UNIVERSITÀ
degli STUDI
di CATANIA

Distributed systems and big data
A.A 2022-2023

Elaborato prova in itinere

Federica Rizza - 1000042315
Giorgia Rumore Pagano - 1000042300

Indice

1 Abstract

Si vuole implementare un sistema per l'elaborazione e il monitoraggio di dati forniti da un server Prometheus che espone delle metriche ottenute in seguito all'osservazione di una applicazione. Per la realizzazione dell'applicazione si utilizza il modello architetturale a microservizi, utilizzando la piattaforma Docker per il loro deploy in container. Nel dettaglio il sistema prevede, da parte del servizio **ETL Data Pipeline**, l'elaborazione di una serie di informazioni ottenute dal server Prometheus. Una volta aver *prodotto* i dati esso li invia ad un broker Kafka utilizzando il topic *prometheusdata*. Successivamente, i dati vengono *consumati* dal servizio **DataStorage** e inviati ad un DB MySQL. Attraverso l'utilizzo di REST API il servizio **Data Retrieval** esegue delle query al DB per ritornare i dati elaborati precedentemente. Infine il servizio **SLA Manager**, basato su gRPC, permette di definire un set ristretto di metriche sulle quali si eseguono ulteriori elaborazioni e per le quali si possono verificare le eventuali violazioni passate e future, sulla base di un range di valori definiti dall'utente.

1.1 Scelte implementative

Il server Prometheus utilizzato, come mostrato in Figura 2, si trova all'indirizzo 15.160.61.227:29090. Prima di iniziare con l'elaborazione dei dati, si ci è preoccupati di scegliere le metriche da monitorare. In particolare si è deciso di filtrarle per job = 'summary' e per questo job si è deciso di lavorare sul seguente *metric_set*: "cpuLoad", "cpuTemp", "diskUsage", "availableMem", "realUsedMem", "networkThroughput", "inodeUsage". Vengono quindi presi in considerazione anche tutti i nodi caratterizzati dai nomi citati prima. Di conseguenza ogni "metric_name" (definito tramite il nome della metrica e l'insieme di label fornite da Prometheus) corrisponde ad una serie temporale diversa. Come accennato prima si è deciso di implementare un'applicazione docker-based e multi-container e per gestire in maniera immediata il deploy dei servizi nei vari container, è stato creato un file docker-compose. In questo modo i diversi microservizi vengono deployati in un singolo docker-host, e con dei semplici comandi è possibile avviare tutto o eliminare tutto. A supporto dei microservizi implementati si è resa necessaria la creazione dei seguenti container:

- **db**: costruito a partire dall'immagine *mysql* presente su DockerHub. A quest'ultimo viene associato un volume in modo tale che i dati contenuti in esso non siano effimeri. La struttura è mostrata in Figura 1;

- **broker_kafka**: costruito usando l'immagine *confluentinc/cp-kafka* presente su DockerHub, istanzia un broker Kafka, a supporto del quale vengono creati i container **init_kafka**, che si occupa dell'inizializzazione del topic *prometheusdata*, e **zookeeper**, il quale gestisce la configurazione del broker.

```
CREATE TABLE IF NOT EXISTS datas (
    ID_metrica INT NOT NULL AUTO_INCREMENT,
    metric_name VARCHAR(16000) NOT NULL,
    slug VARCHAR(64) AS (SHA2 (metric_name, 256)) STORED NOT NULL,
    max_1h  float,
    max_3h  float,
    max_12h  float,
    min_1h  float,
    min_3h  float,
    min_12h  float,
    avg_1h  float,
    avg_3h  float,
    avg_12h  float,
    devstd_1h   float,
    devstd_3h   float,
    devstd_12h  float,
    max_predicted float DEFAULT NULL,
    min_predicted float DEFAULT NULL,
    avg_predicted   float DEFAULT NULL,
    stazionarieta   BOOLEAN,
    stagionalita    INT,
    PRIMARY KEY (ID_metrica),
    UNIQUE (slug)
);

CREATE TABLE IF NOT EXISTS acf (
    ID_metrica INT NOT NULL,
    acf_lag INT,
    acf_value float,
    PRIMARY KEY(ID_metrica, acf_lag)

);

CREATE TABLE IF NOT EXISTS sla (
        metric_name VARCHAR(255) NOT NULL,
        min INT,
        max INT,
        PRIMARY KEY (metric_name)
);
```

Figura 1: struttura del database *prometheus_data*

2 Componenti dell'applicazione

L'applicazione sviluppata è caratterizzata da un database MySQL, chiamato *prometheus-data*, all'interno del quale vengono salvati tutti i dati calcolati dal sistema. Il database è formato da tre tabelle: datas, acf ed sla. Il sistema è costituito da diversi microservizi, sviluppati in python come illustrato in Figura 2. Tra questi servizi ne distinguiamo quattro principali che si occupano di eseguire le operazioni necessarie al raggiungimento dell'obiettivo del sistema. Di seguito vengono spiegati nel dettaglio.

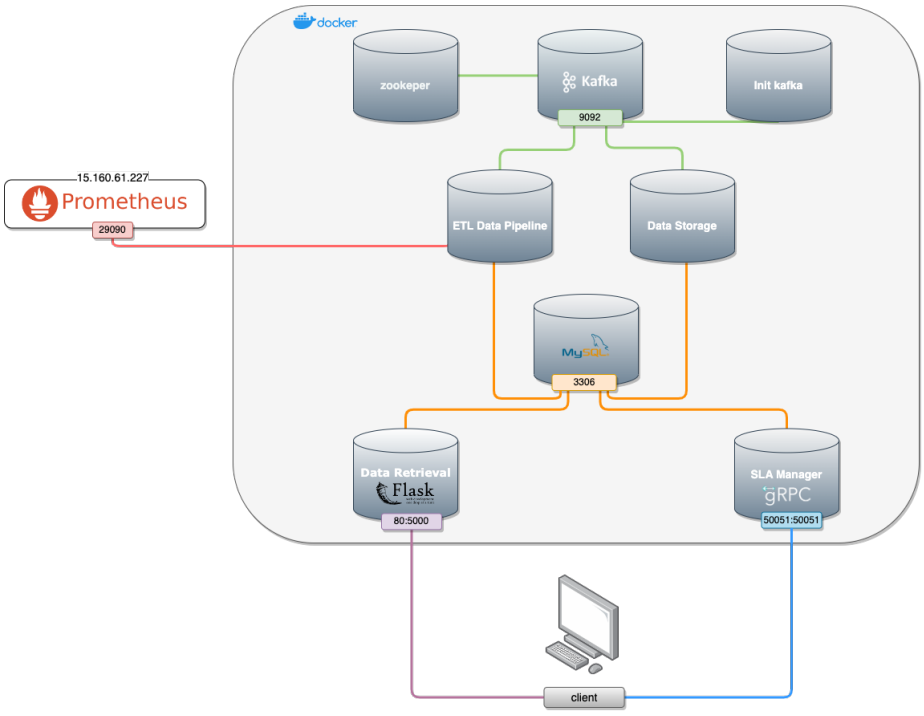

Figura 2: architettura dell'applicazione sviluppata

2.1 ETL Data Pipeline

ETL Data Pipeline rappresenta il primo micorservizio sviluppato. Esso viene definito da un processo principale e da un thread separato. Entrambi si occupano di estrarre i dati da Prometheus mediante una query che permette di recuperare le metriche definite nel *metric_set*.

Il thread viene lanciato per recuperare i dati da Prometheus relativi agli ultimi 7 giorni, il quale li converte in dataframe e calcola i seguenti metadati:

- Stazionarietà: per calcolare la stazionarietà si ci è affidati al cosidetto Augmented Dickey-Fuller test che permette di valutare se in una determinata serie temporale sono presenti comportamenti stocastici e se quindi è stazionaria. In python tale test viene eseguito mediante la funzione *adfuller* appartenente al package *statsmodels*. In tal caso si è deciso di considerare stazionarie tutte le serie per le quali risulta *p-value* $<= 0.01$. Tale valore viene ritornato dalla funzione *adfuller*;

- Autocorrelazione: per quanto riguarda l'autocorrelazione si è deciso di memorizzarne il valore per tutti quei lag il cui coefficiente di autocorrelazione risulta maggiore di 0.25. Anche per l'autocorrelazione è stata utilizzata una funzione del package *statsmodels*, cioè *acf*

- Stagionalità: al fine di valutare la stagionalità per prima cosa viene controllato che la serie non sia stazionaria, e in tal caso si applica la trasformata discreta di Fourier. In questo modo si riesce a risalire alla frequenza massima, che invertita rappresenta il periodo di ripetizione del ciclo stagionale.

Si è scelto di eseguire il calcolo di questi metadati in un thread separato, che viene lanciato una volta al giorno, per alleggerire il carico computazionale del processo principale, in quanto si suppone che questa tipologia di dati subisca variazioni solo dopo lunghi periodi.

Il processo principale invece richiede ogni 5 minuti le metriche delle ultime 12h, e anch'esso le converte in dataframe per effettuare le seguenti operazioni:

- Calcolo dei valori di massimo, minimo, media e deviazione standard delle ultime 1, 3 e 12 ore di dati. Per fare questo è stata utilizzata la funzione *last* della libreria *pandas* in modo da ricavare i tre slot temporali;

- Predizione dei valori di massimo, minimo e media dei successivi 10 minuti per le metriche appartenenti a SLA set definito nel microservizio SLA Manager, che vedremo nella sottosezione 2.4, e salvato nella tabella *sla* del database. A tal proposito, per recuperare le metriche di SLA set si è deciso di collegare il microservizio al database. Per eseguire la predizione dei valori è stato realizzato un modello della serie temporale partendo dai dati a disposizione, utilizzando la funzione *ExponentialSmoothing* della libreria *statsmodels* con settaggi differenti in base alle caratteristiche della

serie. In particolare se la serie risulta stazionaria, e quindi non presenta trend o stagionalità, viene utilizzato il Simple Exponential Smoothing. Si costruiscono tre modelli differenti variando il parametro α e si valuta il migliore per effettuare la predizione in base al valore minore della radice dell'errore quadratico medio. Se invece la serie presenta stagionalità viene utilizzato l'Exponential Smoothing, considerando quattro modelli ottenuti combinando le tipologie additivo e moltiplicativo del trend e della stagionalità. Anche in questo caso si seleziona il modello migliore calcolando la radice dell'errore quadratico medio. Infine si è deciso di utilizzare il modello per le serie stazionarie anche per quelle serie che, tramite il calcolo della trasformata discreta di Fourier, presentano un periodo maggiore di un giorno, poiché osservando l'andamento delle serie prese in considerazione non si è mai riscontrato un periodo più grande di 12 ore; pertanto si suppone che queste serie siano cicliche, ovvero presentano un andamento che si ripete con un periodo non fisso.

Una volta terminate le varie operazioni, i dati calcolati sia dal processo principale che dal thread vengono inseriti in un dictionary e inviati al microservizio Datastorage attraverso un producer Kafka utilizzando il topic name *prometheusdata*.

Per finire, in questo servizio viene implementato un sistema di log mediante il modulo *logging*. Attraverso quest'ultimo viene creato e aggiornato un file di log limitato in dimensioni (100MB) chiamato *metric-log*. Questo modulo permette di cancellare le informazioni di log più vecchie dal file evitando quindi la costruzione di un file molto pesante. Nello specifico in questo file vengono scritti i tempi di query al server Prometheus, d'esecuzione del calcolo dei metadati (stazionarietà, stagionalità e autocorrelazione), del calcolo dei valori di massimo, minimo, media e deviazione standard distinguendo 1, 3 e 12 ore, e della predizione di quest'ultimi per il sottoinsieme di metriche contenute in SLA set.

2.2 Data Storage

Il microservizio Data Storage si occupa di salvare i dati prodotti dal microservizio ETL Data Pipeline, i quali vengono recuperati da un consumer Kafka iscritto al topic *prometheusdata*. Se si ottengono errori *retriable* si ritenta la subscription dopo 5 secondi. Una volta fatto questo, il servizio si occupa di fare il polling continuo dei messaggi inseriti nel topic dal producer ETL Data Pipeline. In particolare, i messaggi sono differenziati dal campo *datatype* presente

nel dizionario, per indicare se contengono dati o metadati, e in base a questo inserisce i valori in determinati campi delle tabelle del database.

2.3 Data Retrieval

Per il servizio Data Retrieval si è deciso di offrire una interfaccia REST che permette di recuperare i dati prodotti dal microservizio ETL Data Pipeline e salvati nel database. In particolare viene utilizzato un framework Web di python chiamato *Flask*. Per prima cosa si è connesso il database al servizio e, attraverso il decorator *@app.route()*, sono stati implementati quattro metodi GET per effettuare le query. Questi ultimi vengono descritti in dettaglio nel capitolo 3 sottosezione 3.1.
Il servizio Data Retrieval viene deployato in un container che espone la porta 5000, cioè quella di default per Flask mappandola sulla porta host 80, in modo che l'utente possa accedere agli endpoint illustrati più avanti.

2.4 SLA Manager

Questo microservizio gestisce la creazione e la modifica di un set di SLA per un sottoinsieme di *metric_set*, contenente un massimo di 5 elementi decisi dall'utente. Per ogni metrica è possibile inserire le soglie di minimo e massimo desiderate. Il sistema inoltre permette di visionare l'eventuale superamento di tali soglie, visualizzando sia se vi sono state violazioni nelle ultime 1, 3 e 12 ore, sia se ve ne saranno nei prossimi 10 minuti. Il sistema è progettato per contenere al più 5 nomi di metriche, nel caso si proceda con l'inserimento di un sesto elemento è prevista la cancellazione del precedente set e l'inserimento della nuova metrica. Per la comunicazione del servizio con l'utente si è scelto di utilizzare il protocollo di comunicazione RPC tramite il framework gRPC, realizzando pertanto un client che si interfaccia col server contenuto nel microservizio. Il set di SLA viene memorizzato nella tabella *sla* del database, la quale viene acceduta anche dal microservizio ETL Data Pipeline, come specificato nel paragrafo 2.1. Le API esposte sono trattate nel capitolo sottosezione 3.2.

3 API implementate

Di seguito sono illustrate le API esposte all'utente relative ai servizi Data Retrieval e SLA Manager.

3.1 API Data Retrieval

Il servizio che espone le API è disponibile alla porta 80 dell'host e fornisce i seguenti metodi GET:

- **/metrics_available**: mi restituisce tutte le metriche disponibili nel database;
- **metrics_available/<id_metrica>/metadata**: per la metrica avente valore di id pari a *id-metrica* mi ritorna i valori di stazionarietà, autocorrelazione e stagionalità;
- **/metrics_available/<id_metrica>/values**: stampa i valori di massimo, minimo, media e deviazione standard per le ultime 1, 3 e 12 ore della metrica avente id pari a *id-metrica*;
- **/metrics_available/<id_metrica>/prediction_values**: per la metrica con id pari a *id-metrica* ottengo i rispettivi valori di massimo, minimo e media predetti per i successivi 10 minuti considerando i campioni delle ultime 1, 3 e 12 ore.

3.2 API SLA Manager

Il Manager è disponibile alla porta 50051 dell'host ed implementa i seguenti metodi:

- **SetSla (MetricValue) returns (SlaReply)**: permette di inserire una delle metriche presenti nel *metric_set* con i relativi valori di minimo e massimo;
- **SlaStatus (SlaRequest) returns (SlaReply)**: visualizza l'intero SLA set memorizzato;
- **GetViolation (SlaRequest) returns (stream Violation)**: confrontando i valori massimi e minimi delle metriche con le soglie impostate nel set, restituisce le violazioni avvenute nelle ultime 1, 3, 12 ore, fornendo anche il valore del superamento della soglia;

- **GetFutureViolation (SlaRequest) returns (stream Violation)**: confrontando i valori massimi e minimi predetti con le soglie impostate nel set, restituisce le violazioni che potrebbero avvenire nei prossimi 10 minuti e il valore di superamento della soglia.

4 Istruzioni per l'utilizzo

Per avviare il sistema bisogna posizionarsi da terminale nella cartella contenente il file *docker-compose.yml*. Una volta fatto questo per il primo avvio è necessario costruire le immagini, definite nel Dockerfile di ogni microservizio, tramite:

```
$docker-compose build
```

Successivamente si avvia il tutto tramite:

```
$docker-compose up -d
```

In questo modo Docker-Compose avvia tutti i container, creando automaticamente una rete interna, nella quale collega tra loro i servizi ed espone verso l'host i microservizi Data Retrieval e SLA Manager tramite port mapping delle porte definite nel file.
Il sistema è quindi in funzione, con i processi di recupero dati e immagazzinamento in esecuzione.
Per visualizzare i risultati delle query del Data Retrieval è sufficiente interrogare l'URL "*localhost:80/*" seguito dalle route illustrate al capitolo 3 sottosezione 3.1.
Per la scelta del valore <id_metrica> da inserire è necessario andare all'URL

```
localhost/metrics_available
```

in modo da visualizzare per ogni metrica ed ogni nodo l'id associato. Per una migliore visualizzazione dei dati è consigliato l'utilizzo di Postman inserendo l'URL scelto, selezionando il metodo GET e impostando la visualizzazione della risposta nel formato JSON.
Per interagire con il servizio SLA Manager bisogna utilizzare il client posto nella cartella /SLAmanager e avviarlo in un terminale utilizzando il comando:

```
$python ./SLAmanager/client.py
```

dopodiché si può scegliere dal menù il comando da eseguire inserendo il numero corrispondente.
N.B.: è necessario che il nome della metrica venga inserito rispettando la nota-

zione Camel Case presente in Prometheus, con l'iniziale della prima parola in minuscolo.

4.1 Esempio di utilizzo

Una volta avviati tutti i container si vogliono visualizzare i valori ottenuti. Si procede quindi con l'accesso all'URL *localhost/metrics_available* dove è possibile visualizzare tutte le metriche disponibili del DB e scegliere quindi l'id di quelle per cui vogliono essere visualizzate altre informazioni.

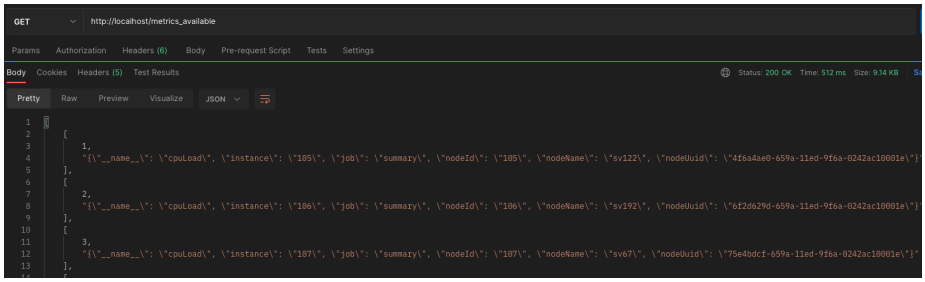

Figura 3: visualizzazione delle metriche disponibili

Per ogni metrica scelta mediante l'id è quindi possibile visualizzare:

- massimo, minimo, media e deviazione standard per 1h, 3h e 12h

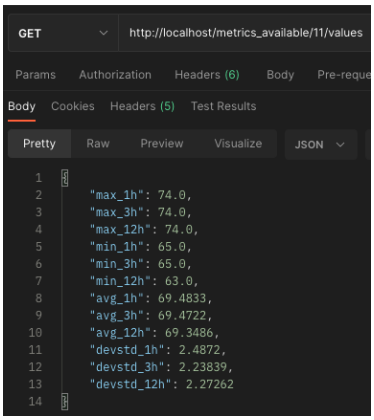

Figura 4: max, min, mean e std dev per la metrica con id 11

- i metadati dove, per la rispettiva metrica, vediamo, nel seguente ordine: il lag, il valore di autocorrelazione per quel lag, la stazionarietà e il periodo

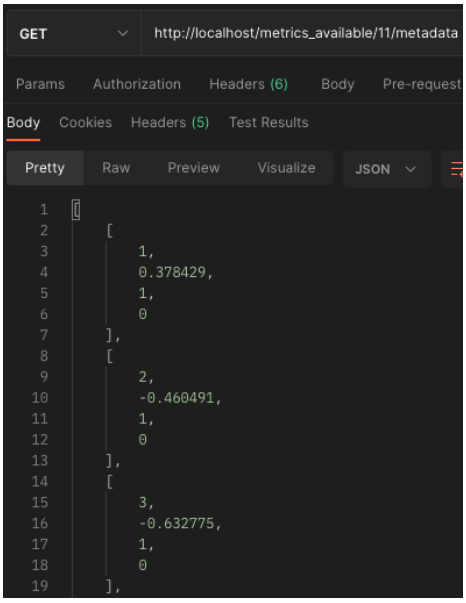

Figura 5: metadati per la metrica con id 11

- i valori predetti

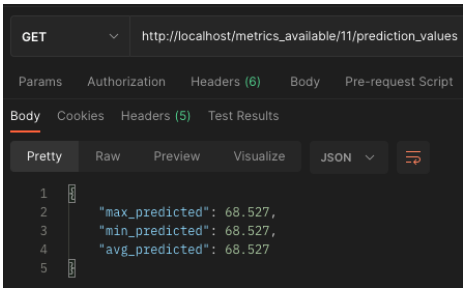

Figura 6: valori predetti nel caso della metrica con id 11 e appartenente all'SLA set definito

Al primo avvio SLA set non contiene elementi, si procede quindi all'inserimento di una metrica. Si avvia in un terminale il client contenuto nella cartella SLAmanager, il quale mostrerà il seguente menù.

```
PS C:\Users\feder\Documents\UProjects\DSBD\Progetto> python .\SLAmanager\client.py
1) Aggiungi SLA  2) Mostra SLA  3) Mostra violazioni  4) Mostra violazioni future  0) Exit
```

Selezionando il secondo comando è possibile vedere il set vuoto

```
PROBLEMS 3   OUTPUT   DEBUG CONSOLE   TERMINAL

MacBook Pro di Giorgia:DSBD_project giorgiarumorepagano$ /usr/local/bin/pytho
1) Add sla  2) Print sla  3) Get Violation  4) Get Future Violation  0) Exit
2
Richiesta SLA Status in corso...
SLA Status (nome, min, max):
```

Si procede quindi all'inserimento di una metrica con le relative soglie, in questo esempio si è scelto di inserire la metrica diskUsage con valore di minimo 0 e valore di massimo 30. Successivamente si inserisce una seconda metrica, cpuTemp con minimo 0 e massimo 50.

```
1) Aggiungi/modifica SLA  2) Mostra SLA  3) Mostra violazioni  4) Mostra violazioni future  0) Exit
1
Inserire il nome della metrica, il minimo e il massimo separati da spazi
diskUsage 0 30
Inserimento SLA
diskUsage 0 30
Confermare? y/n y
Valori per diskUsage inseriti correttamente
1) Aggiungi/modifica SLA  2) Mostra SLA  3) Mostra violazioni  4) Mostra violazioni future  0) Exit
1
Inserire il nome della metrica, il minimo e il massimo separati da spazi
cpuTemp 0 50
Inserimento SLA
cpuTemp 0 50
Confermare? y/n y
Valori per cpuTemp inseriti correttamente
```

A questo punto richiamando SLA set verrà visualizzata la lista con i valori appena inseriti

```
1) Aggiungi/modifica SLA  2) Mostra SLA  3) Mostra violazioni  4) Mostra violazioni future  0) Exit
2
Richiesta SLA Status in corso...
SLA Status (nome, min, max):
cpuTemp 0 50
diskUsage 0 30
```

Infine si vogliono visualizzare gli eventuali nodi che hanno violato i valori definiti da SLA

```
1) Aggiungi/modifica SLA  2) Mostra SLA  3) Mostra violazioni  4) Mostra violazioni future  0) Exit
3
Recupero violazioni SLA in corso...

Nome:
{"__name__": "cpuTemp", "instance": "108", "job": "summary", "nodeId": "108", "nodeName": "i70", "nodeUuid": "c22380bf-659a-11ed-9f6a-0242ac10001e"}
Soglia massima superata nelle precedenti 1 ore di 23.0

Nome:
{"__name__": "cpuTemp", "instance": "108", "job": "summary", "nodeId": "108", "nodeName": "i70", "nodeUuid": "c22380bf-659a-11ed-9f6a-0242ac10001e"}
Soglia massima superata nelle precedenti 3 ore di 24.0

Nome:
{"__name__": "cpuTemp", "instance": "108", "job": "summary", "nodeId": "108", "nodeName": "i70", "nodeUuid": "c22380bf-659a-11ed-9f6a-0242ac10001e"}
Soglia massima superata nelle precedenti 12 ore di 24.0

Nome:
{"__name__": "cpuTemp", "instance": "111", "job": "summary", "nodeId": "111", "nodeName": "invader-26", "nodeUuid": "2d430e81-659b-11ed-9f6a-0242ac10001e"}
Soglia massima superata nelle precedenti 1 ore di 21.0

Nome:
{"__name__": "cpuTemp", "instance": "111", "job": "summary", "nodeId": "111", "nodeName": "invader-26", "nodeUuid": "2d430e81-659b-11ed-9f6a-0242ac10001e"}
Soglia massima superata nelle precedenti 3 ore di 22.0

Nome:
{"__name__": "cpuTemp", "instance": "111", "job": "summary", "nodeId": "111", "nodeName": "invader-26", "nodeUuid": "2d430e81-659b-11ed-9f6a-0242ac10001e"}
Soglia massima superata nelle precedenti 12 ore di 25.0

Violazioni di cpuTemp totali: 6
```