

Capitolo 5

Implementazione del sistema

5.1 Ambiente di sviluppo

In questo capitolo viene descritta l'implementazione effettiva dell'architettura edge-cloud realizzata, a partire dalla predisposizione delle macchine virtuali e della configurazione della rete, fino alla verifica della connettività e alla preparazione per il deploy dei microservizi.

VM	RAM (MB)	CPU Core	Disk (GB)
ubuntu-cloudcorevm	6000	4	60
ubuntu-edgecorevm	3000	3	60
ubuntu-edgecorevm2	3000	2	25
ubuntu-edgecorevm3	3000	3	25

Tabella 5.1: Allocazione delle risorse nelle macchine virtuali.

La scelta di assegnare più risorse al CloudCore è motivata dal suo ruolo centrale, essendo responsabile della gestione del cluster, della persistenza e della visualizzazione dei dati, mentre gli Edge operano solamente come trasmettitori di dati, richiedendo quindi un carico computazionale inferiore.

Host & hypervisor Tutte le VM sono ospitate su un MSI Thin GF63-12UDX (32 GB RAM) ed eseguite tramite Oracle VirtualBox, con la configurazione della rete in modalità bridge per la comunicazione delle macchine.

Versioni Sul nodo cloud è stato utilizzato MicroK8s v1.32.3 (rev. 8148), l'installazione di KubeEdge è stata gestita mediante `keadm` v1.20.0 su cloud ed edge. Come runtime dei container è stato scelto `containerd`.

5.2 Rete e configurazione delle VM

5.2.1 Modalità Bridge Networking

Tutte le VM sono state impostate in modalità **bridge networking**, che consente di assegnare a ciascuna macchina un indirizzo IP nella stessa subnet dell'host.

È la scelta più semplice per permettere agli Edge di contattare direttamente il CloudCore (10000/tcp) e i servizi applicativi (1883, 3000) senza port-forward o NAT aggiuntivi.

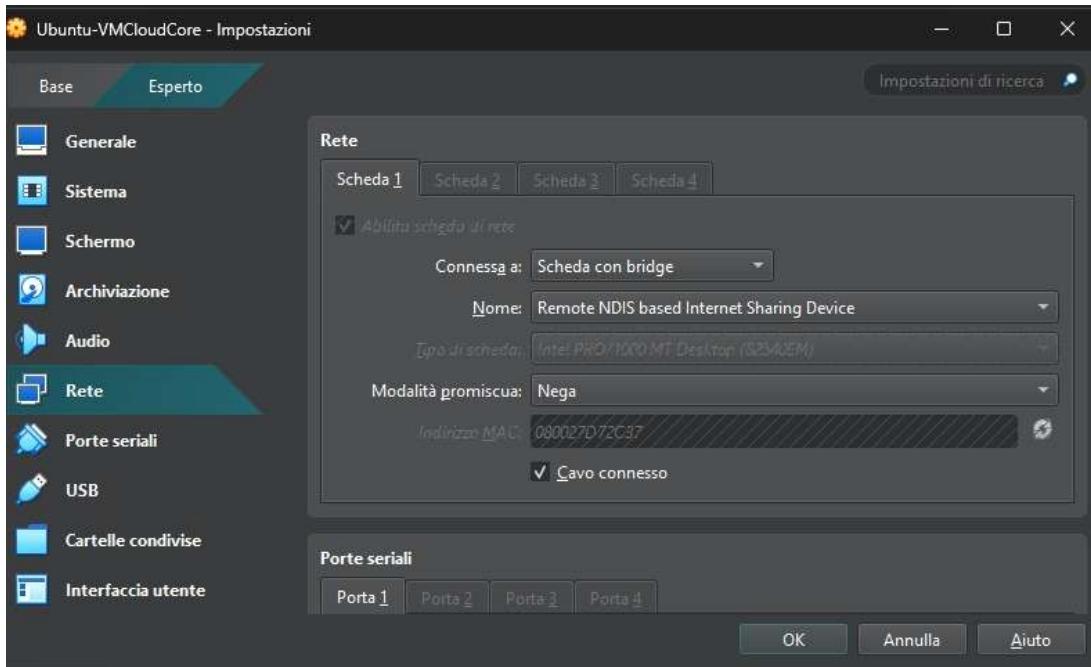


Figura 5.1: Configurazione della rete in modalità bridge su VirtualBox

5.2.2 Connattività e sistema

Ogni VM esegue Ubuntu 22.04 LTS a 64 bit (amd64), dunque è consigliabile aggiornare i pacchetti all'ultima versione ed eventualmente eseguire un riavvio prima di procedere con l'installazione.

```
sudo apt update && sudo apt upgrade -y  
sudo reboot
```

È buona norma verificare anche l'orario/NTP (`timedatectl status`) per evitare problemi con certificati e misure di latenza.

Come prima verifica di connattività tra le macchine si può eseguire il comando ping (`ping <IP_VM>`) e se il test da esito positivo senza perdita di pacchetti le VM riescono a comunicare.

5.2.3 Porte e firewall

Il CloudCore richiede l'apertura di alcune porte, in particolare le 10000, 10002, utilizzate per le comunicazioni principali. Per garantire il corretto funzionamento dei servizi, è necessario verificare che tutte le porte previste siano effettivamente raggiungibili. La disponibilità può essere controllata tramite il seguente comando:

```
sudo ss -lntp | egrep ':*(10000|31000|10002|1883|5432|3000)\b'
```

Vedremo all'interno della configurazione del sistema le motivazioni dell'output riportato in Figura 5.2.

```
lorenzofuse@Ubuntu-VMCloudCore:~$ sudo ss -lntp | egrep ':*(10000|31000|10002|1883|5432|3000)\b'
LISTEN 0      100          0.0.0.0:1883          0.0.0.0:*      users:(("mosquitto",pid=1249,fd=5))
LISTEN 0      200          127.0.0.1:5432          0.0.0.0:*      users:(("postgres",pid=1296,fd=6))
LISTEN 0      4096         *:10000              *:*           users:(("cloudcore",pid=3943,fd=11))
LISTEN 0      4096         *:10002              *:*           users:(("cloudcore",pid=3943,fd=9))
LISTEN 0      100          [::]:1883             [::]:*        users:(("mosquitto",pid=1249,fd=6))
LISTEN 0      4096         *:3000               *:*           users:(("grafana",pid=2289,fd=11))
```

Figura 5.2: Configurazione porte

Osservazioni : MQTT (1883) ascolta su tutte le interfacce; PostgreSQL è intenzionalmente bindato a loopback (127.0.0.1:5432); CloudCore espone 10000 (WebSocket) e 10002.

5.2.4 Raggiungibilità dal lato Edge

Oltre al test precedentemente mostrato con il comando ping, per verificare la corretta comunicazione tra i vari nodi e i relativi servizi, si possono eseguire test che riportano l'indirizzo IP e il numero di porta. L'esecuzione in questo caso viene riportata all'interno della vm ubuntu-edgecorevm2 ma replicabile sulle altre in quanto presentano le medesime configurazioni.

```
nc -vz 192.168.9.128 10000
nc -vz 192.168.9.128 1883
nc -vz 192.168.9.128 5432
nc -vz 192.168.9.128 3000
```

```
lorenzofuse@Ubuntu-EdgeCoreVM2:~$ nc -vz 192.168.9.128 10000
Connection to 192.168.9.128 10000 port [tcp/webmin] succeeded!
lorenzofuse@Ubuntu-EdgeCoreVM2:~$ nc -vz 192.168.9.128 1883
Connection to 192.168.9.128 1883 port [tcp/*] succeeded!
lorenzofuse@Ubuntu-EdgeCoreVM2:~$ nc -vz 192.168.9.128 5432
nc: connect to 192.168.9.128 port 5432 (tcp) failed: Connection refused
lorenzofuse@Ubuntu-EdgeCoreVM2:~$ nc -vz 192.168.9.128 3000
Connection to 192.168.9.128 3000 port [tcp/*] succeeded!
```

Figura 5.3: Output atteso

Il rifiuto su 5432 è coerente con la scelta di non esporre il DB all'esterno e usare un microservizio dedicato posizionato sul CloudCore.

5.3 Configurazione del nodo Cloud

Il nodo cloud rappresenta il piano di controllo dell’architettura implementata, su cui risiede il componente **CloudCore** di KubeEdge, incaricato di orchestrare i nodi periferici e di garantire la gestione delle risorse.

Come evidenziato nel Capitolo 3, KubeEdge non sostituisce Kubernetes, bensì ne estende le capacità. La presenza dunque di un **control plane** K8s è una condizione necessaria per un corretto funzionamento dell’architettura proposta, poiché costituisce la base su cui operano i meccanismi di orchestrazione, deploy e sincronizzazione tra cloud ed edge [5].

5.3.1 Installazione Microk8s e kubeconfig

1. **Installare Microk8s:** disponibile come pacchetto all’interno degli snap Ubuntu.

Questo comando andrà ad inizializzare un cluster Kubernetes single-node completo sul nodo cloud [46].

```
sudo snap install microk8s --classic  
#abilitazione degli add on  
sudo microk8s enable dns  
sudo microk8s enable dashboard
```

2. **Verificare lo stato di Microk8s:** utilizzare il comando **status**. In alternativa, si può utilizzare l’opzione **--wait-ready** per attendere che tutti i servizi interni di K8s siano pronti.

```
sudo microk8s status  
sudo microk8s --wait-ready
```

In caso di problemi:

```
sudo microk8s inspect
```

3. **Opzionale - Aggiungere l’utente al gruppo Microk8s:** nel caso in cui si desidera eseguire i comandi di MicroK8s senza **sudo** e come utente normale, si può aggiungere l’utente corrente al gruppo **microk8s** e regolare i permessi della **kubeconfig**.

```
sudo usermod -a -G microk8s $USER  
mkdir -p ~/.kube  
sudo microk8s config > ~/.kube/config  
sudo chown -R $USER ~/.kube
```

In alternativa è possibile continuare a usare **sudo microk8s ...** per i comandi di Microk8s. La riga **microk8s config > ./kube/config** salva la configurazione **kubeconfig** di MicroK8s nell’home dell’utente, permettendo, nel nostro caso, di fornire il **kubeconfig** a **keadm**.

5.3.2 Installazione Docker

In questo progetto Docker non viene usato come runtime dei Pod, ma esclusivamente per costruire le immagini dei microservizi in locale e poi importarle nel runtime di MicroK8s.¹

1. **Installazione dei pacchetti Docker:** disponibili nel repository ufficiale di Ubuntu.

```
sudo apt install -y docker.io
```

2. **Abilitazione e verifica del servizio:** il servizio di Docker deve essere avviato e abilitato all'esecuzione automatica al boot del sistema.

```
sudo systemctl start docker
sudo systemctl enable docker
docker --version
```

```
lorenzofuse@Ubuntu-VMcloudCore:~$ sudo systemctl status docker
● docker.service - Docker Application Container Engine
  Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: enabled)
  Active: active (running) since Thu 2025-09-11 13:49:56 CEST; 6h ago
TriggeredBy: ● docker.socket
    Docs: https://docs.docker.com
   Main PID: 1694 (dockerd)
      Tasks: 10
     Memory: 103.5M (peak: 113.6M)
        CPU: 12.056s
      CGROUP: /system.slice/docker.service
              └─1694 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

Figura 5.4: Verifica dello stato di Docker

La corretta attivazione può essere verificata tramite il comando `systemctl status docker` come mostrato in Figura 5.4.

5.3.3 Installazione di keadm v1.20.0

Keadm è lo strumento a riga di comando per installare e gestire i componenti di KubeEdge (CloudCore e EdgeCore). Non si occupa di installare Kubernetes stesso, fornito da MicroK8s, né il runtime dei container, ma fornisce gli strumenti per le fasi di deployment e di join dei nodi al cluster [34].

1. **Scaricare il pacchetto keadm:** i binari di KubeEdge (incluso keadm) sono distribuiti su GitHub. Scaricare l'archivio tar.gz per keadm v1.20.0 (amd64).

```
export KEADM_VERSION="v1.20.0"
wget https://github.com/kubeedge/kubeedge/releases/
      download/${KEADM_VERSION}/keadm-${KEADM_VERSION}-linux-
      amd64.tar.gz
```

¹La motivazione di questa scelta rispetto ad alternative è discussa nella Sezione 5.10.2.

```
tar -xzf keadm-${KEADM_VERSION}-linux-amd64.tar.gz
```

Il comando `wget` scarica il tarball di `keadm` (verificare la versione scaricata che sia compatibile con l'architettura appropriata, nel nostro caso Ubuntu 22.04 LTS → amd64).

2. **Installare il binario `keadm`:** copiare il binario `keadm` estratto in una directory presente nel PATH, ad esempio `/usr/local/bin`.

```
sudo cp keadm-v1.20.0-linux-amd64/keadm/keadm /usr/local/bin/keadm  
sudo chmod +x /usr/local/bin/keadm
```

In questo modo il comando `keadm` è disponibile globalmente.

Per verificare la corretta esecuzione dei comandi riportati, eseguire:

```
keadm version
```

```
lorenzofuse@Ubuntu-VMCloudCore:~$ keadm version  
version: version.Info{Major:"1", Minor:"20", GitVersion:"v1.20.0", GitCommit:"bae61505d4919c404665f70347ad0646aaa98958", GitTreeState:"clean", BuildDate:"2025-01-21T12:47:49Z", GoVersion:"go1.22.9", Compiler:"gc", Platform:"linux/amd64"}
```

Figura 5.5: Verifica della versione di Keadm installata.

5.3.4 Inizializzazione del CloudCore sul nodo Cloud

In seguito all'attivazione del cluster con MicroK8s e `keadm` installato, si può procedere a inizializzare il **CloudCore**, ovvero installare i componenti cloud di KubeEdge.

1. **Preparare `kubeconfig`:** assicurarsi di avere il file `kubeconfig` del cluster Kubernetes pronto in `/.kube/config`. `Keadm` lo userà per connettersi al cluster K8s ed effettuare il deploy del CloudCore. Se non è presente, è possibile generarlo con:

```
sudo microk8s config > /root/.kube/config
```

2. **Eseguire `keadm init`:** con questo comando si installa il CloudCore. Bisogna specificare l'indirizzo IP esposto e la versione di KubeEdge, oltre al `kubeconfig`.

```
keadm init --advertise-address="" --kubeedge-version="${KEADM_VERSION}" --kube-config=/root/.kube/config
```

- `--advertise-address`: l'IP del nodo Cloud che i nodi Edge useranno per connettersi. In un ambiente a schema bridge come quello precedentemente configurato, è l'IP privato della VM Ubuntu-VMCloudCore. Questo IP verrà incluso nel certificato del CloudCore [34];

- **--kubededge-version**: specifica la versione da installare, nel caso di studio v1.20.0, come definito nella variabile d'ambiente;
- **--kube-config**: specifica il path al file di configurazione kubectl per il cluster k8s generato da MicroK8s. Keadm lo userà per verificare la connessione al cluster e deployare il CloudCore.

Durante l'esecuzione, keadm effettuerà un check della versione di Kubernetes e poi procederà all'installazione. Internamente, verrà utilizzato **Helm** per installare CloudCore come pod di K8s (in namespace **kubededge**).

L'output atteso dopo questi passaggi deve essere:

```
Kubernetes version verification passed , KubeEdge
installation will start...
KubeEdge cloudcore is running , For logs visit: /var/log/
kubededge/cloudcore.log
CloudCore started
```

Osservazione keadm init in KubeEdge v1.20 di default installa CloudCore in container, creando un deployment k8s. Non viene creata una service unit systemd per CloudCore sul nodo host, poiché gira come pod nel cluster. Per configurazioni alternative, fare riferimento alla documentazione di KubeEdge [32].

3. **Verificare l'installazione del CloudCore**: in seguito all'esecuzione dei comandi precedentemente indicati, per controllare che il pod CloudCore sia effettivamente in esecuzione nel cluster MicroK8s, eseguire:

```
microk8s kubectl get all -n kubededge
```

Se il comando produce un output simile a quello in Figura 5.6, contenente un pod **cloudcore-xxxxx** in stato **Running** e un servizio associato sulle porte 10000/10002, l'operazione è andata a buon fine.

```
lorenzofuse@Ubuntu-VMCloudCore:~$ microk8s kubectl get all -n kubededge
NAME                           READY   STATUS    RESTARTS   AGE
pod/cloud-iptables-manager-5fw6k   1/1    Running   48 (6h4m ago)   118d
pod/cloudcore-5d9ccb9dc8-p5xlf   1/1    Running   48 (6h4m ago)   118d
pod/edge-eclipse-mosquitto-czd4w  1/1    Running   46 (6h3m ago)   118d
pod/edge-eclipse-mosquitto-dw8w8  1/1    Running   534 (9m28s ago)  117d
pod/edge-eclipse-mosquitto-q5jgk  1/1    Running   51 (6h3m ago)   118d

NAME              TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)           AGE
service/cloudcore  NodePort   10.152.183.228  <none>          10000:31000/TCP,10001:30199/UDP,10002:31339/TCP,10003:31438/TCP,10004:30749/TCP  118d

NAME          DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE SELECTOR  AGE
daemonset.apps/cloud-iptables-manager  1        1        1        1        1        <none>       118d
daemonset.apps/edge-eclipse-mosquitto  3        3        3        3        3        <none>       118d

NAME          READY  UP-TO-DATE  AVAILABLE  AGE
deployment.apps/cloudcore  1/1     1           1           118d

NAME          DESIRED  CURRENT  READY  AGE
replicaset.apps/cloudcore-5d9ccb9dc8  1        1        1        118d
```

Figura 5.6: Output del comando per la verifica dello stato del CloudCore e dei servizi associati

In caso di problemi, è possibile controllare i log del CloudCore e assicurarsi che sia partito correttamente in assenza di errori, oppure verificare che il processo sia attivo con il secondo comando:

```
microk8s kubectl logs -n kubeedge deployment.apps/
  clouddcore
ps -ef | grep clouddcore
```

4. **Ottenere il token di Join:** Il CloudCore genera un **token** di sicurezza necessario ai nodi Edge per unirsi al cluster, usato per l'autenticazione iniziale alla richiesta del certificato. Per recuperarlo, eseguire sul nodo Cloud:

```
keadm gettoken --kube-config /home/lorenzofuse/.kube/
config
```

Utilizzando il comando di default **keadm gettoken**, quest'ultimo effettua la ricerca per default nel path di file kubeconfig posizionato in **/root/.kube/config**, ma avendolo copiato in precedenza si dovrà sostituire il nome utente con il proprio.

Questo comando restituirà una stringa alfanumerica separata da tre punti come mostrato in Figura 5.7.

```
lorenzofuse@Ubuntu-VHCloudCore:~/keadm-v1.20.0-linux-amd64$ sudo ./keadm/gettoken
failed to get token, err is get kube config failed with error: stat /root/.kube/config: no such file or directory
Error: get kube config failed with error: stat /root/.kube/config: no such file or directory
Usage:
  keadm gettoken [flags]

Examples:

keadm gettoken --kube-config /root/.kube/config
- kube-config is the absolute path of kubeconfig which used to build secure connectivity between keadm and kube-apiserver
to get the token.

Flags:
  -h, --help           help for gettoken
  --kube-config string Use this key to set kube-config path, eg: $HOME/.kube/config (default "/root/.kube/config")

Global Flags:
  -v, --v Level    number for the log level verbosity

execute keadm command failed: get kube config failed with error: stat /root/.kube/config: no such file or directory
lorenzofuse@Ubuntu-VHCloudCore:~/keadm-v1.20.0-linux-amd64$ sudo ./keadm/gettoken --kube-config /var/snap/microk8s/current/credentials/client.config
a3b18a8c9f2ca543589571547f15acc2aa0277be8253ca1d36ee2c41c15441fa.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpxVCJ9.eyJleHAiOiE3NDIzMTISMTZ9.YSu9L2Nrz9B_5sFFx9691JZkypSt0ITFEa67EfThQ-o
```

Figura 5.7: Generazione del token sul cloud

5.4 Configurazione dei nodi Edge - EdgeCore

Su ciascuna delle tre VM Edge verrà utilizzato **keadm** per installare ed avviare il componente **EdgeCore**, unendo ogni nodo edge al CloudCore, e la relativa installazione del runtime dei container **containerd**.

I passaggi che verranno mostrati in questa sezione devono essere ripetuti su ciascuno dei tre nodi Edge, assicurandosi che ogni nodo abbia un hostname univoco, in quanto rappresenterà il nome all'interno del cluster K8s.

5.4.1 Installazione di containerd

- Installare containerd tramite apt:** Ubuntu 22.04 include il pacchetto di **containerd** nei repository ufficiali (**containerd** e non **containerd.io**, quest'ultimo legato a Docker e potrebbe non includere il supporto CRI corretto [21]).

```
sudo apt update
sudo apt install -y containerd
```

Questo installerà il demone containerd e ne avvierà il servizio, per poterlo verificare eseguire `systemctl status containerd`.

```
lorenzofuse@Ubuntu-EdgeCoreVM3:~$ systemctl status containerd
● containerd.service - containerd container runtime
   Loaded: loaded (/usr/lib/systemd/system/containerd.service; enabled; preset: enabled)
   Active: active (running) since Tue 2025-09-02 10:40:17 CEST; 7h ago
     Docs: https://containerd.io
 Main PID: 1119 (containerd)
    Tasks: 69
   Memory: 114.8M (peak: 121.1M)
      CPU: 37min 30.080s
    CGroup: /system.slice/containerd.service
            └─1119 /usr/bin/containerd
                  ├─2237 /usr/bin/containerd-shim-runc-v2 -namespace k8s.io -id 902320aa76f431238e35
                  ├─2243 /usr/bin/containerd-shim-runc-v2 -namespace k8s.io -id cff893e90707a17c2662
                  ├─2306 /usr/bin/containerd-shim-runc-v2 -namespace k8s.io -id 45972983d56227032e30
                  ├─2330 /usr/bin/containerd-shim-runc-v2 -namespace k8s.io -id 8413991bbacd4d2b0abd
```

Figura 5.8: Stato del servizio containerd dopo l'installazione.

2. **Configurare containerd per KubeEdge** Se containerd è stato installato senza un file di configurazione personalizzato, si consiglia di generarne uno e di assicurarsi che il plugin CRI sia abilitato.²

```
sudo mkdir -p /etc/containerd
sudo containerd config default | sudo tee /etc/containerd
    /config.toml
sudo systemctl restart containerd
sudo systemctl enable containerd
```

Il comando `containerd config default` genera un file di configurazione di default in `/etc/containerd/config.toml`. All'interno del file bisogna verificare che non ci sia l'opzione `disable_plugins` con "cri" impostato, come mostrato in Figura 5.9. In caso contrario, bisogna rimuoverlo per il corretto funzionamento di containerd con Kubernetes [33].

3. **Verifica del plugin CRI:** per essere sicuri che containerd sia pronto per KubeEdge, si può eseguire:

```
sudo ctr plugins ls | grep io.containerd.grpc.v1.cri
sudo cat /etc/containerd/config.toml
```

²In questa sezione viene omessa la spiegazione dell'installazione a Docker. Per costruire le immagini direttamente sui nodi Edge, ripetere la stessa installazione di eseguita sul CloudCore (Sezione 5.3.2)

```

lorenzofuse@Ubuntu-EdgeCoreVM3:~$ sudo ctr plugins ls | grep cri
io.containerd.grpc.v1 cri linux/amd64 ok
lorenzofuse@Ubuntu-EdgeCoreVM3:~$ sudo cat /etc/containerd/config.toml
disabled_plugins = []
imports = []
oom_score = 0
plugin_dir = ""
required_plugins = []
root = "/var/lib/containerd"
state = "/run/containerd"
temp = ""
version = 2

[cgroup]
  path = ""

[debug]
  address = ""
  format = ""
  gid = 0
  level = ""
  uid = 0

```

Figura 5.9: Output atteso

Questo conferma che containerd espone l’interfaccia CRI, necessaria affinché EdgeCore possa creare i pod sui nodi edge. In caso di problemi, provare a ravviare containerd con `sudo systemctl restart containerd` e riprendere il procedimento di verifica del plugin.

5.4.2 Installazione keadm v1.20.0

Per consentire l’unione di un nodo edge al cloud tramite l’operazione di **Join**, è necessario utilizzare `keadm`, che provvede all’installazione e alla configurazione del servizio **EdgeCore**.

La procedura di installazione è la medesima eseguita sul cloud, seguendo i passaggi riportati di seguito:

1. **Scaricare il tarball di keadm:** dalla repository Github, versione v1.20.0 per architettura amd64:

```

export KEADM_VERSION="v1.20.0"
wget https://github.com/kubeeedge/kubeeedge/releases/
      download/${KEADM_VERSION}/keadm-${KEADM_VERSION}-linux-
      amd64.tar.gz
tar -xzf keadm-${KEADM_VERSION}-linux-amd64.tar.gz

```

2. **Installare il binario keadm:** copiando l’eseguibile nella directory `/usr/local/bin` e rendendolo disponibile a tutti gli utenti del sistema.

```

sudo cp keadm-${KEADM_VERSION}-linux-amd64/keadm/keadm /
  usr/local/bin/
sudo chmod +x /usr/local/bin/keadm

```

5.4.3 Join di ogni nodo Edge al Cloud

A questo punto è possibile collegare ciascun nodo edge al cloud eseguendo il comando **keadm join**.

Prima di procedere, è necessario disporre di:

- L'**indirizzo IP del nodo Cloud** usato in precedenza come `--advertise-address` durante l'`init`.
- Il **token** generato in precedenza sul cloud (`keadm gettoken`).
- Eventualmente, il percorso della socket di containerd, in genere `/run/containerd/containerd.sock`

Ripetere le seguenti operazioni su ciascun nodo Edge:

1. **Eseguire keadm join:** utilizzare il comando seguente sostituendo `<IP_CLOUD>` con l'IP del nodo Cloud e `<TOKEN>` con il token copiato.

```
keadm join --cloudcore-ipport=<IP_CLOUD>:10000 \
--token=<TOKEN> \
--kubedge-version="v1.20.0" \
--remote-runtime-endpoint=unix:///run/containerd/
    containerd.sock
```

- `--cloudcore-ipport=<IP_CLOUD>:10000` specifica l'indirizzo IP e porta del CloudCore a cui collegarsi. La porta predefinita è 10000, configurata di default da CloudCore;
- `--token=<TOKEN>` fornisce il token di autenticazione generato sul cloud, necessario per richiedere ed ottenere i certificati dal cloud;
- `--kubedge-version=v1.20.0` indica la versione di EdgeCore da installare;
- `--remote-runtime-endpoint=unix:///run/containerd/containerd.sock` configura l'endpoint CRI del runtime container sul nodo edge. Questo procedimento è necessario in quanto stiamo indicando a EdgeCore di usare **containerd** tramite la sua UNIX socket (percorso predefinito).

Questo passaggio è fondamentale, in quanto di default `keadm join` potrebbe cercare Docker, indicandolo a riga di comando come specificato viene indirizzato a Containerd.

Se l'esecuzione dei comandi precedentemente mostrati va a buon fine, il terminale mostrerà un messaggio di successo come segue:

```
KubeEdge edgecore is running,
For logs visit: journalctl -u edgecore.service -xe
```

2. **Verificare lo stato di EdgeCore:** in seguito all'operazione di join, controllare che il servizio `edgecore` stia girando correttamente sul nodo in questione, eseguendo:

```
sudo systemctl status edgecore
```

L'output atteso deve essere come in Figura 5.10:

```
lorenzofuse@Ubuntu-EdgeCoreVM3:~$ sudo systemctl status edgecore
● edgecore.service
    Loaded: loaded (/etc/systemd/system/edgecore.service; enabled; preset: enabled)
    Active: active (running) since Wed 2025-09-03 11:43:21 CEST; 3h 20min ago
      Main PID: 1972 (edgecore)
        Tasks: 16 (limit: 3387)
       Memory: 44.0M (peak: 45.6M)
         CPU: 13min 59.284s
      CGroup: /system.slice/edgecore.service
              └─1972 /usr/local/bin/edgecore
```

Figura 5.10: Verifica di stato sul servizio EdgeCore

In caso lo stato non sia **Running**, visualizzare i log ed identificare eventuali errori, come ad esempio token non valido oppure connessione al cloud fallita. Se EdgeCore è attivo, significa che il nodo edge ha stabilito una nuova connessione con il CloudCore.

5.4.4 Verifica del corretto funzionamento dell'ambiente KubeEdge

Dopo aver configurato il cloud e unito tutti i nodi edge, si può procedere a verificare che l'intero sistema KubeEdge funzioni come previsto:

Verifica dello stato

Sul nodo cloud, controllare che CloudCore sia in esecuzione nel cluster.

- Eseguire `microk8s kubectl get pods -n kubeedge` e verificare che il pod CloudCore sia **Running**; in caso di problemi controllare i log con `microk8s kubectl logs -n kubeedge deployment/clouddcore`.
- Accertarsi che le porte di servizio (10000 e 10002) siano effettivamente in ascolto. Eseguire `ss -lntp | grep 10000` oppure verificare tramite `microk8s kubectl get svc -n kubeedge` che il service `clouddcore` esponga tali porte.
- Controllare se nel cluster compaiano i nodi edge registrati:

```
microk8s kubectl get nodes -o wide
```

Oltre al nodo cloud devono essere presenti i nodi con nomi corrispondenti agli hostname delle VM edge. Inizialmente i nodi edge potrebbero risultare **Not Ready**; attendendo qualche secondo e rieseguendo il comando dovrebbero passare a **Ready**. Se non compaiono, verificare i passaggi precedenti.

Su ciascun nodo edge, assicurarsi che il servizio `edgecore` continui a funzionare correttamente, tramite i comandi precedentemente illustrati `sudo systemctl status edgecore` ed eventualmente se sono presenti errori, visualizzare i log tramite `journalctl -u edgecore -n`.

```
lorenzofuse@Ubuntu-VMCloudCore:~$ microk8s kubectl get pods -n kubeedge
NAME                   READY   STATUS    RESTARTS   AGE
cloud-iptables-manager-5fw6k   1/1    Running   49 (4h5m ago)   119d
cloudcore-5d9ccb9dc8-psxf   1/1    Running   49 (4h5m ago)   119d
edge-eclipse-mosquitto-czd4w  1/1    Running   48 (46m ago)   119d
edge-eclipse-mosquitto-dw8w8  1/1    Running   535 (3h59m ago)  118d
edge-eclipse-mosquitto-q5jk   1/1    Running   52 (3h48m ago)  119d
lorenzofuse@Ubuntu-VMCloudCore:~$ microk8s kubectl get svc -n kubeedge
NAME      TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
cloudcore   NodePort   10.152.183.228 <none>        10000:31000/TCP,10001:30199/UDP,10002:31339/TCP,10003:31438/TCP,10004:30749/TCP   119d
lorenzofuse@Ubuntu-VMCloudCore:~$ ss -lntp | grep 10000
LISTEN 0      *:10000           *:*
lorenzofuse@Ubuntu-VMCloudCore:~$ ss -lntp | grep 10002
LISTEN 0      4096             *:10002          *:*
lorenzofuse@Ubuntu-VMCloudCore:~$ microk8s kubectl get nodes -o wide
NAME            STATUS   ROLES   AGE     VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE       KERNEL-VERSION   CONTAINER-RUNTIME
ubuntu-edgecorevm   Ready   agent,edge   119d   v1.30.7-kubeedge-v1.20.0   192.168.9.163   <none>        Ubuntu 24.04.2 LTS   6.11.0-29-generic   containerd://1.7.27
ubuntu-edgecorevm2   Ready   agent,edge   119d   v1.30.7-kubeedge-v1.20.0   192.168.9.127   <none>        Ubuntu 24.04.2 LTS   6.14.0-29-generic   containerd://1.7.27
ubuntu-edgecorevm3   Ready   agent,edge   119d   v1.30.7-kubeedge-v1.20.0   192.168.9.191   <none>        Ubuntu 24.04.2 LTS   6.11.0-29-generic   containerd://1.7.27
ubuntu-vmcloudcore   Ready   <none>    119d   v1.32.3        192.168.9.128   <none>        Ubuntu 24.04.2 LTS   6.14.0-29-generic   containerd://1.6.36
```

Figura 5.11: Output dei comandi per la verifica di CloudCore, servizi e nodi del cluster KubeEdge

5.5 Broker MQTT - Mosquitto

Il broker MQTT **Eclipse Mosquitto** è installato sul nodo cloud come servizio `systemd` (non containerizzato), esponendo un unico endpoint 1883/tcp verso tutti gli edge.

1. Installazione nodo cloud utilizzando i pacchetti forniti da `apt`.

L'esecuzione dei seguenti comandi permette l'installazione del demone `mosquitto` e i client CLI. Per garantire la continuità del servizio anche in seguito a riavvii della macchina, il servizio `systemd` viene abilitato e avviato immediatamente; si conclude con la verifica dello stato per assicurarsi del corretto funzionamento.

```
sudo apt install -y mosquitto mosquitto-clients
sudo systemctl enable --now mosquitto
systemctl status mosquitto --no-pager
```

```
lorenzofuse@Ubuntu-VMCloudCore:~$ sudo systemctl status mosquitto
● mosquitto.service - Mosquitto MQTT Broker
   Loaded: loaded (/usr/lib/systemd/system/mosquitto.service; enabled; preset: enabled)
   Active: active (running) since Thu 2025-09-04 11:20:22 CEST; 4h 5min ago
     Docs: man:mosquitto.conf(5)
           man:mosquitto(8)
   Process: 1188 ExecStartPre=/bin/mkdir -m 740 -p /var/log/mosquitto (code=exited, status=0/SUCCESS)
   Process: 1214 ExecStartPre=/bin/chown mosquitto:mosquitto /var/log/mosquitto (code=exited, status=0/SUCCESS)
   Process: 1229 ExecStartPre=/bin/mkdir -m 740 -p /run/mosquitto (code=exited, status=0/SUCCESS)
   Process: 1245 ExecStartPre=/bin/chown mosquitto:mosquitto /run/mosquitto (code=exited, status=0/SUCCESS)
 Main PID: 1279 (mosquitto)
   Tasks: 1 (limit: 6845)
     Memory: 2.3M (peak: 3.2M)
        CPU: 3min 2.703s
      CGroup: /system.slice/mosquitto.service
              └─1279 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

Sep 04 11:20:22 Ubuntu-VMCloudCore systemd[1]: Starting mosquitto.service - Mosquitto MQTT Broker...
Sep 04 11:20:22 Ubuntu-VMCloudCore systemd[1]: Started mosquitto.service - Mosquitto MQTT Broker.
```

Figura 5.12: Verifica di stato del servizio Mosquitto

L'abilitazione del servizio serve ad evitare falsi positivi, nel caso in cui al ravvio della macchina `mosquitto` non dovesse essere in stato di `Running`.

2. **Configurazione del listener:** prevede la persistenza dello stato e il logging su file, oltre alla pubblicazione di un listener sul socket `0.0.0.0:1883` in modo che il broker sia raggiungibile dai nodi edge sulla stessa sottorete.

```
sudo nano /etc/mosquitto/mosquitto.conf
#aggiungere
listener 1883 0.0.0.0
```

```
GNU nano 7.2                                     /etc/mosquitto/mosquitto.conf
# Place your local configuration in /etc/mosquitto/conf.d/
#
# A full description of the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

#pid_file /run/mosquitto/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

include_dir /etc/mosquitto/conf.d

listener 1883
allow_anonymous true
```

Figura 5.13: File di configurazione di Mosquitto

Per un'ulteriore verifica sulla porta 1883, una volta riavviato il servizio tramite `sudo systemctl restart mosquitto`, è possibile eseguire `sudo ss -tuln | grep 1883` per verificare che il processo sia in ascolto sull'indirizzo previsto.

```
lorenzofuse@Ubuntu-VMCloudCore:~$ sudo ss -tuln | grep 1883
tcp    LISTEN  0          100           0.0.0.0:1883          0.0.0.0:*
tcp    LISTEN  0          100           [::]:1883            [::]:*
```

Figura 5.14: Verifica sulla porta 1883

3. **Installazione nodi Edge:** sulle VM Edge è necessario installare i client MQTT per consentire le pubblicazioni. La procedura è analoga a quella eseguita sul cloud:

```
sudo apt install mosquitto-clients
```

Completata l'installazione, si può verificare tramite il comando `nc -vz <IP_CLOUD> 1883` la raggiungibilità del broker posizionato sul cloud.

```

lorenzofuse@Ubuntu-EdgeCoreVM2:~$ nc -vz 192.168.9.128 1883
Connection to 192.168.9.128 1883 port [tcp/*] succeeded!
lorenzofuse@Ubuntu-EdgeCoreVM2:~$ 

```

Figura 5.15: Output atteso

4. **Configurazione di EdgeCore:** Per consentire alle applicazioni in esecuzione sui nodi edge di scambiare informazioni via MQTT senza dipendenza diretta di un broker locale, il modulo `EventBus` di KubeEdge è stato configurato per utilizzare il broker centralizzato posizionato in cloud.

La modifica da effettuare deve essere eseguita all'interno del file di configurazione di `edgecore` e si indirizza il `mqttServerExternal` verso l'indirizzo IP del cloud.

```

eventBus:
  enable: true
  eventBusTLS:
    enable: false
    tlsMqttCAFfile: /etc/kubeedge/ca/rootCA.crt
    tlsMqttCertFile: /etc/kubeedge/certs/server.crt
    tlsMqttPrivateKeyFile: /etc/kubeedge/certs/server.key
  mqttMode: 0 #prima a 2, messo a 0 per disattivare mqtt interno
  mqttPassword: ""
  mqttPubClientID: ""
  mqttQOS: 0
  mqttRetain: false
  mqttServerExternal: tcp://192.168.9.128:1883
  mqttServerInternal: tcp://127.0.0.1:1884
  mqttSessionQueueSize: 100
  mqttSubClientID: ""
  mqttUsername: ""

```

Figura 5.16: Modifiche da effettuare all'interno del file config

Dopo l'aggiornamento della configurazione si riavvia il servizio e si verificano i registri per accertare l'avvenuta connessione al broker.

```

sudo systemctl restart edgecore
journalctl -u edgecore -b | egrep -i 'eventbus|mqtt|connect'

```

```

lorenzofuse@Ubuntu-EdgeCoreVM3:~$ sudo systemctl status edgecore
● edgecore.service
   Loaded: loaded (/etc/systemd/system/edgecore.service; enabled; preset: enabled)
   Active: active (running) since Wed 2025-09-03 11:43:21 CEST; 3h 20min ago
     Main PID: 1972 (edgecore)
        Tasks: 16 (limit: 3387)
       Memory: 44.0M (peak: 45.6M)
          CPU: 13min 59.284s
        CGroup: /system.slice/edgecore.service
                  └─1972 /usr/local/bin/edgecore

```

Figura 5.17: Servizio EdgeCore in seguito alle modifiche effettuate

5.6 Struttura dei file e collocazione

5.6.1 Panoramica

L'organizzazione dei file del progetto riflette i ruoli funzionali dei nodi nel cluster KubeEdge.

Il nodo centrale che ospita la componente **CloudCore** contiene:

- il microservizio che si occupa di ricezione, verifica, persistenza e relativa visualizzazione dei dati;
- il microservizio che si occupa di pubblicare le metriche di sistema relativi alla propria VM;
- il microservizio che si occupa di ricevere tutti i parametri delle VM del cluster, persistenza e visualizzazione di tali dati.

Il tre nodi **Edge** eseguono:

- il microservizio di pubblicazione dei dati ambientali, ciascuno riferito a specifiche zone assegnate dal dataset **BuildPred**.
- il microservizio relativo alla pubblicazione delle metriche di sistema relative alla propria VM come nel cloud.

5.6.2 Mappatura dei microservizi

La distribuzione dei microservizi IAQ nei nodi Edge è stata effettuata assegnando a ciascun **publisher** una determinata zona che include diverse stanze. La Figura 5.18 mostra la collocazione dei microservizi rispetto alla planimetria dell'edificio: ogni colore rappresenta un nodo edge e i rettangoli evidenziano le zone e stanze coperte dal relativo microservizio **iaq-publisher**.

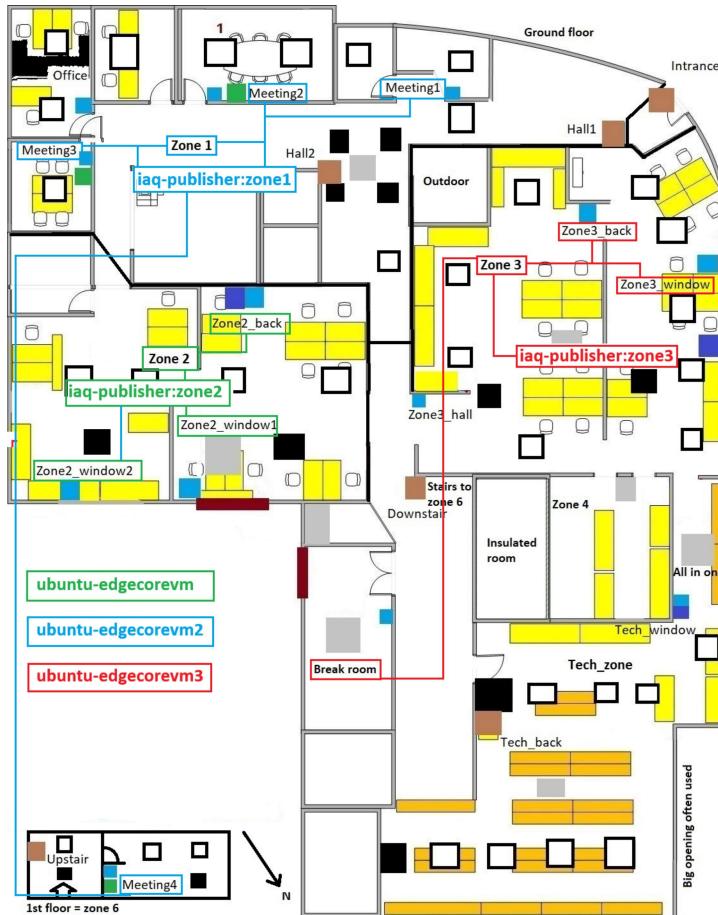


Figura 5.18: Mappatura dei microservizi a zone IAQ

5.6.3 Organizzazione del progetto

Ogni microservizio dispone di una propria directory, organizzata in modo da contenere sia il codice sia i file di configurazione. I microservizi IAQ operano come *publisher* sui nodi edge e come *subscriber* sul cloud, mentre i microservizi METRICE raccolgono e pubblicano le risorse delle VM coinvolte, che vengono poi gestite dal cloud. Di seguito vengono illustrate le cartelle effettive con una descrizione, mentre i dettagli relativi ai meccanismi di cifratura (ECDH/ECDSA/HKDF/AES) e ai file di configurazione (Dockerfile e Manifest YAML) saranno trattati in modo più approfondito all'interno del Capitolo.

Nodo Edge - publisher IAQ

Nei nodi Edge, la directory `kubeedge_microservices/app/` contiene:

- `publisher.py`: legge il dataset `BuildPred`, genera il payload cifrato e firmato e lo pubblica su MQTT;
- `C02_Adeunis.xlsx` e `temperatures_Adeunis.xlsx`: sorgenti dati;
- chiavi ECC (`ecc_private_edgeX.pem`, `ecc_public_cloud.pem`), la cui generazione sarà descritta nella Sezione 5.7 *Crittografia e sicurezza*;

- Dockerfile per la costruzione dell'immagine del publisher (descritta nella Sezione 5.10 Build & Deploy)

```
lorenzofuse@Ubuntu-EdgeCoreVM:~/kubeeedge_microservices/app$ 
.
├── CO2_Adeunis.xlsx
├── ecc_private_edge.pem
├── ecc_public_cloud.pem
└── ecc_public_edge.pem
├── generator.py
└── iaq-publisher-zone2.tar
├── publisher.py
└── temperatures_Adeunis.xlsx
```

Figura 5.19: Directory IAQ su un nodo Edge (publisher.py e dataset).

Il *publisher* carica e unisce i due fogli Excel di CO₂ e temperatura, e simula la trasmissione dei dati pubblicando ciclicamente, con frequenza accelerata a fini sperimentali (un secondo), verso i topic corrispondenti a zona e stanza.

La cifratura e la firma sono gestite dalla funzione `encrypt_and_sign()`.³

```
1 df = pd.merge(
2     pd.read_excel("CO2_Adeunis.xlsx") [["Date"] + SENSORS],
3     pd.read_excel("temperatures_Adeunis.xlsx") [["Date"] +
4         SENSORS],
5     on="Date", suffixes=('_co2', '_temp')
6 )
7 #.....
8 for idx, row in df.iterrows():
9     for sensor in SENSORS:
10         topic = f"iaq/{ZONE}/{sensor.lower()}"
11         #costruzione messaggio con ts pubblicazione,
12         ts sensore e valore
13         msg = f"{datetime.now()}, {row['Date']}, CO2: {row[f'{sensor}_co2']:.2f} ppm"
14         encrypted = encrypt_and_sign(msg)    # cifratura +
            firma
         client.publish(topic, encrypted)
```

³La logica di cifratura/decifratura sarà analizzata nel dettaglio nella sezione dedicata alla sicurezza.

Nodo CloudCore - subscriber IAQ

Sul nodo centrale, la directory `kubeedge_microservices/app/` contiene:

- `subscriber.py`: si sottoscrive ai topic `iaq/#`, riceve i messaggi, li decifra e ne verifica la firma, calcola la latenza e inserisce i dati in PostgreSQL;
- chiavi ECC del cloud e chiavi pubbliche degli edge, necessarie per la verifica. (`ecc_private_cloud.pem`, `ecc_public_edgeX.pem`).

```
lorenzofuse@Ubuntu-VMCloudCore:~/kubeedge_microservices/app$ 
.
├── ecc_private_cloud.pem
├── ecc_public_cloud.pem
├── ecc_public_edge2.pem
├── ecc_public_edge3.pem
├── ecc_public_edge.pem
└── generator.py
    └── subscriber.py
```

Figura 5.20: Directory IAQ sul CloudCore (`subscriber.py` e chiavi ECC).

All'avvio, il subscriber svuota la tabella `iaq_data` per evitare inconsistenze nelle dashboard. Per ogni messaggio ricevuto richiama la funzione `decrypt_and_verify()`, ed effettua l'inserimento nel database.

```
1 cursor.execute("TRUNCATE TABLE iaq_schema.iaq_data;"); conn.
2     commit()
3 #.....
4 def on_message(client, userdata, msg):
5     ts_arrivo = datetime.now()
6     payload   = decrypt_and_verify(msg.payload.decode())
7     ts_pub_raw, ts_sensore_raw, misura = payload.split(", ", 3)
8     cursor.execute("""
9         INSERT INTO iaq_schema.iaq_data (ts_arrivo, topic,
10             ts_sensore, tipo, valore, latenza_pub_sub)
11             VALUES (%s, %s, %s, %s, %s, %s)
12             """, (ts_arrivo, msg.topic, ts_pub_raw, tipo, valore,
13                 latenza))
14     conn.commit()
```

Nodi Edge - publisher Metriche

Su ciascun nodo Edge, nella directory `kubeedge-metrics/vm-metrics-publisher/`, sono presenti lo script `vm-metrics-publisher.py` e il relativo Dockerfile.

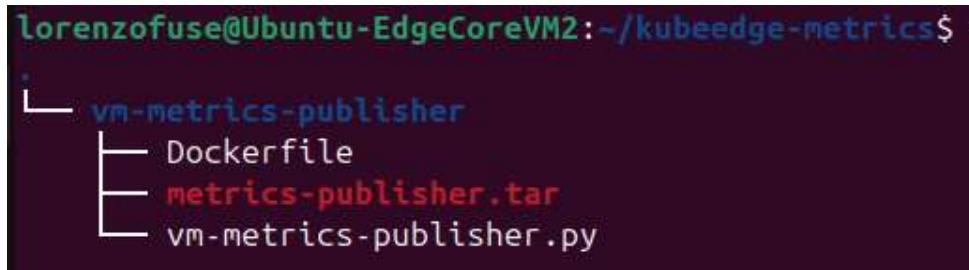


Figura 5.21: Directory Metriche su un nodo Edge (`vm-metrics-publisher.py`).

Il microservizio raccoglie periodicamente le metriche locali della VM (CPU, RAM, disco e rete) utilizzando la libreria `psutil`, costruisce un payload in formato JSON e lo pubblica via MQTT verso il broker centralizzato:

```
1 import psutil
2
3 while True:
4     cpu = psutil.cpu_percent(interval=1)
5     ram = psutil.virtual_memory()
6     disk = psutil.disk_usage('/')
7     net = psutil.net_io_counters()
8
9     payload = {
10         "timestamp": time.strftime('%Y-%m-%d %H:%M:%S'),
11         "hostname": socket.gethostname(),
12         "cpu_percent": cpu,
13         "ram_used_mb": ram.used // (1024 * 1024),
14         "ram_total_mb": ram.total // (1024 * 1024),
15         "disk_used_gb": round(disk.used / (1024**3), 2),
16         "net_in_kbps": round(net.bytes_recv / 1024, 2),
17         "net_out_kbps": round(net.bytes_sent / 1024, 2)
18     }
19
20     client.publish(topic, json.dumps(payload))
21     time.sleep(10)
```

Il topic di pubblicazione segue la forma `vm/metrics/<hostname>`, così da distinguere univocamente i nodi.

Nodo Cloudcore - publisher e subscriber Metriche

Sul nodo CloudCore la cartella `kubeedge-metrics/` ospita due directory:

- `vm-metrics-publisher-cloudcore/`: pubblica le metriche della VM cloud secondo la stessa logica dei publisher edge;
- `vm-metrics-subscriber/`: raccoglie tutti i messaggi pubblicati dai nodi Edge e dal CloudCore, e li persiste nel database PostgreSQL.

Lo script `vm-metrics-subscriber.py` si sottoscrive al topic `vm/metrics/#` e, per ogni messaggio ricevuto, effettua il parsing del JSON e inserisce i valori nella tabella `vm_metrics`:

```
1 def on_message(client, userdata, msg):
2     data = json.loads(msg.payload.decode())
3     cursor.execute("""
4         INSERT INTO iaq_schema.vm_metrics (
5             timestamp, hostname, cpu_percent,
6             ram_used_mb, ram_total_mb, disk_used_gb,
7             net_in_kbps, net_out_kbps
8         )
9         VALUES (%s, %s, %s, %s, %s, %s, %s)
10    """, (
11        data["timestamp"], data["hostname"],
12        data["cpu_percent"], data["ram_used_mb"],
13        data["ram_total_mb"], data["disk_used_gb"],
14        data["net_in_kbps"], data["net_out_kbps"]
15    ))
16    conn.commit()
```

In questo modo, il CloudCore raccoglie in modo centralizzato le metriche di tutte le VM (edge e cloud), consentendo di analizzare le prestazioni complessive del sistema e visualizzarle successivamente in Grafana.

5.7 Crittografia e sicurezza applicativa

La protezione dei messaggi scambiati tra i nodi edge e il cloud è stata implementata combinando l'utilizzo di diversi algoritmi crittografici. Ogni messaggio viene cifrato con AES-256 in modalità Cipher Feedback, utilizzando la chiave simmetrica derivata dinamicamente tramite **Elliptic Curve Diffie–Hellman Ephemeral (ECDHE)** e **HMAC-based Key Derivation Function (HKDF)**.

Per garantire l'autenticità e l'integrità del contenuto, il testo cifrato viene firmato digitalmente con l'algoritmo **Elliptic Curve Digital Signature Algorithm (ECDSA)**.

5.7.1 Generazione delle chiavi

Per ogni componente che partecipa allo scambio dei dati cifrati, le chiavi ECC sono state generate localmente all'interno della rispettiva directory del microservizio IAQ, utilizzando la curva `secp384r1`.

All'interno della directory del microservizio IAQ accenata in precedenza, si è predisposto un ambiente Python isolato ed installate le dipendenze necessarie per la corretta generazione (`cryptography`). Di seguito i comandi eseguiti:

```
cd ~/kubeeedge_microservices/app
python3 -m venv venv
source venv/bin/activate
pip install cryptography
```

Lo script che si occupa della generazione di una coppia di chiavi ECC e della serializzazione in formato PEM.⁴

```
1 from cryptography.hazmat.primitives.asymmetric import ec
2 from cryptography.hazmat.primitives import serialization
3
4 priv = ec.generate_private_key(ec.SECP384R1())
5 pub = priv.public_key()
6
7 with open("ecc_private_cloud.pem", "wb") as f:
8     #ecc_private_edgeX.pem
9     f.write(priv.private_bytes(
10         encoding=serialization.Encoding.PEM,
11         format=serialization.PrivateFormat.TraditionalOpenSSL,
12         encryption_algorithm=serialization.NoEncryption()
13     ))
14
15 with open("ecc_public_cloud.pem", "wb") as f:
16     #ecc_public_edgeX.pem
17     f.write(pub.public_bytes(
18         encoding=serialization.Encoding.PEM,
19         format=serialization.PublicFormat.SubjectPublicKeyInfo
20     ))
```

Per permettere la creazione della coppia di chiavi, verificare che l'ambiente sia attivo e successivamente eseguire lo script.

```
source venv/bin/activate
python generator.py
deactivate
```

⁴Il codice è analogo per Cloud ed Edge; cambiano i nomi dei file che verranno generati.

Una volta verificata la corretta esecuzione dello script, con la relativa creazione dei file .pem, disattivare l'ambiente con l'ultimo comando mostrato in precedenza.

```
lorenzofuse@Ubuntu-VMCloudCore:~/kubeeedge_microservices/app$ 
.
├── ecc_private_cloud.pem
├── ecc_public_cloud.pem
├── generator.py
└── subscriber.py
└── venv
    ├── bin
    │   ├── activate
    │   ├── activate.csh
    │   ├── activate.fish
    │   ├── Activate.ps1
    │   ├── pip
    │   ├── pip3
    │   ├── pip3.12
    │   ├── python -> python3
    │   ├── python3 -> /usr/bin/python3
    │   └── python3.12 -> python3
    ├── include
    └── lib
        └── python3.12
    lib64 -> lib
    pyvenv.cfg
    tld
        └── python3.12
    lib64 -> lib
    pyvenv.cfg
```

Figura 5.22: Struttura della directory in seguito alla generazione delle chiavi

5.7.2 Distribuzione e caricamento delle chiavi pubbliche

La corretta gestione delle chiavi è un prerequisito fondamentale della cifratura end-to-end.

In questo progetto, ogni nodo edge dispone di una propria coppia di chiavi ECC (privata e pubblica), oltre alla chiave pubblica del cloud. Quest'ultimo, responsabile della ricezione dei dati provenienti da tutti i nodi, necessita a sua volta, oltre la propria coppia di chiavi, anche delle chiavi pubbliche di tutti i nodi edge, così da poter verificare l'autenticità dei messaggi ricevuti.

La distribuzione delle chiavi pubbliche avviene attraverso un canale testuale dedicato, ospitato su un server Discord, lo stesso utilizzato per le notifiche di sistema.

Flusso di distribuzione

1. In ogni nodo vengono generate le chiavi ECC come illustrato in 5.7.1 e viene condivisa solo la **chiave pubblica** nel canale Discord dedicato;
2. Sul cloud vengono scaricate le chiavi pubbliche di tutti gli edge (`ecc_public_edgeX.pem`) e collocate nella directory in cui è presente lo script `subscriber.py`;
3. Su ciascun nodo edge viene scaricata la chiave pubblica del cloud (`ecc_public_cloud.pem`), che viene posizionata nella directory in cui è presente lo script `publisher.py`.

```
lorenzofuse@Ubuntu-EdgeCoreVM3:~/kubeeedge_microservices/app$  
.  
└── CO2_Adeunis.xlsx  
└── ecc_private_edge3.pem  
└── ecc_public_cloud.pem  
└── ecc_public_edge3.pem  
└── generator.py  
└── publisher.py  
└── temperatures_Adeunis.xlsx  
  
lorenzofuse@Ubuntu-VMCloudCore:~/kubeeedge_microservices/app$  
.  
└── ecc_private_cloud.pem  
└── ecc_public_cloud.pem  
└── ecc_public_edge2.pem  
└── ecc_public_edge3.pem  
└── ecc_public_edge.pem  
└── generator.py  
└── subscriber.py
```

Figura 5.23: Struttura directory

Caricamento nel codice

Per poter applicare le operazioni crittografiche, gli script Python necessitano dell'accesso alle chiavi generate e distribuite come mostrato in precedenza. In fase di avvio, vengono caricate le chiavi utili al fine di firmare, cifrare o decifrare i dati in coerenza con il ruolo svolto (*publisher* o *subscriber*).

Il *publisher* carica la propria **chiave privata** e la **chiave pubblica del cloud** dai percorsi previsti (/app/...):

```
1 with open("/app/ecc_private_edge2.pem", "rb") as f:
2     priv_edge = serialization.load_pem_private_key(f.read(), 
3                                                 None)
4
5 with open("/app/ecc_public_cloud.pem", "rb") as f:
6     pub_cloud = serialization.load_pem_public_key(f.read())
```

Questa scelta permette al nodo edge di:

1. firmare il testo cifrato con la propria chiave privata,
2. derivare la chiave di sessione ECDH/ECDHE usando la chiave pubblica del cloud.

Parallelamente nel *subscriber* si caricano le chiavi necessarie per l'operazione di decifratura: la propria chiave privata e le chiavi pubbliche di tutti i nodi edge.

```
1
2 with open("/app/ecc_private_cloud.pem", "rb") as f:
3     priv_cloud = serialization.load_pem_private_key(f.read(), 
4                                                 None)
5
6 pub_keys = {
7     "edge":    serialization.load_pem_public_key(
8                 open("/app/ecc_public_edge.pem", "rb").read()
9                 ),
10    "edge2":   serialization.load_pem_public_key(
11                 open("/app/ecc_public_edge2.pem", "rb").read()
12                 ),
13    "edge3":   serialization.load_pem_public_key(
14                 open("/app/ecc_public_edge3.pem", "rb").read()
15                 )
16 }
```

Il dizionario **pub_keys** associa un identificativo **sender** (presente nel payload) alla corrispondente chiave pubblica. Questa operazione è fondamentale per la verifica ECDSA, in quanto un messaggio ricevuto da un **sender** non riconosciuto viene scartato a priori.

5.7.3 Struttura del payload crittografato

Ogni messaggio generato dai microservizi IAQ non viene inviato in chiaro, ma incapsulato in un payload JSON che contiene sia i dati cifrati sia i metadati necessari al *subscriber* per ricostruire la chiave di sessione ed effettuare la verifica. La struttura è la seguente:

- **chipertext**: contiene il vettore di inizializzazione IV concatenato al testo cifrato, entrambi codificati in Base64;
- **signature**: firma ECDSA del testo cifrato, generato con la chiave privata dell'edge;
- **ephemeral_pubkey**: chiave pubblica effimera generata ad ogni messaggio, necessaria al *subscriber* per derivare la chiave simmetrica AES;
- **sender**: identificativo del noto mittente (edge, edge2, edge3).

```

1 SENDER = "edge2"
2 #...
3     return json.dumps({
4         "ciphertext": base64.b64encode(iv + encrypted).decode()
5             (),
6         "signature": base64.b64encode(signature).decode(),
7         "ephemeral_pubkey": eph.public_key().public_bytes(
8             serialization.Encoding.PEM,
9             serialization.PublicFormat.SubjectPublicKeyInfo
10            ).decode(),
11         "sender": SENDER
12     })

```

5.7.4 Cifratura e firma lato Edge

Lato edge, i valori letti dal dataset BuildPred vengono incapsulati e protetti dalla funzione `encrypt_and_sign()`, che realizza i seguenti passaggi:

1. **Generazione di una chiave effimera**: ad ogni messaggio viene generata una chiave privata ECC di sessione.
2. **Derivazione della chiave simmetrica**: tramite ECDH e HKDF si ottiene una chiave AES-256 bit.
3. **Cifratura e firma**: il messaggio è cifrato con AES in modalità CFB e firmato con la chiave privata dell'edge.

```

1 def encrypt_and_sign(message: str) -> str:
2     #1. chiave effimera
3     eph      = ec.generate_private_key(ec.SECP384R1())
4     #segreto condiviso con il Cloud
5     shared   = eph.exchange(ec.ECDH(), pub_cloud)
6     #2. derivazione chiave AES
7     aes_key = HKDF(hashes.SHA256(), 32, None, b'handshake').
8         derive(shared)
9     #IV casuale
10    iv = os.urandom(16)
11    cipher = Cipher(algorithms.AES(aes_key), modes.CFB(iv))
12    encrypted = cipher.encryptor().update(message.encode())

```

```

13
14 #3. firma ECDSA
15 signature = priv_edge.sign(encrypted, ec.ECDSA(hashes.
16                           SHA256()))
17
18 return json.dumps({
19     "ciphertext": base64.b64encode(iv + encrypted).decode()
20     (),
21     "signature": base64.b64encode(signature).decode(),
22     "ephemeral_pubkey": eph.public_key().public_bytes(
23         serialization.Encoding.PEM,
24         serialization.PublicFormat.SubjectPublicKeyInfo).
25         decode(),
26     "sender": SENDER
27 })

```

La presenza della chiave pubblica effimera permette al *subscriber* di ricostruire la stessa chiave simmetrica senza che venga trasmessa.

5.7.5 Decifratura e verifica lato Cloud

Il microservizio **subscriber**, riceve i messaggi dai topic `iaq/#` e li passa alla funzione `decrypt_and_verify()`. Questa funziona si occupa di effettuare il percorso inverso rispetto al *publisher*:

1. **Parsing del payload:** viene decodificato il JSON e identificato il mittente.
2. **Ricostruzione della chiave simmetrica:** usando la chiave privata del cloud e la chiave pubblica effimera inclusa nel payload, si ottiene la stessa chiave AES-256 utilizzata per la cifratura.
3. **Decifratura:** il testo cifrato viene decifrato con AES in modalità CFB.
4. **Verifica firma:** la firma viene controllata con la chiave pubblica del mittente per assicurare autenticità e integrità.

```

1 def decrypt_and_verify(payload_json):
2     obj          = json.loads(payload_json)
3     ciphertext   = base64.b64decode(obj["ciphertext"])
4     signature    = base64.b64decode(obj["signature"])
5     sender       = obj.get("sender")
6
7     if sender not in pub_keys:
8         raise ValueError(f"Sender '{sender}' sconosciuto")
9
10    ephemeral   = serialization.load_pem_public_key(obj[""
11                                                 "ephemeral_pubkey"].encode())
12    shared       = priv_cloud.exchange(ec.ECDH(), ephemeral)
13    aes_key     = HKDF(hashes.SHA256(), 32, None, b'handshake')
14    .derive(shared)

```

```

13      iv, encrypted = ciphertext[:16], ciphertext[16:]      #
14      separazione IV
15      cipher      = Cipher(algorithms.AES(aes_key), modes.CFB(iv))
16      clear       = cipher.decryptor().update(encrypted)
17
18      pub_keys[sender].verify(signature, encrypted, ec.ECDSA(
19          hashes.SHA256())) # verifica firma
20
21      return clear.decode()

```

Se la firma è valida e la decifratura ha successo, il contenuto viene salvato nel database. In caso di errore (firma non valida, mittente sconosciuto, formato errato), il messaggio viene rifiutato e scartato, garantendo la consistenza delle tabelle nel database.

5.8 Persistenza dei dati - PostgreSQL

Il passo successivo al processo di decifratura e verifica è la memorizzazione dei messaggi all'interno del database, così da consentirne l'analisi e la visualizzazione tramite dashboard. Nel progetto è stato adottato PostgreSQL, installato sul nodo CloudCore, che centralizza sia i dati ambientali provenienti dai microservizi IAQ, sia le metriche relative all'utilizzo delle risorse delle VM.

5.8.1 Installazione e preparazione dell'ambiente

L'installazione è avvenuta tramite i pacchetti ufficiali di Ubuntu, seguita dalla creazione del database e di un utente dedicato:

```
sudo apt install postgresql postgresql-contrib -y
```

Eseguire l'accesso a PostgreSQL:

```
sudo -u postgres psql
```

All'interno della shell `psql` si procede con la creazione del database per l'utente con il relativo schema dedicato, per ridurre i conflitti di permessi:

```

CREATE DATABASE iaq_db;
CREATE USER iaq_user WITH PASSWORD 'iaq_pass';
GRANT ALL PRIVILEGES ON DATABASE iaq_db TO iaq_user;

CREATE SCHEMA iaq_schema AUTHORIZATION iaq_user;

```

5.8.2 Definizione delle tabelle

Per organizzare i dati che il broker riceve da tutti i nodi edge sono state create due tabelle, ciascuna con un ruolo specifico.

La prima tabella, `iaq_data`, raccoglie i valori ambientali (temperatura e CO₂) delle varie zone dell'edificio. Essa include i riferimenti temporali (timestamp di arrivo e timestamp di pubblicazione del sensore nella simulazione), il topic MQTT di provenienza, il tipo di misurazione (TEMP o CO2) e il valore effettivo, oltre alla latenza calcolata tra la pubblicazione e la ricezione.

```
CREATE TABLE iaq_schema.iaq_data (
    id SERIAL PRIMARY KEY ,
    ts_arrivo TIMESTAMP ,
    topic TEXT ,
    ts_sensore TIMESTAMP ,
    tipo TEXT ,
    valore FLOAT ,
    latenza_pub_sub FLOAT
);
```

La seconda tabella, `vm_metrics`, è dedicata alle risorse di sistema delle VM (cloud + edge). Le colonne memorizzano timestamp, CPU, RAM, disco e rete, permettendo di confrontare l'utilizzo delle risorse tra i vari nodi nel tempo.

```
CREATE TABLE iaq_schema.vm_metrics (
    timestamp      TIMESTAMP NOT NULL ,
    hostname       TEXT ,
    cpu_percent    REAL ,
    ram_used_mb   INTEGER ,
    ram_total_mb  INTEGER ,
    disk_used_gb  REAL ,
    net_in_kbps   REAL ,
    net_out_kbps  REAL
);
```

5.8.3 Connessione al database

Il primo passo per consentire l'inserimento dei dati consiste nella connessione a PostgreSQL tramite la libreria `psycopg2`. Per stabilire la connessione vengono forniti i parametri principali come segue:

```
1 conn = psycopg2.connect(
2     dbname="iaq_db",
3     user="iaq_user",
4     password="iaq_pass",
5     host="localhost"
6 )
7 cursor = conn.cursor()
```

L'oggetto `cursor`, ottenuto dalla connessione, rappresenta l'interfaccia attraverso cui è possibile eseguire i comandi SQL. Ogni query inviata al database passa attraverso di esso: dalla creazione e modifica delle tabelle, fino all'inserimento dei dati durante l'esecuzione del microservizio.

Inizializzazione della tabella

All'avvio del microservizio, la tabella `iaq_data` viene svuotata tramite TRUNCATE. Si è scelto di operare in questo verso per evitare di introdurre dati duplicati nelle dashboard di Grafana e mantenere una coerenza con il dataset.

```
1 cursor.execute("TRUNCATE TABLE iaq_schema.iaq_data;")  
2 conn.commit()
```

5.8.4 Inserimento dei dati ambientali

Una volta decifrati e verificati i dati dal *subscriber*, i dati vengono inseriti sia all'interno della tabella `iaq_data` sia all'interno di un file CSV `iaq_log.csv` per logging.

```
1 try:  
2     #....  
3     with open(LOG_FILE, 'a', newline='') as f:  
4         writer = csv.writer(f)  
5         writer.writerow([ts_arrivo, ts_pub, topic,  
6                         ts_sensore, tipo.strip(), valore, latenza])  
7  
8     cursor.execute(  
9         """  
10            INSERT INTO iaq_schema.iaq_data (  
11                ts_arrivo, topic, ts_sensore,  
12                tipo, valore, latenza_pub_sub)  
13                VALUES (%s, %s, %s, %s, %s, %s)  
14            """ , (ts_arrivo, topic, ts_sensore, tipo.strip(),  
15                         valore, latenza))  
16     conn.commit()  
17 except Exception as e:  
18     print(f"Errore parsing/decrittazione/firma: {e}  
19           Messaggio grezzo: {msg.payload[:80]}")  
20     conn.rollback()
```

5.8.5 Persistenza delle metriche di sistema

Oltre ai dati ambientali, il sistema raccoglie le metriche relative all'utilizzo delle risorse come precedentemente spiegato. Anche in questo caso, lo script Python, `vm-metrics-subscriber.py`, presente all'interno del microservizio dedicato, si occupa di ricevere i messaggi pubblicati via MQTT e inserirli nella tabella `iaq_schema.vm_metrics`

La connessione al database è la medesima mostrata in precedenza nella Sezione 5.8.3.

Parsing dei messaggi

Ogni messaggio ricevuto dal topic `vm/metrics/#` viene decodificato dal formato JSON, estraendo i campi rilevanti: timestamp, hostname, utilizzo CPU, memoria, disco e rete.

```
1 #payload generato dal publisher delle metriche
2 payload = {
3     "timestamp": time.strftime('%Y-%m-%d %H:%M:%S') ,
4     "hostname": socket.gethostname() ,
5     "cpu_percent": cpu,
6     "ram_used_mb": ram.used // (1024 * 1024) ,
7     "ram_total_mb": ram.total // (1024 * 1024) ,
8     "disk_used_gb": round(disk.used / (1024**3), 2) ,
9     "net_in_kbps": round(net.bytes_recv / 1024, 2),
10    "net_out_kbps": round(net.bytes_sent / 1024, 2)
11 }
12
13 #parsing effettuato dal subscriber
14 def on_message(client, userdata, msg):
15     try:
16         data = json.loads(msg.payload.decode())
17
18         cursor.execute("""
19             INSERT INTO iaq_schema.vm_metrics (
20                 timestamp, hostname, cpu_percent,
21                 ram_used_mb, ram_total_mb, disk_used_gb,
22                 net_in_kbps, net_out_kbps
23             )
24             VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
25         """, (
26             data["timestamp"],
27             data["hostname"],
28             data["cpu_percent"],
29             data["ram_used_mb"],
30             data["ram_total_mb"],
31             data["disk_used_gb"],
32             data["net_in_kbps"],
33             data["net_out_kbps"]
34         ))
35         conn.commit()
36
37     except Exception as e:
38         print("Errore nel salvataggio su DB:", e)
```

L'inserimento all'interno del database tramite il cursore avviene solamente se il formato è completo e valido. In caso di errore il dato non viene inserito per evitare incongruenze nello storico delle prestazioni, e della successiva visualizzazione tramite dashboard delle prestazioni delle VM.

5.9 Visualizzazione e dashboard - Grafana

La fase conclusiva del flusso dati prevede la loro visualizzazione tramite **Grafana**, che consente di creare dashboard interattive collegate a diversi *data source*, tra cui PostgreSQL.

5.9.1 Installazione

L'installazione è avvenuta sul nodo cloud utilizzando i pacchetti ufficiali messi a disposizione dal repository di Grafana. Dopo aver aggiunto il repository e la chiave GPG, l'installazione e l'attivazione del servizio si effettuano con i seguenti comandi:

```
sudo apt install -y software-properties-common

# importazione chiave GPG e repository
sudo wget -q -O - https://packages.grafana.com/gpg.key \
| sudo gpg --dearmor -o /usr/share/keyrings/grafana.gpg

echo "deb [signed-by=/usr/share/keyrings/grafana.gpg] \
https://packages.grafana.com/oss/deb stable main" \
| sudo tee /etc/apt/sources.list.d/grafana.list > /dev/null

# installazione e avvio del servizio
sudo apt install -y grafana
sudo systemctl start grafana-server
sudo systemctl enable grafana-server
sudo systemctl status grafana-server
```



```
lorenzofuse@Ubuntu-VMCloudCore:~$ sudo systemctl status grafana-server
● grafana-server.service - Grafana instance
  Loaded: loaded (/usr/lib/systemd/system/grafana-server.service; enabled; preset: enabled)
  Active: active (running) since Mon 2025-09-08 13:56:29 CEST; 1h 42min ago
    Docs: http://docs.grafana.org
   Main PID: 2237 (grafana)
      Tasks: 18 (limit: 6845)
     Memory: 319.9M (peak: 350.3M)
        CPU: 5min 21.806s
       CGroupl: /system.slice/grafana-server.service
                  └─2237 /usr/share/grafana/bin/grafana server --config=/etc/grafana/grafana.ini --pid=
```

Figura 5.24: Verifica del servizio Grafana

5.9.2 Accesso e configurazione

In seguito all'abilitazione di Grafana, il prossimo passo è collegare il database come *data source*, in modo da poter interrogare e visualizzare i dati raccolti dai microservizi.

Nella schermata di configurazione vengono forniti i parametri relativi alla connessione al database:

Campo	Valore
Name	iaq-postgres
Host	localhost:5432
Database	iaq_db
User	iaq_user
Password	iaq_pass
SSL mode	require

Tabella 5.2: Configurazione della connessione PostgreSQL in Grafana.

Una volta inseriti i parametri, si procede con il salvataggio e la verifica tramite il pulsante **Save & Test**. In caso di successo, Grafana conferma l'avvenuta connessione al database, consentendo di costruire le dashboard basate sulle tabelle **iaq_data** e **vm_metrics**.

Per garantire chiarezza e facilità di interpretazione, le dashboard sono state organizzate in due cartelle principali

- **IAQ:** dedicate al monitoraggio della qualità dell'aria (temperatura e CO₂) per l'intera struttura e per ciascuna zona/stanza;
- **VM METRICS:** dedicate alle risorse di sistema delle macchine virtuali coinvolte nel progetto.

All'interno di ciascuna cartella, le dashboard sono state strutturate con un livello **aggregato**, che forniscono una panoramica generale, e un livello **dettagliato**, che consentono di analizzare singolarmente i valori registrati per ogni stanza o VM.

Tutti i pannelli condividono alcune scelte comuni:

- **Pannelli di sintesi:** i primi due pannelli forniscono una visuale complessiva sull'andamento di tutte le stanze monitorate in questa zona, con valori medi aggregati su intervalli di un minuto.
- **Pannelli di dettaglio:** ogni pannello utilizza una query dedicata al singolo topic (es. `iaq/zone1/meeting1`), con la stessa logica vista per le medie, ma senza aggregazione sul minuto, così da ottenere l'intera serie temporale nella stanza.

La logica di interrogazione nelle varie dashboard temporali rimane identica per tutte le zone, variando unicamente nel filtro sul campo **topic** (es. `iaq/zone1/%`, `iaq/zone2/%`).

5.9.3 Indoor Air Quality - Overview

Le dashboard **Indoor Air Quality - Overview** forniscono una rappresentazione sintetica sull'intero edificio dei parametri ambientali: I primi due pannelli riportano tutte le misurazioni di temperatura e CO₂ effettuate nelle varie stanze, mentre quelli sottostanti mostrano le stesse misure aggregate su intervalli di un minuto. In ciascun pannello sono riportate delle **soglie di riferimento**, configurate all'interno del pannello Grafana, permettendo di interpretare rapidamente la qualità dell'ambiente, distinguendo condizioni normali da potenziali valori non conformi.

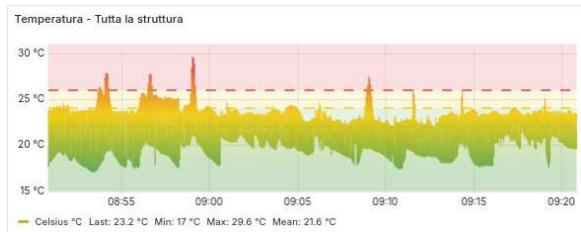


Figura 5.25: Temperatura rilevata su tutta la struttura

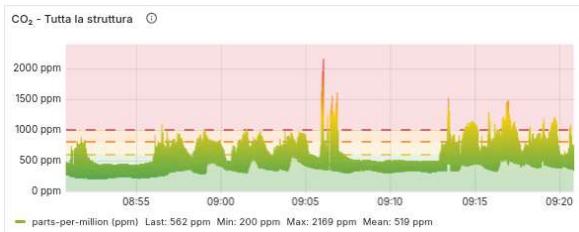


Figura 5.26: CO₂ rilevato su tutta la struttura

```
SELECT ts_arrivo AS "time",
       valore, topic
  FROM iaq_schema.iaq_data
 WHERE tipo='TEMP'
 ORDER BY ts_arrivo
```

```
SELECT ts_arrivo AS "time",
       valore, topic
  FROM iaq_schema.iaq_data
 WHERE tipo='CO2'
 ORDER BY ts_arrivo
```

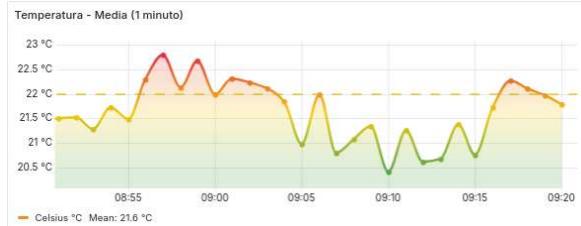


Figura 5.27: Rilevazioni aggregate della Temperatura



Figura 5.28: Rilevazioni aggregate di CO₂

```
SELECT
    date_trunc('minute',
               ts_arrivo) AS "time",
    avg(valore) as valore
  FROM iaq_schema.iaq_data
 WHERE tipo='TEMP'
 GROUP BY time
 ORDER BY time
```

```
SELECT
    date_trunc('minute',
               ts_arrivo) AS "time",
    avg(valore) as valore
  FROM iaq_schema.iaq_data
 WHERE tipo='CO2'
 GROUP BY time
 ORDER BY time
```

5.9.4 Zona 1

Per finalità di implementazione viene riportato come esempio il caso della **Zona 1**, che mostra la logica di costruzione delle dashboard IAQ nelle varie zone. Le altre zone (Zone 2, Zone 3) seguono una struttura identica, variando unicamente nel filtro applicato sul campo `topic` (es. `iaq/zone2/%`, `iaq/zone3/%`, `iaq/zone3/window`, ecc.) all'interno della query.

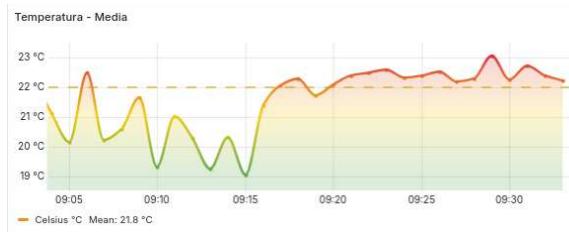


Figura 5.29: Temperatura media | Zone 1



Figura 5.30: CO₂ media | Zone 1

```
SELECT
    date_trunc('minute',
        ts_arrivo) AS "time",
    AVG(valore) AS valore
FROM iaq_schema.iaq_data
WHERE tipo = 'TEMP' AND
    topic LIKE 'iaq/zone1/%'
GROUP BY time
ORDER BY time
```

```
SELECT
    date_trunc('minute',
        ts_arrivo) AS "time",
    AVG(valore) AS valore
FROM iaq_schema.iaq_data
WHERE tipo = 'CO2' AND topic
    LIKE 'iaq/zone1/%'
GROUP BY time
ORDER BY time
```

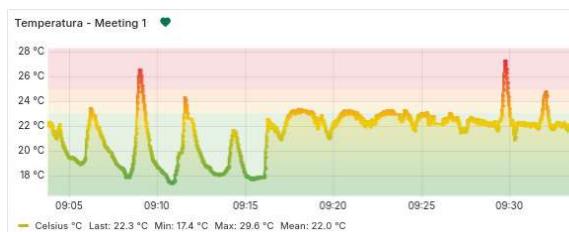


Figura 5.31: Temperatura | Meeting 1

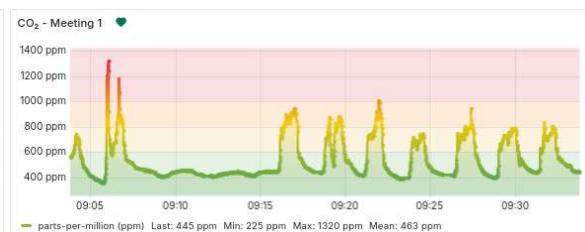


Figura 5.32: CO₂ | Meeting 1

```
SELECT
    ts_arrivo AS "time",
    valore
FROM iaq_schema.iaq_data
WHERE tipo = 'TEMP' AND topic =
    'iaq/zone1/meeting1'
ORDER BY ts_arrivo
```

```
SELECT
    ts_arrivo AS "time",
    valore
FROM iaq_schema.iaq_data
WHERE tipo = 'CO2' AND topic =
    'iaq/zone1/meeting1'
ORDER BY ts_arrivo
```



Figura 5.33: Temperatura | Meeting 2

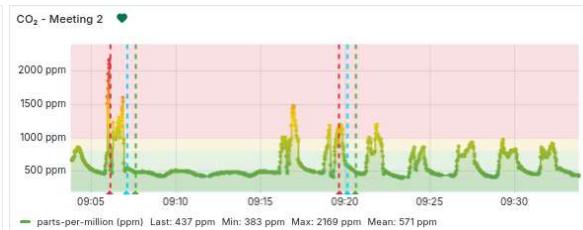


Figura 5.34: CO₂ | Meeting 2

```
SELECT
    ts_arrivo AS "time",
    valore
FROM iaq_schema.iaq_data
WHERE tipo = 'TEMP' AND topic =
    'iaq/zone1/meeting2'
ORDER BY ts_arrivo
```

```
SELECT
    ts_arrivo AS "time",
    valore
FROM iaq_schema.iaq_data
WHERE tipo = 'CO2' AND topic =
    'iaq/zone1/meeting2'
ORDER BY ts_arrivo
```

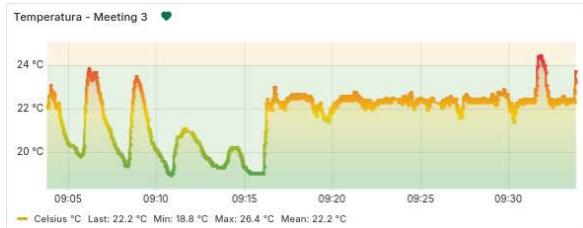


Figura 5.35: Temperatura | Meeting 3

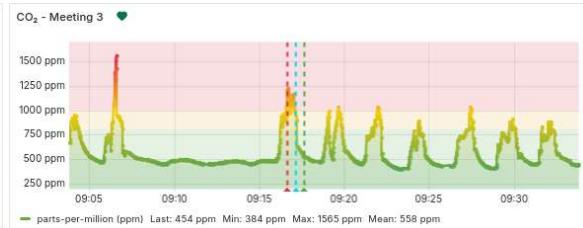


Figura 5.36: CO₂ | Meeting 3

```
SELECT
    ts_arrivo AS "time",
    valore
FROM iaq_schema.iaq_data
WHERE tipo = 'TEMP' AND topic =
    'iaq/zone1/meeting3'
ORDER BY ts_arrivo
```

```
SELECT
    ts_arrivo AS "time",
    valore
FROM iaq_schema.iaq_data
WHERE tipo = 'CO2' AND topic =
    'iaq/zone1/meeting3'
ORDER BY ts_arrivo
```

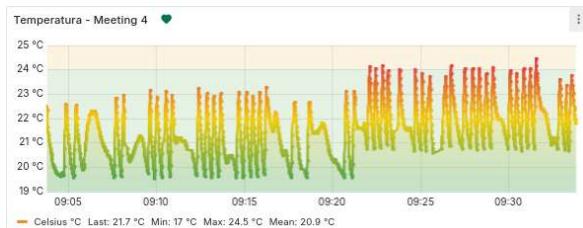


Figura 5.37: Temperatura | Meeting 4

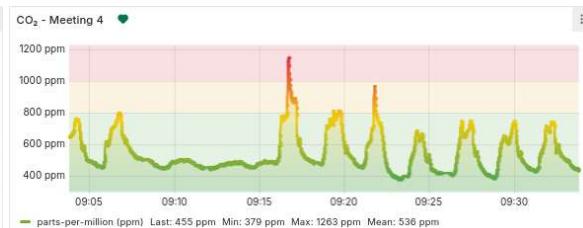


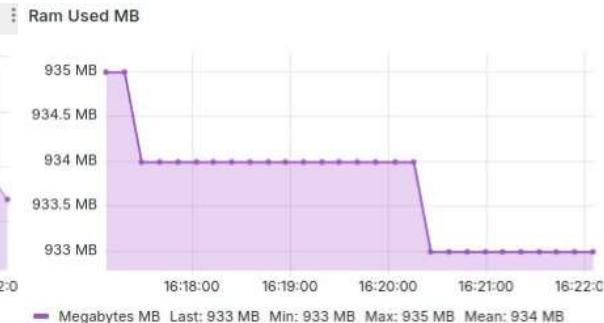
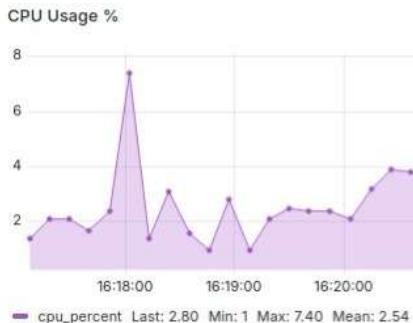
Figura 5.38: CO₂ | Meeting 4

```
SELECT
    ts_arrivo AS "time",
    valore
FROM iaq_schema.iaq_data
WHERE tipo = 'TEMP' AND topic =
    'iaq/zone1/meeting4'
ORDER BY ts_arrivo
```

```
SELECT
    ts_arrivo AS "time",
    valore
FROM iaq_schema.iaq_data
WHERE tipo = 'CO2' AND topic =
    'iaq/zone1/meeting4'
ORDER BY ts_arrivo
```

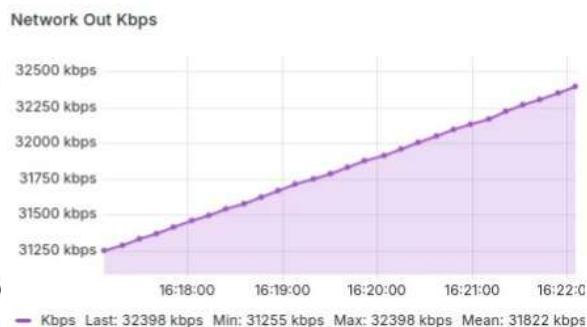
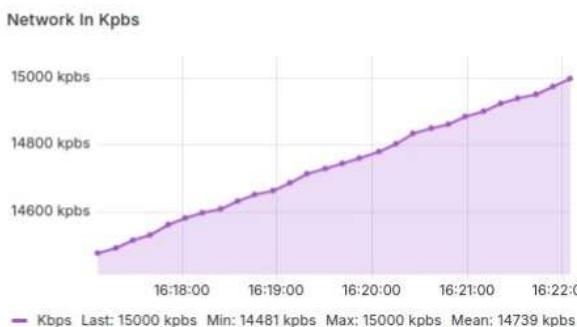
5.9.5 Dashboard VM Metrics

Le dashboard all'interno della cartella **VM Metrics** visualizzano l'andamento delle risorse di sistema (CPU, RAM, rete e disco) di ogni macchina virtuale. Ogni dashboard segue lo stesso layout: cinque pannelli *time series* con identica logica di interrogazione sul database, variando in questo caso unicamente sull'`hostname`. Di seguito vengono riportate le query SQL utilizzate, organizzate nello stesso layout della dashboard per chiarezza espositiva della VM Ubuntu-VMCloudCore.



```
SELECT timestamp AS "time",
       cpu_percent
  FROM iaq_schema.vm_metrics
 WHERE hostname = 'Ubuntu-
  VMCloudCore'
   AND $__timeFilter(
      timestamp)
 ORDER BY timestamp;
```

```
SELECT timestamp AS "time",
       ram_used_mb
  FROM iaq_schema.vm_metrics
 WHERE hostname = 'Ubuntu-
  VMCloudCore'
   AND $__timeFilter(
      timestamp)
 ORDER BY timestamp;
```



```
SELECT timestamp AS "time",
       net_in_kbps
  FROM iaq_schema.vm_metrics
 WHERE hostname = 'Ubuntu-
  VMCloudCore'
   AND $__timeFilter(
      timestamp)
 ORDER BY timestamp;
```

```
SELECT timestamp AS "time",
       net_out_kbps
  FROM iaq_schema.vm_metrics
 WHERE hostname = 'Ubuntu-
  VMCloudCore'
   AND $__timeFilter(
      timestamp)
 ORDER BY timestamp;
```

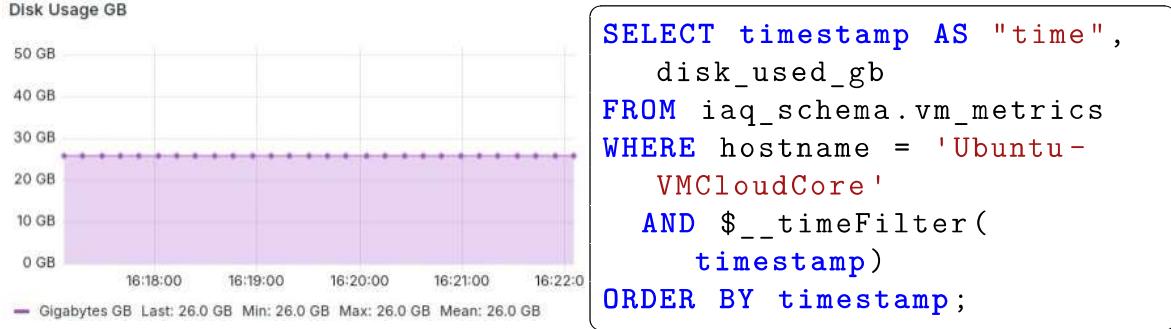


Figura 5.43: Disk Usage %

5.10 Build & Deploy dei microservizi

Dopo aver descritto nel Capitolo 3 l’architettura del sistema e le tecnologie adottate, in questa sezione si illustra l’effettiva modalità di distribuzione dei microservizi sui nodi del cluster **KubeEdge**.

Ogni componente è stato incapsulato in un **container**, costruito localmente e poi importato nel runtime **containerd**, così da essere avviato come **Pod statico** sui nodi Cloud ed Edge.

5.10.1 Verifica del cluster e organizzazione dei manifest

Come già anticipato nella Sezione 5.4.4, prima di procedere con la build e il successivo deploy dei microservizi, è necessario verificare il corretto funzionamento del cluster.

In particolare, dal nodo cloud si controlla lo stato dei nodi, in seguito all’operazione di join effettuate durante la configurazione, tramite il comando:

```
microk8s kubectl get nodes -o wide
```

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE	KERNEL-VERSION	COUNTAINER-RUNTIME
ubuntu-edgecorevm	Ready	agent,edge	126d	v1.30.7-kubeedge-v1.20.0	192.168.9.163	<none>	Ubuntu 24.04.2 LTS	6.14.0-29-generic	containerd://1.7.27
ubuntu-edgecorevm2	Ready	agent,edge	127d	v1.30.7-kubeedge-v1.20.0	192.168.9.127	<none>	Ubuntu 24.04.2 LTS	6.14.0-29-generic	containerd://1.7.27
ubuntu-edgecorevm3	Ready	agent,edge	127d	v1.30.7-kubeedge-v1.20.0	192.168.9.191	<none>	Ubuntu 24.04.2 LTS	6.14.0-29-generic	containerd://1.7.27
ubuntu-vmcloudcore	Ready	<none>	127d	v1.32.8	192.168.9.128	<none>	Ubuntu 24.04.2 LTS	6.14.0-29-generic	containerd://1.6.36

Figura 5.44: Verifica dei nodi nel cluster KubeEdge

Tutti i nodi devono risultare nello stato **Ready**, con hostname e indirizzo IP coerenti con quanto configurato. Per scelta organizzativa, tutti i file **YAML** dei Manifest Kubernetes sono stati posizionati in una directory unica **~/kubeedge_manifests** posizionata sul nodo cloud. I Manifest vengono creati manualmente (**sudo nano <manifest>.yaml**) ed applicati al cluster da questa cartella come mostrato in Figura 5.45.

```

lorenzofuse@Ubuntu-VMCloudCore:~/kubeeedge_manifests$ 
.
├── metric-publisher-cloudcore.yaml
├── metrics-publisher2.yaml
├── metrics-publisher3.yaml
├── metrics-publisher.yaml
├── metrics-subscriber.yaml
├── publisher-zone1.yaml
├── publisher-zone2.yaml
└── publisher-zone3.yaml
└── subscriber.yaml

```

Figura 5.45: Locazione dei file YAML

5.10.2 Costruzione delle immagini

Sebbene il runtime scelto per l'esecuzione dei container sia **Containerd**, la costruzione delle immagini è stata realizzata tramite **Docker**. La scelta è motivata dalla maggiore semplicità operativa che quest'ultimo offre, fornendo un toolchain integrato (`docker build`, `docker save`, `docker load`) che riduce i passaggi necessari rispetto a Containerd, il quale per la build necessita di strumenti aggiuntivi come `nerdctl` o `buildkit`, che richiedono un'installazione e configurazione separate.

Utilizzando Docker, il flusso operativo è stato il seguente:

1. costruzione dell'immagine con `docker build`;
2. esportazione in formato `.tar` tramite `docker save`;
3. importazione in Containerd con `ctr -n k8s.io images import`, così da renderla disponibile nel namespace usato da MicroK8s.

Esempio di Build

Vengono assegnati dei tag descrittivi in base al ruolo del servizio (publisher, subscriber, metrics) e alla zona ad esso assegnata nel caso dei nodi edge.

```

#ubuntu-edgecorevm2
cd ~/kubeeedge_microservices

#Publisher IAQ della Zone 1
sudo docker build -t iaq-publisher:zone1 .
sudo docker save -o iaq-publisher-zone1.tar iaq-publisher:
    zone1
sudo ctr -n k8s.io image import iaq-publisher-zone1.tar

```

```
#ubuntu-vmcloudcore
cd ~/kubeeedge_microservices

#Subscriber IAQ
sudo docker build -t iaq-subscriber:latest .
sudo docker save -o iaq-subscriber.tar iaq-subscriber:latest
microk8s.ctr image import iaq-subscriber.tar
```

```
#ubuntu-edgecorevm3
cd ~/kubeeedge-metrics/vm-metrics-publisher

#Publisher METRICS
sudo docker build -t metrics-publisher:latest .
sudo docker save -o metrics-publisher.tar metrics-publisher:
latest
sudo ctr -n k8s.io image import metrics-publisher.tar

#ubuntu-vmcloudcore
cd ~/kubeeedge-metrics/vm-metrics-subscriber

#Subscriber METRICS
sudo docker build -t metrics-subscriber:latest .
sudo docker save -o metrics-subscriber.tar metrics-subscriber
:latest
microk8s.ctr image import metrics-subscriber.tar
```

Questa procedura deve essere ripetuta per tutti i microservizi, rendendo così le immagini disponibili a Containerd senza utilizzare un registry remoto.

5.10.3 Definizione dei manifest Kubernetes

Dopo aver importato le immagini nei nodi, i microservizi vengono descritti tramite appositi Manifest YAML necessari per il loro deployment. Tutti i Manifest presentano delle scelte comuni, tra cui:

- **hostNetwork: true**: per permettere la comunicazione diretta con il broker MQTT (porta 1883) e con il database PostgreSQL (porta 5432, sul nodo Cloud);
- **nodeSelector**: per vincolare l'esecuzione al nodo corretto (cloud o edge) in base all'hostname della VM;
- **imagePullPolicy: IfNotPresent**: per forzare l'uso delle immagini importate in containerd in locale.

Subscriber IAQ - `subscriber.yaml`

Di seguito è riportato un esempio di Manifest per il microservizio che gestisce la raccolta dei dati IAQ sul nodo cloud:

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: subscriber
5 spec:
6   nodeSelector:
7     kubernetes.io/hostname: ubuntu-vmcloudcore
8   hostNetwork: true
9   containers:
10    - name: mqtt-subscriber
11      image: docker.io/library/iaq-subscriber:latest
12      imagePullPolicy: IfNotPresent
13      volumeMounts:
14        - name: log-volume
15          mountPath: /app/logs
16   volumes:
17    - name: log-volume
18      hostPath:
19        path: /var/log/mqtt
20        type: DirectoryOrCreate

```

Publisher IAQ - publisher-zoneX.yaml

La logica è identica per tutti i publisher, variando soltanto l'hostname del nodo e il tag dell'immagine.

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: iaq-publisher-zone1
5 spec:
6   nodeSelector:
7     kubernetes.io/hostname: ubuntu-edgecorevm2
8   hostNetwork: true
9   containers:
10    - name: publisher
11      image: iaq-publisher:zone1
12      imagePullPolicy: IfNotPresent
13      volumeMounts:
14        - name: log-volume
15          mountPath: /logs
16   volumes:
17    - name: log-volume
18      hostPath:
19        path: /var/log/mqtt
20        type: DirectoryOrCreate

```

Microservizi delle metriche

All'interno della cartella `kubeedge_manifests` sono presenti anche i Manifest dei publisher/subscriber dedicati al monitoraggio delle risorse delle VM. Seguono la stessa logica dei file precedenti, variando soltanto l'immagine e il nome del Pod.

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: metrics-subscriber
5 spec:
6   nodeSelector:
7     kubernetes.io/hostname: ubuntu-vmcloudcore
8   hostNetwork: true
9   containers:
10    - name: metrics-subscriber
11      image: metrics-subscriber:latest
12      imagePullPolicy: IfNotPresent
```

E per un publisher su un nodo Edge, `metrics-publisherX.yaml`:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: metrics-publisher
5 spec:
6   nodeSelector:
7     kubernetes.io/hostname: ubuntu-edgecorevm
8   hostNetwork: true
9   containers:
10    - name: metrics-publisher
11      image: metrics-publisher:latest
12      imagePullPolicy: IfNotPresent
```

5.10.4 Deploy dei manifest

Dopo aver definito i file Manifest, questi vengono applicati al cluster direttamente dal nodo cloud utilizzando i seguenti comandi:

```
microk8s kubectl apply -f subscriber.yaml
microk8s kubectl apply -f publisher-zone1.yaml
microk8s kubectl apply -f metrics-subscriber.yaml
microk8s kubectl apply -f metrics-publisher.yaml
```

Per verificare l'avvenuto deploy e lo stato dei Pod è possibile eseguire:

```
microk8s kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
iaq-publisher-zone1	1/1	Running	3 (157m ago)	23h	192.168.9.127	ubuntu-edgecorevm2
iaq-publisher-zone2	1/1	Running	1 (3h31m ago)	23h	192.168.9.163	ubuntu-edgecorevm
iaq-publisher-zone3	1/1	Running	2 (3h28m ago)	23h	192.168.9.191	ubuntu-edgecorevm3
metrics-publisher	1/1	Running	1 (3h31m ago)	23h	192.168.9.163	ubuntu-edgecorevm
metrics-publisher-cloudcore	1/1	Running	1 (3h37m ago)	23h	192.168.9.128	ubuntu-vmcloudcore
metrics-publisher-vm2	1/1	Running	3 (157m ago)	23h	192.168.9.127	ubuntu-edgecorevm2
metrics-publisher-vm3	1/1	Running	2 (3h28m ago)	23h	192.168.9.191	ubuntu-edgecorevm3
metrics-subscriber	1/1	Running	1 (3h37m ago)	23h	192.168.9.128	ubuntu-vmcloudcore
subscriber	1/1	Running	1 (3h37m ago)	23h	192.168.9.128	ubuntu-vmcloudcore

Figura 5.46: Verifica dei Pod

5.10.5 Troubleshooting

Durante la fase di deploy possono emergere errori dovuti non solo a configurazioni errate nei Manifest, ma anche a problemi nei vari file creati in precedenza (Dockerfile, script Python, mancanze di librerie), risolvibili con dei controlli di configurazione sull'errore indicato.

- **ImagePullBackOff - Immagine non trovata:**

verificare che l'immagine sia stata importata correttamente nel runtime Containerd:

```
sudo ctr -n k8s.io image ls | grep nome
```

- **Pod Pending:** controllare il campo nodeSelector e che il nodo risulti Ready;
- **Connessioni rifiutate con hostNetwork: true** i servizi devono essere in ascolto sull'IP corretto. Verificare la configurazione nella Sezione 5.2.4;
- **Log vuoti** controllare i volumi su /var/log/mqtt e i permessi della directory.

5.10.6 Aggiornamenti e rollback

In fase di sviluppo è spesso necessario aggiornare i microservizi. La procedura dei comandi da effettuare è la seguente:

1. ricostruire l'immagine con nuovo o stesso tag;
2. esportarla e importarla in Containerd sui nodi interessati;
3. aggiornare il Manifest in caso di modifica del tag e rieseguire
`microk8s kubectl apply -f <manifest>.yaml`

Prima di procedere con il deploy del microservizio modificato, è necessario eliminare quello precedentemente creato e riapplicare il Manifest:

```
microk8s kubectl delete pod <nome>
microk8s kubectl apply -f <manifest>.yaml
```

5.10.7 Verifica tramite log

Oltre ai log nativi di Kubernetes, si è predisposto il container in modo tale da poter monitorare i vari messaggi pubblicati o ricevuti dai vari nodi:

- i publisher salvano in `/var/log/mqtt/iaq_pub.log` i messaggi cifrati inviati;
- i subscriber registrano in `/var/log/mqtt/iaq_log.csv` i messaggi ricevuti.

```
tail -f /var/log/mqtt/iaq_pub.log
tail -f /var/log/mqtt/iaq_log.csv
```

```
lorenzofuse@Ubuntu-EdgeCoreVM3: $ tail -f /var/log/mqtt/iaq_pub.log
{"ciphertext": "eGF6Vhd7xF1wEYD5M595bTv8nSye1Wp+uuA002EN+YKczTtpsCT97838XjhH+usH+MEHQFmKXKW+JrjqGd3/u7FzV2X6eDpM5jYWofhqw=", "signature": "MGUCMQC0+e1b90A6Eam3Bnm/X1vu3Ri23Q0e8/OUJH78z8INRo1RmwAss6IqjYRwlXTcECMASJCxs1e4smXDv9y001BzNztqZ3rSOC15Fa1vFYRYYYlhEUradJuAwnPrynP0sxrA==", "ephemeral_pubkey": "-----BEGIN PUBLIC KEY-----\nMHYwEAYHKoZIzj0CAQYFK4EEACIDYgAEexEUdoaIafA/J4QCmNF7oaQxfpSDVS1E\nyX0PS/rsw32exVEIy43/t2wNOP6Hv8+l50aU+wBzkFJDkhzLsvX+vFqaG02QGd\nnPyRdiwKZhC4BUMwafIk8LF6ye+iw/oJd\n-----END PUBLIC KEY-----\n", "sender": "edge3"}
{"ciphertext": "atCdLMNUBn8n+b8L3PErzniP/CksgWtT52ejwKCqLcd0jRhkezVxEBdFnkBc66iXkhcYbT637n80Xna+qpjBvn6IpNu16l1dcvdNYnePA=", "signature": "MGUCMH+xPVasAC94a4xBEL8zBxMyCrZgy70fEhw5gZm0ZBxsnxic8JYiBd84PAFwHty4wIxAnkgjSBHQxSIVTrEt43o0bzCECWA/1L2kp+MiVtkErE7pikk10UTM10Z8UyNCC3Hw==", "ephemeral_pubkey": "-----BEGIN PUBLIC KEY-----\nMHYwEAYHKoZIzj0CAQYFK4EEACIDYgAEkaolQKqU12kh55lpDoy+Q0+6rSFRLh\nnn3GvL578w54zwFLM3cjwhdkfTEK/IAw5cn3HS3Z7d7lkVmK8WRzpCi4QUFaPwPj9/noeE1XB6D+/ti1FbpxctGTj7b9PDj521\n-----END PUBLIC KEY-----\n", "sender": "edge3"}
```

(a) Log dei messaggi pubblicati.

```
lorenzofuse@Ubuntu-VMCloudCore: $ tail -f /var/log/mqtt/iaq_log.csv
2025-09-11 16:12:42.163587,2025-09-11 16:12:42.127153,iaq/zone3/break_room,2025-09-11 16:12:42.127153,C02,638.64,0.036434
2025-09-11 16:12:42.169575,2025-09-11 16:12:42.128938,iaq/zone3/break_room,2025-09-11 16:12:42.128938,TEMP,22.35,0.040637
2025-09-11 16:12:42.719846,2025-09-11 16:12:42.666416,iaq/zone1/meeting1,2025-09-11 16:12:42.666416,C02,683.55,0.05343
2025-09-11 16:12:42.750046,2025-09-11 16:12:42.672606,iaq/zone1/meeting1,2025-09-11 16:12:42.672606,TEMP,24.45,0.08344
2025-09-11 16:12:42.780276,2025-09-11 16:12:42.687567,iaq/zone1/meeting2,2025-09-11 16:12:42.687567,C02,821.64,0.092709
2025-09-11 16:12:42.787365,2025-09-11 16:12:42.696007,iaq/zone1/meeting2,2025-09-11 16:12:42.696007,TEMP,23.51,0.091358
2025-09-11 16:12:42.796619,2025-09-11 16:12:42.699857,iaq/zone1/meeting3,2025-09-11 16:12:42.699857,C02,698.36,0.096762
2025-09-11 16:12:42.803673,2025-09-11 16:12:42.702759,iaq/zone1/meeting3,2025-09-11 16:12:42.702759,TEMP,23.85,0.100914
2025-09-11 16:12:42.817105,2025-09-11 16:12:42.705246,iaq/zone1/meeting4,2025-09-11 16:12:42.705246,C02,710.36,0.111859
2025-09-11 16:12:42.825958,2025-09-11 16:12:42.707248,iaq/zone1/meeting4,2025-09-11 16:12:42.707248,TEMP,20.46,0.11871
```

(b) Log dei messaggi ricevuti.

Figura 5.47: Monitoraggio in tempo reale via `tail -f`: publisher (sopra) e subscriber (sotto).

Nota sulla scelta architetturale

Nel progetto i microservizi sono stati eseguiti come **Pod statici**, ovvero definiti direttamente tramite Manifest YAML sui nodi in questo modo:

```
1 apiVersion: v1
2 kind: Pod / Deployment / StatefulSet
```

Tuttavia, si potrebbero usare altre modalità di tipo **Deployment** (per garantire la ricreazione automatica dei pod in caso di crash) o **StatefulSet** (per mantenere identità e volumi persistenti dei pod).

Nel caso specifico del **subscriber**, lo script Python è stato progettato per eseguire un TRUNCATE iniziale sulla tabella `iaq_data` in PostgreSQL, così che le dashboard Grafana riflettano soltanto i dati della sessione corrente, senza sovrapposizioni con inserimenti precedenti.