Distributed Systems and Big Data 2023-2024

Weather Event Notifier

Alessandro Genovese 1000002043

Francesco Pennisi 1000055702

Indice

1) Abstract

Lo scopo di questo documento è illustrare le scelte compiute nella progettazione e nello sviluppo di un sistema distribuito che possa essere eseguito su piattaforme architecture-independent.

Il sistema in oggetto è realizzato seguendo un pattern architetturale a microservizi, i quali vengono impacchettati in appositi container facendo uso della tecnologia di containerizzazione Docker, la quale permette di raccogliere e isolare i microservizi in ambienti runtime completi e corredati di tutti i file necessari per l'esecuzione, in modo da garantire la portabilità su qualunque infrastruttura (hardware e software) che sia Docker-enabled.

Il sistema in esame ha l'obiettivo di permettere agli utenti registrati di indicare, per ogni località di interesse, dei parametri meteorologici, ad esempio la massima temperatura o l'eventuale presenza di pioggia. Tali parametri saranno monitorati dal sistema stesso e, in caso di violazione delle condizioni meteo specificate, gli utenti verranno notificati tramite e-mail.

A tale scopo, ad intervalli regolari, anch'essi a discrezione dell'utente, il sistema recupera le informazioni meteorologiche, appoggiandosi al servizio terzo *OpenWeather* (https://openweathermap.org/api) mediante richieste REST API. Tali informazioni vengono filtrate e opportunamente elaborate sulla base delle condizioni sottomesse dagli utenti.

2) Architettura del sistema

Il sistema, dal punto di vista funzionale, è composto dai seguenti microservizi: Weather Management Service (abbreviato in WMS), User Manager Service (UM), Worker Service e Notifier Service.

Inoltre, è stata prevista l'introduzione di un'attività di QoS management che prevede l'inserimento nell'architettura del sistema di tre ulteriori microservizi: Prometheus, Cadvisor e Service Level Agreement (SLA) Manager.

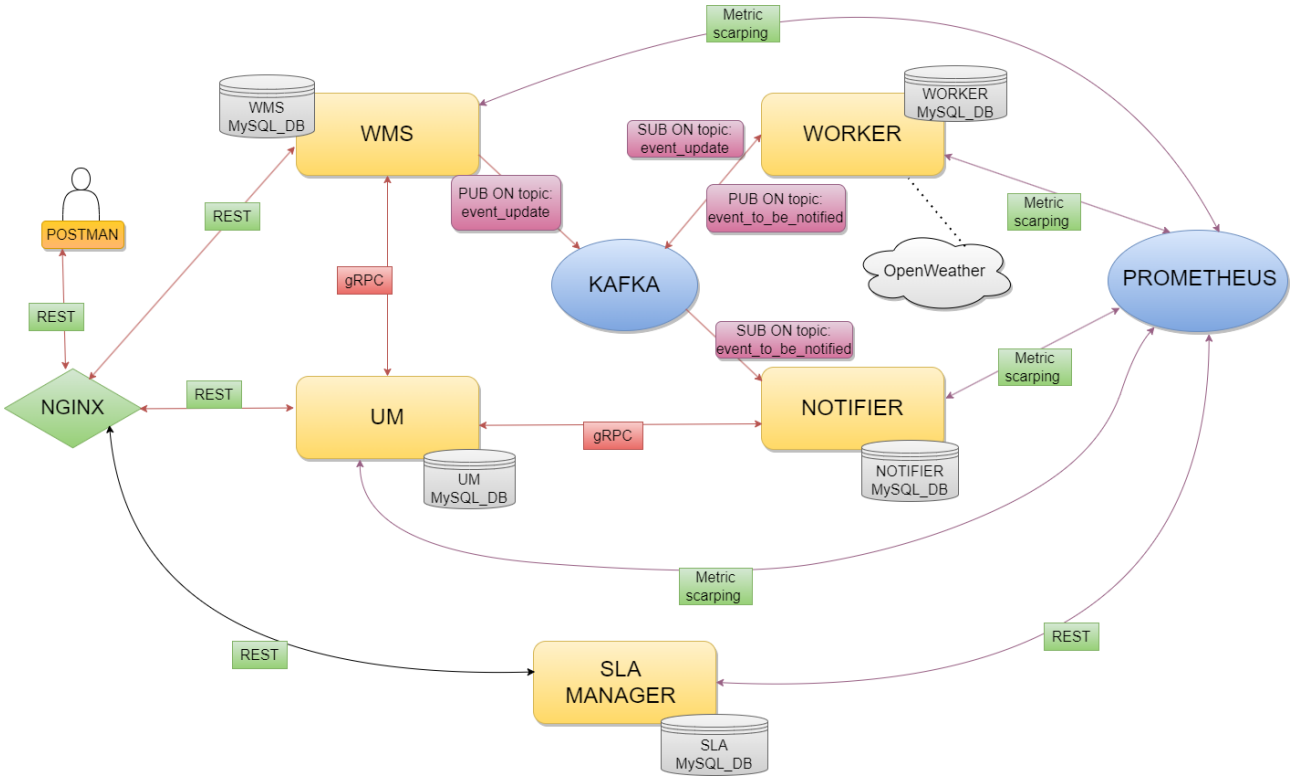

Figura 1 Architettura del sistema

Di seguito sono brevemente spiegati i microservizi utilizzati:

- **User Manager Service (UM):** consente la registrazione e la successiva autenticazione degli utenti nel sistema. L'autenticazione sfrutta la tecnologia JWT token; l'utente, dunque, attraverso l'utilizzo di un client (e.g. *Postman*) effettua il login tramite una richiesta REST API e ottiene un token JWT, il quale viene memorizzato e inserito nell'header *HTTP Authorization* in tutte le successive richieste effettuate dall'utente stesso al sistema.
  Nel momento in cui l'utente volesse inserire/modificare dei parametri meteorologici per una determinata località, egli contatterà il microservizio WMS, inserendo il token di autorizzazione nell'intestazione della richiesta. Quest'ultimo viene immediatamente passato allo User Manager per l'autenticazione dell'utente e, dopo aver validato il token, lo User Manager stesso restituirà al WMS le informazioni necessarie affinché la richiesta dell'utente possa essere soddisfatta con successo.

L'UM è collegato a un suo database MySQL che include la tabella *Users*. Tale tabella contiene le informazioni relative all'utente: indirizzo e-mail, password e codice identificativo univoco dell'utente.

- **Weather Management Service (WMS):** permette all'utente l'inserimento, la modifica e l'eliminazione dei parametri meteorologici relativi a una o più località a scelta dell'utente.
  Il WMS è collegato a un suo database MySQL che include la tabella *Locations*, in cui vengono memorizzate le informazioni relative alle località di interesse degli utenti, e la tabella *UserConstraints*.
  Quest'ultima contiene le regole specificate dagli utenti, un *trigger_period,* che indica ogni quanti minuti l'utente desidera che il sistema monitori le regole sottoscritte, e un timestamp, che viene aggiornato ogni volta che il WMS prende in considerazione la specifica entry.
  Il WMS, ciclicamente, attraverso l'utilizzo di un timer, controlla la tabella *UserConstraints* e, di volta in volta, prende in considerazione solo le entries per cui è trascorso il *trigger_period* dall'ultimo controllo.
  Il compito del WMS, in sostanza, è quello di costruire dei messaggi Kafka da pubblicare sul topic *event_update*, del quale il microservizio Worker è sottoscrittore.
  Tali messaggi includono le regole degli utenti di cui il Worker dovrà controllare le eventuali violazioni. La costruzione del messaggio è approfondita al paragrafo § 3.3.

- **Worker Service:** ha la responsabilità di contattare il servizio *OpenWeather* per ottenere i dati metereologici attuali e verificare l'eventuale violazione dei parametri specificati dagli utenti.
  Ciò avviene attraverso il seguente ordine procedurale: il worker preleva dal topic Kafka *event_update* i messaggi pubblicati dal WMS, estrae le regole da controllare, contatta il servizio *OpenWeather* e confronta i risultati ottenuti con quelli ricevuti dal WMS.
  Nel caso in cui si presentino delle violazioni, il Worker le pubblicherà in un messaggio sul topic Kafka *event_to_be_notified*. All'interno del messaggio, ogni parametro violato è associato all'utente, che ne ha richiesto il monitoraggio tramite, il suo codice identificativo univoco.

- **Notifier Service:** il suo compito è quello di prelevare eventuali messaggi Kafka pubblicati sul topic *event_to_be_notified* dal Worker.
  Una volta estratto il messaggio, si occupa di avvisare gli utenti delle avvenute violazioni mediante l'invio di una mail.

Il Notifier è in grado di recuperare l'e-mail dell'utente contattando lo User Manager tramite il meccanismo di comunicazione gRPC; è possibile identificare univocamente l'utente grazie al suo codice identificativo contenuto nel messaggio Kafka.

- **Prometheus:** è un sistema open source utilizzato come strumento di monitoraggio che offre la possibilità di monitorare le proprie applicazioni conservando le metriche monitorate.
  Ogni microservizio precedentemente trattato espone delle metriche al proprio endpoint *metrics*.
  Prometheus effettua, ad intervalli regolari, uno scraping di tali metriche attraverso una richiesta REST API all'endpoint sopra menzionato.
  Le metriche collezionate sono a loro volta reperibili, da Prometheus stesso, tramite richieste REST API.

- **SLA Manager:** permette di definire il *Service Level Agreement* dell'applicazione come composizione di un set di metriche specificate dall'admin del sistema. Più specificatamente, permette all'amministratore di sistema di aggiungere, modificare e rimuovere le metriche del *Service Level Agreement*. L'admin stesso, per ogni metrica, specifica i valori target che costituisco i *Service Level Objectives* (SLO).
  Il microservizio in questione ottiene i *Service Level Indicators* (SLI) attraverso delle query in *PROMQL* al microservizio Prometheus.
  In particolare, esso permette di controllare lo stato di ogni metrica e di confrontare i valori attuali con quelli target, segnalando il numero di violazioni avvenute.
  In aggiunta, è stato implementato un meccanismo di previsione dei valori futuri delle metriche sottoscritte dall'admin. In tal modo, è possibile calcolare la probabilità che ci siano delle violazioni in un futuro prossimo.

- **Kafka:** è un sistema di messaggistica distribuita di tipo publish-subscribe, progettato per essere veloce, scalabile e consentire la persistenza dei messaggi. Tutti gli attori che interagiscono con Kafka possono essere considerati client, publisher e/o subscriber.
  I publisher pubblicano dei messaggi su un topic, mentre i subscriber reperiscono i messaggi pubblicati sui topic a cui si sono sottoscritti.
  La comunicazione è pensata per essere asincrona e indiretta, in modo che non ci sia il bisogno che i processi comunicanti conoscano i rispettivi indirizzi IP. Ogni client comunica in maniera diretta esclusivamente con il broker Kafka.

Un messaggio è costituito da dati di qualsiasi natura. Spetta ai client definire il formato del messaggio stesso.

- **Nginx:** è un web-server open-source ad alte prestazioni, utilizzato anche come reverse proxy. Garantisce un basso consumo di memoria e un'elevata concorrenza. Anziché creare nuovi processi per ogni richiesta web, Nginx utilizza un approccio asincrono, basato sugli eventi, in cui le richieste vengono gestite in un singolo thread. Nel progetto in esame è stato utilizzato con la funzione di API gateway, in modo da assicurare la *Location transparency* e la *Partition transparency*.
  Nginx riceve le richieste dell'utente tramite richieste REST API e le inoltra all'appropriato microservizio tra User Manager, WMS ed SLA Manager.

3) Scelte Progettuali

Nel seguente capitolo sono illustrate le scelte progettuali in merito alla gestione della fault tolerance, delle repliche e della comunicazione.

3.1) Fault tolerance

Ogni microservizio è autonomo e indipendente dagli altri; nonostante ciò, ognuno di essi può subire un proprio guasto, creando un effetto domino che può far crollare l'intero sistema. Implementare dei meccanismi di fault tolerance assicura che un sistema possa continuare a funzionare e fornire servizi anche se qualcosa andasse storto.

Nel sistema in questione la fault tolerance è stata implementata grazie all'utilizzo di database MySQL agganciati ai relativi microservizi. Nello specifico, i database associati al Worker e al Notifier hanno lo scopo di memorizzare il loro lavoro corrente.

Sia il Worker che il Notifier sono dei client subscriber di Kafka. Prendendo in considerazione il Worker, una volta prelevati i dati ed eseguito il commit su Kafka, esso è l'unico nodo che possiede i dati stessi; dunque, se dovesse andare in down, perderebbe le informazioni acquisite e quindi si avrebbe un fallimento del sistema. Per risolvere questo problema, è stato collegato un database MySQL al Worker stesso, in modo tale che quest'ultimo, prima di fare il commit a Kafka, conservi le informazioni nel suo DB. In tal modo, se il nodo dovesse cadere dopo aver fatto il commit a Kafka, al suo riavvio sarebbe in grado di svolgere correttamente il suo lavoro. Invece, se dovesse cadere prima di aver memorizzato i dati nel suo DB, il problema verrebbe

risolto da Kafka stesso, poiché il broker, non avendo ricevuto il commit, al riavvio del Worker invierebbe nuovamente lo stesso messaggio.

Il caso più delicato si verificherebbe se il Worker cadesse dopo aver memorizzato i dati nel proprio DB, ma prima di aver eseguito il commit. In questo caso specifico, il worker, al suo riavvio, prima gestirebbe il lavoro in sospeso presente nel suo DB, non ancora portato a termine, dopodiché riceverebbe da Kafka per la seconda volta lo stesso messaggio e dunque lo elaborerebbe due volte.

Nel sistema in esame, tuttavia, questo non rappresenta un problema, poiché la doppia rielaborazione di uno stesso messaggio potrebbe al più comportare l'invio di due mail uguali all'utente. Comunque, il possibile verificarsi di tale situazione non rappresenta un fallimento del sistema, difatti si è preferito implementare Kafka seguendo l'approccio *at least once*, ovvero si è voluto assicurare che ogni messaggio arrivi a destinazione almeno una volta. Sarebbe molto più grave per il sistema, infatti, perdere del tutto un messaggio e mancare l'invio della notifica all'utente.

Per quanto riguarda il Notifier il discorso è analogo, poiché esso semplicemente estrae i dati pubblicati dal Worker e si occupa dell'effettivo invio della notifica.

È bene sottolineare che il Worker pubblicherà un messaggio solo se sono avvenute delle violazioni dei parametri indicati dall'utente. Questo significa che, qualora il Worker dovesse rientrare nel caso precedentemente esposto e dunque ricevere due messaggi uguali, se tali messaggi non presentassero delle violazioni, allora non avverrebbe la doppia pubblicazione sul topic Kafka e dunque non ci sarebbe nemmeno la duplicazione della notifica per l'utente.

Infine, nel database del Notifier è stato previsto un campo *sent*, il quale indica se una data notifica è stata già inviata o meno. Ciò consente di mantenere uno storico delle notifiche inviate e, per ognuna di esse, vi è associato anche un timestamp per indicare il momento dell'invio. In più, in caso di riavvio del microservizio, questo campo permette di recuperare eventuali notifiche non ancora inviate; difatti, il Notifier appena avviato come prima azione controlla se ha nel suo DB delle notifiche da inviare.

3.2) Gestione delle repliche

Il sistema in esame è adatto alla replicazione dei suoi microservizi. Ciò consente di distribuire il carico di lavoro di ogni microservizio alle rispettive

repliche, mantenendo la consistenza dei dati e permettendo una maggiore scalabilità del sistema stesso.

Più nel dettaglio, ogni istanza del Worker e del Notifier è caratterizzata da un proprio codice identificativo univoco (*worker_id, notifier_id*). Tale codice costituisce uno dei campi dei loro rispettivi DB. In tal modo, ogni replica andrà ad agire solo sulle entries di sua pertinenza, velocizzando l'esecuzione delle query pur mantenendo un unico database condiviso tra le varie repliche. Questo è possibile poiché i precedenti microservizi non hanno bisogno di mantenere uno stato; semplicemente, l'uso dei loro database è pensato solo per fornire un meccanismo di fault tolerance.

Per quanto riguarda il WMS, anch'esso può essere replicato. A differenza dei microservizi precedenti, non è stato previsto l'utilizzo di un codice identificativo univoco nel database. Il motivo di questa scelta risiede nel fatto che il WMS ha il compito di mantenere l'elenco aggiornato delle regole di ciascun utente. Infatti, dato che un utente può modificare le proprie regole, l'uso di un codice identificativo della specifica istanza del WMS avrebbe ostacolato la modifica dei parametri metereologici degli utenti, perché sarebbe stato necessario indirizzare le richieste di ogni utente alla specifica istanza che ne detenesse le regole. Ciò avrebbe comportato la perdita della *Location transparency* e una diminuzione della scalabilità.

Tuttavia, il WMS è comunque replicabile essendo che le diverse repliche sfruttano un *timestamp,* presente nel database, che indica il momento in cui una specifica regola è stata presa in carico da una qualsiasi istanza del WMS, ovvero utilizzata per la costruzione di un messaggio Kafka da pubblicare sul topic *event_update*.

Dunque, ogni istanza del WMS, a intervalli regolari, controlla se esistono delle entries per cui sia trascorso il *trigger_period* a partire dal valore presente nel campo *timestamp*. Se ciò si dovesse verificare, allora l'istanza provvederebbe alla costruzione del messaggio Kafka inserendovi i dati di quelle specifiche entries. Il campo *timestamp* verrà aggiornato solo quando l'istanza del WMS avrà ricevuto da Kafka l'ack della corretta consegna.

Durante l'attesa dell'ack, potrebbe capitare che un'altra istanza acceda al database e prenda in carico le stesse entries considerate nel messaggio Kafka di cui si attende l'ack; ciò comporterebbe la costruzione di due messaggi Kafka identici in un tempo minore rispetto a quello del *trigger_period.* Questo avrebbe come conseguenza la possibilità dell'invio di due notifiche esattamente identiche per lo stesso utente. Per risolvere tale problema, è stato aggiunto un campo booleano *checked* nella tabella

UserConstraints del database del WMS. Questo campo permette di segnalare che una certa istanza ha già preso in carico quella specifica entry, ha costruito un messaggio Kafka e ne attende l'ack. Nessun'altra istanza del WMS dovrà prendere in carico la stessa entry fino alla ricezione dell'ack del messaggio in cui questa entry è stata incorporata e, per assicurare ciò, si fa uso del campo *checked*.

Infine, anche lo User Manager è stato implementato in modo tale da essere replicabile, poiché vi è un database condiviso fra tutti le repliche degli UM Services. Inoltre, avendo implementato un meccanismo di autenticazione basato sull'utilizzo di un token JWT, il quale viene incluso nell'header HTTP di ogni richiesta proveniente dal client, ciascuna replica potrà autenticare l'utente senza che venga instaurata una sessione.

3.3) Comunicazioni

Le tecnologie di comunicazione che sono state sfruttate sono:

- **REST API**
- **gRPC**
- **Apache Kafka Message**

La comunicazione REST API viene utilizzata dal client (e.g. *Postman)* cosicché le richieste dell'utente possano arrivare all'API gateway Nginx. A sua volta, Nginx sfrutta la stessa tecnologia per comunicare con i microservizi WMS, UM ed SLA Manager. Infine, anche Prometheus la sfrutta per effettuare lo scraping delle metriche. È stata scelta tale tecnologia per via della sua semplicità e per l'alta disponibilità di client già esistenti in grado di eseguire richieste REST. Nei cinque microservizi sviluppati nel sistema in esame (visibili in giallo nella Figura 1 *Architettura del sistema*) è stato implementato un server Flask per esporre i relativi endpoint.

La tecnologia gRPC viene utilizzata per la comunicazione interna del sistema, ovvero per creare un canale da WMS a User Manager per autenticare l'utente tramite l'utilizzo del token JWT, passato come parametro della richiesta gRPC. Inoltre, è impiegata anche per far comunicare il Notifier con l'UM al fine di ottenere l'indirizzo e-mail dell'utente. È stata scelta questa tecnologia perché permette di avere elevate prestazioni e non ha la necessità di esporre degli endpoint. Inoltre, tramite questa soluzione, il programmatore non si occupa degli aspetti espliciti della comunicazione, ma solo di definire le interfacce delle procedure remote, richiamandole

come delle funzioni locali. Ciò rende più snella e chiara l'interazione tra i microservizi che compongono il sistema.

La tecnologia Apache Kafka Message è stata già discussa. In questo paragrafo è spiegato come vengono costruiti i messaggi Kafka da pubblicare sui topic *event_update* ed *event_to_be_notified*.

Il WMS, invece di pubblicare un messaggio per ogni entry nel suo database, costruisce un messaggio Kafka per città di interesse, ovvero raggruppa tutti gli utenti interessati alla specifica località e aggiunge le regole di ciascun utente. Quindi, verranno pubblicati tanti messaggi quante sono le località differenti, anziché quante sono le entries. Questo evita di appesantire il broker Kafka. Ad esempio, supponendo ci siano dieci utenti interessati alla città di Catania e ognuno specifichi dieci parametri da controllare, allora nel database del WMS ci sarebbero dieci entries, una per utente. In questo caso, il WMS non costruirà dieci messaggi, bensì, essendo tutti gli utenti interessati alla città di Catania, costruirà un singolo messaggio contenente tutte le informazioni riguardanti gli utenti e le rispettive regole.

Inoltre, il messaggio così costruito permette al Worker, che è sottoscrittore al topic, di effettuare ad *OpenWeather* una sola richiesta per città. In questo modo, il Worker può controllare tutti i parametri specificati da tutti gli utenti interessati a quella città con un'unica richiesta, riducendo al minimo la dipendenza dal servizio esterno.

Riprendendo l'esempio precedente, ponendo il caso che ognuno dei dieci utenti abbia indicato dieci parametri metereologici da controllare, quindi per un totale di cento parametri, il worker potrà verificarli tutti attraverso una singola richiesta.

Tale strategia di costruzione dei messaggi favorisce la scalabilità dei microservizi, poiché riduce enormemente il carico di lavoro, sia del WMS, sia del broker Kafka, sia del Worker.

I messaggi Kafka sono delle stringhe codificate in *json*.

A titolo d'esempio, si riporta un possibile messaggio pubblicato dal WMS.

```
{
  "user_id": [1,2,3]
  "location": ["nome","lat","long","country","state"]
  "max_temp" : [40, 30,39]
  "min_temp": [2, "null", "null"]
  "max_humidity": ["null","null", 2]
  "min_humidity": [50, 70, "null"]
  "max_pressure": [1100, 1070, 1050]
  "min_pressure": [980, 1000, "null"]
  "max_wind_speed": ["null", "null", 10]
  "min_wind_speed": ["null", 5, "null"]
  "wind_direction": ["SE", "E", "N"]
  "rain": [True, "null", True]
  "snow": [True, "null", "null"]
  "clouds_max": [90, 80, 70]
  "clouds_min": [10, 20, 25]
}
```

Figura 2 Esempio messaggio Kafka pubblicato dal WMS

La coppia la cui chiave è "user_id" presenta come valore una lista dei codici identificativi degli utenti interessati alla località specificata nel messaggio, le cui informazioni sono riportate nella lista corrispondente alla chiave "location".

Le successive coppie chiave-valore possiedono una chiave che identifica la regola da monitorare e hanno come valore una lista contenente o i valori target scelti dagli utenti oppure un valore "null" qualora l'utente non fosse interessato al monitoraggio del parametro. L'associazione dell'utente con il suo valore target della regola è impostata su base colonna, ovvero il primo utente della chiave "user_id" è associato a tutti i primi elementi delle liste corrispondenti ad ogni regola.

Per quanto riguarda la costruzione del messaggio Kafka da parte del Worker, si è adottata una logica similare a quella del caso appena discusso, ovvero si è seguita l'idea di pubblicare un solo messaggio per località. Quest'ultimo contiene le informazioni sulla località e delle coppie chiave-valore, la cui chiave è lo "user_id" e il cui valore è una lista di coppie che contengono, a loro volta, il nome del parametro violato come chiave, associata al valore attuale. Anche in questo caso, la strategia di costruzione adottata consente di ridurre al minimo il numero di pubblicazioni che il Worker deve compiere e, inoltre, permette di legarsi perfettamente con la logica utilizzata per la costruzione del messaggio da parte del WMS.

Si riporta, a titolo d'esempio, un possibile messaggio pubblicato dal Worker.

```
{
"location": [name, lat, long, country, state]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
"user_id" : [{violated_rule: value}, ... ]
}
```

Figura 3 Esempio messaggio Kafka pubblicato dal worker

Per come è costruito il messaggio, il Notifier sarà facilmente in grado di estrarre le informazioni di suo interesse e di inviare l'e-mail di notifica ad ogni utente.

3.4) Endpoint REST

Di seguito sono indicati gli endpoint esposti per le richieste REST da parte del client. Essi sono suddivisi in base al microservizio a cui afferiscono.

User Manager:

- */register* (POST): per la registrazione degli utenti, i quali forniscono e-mail e password.
- */login* (POST): per l'autenticazione degli utenti; il client riceve come risposta il token JWT.
- */delete_account* (POST): per l'eliminazione dell'account dell'utente e delle relative regole.

WMS:

- */update_rules* (POST): per l'inserimento o l'aggiornamento dei parametri meteorologici, per una data località, scelti dall'utente.
- */show_rules* (POST): per visualizzare le regoli correnti dell'utente loggato.
- */update_rules/delete_user_constraints_by_location* (POST): per eliminare tutte le regole dell'utente relative a una data località.

SLA Manager:

- */adminlogin* (POST): per il login dell'admin; il client riceve come risposta il token JWT.
- */SLA_update_metrics* (POST): per l'inserimento o l'aggiornamento del set di metriche da monitorare.
- */SLA_delete_metrics* (POST): per l'eliminazione delle metriche selezionate.
- */SLA_metrics_status* (GET): per visualizzare lo stato di ogni metrica, ovvero se è avvenuta o meno una violazione.
- */SLA_metrics_violations* (GET): per visualizzare il numero di violazioni avvenute nell'ultima ora e nelle ultime tre e sei ore.
- */SLA_forecasting_violations* (GET): per ottenere la probabilità di violazione di una data metrica nei prossimi minuti.

Il dettaglio dei parametri necessari per ogni richiesta è indicato nel file "README.md".

Infine, i microservizi che necessitano di monitoraggio sulle metriche di performance presentano un endpoint (GET) */metrics*, il quale viene sfruttato da Prometheus per effettuare lo scraping.

3.5) Implementazione pattern SAGA

Nell'ambito dell'esecuzione della richiesta corrispondente all'endpoint */delete_account*, è necessaria, oltre che l'eliminazione dell'account stesso, presente nel database dell'UM, anche l'eliminazione delle sue regole, presenti nel database del WMS. Pertanto, è necessario realizzare una transazione distribuita per assicurare l'atomicità delle azioni di eliminazione e, di conseguenza, la consistenza dei dati.

A tale scopo, si è deciso di seguire il pattern SAGA.

Secondo questo modello, la transazione è divisa in passi più piccoli chiamati "compensazioni" o "sottotransazioni". Ogni passo rappresenta una parte della transazione che può essere eseguita in modo atomico. I passi della transazione vengono eseguiti in sequenza. Ognuno di essi ha un'operazione di esecuzione associata e un'operazione di compensazione. La prima completa il lavoro previsto, mentre la seconda, nel caso in cui si verifichi un problema, annulla il lavoro svolto dal passo precedente. La coordinazione tra i vari passi viene gestita in modo che, se uno dei passi fallisce, venga attivata la sequenza di compensazione per ripristinare lo stato coerente del sistema.

Il modello di progettazione SAGA aiuta a garantire che, anche in presenza di guasti o errori, il sistema possa recuperare in modo coerente le informazioni e che le transazioni distribuite siano trattate in modo robusto.

Nel sistema in questione, il primo passo della transazione è l'eliminazione delle regole sottoscritte dall'utente. Successivamente, si procede all'eliminazione dell'account. Se non dovesse andare a buon fine l'eliminazione dell'account, allora si procederebbe a ripristinare i dati eliminati dal database del WMS, riportando il sistema ad uno stato consistente.

3.6) Gestione della sicurezza mediante l'uso dei Secrets

In questo paragrafo sono trattate le problematiche relative alla sicurezza.

Per evitare di esporre le password di accesso ai database, la password dell'e-mail impiegata per l'invio delle notifiche e l'API key utilizzata per beneficiare del servizio esterno offerto da *OpenWeather*, si è scelto di implementare i Secrets.

Un Secret è un qualsiasi dato che contiene informazioni sensibili, che non deve essere trasmesso su una rete o archiviato non crittografato in un Dockerfile o nel codice sorgente dell'applicazione. La gestione dei Secrets è un procedimento che assicura che le informazioni sensibili, necessarie per l'esecuzione delle operazioni quotidiane, restino confidenziali. Rafforza la sicurezza in tutti gli ambienti di sviluppo e produzione aziendali senza ostacolare però i flussi di lavoro.

Nel sistema in esame, i Secrets sono pensati per fare in modo che le informazioni sensibili vengano immesse, in fase di building locale delle immagini, ognuna all'interno di un file testuale posizionato nel file system del container. In tal modo, si permette di recuperare l'informazione ma al tempo stesso si evita di cristallizzare il contenuto sensibile nelle immagini dei vari microservizi pubblicate sul Docker Hub.

4) Monitoraggio QoS

L'attività di monitoraggio del sistema prevede l'utilizzo di un server Prometheus che permette di estrarre le metriche esposte dai microservizi da monitorare, collezionarle in maniera persistente e renderle disponibili per il microservizio SLA Manager.

4.1) Serie temporali analizzate

Le metriche esposte sono collezionate da Prometheus sottoforma di serie temporali.

Nel presente progetto si è deciso di monitorare delle metriche di performance, di controllo sullo stato di salute del sistema e di monitoraggio delle risorse sfruttate dai container.

Metriche di performance:

- Tempi di esecuzione delle query ai database, differenziandole in base al relativo microservizio.
- Tempo di risposta alle richieste del client sia da parte del WMS che da parte dell'UM.
- Latenza nel tempo di notifica verso l'utente, ovvero tempo impiegato per inviare le notifiche agli utenti dopo la rilevazione di condizioni metereologiche avverse.
- Tempo di risposta del servizio esterno *OpenWeather* per valutarne la qualità.

Metriche sullo stato di salute del sistema:

- Numero di richieste pervenute allo User Manager.
- Numero di richieste pervenute al WMS.
- Conteggio degli errori HTTP per identificare problematiche specifiche.
- Numero di notifiche inviate.
- Numero di errori verificatisi durante il processo di invio delle notifiche.
- Numero di richieste effettuate al servizio esterno *OpenWeather* per monitorare la dipendenza da tale servizio.
- Numero delle regole attive nel WMS.
- Numero di utenti registrati.
- Numero di comunicazioni tra Notifier e User Manager.
- Numero di comunicazioni tra WMS e User Manager.
- Numero di messaggi pubblicati su Kafka da parte del WMS.
- Numero di messaggi pubblicati su Kafka da parte del Worker.
- Numero di ACK ricevuti dal WMS da parte di Kafka.

- Numero di ACK ricevuti dal Worker da parte di Kafka.

Per quel che riguarda le metriche di monitoraggio delle risorse sfruttate dai container si è previsto l'inserimento del microservizio Cadvisor, che è uno strumento realizzato da Google che produce numerose metriche relative ai componenti del sistema, dalle statistiche dell'hardware, sia per la CPU che per la memoria, all'utilizzo della rete da parte dei container. Esempi di metriche raccolte da Cadvisor sono le seguenti:

- *"container_memory_usage_bytes"*: contiene i dati circa l'utilizzo attuale della memoria per ogni container;
- *"container_cpu_system_seconds_total"*: rappresenta il tempo cumulativo di esecuzione della CPU per ogni container;
- *"container_memory_cache"*: rappresenta il numero di bytes nella memoria cache per ogni container;
- *"container_network_transmit_bytes_total"*: mostra il numero totale di bytes trasmessi per ogni container;
- *"container_network_receive_bytes_total"*: fornisce il numero totale di bytes ricevuti per ogni container.

Il server Prometheus è stato configurato per calcolare delle metriche aggregate personalizzate a partire da quelle estratte direttamente dai microservizi. L'implementazione è stata realizzata attraverso le *Prometheus Rules*.

Le rules specificate nel file di configurazione sono le seguenti:

- Numero di richieste da parte del client avvenute con successo, sia per il WMS sia per l'UM.
- Numero di richieste da parte del Worker avvenute con successo verso il servizio esterno *OpenWeather*.
- Rate di richieste avvenute nell'ultima ora.
- Percentuali di successo delle richieste pervenute sia al WMS sia all'UM sia a *OpenWeather*.
- La media, la mediana e il novantesimo percentile dei tempi di esecuzione delle query ai database, sia per il Worker, sia per il WMS, sia per l'UM, sia per il Notifier.

4.2) Forecasting

Il microservizio SLA Manager, come già precedentemente spiegato, espone un endpoint per il forecasting delle metriche, in modo da ottenere una previsione dei valori futuri per i prossimi minuti. È possibile specificare la metrica alla quale si è interessati e il numero di minuti come parametri della richiesta GET.

Gli step che sono stati seguiti per il forecasting sono i seguenti:

- Prelievo dei dati, relativi alla metrica, da Prometheus.
- Data Manipulation: viene effettuato il resample della serie temporale, per assegnare ai dati prelevati una frequenza pari a quella di scraping di Prometheus; successivamente viene eseguita un'interpolazione dei dati per assicurare che non ci siano valori nulli.
- Il modello scelto per la predizione dipende dalla metrica selezionata ed è a discrezione dell'admin. Le possibili scelte ricadono sui modelli *Holt's Method* (Double Exponential Smoothing) e *Holt-Winters Methods* (Triple Exponential Smoothing). In tal modo l'admin può scegliere quali metriche necessitano di un'analisi che tenga conto del trend e della stagionalità della serie (Holt-Winters Method) oppure quali solo del trend (Holt's Method).
- Splitting dei dati in training e test data sets, seguendo la regola 90/10.
- Adattamento del modello ai dati di training
- Forecasting effettivo.

A scopo illustrativo sono riportati degli esempi di predizione di alcune serie temporali.

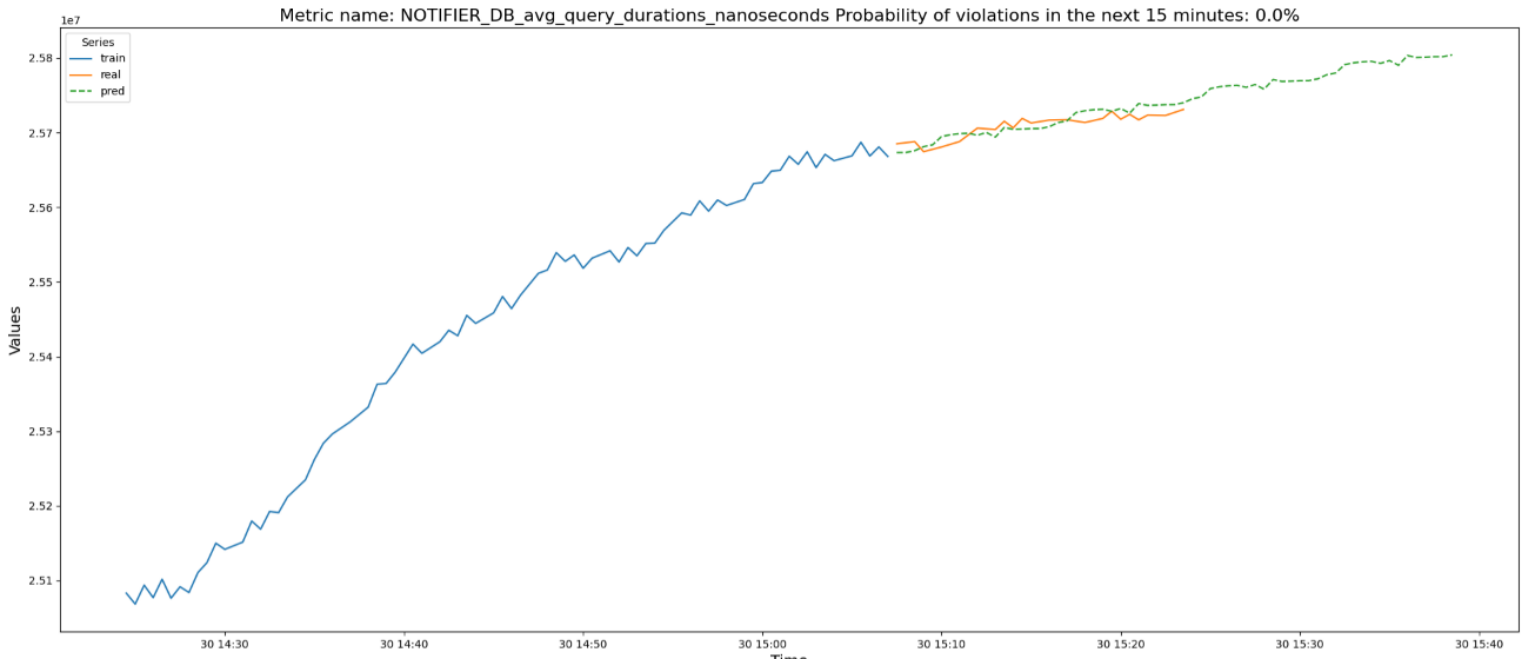

Figura 4 Esempio Forecasting Notifier_DB_avg_query_durations

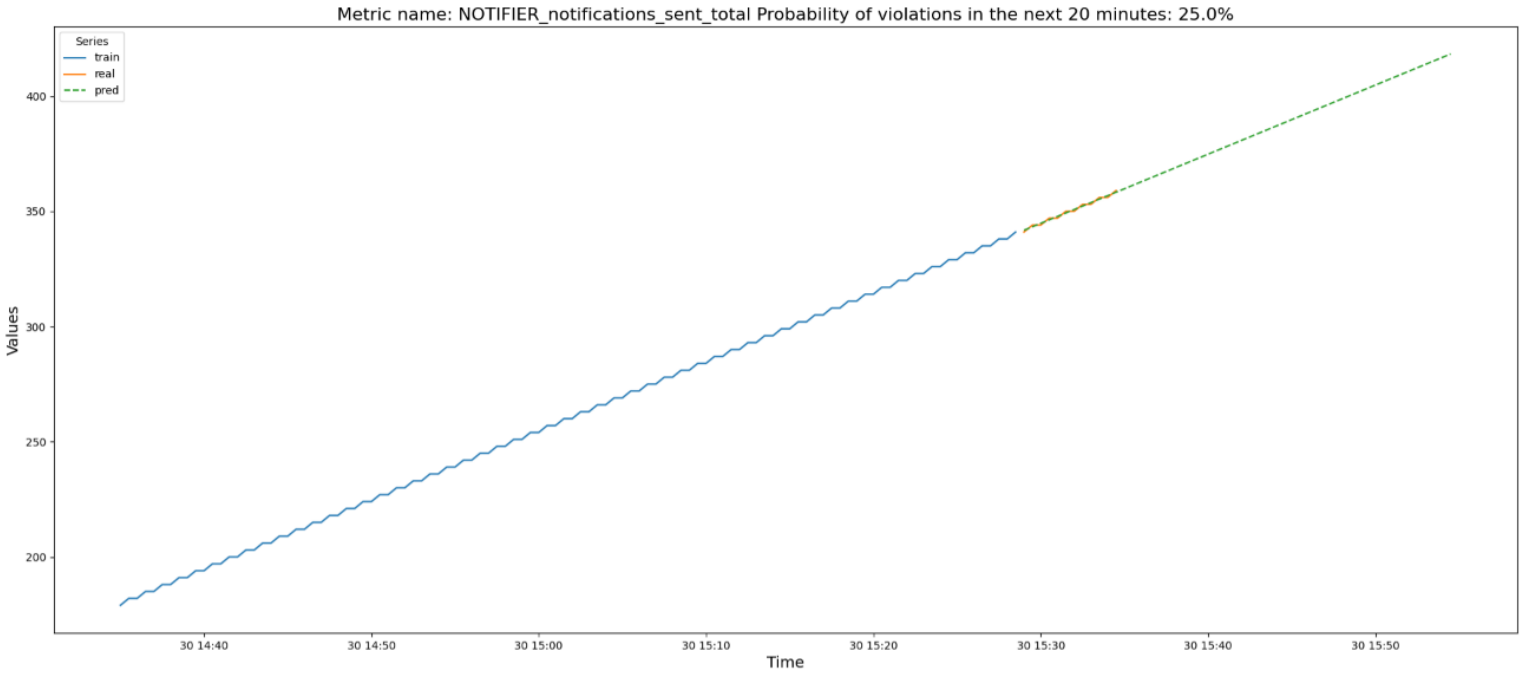

Figura 5 Esempio Forecasting Notifier_notifications_sent_total