Abstract

Il progetto realizzato implementa un file system distribuito che supporta varie operazioni su diverse tipologie di file.

L'obiettivo che il sistema si pone è la massimizzazione del throughput in download.

A questo scopo, durante l'operazione iniziale di upload di un file esterno, il flusso in ingresso viene parallelamente diviso tra gruppi di n macchine interne, chiamate "fileSystems".

Questo ha permesso di avere uno scaling sui dati oltre che orizzontalmente.

Durante la successiva operazione di download sui tre flussi viene applicato l'interleaving allo scopo di ricostruire il file iniziale.

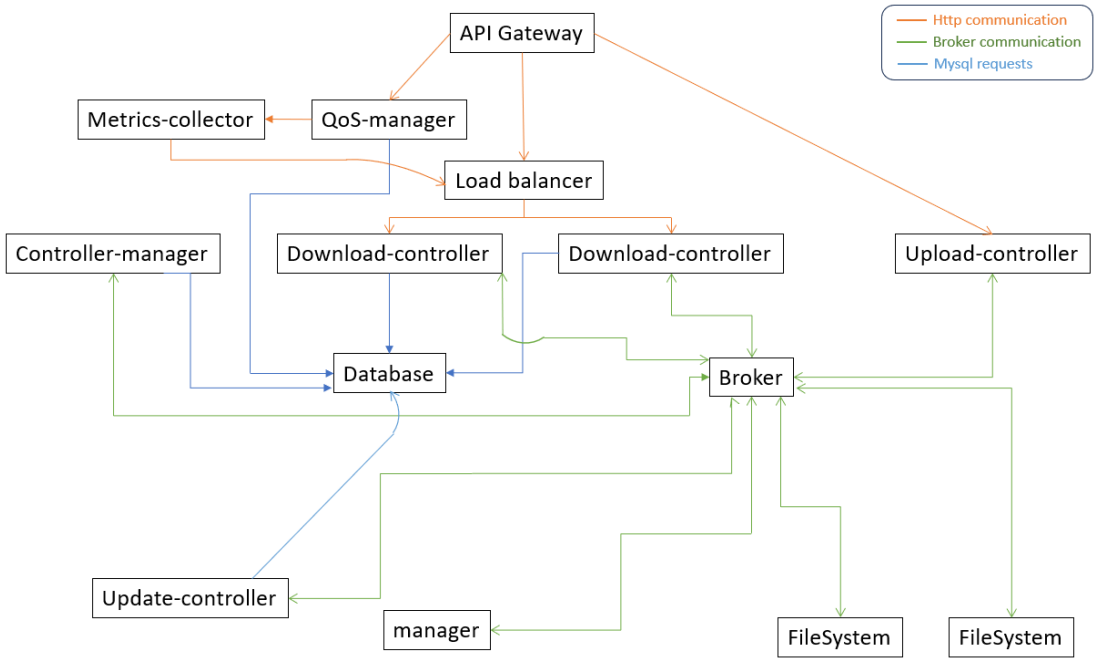
A causa di ciò il throughput tende a salire, ma, allo stesso tempo, aumenta anche la latenza, poiché per l'invio del file è necessario aspettare il flusso trasmesso dal fileSystem più lento.

Attenzioneremo in seguito la gestione della latenza massima con modelli predittivi ARIMA.

Per la comunicazione tra i vari attori del nostro sistema è stata impiegata, quasi nella totalità dei casi, la comunicazione indiretta tramite broker Kafka poiché essa fornisce importanti funzionalità come la possibilità di inviare a vari attori lo stesso messaggio oppure fornire in Round Robin una serie di richieste ad un gruppo di macchine, allo stesso tempo fornisce un meccanismo di garanzia della ricezione dei messaggi tramite un semplice meccanismo di commit e retain.

Per sopperire ai cambiamenti di traffico in download e cercare di mantenere un throughput che permetta una buona usabilità del sistema sono state utilizzate le api dell'orchestratore Kubernetes cercando di scalare orizzontalmente utilizzando previsioni basate su modelli predittivi ARIMA.

In aggiunta è stato implementato un sistema di online learning per aggiornare i modelli predittivi ogni 10 minuti.



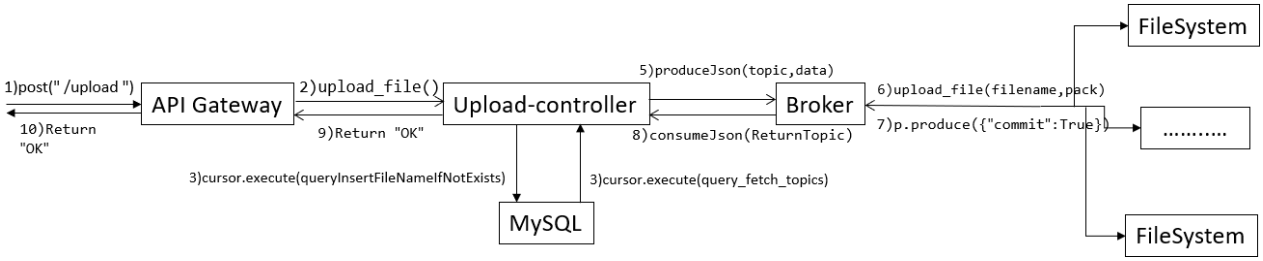
API sviluppate

Durante lo svolgimento del progetto si è deciso di sviluppare quelle che sono le minime API per il nostro sistema allo scopo di poter automatizzare il processo di upload e download.

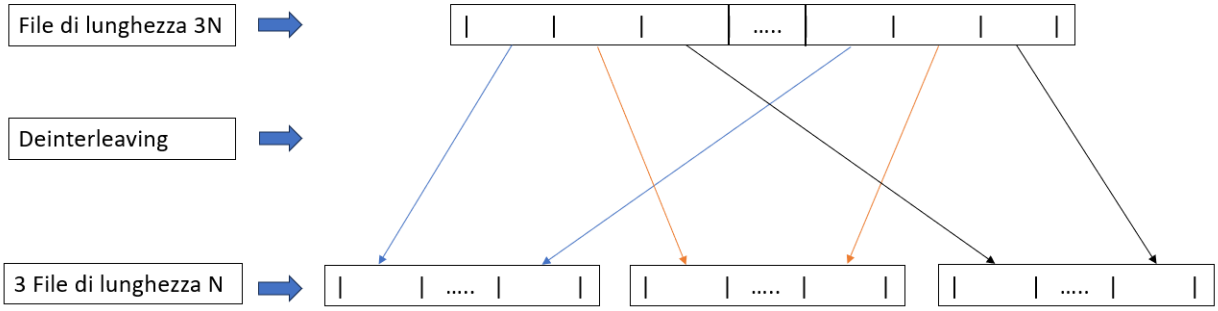
1) API di upload:

La prima api realizzata è quella che permette l'inizio delle attività, cioè l'upload di un file su protocollo HTTP. A questo scopo abbiamo utilizzato il framework Flask per implementare un'interfaccia sulla route "/upload", abilitato a rispondere a messaggi di tipo POST o GET.

La prima opzione è pensata proprio per l'invio del file mentre la seconda fornisce un'interfaccia minimale per l'upload interattivo tramite browser.



Come mostrato in figura per l'operazione di upload il nostro API gateway (nginx) inoltra il messaggio http ad un agente detto Upload-controller: esso controlla se il nome del file è già presente nel nostro database e, in caso contrario, chiede anche quanti topic sono disponibili per l'upload. Dal risultato di questa chiamata si decide in quante parti si deve alternativamente spezzare il file, sotto è riportato un esempio di questa operazione con topic=3.

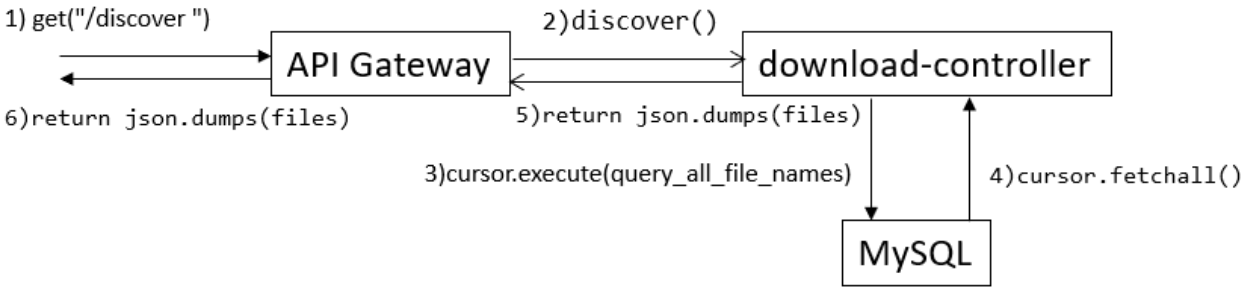


I tre file vengono successivamente inviati nei diversi topic "upload"+numeroTopic e letti da tutti i fileSystem associati a quel topic.

2) API di discover:

La seconda API realizzata serve a chiedere al sistema quali sono i nomi di tutti i file caricati, allo scopo di poter conoscere se il file che si cerca è memorizzato nel sistema.

Sempre utilizzando il framework Flask abbiamo implementato un'interfaccia sulla route "/discover" abilitata, questa volta, a rispondere solo a messaggi di tipo GET.

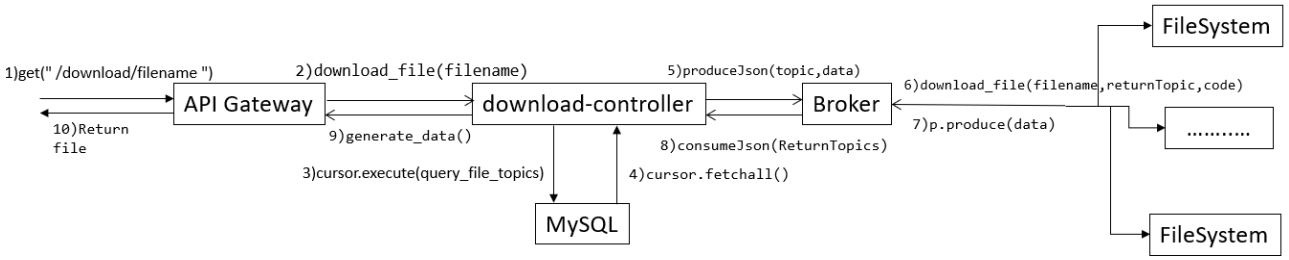


A questo scopo un agente, chiamato download-controller, esegue una query al database, cercando le informazioni inserite dall'upload-controller nell'API precedente e ritorna un file json contenente tutti i nomi dei file presenti nei fileSystem.

3) API di download:

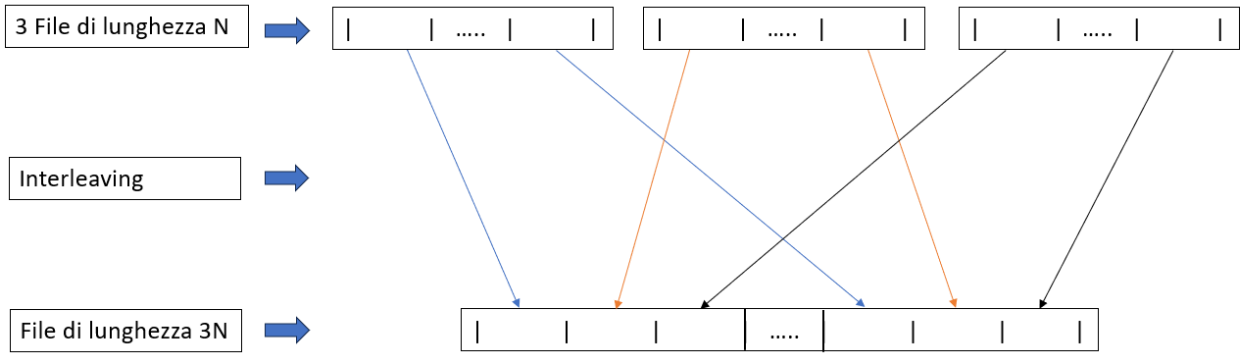
La terza API serve a scaricare uno dei file precedentemente inseriti nel sistema tramite la route Flask ”/download/<path:filename>”, questa volta contenente un secondo parametro filename allo scopo di identificare il file da inviare,sempre abilitata a rispondere a messaggi di tipo GET.

A questo scopo il dowload-controller richiede se esistono topic associati a quel file name e, in caso positivo, invia una richiesta in tutti i topic “request”+numeroTopic contenente un codice, un nome del file ed un topic su cui tornare il file.



Il messaggio viene resituito, tramite il broker Kafka, in diversi chunk di uguale lunghezza e per ogni N di loro (uguale al numero di topic del file) il download-controller deve effettuare l’operazione di interleaving e restituire un nuovo chunk all’API Gateway che sarà di lunghezza N*len(chunk).

Sotto è riportato un esempio con N=3



4) API per il recupero delle metriche del progetto (richiesta del progetto):

Per l'implementazione dell'API per il recupero delle metriche del progetto abbiamo usato, come nelle altre 3 API una route Flask implementata stavolta da un agente chiamato QoS-manager.

Da richieste si rendeva necessario acquisire le seguenti informazioni:

- valore attuale della metrica
- valore desiderato
- stato della metrica rispetto l'SLA - > violazione (true/false)
- numero di violazioni in un arco temporale predeterminato(ore)
- probabilità di violazione nel prossimo intervallo di tempo X (minuti).

Per il nostro sistema, come introdotto nell'abstract, abbiamo cercato di mantenere un buon throughput aumentando però la latenza iniziale, per questo abbiamo deciso di raccogliere queste due metriche (prodotte dal download-controller) tramite la configurazione di un server Prometheus interno al nostro sistema, con polling time di 2s.

Successivamente alla raccolta dei dati, un thread del QoS-manager si è occupato di richiedere il valore attuale delle misurazioni e di salvara il dato peggiore in una tabella del nostro database (poiché l'agente download-controller sarà replicato e quindi si avranno N metriche raccolte).

La route Flask definita nel QoS-manager è "/query" ancora una volta abilitata a rispondere a messaggi di tipo GET.

Dato un parametro "type", la route restituisce uno o più dei dati richiesti, ma anche alcuni utili allo sviluppo, ovviamente indicando in un parametro diverso, chiamato "query", il nome della metrica richiesta, oltre a parametri specifici per ogni diversa richiesta.

I diversi valori di type accettati sono:

1) "value"
2) "desired_value"- "discover"
3) "violation"
4) "all_data"
5) "violation_probability"

Spieghiamo adesso ognuno di loro quali parametri richiede ed il risultato restituito

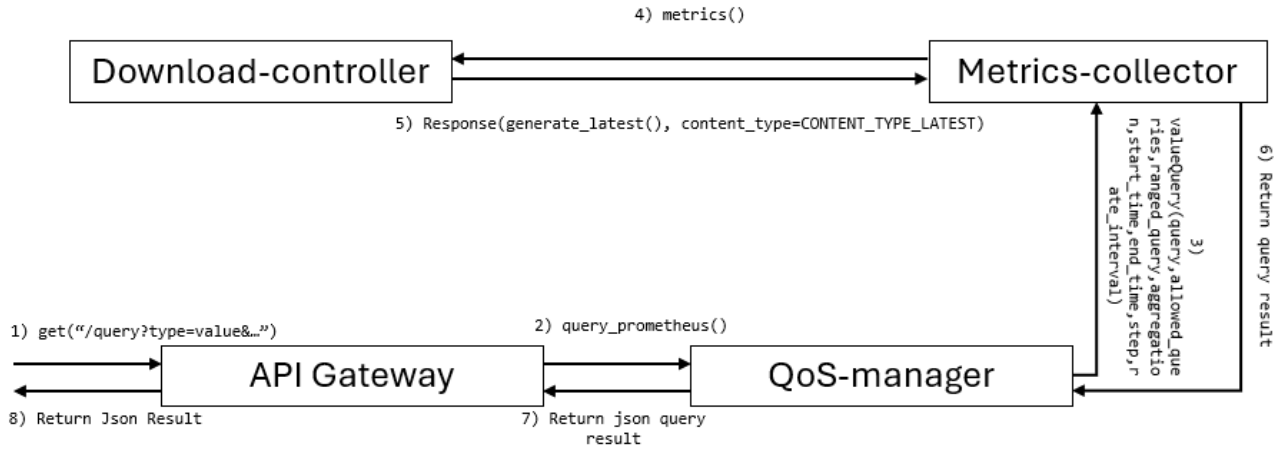
1)"value":

La chiamata di tipo "**value**" presenta nella richiesta **HTTP** di tipo **GET** i seguenti parametri, oltre al parametro "type":

- **query**, che rappresenta la PromQL Query all'Host Prometheus;
- **range**, la quale può assumere i valori "true" o "false", quest'ultimo valore di default. Se impostato a "true", tale parametro esegue una query al sistema Prometheus che presenta come parametri aggiuntivi **start**, **end** e **step** per definire un criterio di selezione temporale delle metriche raccolte da Prometheus stesso;
- **start**, indica il numero di ore antecedenti al momento di esecuzione della query, in modo da filtrare la selezione delle metriche richieste a partire dall'osservazione del loro timestamp di creazione temporale;
- **end**, ha lo stesso principio di funzionamento di "start", ma in questo caso stabilisce il momento temporale finale per filtrare la selezione delle metriche di Prometheus ;

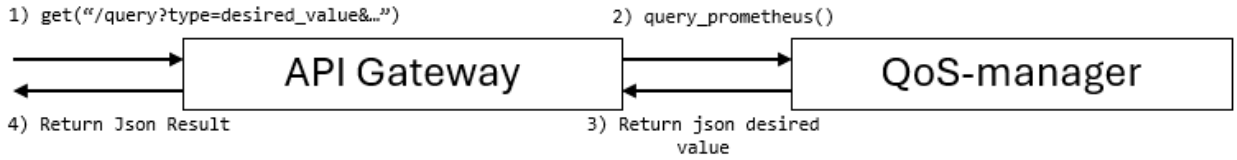
- **step**, indica il gradino di risoluzione temporale della query per Prometheus, in termini di secondi;
- **aggregation**, indica l'aggregazione da effettuare sulla query PromQL, da un insieme di aggregazioni permesse (es. "avg", "min", "count", "rate"). Questa implementazione non supporta l'aggregazione di query che presentano già aggregazioni;
- **rate_interval**, se il tipo di aggregazione della query è "rate", allora è necessario il parametro corrente per specificare l'intervallo di tempo su cui calcolare il tasso medio di aumento al secondo della serie temporale richiesta a Prometheus (se non specificato, il parametro ha come valore di default "5m", ovvero la finestra temporale che racchiude gli ultimi cinque minuti).

Inoltre, le query permesse sono quelle che presentano come metriche "download_file_latency_seconds" e "download_file_throughput_bytes" (le metriche esportate dal Download-controller). In conclusione, il risultato restituito dalla chiamata è un json relativo alla risposta della query effettuata a Prometheus o di un eventuale errore avvenuto durante la richiesta.



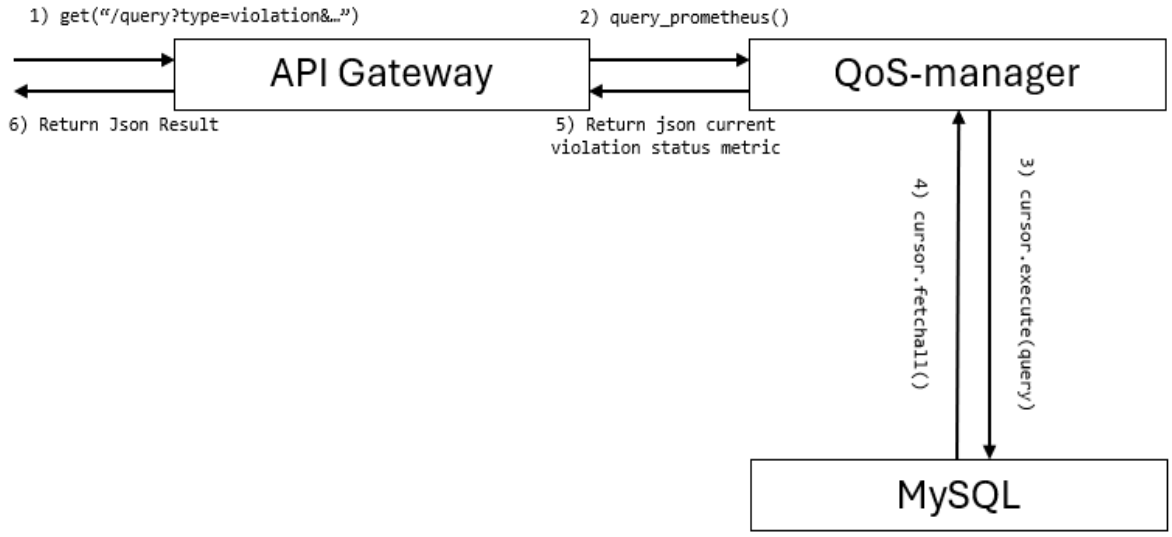
2)"desired_value"- "discover"

Il valore desiderato e la discover sono le chiamate più semplici alla nostra API poiché essendo informazioni hard-coded infatti, nella discover possiamo restituire direttamente il vettore delle query permesse, invece, nei valori desiderati, dopo aver distinto di quale delle due metriche ci viene fatta richiesta, è possibile ritornare direttamente il valore numerico desiderato.



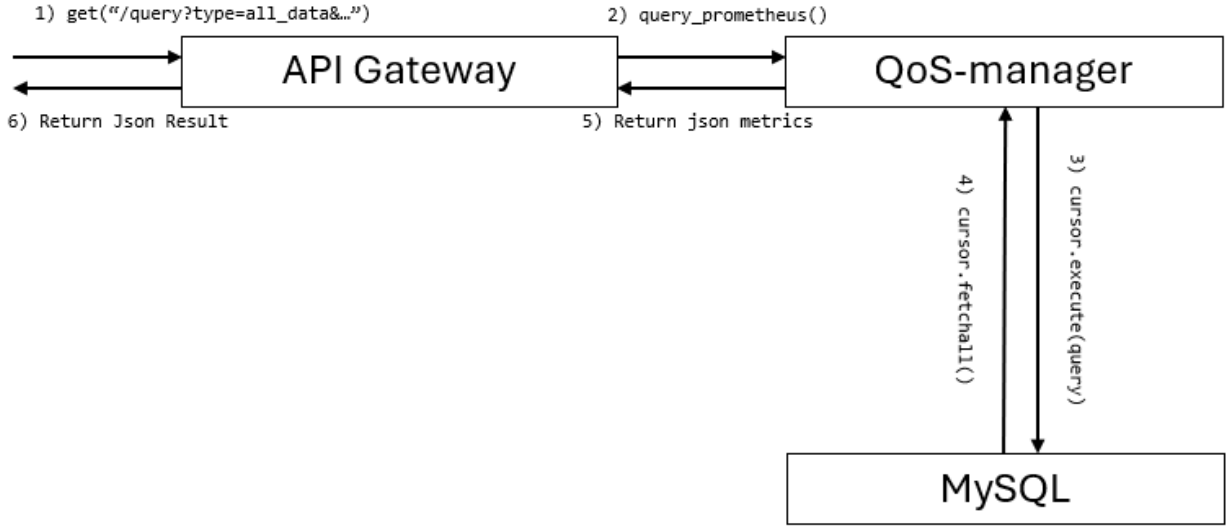
3)"violation":

Per questo tipo di richiesta il QoS-Manager va a fare una query sul Database, chiedendo quali dei valori salvati precedentemente è oltre il valore desiderato per la metrica richiesta e, tramite un parametro "start", seleziona quante ore passate deve considerare. Restituisce il numero di violazioni nel periodo di ore richiesto.



4)"all_data":

Questa API restituisce tutti i cambiamenti avvenuti su una singola metrica. È stata realizzata soprattutto per scopo amministrativo, ma anche per l'allenamento iniziale dei modelli predittivi ARIMA. Infatti, per l'allenamento abbiamo avuto bisogno di dati reali provenienti dal nostro sistema. (Dati estratti lanciando due copie dello script client.py nella cartella QoS-manager).

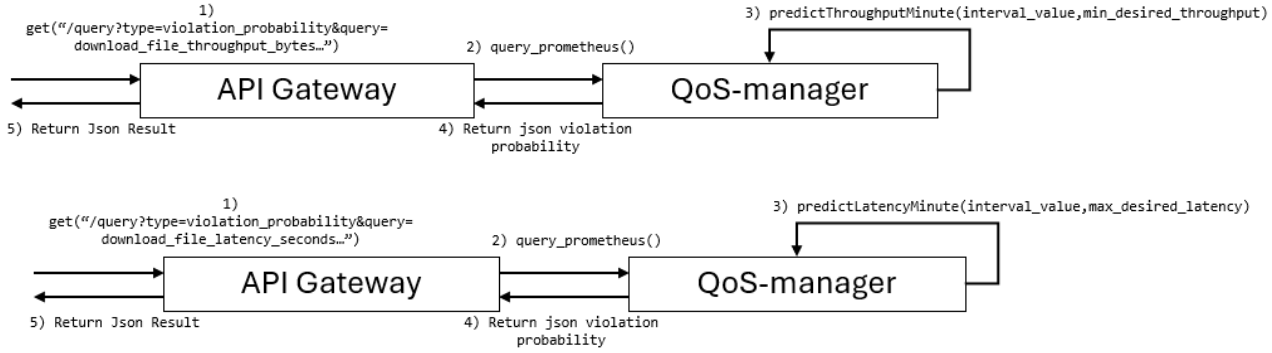


5)"violation_probability":

Per la restituzione della probabilità di violazione nei prossimi X minuti ci siamo serviti di due modelli ARIMA per ogni metrica, andando a predire valor medio e deviazione standard nei prossimi istanti di tempo e restituendo la probabilità che, almeno una volta, il sistema vada oltre le soglie di valore desiderato.

Restituiamo il valore di probabilità di violazione nei prossimi X minuti, la massima probabilità istantanea di violazione nei singoli istanti nel periodo e la media della predizione della metrica nel periodo.

Approfondiremo questa API nella sezione "Metodo predittivo".



Scelta progettuale del Database e del broker

Database:

Durante le prime scelte progettuali abbiamo deciso di optare per la realizzazione del sistema con un unico database MySQL centralizzato poiché per la gestione del sistema abbiamo una serie di dati altamente strutturati adatti a questo tipo di database.
La scrittura per ogni tabella è stata limitata ad un solo agente in modo tale da evitare problemi di consistenza.
Lista tabelle e attuale gestore:

Tabella partitions: gestore "manager"

```
CREATE TABLE IF NOT EXISTS partitions (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    partition_name VARCHAR(255) UNIQUE NOT NULL,
    topic INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

La tabella raccoglie le informazioni dei filesystem nel sistema ed il loro topic assegnato.

Tabella files: gestore "Upload-controller"

```
CREATE TABLE IF NOT EXISTS files (
    file_id INT UNSIGNED AUTO_INCREMENT,
    file_name VARCHAR(255) NOT NULL,
    partition_id INT UNSIGNED,
    ready BOOL ,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (file_id,partition_id),
    FOREIGN KEY (partition_id) REFERENCES partitions (id)
);
```

La tabella files raccoglie le informazioni di ogni partizione in cui è stato diviso il file ed il rispettivo topic, inoltre poiché l'attore "upload-controller" potrebbe essere replicato si è aggiunto un campo ready per la gestione della consistenza.

Tabella controller: gestore “Controller-manager”

```
CREATE TABLE IF NOT EXISTS controller (
    id_controller INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    controller_name VARCHAR (255) UNIQUE NOT NULL,
    cType VARCHAR(20)  NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

La tabella controller raccoglie le informazioni dei controller nel sistema (solo quelli di download che di upload visto che non è detto siano unici).

Tabella controllertopic: gestore “Controller-manager”

```
CREATE TABLE IF NOT EXISTS controllertopic (
    id_controller INT UNSIGNED,
    topic INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id_controller,topic),
    FOREIGN KEY (id_controller) REFERENCES controller (id_controller)
);
```

La tabella controllertopic associa ad ogni upload-controller ogni topic di cui esso è responsabile (in questa configurazione ogni upload-controller gestisce tutti i topic senza una diversa gestione di politiche di marketing.

Tabella metrics: gestore “QoS-manager”

```
CREATE TABLE IF NOT EXISTS metrics (
    id_metric INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    metric_name VARCHAR (255) NOT NULL,
    metric_value FLOAT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

La tabella metrics si occupa di mantenere lo storico delle metriche raccolte dal server prometheus per l’uso nelle previsioni o per semplice visione tramite query

Broker:

Come preventivato nell’abstract, per la quasi totalità delle comunicazioni interne è stato usato il broker Kafka poiché adatto al tipo di operazioni implementate e poiché esso fornisce importanti funzionalità come la possibilità di inviare a vari attori lo stesso messaggio con un paradigma produttore-consumatore, oppure fornire in Round Robin una serie di richieste ad un gruppo di macchine fungendo da load balancer; allo stesso tempo fornisce un meccanismo di garanzia della ricezione dei messaggi tramite un semplice meccanismo di commit e retain dei messaggi scambiati.

Nel nostro sistema il retain dei log è stato ridotto a soli 2 minuti poiché le risorse locali non hanno permesso di mantenere il limite di default di 7 giorni.

Foto tabelle database

Topic per agente:

fileSystem:

subscriptions topics:

1) ”FirstCallAck”
2) “Request” + Topic
3) “Delete” + Topic
4) “Upload” + Topic
5) “UpdateDownload”

Production topics:

1) ”firstCall”
2) “UpdateRequest”
3) Topic forniti dinamicamente dai download controller

Download-controller:

subscriptions topics:

1) ”CFirstCallAck”
2) Topic forniti dinamicamente dal controller-manager

Production topics:

1) “Request” + Topics
2) “CFirstCall”

Upload-controller:

subscriptions topics:

1) ”CFirstCallAck”
2) “UpdateTopics”
3) Str(socket.gethostname())
4) "Delete"+str(topic)

Production topics:

1) “Upload” + Topics
2) “CFirstCall”

Controller-manager :

subscriptions topics:

1) ” CFirstCall”

Production topics:

2) “Upload” + Topics
3) “CFirstCallAck”
4) “UpdateTopics”

Manager :

subscriptions topics:

1) ” FirstCall”

Production topics:

2) “Upload” + Topics
3) “FirstCallAck”

4) “UpdateTopics”

Update-Controller :

subscriptions topics:

1) ”UpdateRequest”
2) “UpdateIntermediate”

Production topics:

3) “Request” + Topics

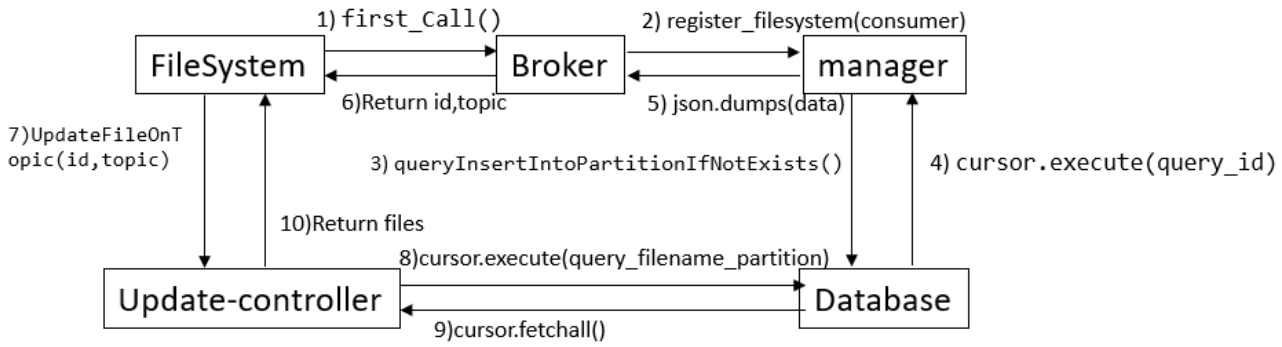
Spiegheremo in dettaglio il significato dei topic nella sezione “Metodi di inizializzazione dinamico”.

Aggiungiamo che sia il database che il broker nel nostro sistema rappresentano un single point of failure e quindi dovrebbero essere replicati e si dovrebbe mantenere la consistenza tra le diverse repliche.
In questa implementazione però entrambi vengono lasciati come singole repliche per ragioni di tempistiche.

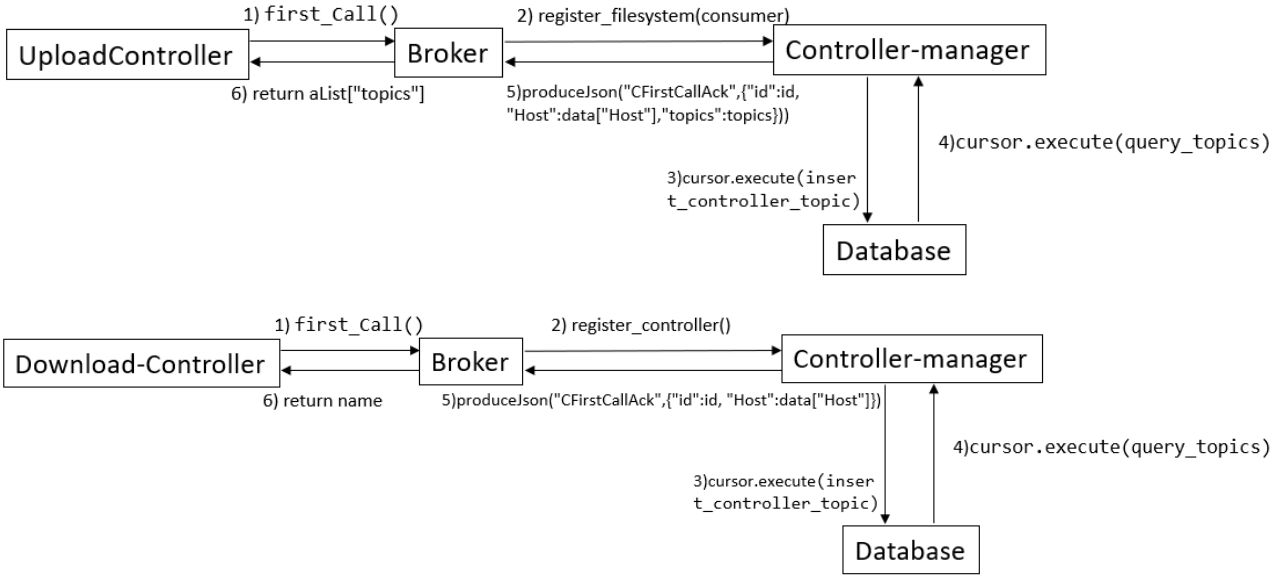
Metodo di inizializzazione dinamico

Per rendere il nostro sistema elastico rispetto ai traffici e politiche di marketing, suggestivamente implementabili, abbiamo deciso di creare dei manager che si occupino di inserire, in una determinata famiglia, i nostri attori come i fileSystem, gli uploadController ed i download-controller. A questo scopo, per i fileSystem è stato implementato un attore chiamato “manager”, invece per i controller è stato creato un “controller-manager”.

Ruolo del manager:
All’inizio dell’attività, ogni fileSystem tramite il broker chiede al manager in quale topic posizionarsi per le successive operazioni di upload – delete e request .
I FileSystem si iscrivono nei topic usando diversi gruppi per le prime due operazioni e lo stesso gruppo per le operazioni di request , in modo tale da ricevere tutti gli aggiornamenti dei file interni, ma potendo smaltire parallelamente diverse richieste di download.
Subito dopo aver stabilito il topic di appartenenza, sempre tramite il broker (che in figura non è rappresentato) il filesystem chiede ad un controller (Update-controller) di allineare i suoi file locali con quelli delle altre copie già attive prima della sua creazione.
A questo punto il filesystem comincia ad operare e cercare di smaltire il nuovo traffico



Ruolo del Controller-manager:
A differenza del manager, il controller-manager offre per lo più un servizio di possibile differenziazione di qualità di servizio, facilmente implementabile, poiché definisce in quanti topic dividere il file iniziale dando semplicemente una diversa lista all'upload controller.
Allo stesso modo, controlla la creazione dei topic di ritorno del download controller e, quindi, anche qui potrebbe differenziarne il comportamento per politiche interne o di marketing.
Nella nostra implementazione attuale non introduciamo differenza, ma vengono tutti aggiornati al numero totale di topic utilizzati dai fileSystem, concentrandoci sulle prestazioni.
Di seguito gli schemi delle operazioni di firstCall, del tutto analoghe a quelle dei fileSystem

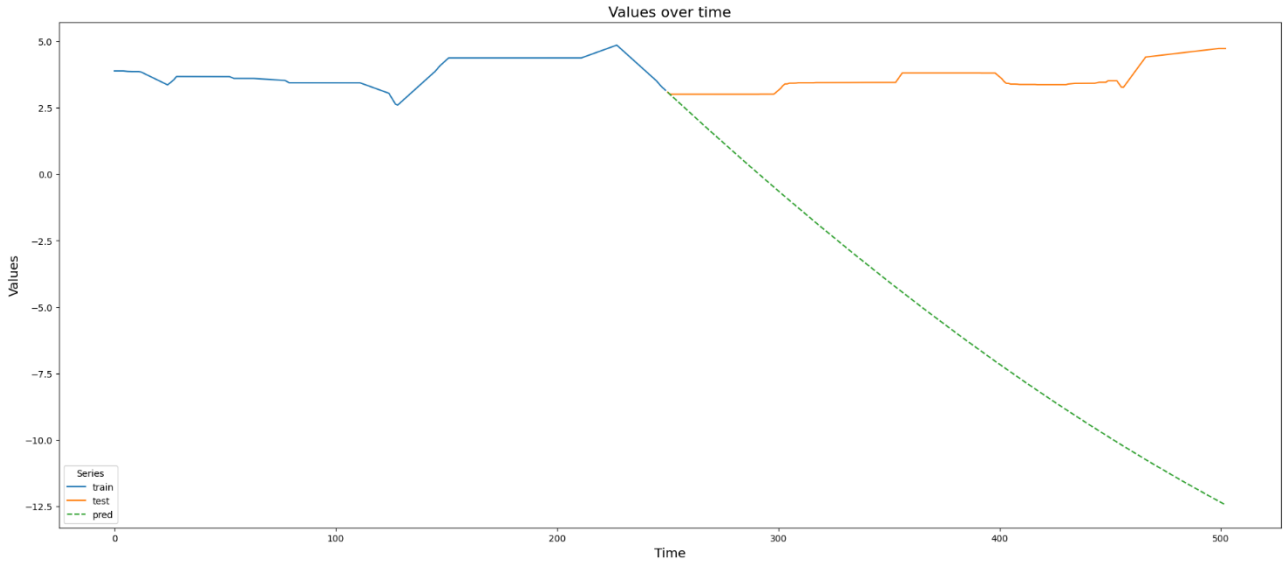


Metodo predittivo

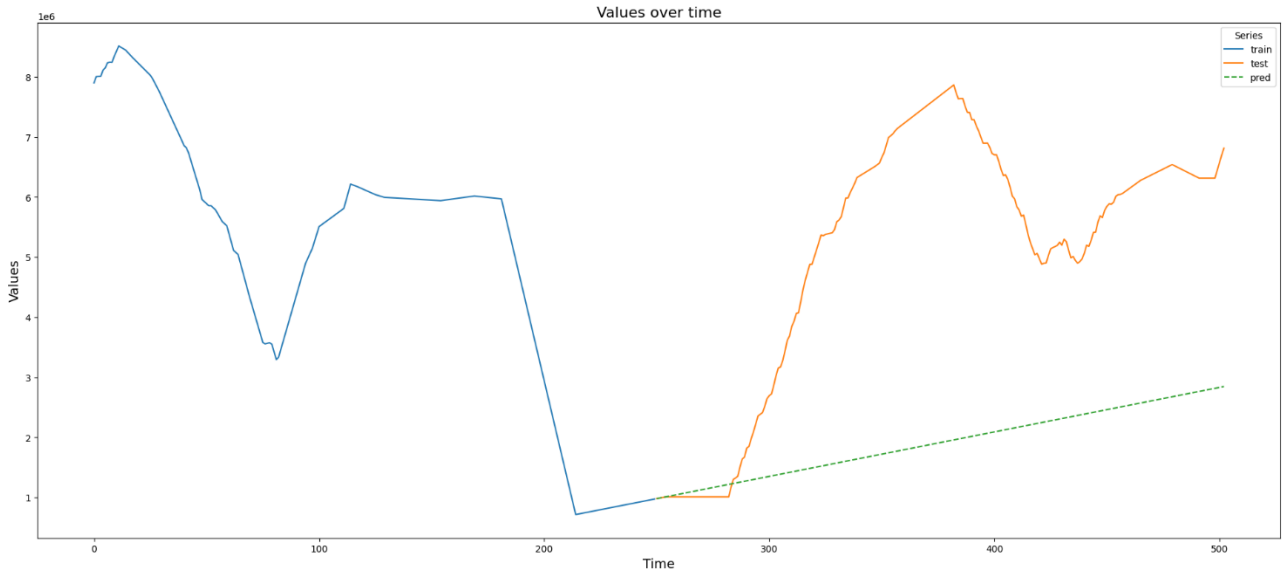
Il modello predittivo usato in questo progetto è basato sui modelli ARIMA e l'utilizzo della scomposizione in trend e seasonality.

Estrapolazione dei dati di allenamento:
Dai dati raccolti nei nostro sistema , estrapolati tramite l'API "/query" con type "all_data" abbiamo creato una time series di campioni usando le misurazioni di circa un'ora e come step tra i vari campioni abbiamo dato una differenza di 5 secondi.
Allo scopo di estrapolare per ogni metrica la probabilità di violazione dell'intervallo di tempo x , poiché nativamente i modelli ARIMA non supportano una predizione probabilistica ma piuttosto una previsione puntuale , abbiamo ipotizzato che le metriche misurate ,cioè throughput e latenza abbiano un comportamento gaussiano e quindi abbiamo calcolato , ogni 100 campioni la media del segnale e la corrispondete deviazione standard .Facendo scorrere la finestra dei 100 campioni fino alla fine del segnale otteniamo due nuovi segnali (quello della media e quello della deviazione standard) che hanno una lunghezza uguale a Len(segnale iniziale) – 100 campioni.
Scelta del modello:
Per la scelta del modello abbiamo inizialmente considerato l'opzione di estrarre il trend dai nostri dati tramite diversi parametri di period della funzione di scomposizione e provare ad ottenere un modello ARIMA tramite la funzione auto_arima() ed allenando i modelli con i nostri dati ottenendo dei risultati a dir poco pessimi.
Mostriamo i 4 grafici ottenuti da una delle molte prove.

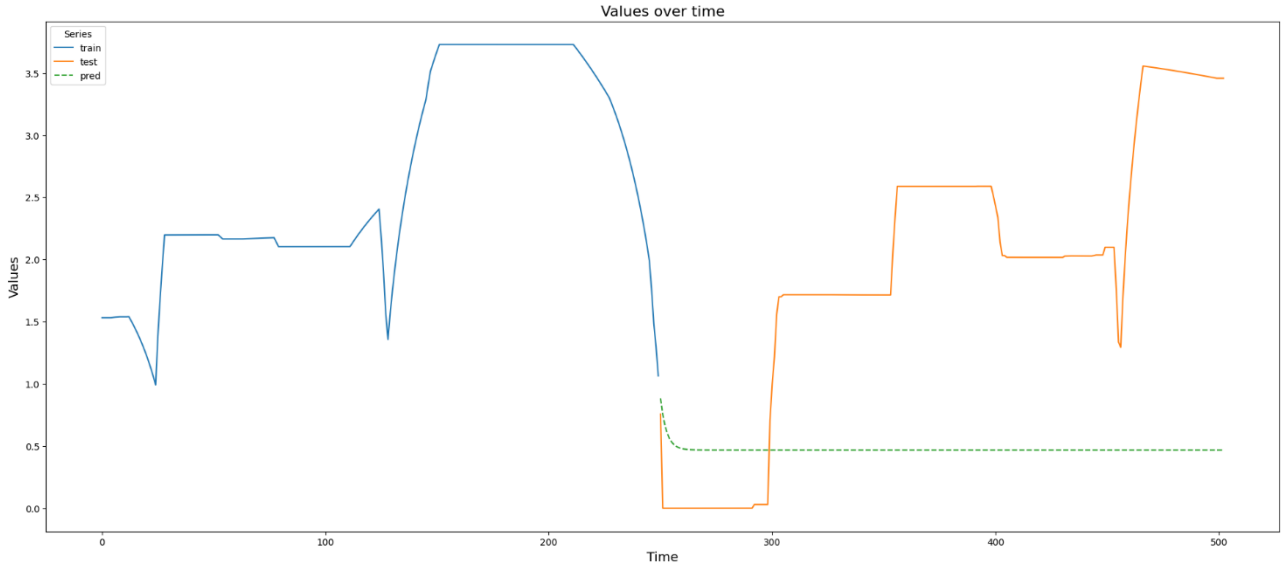
---Trend



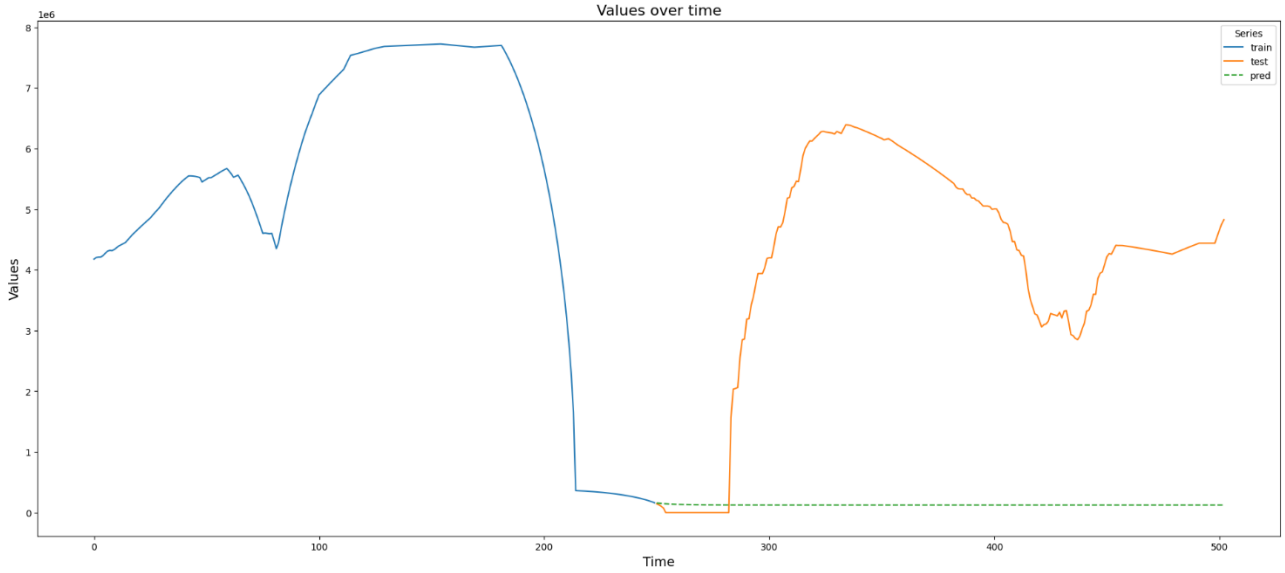
Latenza Media (Trend)



Throughput Medio (Trend)



Deviazione Std Latenza (Trend)

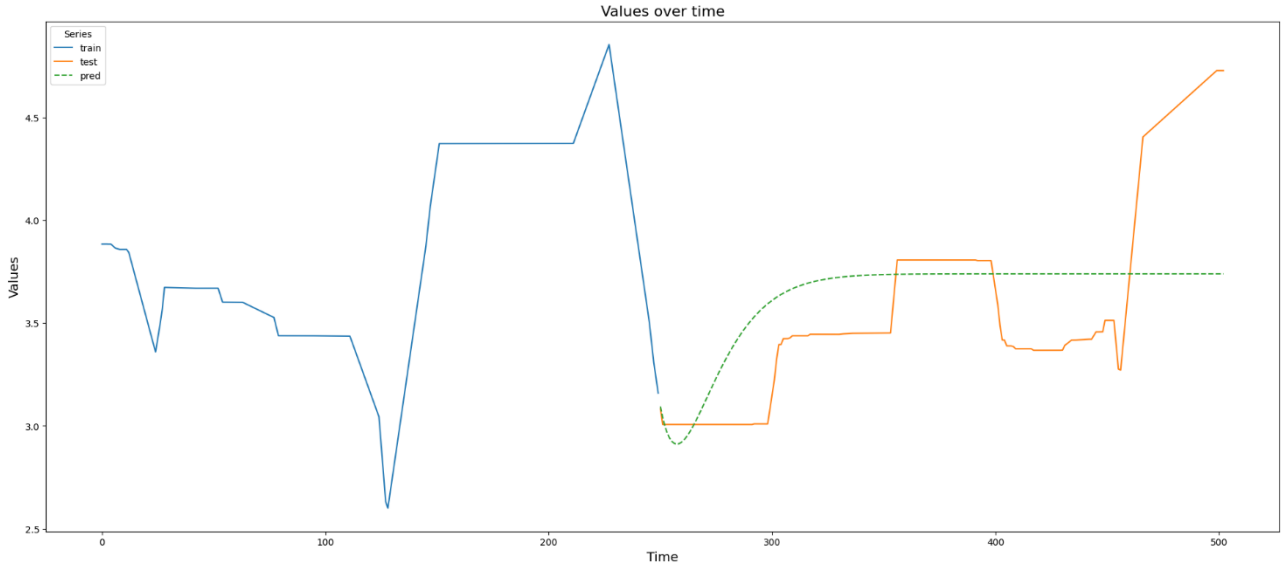


Deviazione Std Throughput (Trend)

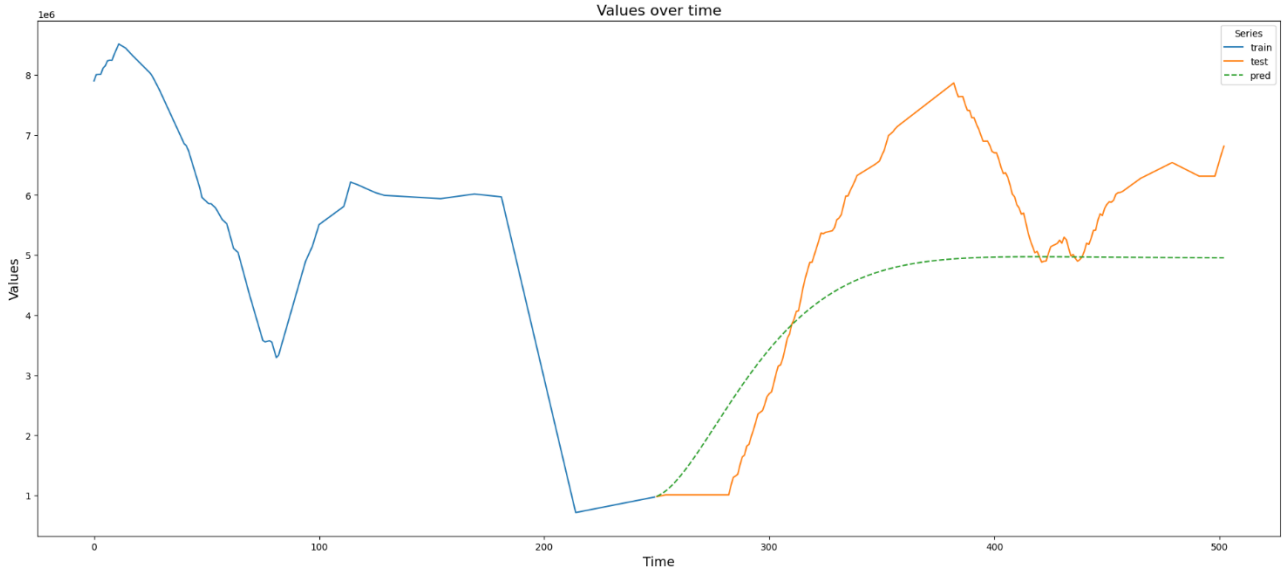
---Seasonality

A questo punto , riflettendo sulla provenienza dei dati , ci siamo resi conto che il throughput minimo e la latenza massima non sono delle funzioni che naturalmente tendono a crescere o decrescere ma sono derivati dal flusso del traffico e fluttuazioni dello stesso ,quindi abbiamo deciso di utilizzare i valori estrapolati dalla seasonality estratta da seasonal_decompose.

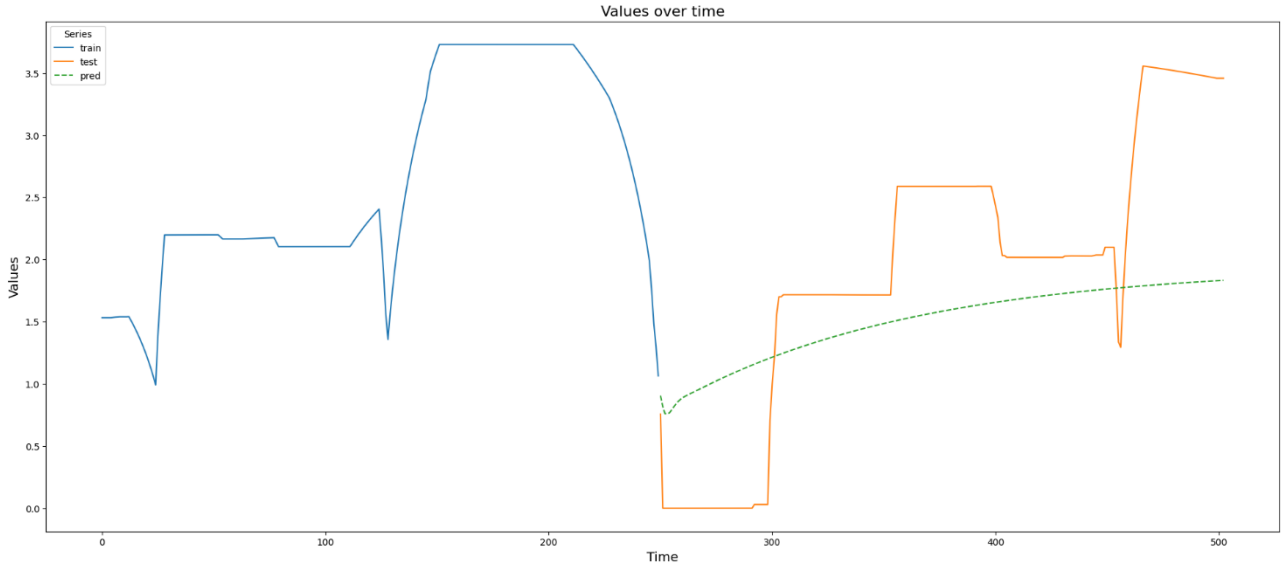
Utilizzando questi dati abbiamo trovato dei modelli che fittano abbastanza bene i nostri dati , come mostrato sotto.



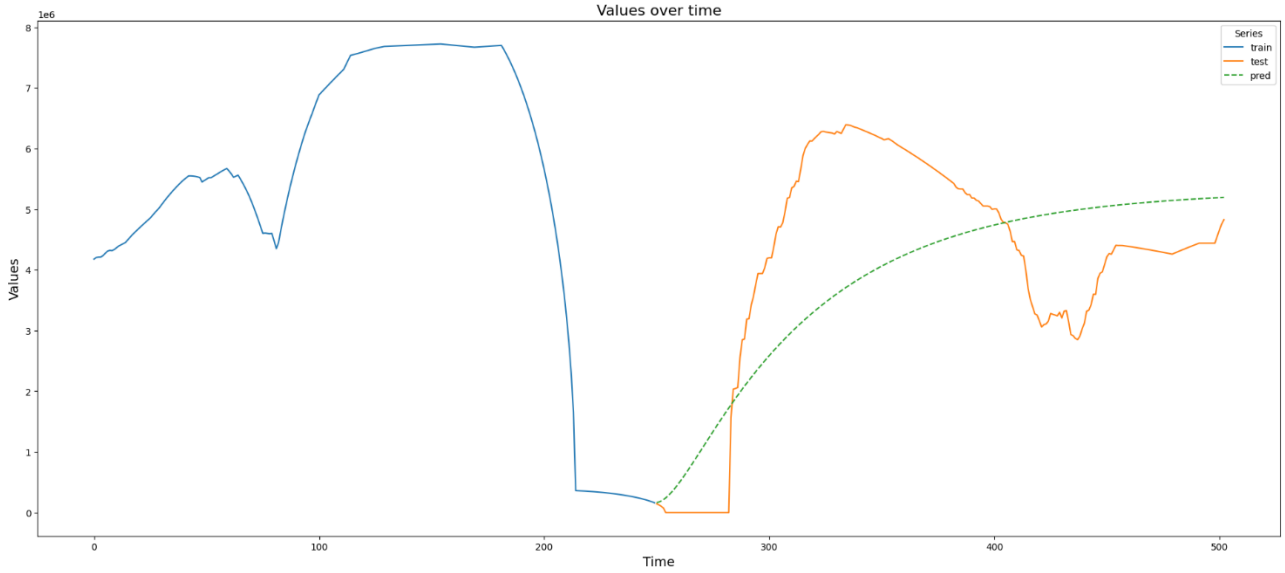
Latenza Media (Seasonality)



Throughput Medio (Seasonality)



Deviazione Std Latenza (Seasonality)



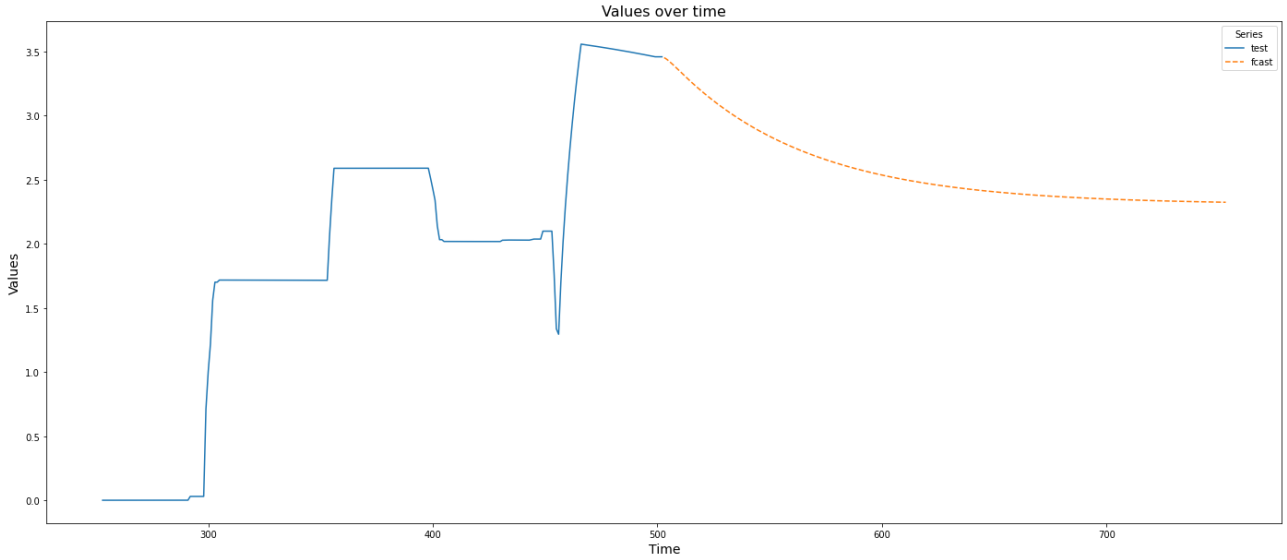
Deviazione Std Throughput (Seasonality)

Predizione post-training:

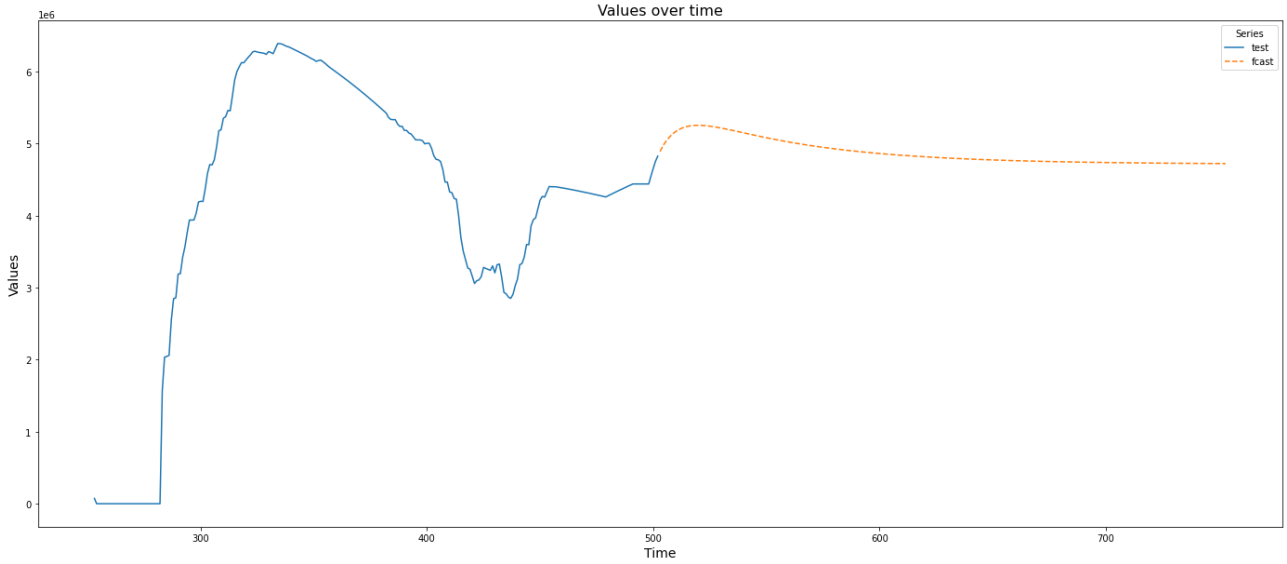
Per attuare la predizione di violazione della metrica nei prossimi x minuti viene effettuato il forecast dei modelli corrispondenti alla metrica richiesta e viene calcolata la possibilità di non violare la metrica per ogni punto , a questo punto la probabilità di non violare mai la metrica è data dall'operazione prodotto tra le singole probabilità .Troviamo la probabilità di violare almeno una volta tramite la formula:

$$\mathrm{p}(x) = 1 - \prod_{k=1}^{x*12} pk \quad , k \in 0{,}1, \dots, x * 12$$

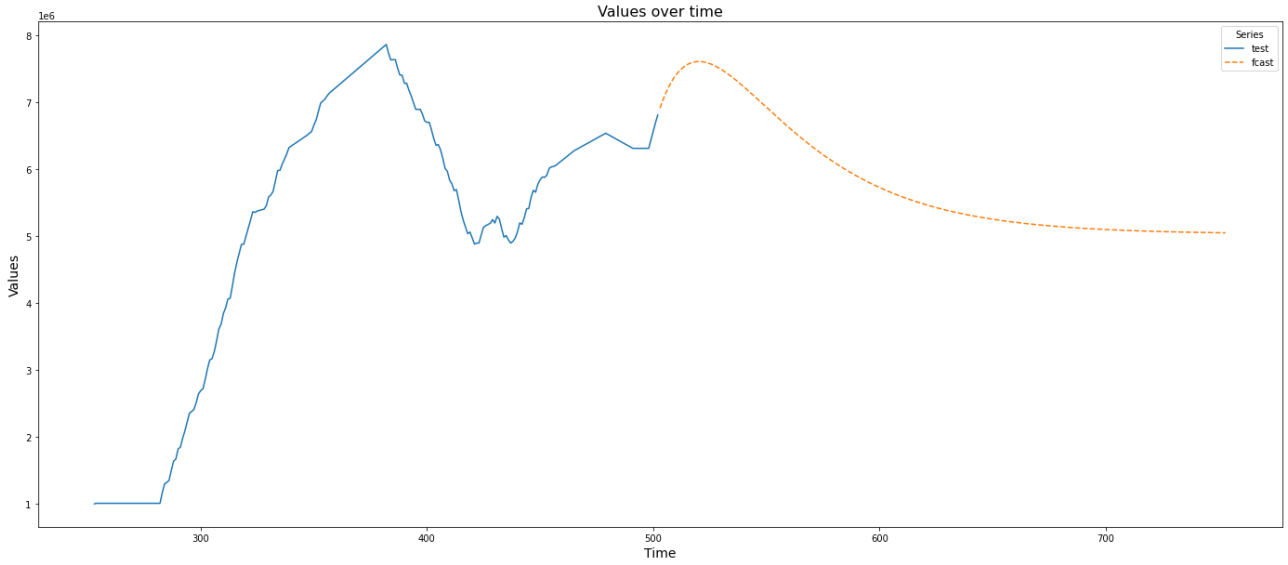
il 12 visto nelle formule deriva dal fatto che l'allenamento è stato fatto con campioni in 5 secondi, invece la richiesta dell'API esterna è fatta in minuti.



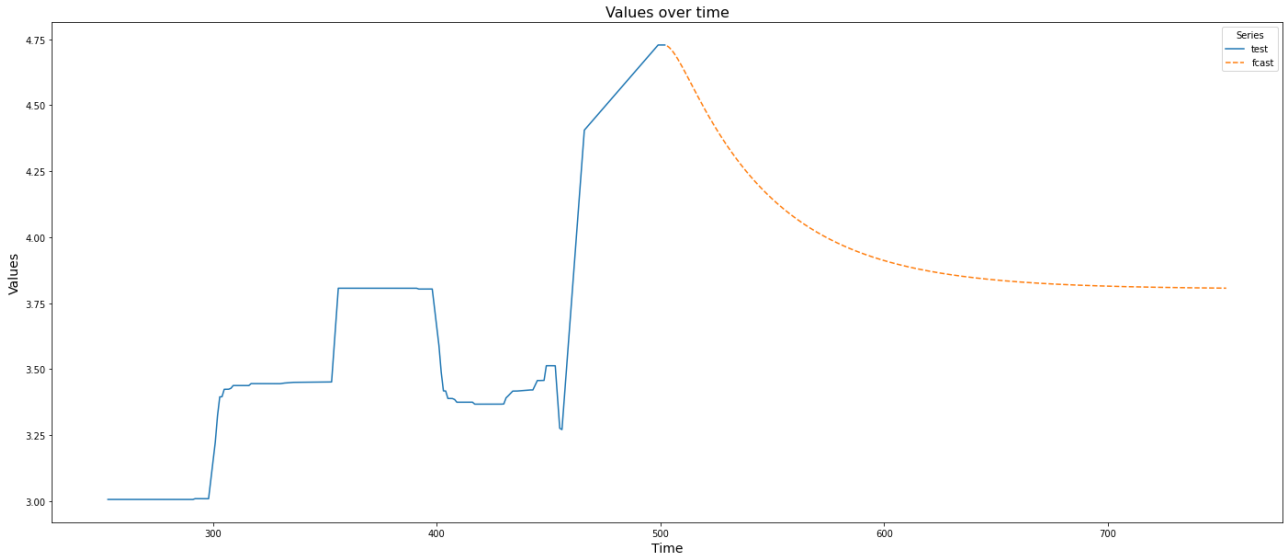
Forecast Deviazione Std Latenza



Forecast Deviazione Std Throughput



Forecast Throughput Medio



Forecast Latenza Media

Online learning

Dalle nostre analisi abbiamo notato che tutti i modelli ARIMA da noi allenati ad un certo punto tendono ad un valore costante non fornendo un forecast corretto dopo alcune decine di minuti a questo punto abbiamo deciso di implementare una tecnica di online learning in cui dopo aver trovato l'ordine del modello, esso viene mantenuto nel tempo, ma ogni 10 minuti viene rifatto il model.fit() dei dati sull'ora precedente di misurazioni, fornendo un forecast più accurato