SBDIO I4.0
Big Data
for Industry

O1.3

Rapporto di validazione del laboratorio dimostrativo federato di SBDIOI40: piattaforma cloud federata SBDIOI40 e algoritmi di advanced analytics SBDIOI40 sui data set in e out

T3LAB

Code	O1.3
Data	18-06-2021
Type	Confidential
Participants	T3LAB
Authors	Luca Padovan
Corresponding authors	Mirko Falavigna

POR FESR EMILIA-ROMAGNA 2014/2020
FSC Fondo per lo Sviluppo e la Coesione
UNIONE EUROPEA Fondo europeo di sviluppo regionale
RegioneEmilia-Romagna

SBDIO I4.0
Big Data
for Industry

Sommario

1 Obiettivi

Gli obiettivi di questa parte del progetto sono i seguenti:

- Realizzare, a partire dalle librerie Python sviluppate da UniMoRe per il Now Casting e l'Anomaly Detection del comportamento di macchinari industriali, delle applicazioni web che consentano ad un utente senza conoscenze di programmazione o scripting di utilizzarle in completa autonomia;
- Containerizzare queste applicazioni web per effettuarne il deploy su un cluster Kubernetes realizzato utilizzando macchine virtuali ospitate da OpenStack;
- Testare il funzionamento delle applicazioni realizzate.

Tutto il progetto è basato sull'utilizzo di OpenStack per la realizzazione del cloud federato di progetto (in particolare i nodi T3LAB e UniBo) e sull'uso di Kubernetes per il deploy e l'orchestrazione delle applicazioni containerizzate sviluppate dagli altri partner del progetto.

2 Cloud OpenStack T3LAB

Il cloud OpenStack di T3LAB è stato realizzato con l'utilizzo di PackStack. Il funzionamento di PackStack, le sue potenzialità e le sue criticità sono state ampiamente trattate in [2] e non verranno ulteriormente approfondite in questo documento.
La struttura del cloud di T3LAB ha subito però dei cambiamenti da quella presentata in [2]. In Figura 1 è illustrata la sua nuova struttura.

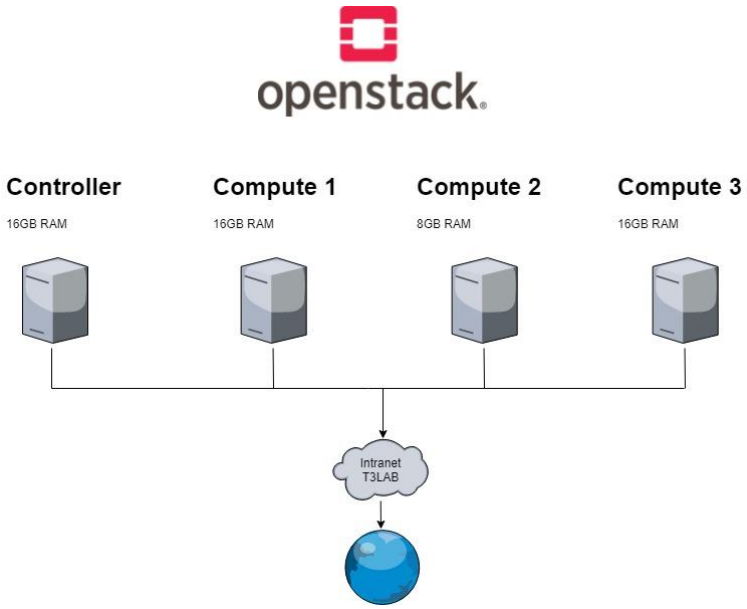

Figura 1 Struttura del cloud OpenStack federato T3LAB

Il cloud è ora composto da 4 nodi fisici, uno che svolge la funzione di Controller (ma anche di Compute) e tre nodi di calcolo. Inoltre i tre nodi di calcolo hanno anche caratteristiche hardware diverse tra di loro: il nodo Compute 3 ha a disposizione una CPU molto più recente rispetto alle altre, oltre che un SSD che ci consente di svolgere in maniera più performante determinate operazioni, soprattutto quelle che hanno a che fare con le tecniche di Machine Learning o Deep Learning con le quali ci siamo dovuti confrontare in questa parte del progetto. Queste tecnologie utilizzano dei framework che non avremmo potuto utilizzare su macchine come Compute 1 o Compute 2, che utilizzano hardware non particolarmente recenti.
A causa di questo, abbiamo dovuto assicurarci che le applicazioni che abbiamo implementato per rendere fruibili le librerie di UniMoRe fossero effettivamente deployate ed eseguite sul nodo Compute 3, che aveva i requisiti necessari richiesti dai framework utilizzati da UniMore e non su altri nodi di calcolo dove non avrebbero funzionato correttamente. Questa situazione peraltro è abbastanza verosimile, dal momento che spesso i cloud sono composti da macchine eterogenee tra loro: non tutte le macchine hanno le stesse caratteristiche hardware o le stesse potenzialità, e spesso le applicazioni richiedono determinati requisiti hardware per poter funzionare correttamente, costringendo quindi a effettuarne il deploy su una macchina specifica (o su un insieme di macchine) escludendone altre. Il modo in cui abbiamo implementato questo vincolo verrà trattato più avanti nel documento.

3 Kubernetes

L'utilizzo di Kubernetes nel contesto del progetto SBDIO I4.0 è stato studiato inizialmente da CiriICT. In base alle indicazioni di CiriICT T3LAB si è quindi occupato di fare evolvere la propria infrastruttura cloud affinché essa potesse supportare anche cluster Kubernetes.
Questo capitolo è basato su [3]. Kubernetes è un progetto sviluppato inizialmente da Google e reso open-source nel 2014. Lo scopo principale di Kubernetes è quello di facilitare sia la configurazione dichiarativa che l'automazione di applicazioni containerizzate. I principali punti di forza di Kubernetes sono dati dall'utilizzo dei container per il deploy delle applicazioni:

- i container presentano uno modello di isolamento più leggero di quello delle macchine virtuali, garantendo comunque la non interferenza tra container differenti e quindi tra le applicazioni;
- coerenza di ambiente tra sviluppo, test e produzione: i container funzionano allo stesso modo sia su un computer portatile che sulle macchine di un cloud;
- portabilità tra sistemi cloud e sistemi operativi differenti;
- microservizi liberamente combinabili, distribuiti e ad alta scalabilità: le applicazioni sono suddivise in componenti più piccoli e indipendenti tra loro, che possono essere distribuiti e gestiti dinamicamente;
- in un container le risorse sono isolate dagli altri container, ciò consente di prevedere le prestazioni delle applicazioni.

Kubernetes quindi ci permette di creare dei cluster (composti da macchine fisiche e/o virtuali, nel nostro caso macchine virtuali in un ambiente cloud OpenStack) sui quali possiamo effettuare il deploy di applicazioni containerizzate. Inoltre Kubernetes non si limita a gestire singoli container di

applicazioni, infatti è un vero e proprio orchestratore di container: si occupa della scalabilità, del failover e della distribuzione dei container che formano le nostre applicazioni.
Le principali funzionalità di Kubernetes sono:

- **Scoperta dei servizi e bilanciamento del carico**: è possibile esporre un container (o un gruppo di container) verso l'esterno usando un nome DNS o un semplice indirizzo IP. Kubernetes è in grado monitorare il traffico verso un determinato container e reindirizzarlo verso altri in modo che il servizio rimanga disponibile e stabile;
- **Rollout e rollback automatici**: possiamo descrivere quale vogliamo che sia lo stato finale desiderato per i nostri container e Kubernetes si occuperà di effettuare le operazioni necessarie per portarli allo stato che abbiamo indicato. Possiamo dire a Kubernetes di creare nuovi container per un nuovo servizio, di aggiornare le immagini di tutti i container di una determinata applicazione e di effettuare un rollback nel caso i cambiamenti non siano andati a buon fine;
- **Ottimizzazione dei carichi**: se forniamo un cluster di nodi sul quale è possibile eseguire i container e indichiamo di quanta CPU e RAM gli stessi hanno bisogno, allora Kubernetes allocherà i container sui nodi in modo tale da ottimizzare l'uso delle risorse;

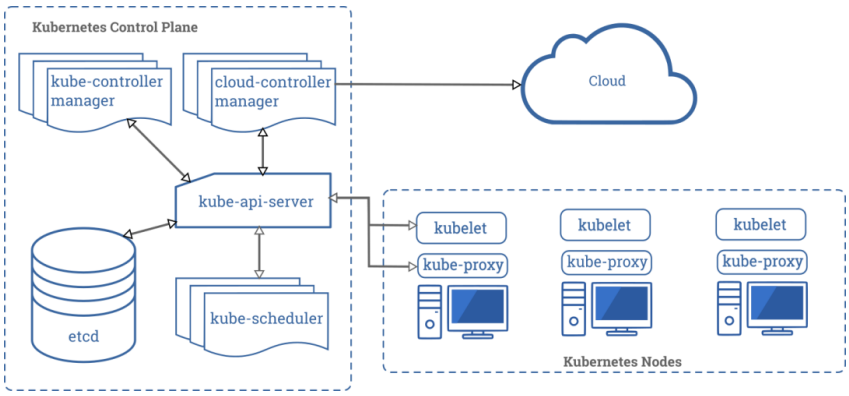

Figura 2 Diagramma di un cluster Kubernetes e delle interazioni dei suoi componenti

- **Self-healing**: Kubernetes elimina i container che si bloccano e li sostituisce, elimina i container che non rispondono agli health checks e evita di far arrivare traffico ai container che non sono pronti per gestire richieste.

3.1 I componenti di Kubernetes

Una volta fatto il deploy di Kubernetes, otteniamo un cluster. Un cluster Kubernetes non è altro che un insieme di macchine (fisiche e/o virtuali) che chiamiamo nodi, che eseguono container gestiti e coordinati da Kubernetes. Ogni cluster deve avere almeno un Worker Node e un nodo (che prende il nome di Master Node) che ospita il Control Plane che si occuperà di coordinare le operazioni del cluster. I Worker Node ospitano i Pod che eseguono i Workload dell'utente (vedi sez. 3.2). Il Control Plane gestisce i Worker Node. In Figura 2 viene riportato un diagramma dei componenti di un cluster Kubernetes e di come interagiscono tra di loro.

3.1.1 Componenti del Control Plane

I componenti del Control Plane sono responsabili delle decisioni che riguardano tutto il cluster:

- **kube-api-server**: espone le Kubernetes APIs. È il frontend del Control Plane di Kubernetes ed è completamente scalabile orizzontalmente (è possibile cioè eseguire più istanze e bilanciare il carico tra esse);
- **etcd**: è un database key-value usato da Kubernetes per salvare tutte le informazioni del cluster;
- **kube-scheduler**: si occupa di assegnare i Pod appena creati ad un nodo del cluster;
- **kube-controller-manager**: si occupa della gestione dei controller. I controller si occupano di supervisionare il cluster e il suo stato. Esistono diversi tipi di controller per diversi scopi, tra cui i più importanti sono il Node Controller che monitora lo stato di salute dei nodi del cluster e il Replication Controller, responsabile per il mantenimento dei corretto numero di Pod per ogni ReplicaSet (il numero di container che vogliamo per ogni componente dell'applicazione) presente nel sistema;
- **cloud-controller-manager**: permette di collegare il cluster alle API del cloud provider e separare quindi le componenti che interagiscono con la piattaforma cloud dai componenti che interagiscono solamente con il cluster.

3.1.2 Componenti dei nodi

I componenti dei nodi vengono eseguiti su ogni nodo del cluster, mantenendo i Pod in esecuzione e fornendo l'ambiente di runtime a Kubernetes:

- **kubelet:** un agent eseguito su ogni nodo del cluster. Si assicura che tutti i container siano eseguiti su un Pod, funzionino correttamente e siano sani;
- **kube-proxy**: gestisce un servizio di proxy su ogni modo che interagisce con le singole sotto reti della macchina host ed espone i servizi al mondo esterno. Esegue anche l'inoltro delle richieste ai Pod destinatari situati nei vari nodi del cluster;
- **Container Runtime**: software responsabile per l'esecuzione dei container. Kubernetes supporta diversi container runtime, anche se quello maggiormente utilizzato è sicuramente Docker (che è quello che è stato utilizzato nel progetto).

3.2 Concetti chiave di Kubernetes

Per poter utilizzare Kubernetes, è necessario comprendere alcuni concetti e astrazioni di base che vengono utilizzati per rappresentare lo stato del sistema e del nostro cluster. Di seguito vengono elencate le più importanti per i nostri scopi.

3.2.1 Pod

I nodi worker del cluster eseguono i Pod. I Pod sono i componenti più semplici e basilari di Kubernetes. Ogni Pod rappresenta una singola istanza di un'applicazione o di un processo in esecuzione su Kubernetes e consiste di uno (solitamente) o più container. Kubernetes si occupa di eseguire, interrompere o replicare tutti i container in un Pod trattandoli come un gruppo.
I Pod situati nei Worker Node vengono dunque creati e distrutti a seconda dello stato desiderato dall'utente, e rappresentano anche l'unità di scaling di Kubernetes: se dovesse servire scalare

orizzontalmente un'applicazione si aggiungono nuovi Pod sui quali verranno eseguiti nuovi container.

3.2.2 Service

Come detto precedentemente, i Pod sono unità effimere e volatili, e quindi non affidabili. Un Pod potrebbe per esempio essere smantellato in qualsiasi momento secondo politiche sia interne che esterne a Kubernetes. Se un Pod dovesse interrompersi, Kubernetes non si occuperà di riportarlo in vita: semplicemente lo distruggerà e ne creerà uno nuovo. Il nuovo Pod potrebbe anche sembrare lo stesso del precedente, ma non è così: avrà infatti un nuovo indirizzo IP ed uno stato differente. Per far fronte a questo problema, Kubernetes introduce il concetto di Service, il cui scopo è quello di fornire (a livello di networking) un punto di accesso stabile ad un insieme volatile di Pod che forniscono uno stesso servizio.
I Service quindi forniscono un frontend stabile costituito da un nome DNS o da un IP ed una porta, bilanciando il carico tra diversi Pod. Inoltre monitorano lo stato di salute di ogni singolo Pod che si ritrovano a gestire, in modo tale da non indirizzare richieste ad un Pod non in grado di servirle.

3.2.3 Deployment

Una delle caratteristiche più importanti che un'applicazione cloud nativa deve garantire è la resilienza.
I Pod, di per sé, non sono in grado di gestire eventuali malfunzionamenti, non sono in grado di scalare autonomamente e non forniscono nessun meccanismo di gestione di aggiornamenti e rollback delle immagini dei container. In Kubernetes questo ruolo viene ricoperto dai Deployment.
Di fatto i Pod vengono messi in esecuzione sempre attraverso dei Deployment, ognuno dei quali è responsabile della gestione di un singolo tipo di Pod (si pensi ad un'applicazione web con backend e frontend, dove dovranno essere utilizzati due tipi di Pod diversi).
A livello pratico, un Deployment descrive lo stato desiderato di un insieme di Pod, basandosi su un file YAML [4], e si assicura che lo stato descritto dal file venga raggiunto da parte dei Pod. Un Deployment per esempio fornisce la descrizione di come è strutturata un'applicazione a livello di container, specificando le regole di strutturazione della stessa e la politica di replicazione dei Pod.

4 Modelli di interazione e ruoli dei laboratori

L'utilizzo di una infrastruttura basata su container ha portato a rivedere le politiche di deployment che erano state definite in sez. 2 di [1].
Il modello precedentemente utilizzato può essere così riassunto:

- T3LAB in quanto gestore di un nodo cloud basato su un insieme di risorse fisiche (T3LAB-IaaS) definisce la configurazione funzionale del nodo (quali servizi di OpenStack devono essere installati su quali macchine fisiche) e installa congruentemente OpenStack su tutte le macchine fisiche coinvolte;
- T3LAB-IaaS in quanto gestore di un nodo cloud crea un tenant al quale è allocato un pool di risorse virtuali che rappresenteranno la base di definizione della sua infrastruttura (ovviamente il nodo cloud potrà ospitare più tenant). Il tenant è creato sulla base di una richiesta di T3LAB-PaaS, che vuole supportare una piattaforma cloud per lo sviluppatore di applicazioni;

- Sulla base delle necessità dello sviluppatore dell'applicazione, T3LAB-PaaS compie due operazioni:
  - definisce nel contesto del tenant una infrastruttura virtuale di computazione (un insieme di macchine e LAN virtuali) compatibile con il pool di risorse allocato al tenant stesso e adeguato per il supporto dell'applicazione finale;
  - sull'infrastruttura così definita installa e configura la piattaforma middleware (emulando una situazione PaaS) sulla quale verrà poi realizzata l'applicazione finale;
- lo sviluppatore, sulla piattaforma middleware, crea e rende disponibile l'applicazione finale (emulando una situazione SaaS).

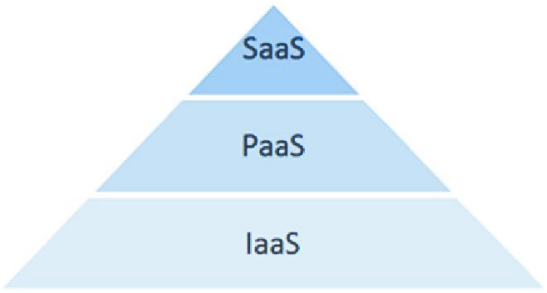

Figura 3 Stack dei possibili servizi cloud

Il modello di interazione utilizzato durante la prima parte del progetto, in cui si sono seguiti tutti i possibili livelli di servizio cloud, non è risultato però conveniente nel momento in cui l'offerta del servizio SaaS è basata su un cluster Kubernetes, anche nel momento in cui Kubernetes stesso è inserito in un contesto cloud/OpenStack.
Gli ultimi due punti del modello di interazione descritto in precedenza sono quindi stati modificati come segue:

- l'utente che richiede i servizi cloud sviluppa l'applicazione o la libreria finale utilizzando la opportuna piattaforma middleware;
- T3LAB-IaaS offre all'utente il servizio di containerizzazione e deployment della sua applicazione su un cluster Kubernetes che si appoggia alle risorse computazionali offerte dal nodo cloud T3LAB. A questo scopo T3LAB-IaaS:
  - definisce nel contesto del tenant una infrastruttura virtuale di computazione (un insieme di macchine e LAN virtuali) compatibile con il pool di risorse allocato al tenant e adeguato per il supporto dell'applicazione finale;
  - installa su questa infrastruttura virtuale di computazione un cluster Kubernetes;
  - effettua la containerizzazione docker dell'applicazione finale
  - effettua il deployment dell'applicazione finale containerizzata sul cluster Kubernetes

Nel nuovo modello di deployment basato su Kubernetes T3LAB non gioca quindi più il ruolo di fornitore di servizi PaaS ma viene a potenziare il suo ruolo di fornitore di sevizi IaaS e assume il ruolo di fornitore dei servizi di containerizzazione e deployment su Kubernetes dell'applicazione finale.

5 Containerizzazione e deployment di applicazioni su Kubernetes

5.1 Struttura del cluster Kubernetes configurato

La struttura del cluster Kubernetes destinato a supportare l'applicazione le applicazioni che abbiamo sviluppato è costituita da 3 macchine, nel nostro caso 3 macchine virtuali OpenStack, così come illustrato nella Figura 4.
Il deployment di Kubernets, essendo basato sull'utilizzo di VM in una infrastruttura cloud, prevede ovviamente che questa infrastruttura virtuale sia stata precedentemente creata all'interno del tenant OpenStack.
Questa operazione è assolutamente analoga a quanto fatto nella fase 1 del progetto [1] ma in questo caso la creazione dell'infrastruttura è stata effettuata interattivamente tramite la dashboard di OpenStack (servizio Horizon).
Questa volta inoltre abbiamo deciso di sfruttare un'ulteriore funzione offerta da OpenStack, e cioè la possibilità di creare dei cosiddetti *Host Aggregates*. Questa funzione ci consente di creare degli insiemi di nodi fisici sui quali possiamo scegliere di deployare delle macchine virtuali. Abbiamo quindi creato un Host Aggregate che comprendesse soltanto il Compute 3 del nostro cloud, per poter istanziare una macchina virtuale obbligatoriamente su questo nodo, in modo tale che una volta entrata a far parte del cluster Kubernetes potesse eseguire correttamente i pod che avrebbero ospitato le applicazioni che richiedessero una CPU con funzionalità più moderne o un SSD.

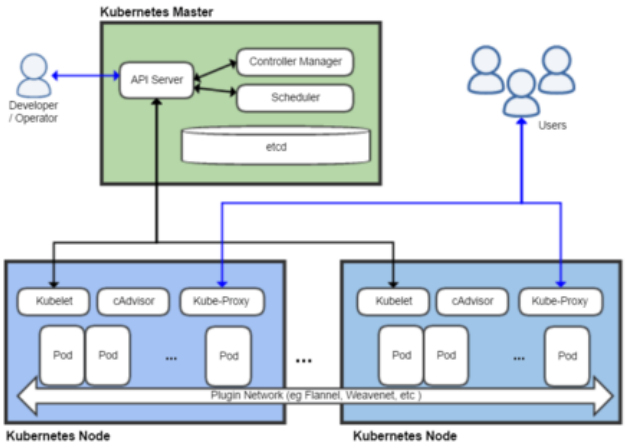

Figura 4 Stack Cluster Kubernetes di T3LAB

5.2 Sviluppo e containerizzazione dell'applicazione PyNowCast di UniMoRe

5.2.1 Sviluppo dell'applicazione PyNowCast

La libreria per in Now Casting sviluppata da UniMoRe ha come obiettivo l'utilizzo di tecniche di Machine Learning e Deep Learning per stabilire le condizioni meteo a partire da una foto, utilizzando

come libreria PyTorch. Il nostro compito è stato quello di realizzare, a partire da questa libreria, un'applicazione web che consentisse ad un utente di utilizzarla in modo agevole e senza conoscenze pregresse in materia.
Abbiamo quindi realizzato un backend Express con NodeJS per esporre delle API che un frontend realizzato in ReactJS potesse chiamare per eseguire un'inferenza sul modello già addestrato. La scelta di non consentire l'addestramento del modello tramite l'applicazione web è stata dettata principalmente dal fatto che non avevamo a disposizione dei dataset per poter effettuare il training.
In Figura 5 e in Figura 6 sono riportati degli screenshot dell'applicazione finale. L'applicazione è molto semplice, l'utente non deve far altro che selezionare un'immagine che vuole classificare e premere il tasto per avviare la classificazione.
In futuro sarebbe interessante aggiungere anche la parte di training del modello, a condizione che sia possibile ottenere un dataset abbastanza grande.

5.2.2 Strumenti per la containerizzazione

Per la containerizzazione delle due applicazioni realizzate (backend e frontend) abbiamo scritto un Dockerfile che abbiamo successivamente buildato tramite la CLI di Docker. Una volta buildata l'immagine, abbiamo provveduto a caricarla su DockerHub, in modo che Kubernetes potesse scaricarla senza problemi durante il deploy dell'applicazione finale.

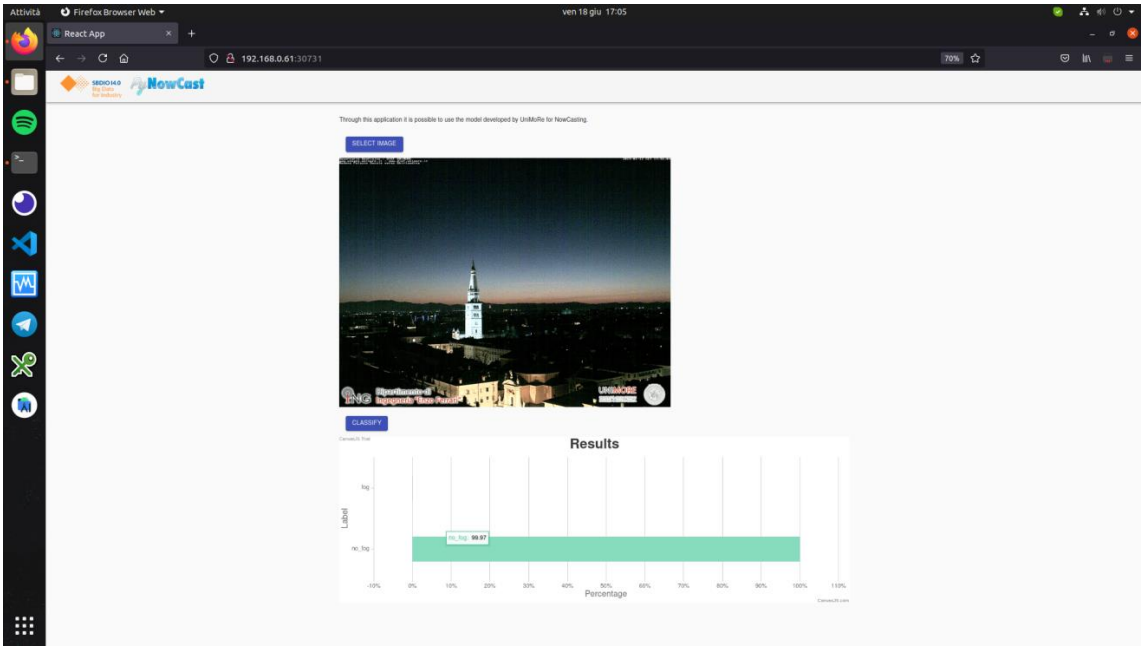

Figura 5 Screenshot dell'applicazione PyNowCast realizzata da T3LAB

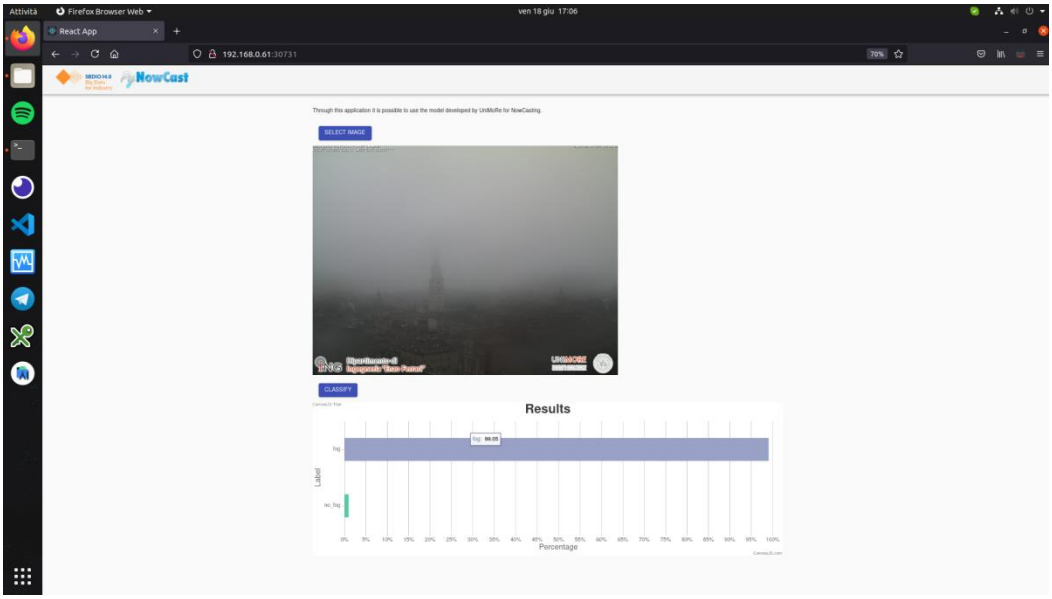

Figura 6 Screenshot dell'applicazione PyNowCast realizzata da T3LAB

5.2.3 Containerizzazione e deploy di PyNowCast

Come abbiamo detto precedentemente, per containerizzare le due componenti che costituiscono l'applicazione PyNowCast abbiamo sviluppato dei Dockerfile che abbiamo successivamente buildato con la CLI di Docker. In Figura 7 vengono riportati i sorgenti dei due Dockerfile realizzati.

```
# Base Image
FROM python:3.8

# create and set working directory
RUN mkdir /app
WORKDIR /app

# Add current directory code to working directory
ADD sbdio_pynowcast_backend /app/

# set default environment variables
ENV PYTHONUNBUFFERED 1
ENV LANG C.UTF-8
ENV DEBIAN_FRONTEND=noninteractive

# set project environment variables
# grab these via Python's os.environ
# these are 100% optional here
ENV PORT=8000

# Install system dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
        tzdata \
        python3-setuptools \
        python3-pip \
        python3-dev \
        python3-venv \
        git \
        nodejs \
        npm \
        && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*


# install environment dependencies
RUN pip3 install --upgrade pip

RUN snap install node --classic --channel=14 && apt install -y npm

# Install project dependencies
RUN pip3 install -r PyNowCast/requirements.txt
RUN pip3 install requests


EXPOSE 8000

RUN cd /app/ && npm install

CMD npm start
```

```
# Stage 1, "builder", based on Node.js, to build and compile the frontend
FROM node:10-alpine as builder
# copy the package.json to install dependencies
COPY package.json package-lock.json ./
# Install the dependencies and make the folder
RUN npm install && mkdir /sbdioi4.0 && mv ./node_modules ./sbdioi4.0
WORKDIR /sbdioi4.0
COPY . .
# Build the project and copy the files
RUN npm run build

# Stage 2 based on Nginx, to have only the compiled app, ready for production with Nginx
FROM nginx:alpine
#!/bin/sh
COPY ./nginx.conf /etc/nginx/nginx.conf
## Remove default nginx index page
RUN rm -rf /usr/share/nginx/html/*
# Copy from the stahg 1
COPY --from=builder /sbdioi4.0/build /usr/share/nginx/html
# Copy .env file and shell script to container
WORKDIR /usr/share/nginx/html
COPY ./env.sh .
RUN sed -i -e 's/\r$//' env.sh
COPY .env .

# Add bash
RUN apk add --no-cache bash
# Make our shell script executable
RUN chmod +x env.sh

EXPOSE 3000 80

# Start Nginx server
CMD ["/bin/bash", "-c", "/usr/share/nginx/html/env.sh && nginx -g \"daemon off;\""]
```

Figura 7 A sinistra, Dockerfile per la containerizzazione del backend, a destra Dockerfile per la containerizzazione del frontend

Una volta realizzate e caricate su DockerHub le immagini docker di frontend e backend, abbiamo scritto due file YAML da utilizzare per effettuarne il deploy su Kubernetes. Successivamente, tramite script bash abbiamo eseguito il deploy vero e proprio delle applicazioni. Questi script non fanno altro che effettuare richiamare la CLI del cluster Kubernetes, kubectl, utilizzando i file YAML precedentemente scritti. In Figura 8 sono riportati i file YAML creati. Per queste applicazioni non c'era nessun vincolo hardware sulla macchina che le avrebbe ospitate, nonostante questo abbiamo deciso di obbligare Kubernetes a deployare il backend sul nodo worker hostato da Compute 3, attraverso un meccanismo di Kubernetes che prende il nome di *Node Affinity*. Questo meccanismo ci consente di forzare Kubernetes ad effettuare il deployment di un pod su un determinato worker node, basandoci su un meccanismo di labeling: ai worker node è possibile associare una o più label. Quando effettuiamo il deploy di un pod, possiamo aggiungere al file YAML del deployment che vogliamo effettuare la proprietà *requiredDuringSchedulingIgnoredDuringExecution*, seguita da una o più label. Kubernetes quindi effettuerà il deploy del pod soltanto sul nodo (o sui nodi) worker che possiedono le label da noi specificate.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sbdio-pynowcast-backend
  namespace: pynowcastspace
  labels:
    app: sbdio-pynowcast-backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: sbdio-pynowcast-backend
  template:
    metadata:
      labels:
        app: sbdio-pynowcast-backend
    spec:
      containers:
      - name: sbdio-pynowcast-backend
        image: lucapadovan/pynowcast_backend:1.0
        ports:
        - containerPort: 8000
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: hd
                operator: In
                values:
                - ssd
---
apiVersion: v1
kind: Service
metadata:
  name: sbdio-pynowcast-service
  namespace: pynowcastspace
  labels:
    run: sbdio-pynowcast-service
spec:
  type: NodePort
  ports:
  - port: 8000
    protocol: TCP
  selector:
    app: sbdio-pynowcast-backend
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sbdio-pynowcast-frontend
  namespace: pynowcastspace
  labels:
    app: sbdio-pynowcast-frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sbdio-pynowcast-frontend
  template:
    metadata:
      labels:
        app: sbdio-pynowcast-frontend
    spec:
      containers:
      - name: sbdio-pynowcast-frontend
        image: lucapadovan/pynowcast_frontend:1.0
        env:
        - name: API_URL
          value: {{API_URL_PROTOCOL}}://{{API_URL_HOST}}:{{API_URL_PORT}}
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: sbdio-pynowcast-frontend-service
  namespace: pynowcastspace
  labels:
    run: sbdio-pynowcast-frontend-service
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: sbdio-pynowcast-frontend
```

Figura 8 A sinistra file YAML per il deploy del backend, a destra file YAML per il deploy del frontend dell'applicazione PyNowCast

5.3 Sviluppo e containerizzazione dell'applicazione Anomaly Detection di UniMoRe

5.3.1 Sviluppo dell'applicazione Anomaly Detection

La libreria per l'Anomaly Detection sviluppata da UniMoRe ha come obiettivo l'utilizzo di tecniche di Machine Learning e Deep Learning per stabilire se un macchinario industriale sta operando in uno stato anomalo rispetto a quelli che un determinato modello conosce come normali. La libreria utilizzata da UniMore utilizza TensorFlow, richiedendo quindi che la CPU sul quale viene eseguita supporti il set di istruzioni AVX o AVX2. Il nostro compito è stato quello di realizzare, a partire da questa libreria, un'applicazione web che consentisse ad un utente di utilizzarla in modo agevole e senza conoscenze pregresse in materia.

Abbiamo quindi realizzato un backend Express con NodeJS per esporre delle API che un frontend realizzato in ReactJS potesse chiamare per eseguire un'inferenza su un modello che viene addestrato di volta in volta tramite dei file CSV.

In Figura 9 e in Figura 10 è possibile vedere alcuni screenshot dell'applicazione al termine dell'inferenza. L'applicazione è molto semplice, l'utente non deve far altro che selezionare uno o più file CSV per istruire il modello su quali siano gli stati normali di funzionamento, un file CSV che rappresenta lo stato da testare per verificare la presenza di eventuali anomalie e avviare l'inferenza sul modello selezionato. Al termine dell'esecuzione, l'applicazione visualizza un grafico nel quale eventuali stati anomali del macchinario che stiamo analizzando sono evidenziati in arancione chiaro.

Figura 9 Screenshot dell'applicazione Anomaly Detection realizzata da T3LAB

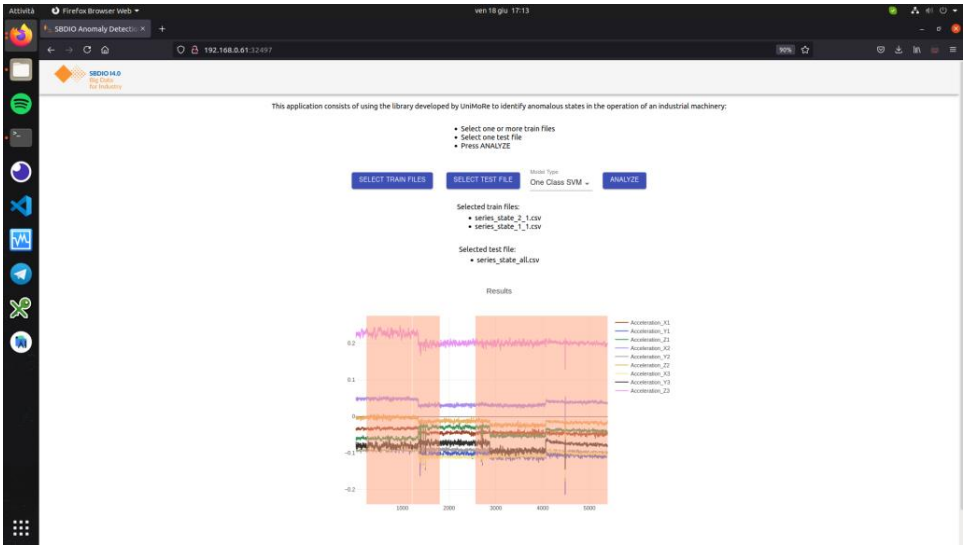

Figura 10 Screenshot dell'applicazione Anomaly Detection realizzata da T3LAB

5.3.2 Strumenti per la containerizzazione

Per la containerizzazione delle due applicazioni realizzate (backend e frontend) abbiamo seguito lo stesso procedimento illustrato in 5.2.2. Abbiamo quindi scritto due Dockerfile che abbiamo successivamente buildato tramite la CLI di Docker. Una volta buildate le immagine, abbiamo provveduto a caricarle su DockerHub.

5.3.3 Containerizzazione e deploy di Anomaly Detection

Come abbiamo detto precedentemente, per containerizzare le due componenti che costituiscono l'applicazione Anomaly Detection abbiamo scritto dei Dockerfile che abbiamo successivamente buildato con la CLI di Docker. In Figura 11 vengono riportati i sorgenti dei due Dockerfile realizzati.

```
# Base Image
FROM python:3.8-slim

# create and set working directory
RUN mkdir /app
WORKDIR /app

# Add current directory code to working directory
ADD sbdio_anomaly_detection_backend /app/

# set default environment variables
ENV PYTHONUNBUFFERED 1
ENV LANG C.UTF-8
# ENV DEBIAN_FRONTEND=noninteractive

# Install npm and node, then the python and nodejs dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    npm \
    nodejs && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/* && \
    pip3 install --upgrade pip && \
    pip3 install -r /app/SBDIO/requirements.txt && \
    pip3 install requests && \
    cd /app/ && npm install

# set project environment variables
# grab these via Python's os.environ
# these are 100% optional here
ENV PORT=8000
EXPOSE 8000

CMD npm start
```

```
# Stage 1, "builder", based on Node.js, to build and compile the frontend
FROM node:10-alpine as builder
# copy the package.json to install dependencies
COPY package.json package-lock.json ./
# Install the dependencies and make the folder
RUN npm install && mkdir /sbdioi4.0 && mv ./node_modules ./sbdioi4.0
WORKDIR /sbdioi4.0
COPY . .
# Build the project and copy the files
RUN npm run build

# Stage 2 based on Nginx, to have only the compiled app, ready for production with Nginx
FROM nginx:alpine
#!/bin/sh
COPY ./nginx.conf /etc/nginx/nginx.conf
## Remove default nginx index page
RUN rm -rf /usr/share/nginx/html/*
# Copy from the stahg 1
COPY --from=builder /sbdioi4.0/build /usr/share/nginx/html
# Copy .env file and shell script to container
WORKDIR /usr/share/nginx/html
COPY ./env.sh .
RUN sed -i -e 's/\r$//' env.sh
COPY .env .

# Add bash
RUN apk add --no-cache bash
# Make our shell script executable
RUN chmod +x env.sh

EXPOSE 3000 80

# Start Nginx server
CMD ["/bin/bash", "-c", "/usr/share/nginx/html/env.sh && nginx -g \"daemon off;\""]
```

Figura 11 A sinistra, Dockerfile per la containerizzazione del backend, a destra Dockerfile per la containerizzazione del frontend

Una volta realizzate e caricate su DockerHub le immagini docker di frontend e backend, abbiamo scritto due file YAML da utilizzare per effettuare il deploy di queste due componenti su Kubernetes. Successivamente, tramite script bash abbiamo eseguito il deploy vero e proprio delle applicazioni. Questi script non fanno altro utilizzare la CLI del cluster Kubernetes, kubectl, utilizzando i file YAML precedentemente scritti. In Figura 12 sono illustrati i file YAML che abbiamo realizzzato. In 5.3.1 abbiamo detto che, per funzionare correttamente, il backend deve essere eseguito su una macchina la cui CPU supporti AVX o AVX2. Per assicurarci questo, siamo ricorsi alle Node Affinity di Kubernetes. Come detto in 5.1, ci siamo assicurati che un worker node (che corrisponde ad una macchina virtuale del cloud OpenStack) del cluster fosse istanziato su Compute 3 la cui CPU supporta per l'appunto AVX e AVX2. A questo punto, tramite il meccanismo della Node Affinity di Kubernetes, abbiamo fatto in modo che i pod che ospitassero il backend fossero ospitati dal worker node istanziato su Compute 3, in modo da rispettare i requisiti hardware di TensorFlow e da consentire così che il backend funzionasse senza problemi.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sbdio-anomaly-backend
  namespace: sbdioanomalyspace
  labels:
    app: sbdio-anomaly-backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: sbdio-anomaly-backend
  template:
    metadata:
      labels:
        app: sbdio-anomaly-backend
    spec:
      containers:
      - name: sbdio-anomaly-backend
        image: mballantit3lab/sbdio_anomaly_backend:1.3
        ports:
        - containerPort: 3000
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: tensorflowsupport
                operator: In
                values:
                - support
---
apiVersion: v1
kind: Service
metadata:
  name: sbdio-anomaly-service
  namespace: sbdioanomalyspace
  labels:
    run: sbdio-anomaly-service
spec:
  type: NodePort
  ports:
  - port: 3000
    protocol: TCP
  selector:
    app: sbdio-anomaly-backend
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sbdio-anomaly-frontend
  namespace: sbdioanomalyspace
  labels:
    app: sbdio-anomaly-frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: sbdio-anomaly-frontend
  template:
    metadata:
      labels:
        app: sbdio-anomaly-frontend
    spec:
      containers:
      - name: sbdio-anomaly-frontend
        image: lucapadovan/anomaly_detection_frontend:1.0
        env:
        - name: API_URL
          value: {{API_URL_PROTOCOL}}://{{API_URL_HOST}}:{{API_URL_PORT}}
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: sbdio-anomaly-frontend-service
  namespace: sbdioanomalyspace
  labels:
    run: sbdio-anomaly-frontend-service
spec:
  type: NodePort
  ports:
  - port: 80
    protocol: TCP
  selector:
    app: sbdio-anomaly-frontend
```

Figura 12 A sinistra file YAML per il deploy del backend, a destra file YAML per il deploy del frontend dell'applicazione Anomaly Detection

6 Script realizzati

Per facilitare la gestione delle applicazioni ospitate nel cluster Kubernetes, abbiamo sviluppato degli script che automatizzassero la procedura di creazione e rimozione delle stesse. Di seguito riportiamo il nome dello script e una breve descrizione, senza riportarne il contenuto.

6.1 PyNowCast

Nome script	Descrizione
startApplication.sh	Prende i file YAML illustrati in 5.2.3 e crea i deployment delle applicazioni relative a PyNowCast nel cluster Kubernetes
stopAndClearApplication.sh	Elimina dal cluster Kubernetes le applicazioni frontend e backend di PyNowCast, ripulendo il cluster da tutti i namespace, i services e i deployment collegati ad esse.

6.2 Anomaly Detection

Nome script	Descrizione
startApplication.sh	Prende i file YAML illustrati in 5.2.3 e crea i deployment delle applicazioni relative a Anomaly Detection nel cluster Kubernetes
stopAndClearApplication.sh	Elimina dal cluster Kubernetes le applicazioni frontend e backend di Anomaly Detection, ripulendo il cluster da tutti i namespace, i services e i deployment collegati ad esse.

7 Bibliografia

[1] T3LAB, "SBDIO I4.0 - O1.1".

[2] T3LAB, "SBDIO I4.0 - O1.2".

[3] «Kubernetes Documentation | Kubernetes» [Online]. Available: https://kubernetes.io/it/docs/home/.

[4] «The Official YAML Web Site » [Online]. Available: https://yaml.org

[5] «Communicate Between Containers in the Same Pod Using a Shared Volume» [Online]. Available: https://kubernetes.io/docs/tasks/access-application-cluster/communicate-containers-same-pod-shared-volume/

[6] https://wiki.openstack.org/wiki/Packstack.

[7] «OpenStack Docs: OpenStack Compute (Nova),» [Online]. Available: https://docs.OpenStack.org/nova/latest/.

[8] «OpenStack Docs: Welcome to Glance's documentation!,» [Online]. Available: https://docs.OpenStack.org/glance/latest/.

[9] «OpenStack Docs: Keystone, the OpenStack Identity Service,» [Online]. Available: https://docs.OpenStack.org/keystone/latest/.

[10] «OpenStack Docs: Horizon: The OpenStack Dashboard Project,» [Online]. Available: https://docs.OpenStack.org/horizon/latest/.

[11] «OpenStack Docs: Welcome to Neutron's documentation!,» [Online]. Available: https://docs.OpenStack.org/neutron/latest/.

[12] «OpenStack Docs: OpenStack Block Storage (Cinder) documentation,» [Online]. Available: https://docs.OpenStack.org/cinder/latest/.

[13] «Active Directory and LDAP Reimagined - JumpCloud,» [Online]. Available: https://jumpcloud.com.

[14] http://www.gazlene.net/demystifying-keystone-federation.html