Gruppo di Lavoro:

Sofia Manno 1000067618, Giuliano Sicali 1000014800

Abstract

Il progetto estende le funzionalità del sistema distribuito precedentemente sviluppato:
viene effettuato il porting della soluzione precedente su Kubernets ed è stato implementato un meccanismo di white-box monitoring.

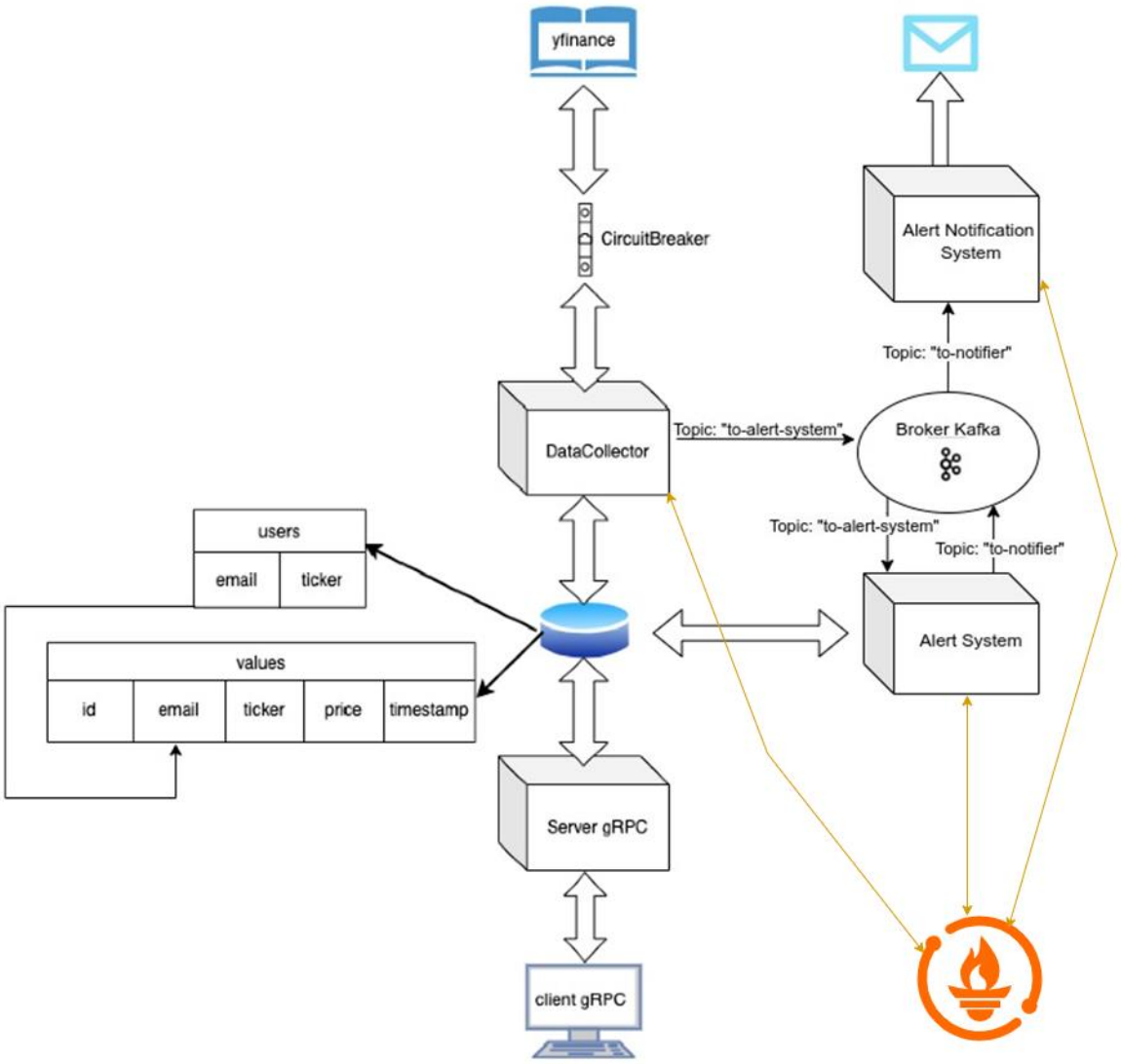
L'intero set di microservizi è stato distribuito su un cluster Kubernetes, creato tramite l'estensione Kubernetes su Docker per garantire una gestione scalabile e affidabile delle applicazioni containerizzate.

Il sistema di monitoraggio è stato sviluppato su exporter Promethues e l'obiettivo è il tracciamento delle performance di tre microservizi: AlertSystem, AlertNotificationSystem e DataCollector. Prometheus è stato configurato per raccogliere metriche dettagliate, tra cui tempi di risposta (GAUGE) e contatori di richieste ed errori (COUNTER), con etichette per identificare il servizio e il nodo di esecuzione.

Grazie a PromQL, sono state elaborate query avanzate per l'analisi dei dati, permettendo di ottenere insight dettagliati sul comportamento del sistema. Questo approccio ha migliorato la visibilità operativa e posto solide basi per la scalabilità e l'affidabilità dell'infrastruttura.

Diagramma Architetturale

Rispetto al precedente diagramma architetturale, è stato aggiunto solo Prometheus.



Kubernets

È stato effettuato il porting del sistema distribuito precedentemente sviluppato in Kubernets, è stata utilizzata l'estensione Kubernetes integrata in Docker.

Questa soluzione consente di eseguire un cluster Kubernetes in locale, sfruttando l'infrastruttura di containerizzazione di Docker per la gestione dei nodi del cluster senza la necessità di strumenti aggiuntivi come kind o minikube. Quindi, viene semplificato il deployment e la gestione dei container in un ambiente di sviluppo locale.

Le immagini Docker relative a ciascun microservizio sono state buildate localmente utilizzando il comando:

```
docker build -t <nome_utente>/<nome_image> .
```

Questo comando esegue la build dell'immagine partendo dal Dockerfile presente nella directory corrente (.) e assegna all'immagine il nome specificato. L'uso del formato *<nome_utente>/<nome_image>* è fondamentale per garantire la compatibilità con Docker Hub, poiché consente il versionamento e la gestione centralizzata delle immagini.

Una volta completata la fase di build, le immagini sono state caricate (push) nel registry remoto **Docker Hub** utilizzando:

```
docker push <nome_utente>/<nome_image>
```

Questa operazione consente di rendere l'immagine disponibile per il cluster Kubernetes, che potrà scaricarla direttamente dal registry pubblico o privato dell'utente.

Per separare la configurazione dall'immagine dei microservizi e garantire un sistema più flessibile e gestibile, sono state utilizzate due **ConfigMap**:

- **ConfigMap del Database**: Contiene la struttura del database, inclusa la definizione delle tabelle necessarie per l'operatività dei microservizi.
- **ConfigMap di Prometheus**: Definisce la porta locale utilizzata per l'esposizione delle metriche di ciascun microservizio, con valori specifici per ogni componente:
  - **DataCollector:** porta `8000`
  - **AlertNotificationSystem:** porta `8001`
  - **AlertSystem:** porta `8002`

Ogni **ConfigMap** è stata referenziata nei manifest deployment.yaml tramite la sezione env per passare i parametri di configurazione ai container, in modo che ogni servizio possa accedere correttamente al database e al sistema di monitoraggio Prometheus. Inoltre, a ciascun microservizio è stata associata una **label** che identifica il servizio e la connessione con il localhost.

Per il deployment dei microservizi all'interno del cluster Kubernetes (gestito tramite l'estensione di Docker Desktop), sono stati utilizzati i manifest definiti nei file *deployment.yaml* e *service.yaml*. L'applicazione delle risorse è avvenuta tramite il comando:

```
kubectl apply -f .
```

Il comando applica tutti i file YAML presenti nella directory corrente, creando o aggiornando le risorse specificate. Nel dettaglio:

1. **Deployment**: Kubernetes controlla l'immagine indicata nei manifest deployment.yaml. Se l'immagine non è presente localmente, viene eseguita una **pull** dal repository Docker Hub.
2. **Validazione e Pull**: Viene verificata la coerenza tra l'immagine richiesta e quella disponibile nel registry. Se l'immagine è valida, viene eseguito il pull.
3. **Creazione dei Pod**: Una volta scaricata l'immagine, Kubernetes genera un Pod, che rappresenta un'istanza del microservizio, eseguendo il container basato sull'immagine indicata.

4. **Service Binding**: Il file service.yaml definisce le regole per l'esposizione della rete del microservizio, creando un endpoint accessibile per il traffico interno o esterno al cluster (es. NodePort).

- **deployment.yaml**: definisce lo stato desiderato dei microservizi, specificando il numero di repliche, le policy di aggiornamento e le specifiche dei container, garantendo che Kubernetes mantenga sempre il numero corretto di pod in esecuzione e si occupi di eventuali riavvii o aggiornamenti.
  Per collegare al db viene usato l'environment (env) specificandone nome e porta.
- **service.yaml**: fornisce l'esposizione di rete dei microservizi attraverso Service di tipo NodePort, garantendo endpoint stabili per la comunicazione tra i componenti del cluster e l'esterno.

Il **Pattern CQRS** prevede la separazione delle operazioni di comando, che si occupano di modificare lo stato, da quelle di query, che invece leggono il dato, consentendo così una scalabilità indipendente dei servizi dedicati.
Per implementare questa architettura su Kubernetes, abbiamo sviluppato due microservizi distinti:

- **user_command_server**: responsabile delle operazioni di comando.
- **User_query_server**: dedicato alle operazioni di query.

Entrambi i servizi sono implementati come container che vengono distribuiti all'interno dello stesso pod, esponendo un unico service.

White Box Monitoring with Prometheus

Per monitorare in modo approfondito il comportamento dei microservizi nel sistema distribuito, è stato implementato un meccanismo di white-box monitoring utilizzando Prometheus. Questo approccio permette di raccogliere informazioni dettagliate sullo stato interno delle applicazioni,

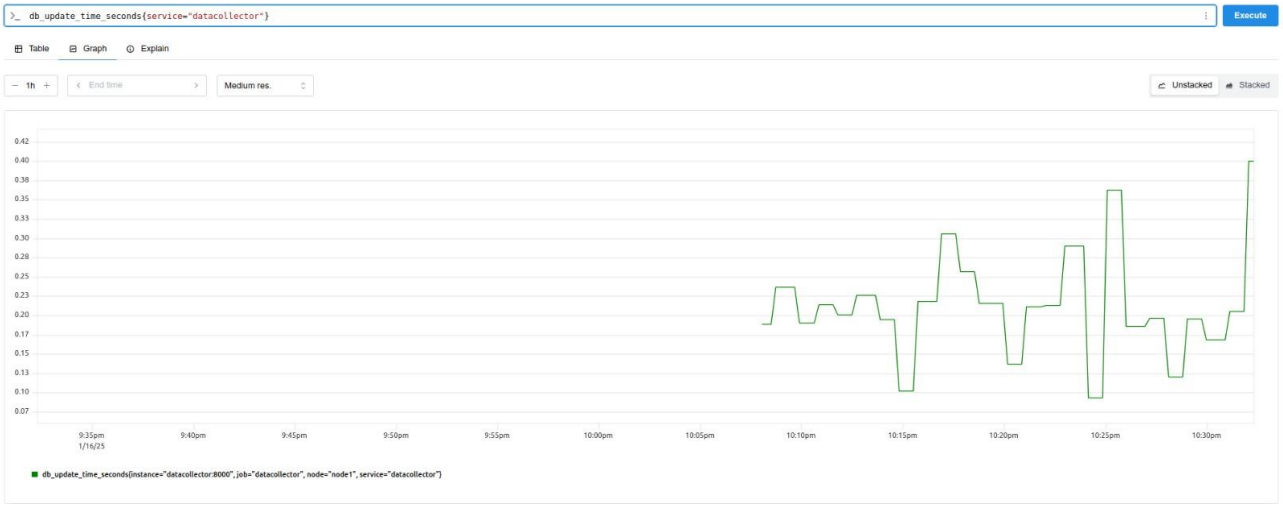
come il numero di richieste gestite, i tempi di risposta e la quantità di errori, facilitando l'individuazione di problemi e il miglioramento delle performance complessive.

Nel progetto, il monitoraggio è stato applicato ai tre microservizi principali: **DataCollector**, **AlertSystem** e **AlertNotificationSystem**, considerati i più critici per il corretto funzionamento del sistema.

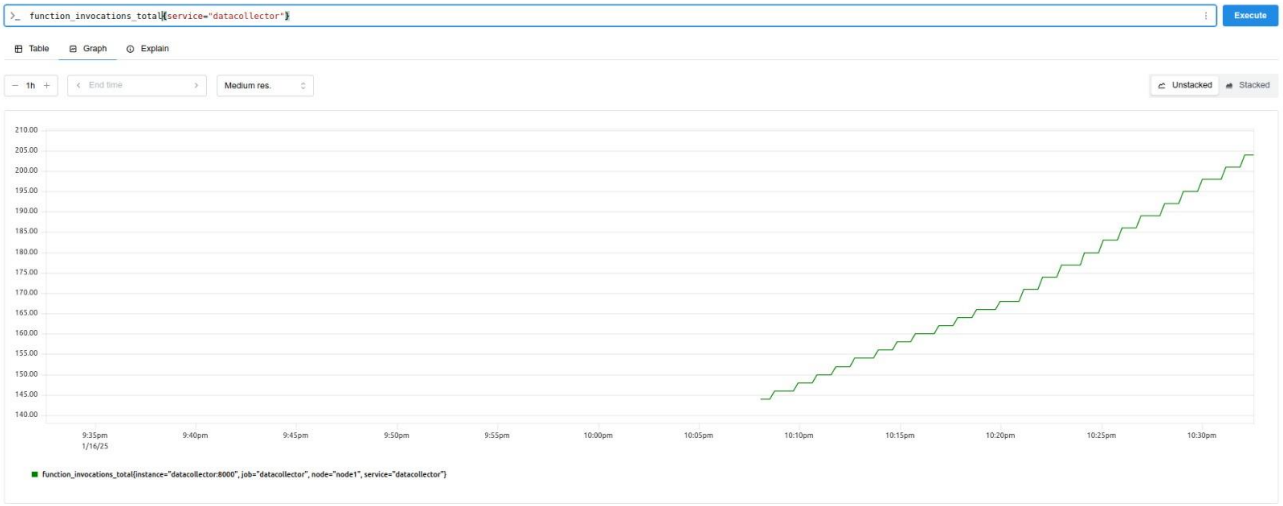
DataCollector

Per il microservizio DataCollector, sono state tracciate metriche chiave per monitorare sia il comportamento generale che le potenziali criticità. Le metriche includono etichette per identificare il servizio (datacollector) e il nodo di esecuzione (node1).

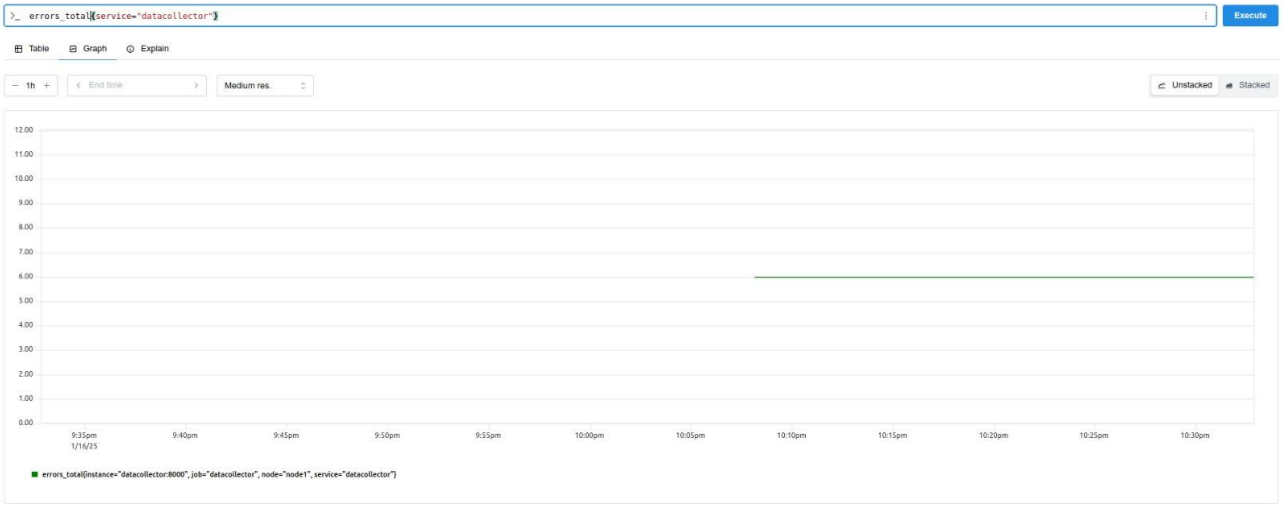
- **db_update_time_seconds (Gauge)**: misura il tempo necessario per aggiornare il database.



- **function_invocation_total (Counter)**: conteggia il numero totale di invocazioni delle funzioni del microservizio.



- **errors_total (Counter)**: registra il numero totale di errori verificatisi.

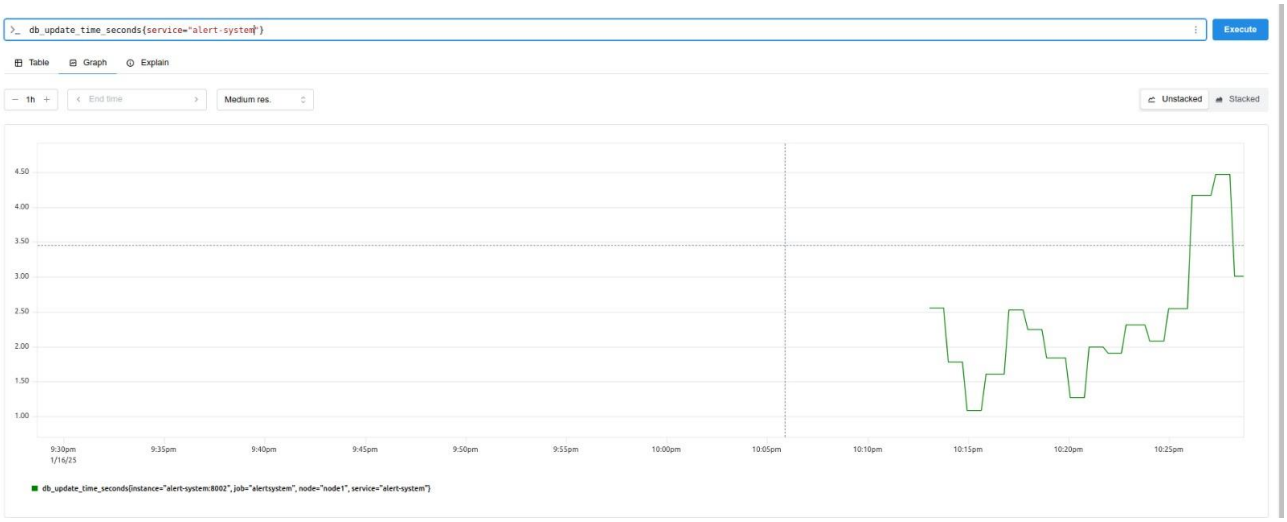


AlertSystem

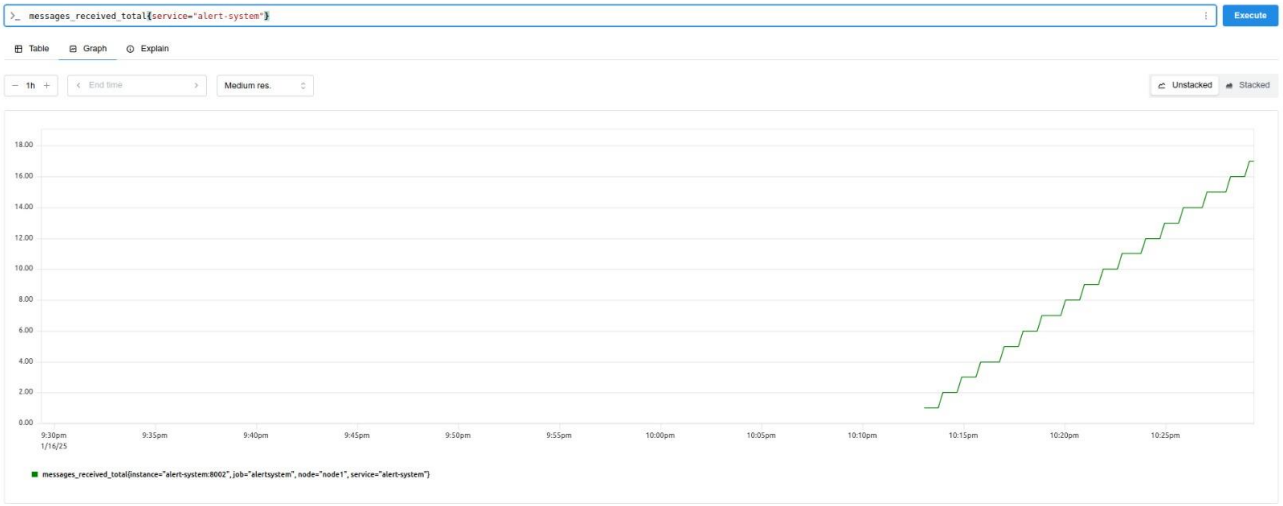
Nel microservizio AlertSystem, il monitoraggio si concentra sia sull'interazione con il database che sul traffico di messaggi. La label associata a questo set di metriche è alert-system e il nodo di esecuzione (node1).

- **response_time_seconds (Gauge)**: registra il tempo di risposta complessivo del microservizio.

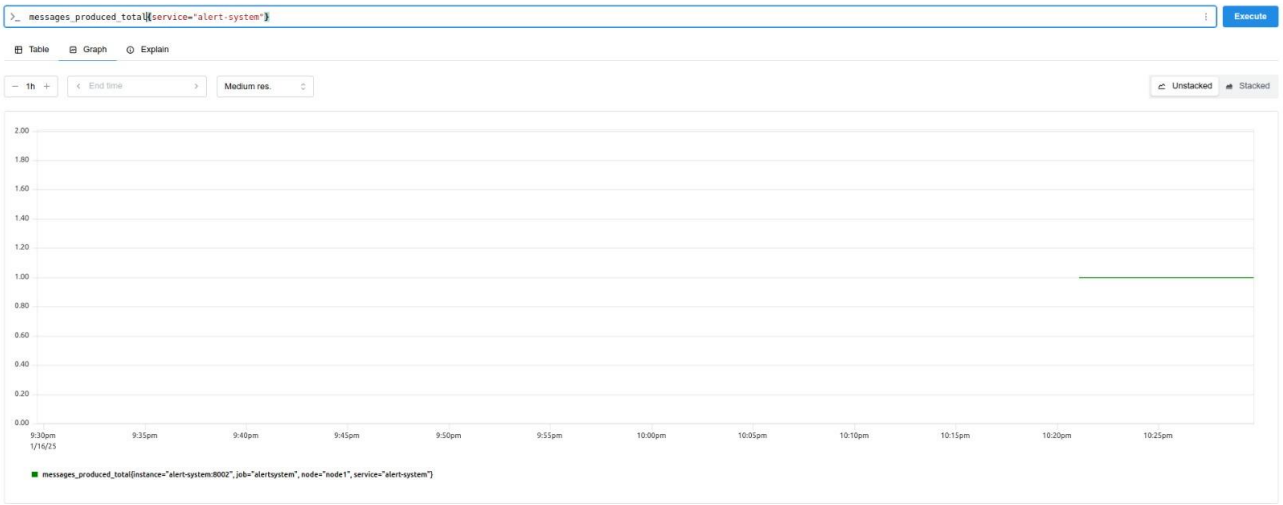
- **db_update_time_seconds (Gauge)**: misura il tempo per completare un aggiornamento del database.



- **messages_received_total (Counter)**: traccia il numero totale di messaggi ricevuti tramite Kafka.



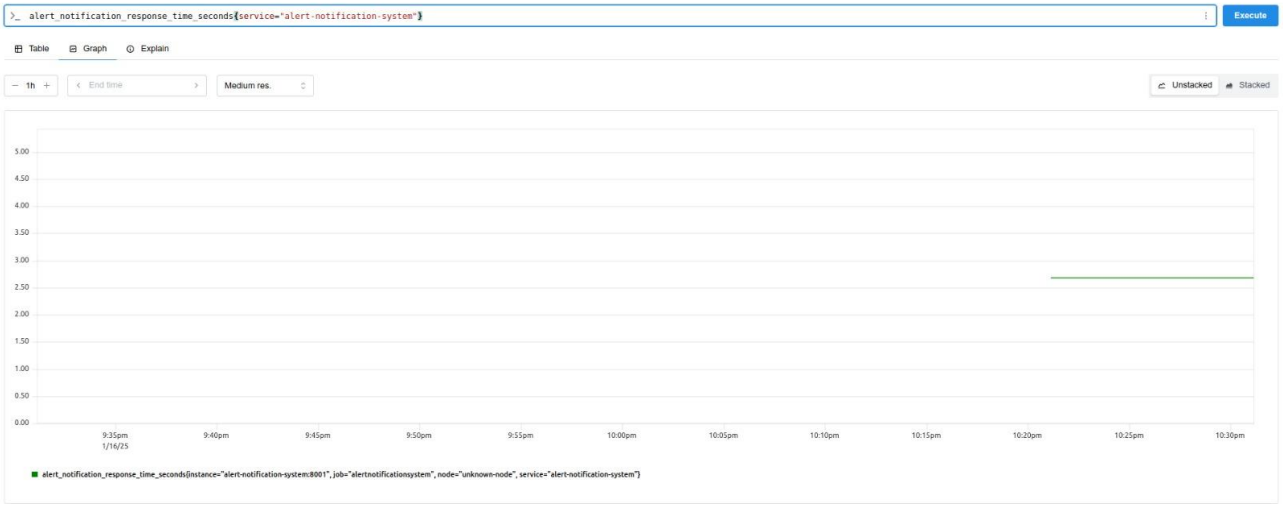
- **messages_produced_total (Counter)**: registra il numero totale di messaggi inviati dal microservizio.



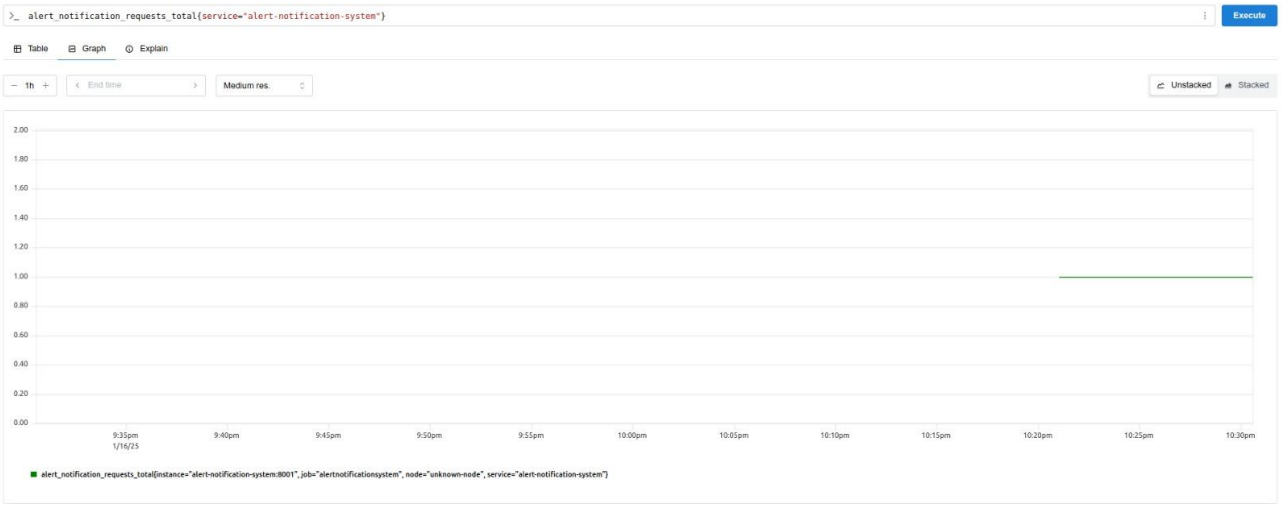
AlertNotificationSystem

Per AlertNotificationSystem, il focus è stato posto sul monitoraggio delle notifiche inviate e degli errori riscontrati. La label associata a questo set di metriche è alert-notification-system e il nodo di esecuzione (node1).

- **alert_notification_response_time_seconds (Gauge)**: registra il tempo di risposta per l'invio delle notifiche.



- **alert_notification_request_total (Counter)**: misura il numero di richieste per l'invio di email ricevute dal sistema.



- **alert_notification_errors_total (Counter)**: registra il numero totale di errori generati durante l'invio delle notifiche.

Utilizzo di PromQL per l'analisi delle metriche

Le metriche raccolte sono state analizzate utilizzando **PromQL** (Prometheus Query Language), un linguaggio specifico per interrogare i dati raccolti da Prometheus. Le query PromQL hanno permesso di:

- **Visualizzare l'andamento temporale delle metriche**: Ad esempio, per osservare come varia nel tempo il numero di richieste.
- **Calcolare il tasso di variazione**: Utilizzando funzioni come rate(), che calcola la velocità di crescita di una metrica nel tempo.
- **Identificare comportamenti anomali**: Come picchi improvvisi di errori o aumenti nei tempi di risposta.

I grafici generati con PromQL sono stati utilizzati per un'analisi dettagliata del comportamento del sistema, fornendo informazioni utili per il miglioramento continuo delle prestazioni e l'affidabilità complessiva."