Kubernetes per i servizi di assistenza sanitaria

Federico Berton
Ingegneria Informatica e Robotica
Virtual Network & Cloud Computing
Università degli Studi di Perugia
Perugia, Italia
federicoberton.fb@gmail.com

Nell'era digitale, il settore sanitario sta vivendo una profonda trasformazione alimentata dai progressi tecnologici. Questo progetto sfrutta le capacità di Kubernetes per migliorare l'accessibilità e la qualità dei servizi sanitari. Integrando dispositivi medici all'avanguardia per l'acquisizione di dati clinici, il sistema facilita il monitoraggio preciso e in tempo reale dei segni vitali e delle metriche sanitarie critiche, migliorando così l'assistenza ai pazienti e consentendo interventi tempestivi.

I. Introduzione

Con l'avanzare delle capacità dei dispositivi medici digitali, diventa essenziale disporre di strumenti per la raccolta e l'analisi in tempo reale dei dati generati da questi dispositivi. Questo progetto mira a sviluppare un'infrastruttura basata su Kubernetes, progettata per garantire una gestione efficiente e scalabile dei dati clinici. Kubernetes, noto per la sua capacità di orchestrare i container in modo dinamico e resiliente, funge da hub per la ricezione, l'elaborazione e l'archiviazione sicura delle informazioni, garantendone l'integrità e la disponibilità costante. I dati elaborati sono resi accessibili attraverso interfacce di facile utilizzo, consentendo agli operatori sanitari di monitorare costantemente e dettagliatamente lo stato di salute dei pazienti. Questo approccio non solo migliora la velocità e l'accuratezza delle decisioni cliniche, ma consente anche di effettuare analisi predittive, identificando precocemente i potenziali rischi e ottimizzando le strategie di trattamento.

In questo progetto vengono implementati due servizi primari: uno riguardante la registrazione della frequenza cardiaca e un altro relativo all'attività elettrica cerebrale.

II. Progettazione dell'infrastruttura e automazione

Innanzitutto, è fondamentale esaminare con attenzione l'architettura del progetto, che è stata ideata per garantire, una volta che il cluster Kubernetes è attivo e funzionante, un'automazione dei processi di testing e deployment in risposta a ogni modifica del codice sorgente. Questo meccanismo di automazione si basa sull'uso di GitHub Actions, un framework che consente l'integrazione e la consegna continua (CI/CD) in modo fluido ed efficiente. Per ogni servizio, le GitHub Actions orchestrano la creazione di immagini Docker, eseguono una suite completa di test automatizzati e gestiscono il deployment della versione finale sul cluster Kubernetes.

Entriamo ora nel dettaglio del processo per realizzare questa configurazione: innanzitutto, è essenziale determinare le macchine fisiche o virtuali che verranno utilizzate. In seguito, è necessario definire la configurazione di Kubernetes, compreso il numero di nodi e altre specifiche, e procedere alla sua installazione. Una volta configurato Kubernetes, possiamo gestire l'integrazione con GitHub Actions e Docker Hub per automatizzare i flussi di lavoro e distribuire in modo efficiente i servizi desiderati.

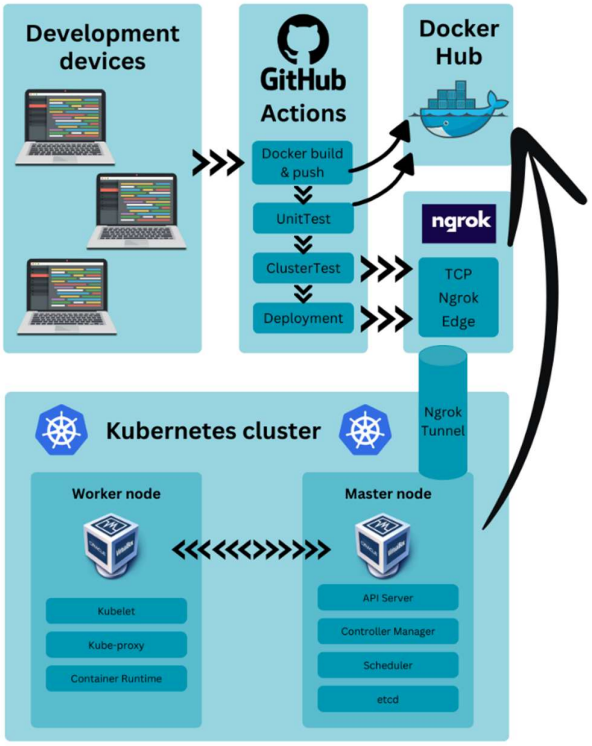

Fig. 1. Progettazione dell'infrastruttura

A. *Simulazione del server e creazione del cluster Kubernetes*

Un cluster Kubernetes può essere testato utilizzando macchine virtuali su un normale computer portatile. In particolare, è possibile creare due macchine virtuali Linux: una come nodo Master e l'altra come nodo Worker. Il nodo Master supervisiona l'intero cluster, coordinando i vari servizi. Le sue funzioni principali includono [2]:

- **API Server:** gestisce le richieste in arrivo (tramite API REST) e funge da interfaccia principale per interagire con il cluster.
- **etcd:** memorizza i dati di configurazione del cluster in modo distribuito e coerente.
- **Controller manager:** esegue i controller che monitorano lo stato del cluster e si assicurano che corrisponda allo stato desiderato.
- **Scheduler:** determina quale nodo deve eseguire un nuovo *pod*, considerando fattori come le risorse disponibili e altro.

Il nodo Worker è responsabile dell'esecuzione effettiva delle applicazioni. Le sue funzioni principali includono [2]:

- **Kubelet:** responsabile della gestione dei *pod*; si assicura che i contenitori siano in esecuzione come specificato e comunica con il server API del master.
- **Kube-proxy:** gestisce le regole di rete sui nodi, consentendo la comunicazione tra i diversi servizi all'interno del cluster.
- **Container runtime:** il componente che esegue effettivamente i container (ad esempio, Docker).

Le macchine virtuali possono essere configurate in diverse modalità di rete in base ai requisiti:

- Se la comunicazione è necessaria solo tra i nodi Master e Worker e non con dispositivi esterni, è sufficiente configurare le macchine virtuali con una rete NAT. Questa configurazione isola il cluster dalle reti esterne, pur consentendo la comunicazione interna tra i nodi.
- Per consentire la comunicazione tra il cluster Kubernetes e i dispositivi esterni (come il laptop host contenente le macchine virtuali o altri dispositivi sulla stessa rete Wi-Fi), le macchine virtuali devono essere configurate con un'impostazione di rete *Bridge adapter*. Questa configurazione consente al cluster di interagire sia con il sistema host che con altri dispositivi sulla rete.

Una volta create, le macchine virtuali devono essere configurate per l'indirizzo IP e l'hostname [1]:

- Modificare il file di configurazione di netplan per impostare l'indirizzo IP statico. Il file si trova solitamente in */etc/netplan/* e potrebbe essere identificato come *01-netcfg.yaml* o *01-network-manager-all.yaml*.
- Per garantire la corretta risoluzione del nome di host all'interno del cluster, modificare */etc/hosts* e */etc/hostname* sia nel Master che nel Worker.

A questo punto si può procedere all'installazione di Kubernetes. Un metodo semplice per questo processo è l'utilizzo di Kubespray [1], che semplifica la distribuzione automatizzando l'impostazione e la configurazione di cluster Kubernetes su più nodi.

B. Tunnel Ngrok [4]

Ngrok è stato utilizzato per esporre il nodo master Kubernetes di una macchina virtuale per facilitare l'integrazione con GitHub Actions. Nonostante l'uso di un *Bridge adapter* nella configurazione della macchina virtuale, la connessione alla rete esterna è limitata alla sottorete del Wi-Fi in uso. Ngrok ha permesso di creare un tunnel sicuro, fornendo un URL per il nodo master Kubernetes. Dopo aver creato il proprio token nell'account Ngrok, è possibile creare una connessione tunnel con il nodo master tramite un collegamento TCP:

Ngrok setup

```
# 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 e DockerHub

Docker è stato utilizzato per costruire, gestire e distribuire servizi in container isolati, sviluppati in Python utilizzando il framework Flask. Di seguito vengono riportate le informazioni necessarie per creare l'immagine:

Dockerfile example

```
# 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"]
```

Il file *requirements.txt* contiene le dipendenze di cui un tale servizio ha bisogno. Infine, per distribuire i servizi sviluppati è necessario creare le immagini Docker locali e caricarle in DockerHub:

Building and publishing image

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

Questo approccio assicura che i servizi siano isolati e facilmente riproducibili, semplificando al contempo il ciclo di sviluppo e distribuzione. Favorisce così una pipeline di rilascio affidabile e scalabile..

D. GitHub e GitHub Actions

GitHub Actions viene utilizzato per automatizzare l'intero ciclo di vita dei processi di sviluppo, test e distribuzione dei servizi. La sua integrazione ha permesso di attivare automaticamente flussi di lavoro predefiniti, scritti in file YAML, a ogni push del repository. Questi flussi di lavoro includono la creazione di immagini Docker, l'esecuzione di test unitari, la creazione di ambienti di test temporanei e infine l'aggiornamento delle distribuzioni di produzione se tutti i test

hanno avuto successo. L'uso delle GitHub Actions garantisce un processo veloce e affidabile, riducendo al minimo l'intervento manuale e migliorando l'efficienza della pipeline di rilascio.

III. Struttura del progetto

Il progetto è strutturato in modo da garantire una gestione efficiente e sicura dei dati clinici attraverso un'architettura basata sui servizi. Il componente centrale del sistema è un database persistente progettato per archiviare in modo sicuro tutti i dati in entrata. Questo database funge da repository centrale, consentendo l'archiviazione e l'accesso continuo ai dati clinici.

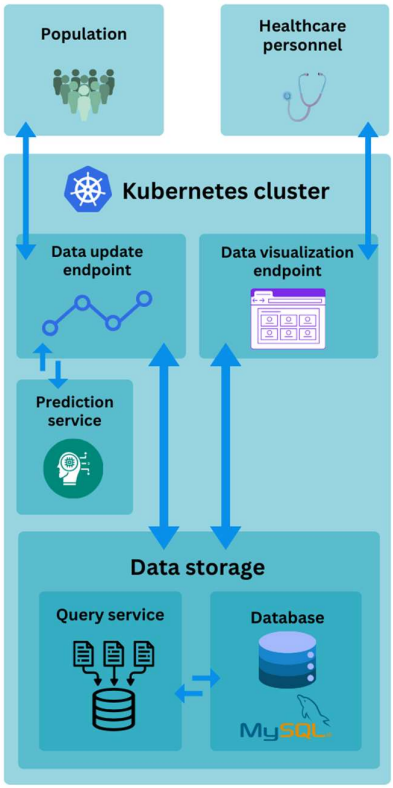

Fig. 2. Struttura del progetto

A. Servizi

Il sistema è suddiviso in diversi servizi, ciascuno con una funzione specifica:

- ***Acquisizione dei dati elettroencefalografici (EEG):*** questo servizio è responsabile dell'ascolto e della ricezione dei dati elettroencefalografici dei soggetti collegati.
- ***Monitoraggio della frequenza cardiaca:*** gestisce l'acquisizione dei dati sulla frequenza cardiaca dei soggetti connessi, ricevendo i dati in tempo reale attraverso dispositivi indossabili compatibili.
- ***Elaborazione predittiva dell'EEG:*** utilizzando un algoritmo predittivo basato su reti neurali, questo servizio elabora i dati EEG per determinare la presenza di episodi epilettici. L'algoritmo è stato addestrato su un set di dati completo per garantire un'elevata precisione e affidabilità.
- ***Interrogazione del database:*** facilita l'interrogazione del database persistente, consentendo l'inserimento di nuovi dati e il recupero di quelli precedentemente registrati.
- ***Visualizzazione dei dati in tempo reale:*** progettato per esporre i dati registrati in tempo reale, questo servizio fornisce una pagina web accessibile al personale sanitario. La pagina web viene aggiornata continuamente per riflettere i dati più recenti.

B. Applicazioni mobili di test

Per supportare lo sviluppo e la convalida del sistema, è stata creata un'applicazione di prova per simulare la trasmissione di dati elettroencefalografici. Questa applicazione utilizza dati EEG precedentemente salvati per emulare scenari reali di acquisizione e trasmissione dei dati.

Inoltre, è stata integrata un'applicazione iPhone esistente per estrarre i dati sulla frequenza cardiaca registrati dall'Apple Watch e trasmetterli in tempo reale a un'API REST dedicata. Questa integrazione consente di utilizzare dispositivi ampiamente disponibili e affidabili per la raccolta dei dati sulla frequenza cardiaca.

IV. Sviluppo dei servizi in Kubernetes

Questa sezione approfondisce il processo di orchestrazione del livello applicativo all'interno del cluster Kubernetes, a partire dalla procedura di containerizzazione di ogni servizio con Docker. In seguito, verrà descritta l'esecuzione di test unitari per convalidare il comportamento del singolo servizio. In caso di successo, si affronterà l'implementazione di test che verifichino il corretto funzionamento delle funzionalità di base di un servizio nel cluster stesso. Infine, si procederà al deployment definitivo, garantendo l'integrazione del servizio nel sistema di produzione.

A. Deployment e servizi Kubernetes

Dopo aver creato l'immagine Docker, è possibile distribuirlo nel cluster Kubernetes. Per farlo in modo efficiente, creare un file *deployment_service.yaml* che includa sia la configurazione del deployment che quella del servizio.

Un *Deployment* gestisce la creazione e l'aggiornamento dei *pod* (unità operative), mentre un *Service* espone un gruppo di *pod*, permettendo loro di essere raggiunti stabilmente da altri servizi o utenti, indipendentemente dai singoli *pod* che possono essere creati o distrutti. Per il deployment, vengono specificati i seguenti aspetti:

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
```

Impostazioni del Deployment nel file YAML	
Chiavi	***Descrizione***
name	Identifica in modo univoco il *Deployment* all'interno del cluster Kubernetes
labels: app	Indica il nome dell'applicazione associata al *Deployment*
labels: component	Specifica il componente dell'applicazione da utilizzare per il *Deployment*
image	Specifica l'immagine Docker da utilizzare per istanziare il *Deployment* e i *pod* associati
containerPort	Definisce la porta all'interno del container su cui l'applicazione è in ascolto (dovrebbe corrispondere a quella configurata nell'immagine Docker)
resources: requests	Quantità minima di risorse che il *pod* richiede per avviarsi
resources: limits	Quantità massima di risorse che il *pod* più utilizzare

Per quanto riguarda *Service*, vengono stabiliti i seguenti parametri:

YAML file for epilepsy prediction service

```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
```

Impostazioni del Service nel file YAML	
Chiavi	***Descrizione***
name	Assegna un'identità unica al servizio
labels: app	Identifica l'applicazione riguardo cui servizio vuole esporre
labels: component	Specifica il componente dell'applicazione a cui è diretto il servizio
protocol	Determina il protocollo di comunicazione utilizzato dal servizio
type	Definisce le modalità di bilanciamento del traffico tra i *pod* e, in aggiunta, anche l'esposizione all'esterno del cluster tramite IP o DNS

Per i servizi accessibili solo all'interno del cluster, è sufficiente configurare il tipo *ClusterIP*. Tuttavia, se è necessario rendere i servizi accessibili dall'esterno del cluster, è necessario utilizzare il tipo *LoadBalancer*. Inoltre, qualora si desideri gestire l'esposizione dei servizi esterni tramite *Ingress*, anche i servizi destinati a ricevere traffico esterno possono essere configurati come *ClusterIP*, poiché è *l'Ingress Controller* stesso a occuparsi dell'esposizione degli endpoint all'esterno del cluster.

In relazione alle distribuzioni, un *HorizontalPodAutoscaler* (HPA) può essere configurato per regolare dinamicamente il numero di repliche *pod* in base all'utilizzo delle risorse. L'HPA in Kubernetes necessita di un *metrics server* per funzionare correttamente. Esso raccoglie e fornisce all'HPA dati sull'utilizzo delle risorse dei *pod*, come CPU e memoria. Senza queste metriche, l'HPA non sarebbe in grado di determinare quando scalare il numero di *pod* in base al carico di lavoro [1].

YAML file for epilepsy prediction HPA

```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
```

Impostazioni di HorizontalPodAutoscaler nel file YAML	
Chiavi	***Descrizione***
name	Specifica il nome univoco della risorsa *HorizontalPodAutoscaler*
minReplicas	Definisce il numero minimo di repliche del pod che l'HPA manterrà
maxReplicas	Definisce il numero massimo di repliche dei pod che l'HPA manterrà
averageUtilization	Indica l'utilizzo medio della CPU di tutti i *pod*, per determinare quando scalare il numero di *pod* verso l'alto o verso il basso.
scaleTargetRef: kind	Specifica il tipo di risorsa a cui l'HPA si rivolge per il ridimensionamento (tipicamente Deployment)
scaleTargetRef: name	Si riferisce al nome della risorsa che viene scalata.

Insieme, questi elementi consentono una gestione e un'esposizione efficace delle applicazioni all'interno e all'esterno del cluster, ottimizzando scalabilità e resilienza.

B. MetalLB

Per fornire il bilanciamento del carico per i servizi Kubernetes in ambienti come le macchine virtuali, che non dispongono del supporto nativo del bilanciatore di carico offerto dai fornitori di cloud pubblici, MetalLB può essere configurato per gestire questo compito [3]. MetalLB offre le seguenti funzionalità:

- ***Esposizione di servizi con IP esterni***: MetalLB assegna indirizzi IP esterni ai servizi Kubernetes, rendendoli accessibili al di fuori del cluster. Ciò consente agli utenti e alle applicazioni esterne della stessa subnet di connettersi ai servizi Kubernetes attraverso indirizzi IP designati.
- ***Gestione del traffico***: utilizzando il protocollo ARP MetalLB gestisce il traffico di rete assegnando indirizzi IP esterni e indirizzandolo ai *pod* appropriati all'interno del cluster. Questo emula il comportamento di un bilanciatore di carico tradizionale, garantendo una distribuzione efficiente del traffico.

Per raggiungere questo obiettivo, è necessario [1]:

- Installare MetalLB nel cluster.
- Configurare il pool di indirizzi IP, creando un oggetto IPAddressPool che definisca la gamma di indirizzi IP disponibili per MetalLB..

C. Database

Nel cluster Kubernetes è stato configurato un database MySQL persistente per archiviare e gestire in modo sicuro i dati relativi alla salute. Questa configurazione prevede diversi componenti chiave [2]:

Risorse del cluster definite per il database	
Componenti	***Descrizione***
PersistentVolume	Risorsa dello storage nel cluster Kubernetes che fornisce spazio di archiviazione persistente
PersistentVolumeClaim	Specifica la quantità di storage necessaria e la modalità di accesso. Quando il *PVC* viene soddisfatto da un *PV*, il *pod* MySQL può montare questo storage e utilizzarlo per salvare i dati del database
Secret	Contiene le credenziali necessarie per l'accesso al database
Deployment	Una risorsa Kubernetes che automatizza la creazione, l'aggiornamento e la gestione del *pod* MySQL
Service	Fornisce un nome di rete stabile e un indirizzo IP fisso per accedere al database

Questa configurazione completa garantisce che il database MySQL sia in grado di mantenere la disponibilità continua e l'integrità dei dati sanitari all'interno dell'ambiente Kubernetes.

D. Ingress

Per gestire l'accesso esterno ai servizi all'interno di un cluster Kubernetes, *Ingress* instrada in modo efficiente il traffico HTTP/S verso i servizi appropriati in base a regole

predefinite. Fornisce un punto centralizzato per la gestione del traffico in entrata, evitando la necessità di configurare i singoli servizi per gestire l'instradamento. Per configurarlo, è necessario:

1. Applicare la configurazione predefinita (dal repository ufficiale di Kubernetes Ingress NGINX) per il *controller NGINX Ingress*, che gestirà le regole di instradamento per il traffico HTTP/S.
2. Creare una nuova *Ingress Class* che utilizzi il controllore NGINX.

YAML file for IngressClass

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

3. Creare una risorsa di *Ingress* che gestisca l'instradamento del traffico HTTP/S in base a regole definite. Indirizza il traffico verso servizi specifici all'interno del cluster in base all'identificativo dell'host e al path richiesto, consentendo così l'accesso esterno ai servizi [1].

YAML file for Ingress set up

```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. Modificare il file */etc/hosts* del sistema operativo per associare il dominio all'indirizzo IP specifico del controller di *Ingress*.

Modification to "/etc/hosts" file

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

V. Analisi Markoviana del numero di serventi

Il modello di code di Markov si può considerare uno strumento efficace per determinare il numero ottimale di *pod* in un cluster Kubernetes, grazie alla sua capacità di modellare sistemi complessi che operano sotto carichi variabili. Il modello M/M/m/m della teoria delle code assume che i processi di arrivo seguano una distribuzione di Poisson, che i tempi di servizio siano distribuiti esponenzialmente e che ci sia un numero fisso di servitori (m) senza possibilità di accogliere richieste in eccesso oltre la capacità del sistema (m), ossia senza coda di attesa. Poiché il cluster di Kubernetes in questione rifiuta o utilizza un time-out molto breve in caso ci sia una richiesta quando tutti i serventi sono occupati, è sensato approssimarlo come M/M/m/m.

Utilizzando tale modello, è possibile calcolare il traffico offerto, definito come il prodotto del tasso di arrivo delle richieste e il tempo medio di servizio, e successivamente applicare la formula di Erlang B per determinare il numero ideale di serventi. Sebbene il *HPA* automatizzi la gestione dei *pod*, è importante impostare un limite massimo di *pod* per evitare richieste eccessive e costi elevati, e ciò può essere approssimato tramite l'approccio Markoviano.

Essendo utilizzato un sistema a coda di tipo feed-forward, nel quale le richieste si muovono in un flusso diretto attraverso i vari servizi senza retroazioni, il teorema di Burke garantisce che il flusso di arrivi in uscita da ciascun servente segua una distribuzione di Poisson, permettendo così di trattare ogni servizio come indipendente dagli altri. Di conseguenza, per ogni servizio è essenziale analizzare separatamente la frequenza di arrivo delle richieste e il tempo di servizio, per poi determinare la probabilità di blocco in funzione del numero di serventi disponibili. In alternativa, è possibile condurre un'analisi preliminare sui volumi di richieste e sui tempi di risposta medi, per determinare il numero ottimale di serventi per garantire una probabilità di blocco desiderata. Questo processo può essere realizzato come segue:

- stabilire, per tale fase ti test, un numero fisso di *pod* del servizio da analizzare.
- effettuare un'analisi preliminare della frequenza di arrivo delle richieste (λ), inclusi i tentativi falliti o rifiutati. Questo monitoraggio può essere realizzato utilizzando strumenti come *Prometheus*, che consente di raccogliere metriche in tempo reale, oppure mediante un'analisi dei log dell'*Ingress*. Per attivare quest'ultima modalità è necessario configurare correttamente l'annotazione nell'*Ingress*, abilitando la registrazione dei dettagli delle richieste nel file di log:

Setting log for in Ingress YAML file

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

A partire dall'analisi dei log o delle metriche raccolte, è possibile stimare il numero totale di richieste che hanno interessato un determinato path (nell'esempio di seguito, "*/my-path*"):

Bash script for calculating total service request

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

- determinare il tempo di servizio (T_s) richiesto da ciascun servente per completare una singola richiesta. Questo valore può essere approssimato inserendo appositi log nel codice del servizio stesso. Ad esempio, è possibile registrare il tempo di inizio e fine di ogni richiesta e calcolare la differenza tra i due momenti, come nel seguente esempio in Python:

Setting log for in Ingress YAML file

```python
# 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
```

- definire la probabilità di fuori servizio (P_B), che rappresenta la percentuale di richieste che il sistema rifiuta una volta raggiunta la capacità massima di serventi disponibili. Questa probabilità di blocco deve essere determinata in base ai requisiti di qualità previsti per l'applicazione.
- determinare il numero ottimale di serventi, applicando il principio di Erlang B, riportato di seguito:

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

Questo principio consente di calcolare il numero ottimale di serventi in funzione della probabilità di blocco desiderata, la frequenza di arrivo delle richieste e del tempo di servizio. Il calcolo diretto può risultare complesso, quindi la formula di Erlang B viene spesso invertita utilizzando software di calcolo o tentativi iterativi con numero di serventi differente per ottenere la probabilità di blocco desiderata.

Questa metodologia consente di stimare con precisione il numero ottimale di *pod* necessari per garantire un livello di servizio adeguato, tenendo conto della media delle richieste in ingresso e della probabilità di blocco desiderata.

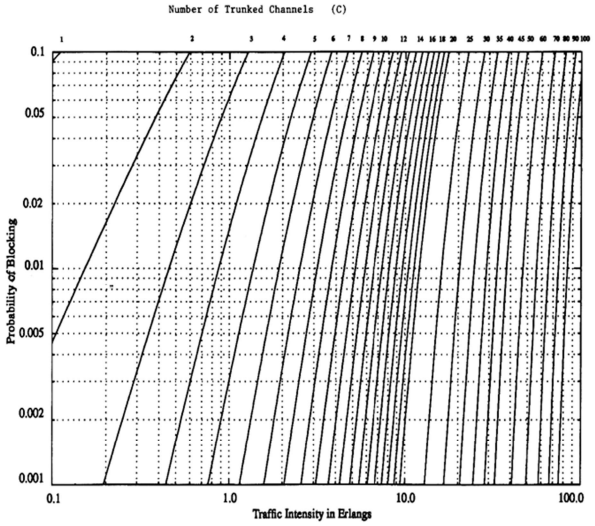

Fig. 3. Grafico che permette la determinazione del numero ottimale di pod in funzione della probabilità di fuori servizio desiderata e del traffico offerto

VI. AUTOMAZIONE DEVOPS

Attraverso l'implementazione di una pipeline automatizzata basata sulle GitHub Actions, è stato realizzato un processo per la gestione continua del ciclo di vita dei servizi software. Questo meccanismo automatizzato viene attivato dai comandi *push* effettuati sul repository, garantendo una serie di operazioni sequenziali che sono fondamentali per mantenere la qualità del software. In dettaglio, il processo automatizzato comprende l'aggiornamento delle immagini Docker, l'esecuzione di test unitari in locale, la creazione di *Deployments* e *Services* temporanei di prova, e infine l'aggiornamento dei relativi *Deployments* e dei *Services* in produzione. Questa modalità di integrazione e distribuzione continua (CI/CD) non solo ottimizza il flusso di lavoro, ma garantisce anche un ciclo di rilascio del software sicuro e affidabile, riducendo al minimo il rischio di introdurre errori nelle versioni di produzione.

Analizziamo un esempio di Pipeline CI/CD, ovvero come procede l'automazione dopo il *commit* e il *push* per quanto riguarda il servizio db_*connection*.

A. Initializzazione dell'ambiente CI/CD

Innanzitutto, all'interno della cartella *.github/workflows* del repository, è necessario creare un file YAML dedicato per ogni servizio per il quale si vuole sviluppare un flusso di lavoro automatizzato, destinato a essere eseguito dopo ogni comando *push*.

Creating CI/CD Pipeline

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

B. *Aggiornamento delle immagini Docker*

Successivamente, è necessario definire il processo di aggiornamento dell'immagine Docker del servizio. In questa fase, l'immagine sarà generata con un *tag* di *test*, poiché inizialmente sarà utilizzata a scopo di collaudo.

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. *Esecuzione di Unittest*

Dopo il rilascio dell'immagine della nuova versione, è essenziale condurre una verifica preliminare della funzionalità del servizio in un ambiente locale. Questa fase di test preliminare è fondamentale per garantire che le modifiche introdotte non abbiano causato regressioni o difetti critici.

Nell'ambito di questi test, si utilizzano i *Mock* per isolare l'analisi esclusivamente sul servizio in esame. L'uso di *Mocks* è fondamentale per replicare le interazioni provenienti da altri servizi senza fare affidamento sugli ambienti di produzione reali.

La pipeline di test è stata implementata per il servizio di predizione epilettogena e per il servizio di accesso verso i dati persistenti. Di seguito è mostrata la parte del file YAML relativa all'Unit Test di quest'ultimo servizio:

Creating CI/CD Pipeline

```yaml
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
```

Il segmento di codice Python invocato dalla Pipeline per l'esecuzione del test unitario è mostrato di seguito. In questo contesto, il ruolo di *Mock* è quello di emulare l'esecuzione della query diretta al database persistente all'interno del cluster:

Mock test in Python for "db_connection" service

```python
@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. *Creazione di deployments/servizi temporanei di test*

Se i test unitari hanno esito positivo, la fase successiva prevede la creazione di un *Deployment* e *Service* temporaneo all'interno del cluster. Questa fase consente di effettuare test più approfonditi, garantendo una validazione accurata del comportamento del servizio in un ambiente con condizioni operative reali e interagendo con altri servizi già attivi. È quindi essenziale stabilire una connessione SSH al server del nodo Master, avviare la creazione dei nuovi *Deployment* e *Services* di prova e attendere la loro attivazione prima di procedere con la fase di test. Indipendentemente dal risultato di questo test, tali elementi verranno rimossi dal cluster.

La porzione del file YAML per il test del deployment/servizio è mostrata di seguito; utilizza i "segreti" memorizzati su GitHub, compresi il nome host e la porta generati da Ngrok, per stabilire una connessione con la macchina virtuale che ospita il nodo Master:

Creating CI/CD Pipeline

```yaml
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
```

La sezione seguente presenta lo script Bash corrispondente al codice mostrato in precedenza. Questa parte assicura che il *pod* di test sia completamente pronta prima di procedere con il test ufficiale, che viene condotto dal successivo codice

Python.

Bash code for smoke test

```bash
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

```python
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. Aggiornamento di deployments/servizi nell'ambiente di produzione

Se tutti i test sono andati a buon fine, si può procedere con l'aggiornamento dei servizi in produzione. Come primo passo, è necessario aggiornare il puntamento dell'ultima immagine Docker a quella di test:

Creating CI/CD Pipeline

```yaml
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
```

Successivamente, il riavvio del deployment corrispondente consentirà di costruirlo con l'immagine aggiornata:

Creating CI/CD Pipeline

```yaml
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
```

In questa fase, il cluster Kubernetes è completamente aggiornato e la sua funzionalità è garantita dal completamento di tutti i test.

F. Test periodici automatici

L'introduzione di test automatici periodici, eseguiti quotidianamente, complementa i test eseguiti post-build per garantire una valutazione continua del sistema. I test periodici, quindi, sono necessari per monitorare la stabilità e le prestazioni del sistema nel tempo; essi assicurano che eventuali regressioni nelle performance o anomalie nel comportamento del sistema vengano rilevate e affrontate tempestivamente.

Di seguito è riportato il file YAML per la creazione di test periodici automatici e il relativo codice Bash:

Creating automatic daily test

```yaml
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. APP MOBILI PER LA REGISTRAZIONE DI DATI SANITARI

A. Simulazione di dati EEG rea-time

Negli ultimi anni sono state sviluppate tecnologie avanzate per la gestione dei dati elettroencefalografici, che prevedono l'impianto di elettrodi persistenti nel cranio umano. Questi elettrodi, alimentati da energia elettrica, registrano i dati cerebrali e li trasmettono in tempo reale a smartphone o direttamente a piattaforme cloud. Questo sistema consente un monitoraggio continuo e immediato di potenziali problemi epilettici, migliorando significativamente la diagnosi e la gestione dei disturbi neurologici.

Per testare il cluster sviluppato, è stata creata un'applicazione utilizzando React Native. Questa app consente la trasmissione di dati elettroencefalografici precedentemente registrati (ottenuti da esami EEG diagnostici) al servizio endpoint del cluster. Per verificare la corretta ricezione ed elaborazione dei dati, possiamo analizzare il *Service* del cluster che permette la visualizzazione dei dati in tempo reale nel browser.

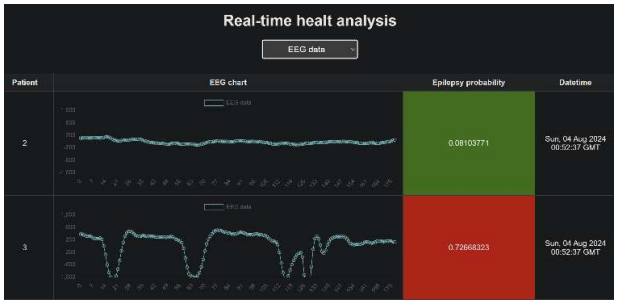

Fig. 4. Servizio che espone i dati EEG ricevuti e la loro elaborazione

Fig. 5. Applicazione utilizzata per testare la trasmissione dei dati EEG

B. Registrazione del battito cardiaco

Per i test relativi alla registrazione, alla condivisione e all'elaborazione dei dati sulla frequenza cardiaca, è stata utilizzata un'applicazione iOS disponibile sugli iPhone. Questa applicazione consente di recuperare in tempo reale i dati registrati dall'Apple Watch e li invia tramite API REST al servizio endpoint del cluster sviluppato. Ciò consente al personale sanitario di monitorare e identificare potenziali problemi cardiaci. Come per i dati EEG, per verificare l'accuratezza di questa registrazione è possibile monitorare gli andamenti attraverso il servizio dedicato nella pagina web.

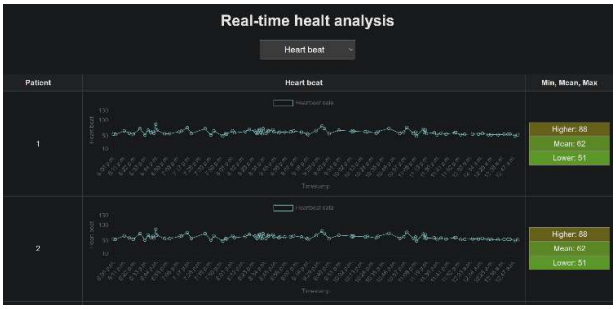

Fig. 6. Servizio cluster che espone i dati del battito cardiaco ricevuti e la loro elaborazione

VIII. OSSERVAZIONI CONCLUSIVE

Il progetto sviluppato è un prototipo di cluster Kubernetes progettato per gestire informazioni sanitarie in tempo reale. Questo sistema è concepito per ricevere, elaborare e analizzare immediatamente i dati sanitari critici, permettendo agli operatori di monitorare costantemente le condizioni dei pazienti e identificare potenziali problemi di salute.

La rilevanza di questo sistema risiede nella sua capacità di fornire un monitoraggio continuo e preciso, migliorando così la qualità e la tempestività delle cure mediche. In un contesto sanitario dove ogni secondo è cruciale, la reattività in caso di emergenza può fare la differenza. Inoltre, l'analisi predittiva dei dati in tempo reale consente di anticipare complicazioni e ottimizzare il percorso terapeutico per ogni paziente.

La scelta di Kubernetes come piattaforma per questo progetto è essenziale. Kubernetes offre scalabilità per gestire grandi volumi di dati, insieme a affidabilità e resilienza necessarie in un ambiente critico come quello sanitario. Automatizzando la distribuzione, la gestione e il monitoraggio delle applicazioni, Kubernetes garantisce che il sistema rimanga operativo e performante anche in condizioni di elevato utilizzo o guasti hardware.

In sintesi, il prototipo sviluppato ha lo scopo di emulare l'integrazione di tecnologie avanzate nel settore sanitario. Mira a migliorare il monitoraggio dei pazienti e la qualità delle cure, aprendo la strada a future innovazioni che promuovono un'assistenza sanitaria più reattiva e personalizzata.

IX. RIFERIMENTI

[1] Reali, Gianluca. Virtual Network & Cloud Computing, Università degli Studi di Perugia.

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

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

[4] Ngrok: Tunnel sicuri e ispezionabili verso localhost. Disponibile https://ngrok.com