UNIVERSITÀ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Advanced Computer Programming**

Monitoraggio dell'orchestratore Kubernetes con Prometheus

Anno Accademico 2023/2024

Candidato
Marcello Esposito
matr. N46006315

Una dedica...

a tutte le persone che sono sempre state al mio fianco, alla mia famiglia che ha continuamente mantenuto in salute il giardino intorno a me per permettermi di sbocciare;

alla mia fidanzata che mi ha dato qualcosa in più di una normale relazione, qualcosa che è difficile da spiegare a parole, crescere insieme è un qualcosa di diverso;

all'angolo della Massellanza, che avete cambiato completamente il modo in cui ho vissuto l'università (e aumentato notevolmente il cibo mangiato);

a Deloitte, che mi ha dato la possibilità di complementare la mia crescita sia lavorativamente che personalmente;

a Napoli Est e tutti i ragazzi di Operazione Talenti, nei quali ho trovato qualcosa di molto più di semplici colleghi;

ed infine... Forza Napoli Sempre.

Indice

Introduzione

Nei sistemi moderni, il deployment delle applicazioni si basa sempre più sui container. Tuttavia gestire il ciclo di vita di numerosi container, richiede molte operazioni ripetitive che possono essere automatizzate utilizzando gli orchestratori di container, mentre per mantenere sotto controllo lo stato di salute del sistema sono stati creati gli strumenti di monitoraggio.

Obiettivo della tesi

L'obiettivo di questo elaborato è di approfondire il monitoraggio di **Kubernetes** [6] attraverso **Prometheus** [7], sfruttando il contesto del progetto *Mutiny* [16], nel quale è stata condotta una campagna di fault / error injection su un cluster Kubernetes con target l'etcd, il database key-value utilizzato per gestire lo stato del cluster.
In particolare è stata approfondita la correlazione tra i tipi di errori causati dalla campagna ed un subset delle metriche ottenute attraverso il tool di monitoraggio **Prometheus** sul cluster, in maniera tale da poter poi dimostrare come sia possibile ricavare delle apposite regole

di **alerting** per prevenire l'indisponibilità dei servizi offerti.

Struttura dell'elaborato

Questo elaborato è suddiviso in diversi capitoli, dove inizialmente sono introdotti i concetti e le tecnologie utilizzate per poi approfondire in che modo possono essere sfruttate.

- **Monitoraggio e Kubernetes**: Nel primo capitolo viene introdotto il concetto di monitoraggio e il funzionamento di un cluster Kubernetes.
- **Prometheus**: In questo capitolo viene approfondito il framework di monitoraggio Prometheus e riprodotto un ambiente simile a quello utilizzato nella campagna di injection, in locale.
- **Analisi dataset**: Nell'ultimo capitolo viene approfondito il dataset ricavato con Prometheus durante la campagna di injection per poi effettuarci una serie di analisi, in modo da comprendere come si possa utilizzare il monitoraggio su un cluster Kubernetes per evitare disservizi agli utenti.

Chapter 1

Monitoraggio e Kubernetes

1.1 Monitoring

Per comprendere al meglio il funzionamento di Prometheus, è necessario prima comprendere cosa sia il concetto di monitoraggio stesso e in che modo può essere utilizzato.

1.1.1 Cosa è?

Il monitoring è un processo che si basa principalmente **sull'osservazione costante dello stato di salute un sistema**, con l'obiettivo di individuare eventuali anomalie prima che queste possano causare disservizi agli utenti.

In particolare, nel modello DevOps, il monitoring funge da congiun-

zione tra le operazioni di sviluppo e quelle operazionali, perché la raccolta di dati in tempo reale sulle prestazioni e la salute del sistema permette di poter pianificare in maniera più accurata le successive iterazioni, favorendo un'evoluzione rapida e continua, permettendo di rispondere efficacemente alle nuove esigenze che nascono durante la vita di un software.

1.1.2 Logging vs Monitoring vs Alerting

Spesso logging e monitoring vengono considerati come attività distinte o addirittura in competizione, ma in realtà sono **complementari**. Mentre il logging si focalizza sulla **registrazione degli eventi** che si verificano all'interno del sistema, il monitoring si occupa di **monitorare lo stato attuale** del sistema. La combinazione di entrambe le tecniche permette di affrontare in maniera più efficiente il *troubleshooting*: i log forniscono dettagli sugli eventi anomali, mentre le metriche del monitoraggio mostrano lo stato del sistema nel momento in cui tali eventi si sono verificati. Questo approccio integrato consente di analizzare i problemi in modo più approfondito e accurato.

L'alerting, invece, è strettamente legato al monitoraggio e si riferisce all'insieme delle operazioni finalizzate a **notificare in modo tempestivo il verificarsi di una condizione critica**. Esempi includono l'eccessivo utilizzo delle risorse in un cluster, o un incremento anomalo nei tempi di risposta a richieste HTTP per un periodo prolungato.

Sebbene concettualmente semplice, l'alerting è di fondamentale importanza poiché permette di evitare il degrado del sistema e di avviare prontamente operazioni di troubleshooting. Inoltre, l'alerting comporta una serie di **considerazioni tecniche**, come il filtraggio degli alert, la scelta della loro severità, la selezione del canale di comunicazione più adatto e altri fattori che incidono direttamente sull'efficacia delle operazioni di monitoraggio.

1.2 Kubernetes

La campagna di fault injection nasce proprio con l'idea di studiare in maniera sistematica gli errori che possono scatenarsi in un cluster **Kubernetes** [6], quindi per comprendere cosa viene effettivamente monitorato in un cluster e in che modo avviene tale processo con Prometheus, anche in questo caso è necessario prima comprendere come funziona Kubernetes stesso.

1.2.1 Cosa è?

Kubernetes è un **orchestratore di container**, ovvero un framework che si occupa sia di automatizzare tutte le operazioni per la gestione di quest'ultimi sia di garantire determinate specifiche (che possono essere di diverse tipologie, ad esempio sotto forma di risorse da utilizzare, di scalabilità e così via...) occupandosi quindi di gestire l'intero ciclo di

vita di un container.

Inizialmente sviluppato da Google, poi donato alla Cloud Native Computing Foundation [12] rendendolo così un software open source, ad oggi è diventato un de facto standard per questa tipologia di strumenti.

Ma prima di approfondirlo è doveroso introdurre i container.

1.2.2 Container

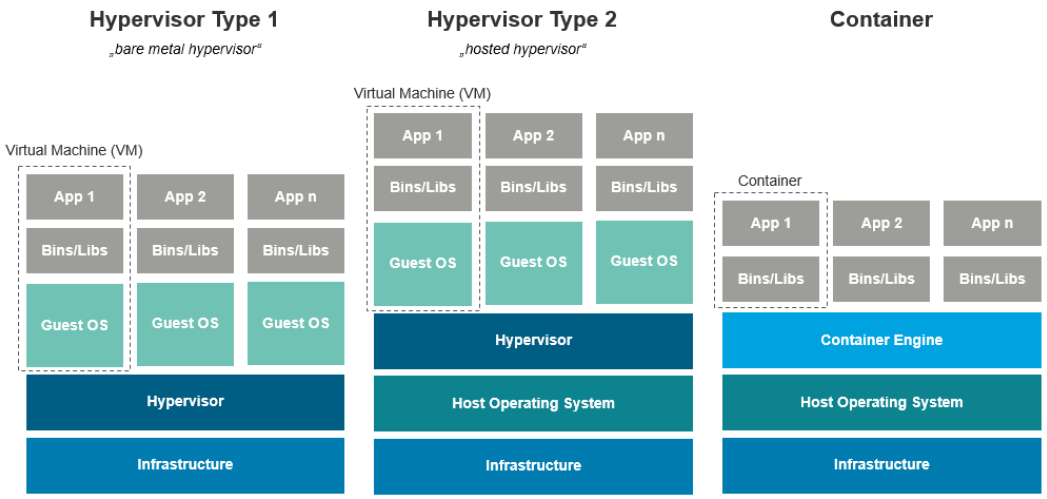

Figure 1.1: confronto virtualizzazione e containerizzazione [21]

Un container non è altro che un pacchetto che racchiude del codice software e tutti i componenti necessari per eseguirlo correttamente, come librerie e altre dipendenze, tutte incorporate in una singola struttura. La containerizzazione di un applicazione porta numerosi vantaggi, innanzitutto grazie all'isolamento dell'applicazione quest'ultima potrà essere eseguita in maniera **indipendente dall'ambiente** in cui si trova, ad esempio a prescindere dal sistema operativo dell'host, ma il

principale vantaggio è dato dalla **portabilità** e **leggerezza** di questi ultimi, infatti a differenza di una macchina virtuale che può essere vista come un sistema a sé stante, un container invece può essere visto come un'estensione di un classico processo di un sistema operativo e **non necessita della virtualizzazione delle risorse**, ma lavora direttamente condividendo il Kernel dell'host [fig. 1.1].

Sfruttano tecnologie come i *namespace*, che permettono di isolare un insieme di risorse, e i *cgroups*, che permettono invece di definire e limitare l'utilizzo delle risorse da parte di un gruppo di processi ad esempio di uno stesso namespace. Questo permette di **ridurre drasticamente l'utilizzo delle risorse** come lo spazio sul disco e in memoria, i tempi di avvio, etc... anche in termini di ordini di grandezza, semplificando la scalabilità di un applicazione.

I container a loro volta non sono altro che delle **istanze** di quelle che vengono chiamate **immagini**, definibili come l'analogo delle classi nella programmazione a oggetti, ma queste vengono a loro volta create effettuando un processo di **build** tramite appositi file di configurazione ed inoltre sarà anche necessario rilasciare e scalare opportunamente tali container. Questo insieme di operazioni dovrebbe poi essere ripetuto per ogni singolo container (che possono facilmente arrivare ad esserne centinaia), ed è per questo scopo che sono nati strumenti come Kubernetes che si occupano di **automatizzare** tutte queste operazioni.

1.2.3 Architettura

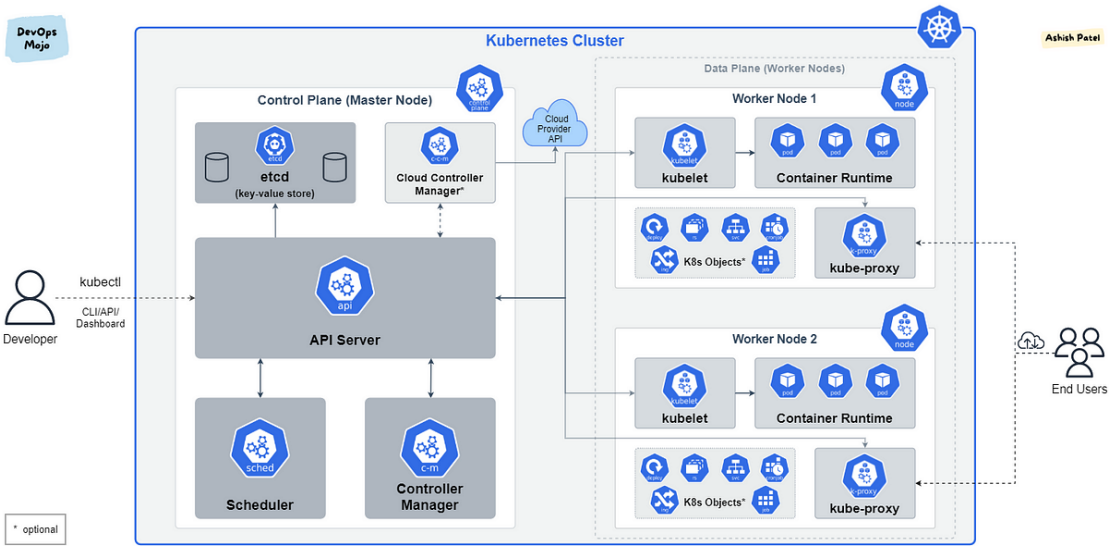

Figure 1.2: architettura di un cluster Kubernetes [22]

Come si può intuire Kubernetes è uno strumento complesso, ed è per questo che per semplificarne lo sviluppo e la comprensibilità è stato suddiviso in diverse componenti coese, raggruppate in quello che viene definito **cluster** [fig. 1.2]: un insieme di macchine comunicanti tra di loro sulla quale vengono eseguite sia tali componenti, sia i container della nostra applicazione.

Un cluster generalmente utilizza uno schema master / worker, ovvero alcune macchine fungeranno da **Control Plane** (dette **nodi master**) che si occuperanno di gestire le restanti macchine (dette **nodi** / **nodi worker**) sulla quale verranno eseguiti i **pod** (gruppi di container) ed altre tipologie di **oggetti** di Kubernetes.

Sia i nodi master che i worker, oltre ad eseguire gli oggetti specifici del sistema che si vuole creare, devono necessariamente eseguire al-

cune componenti che garantiscono il corretto funzionamento del cluster, ovvero:

- **Control Plane**
  - **kube-apiserver**

    espone le **API** di Kubernetes e processa le richieste **REST** o del tool di command line **kubectl**.
  - **etcd**

    database persistente, stateful e key-value, contenente tutte le informazioni riguardanti lo stato del cluster.
  - **kube-scheduler**

    assegna i nuovi pod ai nodi worker in base a diversi criteri.
  - **kube-controller-manager**

    controlla lo stato desiderato degli oggetti nel cluster e lo confronta con lo stato attuale tramite l'Apiserver, eventualmente applica degli step correttivi per riconciliarli.
- **Nodi**
  - **kubelet**

    intermediario tra l'Apiserver e il nodo, seguendo le istruzioni del nodo master avvia e controlla i pod nei nodi worker.
  - **container runtime**

    motore per scaricare, eseguire e gestire i container nei pod.

- **kube-proxy**

  network proxy che si occupa della gestione delle comunicazioni con i pod internamente ed esternamente.

Tutte queste componenti sono **stateless** e **comunicano indirettamente** tra di loro utilizzando l'Etcd, ma solo l'Apiserver può accederci e quindi la comunicazione avviene unicamente tramite l'invocazione delle apposite API.
Questo pattern rende l'Etcd un **componente critico**, ed è proprio su tale criticità che è stata basata la campagna di fault/error injection, dove le iniezioni sono state effettuate proprio nella comunicazione tra Apiserver e Etcd.

1.3 Perché Prometheus?

Prometheus [7] è un framework di **monitoraggio** e **alerting** ormai maturo, che mette a disposizione numerose funzionalità extra come l' immagazzinamento dei dati in maniera persistente, un apposito linguaggio di interrogazione sui dati, etc... che verranno approfondite nel prossimo capitolo.
Ma la vera differenza con i suoi competitor è del far parte della CNCF [12], perché essendo un progetto open source la sua grande diffusione permette di creare una vasta **community** attiva, di fondamentale importanza dato che chiunque può realizzare una propria fork del pro-

getto, chiunque può creare dei nuovi moduli per monitorare sistemi e strumenti non ufficialmente supportati (basta dare una rapida occhiata alle loro liste [24] per vedere come la maggioranza siano sviluppate dalla community), gli aggiornamenti sono continui, le vulnerabilità vengono rilevate e corrette rapidamente e soprattutto la **documentazione** che si può trovare online è immensamente maggiore rispetto a molte sue alternative.

In realtà infatti potremmo dire che Prometheus non abbia dei veri e propri competitor considerata la sua community, ma bensì che esistano delle soluzioni alternative che si focalizzano su specifici obbiettivi, ad esempio per citarne alcune:

- **InfluxDB** [15]: particolarmente efficiente in termini di risorse, ha un linguaggio di interrogazione più semplice da utilizzare e più rapido, ma ha un ecosistema decisamente più acerbo e risulta essere più complesso da configurare.

- **Graphite** [14]: punta tutto sulla semplicità di configurazione e d'uso, ma allo stesso tempo ha meno funzionalità di interrogazione e non è molto adatto per ambienti di monitoraggio complessi.

- **VictoriaMetrics** [30]: probabilmente l'alternativa che più si avvicina a Prometheus, infatti può essere vista come una sua variante ma con l'obiettivo di avere un footprint minore ed offre anche un linguaggio di interrogazione più avanzato, ma al costo

di avere meno funzionalità ed un ecosistema decisamente meno maturo, quindi con documentazione online e supporto della community inferiore.

Quindi anche se esistono numerose altre alternative, analogamente a quanto successo con Kubernetes, anche Prometheus è diventato un de facto standard nel suo ambiente.

Chapter 2

Prometheus

2.1 Cosa è?

Prometheus è un framework di **monitoraggio** e di **alerting** che può essere utilizzato sia per il monitoraggio dello stato di salute di un cluster sia per il monitoraggio di una specifica applicazione. Sulle **metriche** prodotte è anche possibile applicare delle **regole** di alerting per segnalare eventuali criticità su diversi canali di comunicazione come email, slack, etc...

Inizialmente sviluppato da SoundCloud [27], anche questo strumento è stato poi donato alla CNCF [12] e reso open source, diventando uno degli strumenti più diffusi nella sua categoria soprattutto grazie al suo **approccio modulare** che permette a chiunque lo desideri, di creare nuovi moduli per monitorare sistemi / strumenti / etc... non ufficialmente supportati.

2.2 Metriche

I dati utilizzati da Prometheus sono rappresentati sotto forma di metriche chiamate **time series**, composte da un **timestamp** relativo all'istante di collezionamento (nel tempo per una stessa metrica avremo più rilevamenti), e da coppie key-value opzionali dette **label** utilizzate per categorizzarle.

Queste time series possono essere facilmente interrogate e manipolate tramite l'apposito linguaggio di interrogazione detto **PromQL** [26], standardizzandone l'accesso e rendendo possibile interrogare tali metriche da diversi strumenti come la web UI di Prometheus, da richieste HTTP oppure tramite tool di terze parti come **Grafana** [13]. Grafana in particolare permette di creare **dashboard e grafi personalizzati** tramite un'apposita web UI, velocizzando la valutazione dello stato del sistema.

2.3 Architettura

L'approccio utilizzato da Prometheus per recuperare i dati è detto **scraping**, che sarà approfondito a breve, mentre i dati come già detto sono rappresentati sotto forma di metriche.

L'architettura si può suddividere nelle seguenti componenti, considerabili come moduli a se stanti [fig. 2.1]:

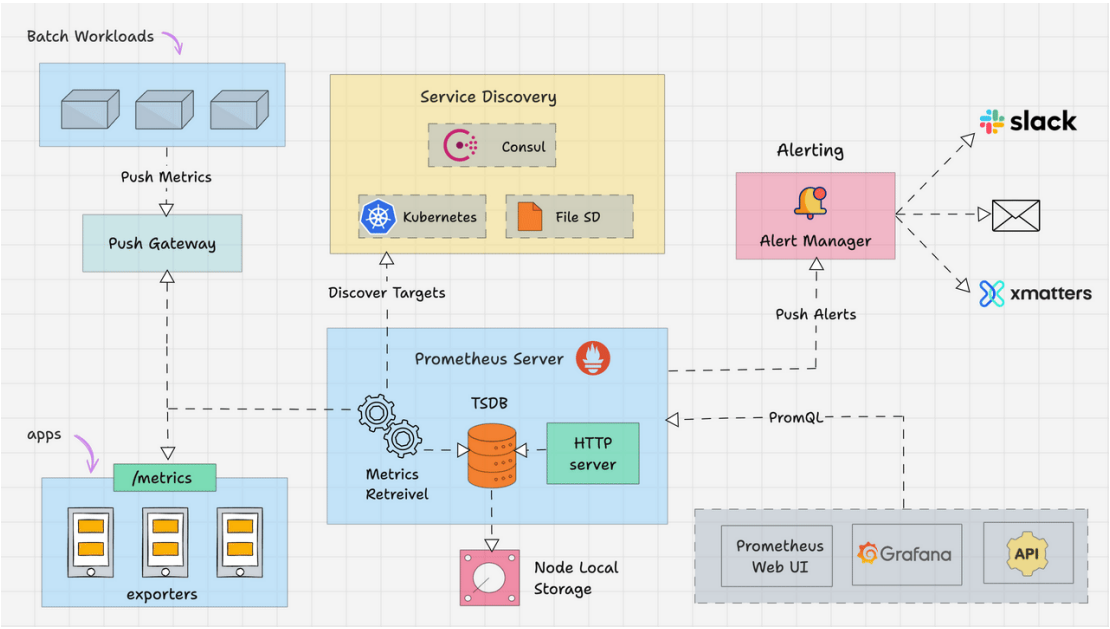

Figure 2.1: architettura framework Prometheus [8]

- **Server**

  cuore di questo strumento, colleziona le metriche dai vari target con un pattern pull, periodicamente tramite richieste HTTP (processo detto **scraping**), dove il periodo di tempo può essere personalizzato. E' composto a sua volta da:

  - **Time-Series Database (TSDB)**: database che immagazzina le time series delle metriche in maniera persistente ed efficiente, sfruttando apposite policy temporali per la conservazione; di default salva i dati in locale sul nodo in cui viene installato Prometheus, ma può essere configurato per il salvataggio in cloud.
  - **HTTP server**: abilita l'interrogazione al TSDB tramite

richieste HTTP ricevute dall'esterno contenenti le query in *PromQL*.

- **Metrics Retrieval**: colleziona le metriche dai **target** inviando richieste HTTP periodicamente.

- **Targets**

  sorgenti per lo scraping del server, in genere utilizzano degli **exporter** che convertono delle metriche da un formato non supportato (per esempio metriche di sistema) ad uno adatto a Prometheus, o delle **librerie cliente** che permettono di esporre delle metriche personalizzate da un applicazione scrivendo direttamente nel codice. Entrambe queste tipologie di moduli possono essere creati da zero da qualsiasi utente, per questo le loro liste vengono suddivise in ufficiali e non [24].

  Negli scenari in cui non è possibile utilizzare il pattern pull come in task di breve durata, allora si può utilizzare il **PushGateway**, un componente a se stante che funge da buffer temporaneo, permettendo ai target di inviare direttamente le metriche (pattern **push**) a quest'ultimo, dalla quale saranno poi estratte come in un target classico.

- **Service Discovery**

  uno dei concetti principali di questo strumento, i target possono essere rilevati in 2 modi:

  - **configurazione statica**: i target sono hard coded nei file di configurazione di Prometheus o in appositi file YAML/Json, ma ciò è possibile solo con target che hanno endpoint statici, che in genere non è nella natura degli oggetti di Kubernetes.
  - **Sevice Discovery**: consente a Prometheus di rilevare e monitorare automaticamente i target senza richiedernee la configurazione manuale per ognuno di essi.

- **Alert Manager**

  per quanto riguarda l'alerting, Prometheus si occupa unicamente di **triggerare** gli alert definendo delle **treshold**, ma tali alert sono poi gestiti dall'**Alert Manager**, il quale dopo averci effettuato diverse manipolazioni come la decuplicazione, raggruppamento, etc... notificherà gli appositi destinatari in base ai diversi criteri di **routing** con il quale è stato configurato.

2.4 Prometheus Operator

Per installare Prometheus si possono usare 2 approcci principalmente, uno **classico** installando il framework come una normale applicazione usando dei file binari pre-compilati, delle immagini Docker o Helm, dove però è richiesto un processo di configurazione lento e tedioso consistente nella creazione e modifica di diversi file di configurazione in maniera statica, aumentando di molto i tempi di setup ed eventual-

mente di troubleshooting.

Invece, un altro approccio che si può utilizzare è quello di sfruttare gli **operatori** di Kubernetes, che potremo dire essere la nuova frontiera per la distribuzione di applicazioni in un cluster, infatti gli operatori non sono altro che delle **estensioni** che permettono di integrare un'applicazione in maniera più stretta con Kubernetes, dando così la possibilità di creare degli **oggetti personalizzati** utilizzando delle **Custom Resource Definitions** (CRDs).

Prometheus mette a disposizione il **Prometheus Operator** [25], che consente di installare in modo semplice l'intero stack di monitoraggio (Prometheus, Alert Manager, Grafana, etc.), semplificando significativamente la configurazione del framework. Grazie a questo strumento, non è più necessario modificare manualmente i file di configurazione, ma si possono invece creare oggetti personalizzati, come il Deployment di Prometheus o dell'Alert Manager. La configurazione può essere suddivisa in più oggetti personalizzati distinti, evitando l'uso di un unico file. Ad esempio, il ServiceMonitor permette di dichiarare quali servizi di Kubernetes monitorare, migliorando la modularità e la gestione della configurazione.

Tutto ciò avviene semplicemente installando il Prometheus Operator e creando nuovi oggetti personalizzati, che verranno automaticamente rilevati tramite l'uso delle label, aggiornando automaticamente tutte le configurazioni che in passato avrebbero richiesto interventi manuali.

2.5 Sistema locale

Come già anticipato per comprendere al meglio il funzionamento di tali strumenti, è stato ricreato un ambiente simile a quello utilizzato nella campagna di fault/error injection con un'applicazione campione [fig. 2.2], con ovvie limitazioni a causa della disponibilità di risorse computazionali.

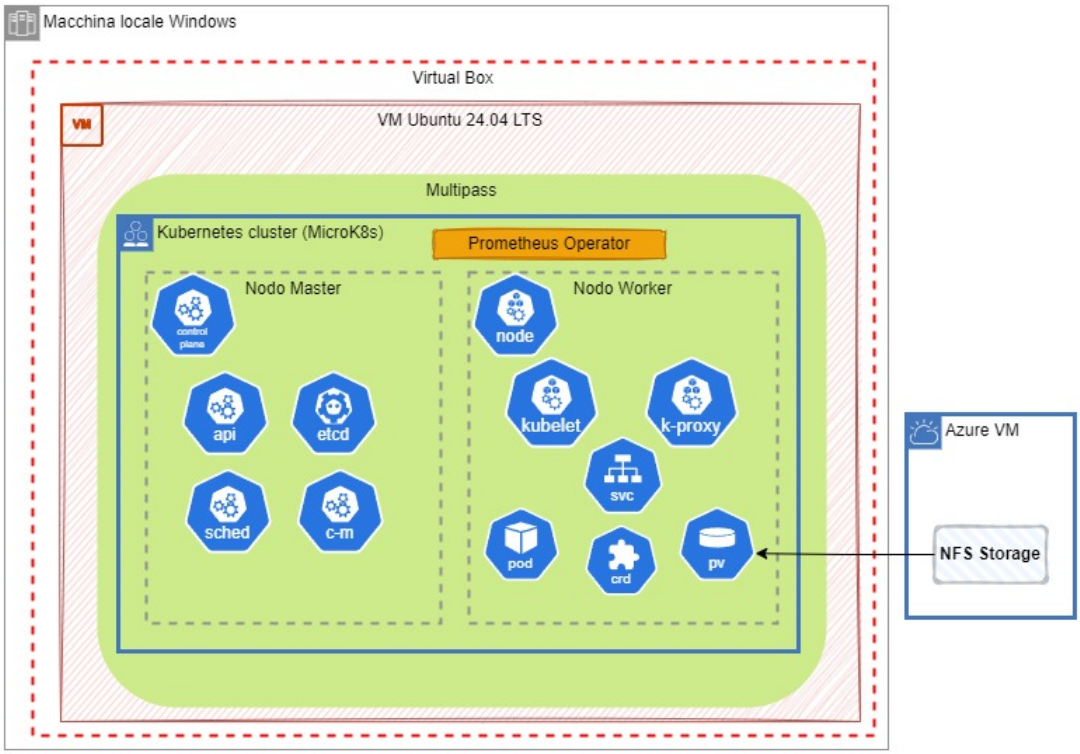

Figure 2.2: infrastruttura cluster locale

2.5.1 Ambiente

Partendo dall'alto verso il basso il sistema gira su una macchina virtuale Ubuntu 24.04 LTS [5] eseguita su VirtualBox 7.0.20 [31], con 4

core e circa 10 GB di memoria.

Su tale VM gira l'intero cluster, composto da 2 nodi (1 master e 1 worker), dove ogni nodo corrisponde a sua volta ad un altra macchina virtuale Ubuntu 24.04 LTS da 1 core e 4 GB di memoria, gestite con Multipass 1.14.0 [3].

2.5.2 MicroK8s

Per creare un cluster Kubernetes è stato utilizzato **MicroK8s** 1.30.0 [2], framework open source che permette di creare un **cluster lite** adatto a sistemi con poche risorse.

Il vero vantaggio di tale framework però è nella **semplicità** della configurazione del cluster, il punto debole di Kubernetes, infatti MicroK8s permette di creare un cluster semplicemente installando il framework su ogni nodo (ad esempio con **Snap** [4]), utilizzare un apposito comando dal nodo master e copiare ed incollare le istruzioni a schermo sui nodi worker, ed il cluster è già configurato.

Un altre grande vantaggio di MicroK8s sono gli **addon**, ovvero delle estensioni attivabili direttamente tramite MicroK8s già preconfigurate e pronte all'uso, in particolare è stato abilitato l'addon **observability** che permette di installare lo stack **Kube-prometheus** [17], che comprende il Prometheus Operator, Grafana, l'Alert Manager e un insieme di exporter per monitorare lo stato del cluster come Kube-state-metrics [28] e node-exporter [11]. Anche in questo caso con un

semplice comando da terminale *microk8s enable observability*, l'intero ecosistema di Prometheus è già in funzione per monitorare lo stato del cluster con anche delle regole di alerting predefinite.

2.5.3 Applicazione campione

Come applicazione da monitorare è stata scelta un'applicazione per la gestione di sensori di temperatura e pressione, costituita da un BE in Flask e da degli script per simulare le richieste dei client [1]. Oltre a monitorare lo stato del cluster, è stata modificata l'applicazione utilizzando una delle client library di Prometheus permettendo di aggiungere l'endpoint per lo scraping e realizzare delle metriche personalizzate [fig. 2.3] per monitorare le richieste ricevute, il tempo di elaborazione e le relative risposte.

2.5.4 Configurazione applicazione campione

Per poter monitorare anche l'applicazione d'esempio è necessario creare degli appositi oggetti di Kubernetes [2] , sia per quanto riguarda la gestione dell'applicazione che per configurare Prometheus ad effettuare lo scraping delle nuove metriche.

[1] Il codice è disponibile su GitHub [10], mentre le immagini dei container su DockerHub [9]

[2] Tutti i file per creare questi oggetti sono disponibili sul repository GitHub [10] in formato .yaml.

```python
from prometheus_client import Counter, Gauge, Histogram, generate_latest, CONTENT_TYPE_LATEST
import os


app = Flask("__name__")

http_requests_count = Counter('sensors_http_requests_count', 'Total HTTP Requests', ['method', 'endpoint'])
request_duration_seconds = Histogram('sensors_request_duration_seconds', 'Request duration in seconds', ['method', 'endpoint'])
http_error_count = Counter('sensors_http_error_count', 'Total HTTP errors', ['method', 'endpoint', 'status_code'])
http_error_arguments_count = Counter('sensors_http_error_arguments_count', 'Total HTTP errors due to bad arguments passed', ['method',
active_requests = Gauge('sensors_active_requests', 'Active HTTP requests', ['method', 'endpoint'])


def get_collection(name: str) -> collection.Collection:
    return MongoClient(os.environ['MONGO_HOST_NAME'], int(os.environ['MONGO_HOST_PORT'])).get_database("sensors").get_collection(name)


@app.post("/sensor")
def create_sensor():
    http_requests_count.labels(method=request.method, endpoint=request.path).inc()
    active_requests.labels(method=request.method, endpoint=request.path).inc()

    with request_duration_seconds.labels(method=request.method, endpoint=request.path).time():
        sensor = request.get_json()

        coll = get_collection("sensor")
        active_requests.labels(method=request.method, endpoint=request.path).dec()
```

Figure 2.3: esempio BE Flask con metriche da client library

Deployment applicazione

Il deploy dell'applicazione è costituito dai seguenti oggetti Kubernetes:

- **Deployment** del BE in Flask.
- **Service** [3] per esporre il BE all'esterno.
- **StatefulSet** per l'istanza di MongoDB.
- **Service** per esporre MongoDB al BE.
- **PersistentVolume** per rendere persistenti i dati di MongoDB, utilizzando una macchina virtuale su Azure come NFS storage.
- **Deployment** per gli script dei client.

[3] I Service sono degli oggetti di Kubernetes che si occupano di rendere disponibile in rete determinati pod, selezionati ad esempio tramite label, e fare loro da load balancer.

Configurazione Prometheus

La configurazione di Prometheus è avvenuta in maniera analoga a quanto fatto precedentemente grazie al Prometheus Operator, con il quale è stato sufficiente creare degli **oggetti personalizzati** per tale operatore senza dover toccare i file di configurazione di Prometheus stesso. Per fare ciò è stato sufficiente creare un numero ridotto di oggetti sfruttando gli oggetti già creati dall'addon observability, impostando le **label** dei nuovi oggetti in maniera opportuna per far funzionare correttamente il **service discovery** di Prometheus. In particolare:

- **Service** per esporre Prometheus, Grafana e l'Alert Manager all'esterno.
- **ServiceMonitor** per abilitare il monitoraggio (scraping) al Service del BE.
- **PrometheusRule** per creare un'alert rule personalizzata campione.

Con questa configurazione possiamo sia monitorare lo stato del cluster sfruttando gli exporter e le dashboard preinstallate [fig. 2.4] o sfruttando metriche e dashboard personalizzate [fig. 2.5], sia applicare delle regole di alerting in pochi semplici passaggi.

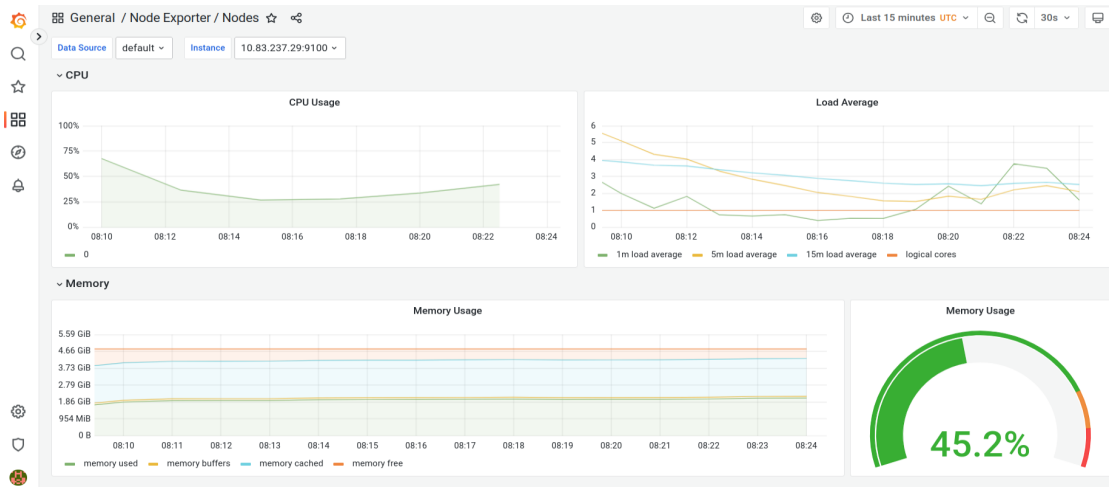

Figure 2.4: dashboard Grafana con metriche da Node Exporter

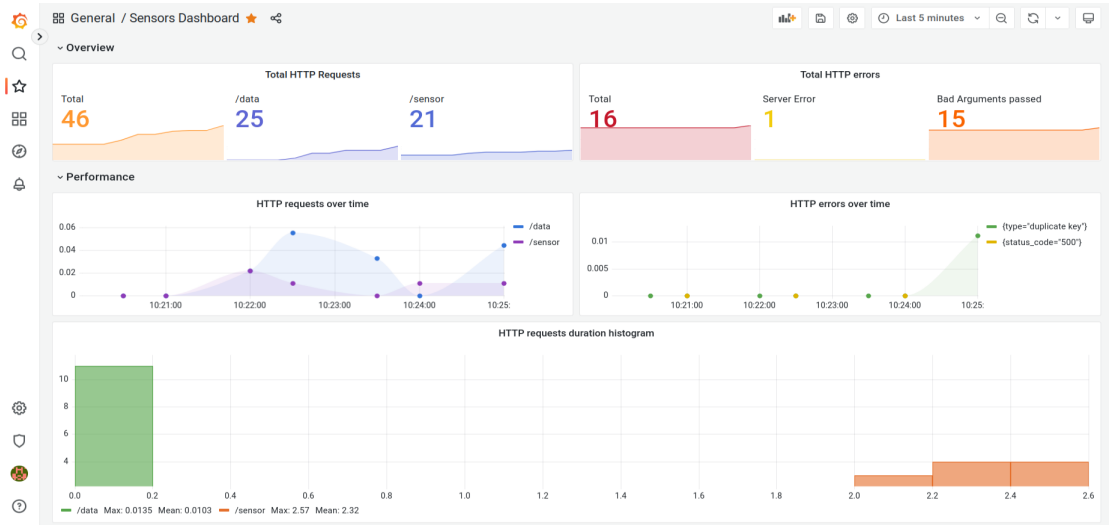

Figure 2.5: dashboard Grafana personalizzata del BE

Chapter 3

Analisi dataset

In questa seconda parte l'obiettivo è quello di ricercare una correlazione tra gli errori scatenati durante la campagna di fault/error injection e le metriche estratte da Prometheus, in maniera tale da mostrare come sia possibile utilizzare tali metriche per creare apposite regole di alerting e far attivare tempestivamente eventuali operazioni di troubleshooting.

3.1 Dataset

3.1.1 Campagna injection

La campagna di fault/error injection è stata condotta utilizzando **Mutiny**, un framework appositamente sviluppato per Kubernetes che consente di manipolare i messaggi scambiati tra le varie componenti del sistema. L'applicazione monitorata è un web server Flask, il quale risponde alle richieste dei client eseguendo elaborazioni casuali. Le manipo-

lazioni dei messaggi possono avvenire durante la comunicazione **tra l'Apiserver e l'Etcd**, con la possibilità di influire direttamente sullo stato attuale o desiderato del cluster, oppure **tra le componenti e l'Apiserver**, sebbene in quest'ultimo caso sia più probabile che i messaggi vengano identificati come corrotti e rigettati dall'Apiserver stesso.

Durante la campagna sono state eseguite tre principali tipologie di iniezione:

- **Bit flip**: viene invertito un singolo bit all'interno del messaggio.
- **Data-type set**: viene impostato un valore boundary o errato, a seconda del tipo di campo interessato.
- **Message drop**: simula la mancata ricezione di un aggiornamento di stato del cluster, dovuta a vari motivi come richieste fallite, bug, etc...

Per garantire un dataset completo, le metriche di Prometheus sono state raccolte utilizzando node exporter [11], kube-state-metrics [28] e metriche personalizzate dell'applicazione. Queste metriche sono state ricavate per ogni singola esecuzione della campagna. In particolare, sono state effettuate 300 run di controllo (*gold runs*), dai cui risultati sono stati estratti i valori di riferimento per confrontare le time series generate durante le 8.782 run di iniezione.

3.1.2 Struttura dataset

La combinazione di metriche ottenute durante la campagna permette di estrarre informazioni riguardanti lo stato dei nodi, degli oggetti Kubernetes e dell'applicazione, offrendo una vista completa sullo stato di salute del sistema.
I dati sui cui sono state effettuate le analisi però sono solo un subset dell'intero dataset della campagna, con un numero limitato di tipologie di metriche e di classi di dati.

3.1.3 Classificazione

Per poter effettuare un'analisi in maniera più sistematica, i risultati delle run durante la campagna sono stati classificati a seconda del tipo di errore **visto dall'utente**:

- Nessun Impatto Significante (**NSI**).
- Higher Response Time (**HRT**), solo ritardi temporali.
- Intermittent Availability (**IA**), risposte di errore a intermittenza.
- Service Unreachable (**SU**), servizio irraggiungibile.

a seconda del tipo di errore al **livello di orchestrazione**, di cui nel subset sono presenti:

- **LeR**: il numero di repliche pronte, Pod creati o endpoint è stabile e inferiore alla baseline.

- **MoR**: il numero di repliche pronte, Pod creati o endpoint è superiore alla baseline.
- **Net**: il numero di repliche pronte e Pod è corretto, ma alcuni non sono raggiungibili o utilizzati nel bilanciamento del carico.

ed a seconda di sotto quale tipologia di **workload** di orchestrazione si è verificato l'errore:

- **Deploy**: crea nuovi Deployment e Pod correlati.
- **Scale**: aumenta il numero di repliche dei Deployment esistenti.
- **Failover** / **Availab**: forza i pod ad essere rigenerati tramite l'iniezione di un errore.

	Deploy				**Scale**				**Failover**			
	NSI	HRT	IA	SU	NSI	HRT	IA	SU	NSI	HRT	IA	SU
No	**1617 (62.2%)**	**84 (3.2%)**	0	0	**1382 (54.5%)**	**77 (3.0%)**	0	0	**2652 (72.7%)**	**137 (3.8%)**	11 (0.3%)	0
Tim	28 (1.1%)	1	0	0	40 (1.6%)	8	1	0	18 (0.5%)	11 (0.3%)	2	0
LeR	109 (4.2%)	138 (5.3%)	4	5	432 (17.0%)	63 (2.5%)	0	0	59 (1.6%)	10 (0.3%)	1	0
MoR	368 (14.2%)	**12 (0.5%)**	2	0	303 (12.0%)	**41 (1.6%)**	7	0	531 (14.6%)	**31 (0.8%)**	0	0
Net	14 (0.5%)	7	6	**107 (4.1%)**	28 (1.1%)	46 (1.8%)	10 (0.4%)	0	8	48 (1.3%)	**40 (1.1%)**	1
Sta	**81 (3.1%)**	**4**	0	0	**81 (3.2%)**	**5**	0	0	**66 (1.8%)**	**8**	0	0
Out	**10 (0.4%)**	**1**	0	**1**	**8**	**1**	0	**1**	**7**	**2**	2	**4**

Figure 3.1: risultati campagna di injection mappati errori - workload

3.2 Analisi

L'approccio scelto per analizzare le metriche di Prometheus a disposizione è quello di realizzare dei grafi che permettano di visualizzare l'andamento delle metriche nei diversi scenari di errore a livello orchestrazione e per tipologia di workload.

Per fare ciò è stato realizzato un **notebook Jupyter** [23], che permette di caricare il dataset in memoria per poi scegliere singolarmente quali grafi realizzare in un secondo momento. Questo è possibile in quanto il flusso per generare un grafo è composto da 3 step separati: il caricamento del dataset in memoria, l'elaborazione dei dati per semplificarne l'utilizzo ed infine la generazione del grafo. [1]

3.2.1 Codice

Caricamento dati

Nel dataset le metriche seguono circa questa struttura:
/tipo errore orchestrazione/[num. run]_workload/file metrica
quindi per caricare i dati viene esplorato l'intero albero di directory, estrapolando i valori dai file delle metriche, file per file, per ottenere un array JSON [fig. 3.2], dove una possibile riga è:

```
{
    'metric':
        {
            '__name__': 'node_memory_MemFree_bytes',
            'instance': '192.168.100.20:9100'
        },
    'value': [1695843909.839, '1976422400'],
    'condition': 'Baselines', 'workload': 'deploy'
}
```

in cui possiamo osservare il tipo di metrica, la time series (timestamp + valore), se la time series fa parte di una gold run o di una delle run di iniezione e sotto quale workload è stata rilevata.

[1] il notebook intero è reperibile sul repository GitHub [10]

```python
# Function to load data from a file
def load_metrics_data(file_path):
    data = []
    metric_name_base = os.path.basename(file_path).split('{')[0].replace("(", " ")

    with open(file_path, 'r', encoding="utf-8") as file:
        for line in file:
            if 'OFFSET' in line:
                continue

            try:
                # Replace single quotes with double quotes only outside the JSON structure
                safe_line = re.sub(r"(?<!\\)'", '"', line)

                # Convert the JSON-like string to a Python dictionary
                metric_data = json.loads(safe_line)
                metric = metric_data['metric']

                if '__name__' not in metric:
                    metric['__name__'] = metric_name_base

                    if metric_name_base in ["sum rate apiserver_request_total", "sum rate rest_client_requests_total"]:
                        metric['status_code'] = "200" if "2" in os.path.basename(file_path) else "400/500"

                        # Extract job or kubernetes_pod_name based on metric name
                        if metric_name_base == "sum rate apiserver_request_total":
                            job_match = re.search(r"job=_([^_]+)_", os.path.basename(file_path))
                            if job_match:
                                metric['job'] = job_match.group(1)
                        else:
                            pod_match = re.search(r"kubernetes_pod_name=_([^_]+)_", os.path.basename(file_path))
                            if pod_match:
                                metric['kubernetes_pod_name'] = pod_match.group(1)

                    data.append(metric_data)
                except json.JSONDecodeError:
                    continue

    return data
```

Figure 3.2: estrazione metrica in JSON

Elaborazione dati

Per poter lavorare più facilmente con i dati, l'array JSON viene convertito in un Dataframe della libreria Pandas [19] [fig. 3.3], libreria che offre strutture dati e metodi per manipolare i dati.

Generazione grafi

La generazione dei grafi avviene utilizzando la libreria **Matplotlib** [18] e il suo toolkit **Seaborn** [20], che semplifica la creazione dei grafi più comuni ed è compatibile con le funzionalità della libreria Pandas, grazie alle quali è stato possibile realizzare boxplot e heatmap [fig. 3.4].

```python
# Convert to DataFrame for easier analysis
df = pd.DataFrame(data)

# Clean the data frame

# Convert 'value' from string to float and extract timestap
df['timestamp'] = df['value'].apply(lambda x: x[0])
df['value'] = df['value'].apply(lambda x: float(x[1]))

# Extract the metric's fields
df['name'] = df['metric'].apply(lambda x: x.get('__name__'))
df['endpoint'] = df['metric'].apply(lambda x: x.get('endpoint'))
df['replicaset'] = df['metric'].apply(lambda x: x.get('replicaset').split('-')[0] if x and 'replicaset' in x else None)
df['device'] = df['metric'].apply(lambda x: x.get('device'))
df['instance'] = df['metric'].apply(lambda x: x.get('instance'))
df['resource'] = df['metric'].apply(lambda x: x.get('resource'))
df['job'] = df['metric'].apply(lambda x: x.get('job'))
df['kubernetes_pod_name'] = df['metric'].apply(lambda x: x.get('kubernetes_pod_name'))

df['container'] = df['metric'].apply(lambda x: x.get('container'))
df['phase'] = df['metric'].apply(lambda x: x.get('phase'))

# reasons metrics
df['reason'] = df['metric'].apply(lambda x: x.get('reason'))

df['uid'] = df['metric'].apply(lambda x: x.get('uid'))

# histogram metrics
df['le'] = df['metric'].apply(lambda x: float(x.get('le')) if x and 'le' in x else None)

# status metrics
df['status_code'] = df['metric'].apply(lambda x: x.get('status_code'))
```

Figure 3.3: conversione array JSON in Dataframe

```python
# Boxplot
def plot_metric_boxplot(df, metric_name, title="Missing title", y_label=None, value_to_show="value"):

    fig, axes = plt.subplots(1, 3, figsize=(18, 6), sharey=True)

    for i, workload in enumerate(["availab", "deploy", "scale"]):

        # Filter the DataFrame for the specific metric
        filtered_df = df[df['name'] == metric_name]
        filtered_df = filtered_df[filtered_df['workload'] == workload]

        # Create a boxplot grouped by condition
        sns.boxplot(x='condition', y=value_to_show, data=filtered_df, hue='condition', showmeans=True, ax=axes[i])

        axes[i].set_title(f'{workload}')
        axes[i].set_xlabel('')
        axes[i].set_ylabel(' ')

    # Set shared labels
    fig.supylabel(y_label if y_label else "Value")
    fig.supxlabel('Condition')
    fig.suptitle(title, fontsize=16)
    fig.set_label(" ")

    plt.tight_layout()
    plt.show()
```

Figure 3.4: codice boxplot

Boxplot e Heatmap

I box plot sono strumenti grafici utili per visualizzare la distribuzione di un insieme di dati. La scatola nel grafico rappresenta l'**intervallo interquartile** (**IQR**), con la linea centrale che indica la mediana,

mentre i *whisker* si estendono fino ai valori minimo e massimo, a meno che questi non siano considerati *outlier* (valori anomali) [fig. 3.5]. Questo tipo di grafico consente di identificare eventuali variazioni nell'andamento delle metriche in funzione della tipologia di errore e del carico di lavoro, permettendo un confronto chiaro delle diverse distribuzioni.

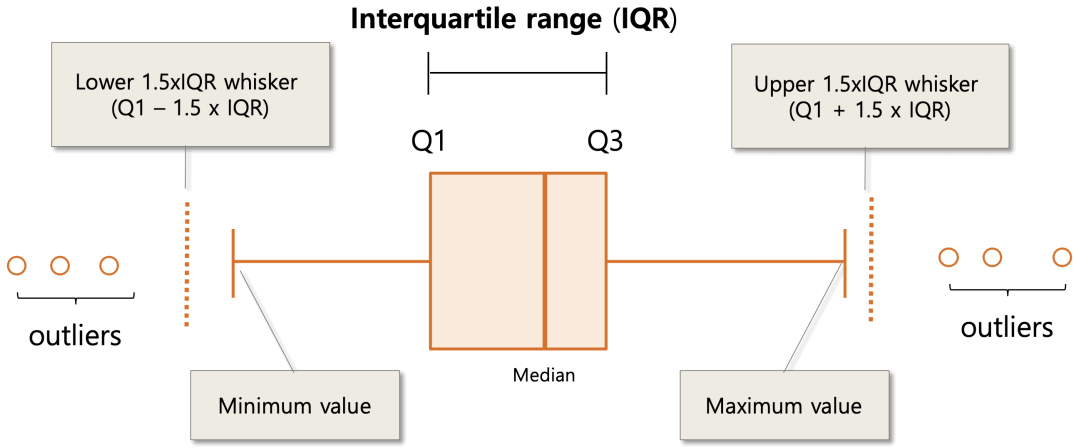

Figure 3.5: esempio di box plot [1]

Le heatmap, invece, sono rappresentazioni grafiche che utilizzano una **scala di colori** per visualizzare la densità dei dati all'interno di un'area specifica. Nel contesto di questo lavoro, le aree rappresentano combinazioni tra il tipo di errore a livello di orchestratore e la tipologia di workload. Le zone con colori più intensi (zone "calde") indicano una maggiore concentrazione di dati, facilitando così l'individuazione di pattern o anomalie nelle distribuzioni [fig. 3.6].

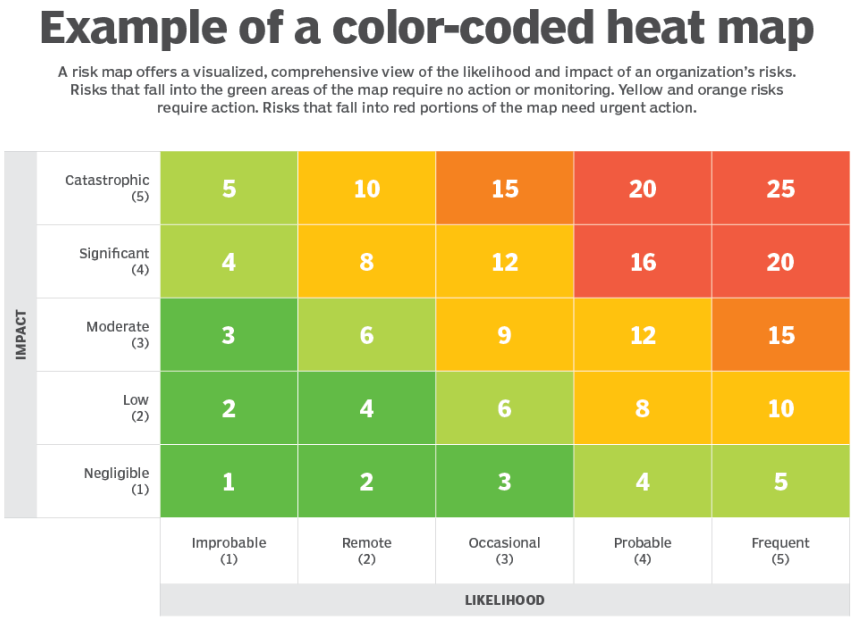

Figure 3.6: esempio di heatmap [29]

3.2.2 Risultati

Lavorando solo su un subset non è stato ovviamente possibile effettuare un'analisi completa, ma il processo utilizzato è facilmente estendibile a qualsiasi altra metrica. Le metriche a disposizione sono circa 50, di cui diverse sono risultate essere non indicatrici di un **comportamento anomalo** come nel caso del ratio dei context switch al secondo dei processi nel nodo master [fig. 3.7], dove non si osservano variazioni significative nell'intervallo interquartile (IQR), e le piccole variazioni presenti nei *whisker* potrebbero essere influenzate da fattori esterni alla campagna di injection.

Di seguito verranno approfondite le metriche principali che hanno rilevato un comportamento anomalo per i possibili errori di orchestrazione confrontandole con il comportamento medio, che quindi possano effet-

tivamente essere utilizzate per l'alerting o che quantomeno siano utili da monitorare.

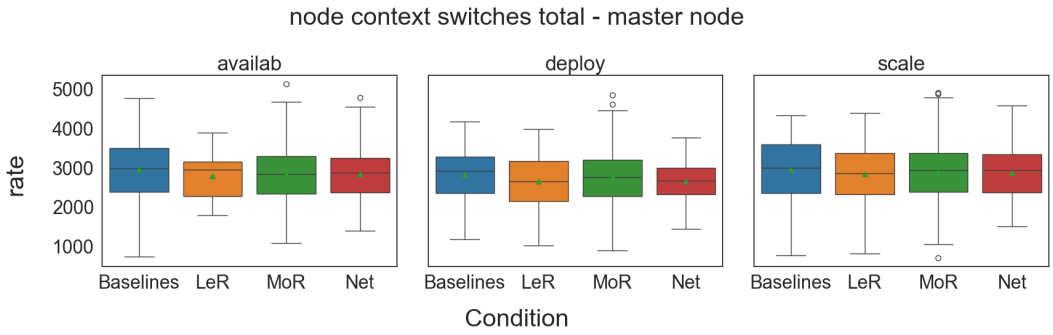

Figure 3.7: esempio di metrica non rilevante

Utilizzo risorse

Come si può osservare in tutte le condizioni di errore e sotto qualsiasi tipologia di workload, il nodo su cui è installato **il Control Plane utilizza decisamente più risorse** (circa 1 GB in più, sia sulla partizione del filesystem [fig. 3.8] sia sulla memoria [fig. 3.9], che su un nodo con 4 GB di ram è decisamente impattante), come si può evin-

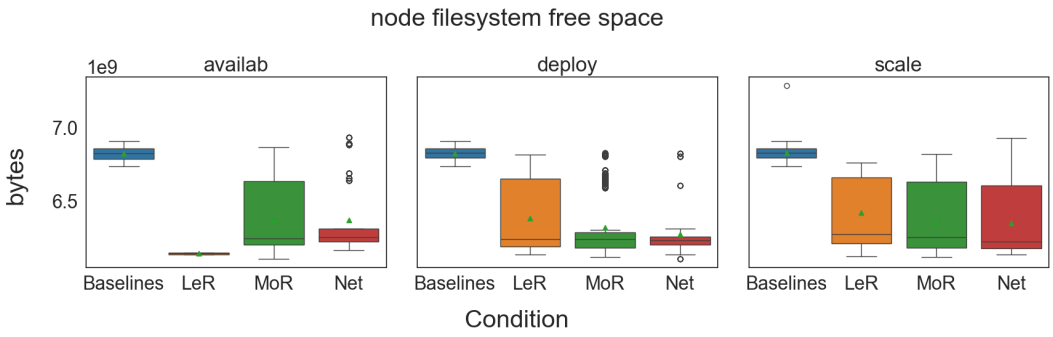

Figure 3.8: spazio libero del file system sul nodo in bytes (scala 10^9)

cere dal fatto che sotto tutti i workload, per entrambe le metriche, i whisker superiori degli errori non fanno nemmeno parte dell'IQR della baseline oltre ad avere una maggiore dispersione dei dati. Ad esempio il valore medio di memoria libero nel caso LeR sotto worload Availab è stato di circa 0.4 GB, mentre nel caso Baseline è di circa 1.7 GB. Questo probabilmente è il risultato di continue operazioni per cercare di portare lo stato del cluster allo stato desiderato senza mai riuscirci.

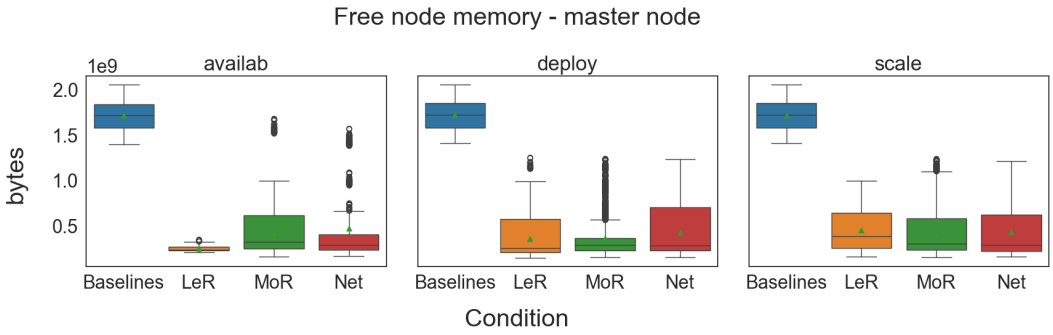

Figure 3.9: spazio libero in memoria sul nodo master in bytes (scala 10^9)

In particolare nel caso di errori Net, la memoria disponibile subisce delle forti variazioni perché si verifica una condizione detta **capacity offset**, in cui la capacità dichiarata di un nodo è diversa dalla capacità effettivamente utilizzabile, a causa delle comunicazioni fallite tra Kubernetes ed il resto del sistema [fig. 3.10]. Questa condizione non si verifica nei casi LeR e MoR perché viene proprio modificata la capacità delle risorse dei nodi dichiarata, anche se non corrisponde con l'effettiva capacità delle risorse fisiche del sistema.

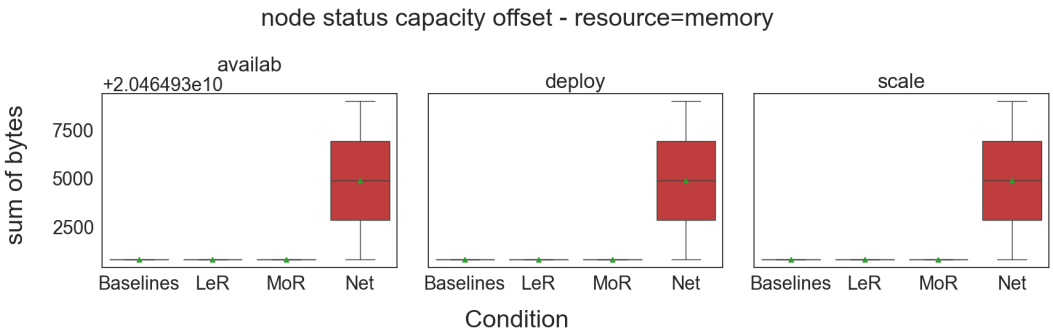

Figure 3.10: discrepanza memoria utilizzabile e memoria dichiarata

Anche il ratio di utilizzo della CPU da parte dei processi può essere un ottimo indicatore, infatti nella condizione MoR si verificano diversi **picchi**, dove in alcuni casi quasi viene raggiunto il 100% di utilizzo, indicati dai numerosi outliners, condizione che quindi si è verificata anche relativamente spesso e che può portare al crash dei pod e disservizi agli utenti [fig. 3.11].

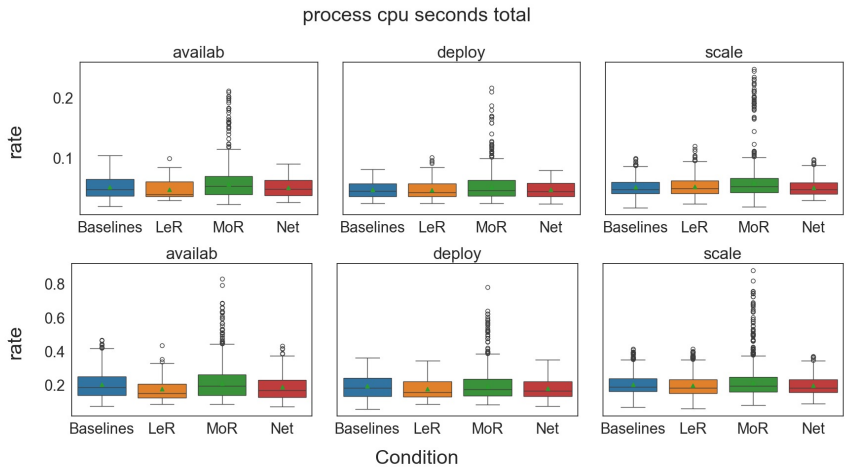

Figure 3.11: ratio utilizzo cpu

Network

Un'altra categoria di metriche da tenere sotto controllo riguarda le connessioni tra i nodi. Durante il workload di deploy, ovvero durante la creazione di nuovi pod o altri oggetti in Kubernetes, e in presenza di errori Net, sia il nodo master che i nodi worker tentano di stabilire un **numero elevato di nuove connessioni**. Questo comportamento si verifica perché molte delle connessioni falliscono, come evidenziato dall'ampiezza dell'intervallo interquartile (IQR) e dalla posizione elevata del whisker superiore. Tale fenomeno è visibile anche nella heatmap delle socket TCP in memoria, dove l'area deploy-Net è più scura rispetto alle altre. In aggiunta, si osserva un significativo **incremento del numero di pacchetti inviati**, a cui segue un **aumento dei pacchetti ritrasmessi** a causa degli errori di comunicazione, chiaramente rilevabile nella heatmap del ratio dei pacchetti ritrasmessi, dove tale area è marcatamente più scura [fig. 3.12].

Pod e container

Lo stato dei pod e dei container va monitorato in maniera tempestiva, dato che sono un ottimo riferimento per controllare la salute dei servizi offerti. In tutti i casi di errore si può osservare come i container dell'applicazione campione utilizzata nella campagna, **terminano numerose volte per errori** di diverso tipo, in particolar modo nel caso MoR, questo perché in generale tale problema risulta

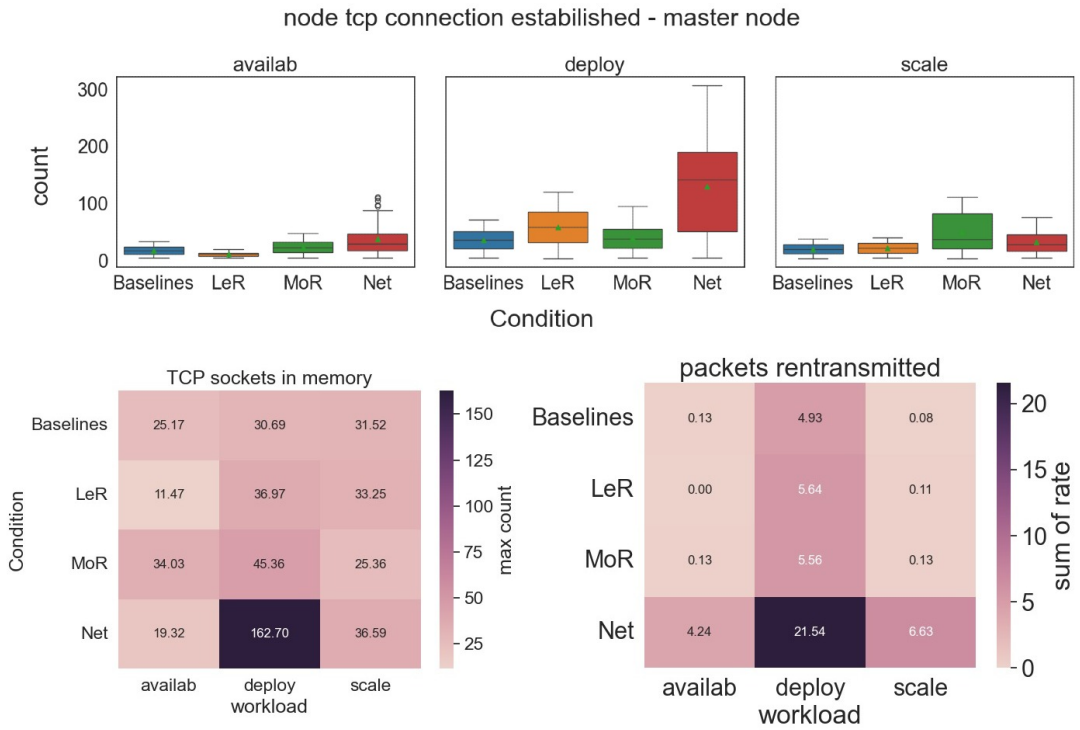

Figure 3.12: statistiche metriche di rete

essere più critico rispetto agli altri dato che vengono assegnate più risorse di quelle disponibili [fig. 3.13].

Un altro dettaglio da notare è come nel caso di LeR sotto workload Failover, **i pod cambiano stato molto meno spesso mentre i container terminati per errore sono decisamente inferiori**, questo proprio perché avendo meno risorse a disposizione vengono creati meno pod e container, quindi una volta che sono stati terminati in maniera forzata molti non vengono più riavviati.

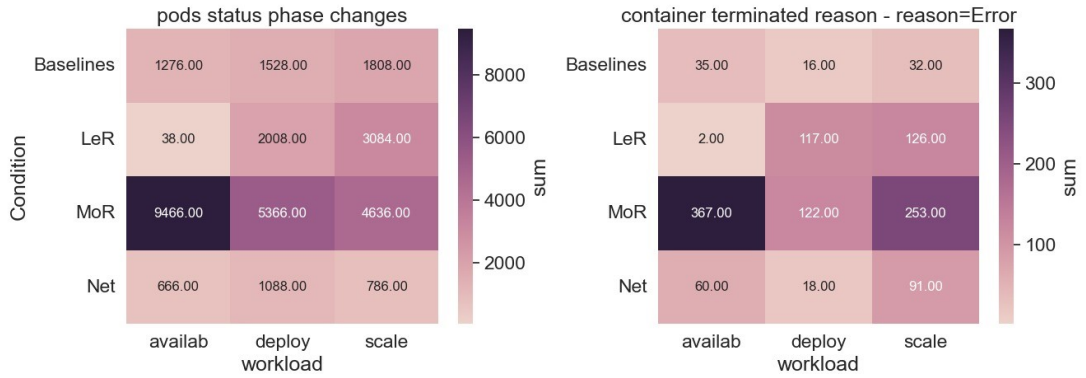

Figure 3.13: variazioni stato pod e container terminati per errore

Workqueue

Lo stato delle workqueue permette anch'esso di identificare un eventuale sovraccarico del sistema, come avviene nel caso MoR in cui nelle queue vengono aggiunti spesso **più task rispetto alla media** [fig. 3.14] credendo di avere maggiori risorse computazionali, situazione segnalata dai numerosi outliners, o addirittura nel caso di LeR sotto Failover risultano avere una **profondità quasi nulla** perché molti dei pod non vengono più riavviati risultando così in un carico molto più basso del normale [fig. 3.15].

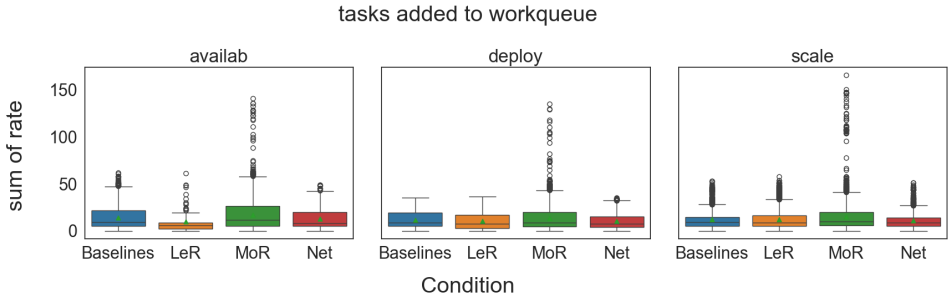

Figure 3.14: inserimento task in una workqueue

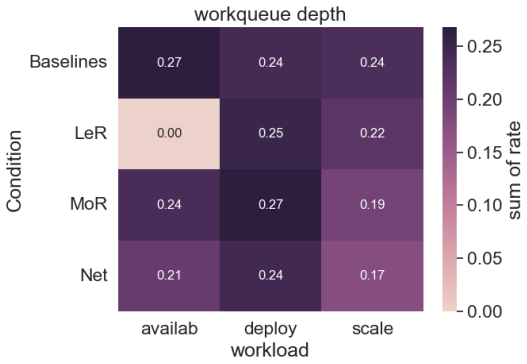

Figure 3.15: profondità queue (LeR-Failover quasi nullo)

Metriche personalizzate

Le metriche personalizzate create utilizzando le client library, permettono di monitorare in maniera più approfondita lo stato del servizio offerto, ad esempio con il server Flask utilizzato nella campagna possiamo controllare le **richieste HTTP ricevute** e relative **risposte** [fig. 3.16]. Controllando ad esempio se in un determinato periodo di tempo le risposte di errore sono superiori alla media dato che nel caso MoR risultano essere particolarmente frequenti, situazione denotabile dai numerosi outliners, a causa del fatto che il BE accetta più richieste di quante possa effettivamente soddisfarne.

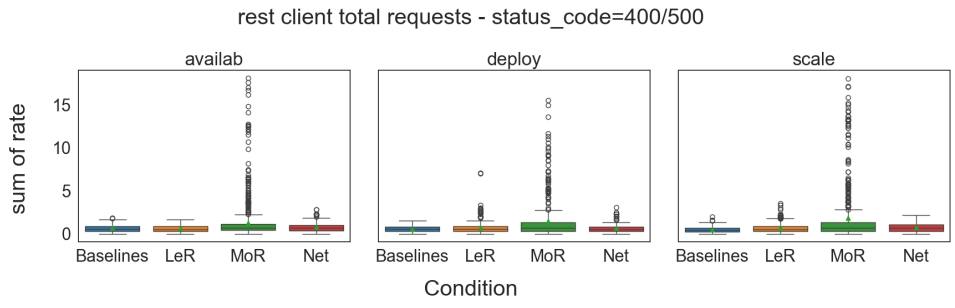

Figure 3.16: richieste HTTP con risposta 400/500

3.2.3 Utilizzo delle metriche

Per utilizzare efficacemente le metriche di Prometheus, è innanzitutto necessario ottenere un dataset iniziale che rappresenti lo stato di salute del cluster. Questo consente di stabilire un punto di riferimento per **identificare eventuali comportamenti anomali**, in maniera simile a quanto fatto precedentemente.

Tuttavia, per implementare delle regole di alerting basate su queste metriche, è essenziale definire soglie (**treshold**) di attivazione appropriate e considerare la possibilità di falsi positivi. Ad esempio, se consideriamo la metrica relativa allo spazio libero in memoria osservato nei nodi [fig. 3.9], si può notare che in condizioni normali il nodo master dispone mediamente di circa 1.7 GB di memoria libera, mentre in caso di errore questo valore scende significativamente. Pertanto, si potrebbe impostare una soglia del 15% inferiore alla media come limite di allerta, utlizzando una query PromQl come:

$$\boldsymbol{node\_memory\_MemFree\_bytes < THRESHOLD}$$

Con una soglia del 15%, i falsi positivi risultano essere inferiori all'1%, mentre il tasso di errori non rilevati è intorno al 2.7%. Sebbene questi valori siano accettabili, è importante considerare che le variazioni nei dati non sono sempre così evidenti. Ad esempio, nel monitoraggio delle richieste HTTP con status code 400 o 500 [fig. 3.16], tali eventi anomali possono essere isolati e limitati a brevi periodi di tempo. Per questo motivo, PromQL consente di **monitorare le metriche per**

un intervallo di tempo specifico, permettendo di verificare, ad esempio, se negli ultimi 3 minuti si è verificato un numero di risposte negative significativamente superiore alla media.

$$sum(rate(rest\_client\_requests\_total\{code=\tilde{}"[45].."\}\ [3m]\ )) > THRESHOLD.$$

3.2.4 Riepilogo metriche analizzate

Le metriche che hanno rilevato un comportamento anomalo in uno o più casi di errore, presenti nel subset analizzato, sono state riepilogate nella tabella in fig. 3.17, dove sono mappate le metriche, le tipologie di errore e i workload. Sono state segnate con un tick i casi in cui le metriche risultano essere utili da monitorare e sulla quale eventualmente applicarci regole di alerting.

	Prometheus Metrics																	
	Spazio file system e memoria			capacity offset memoria			utilizzo cpu			metriche di rete			stato pod e container			workqueue		
Wokload	Availab	Deploy	Scale	Availab	Deploy	Scale	Availab	Deploy	Scale	Availab	Deploy	Scale	Availab	Deploy	Scale	Availab	Deploy	Scale
Error type																		
LeR	✓	✓	✓										✓			✓		
MoR	✓	✓	✓				✓	✓	✓				✓	✓	✓	✓	✓	✓
Net	✓	✓	✓	✓	✓	✓					✓							

Figure 3.17: mappa riepilogativa metriche - errori - workload

Chapter 4

Conclusioni

Nei moderni sistemi il **monitoraggio pro-attivo** combinato con il meccanismo di **alerting** risulta essere un processo di fondamentale importanza che permette di attivarsi repentinamente per risolvere situazioni critiche o fare in modo che non si verifichino.

In particolare è stato approfondito **Prometheus**, un framework che integra entrambe queste tipologie di processi, permettendo di monitorare l'intero sistema target, a partire dallo stato delle macchine su cui viene eseguito, nonché lo stato di uno o più cluster Kubernetes e dei relativi oggetti o monitorare un'applicazione in maniera più dettagliata utilizzando le apposite client library.

I dati raccolti durante il monitoraggio sono salvati in maniera **persistente**, in modo da poterli successivamente utilizzare per effettuare delle **interrogazioni** tramite l'apposito linguaggio **PromQL**, che fa da standard per diverse applicazioni di terze parti come Grafana, che

permettono di rappresentare i dati in maniera grafica, oppure per l'Alert Manager, con il quale possiamo creare delle apposite **regole di alerting** come ad esempio per l'eccessivo utilizzo delle risorse negli ultimi minuti.

Sfruttando alcune delle metriche messe a disposizione da Prometheus durante la campagna di fault / error injection del progetto Mutiny, è stato osservato come avendo la possibilità di monitorare un ampio spettro di risorse, sia possibile riuscire a verificare quando una delle possibili condizioni di errore stia per verificarsi, dato che ognuna di esse mostrerà spesso dei **sintomi** che possono essere utilizzati per l'alerting.

Inoltre, è stato possibile effettuare una seconda verifica su uno degli insights del paper [16], ovvero che la condizione **MoR** risulta essere la **più problematica** dato che un sovraccarico delle risorse può portare ad un aumento dei costi in un ambiente cloud o a terminarle mandando in crash il sistema, infatti risulta anche essere la condizione che generalmente **causa più sintomi**.

Bibliografia

[1] EZ BioCloud. https://help.ezbiocloud.net/box-plot/, 09/2024.

[2] Canonical. Microk8s. https://microk8s.io/, 09/2024.

[3] Canonical. Multipass. https://multipass.run/, 09/2024.

[4] Canonical. Snap. https://snapcraft.io/, 09/2024.

[5] Canonical. Ubuntu. https://www.ubuntu-it.org/, 09/2024.

[6] CNCF. Kubernetes. https://kubernetes.io/it/, 09/2024.

[7] CNCF. Prometheus. https://prometheus.io/, 09/2024.

[8] Arun Lal. Bibin Wilson. DevOpsCube. Prometheus architecture. https://devopscube.com/prometheus-architecture/, 09/2024.

[9] Marcello Esposito. Dockerhub sensor app images. https://hub.docker.com/repository/docker/moriarty2002/flask-sensors/, 09/2024.

[10] Marcello Esposito. Github repository. https://github.com/Moriarty2002/thesis/, 09/2024.

[11] Node exporter. https://github.com/prometheus/node_exporter/, 09/2024.

[12] Cloud Native Computing Foundation. https://www.cncf.io/, 09/2024.

[13] Grafana. https://grafana.com/, 09/2024.

[14] Graphite. https://graphiteapp.org/, 09/2024.

[15] InfluxDB. InfluxDB, 09/2024.

[16] Marco Barletta. Marcello Cinque. Catello Di Martino. Zbigniew T. Kalbarczyk. Ravishankar K. Iyer. Mutiny! how does kubernetes fail, and what can we do about it? *arXiv*, 2024.

[17] Kube-prometheus. https://github.com/prometheus-operator/kube-prometheus/, 09/2024.

[18] Matplotlib library. https://matplotlib.org/, 09/2024.

[19] Pandas library. https://pandas.pydata.org/, 09/2024.

[20] Seaborn library. https://seaborn.pydata.org/, 09/2024.

[21] Martin Kaschke. Medium. Vm vs container. https://mkaschke.medium.com/virtual-machine-vm-vs-container-13ab51f4c177/, 09/2024.

[22] Ashish Patel. DevOps Mojo. K8s architecture. https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-
09/2024.

[23] Jupyter notebook. https://jupyter.org/, 09/2024.

[24] Prometheus. Architecture. https://prometheus.io/docs/instrumenting/exporters/-https://prometheus.io/docs/instrumenting/clientlibs/, 09/2024.

[25] Prometheus. Operator. https://github.com/prometheus-operator/prometheus-operator/, 09/2024.

[26] Prometheus. Query language. https://prometheus.io/docs/prometheus/latest/querying/basics/, 09/2024.

[27] SoundCloud. https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud, 09/2024.

[28] Kube state metrics. https://github.com/kubernetes/kube-state-metrics/, 09/2024.

[29] Rahul Awati. TechTarget. https://www.techtarget.com/searchbusinessanalytics/definition/heat-map, 09/2024.

[30] VictoriaMetrics. https://victoriametrics.com/, 09/2024.

[31] VirtualBox. https://www.virtualbox.org/, 09/2024.