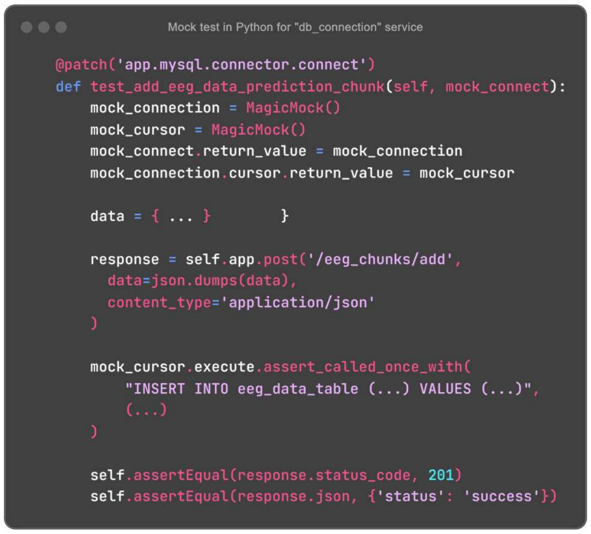
In the context of these tests, *Mocks* are employed to isolate and focus the analysis exclusively on the service under examination. The use of Mocks is vital for replicating external interactions (originating from other services) without relying on actual production resources or environments, thus enabling an isolated evaluation of the newly implemented features or fixes.

The test pipeline was implemented for the epileptogenic prediction service and the access service toward persistent data. The part of the YAML file related to the Unit Test of the latter service is shown below:


Creating CI/CD Pipeline

```
unit-test:
    runs-on: ubuntu-latest
    needs: build
    steps:
    - name: Checkout repository
      uses: actions/checkout@v3
    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v2
    - name: Log in to Docker Hub
      uses: docker/login-action@v2
      with:
        username: ${{ secrets.DOCKER_USERNAME }}
        password: ${{ secrets.DOCKER_PASSWORD }}
    - name: Run Unit Tests
      run: |
        docker run --rm fberton98/db_connection:test \
        python -m unittest discover -s /app
```

The Python code segment invoked by the Pipeline for executing the Unit Test is shown below. In this context, the role of Mock is to emulate the execution of the query directed toward the persistent database within the cluster:

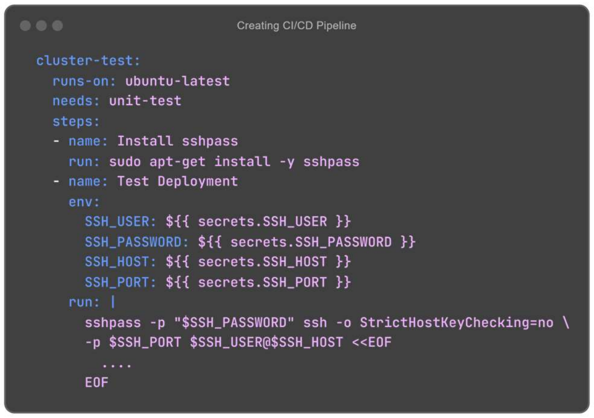

Mock test in Python for "db_connection" service

```
@patch('app.mysql.connector.connect')
def test_add_eeg_data_prediction_chunk(self, mock_connect):
    mock_connection = MagicMock()
    mock_cursor = MagicMock()
    mock_connect.return_value = mock_connection
    mock_connection.cursor.return_value = mock_cursor

    data = { ... }        }

    response = self.app.post('/eeg_chunks/add',
      data=json.dumps(data),
      content_type='application/json'
    )

    mock_cursor.execute.assert_called_once_with(
        "INSERT INTO eeg_data_table (...) VALUES (...)",
        (...)
    )

    self.assertEqual(response.status_code, 201)
    self.assertEqual(response.json, {'status': 'success'})
```

D. *Creation of temporary test deployments/services*

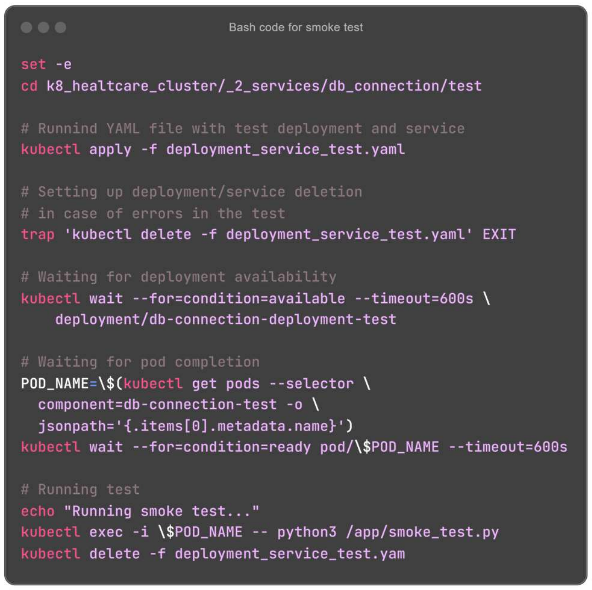
If the unit tests are successful, the next step involves creating a temporary *Deployment* and *Service* within the cluster. This phase allows for more thorough testing, ensuring an accurate validation of the service's behavior in an environment with real operational conditions and interacting with other actual services. It is therefore essential to establish an SSH connection to the Master node server, initiate the creation of the new test *Deployments* and *Services*, and await their activation before proceeding with the testing phase. Regardless of the result of this test, such items will be removed from the cluster.

The *Deployment/Service* test portion of the YAML file is shown below; it utilizes the "Secrets" stored on GitHub, including the hostname and port generated by Ngrok, to establish a connection with the virtual machine hosting the Master node:

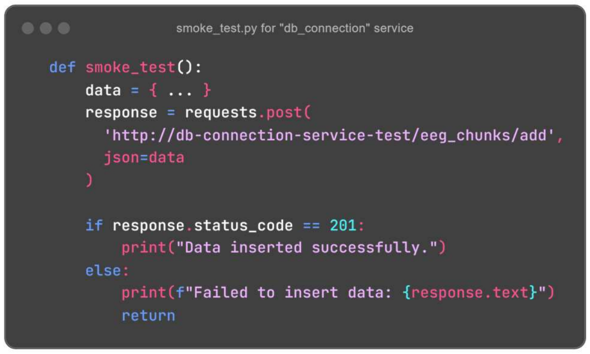

Creating CI/CD Pipeline

```
cluster-test:
  runs-on: ubuntu-latest
  needs: unit-test
  steps:
  - name: Install sshpass
    run: sudo apt-get install -y sshpass
  - name: Test Deployment
    env:
      SSH_USER: ${{ secrets.SSH_USER }}
      SSH_PASSWORD: ${{ secrets.SSH_PASSWORD }}
      SSH_HOST: ${{ secrets.SSH_HOST }}
      SSH_PORT: ${{ secrets.SSH_PORT }}
    run: |
      sshpass -p "$SSH_PASSWORD" ssh -o StrictHostKeyChecking=no \
      -p $SSH_PORT $SSH_USER@$SSH_HOST <<EOF
        ....
      EOF
```

The following section presents the Bash script corresponding to the previously shown code. This portion ensures that the test *pod* is fully ready before proceeding with the official test, which is conducted by the subsequent Python code.


Bash code for smoke test

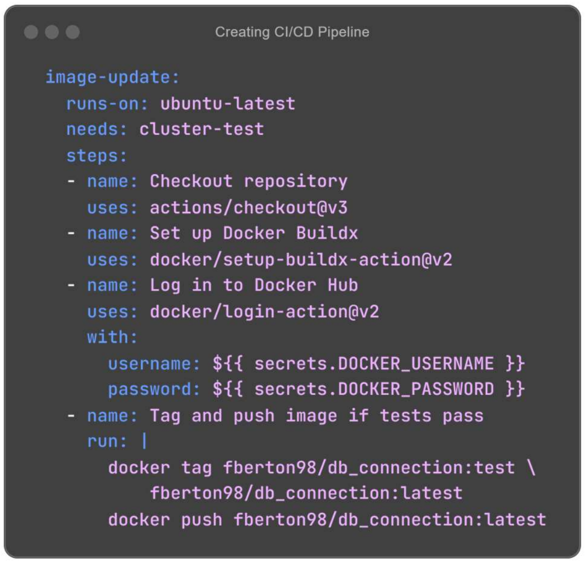
```
set -e
cd k8_healtcare_cluster/_2_services/db_connection/test

# Runnind YAML file with test deployment and service
kubectl apply -f deployment_service_test.yaml

# Setting up deployment/service deletion
# in case of errors in the test
trap 'kubectl delete -f deployment_service_test.yaml' EXIT

# Waiting for deployment availability
kubectl wait --for=condition=available --timeout=600s \
    deployment/db-connection-deployment-test

# Waiting for pod completion
POD_NAME=\$(kubectl get pods --selector \
  component=db-connection-test -o \
  jsonpath='{.items[0].metadata.name}')
kubectl wait --for=condition=ready pod/\$POD_NAME --timeout=600s

# Running test
echo "Running smoke test..."
kubectl exec -i \$POD_NAME -- python3 /app/smoke_test.py
kubectl delete -f deployment_service_test.yam
```


smoke_test.py for "db_connection" service

```
def smoke_test():
    data = { ... }
    response = requests.post(
      'http://db-connection-service-test/eeg_chunks/add',
      json=data
    )

    if response.status_code == 201:
        print("Data inserted successfully.")
    else:
        print(f"Failed to insert data: {response.text}")
        return
```

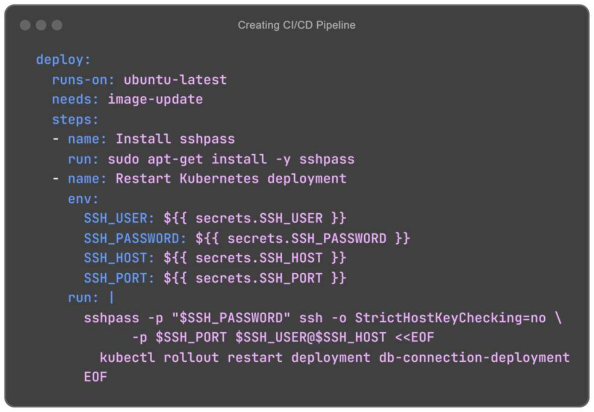
E. *Updating production deployments/services*

If all tests have been successful, we can now proceed with upgrading services in production. As a first step, it is necessary to update the pointing of the *latest* Docker image to the *test* one:


Creating CI/CD Pipeline

```
image-update:
  runs-on: ubuntu-latest
  needs: cluster-test
  steps:
  - name: Checkout repository
    uses: actions/checkout@v3
  - name: Set up Docker Buildx
    uses: docker/setup-buildx-action@v2
  - name: Log in to Docker Hub
    uses: docker/login-action@v2
    with:
      username: ${{ secrets.DOCKER_USERNAME }}
      password: ${{ secrets.DOCKER_PASSWORD }}
  - name: Tag and push image if tests pass
    run: |
      docker tag fberton98/db_connection:test \
          fberton98/db_connection:latest
      docker push fberton98/db_connection:latest
```

Next, restarting the corresponding deployment will allow you to build it with the updated image:


Creating CI/CD Pipeline

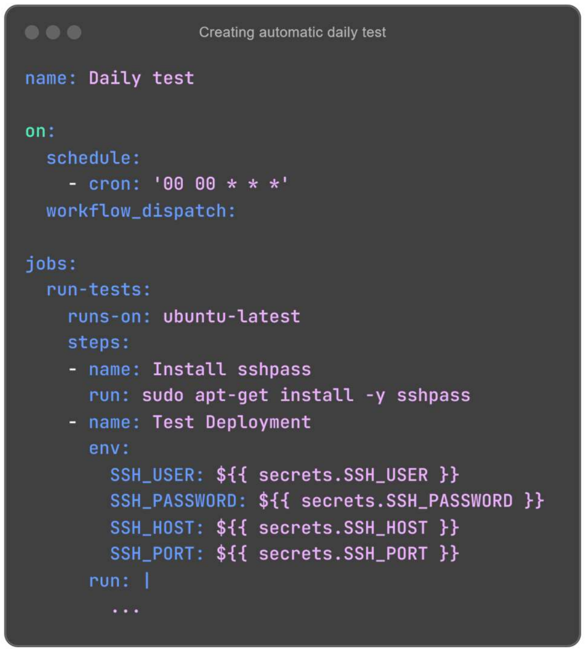
```
deploy:
  runs-on: ubuntu-latest
  needs: image-update
  steps:
  - name: Install sshpass
    run: sudo apt-get install -y sshpass
  - name: Restart Kubernetes deployment
    env:
      SSH_USER: ${{ secrets.SSH_USER }}
      SSH_PASSWORD: ${{ secrets.SSH_PASSWORD }}
      SSH_HOST: ${{ secrets.SSH_HOST }}
      SSH_PORT: ${{ secrets.SSH_PORT }}
    run: |
      sshpass -p "$SSH_PASSWORD" ssh -o StrictHostKeyChecking=no \
            -p $SSH_PORT $SSH_USER@$SSH_HOST <<EOF
        kubectl rollout restart deployment db-connection-deployment
      EOF
```

At this stage, the Kubernetes cluster is fully updated, with its functionality assured by the successful completion of all tests.

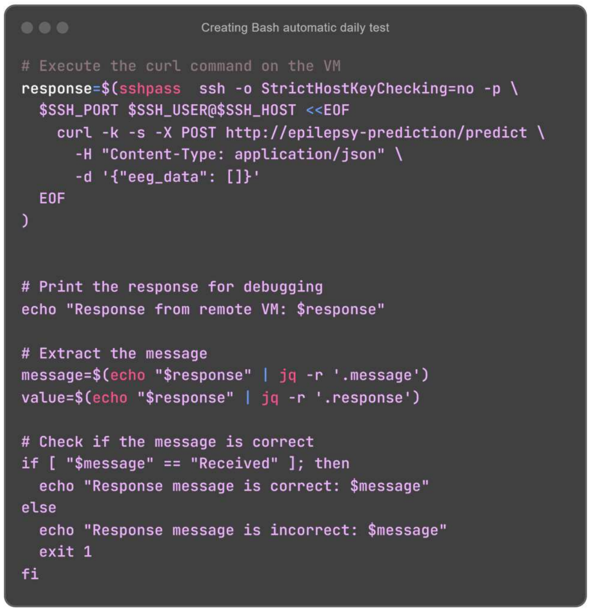
F. *Automatic periodic testing*

The introduction of periodic automated tests, performed on a daily basis, complements the tests executed post-build to ensure continuous evaluation of the system. Periodic tests, therefore, are necessary to monitor the stability and performance of the system over time; they ensure that any regressions in performance or anomalies in system behavior are detected and addressed promptly.

Below is the YAML file for creating automated periodic tests and related Bash code:


Creating automatic daily test

```
name: Daily test

on:
  schedule:
    - cron: '00 00 * * *'
  workflow_dispatch:

jobs:
  run-tests:
    runs-on: ubuntu-latest
    steps:
    - name: Install sshpass
      run: sudo apt-get install -y sshpass
    - name: Test Deployment
      env:
        SSH_USER: ${{ secrets.SSH_USER }}
        SSH_PASSWORD: ${{ secrets.SSH_PASSWORD }}
        SSH_HOST: ${{ secrets.SSH_HOST }}
        SSH_PORT: ${{ secrets.SSH_PORT }}
      run: |
        ...
```


Creating Bash automatic daily test

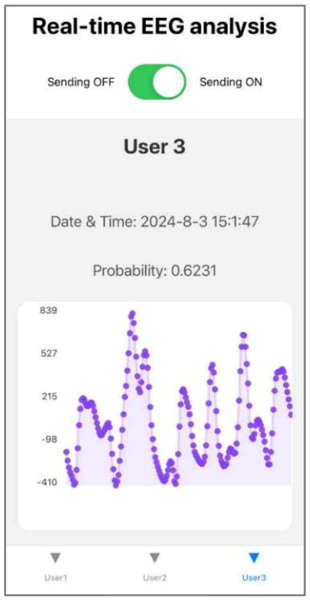
```
# Execute the curl command on the VM
response=$(sshpass  ssh -o StrictHostKeyChecking=no -p \
  $SSH_PORT $SSH_USER@$SSH_HOST <<EOF
    curl -k -s -X POST http://epilepsy-prediction/predict \
      -H "Content-Type: application/json" \
      -d '{"eeg_data": []}'
  EOF
)


# Print the response for debugging
echo "Response from remote VM: $response"

# Extract the message
message=$(echo "$response" | jq -r '.message')
value=$(echo "$response" | jq -r '.response')

# Check if the message is correct
if [ "$message" == "Received" ]; then
  echo "Response message is correct: $message"
else
  echo "Response message is incorrect: $message"
  exit 1
fi
```

VII. MOBILE APPS FOR HEALT DATA REGISTRATION

A. *Real-time EEG data simulation*

In recent years, advanced technologies have been developed for managing electroencephalographic data, involving the implantation of persistent electrodes in the human skull. These electrodes, powered by electrical energy, record brain data and transmit it in real-time to smartphones or directly to cloud platforms. This system enables continuous and immediate monitoring of potential epileptic issues, significantly improving the diagnosis and management of neurological disorders.

To test the developed cluster, an app was created using React Native. This app allows the transmission of previously recorded electroencephalographic data (obtained from diagnostic EEG exams) to the cluster's endpoint service. To verify the correct reception and processing of data, we can analyze the real-time data visualization service in the browser.



Real-time EEG analysis

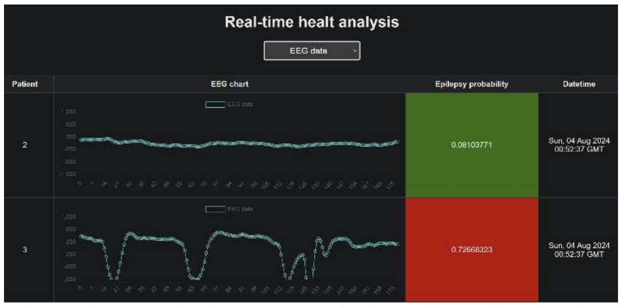
Sending OFF / Sending ON

User 3

Date & Time: 2024-8-3 15:1:47

Probability: 0.6231

Fig. 4. Mobile app used to test EEG data transmission



Real-time healt analysis

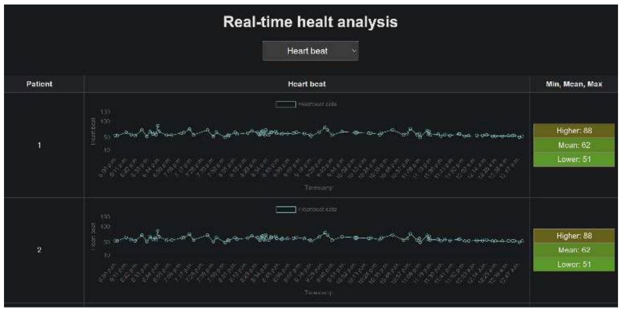
EEG data

Patient	EEG chart	Epilepsy probability	Datetime
2		0.08103771	Sun, 04 Aug 2024 00:52:37 GMT
3		0.72668323	Sun, 04 Aug 2024 00:52:37 GMT

Fig. 5. Cluster service that exposes received EEG data and its processing

B. *Heartbeat registration*

For tests concerning the recording, sharing, and processing of heart rate data, an iOS application available on iPhones was used. This application allows real-time retrieval of data recorded by the Apple Watch and sends it via REST API to the developed cluster's endpoint service. This enables healthcare personnel to monitor and identify potential cardiac issues. Just as with the EEG data, to verify the accuracy of this recording, we can monitor the trends through the web page service of the cluster.



Real-time healt analysis

Heart beat

Patient	Heart beat	Min, Mean, Max
1		Higher: 88 Mean: 62 Lower: 51
2		Higher: 88 Mean: 62 Lower: 51

Fig. 6. Cluster service that exposes received heart beat data and its processing

VIII. Concludin Remarks

The project developed is a prototype Kubernetes cluster designed to manage health information in real time. This system is designed to immediately receive, process, and analyze critical health data, enabling providers to continuously monitor patient conditions and identify potential health problems.

The relevance of this system lies in its ability to provide continuous and accurate monitoring, thereby improving the quality and timeliness of medical care. In a healthcare setting where every second is crucial, responsiveness in an emergency can make all the difference. In addition, predictive analysis of real-time data allows for anticipating complications and optimizing the treatment path for each patient.

The choice of Kubernetes as the platform for this project is essential. Kubernetes offers scalability to handle large volumes of data, along with the reliability and resilience needed in a critical environment such as healthcare. By automating the deployment, management, and monitoring of applications, Kubernetes ensures that the system remains up and running and performing even under conditions of high utilization or hardware failure.

In summary, the developed prototype aims to emulate the integration of advanced technologies in healthcare. It aims to improve patient monitoring and quality of care, paving the way for future innovations that promote more responsive and personalized health care.

IX. References

[1] Reali, Gianluca. Virtual Network & Cloud Computing, University of Perugia.

[2] Kubernetes Documentation. Kubernetes.io. Available: https://kubernetes.io/docs

[3] MetalLB project, "MetalLB: A Load-Balancer for Kubernetes on Bare Metal,". Available: https://metallb.universe.tf/

[4] Ngrok: Secure introspectable tunnels to localhost. Available: https://ngrok.com