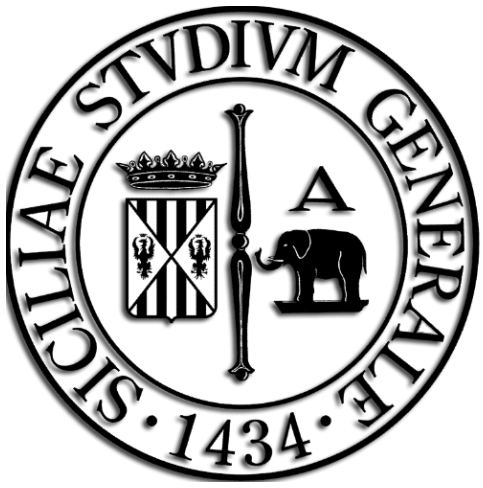

Università degli Studi di Catania

RELAZIONE PROGETTO DSBD "MYHUB"

Corso di laurea magistrale in Ingegneria Informatica

"DISTRIBUTED SYSTEMS AND BIG DATA"

Anno Accademico 2021/2022

Studenti:

Nasca Prospero 1000025183

Valastro Antonio 1000025222

Sommario

Indice delle figure 2
Introduzione 4
MyHub 5
Microservizi 5
Comunicazione 6
Operazioni effettuate 7
Pattern SAGA 8
Monitoring 10
Prometheus e Grafana 11
Settaggio dei tools 11
Analisi delle Statistiche 12
Time series e predizione 13

Indice delle figure

Figura 1 - Monolith vs Microservices 4
Figura 2 - Api Gateway 4
Figura 3 - Database per service 5
Figura 4 - Schema dei microservizi 6
Figura 5 - Diagramma di sequenza per aggiungere un follow 7
Figura 6 - Diagramma di sequenza per effettuare il login 7
Figura 7 - Saga per la creazione del Post 8
Figura 8 - Diagramma di sequenza per la creazione del post 9
Figura 9 - Diagramma a stati SAGA 9
Figura 10 - Prometheus e Grafana schema 10
Figura 11 - CPU usage by pods 12
Figura 12 - CPU usage by namespace 12
Figura 13 - CPU usage by node 12
Figura 14 - Memory usage by pods 13
Figura 15 - Memory usage by namespace 13
Figura 16 - Trend,Stagionalita e Residui (Login) 14
Figura 17 - Test Prediction Login 14
Figura 18 - Future Prediction Login 14

Figura 19 - Trend,Stagionalita e Residui (Post) 14
Figura 20 - Test Prediction Post 14
Figura 21 - Future Predictiom Post 14
Figura 22 - Trend,Stagionalita e Residui (Follow) 14
Figura 23 - Test Prediction Follow 14
Figura 24 - Future Prediction Follow 14

Introduzione

Il progetto verte sulla realizzazione di una Web App sviluppata attraverso l'utilizzo della piattaforma software *Docker*, che permette di creare e distribuire le applicazioni con la massima rapidità, attraverso l'utilizzo di "container", che offrono un ambiente necessario per l'esecuzione del software, includendo librerie, strumenti di sistema, etc., insieme all'utilizzo della piattaforma open-source *Kubernetes*, un *orchestrator*, noto anche come *k8s*, che consente di automatizzare le operazioni dei container e di eliminare molti processi manuali coinvolti nel deployment e nella scalabilità delle applicazioni.
Il problema da risolvere è la gestione di un servizio di pubblicazione e lettura di articoli scientifici, in particolare informatici, pubblicati da utenti registrati alla piattaforma, chiamata **MyHub**, la piattaforma deve permettere la registrazione, l'accesso, la pubblicazione e la lettura degli articoli propri e degli amici seguiti.

La logica applicativa verrà suddivisa in microservizi indipendenti di piccole dimensioni, che comunicano fra loro tramite API ben definite, in maniera tale da permettere di avere un software disaccoppiato, ciascun processo applicativo è visto come un servizio, ciò permette in maniera più agevole di effettuare aggiornamenti e successivi deployment, oltre che essere resiliente a temporanei malfunzionamenti della rete, in quanto i microservizi sono più facilmente replicabili rispetto a un applicazione monolitica, quindi nel caso di malfunzionamento di un servizio, un eventuale richiesta potrebbe essere reindirizzata verso una replica dello stesso.

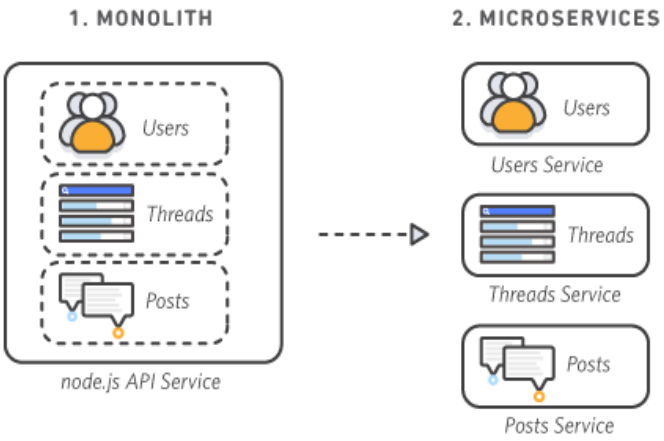

Figura 1 - Monolith vs Microservices

I microservizi comunicano tra loro attraverso REST API, approccio architetturale ideato per creare web API basandosi sul protocollo HTTP.
Il REST è infatti un sistema di trasmissione dei dati che utilizza pienamente le funzioni di HTTP, ricorrendo ai comandi GET, POST, PUT, PATCH e DELETE e OPTIONS.
Le principali proprietà di REST sono:

- Client-Server: in questo caso si parla di separazione dei compiti (SoC)
- Stateless: questo principio si basa sull'assenza di stato durante la comunicazione tra server e client

Tali richieste vengono fatte dal servizio di frontend verso il backend di un determinato microservizio tramite query AJAX, che permette lo scambio dei dati in background tramite una chiamata asincrona, utilizzando l'oggetto XMLHttpRequest.
Dal momento che sono presenti diversi microservizi, è stato pensato un Api Gateway che gestisce il flusso che si interfaccia con il servizio di backend scelto, applica le politiche, l'autenticazione e il controllo generale dell'accesso per le chiamate API; progettato per ottimizzare la comunicazione tra i clienti esterni e i servizi, assicura la scalabilità e l'alta disponibilità dei servizi.
È responsabile dell'instradamento della richiesta al servizio appropriato e dell'invio di una risposta al richiedente.

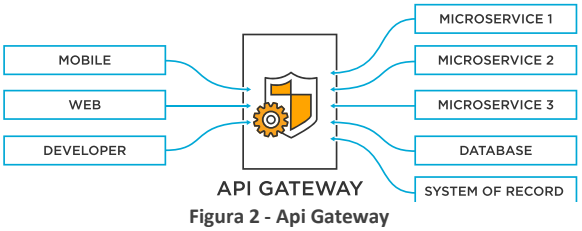

Figura 2 - Api Gateway

MyHub

MyHub è costituito da 4 microservizi indipendenti: un frontend e 3 backend ognuno con il proprio database, per rispettare il pattern *"database per service",* infatti, una delle caratteristiche principali dell'architettura a microservizi è il basso accoppiamento tra i servizi, per questo motivo ogni servizio deve avere i propri database, così eventuali modifiche a un database non influiscono sugli altri microservizi.

Non è possibile accedere al database del servizio direttamente da altri microservizi. È possibile accedere ai dati persistenti di ciascun servizio solo tramite le API REST . Quindi un database per microservizio offre molti vantaggi quali: evolversi rapidamente e supportare sistemi su vasta scala.

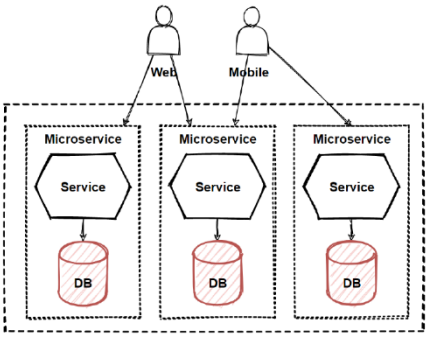

Figura 3 - Database per service

Microservizi

I microservizi di backend sono scritti in Python utilizzando il framework Flask, adatto a sviluppare applicazioni Web in linguaggio di programmazione Python, mentre il frontend è scritto in HTML con l'integrazione di CSS, Javascript e AJAX. Ognuno svolge un compito specifico:

- User: si occupa di effettuare la registrazione e l'accesso dell'utente che intende utilizzare la web app interfacciandosi con un database *MySQL* attraverso il driver *"MySql Connector"* e risponde alle richieste di altri microservizi quali:
    - Aggiornare (incrementando o decrementando) una variabile che si occupa di tenere il conteggio del numero di post effettuati da ciascun utente.
    - Restituire l'id o l'username dell'utente loggato.
- Post: è deputato alla creazione di un post e alla visualizzazione dei post nella home del frontend.
  Un post può essere creato con diverse opzioni, sarà necessario un titolo, un URL e una descrizione. È opzionale invece il campo tag per poter taggare un utente che seguiamo.
  Tale microservizio si interfaccia con un database *MongoDB* non relazionale, attraverso il driver *"Flask-Mongo"*.
- Follow: si occupa di seguire gli utenti interfacciandosi con un database *MySQL*, attraverso il driver *"MySql Connector"* e di mostrare nel frontend gli utenti come suggerimento di amicizia.
- Frontend: è deputato alla navigazione web della nostra app, sono presenti diverse sezioni:
    - L'interfaccia di Login/Registrazione
    - La home per visualizzare i post effettuati dallo user loggato e dagli amici.
    - Il tab per creare il post.
    - Il tab per seguire gli amici.
    - Il pulsante per effettuare il log-out.

  È presente la gestione della variabile di sessione in cui memorizziamo l'id dell'utente loggato tramite Javascript, tale id verrà inserito nei data del body di ogni richiesta AJAX effettuata verso il determinato backend.

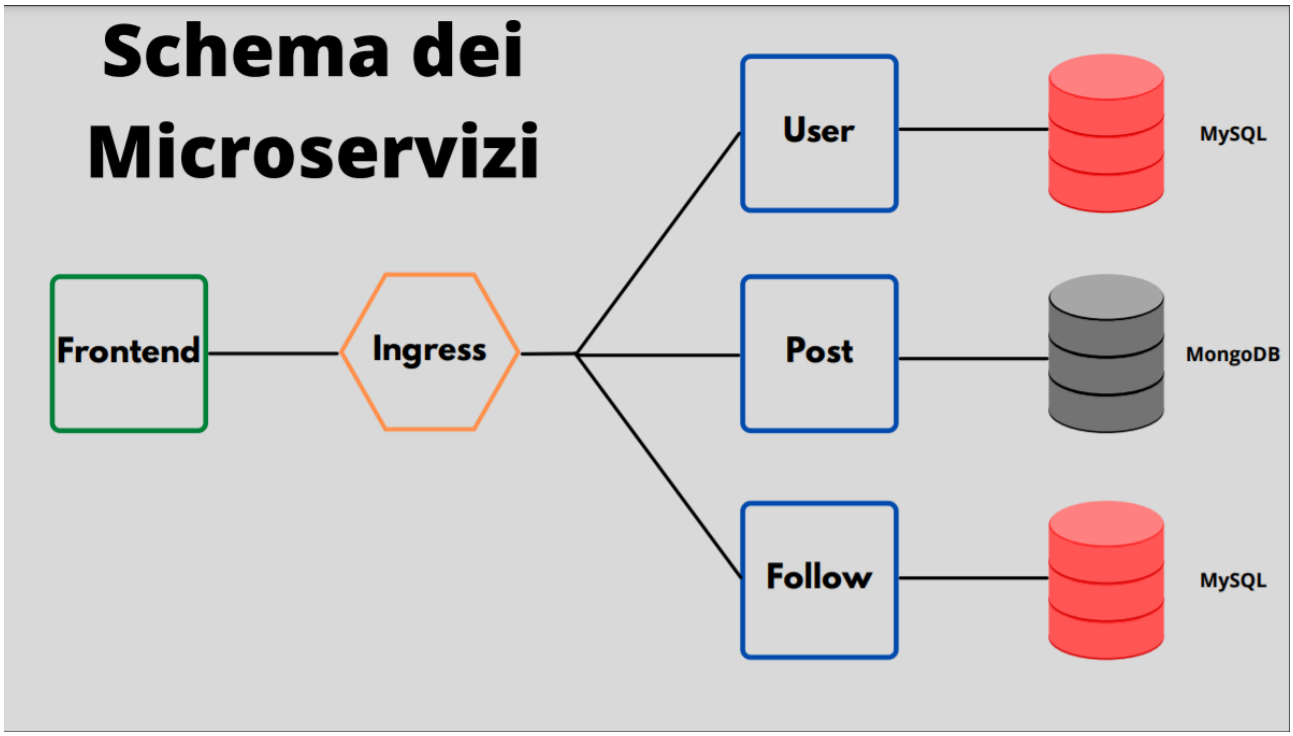

Figura 4 - Schema dei microservizi

Comunicazione

Una tipica richiesta AJAX per la comunicazione è siffatta:

"http://myproject.loc/post/createPost" in cui "myproject.loc" è riferito all'indirizzo IP di k8s, nel file etc/host del sistema locale utilizzando, nel nostro caso Ubuntu versione 21.10 in Virtual Machine tramite "virtualbox" è stata aggiunta una riga composta da "IP_K8s myprogect.loc" per permettere di effettuare la risoluzione dei nomi tramite DNS.

All'interno di ogni microservizio è presente un file nginix.py che si occupa della risoluzione delle route di ciascuna richiesta, dato il path "…/post/createPost", tale file Python elimina la parte "/post" e richiama la route "/createPost" all'interno del microservizio "Post.py".

Ciò è fatto per ogni microservizio di backend, invece per il frontend è presente un "nginx" che effettua il reverse-proxy, modificando le intestazioni delle richieste attraverso la configurazione espressa attraverso la "location" in cui con "http://myproject.loc/", si effettua il passaggio di tale richiesta con "/" all'indirizzo "user/login". Questo indirizzo può essere specificato come nome di dominio o indirizzo IP.

Operazioni effettuate

Di seguito verranno mostrati i diagrammi di sequenza dell'operazione di follow e di login

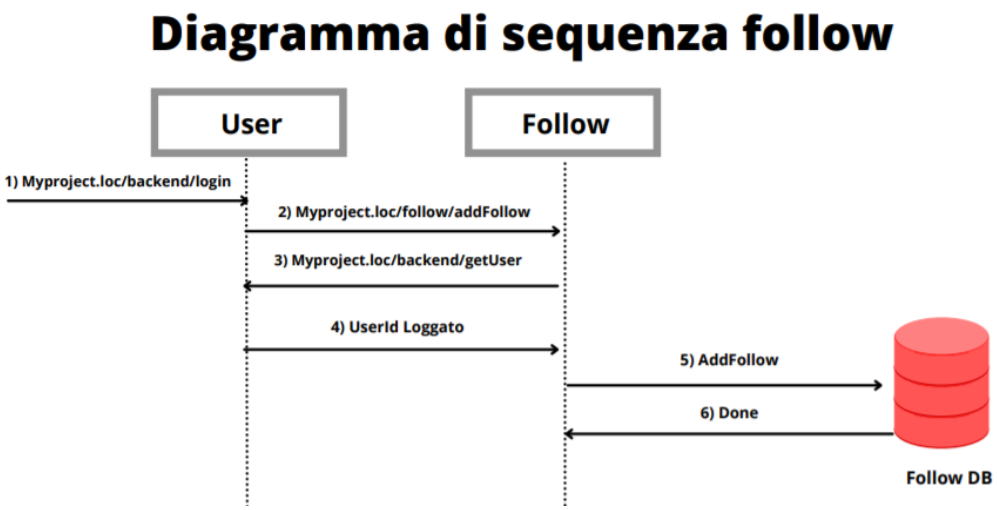

Figura 5 - Diagramma di sequenza per aggiungere un follow

Com'è possibile notare per aggiungere un amico come follow prima bisogna essere loggati; quindi, si esegue il login e solo successivamente si può aggiungere un follow, il microservizio salverà nel database MySql l'id dell'utente loggato insieme all'id dell'utente da seguire.

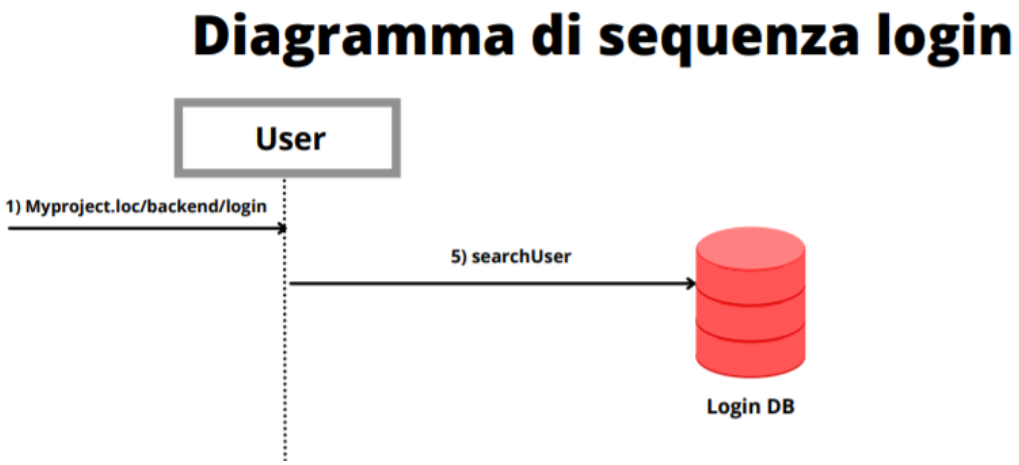

Figura 6 - Diagramma di sequenza per effettuare il login

Per effettuare il login, inserendo le credenziali verrà controllato all'interno del DB l'esistenza dell'username e della password, solo successivamente verrà dato l'accesso e impostata la variabile di sessione.

Pattern SAGA

Nell'ultima operazione possiamo osservare la creazione del post dove entra in atto il pattern SAGA

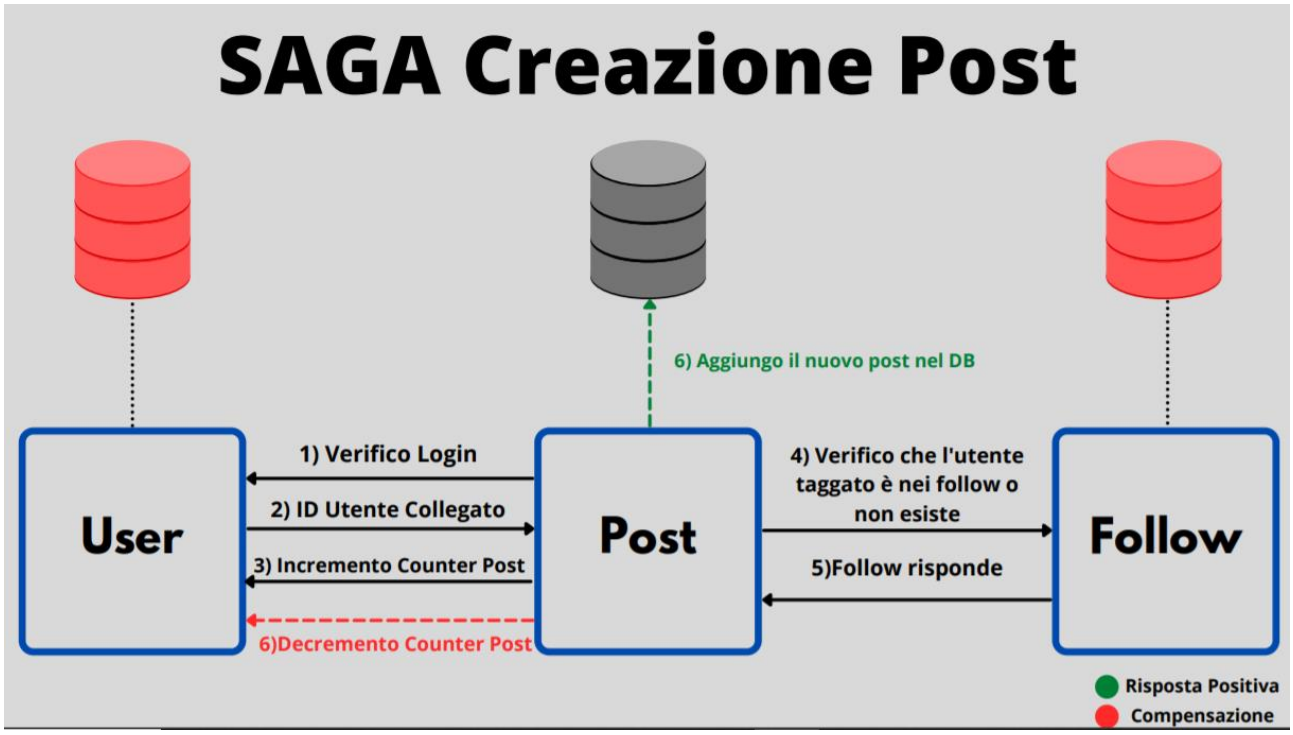

Figura 7 - Saga per la creazione del Post

L'idea alla base del Pattern è la gestione di transazioni distribuite, dall'inizio alla fine, ognuna delle quali è, appunto, una saga.

Lo scopo è garantire l'ordine di esecuzione delle transazioni che fanno parte della stessa saga, ovvero che devono essere eseguite in un ordine prestabilito e il cui effetto non è compromesso da eventuali transazioni intermedie appartenenti a saghe diverse.

Questo pattern aiuta quindi a gestire la consistenza dei dati nell'esecuzione di transazioni distribuite tra microservizi diversi; coinvolge diversi attori (servizi) che vanno ad agire su una stessa entità tramite singole transazioni atte all'aggiornamento di un dato comune.

L'obiettivo è duplice: mantenere l'identità del dato ed effettuare azioni di compensazione per ripristinarlo in caso di errore.

Esistono principalmente due approcci per gestire il pattern in esame:

- events/choreography → nessun coordinatore, i servizi portano avanti la saga senza avere qualcuno che li controlli, ma lavorando insieme
- commands/orchestration → controllo centralizzato, gestito da un orchestratore

Nel nostro caso la SAGA è una choreography e viene applicata quando un utente vuole creare un post e taggare un amico, vediamo i passaggi con un diagramma di sequenza:

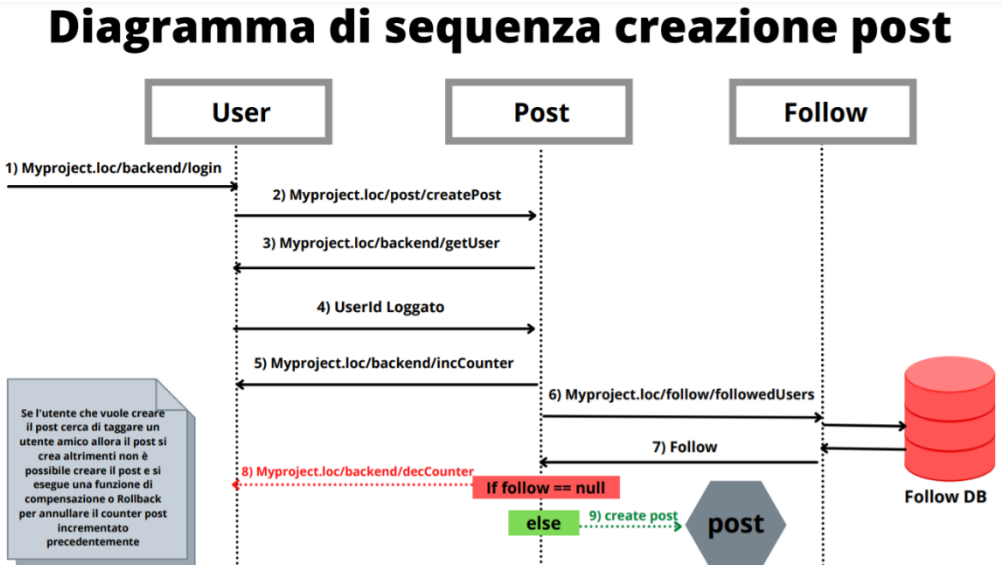

Figura 8 - Diagramma di sequenza per la creazione del post

1. L'utente che deve creare il post deve essere loggato, dunque viene richiesto al microservizio user l'id dell'utente
2. Tramite la variabile di sessione impostata da User se l'utente è loggato si recupera facilmente l'id
3. Il microservizio di post invia una richiesta API asincrona al microservizio User per aggiornare il numero di post effettuati dall'utente, incrementando una variabile presente nel database associata all'utente
4. Per effettuare il tag bisogna prima verificare che l'utente loggato sia veramente amico dell'utente da taggare, per questo si fa una richiesta al microservizio Follow

Da qui possiamo avere due comportamenti:

- Se l'utente segue l'amico allora il post viene confermato e salvato nel database Mongo
- Se l'utente non segue l'amico non può creare il post e di conseguenza si effettuerà una funzione di rollback attraverso un'API verso il microservizio User che effettua la compensazione per annullare la modifica del numero di post effettuati, decrementando la variabile "counter_post" nel database.

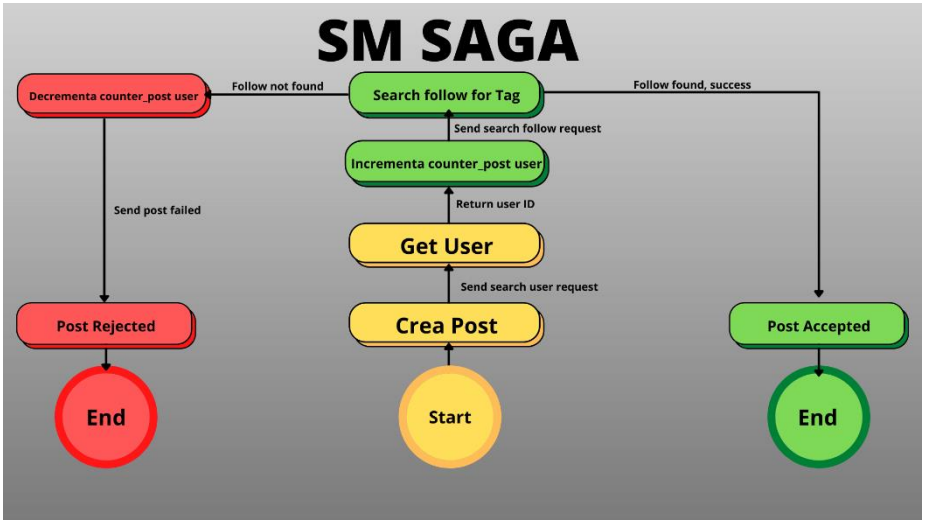

Figura 9 - Diagramma a stati SAGA

Monitoring

Per quanto riguarda il monitoring tra le possibilità disponibili: black-box e white-box, si esegue un black-box monitoring attraverso Prometheus e Grafana sviluppati all'interno di un cluster k8s in un namespace dedicato.

Il black box monitoring si riferisce al monitoraggio dei server con particolare attenzione allo spazio occupato su disco, all'utilizzo della CPU, all'utilizzo della memoria, alle medie di carico, ecc.

Questi sono le metriche di un sistema standard da monitorare.

Il black box monitoring è necessario per rilevare alcuni problemi con i sistemi, come un carico elevato o un traffico di rete elevato.

Vantaggi del monitoraggio Black Box:

- Migliore visione d'insieme.
- Ottimizzazione il codice.
- Introspezione dei programmatori, consapevolezza delle loro azioni.
- Consente di trovare errori nascosti.
- Efficienza nel trovare errori e problemi.

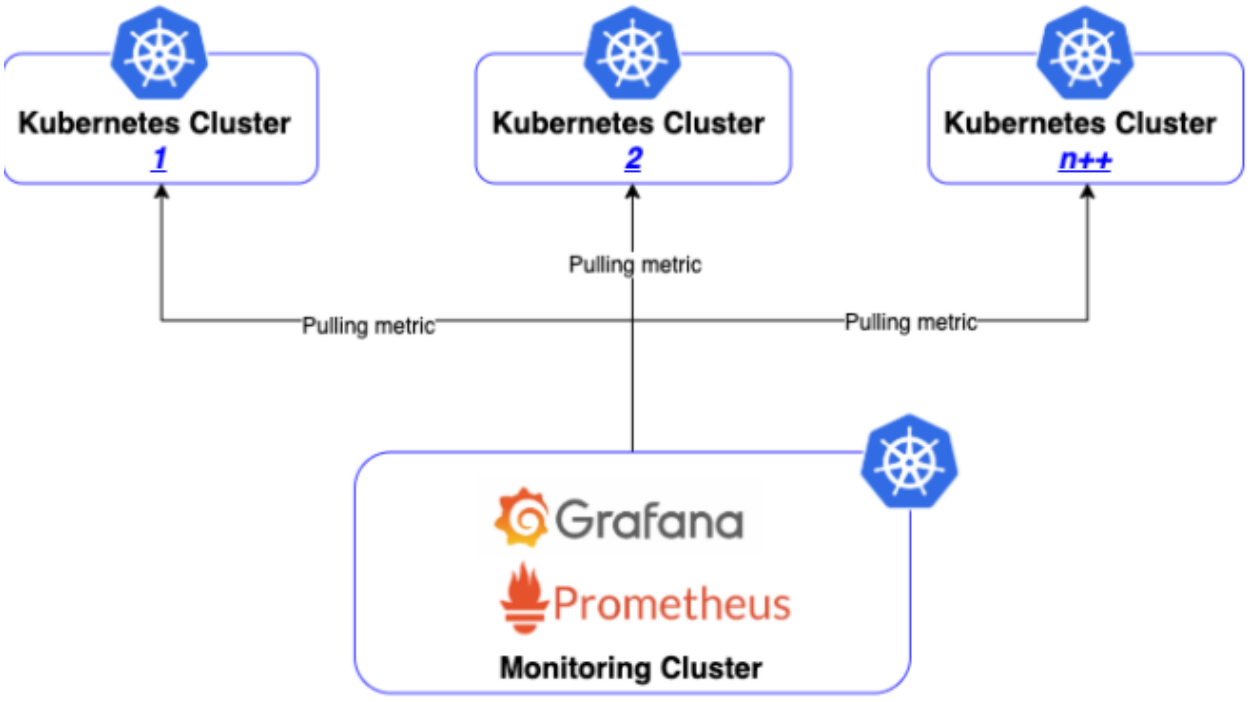

Figura 10 - Prometheus e Grafana schema

Prometheus e Grafana

Prometheus è un tool di monitoraggio e alerting che archivia metriche e Time Series.

Una Time Serie, o serie temporale, è una serie di data point indicizzati (elencati o rappresentati in un grafico) in ordine temporale. Più comunemente, una serie temporale è una sequenza presa in punti successivi ugualmente distanziati nel tempo.

Tramite il linguaggio di query specifico *"PromQL" (*Prometheus Query Language), è possibile recuperare/aggregare i dati accumulati.

Il recupero delle metriche viene fatto in *pull*, ovvero sarà Prometheus stesso a contattare i singoli microservizi i quali non dovranno far altro che esporre la propria lista di metriche.

Grafana mette a disposizione un data source che permette di accedere in maniera semplice a tutti i dati immagazzinati da Prometheus.

Una volta collegati tra loro Prometheus e Grafana, si possono creare delle query parametrizzabili e visualizzarne i risultati in comodi grafici mettendo in relazione fra loro le varie metriche.

È utile fare utilizzo di questi due sistemi per monitorare in tempo reale il consumo di CPU/RAM di ogni singolo servizio di un'architettura a microservizi.

I dati raccolti, messi in relazione tra loro, permettono di analizzare alcuni malfunzionamenti e risalire alla loro causa.

Settaggio dei tools

Per sviluppare Prometheus e Grafana sono stati creati i file *".yaml"* del deployment e del service all'interno di un namespace dedicato al monitoring.

Prometheus prenderà la sua configurazione da una Config Map, questo ci permette di aggiornare la configurazione separatamente dall'immagine.

La parte rilevante della configurazione è in data/prometheus.yml, è una configurazione di Prometheus incorporata nel manifesto di k8s.

Nella sezione dei metadati del deployment di Prometheus, diamo al pod un'etichetta con una chiave che ha un nome e un valore.

Nelle annotazioni, impostiamo un paio di coppie chiave/valore che consentiranno effettivamente a Prometheus l' *"autodiscover"* e lo *"scrape".*

Stiamo usando un volume *"emptyDir"* per i dati di Prometheus. Questa è fondamentalmente una directory temporanea che verrà cancellata ad ogni riavvio del contenitore.

Analisi delle Statistiche

Sono state raccolte diverse statistiche sotto forma di grafici riguardo la CPU utilizzata dai vari pod, dai node e dai namespace e la memoria utilizzata dai pod, dai node e dai namespace.

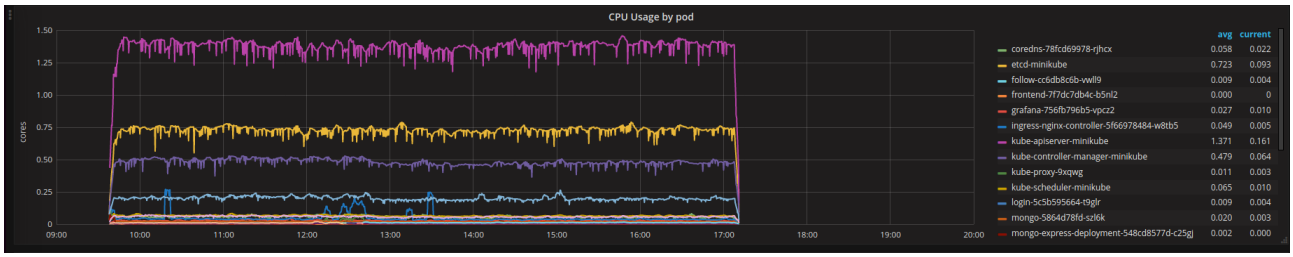

Figura 11 - CPU usage by pods

Vediamo che tra i vari pod, quelli interni di k8s consumano tanta CPU, poiché siamo su macchina virtuale k8s consuma tanta risorsa CPU. I nostri microservizi invece consumano poca CPU, sebbene si possano notare dei piccoli incrementi corrispondenti ai momenti in cui abbiamo inviato delle richieste ai servizi.

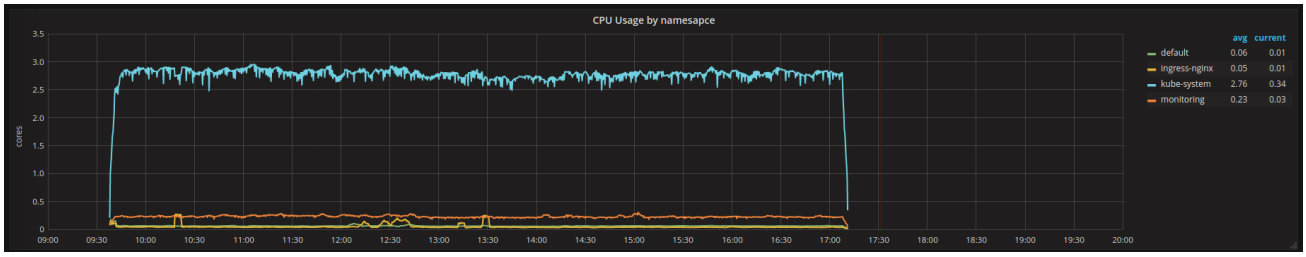

Figura 12 - CPU usage by namespace

Il namespace che consuma più CPU è quello relativo a k8s, avendo solo un nodo in definitiva il consumo della CPU è quasi pari a 4 cores, il massimo che avevamo riservato per far partire k8s.

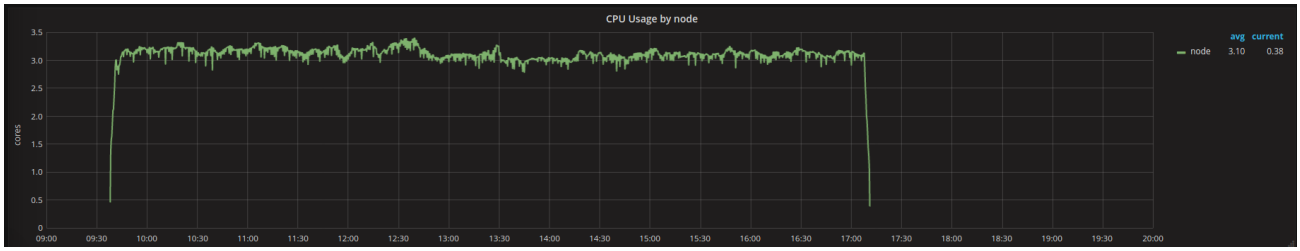

Figura 13 - CPU usage by node

In termini di memoria consumata invece, possiamo vedere che il database MySql per il login consuma quasi 1 giga di spazio su disco, ricordiamo che il “persistent volume claim” è stato creato con le dimensioni di 1 giga; mentre Mongo consuma solo 300 mega.

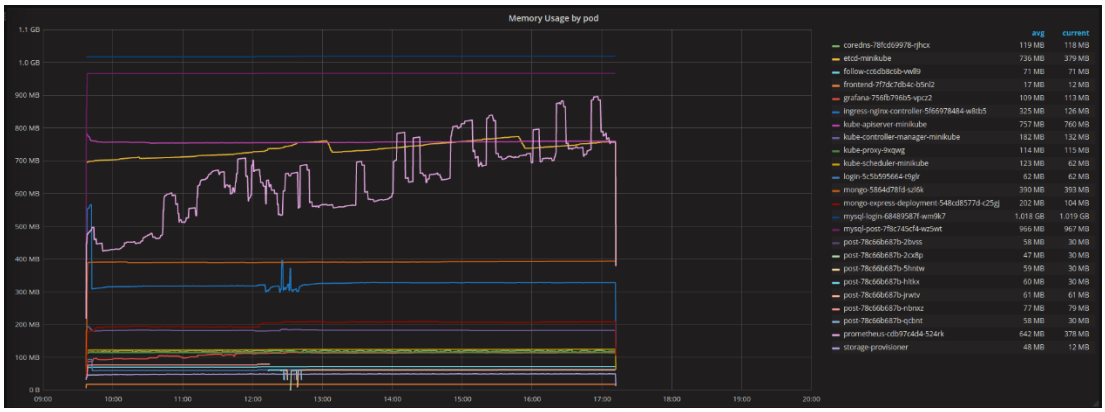

Figura 14 - Memory usage by pods

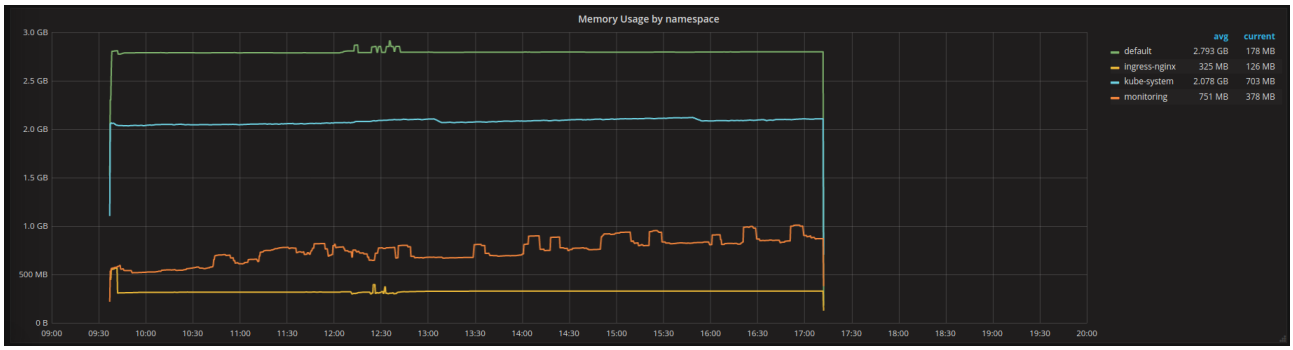

Figura 15 - Memory usage by namespace

In definitiva lo spazio totale consumato dal sistema è 5 giga, considerando anche il namespace del monitoring in cui ci sono Prometheus e Grafana.

Time series e predizione

Per lo sviluppo di un predittore è stato scelto il metodo Holt-Winters e sono state riformulate le query da Grafana per isolare in un breve arco di tempo di 20 minuti, il comportamento di un solo pod per volta per quando riguarda l'uso della CPU.

Lo studio di una serie storica comprende la decomposizione in componenti di trend e stagionalità, allo scopo di effettuare previsioni su osservazioni future (o comunque non misurate).

Abbiamo accennato e sfruttato la decomposizione tramite media mobile, Il comando decompose, richiede come input un vettore di numeri reali (la serie storica) come oggetto time-series.

La decomposizione è tanto migliore quanto i residui somigliano al white noise gaussiano e hanno una media nulla

Il metodo moltiplicativo di Holt-Winters consente inoltre di calcolare valori con smoothing esponenziale per livello e trend, nonché adeguamenti stagionali per le previsioni. Il metodo moltiplicativo stagionale moltiplica la previsione basata sul trend per la componente stagionale, producendo la previsione moltiplicativa di Holt-Winters.

Questo è il metodo migliore per dati con una componente stagionale che aumenta nel tempo. Il risultato è una curva di previsione che riproduce le variazioni stagionali dei dati.

Predizione Login

Di seguito vengono riportati gli screen ottenuti dalle statistiche ricavate sull'uso della CPU da parte del pod "Login", abbiamo trovato che la seasonability della serie è 8, i residui sono a media nulla e le predizioni sono abbastanza accurate seppur su un numero ristretto di campioni, risultato prevedibile dato il poco tempo di simulazione per allenare la rete. Tuttavia, per la natura della statistica, ovvero l'utilizzo della CPU, è accettabile avere una previsione nel prossimo minuto senza pretendere previsioni sul lungo periodo.

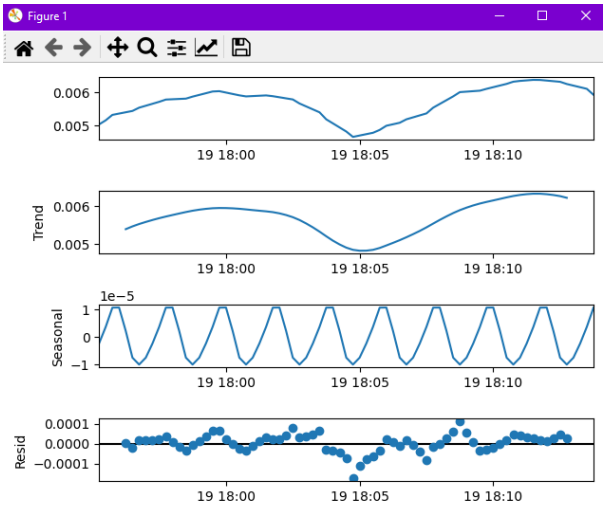

Figura 16 - Trend,Stagionalita e Residui (Login)

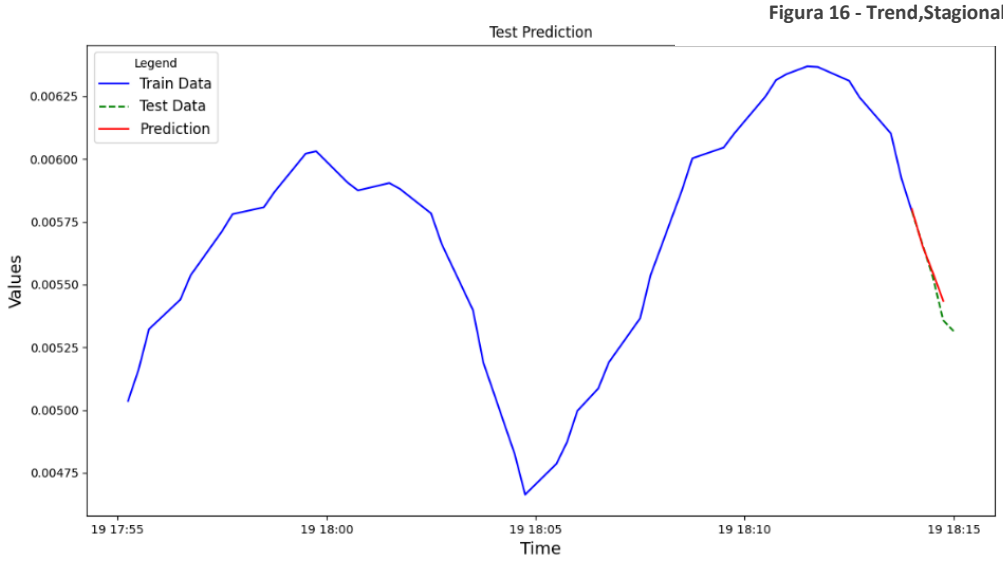

Figura 17 - Test Prediction Login

I risultati sui test data risultano abbastanza precisi e dunque se vogliamo fare una predizione futura ecco il risultato che otteniamo:

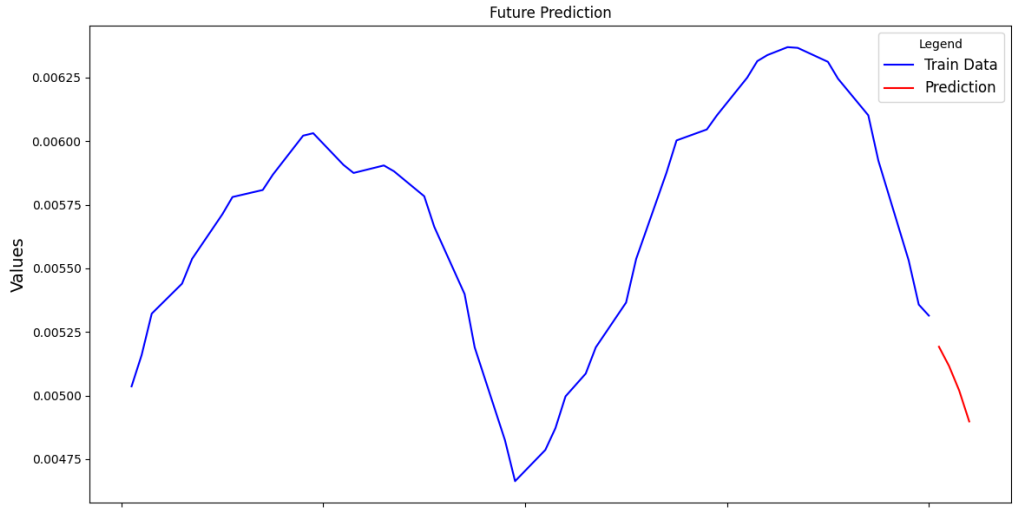

Figura 18 - Future Prediction Login

Predizione Post

Di seguito vengono riportati gli screen ottenuti dalle statistiche ricavate sull'uso della CPU da parte del pod "Post", abbiamo trovato che la seasonability della serie è 28, i residui sono a media nulla e anche in questo caso le predizioni sono abbastanza accurate seppur su un numero ristretto di campioni, risultato prevedibile dato il poco tempo di simulazione per allenare la rete. Tuttavia, per la natura della statistica, ovvero l'utilizzo della CPU, è accettabile avere una previsione nel prossimo minuto senza pretendere previsioni sul lungo periodo.

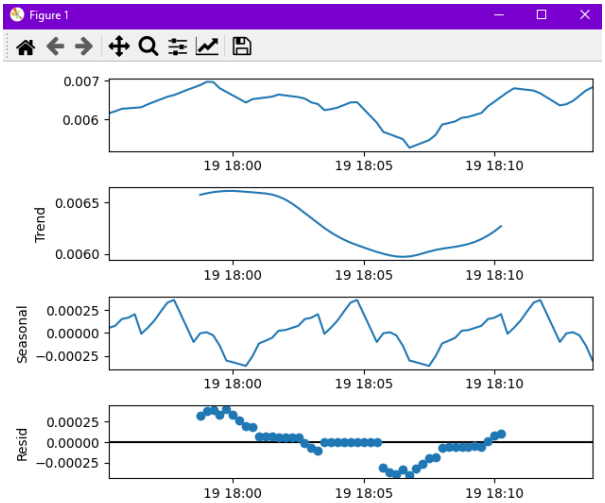

Figura 19 - Trend,Stagionalita e Residui (Post)

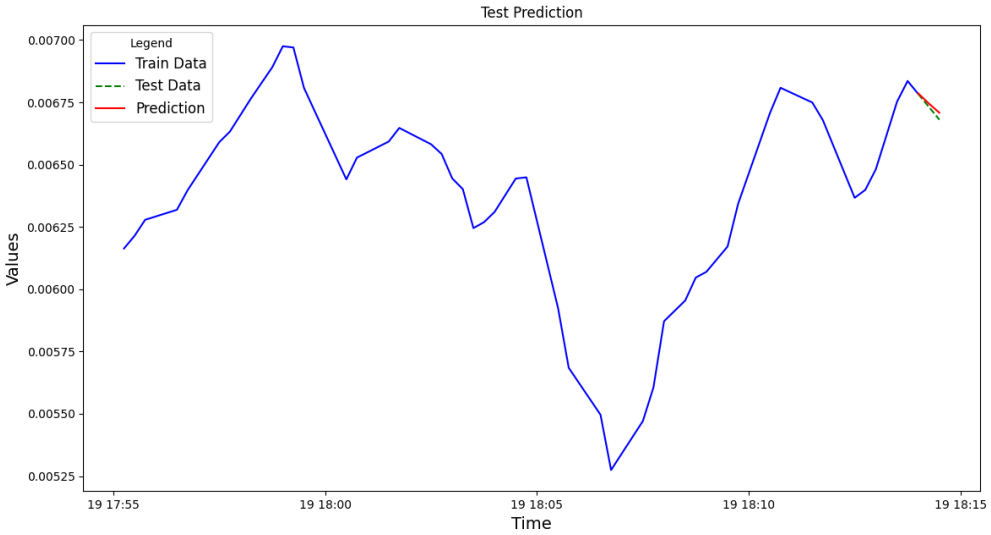

Figura 20 - Test Prediction Post

I risultati sui test data risultano abbastanza precisi e dunque se vogliamo fare una predizione futura ecco il risultato che otteniamo:

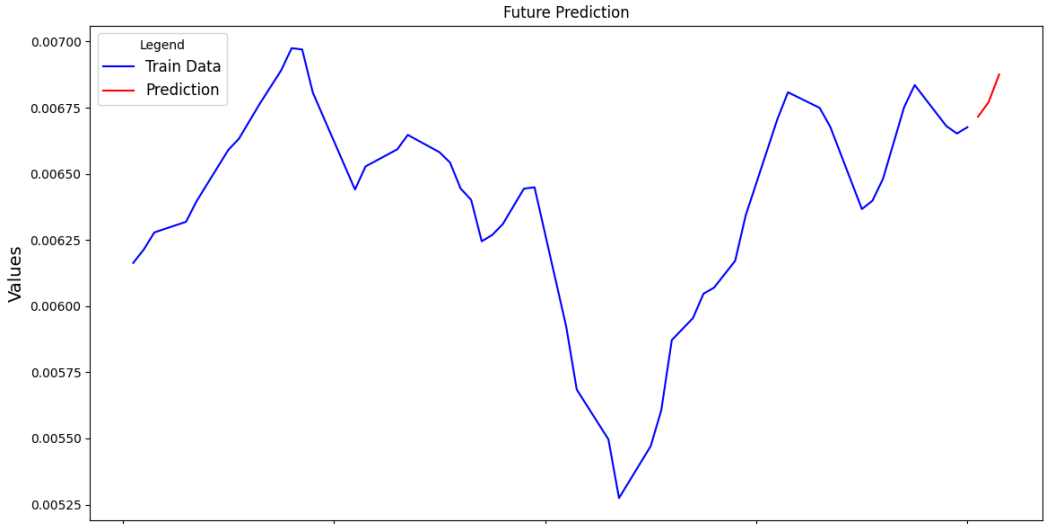

Figura 21 - Future Predictiom Post

Predizione Follow

Di seguito vengono riportati gli screen ottenuti dalle statistiche ricavate sull'uso della CPU da parte del pod "Follow", abbiamo trovato che la seasonability della serie è 12, i residui sono a media nulla e anche in questo caso le predizioni sono abbastanza accurate seppur su un numero ristretto di campioni, risultato prevedibile dato il poco tempo di simulazione per allenare la rete. Tuttavia, per la natura della statistica, ovvero l'utilizzo della CPU, è accettabile avere una previsione nel prossimo minuto senza pretendere previsioni sul lungo periodo.

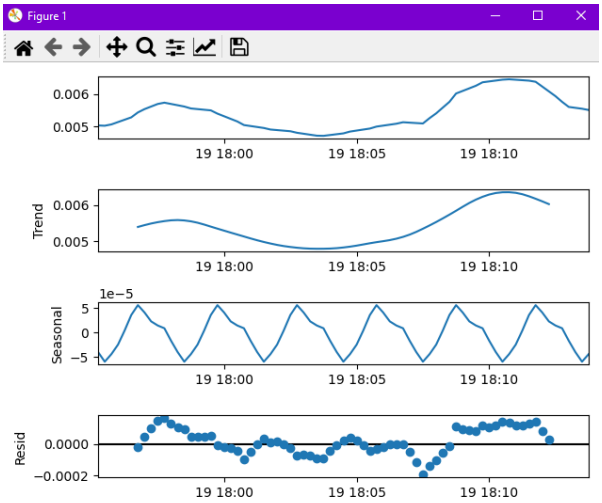

Figura 22 - Trend,Stagionalita e Residui (Follow)

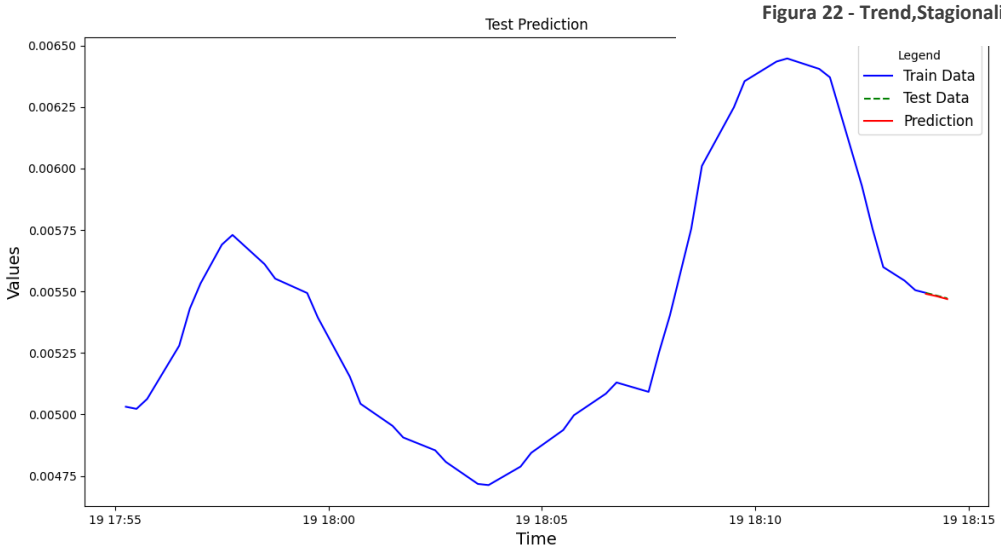

Figura 23 - Test Prediction Follow

I risultati sui test data risultano abbastanza precisi e dunque se vogliamo fare una predizione futura ecco il risultato che otteniamo:

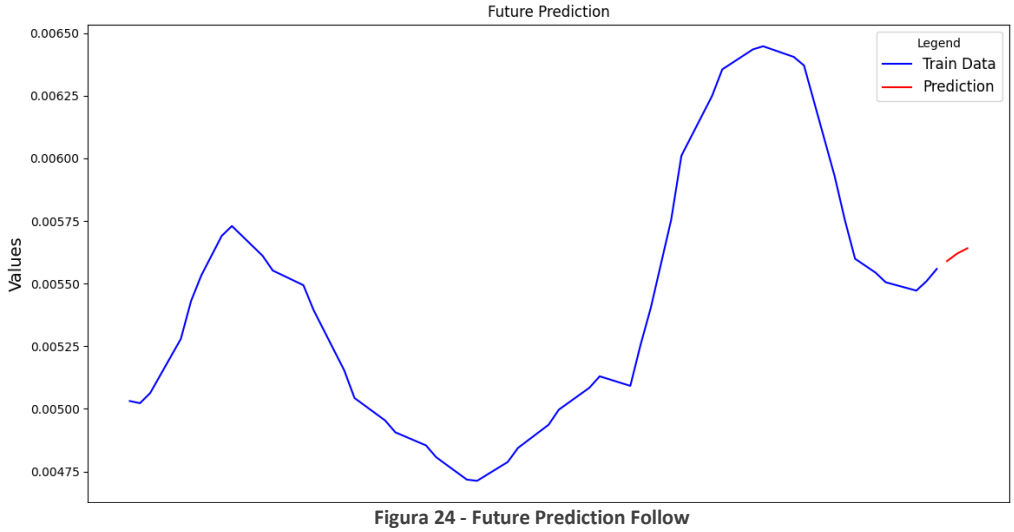

Figura 24 - Future Prediction Follow