Relazione progetto HW3 + Estensione

Gabriele Florio – Enricomaria Di Rosolini

Sommario

Descrizione

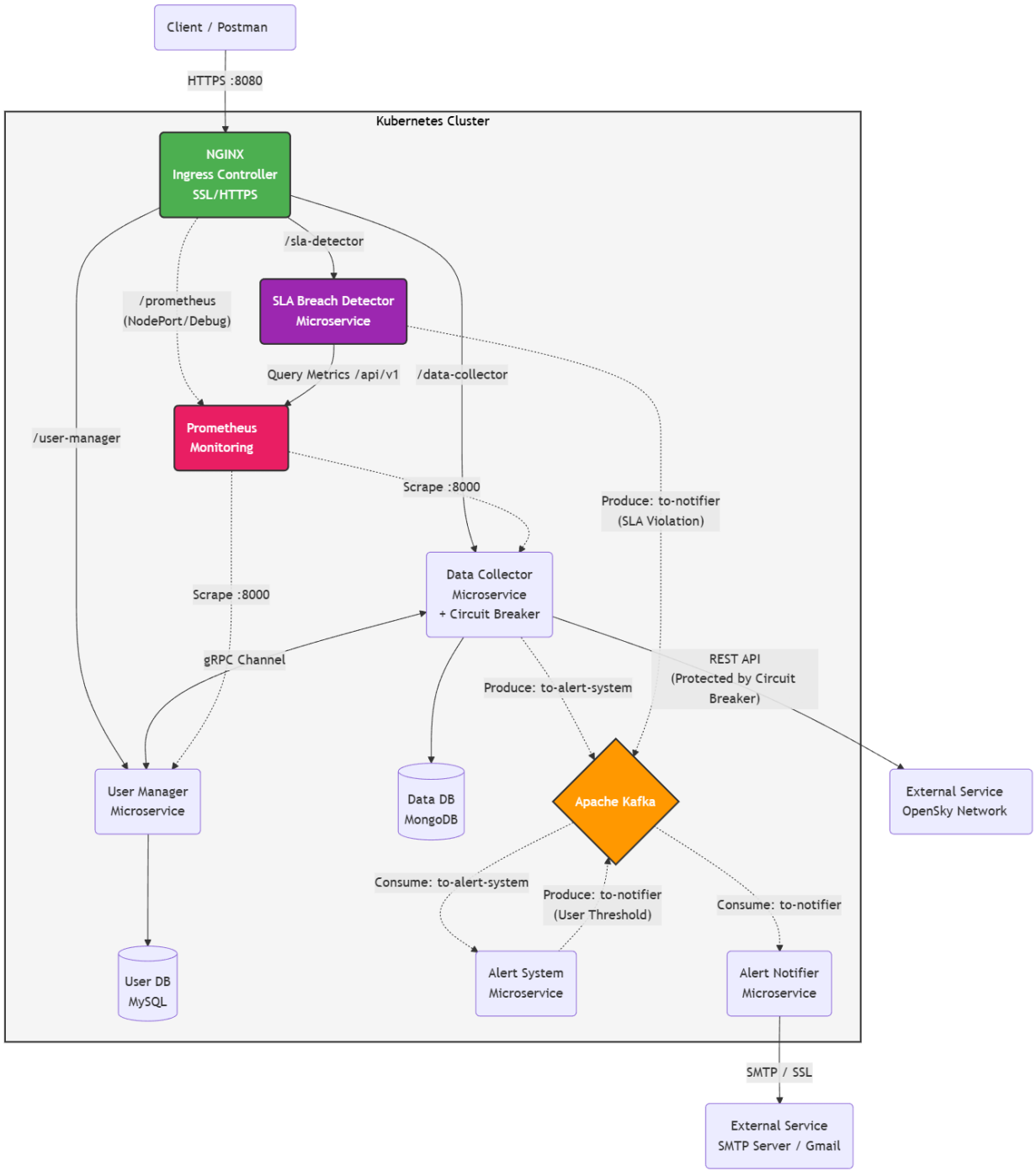
Il presente documento descrive l'evoluzione architetturale e funzionale del sistema distribuito per il monitoraggio del traffico aereo, sviluppato nelle fasi precedenti del corso.

Il sistema, originariamente progettato per raccogliere dati dalla *OpenSky Network* e gestire le sottoscrizioni degli utenti tramite un'architettura a microservizi, ha subito un significativo aggiornamento infrastrutturale. Dopo aver consolidato la comunicazione asincrona tramite Kafka e la resilienza tramite Circuit Breaker, il focus del progetto si è spostato su tre nuovi pilastri fondamentali per un sistema in produzione: **Orchestrazione**, **Osservabilità** e **Garanzia della Qualità del Servizio (QoS)**.

L'obiettivo di questa fase finale è stato duplice, coprendo sia i requisiti dell'Homework 3 che dell'Estensione:

- **Migrazione su Kubernetes:** È stata abbandonata l'orchestrazione locale basata su Docker Compose in favore di Kubernetes (utilizzando *Kind*). Questo passaggio ha permesso di introdurre concetti avanzati quali il *Self-Healing* dei pod, la gestione dichiarativa delle risorse (Deployment e StatefulSet) e un networking avanzato tramite Ingress Controller, simulando un reale ambiente di produzione.
- **Monitoraggio White-Box e SLA :** È stato implementato uno stack di osservabilità basato su **Prometheus** per la raccolta di metriche interne (White-box monitoring) dai microservizi. Parallelamente, è stato sviluppato un nuovo componente, lo **SLA Breach Detector**, incaricato di analizzare tali metriche in tempo reale e notificare tempestivamente eventuali violazioni degli accordi sul livello di servizio (SLA), chiudendo il ciclo di feedback sulla qualità del sistema.

Definizione dei componenti architetturali



L'architettura adottata segue il pattern a microservizi dockerizzati, dove ognuno di essi ha una responsabilità ben definita.

Il sistema è composto dai seguenti componenti principali:

- **User Manager:** Gestisce la registrazione e l'anagrafica utente. Espone metriche Prometheus per il monitoraggio.
- **Data Collector:** Il cuore operativo del sistema. Recupera periodicamente i dati di volo dalla *OpenSky Network* e gestisce gli interessi degli utenti. Integra un pattern **Circuit**

Breaker per proteggere il sistema da malfunzionamenti delle API esterne e comunica i nuovi dati via Kafka.

- **Message Broker (Kafka):** Disaccoppia la produzione dei dati dal sistema di allerta e notifica.
- **API Gateway (Ingress NGINX):** Agisce come *API Gateway* e punto di ingresso unico per il cluster. Gestisce il routing del traffico HTTP esterno (esposto sulla porta 8080 dell'host) verso i corretti servizi interni (/user-manager, /data-collector), garantendo l'astrazione della topologia di rete sottostante.
- **Monitoring Stack (Nuovo):**
    - **Prometheus:** Colleziona metriche dai microservizi tramite scraping periodico.
    - **SLA Breach Detector:** Microservizio dedicato alla Quality of Service. Interroga Prometheus per verificare il rispetto delle soglie SLA configurate. In caso di violazione, agisce come *Producer* Kafka inviando un evento di allerta.
- **Consumer/Producer Services:**
    - **Alert System:** Consumer e producer Kafka che analizza i dati di volo rispetto alle soglie personalizzate dagli utenti. Se una soglia viene superata, genera un messaggio e lo invia sul topic “to-notifier”
    - **Alert Notifier:** Componente terminale che consuma gli eventi di notifica (sia per voli che per violazioni SLA) e si interfaccia con il server SMTP esterno (Gmail) per l'invio delle e-mail agli utenti finali o agli amministratori.

Per rispondere ai requisiti di scalabilità, resilienza e gestione automatizzata del ciclo di vita delle applicazioni, l'intera infrastruttura è stata migrata su **Kubernetes**. Per l'ambiente di sviluppo locale è stato utilizzato **Kind** (Kubernetes in Docker), configurando un cluster con nodi predisposti per l'Ingress controller.

L'architettura dispiegata si basa sui seguenti oggetti nativi di Kubernetes, definiti tramite manifest YAML dichiarativi.

Microservizi e scelte progettuali

Comunicazione sincrona

Per quanto riguarda la comunicazione, come si evince dal diagramma è stato adottato un approccio ibrido:

- **REST API:** utilizzato per le iterazioni tra client e i microservizi, in quanto molto semplice e ottimizzato nell’esporre risorse web.
- **gRPC:** utilizzato per la comunicazione tra i microservizi, tra il data collector e lo user manager. La scelta del gRPC è dovuta al fatto che offre delle prestazioni migliori, con una bassa latenza. Una delle scelte progettuali più significative riguarda la gestione

della comunicazione gRPC. Sebbene il Data Collector debba interrogare lo User Manager per verificare l'esistenza degli utenti, il sistema è stato progettato implementando **due canali gRPC distinti**, rendendo di fatto lo User Manager un componente ibrido che agisce sia da Server che da Client a seconda del contesto.

- **Canale 1: Data Collector (Client) - User Manager (Server) – Check user:** questo è il canale standard richiesto dalle specifiche. Quando un utente richiede di tracciare un aeroporto, il Data Collector invoca una procedura remota sullo User Manager per validare l'esistenza dell'utente prima di salvare l'interesse.
- **Canale 2: User Manager (Client) - Data Collector (Server) - Gestione della Cancellazione:** è stato implementato un secondo canale inverso per gestire la cancellazione dell'utente. Quando viene invocata l'API DELETE /users, lo User Manager (che qui agisce da client gRPC) notifica immediatamente il Data Collector affinché rimuova tutti i dati associati a quell'utente. Questa scelta è stata preferita rispetto ad un approccio passivo (ogni qualvolta il database viene aggiornato venga controllato se effettivamente l'utente che ha mostrato interesse per uno o più aeroporti sia ancora registrato, in caso contrario vengono eliminati gli interessi) per avere una maggiore consistenza immediata tra i due microservizi, evitando chiamate esterne per utenti cancellati, disaccoppiando quindi la logica di business dai cicli periodici di aggiornamento.

Comunicazione asincrona

- **Kafka:** per la gestione delle notifiche di soglia è stato abbandonato il paradigma richiesta/risposta sincrono adottato per la comunicazione tra i due microservizi (user manager e data collector) in favore di un approccio Event-Driven.

User-Manager

Per il microservizio user manager sono state adottate le seguenti scelte progettuali:

1. **Politica At-Most-Once:** Il sistema garantisce che una richiesta di registrazione venga elaborata al massimo una volta, in caso di richieste duplicate o ritrasmissioni, il sistema non crea duplicati ma restituisce lo stato coerente della risorsa. Questo viene gestito tramite la memoria cache all'interno della quale vengono conservati gli id delle richieste soddisfatte, conservando, inoltre, l'esito e il timestamp dell'operazione.
2. **Database MySQL**: Per la persistenza dei dati dello User Manager è stato selezionato un database relazionale MySQL. La scelta è motivata dalla natura dei dati trattati:
    a. **Struttura definita**: Le informazioni utente (email, password, dati anagrafici) seguono uno schema rigido e predefinito.
    b. **Integrità referenziale e Vincoli**: MySQL permette di definire vincoli forti (come PRIMARY KEY sull'email) che impediscono la duplicazione dei record, offrendo un ulteriore livello di sicurezza per la politica at-most-once.

c. **Transazionalità (ACID)**: Garantisce che le operazioni di registrazione e cancellazione siano atomiche e consistenti.

Data-Collector

Per il microservizio data collector sono state adottare le seguenti scelte progettuali:

1. **Scheduler:** è stato implementato uno scheduler, quindi un processo parallelo che interroga ciclicamente con un intervallo di 12 ore OpenSky Network per gli aeroporti monitorati dagli utenti e salva i risultati all'interno del database.
2. **Database:** Per la persistenza dei dati si è optato per un database NoSQL, in particolare MongoDB per i seguenti motivi:
   a. **Tipo di informazioni:** i dati recuperati dalle API REST di OpenSky Network sono in formato JSON, quindi l'utilizzo di MongoDB permette di memorizzarli in maniera rapida e senza la necessità di effettuare delle operazioni di trasformazioni in tabelle relazioni.
   b. **Flessibilità:** a differenza di MySQL, MongoDB non avendo una struttura rigida permette di gestire qualunque tipo di variazione dei dati sui voli senza la necessità di modificare la struttura del database.
   c. **Velocità di scrittura:** poiché il data collector opera periodicamente e potrebbe generare una quantità di dati elevata, l'utilizzo di MongoDB facilita e accelera la scrittura all'interno del DB

   Nel microservizio user-manager serviva rigore, qui si predilige la flessibilità e la velocità di scrittura.

Sono state adottate le seguenti scelte progettuali a tal fine da rendere l'applicazione più scalabile e sicura.

- **Comunicazione Asincrona (Apache Kafka):** per la gestione delle notifiche di soglia è stato abbandonato il paradigma richiesta/risposta sincrono adottato per la comunicazione tra i due microservizi (user manager e data collector) in favore di un approccio Event-Driven. Il flusso è diviso su due topic distinti:
  - **Topic to-alert-system:** Il Data Collector agisce da Producer. Dopo aver aggiornato i dati dei voli, pubblica un messaggio contenente i nuovi dati e la lista degli utenti interessati. Il Data Collector invia nel messaggio Kafka sia i dati dei voli che i profili utente interessati per evitare che l'Alert System debba interrogare il database appartenente al microservizio data-collector.
  - **Topic to-notifier**: L'Alert System, agisce da Consumer e Producer, processa i dati e se rileva il superamento di una soglia, pubblica un evento di "notifica" su questo topic.
- **Circuit Breaker:** L'implementazione del Circuit Breaker non si limita a conteggiare gli errori generici, ma adotta una logica per distinguere tra malfunzionamenti transitori e risposte valide. Il componente gestisce tre stati distinti:

- **Closed:** Stato normale, le richieste passano OPEN**:** Dopo il superamento della soglia di fallimenti (failure_threshold=5), le richieste vengono bloccate preventivamente sollevando un'eccezione CircuitBreakerOpenException, riducendo il carico sul servizio esterno in difficoltà.
- **Half-open:** Trascorso il recovery_timeout (30), il sistema lascia passare una singola richiesta "sonda". Se questa ha successo, il circuito si chiude e il conteggio dei fallimenti viene resettato; altrimenti torna aperto.

Gestione Eccezioni (404 vs 500): È stata introdotta una distinzione fondamentale nella gestione degli errori HTTPS. Mentre gli errori server (5xx) o i timeout incrementano il contatore dei fallimenti, gli errori client come il 404 Not Found vengono ignorati dal Circuit Breaker. Questa scelta è dovuta da OpenSky Network che non trova voli in partenza o in arrivo per determinate fasce orarie restituendo un errore 404 Not Found, non strettamente legato alla mancanza di voli che effettivamente partono da un determinato aeroporto, ma bensì da problemi di OpenSky Network.

AlertSystem

Il microservizio AlertSystem è un servizio completamente indipendente e leggero che agisce sia da consumer che da producer, prendendo i dati dal topic “to-alert-system” e verificando se la somma del numero di voli in partenza e arrivo da un determinato aeroporto violino le soglie impostate dall’utente, in caso affermativo produce un messaggio contenente i parametri utente, aeroporto e condizione di superamento soglia, pubblicandolo sul topic “to-notifier”

AlertNotifierSystem

Il microservizio AlertNotifierSystem è un servizio completamente indipendente che agisce da consumer, prendendo i dati dal topic “to-notifier” e notificando l’utente del superamento della soglia da lui imposta tramite server SMTP (Gmail).

Monitoraggio con Prometheus

L'introduzione dell'orchestrazione su Kubernetes è stata affiancata dall'implementazione di un sistema di monitoraggio **White-box**. A differenza del monitoraggio black-box (che verifica solo se un servizio risponde), l'approccio white-box permette di ispezionare lo stato interno dei microservizi, esponendo metriche applicative e infrastrutturali precise.

Il cuore del sistema di monitoraggio è **Prometheus**, distribuito all'interno del cluster. Prometheus è configurato per effettuare lo scraping (raccolta dati) ogni 15 secondi dagli endpoint /metrics esposti dai microservizi User Manager e Data Collector sulla porta dedicata **8000**.

I microservizi User-Manager e Data-Collector sono stati strumentati utilizzando la libreria client di Prometheus, definendo metriche di due tipologie principali, arricchite con le label service e node per identificare univocamente la sorgente.

Monitoraggio User-Manager

Il microservizio User Manager espone un set di metriche volto a controllare sia le performance del database che la correttezza delle logiche di business (es. deduplicazione).

Database update duration

- **Query metrica**: user_manager_db_update_duration_seconds
- **Tipo:** Gauge
- **Descrizione:** traccia la durata in secondi di aggiornamento del database quando viene effettuata un'operazione.

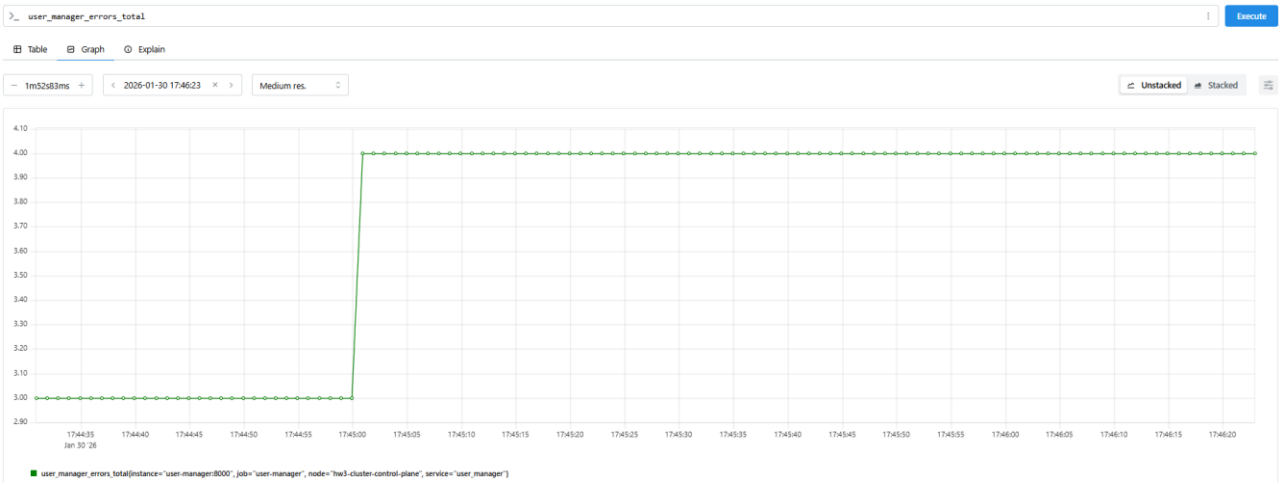
Total user

- **Query metrica**: total_user
- **Tipo:** Gauge
- **Descrizione:** traccia il numero dei nuovi utenti inseriti nel database, al netto di quelli eliminati.

Errori nello user manager

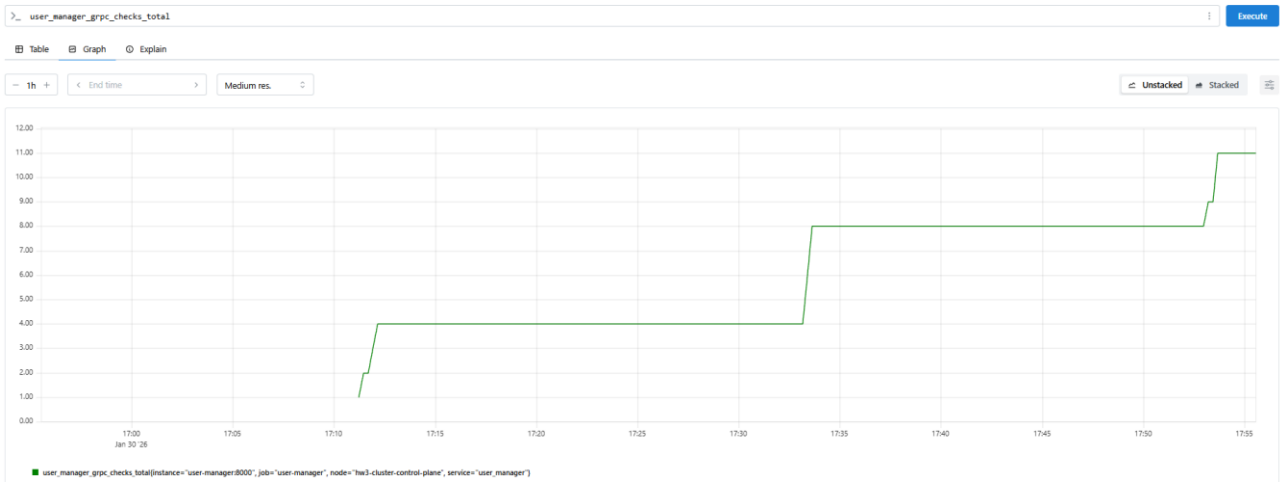
- **Query metrica**: user_manager_errors_total
- **Tipo:** Counter

- **Descrizione:** Monitora il numero di errori che si verificano durante le operazioni del microservizio.



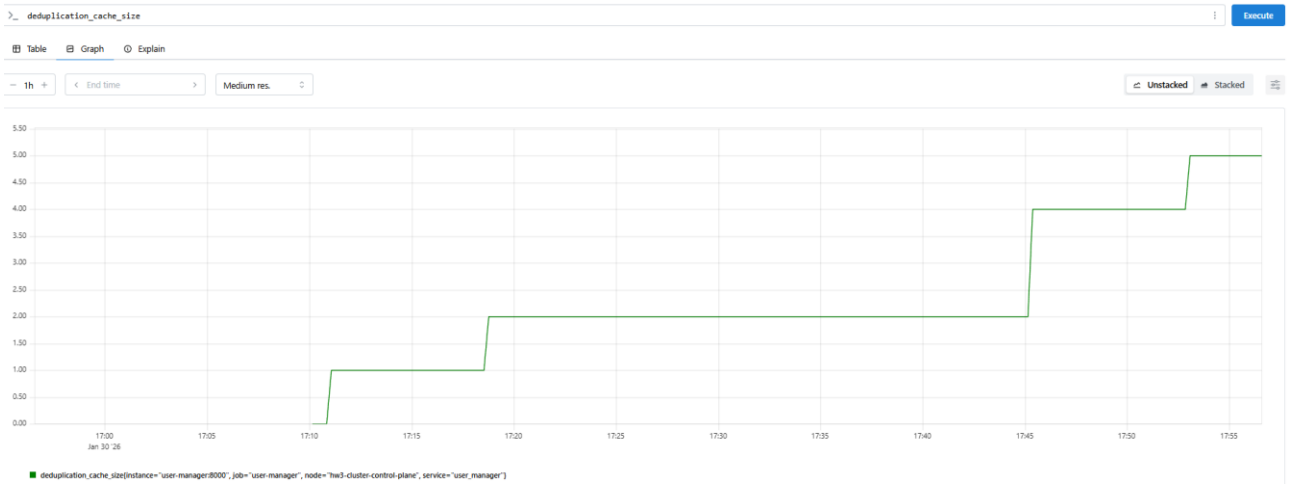
Numero di comunicazioni grpc effettuati

- **Query metrica**: user_manager_grpc_checks_total
- **Tipo:** Counter
- **Descrizione:** Traccia il numero totale di richieste gRPC ricevute per la verifica dell'esistenza utente. Utile per analizzare il carico di traffico interno.



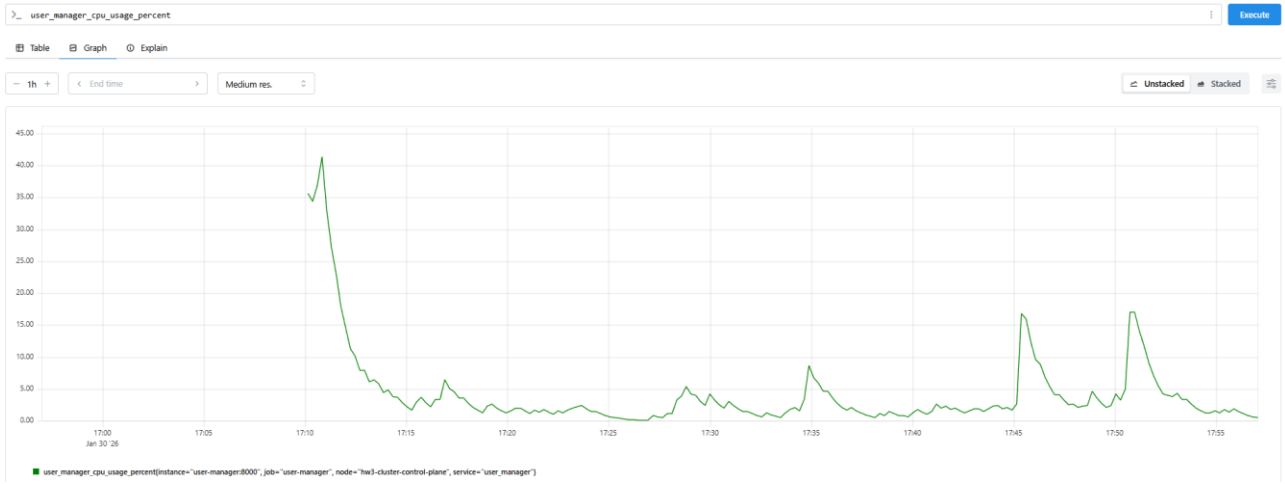
deduplication_cache_size

- **Query metrica**: deduplication_cache_size
- **Tipo:** Gauge
- **Descrizione:** Monitora in tempo reale il numero di request_id memorizzati nella cache per la politica At-Most-Once. Un aumento incontrollato di questo valore potrebbe indicare problemi di pulizia della memoria



user_manager_cpu_usage_percent

- **Query metrica**: user_manager_cpu_usage_percent
- **Tipo:** Gauge
- **Descrizione:** Misura l'utilizzo percentuale della CPU del container.

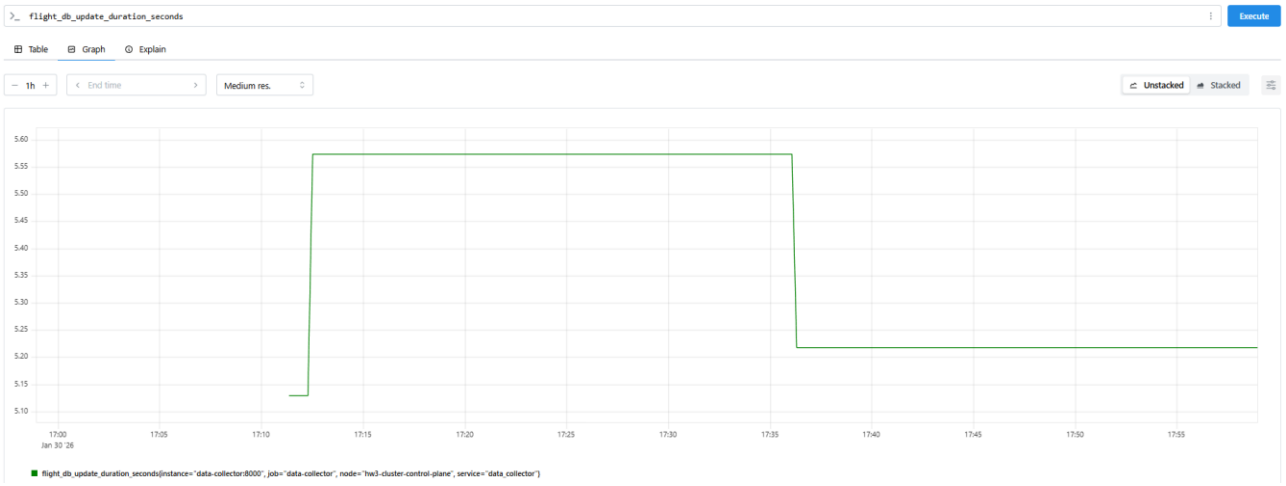


Monitoraggio Data-Collector

Dato il ruolo critico di questo componente nel recupero dati, le metriche si concentrano sulla salute dello scheduler e sull'interazione con i sistemi esterni (OpenSky, Kafka).

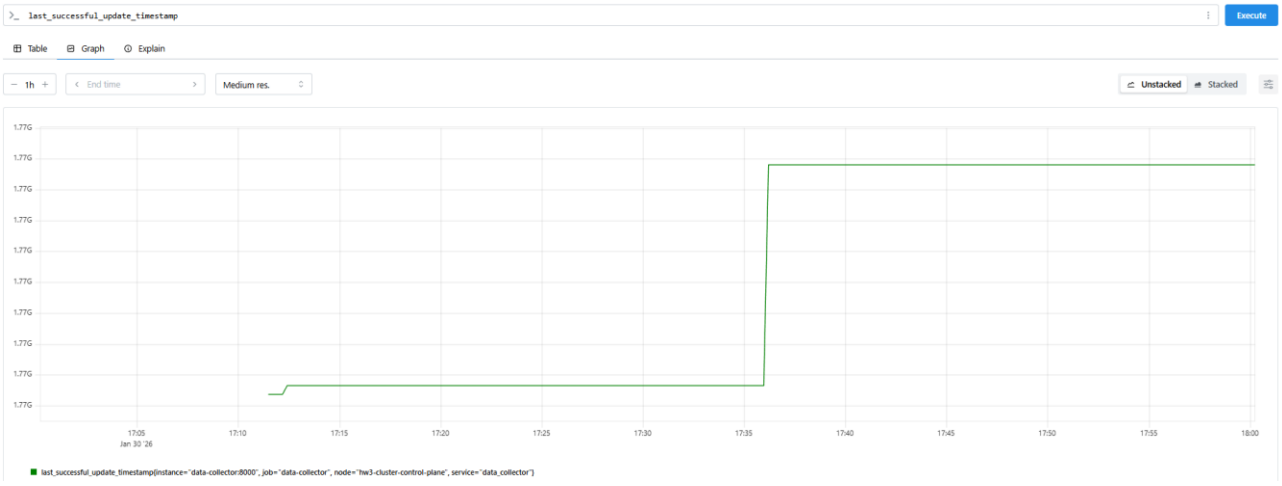
flight_db_update_duration_seconds

- **Query metrica**: flight_db_update_duration_seconds
- **Tipo:** Gauge
- **Descrizione:** Misura la durata dell'ultimo ciclo di aggiornamento dei dati di volo. Essenziale per rilevare latenze nel database o nelle API di OpenSky.



last_successful_update_timestamp

- **Query metrica**: last_successful_update_timestamp
- **Tipo:** Gauge
- **Descrizione:** Timestamp Unix dell'ultimo aggiornamento completato con successo. Questa metrica funge da Heartbeat: se il timestamp non avanza, lo scheduler si è bloccato.



total_flights_fetched

- **Query metrica**: total_flights_fetched
- **Tipo:** Counter
- **Descrizione:** contatore cumulativo dei voli scaricati.

Data_collector_cpu_usage_percent

- **Query metrica**: Data_collector_cpu_usage_percent
- **Tipo:** Gauge
- **Descrizione:** (**Gauge**): Misura l'utilizzo percentuale della CPU del container.

kafka_message_sent_total

- **Query metrica**: kafka_message_sent_total
- **Tipo:** Counter

- **Descrizione:** Monitora la quantità di messaggi kafka inviati al topic “to-notifier”.

Circuit_breaker_rejections_total

- **Query metrica**: kafka_message_sent_total
- **Tipo:** Counter
- **Descrizione:** Monitora la quantità di fallimenti delle chiamata a servizi esterni.

SLA Breach Detector

Per automatizzare il controllo della qualità del servizio, è stato sviluppato lo **SLA Breach Detector**, un microservizio indipendente progettato per rilevare violazioni degli accordi di servizio (SLA) basandosi sulle metriche raccolte da Prometheus.

Il servizio adotta un approccio configuration-first: all'avvio carica le regole di SLA definite in una **ConfigMap** Kubernetes dedicata. Questo design pattern disaccoppia la logica di business dalla configurazione, permettendo di modificare i parametri di soglia senza dover ricompilare il codice sorgente.

Il core del servizio esegue un ciclo di controllo periodico con un intervallo T_check, rispettando il vincolo progettuale T_check >= T_scape *5

L'algoritmo di rilevamento implementa una logica di stabilità per evitare falsi positivi dovuti a picchi transitori:

1. Il servizio interroga le API HTTP di Prometheus (/api/v1/query) per ottenere il valore corrente delle metriche monitorate.
2. Mantiene una memoria storica locale degli ultimi campioni.
3. Una violazione viene notificata **solo se** vengono rilevati almeno **3 campioni** che violano la soglia (valore < min oppure valore > max) all'interno della finestra di osservazione (finestra semi-dinamica).

Al rilevamento di una violazione confermata, lo SLA Breach Detector non invia direttamente notifiche, ma agisce come Producer Kafka pubblicando un messaggio sul topic to-notifier con tipo SLA_VIOLATION. Il payload include dettagli critici quali la metrica violata, il valore rilevato, le soglie di riferimento e il timestamp.

Il microservizio **Alert Notifier**, agendo da Consumer, intercetta questo evento specifico e invia un’e-mail di allerta all'amministratore di sistema, chiudendo il ciclo di feedback del monitoraggio.

Le metriche per le quali sono state definite dei SLA sono le seguenti:

- **Flight_db_update_duration_seconds**
- **Data_collector_cpu_usage_percent**
- **User_managaer_db_update_duration_seconds**

- **Total_user**
- **User_manager_cpu_usage_percent**
- **Deduplication_cache_size**
- **Time()-last_successful_update_timestamp**

Insieme a queste metriche sono stati definiti dei valori minimi e massimi che devono rispettare. Tali valori possono essere letti e modificati tramite le api esposte dal microservizio (update_sla, read_sla, breach_stats)

Deployment su kubernetes

Per soddisfare i requisiti di scalabilità, resilienza e gestione dichiarativa dell'infrastruttura, il sistema è stato migrato da un'orchestrazione locale basata su Docker Compose a **Kubernetes**. Per l'ambiente di sviluppo e test è stato utilizzato **Kind (Kubernetes in Docker)**, configurando un cluster locale predisposto per l'utilizzo di Ingress Controller tramite mappatura delle porte sull'host.

Configurazione Cluster

La creazione del cluster non è avvenuta con configurazioni di default, ma attraverso un manifesto personalizzato (kind-config.yaml) necessario per abilitare l'Ingress Controller.

- **Custom Node Configuration:** Al nodo del cluster è stato assegnato il ruolo control-plane e, tramite kubeadmConfigPatches, è stata applicata una configurazione specifica di kubelet per etichettare il nodo con ingress-ready=true. Questa label è fondamentale (Node Selector) per permettere al Pod dell'Ingress NGINX di essere schedulato sul nodo corretto.
- **Port Mapping Host-Container:** È stato definito un mapping esplicito tramite extraPortMappings per esporre la porta 80 del container Docker (che ospita il nodo Kubernetes) sulla porta 8080 della macchina host. Questo permette di accedere alle API REST dall'esterno del cluster (es. via Postman/Browser) bypassando la rete isolata di Kubernetes.

Gestione del traffico

L'accesso dall'esterno è centralizzato tramite una risorsa **Ingress** che agisce da Reverse Proxy intelligente (L7 Load Balancer), eliminando la necessità di esporre ogni singolo servizio tramite NodePort. Le regole di instradamento definite in k8s-ingress.yaml utilizzano il Path-based routing:

- Le richieste dirette a /user-manager vengono riscritte e inoltrate al servizio user-manager sulla porta 5000.
- Le richieste dirette a /data-collector vengono inoltrate al servizio data-collector.

L'annotazione nginx.ingress.kubernetes.io/rewrite-target: /$2 è fondamentale per rimuovere il prefisso del percorso (es. /user-manager) prima di inoltrare la richiesta al container, che si aspetta percorsi "puliti" (es. /add_user e non /user-manager/add_user).

Workload Stateless (Deployment)

I microservizi che non mantengono stato locale (User Manager, Data Collector, Alert System, Alert Notifier, SLA breach detector) sono stati definiti come risorse **Deployment**.

All'interno dei manifest, è stata posta particolare attenzione alla gestione delle risorse computazionali per garantire la stabilità del nodo (Quality of Service):

- **Resource Requests & Limits:** Ogni container dichiara esplicitamente le risorse necessarie.
    - Requests: La quantità minima garantita (es. cpu: "100m", memory: "128Mi" per User Manager). Kubernetes usa questo valore per lo scheduling.
    - Limits: Il tetto massimo utilizzabile (es. cpu: "500m", memory: "512Mi"). Se un container supera il limite di memoria, viene terminato (OOMKilled) per proteggere gli altri servizi.
- **Self-Healing:** L'uso dei Deployment garantisce che, in caso di crash dell'applicazione, il ReplicaSet sottostante ricrei automaticamente il Pod per mantenere il numero di repliche desiderato (replicas: 1).

Per soddisfare il requisito di monitoraggio che richiede di conoscere il nodo ospitante, è stata utilizzata la **Kubernetes Downward API**. Nel manifesto dei servizi (user-manager e data-collector), la variabile d'ambiente NODE_NAME viene popolata dinamicamente a runtime referenziando il campo spec.nodeName del Pod. Questo valore viene poi letto dal codice Python ed esposto nelle label delle metriche Prometheus.

Workload Stateful (StatefulSets)

Per i database (MySQL, MongoDB), la scelta architetturale è ricaduta sull'uso di **StatefulSet** invece dei Deployment. Questa primitiva è specifica per applicazioni che richiedono:

1. **Identità di Rete Stabile:** A differenza dei Pod dei Deployment (i cui nomi sono hash casuali), i Pod dello StatefulSet hanno nomi ordinali predicibili (es. mysql-0). Questo è cruciale per la stabilità delle connessioni interne e per eventuali configurazioni master-slave future.
2. **Persistenza dello Storage (PersistentVolumeClaim):** È stato definito un volumeClaimTemplate per ogni StatefulSet (accessModes: ["ReadWriteOnce"], storage: 1Gi).
    - Questo meccanismo esegue il provisioning dinamico di un PersistentVolume (PV).

- Se il Pod del database viene riavviato o spostato, il volume viene "smontato" dal vecchio Pod e "rimontato" su quello nuovo, garantendo che i dati degli utenti e dei voli non vadano persi (data durability).

Per il database MySQL, è stata sfruttata la flessibilità dei volumi per l'inizializzazione. Uno script SQL (init.sql) definito in una ConfigMap (db-init-script) viene montato come volume nella directory /docker-entrypoint-initdb.d del container. All'avvio, l'immagine MySQL esegue automaticamente questo script creando il database user_db e la tabella users.

Il deployment di **Kafka** in Kubernetes presenta complessità specifiche legate all'indirizzamento. Nel manifest k8s-infra.yaml, è stata configurata la variabile KAFKA_ADVERTISED_LISTENERS con valore PLAINTEXT://kafka-service:9092. Questo istruisce il broker a pubblicizzare il proprio indirizzo DNS interno ai client (producer/consumer), permettendo loro di connettersi correttamente all'interno della rete overlay del cluster. Inoltre, il cluster Kafka è configurato in modalità **KRaft** (senza ZooKeeper).

Configurazione

L'architettura separa rigorosamente il codice dalla configurazione:

- **ConfigMaps (shared-config):** Utilizzate per parametri non sensibili. Le variabili come DB_HOST, DB_NAME e KAFKA_BOOTSTRAP_SERVERS sono centralizzate in un unico oggetto Kubernetes e iniettate nei container come variabili d'ambiente. Questo permette di cambiare l'indirizzamento dei servizi senza ricostruire le immagini Docker.
- **Secrets (shared-secrets):** Dati critici come MYSQL_ROOT_PASSWORD, OPENSKY_PASSWORD e le credenziali SMTP sono memorizzati in oggetti Secret codificati in Base64. Questi vengono montati a runtime come variabili d'ambiente, garantendo che le password non appaiano in chiaro nei log o nelle descrizioni dei Pod.

Networking e Service Discovery

La comunicazione tra i microservizi non avviene più tramite link diretti o reti Docker bridge, ma sfrutta il meccanismo di **Service Discovery** interno di Kubernetes basato su DNS.

- **ClusterIP Services:** Sono stati configurati servizi di tipo ClusterIP per esporre i pod internamente al cluster. Ad esempio, il microservizio Data Collector comunica con User Manager risolvendo il nome DNS user-manager sulle porte 50051 (gRPC) e 5000 (HTTP).
- **NodePort Services:** Per permettere l'accesso visuale alla dashboard di monitoraggio dall'host locale, il servizio prometheus è stato esposto tramite NodePort.

Osservabilità e monitoraggio

Il monitoraggio è integrato nativamente nell'infrastruttura.

- **Deployment Prometheus:** Un'istanza di Prometheus è deployata nel cluster. La configurazione di scraping è iniettata tramite una ConfigMap (prometheus-config) che definisce i job statici per user-manager e data-collector sulla porta 8000.
- **Esposizione Dashboard:** A differenza di un approccio basato su NodePort, Prometheus è stato integrato dietro l'**Ingress Controller**. È stata definita una regola di instradamento specifica (/prometheus) nel file k8s-ingress.yaml.

Lista API

User manager

Tutte le API rivolte al microservizio User Manager dovranno adottare questo tipo di url:

- **http://localhost:8080/user-manager/ENDPOINT**

Metodo	Endpoint	Descrizione	Body (JSON)
POST	/add_user	Registrazione nuovo utente	{ "request_id": ..., "email" : "...", "name" : "...", "surname": "...", "age" : ..., "CF" : "...", "phone": "..." }
POST	/rmv_user	Rimuovi utente	{ "email" : "..." }
POST	/get_user	Ottieni informazioni utente	{ "email" : "..." }

Data-collector

Tutte le API rivolte al microservizio Data collector dovranno adottare questo tipo di url:

- **http://localhost:8080/data-collector/ENDPOINT**

Metodo	Endpoint	Descrizione	Body (JSON)
POST	/add_interest	Aggiunge un nuovo interesse	{"email" : "...", "airport_code": "..." "lowValue" : ..., "highValue": ...}
POST	/rmv_interest	Rimuove un interesse	{"email" : "...", "airport_code": "..." }
POST	/list_interest	Mostra la lista degli interessi dell'ut.	{"email" : "..." }
POST	/get_flight	Ottiene i voli in arrivo e in partenza dall'aeroporto	{"email" : "...", "airport_code": "..." }
POST	/force_update	Forza l'aggiornamento del db	//
POST	/get_last_flight	Ottiene l'ultimo volo in arrivo e in partenza dall'aeroporto	{"email" : "...", "airport_code": "..." }
POST	/average	Calcola la media voli degli ultimi x giorni	{"email" : "...", "airport_code": "..."

			“days”: ...}
POST	/get_arrivals	Ottiene i voli in arrivo che partono da un determinato aeroporto	{"email" : "...", "airport_code_arrivals":"...", "airport_code_departures":"..." }

SLA-Detector

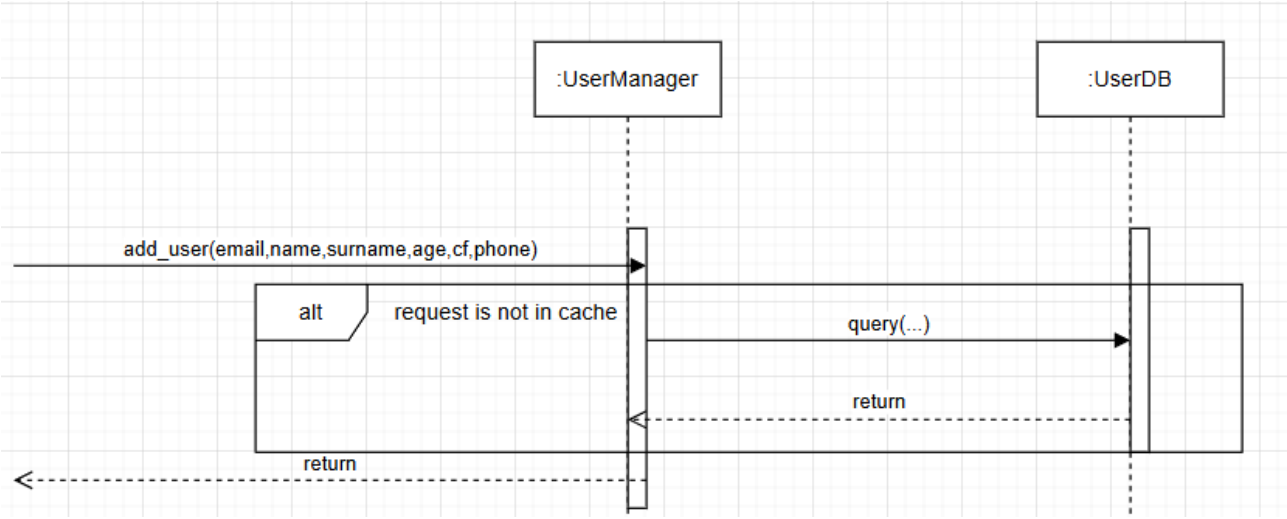
Tutte le API rivolte al microservizio Data collector dovranno adottare questo tipo di url:

- **http://localhost:8080/sla-detector /ENDPOINT**

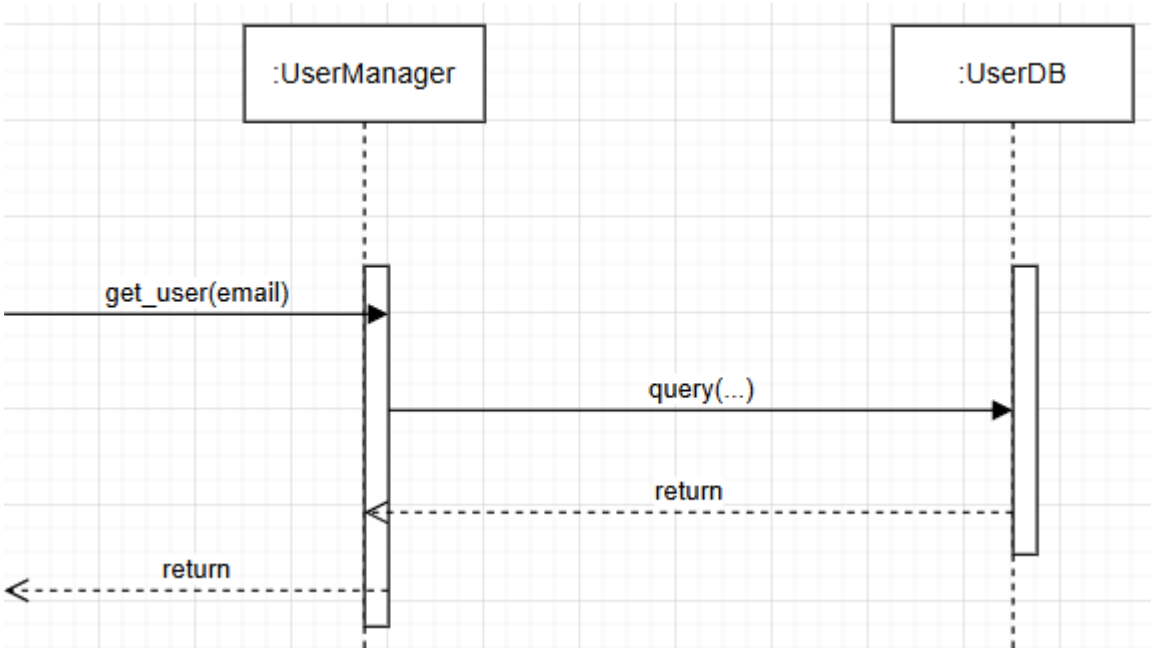
Metodo	Endpoint	Descrizione	Body (JSON)
POST	/update_sla	Aggiorna i valori del SLA di una determinata metrica	{ "min": ... , "max" : ... , "metric" : "...", "query" : "... " }
POST	/read_sla	Legge i parametri del SLA di una metrica	{ "metric" : "..." }
POST	/breach_stats	Legge quali metriche hanno avuto breach e quanti dall’avvio del servizio	{ }

DIAGRAMMA DELLE INTERAZIONI

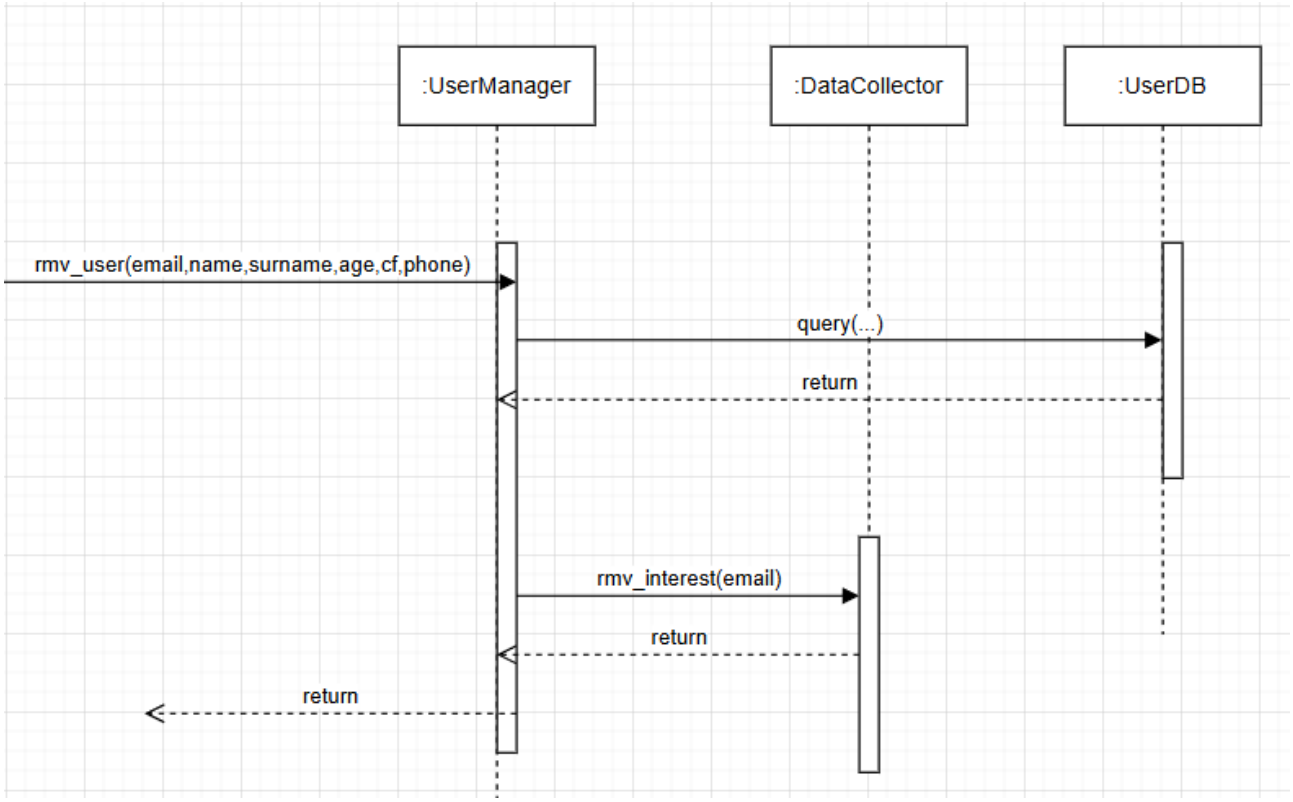
Add_user



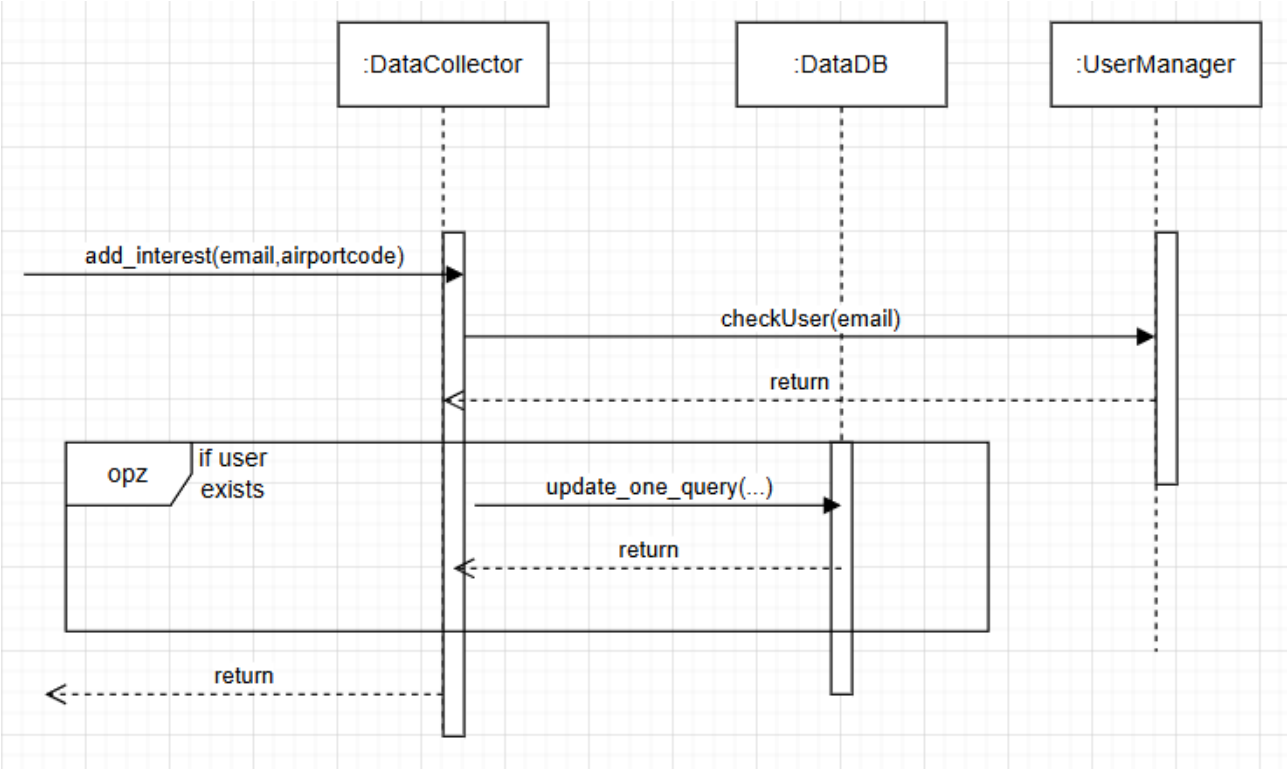
Get_user



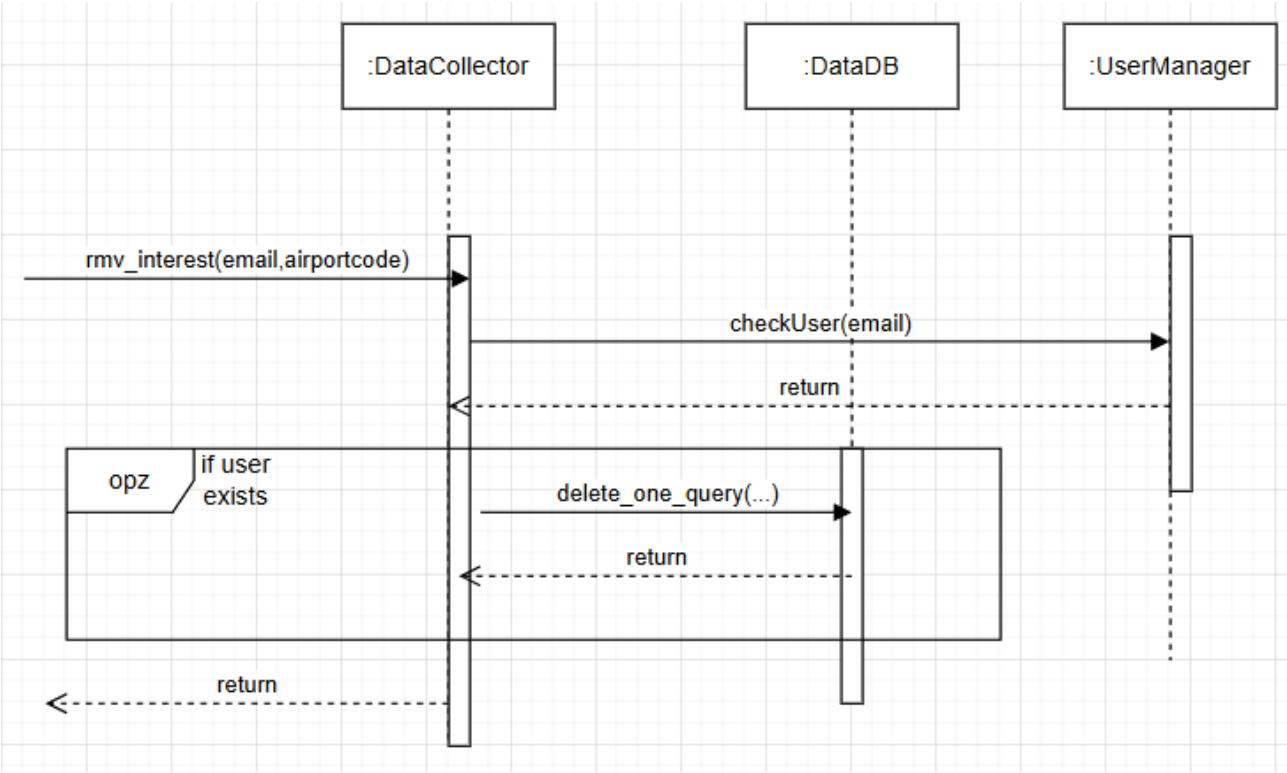
Rmv_user



Add_interest



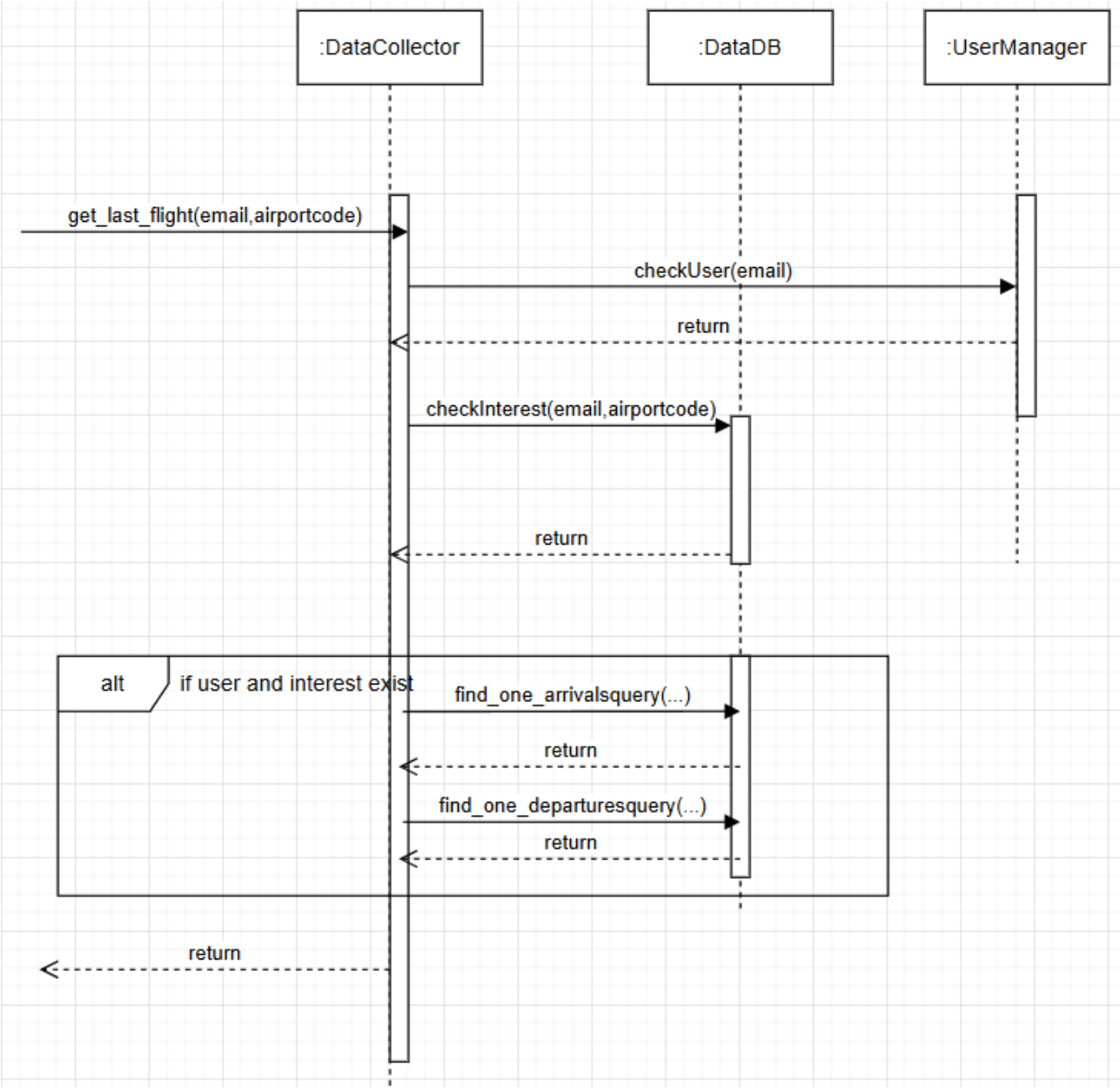
Rmv_interest



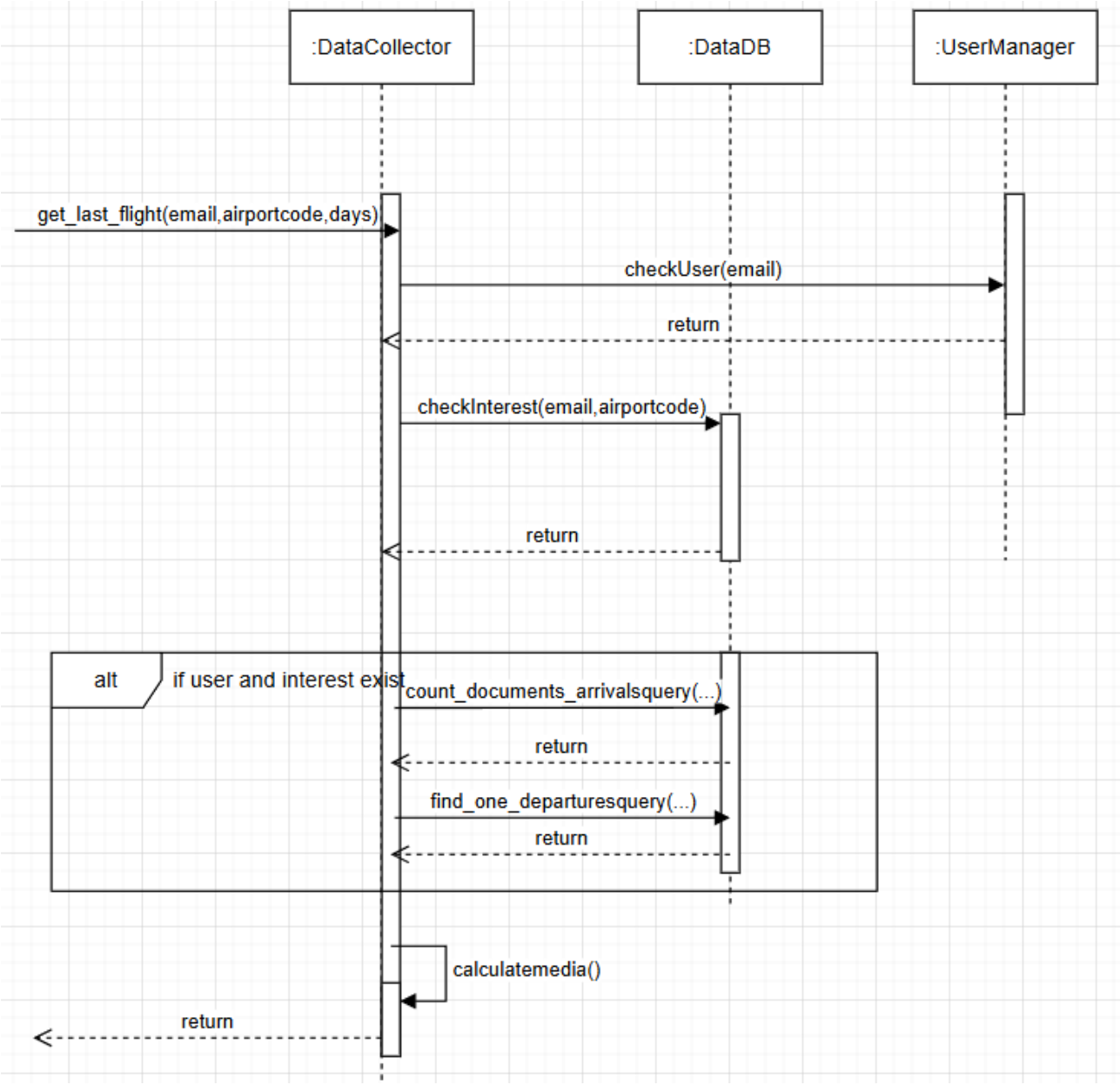
List_interest

Diagramma simile ai precedenti. Cambia solo la query con il db.

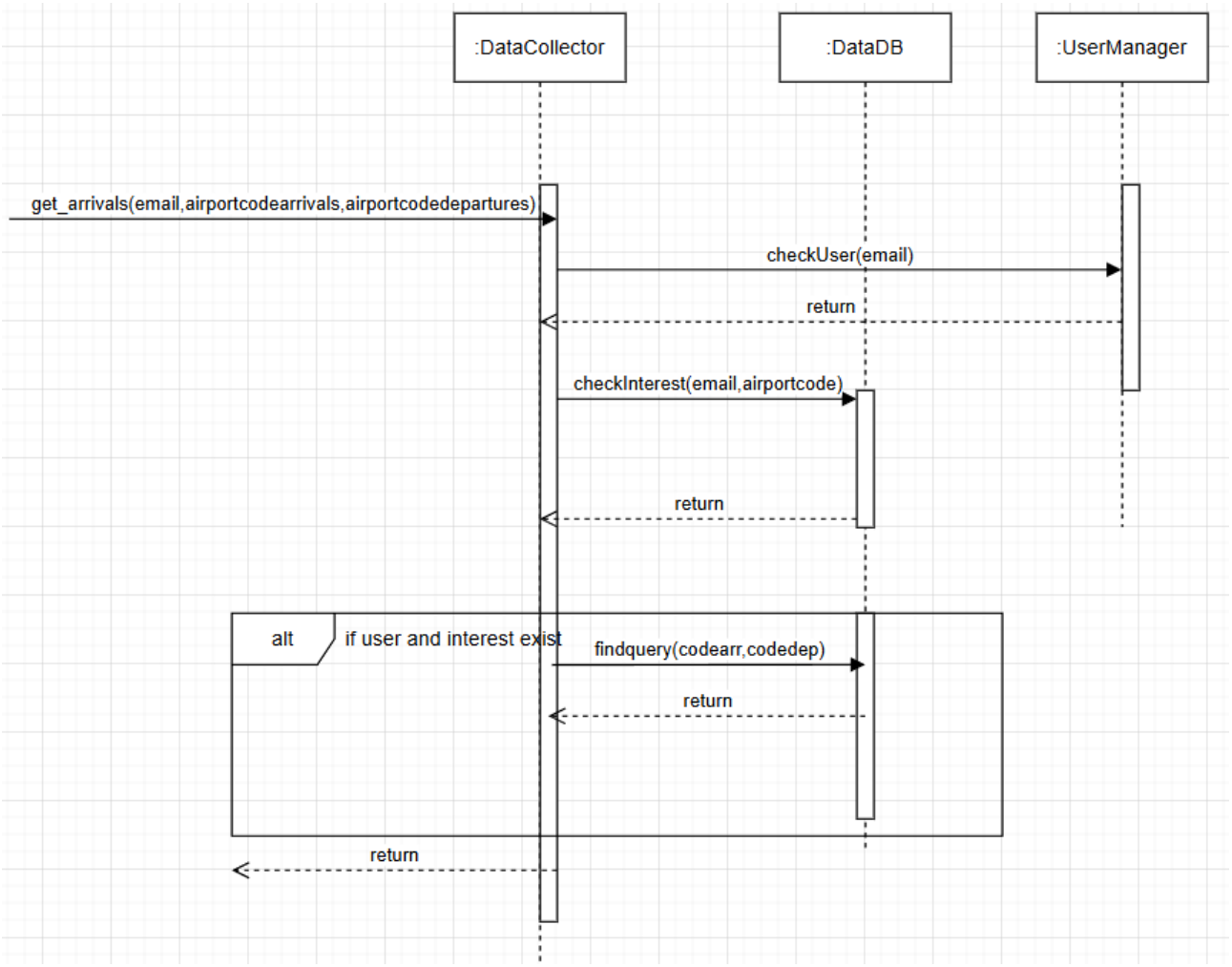
Get_last_flight



Get_average_flights



Get_arrivals



Get_flight

Diagramma delle interazioni simile ai precedenti

Update_flight_data

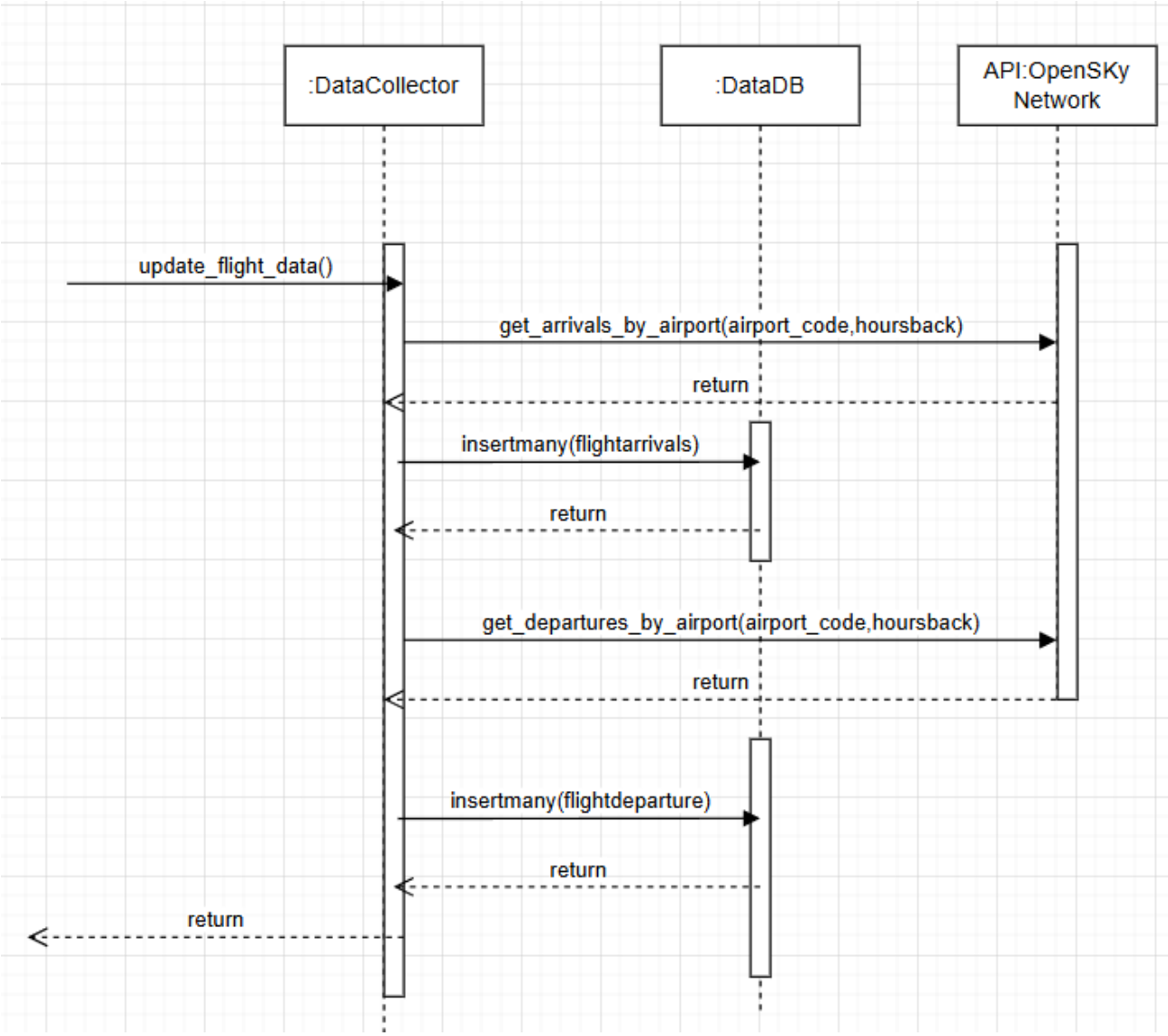


Diagramma delle interazioni di invio email

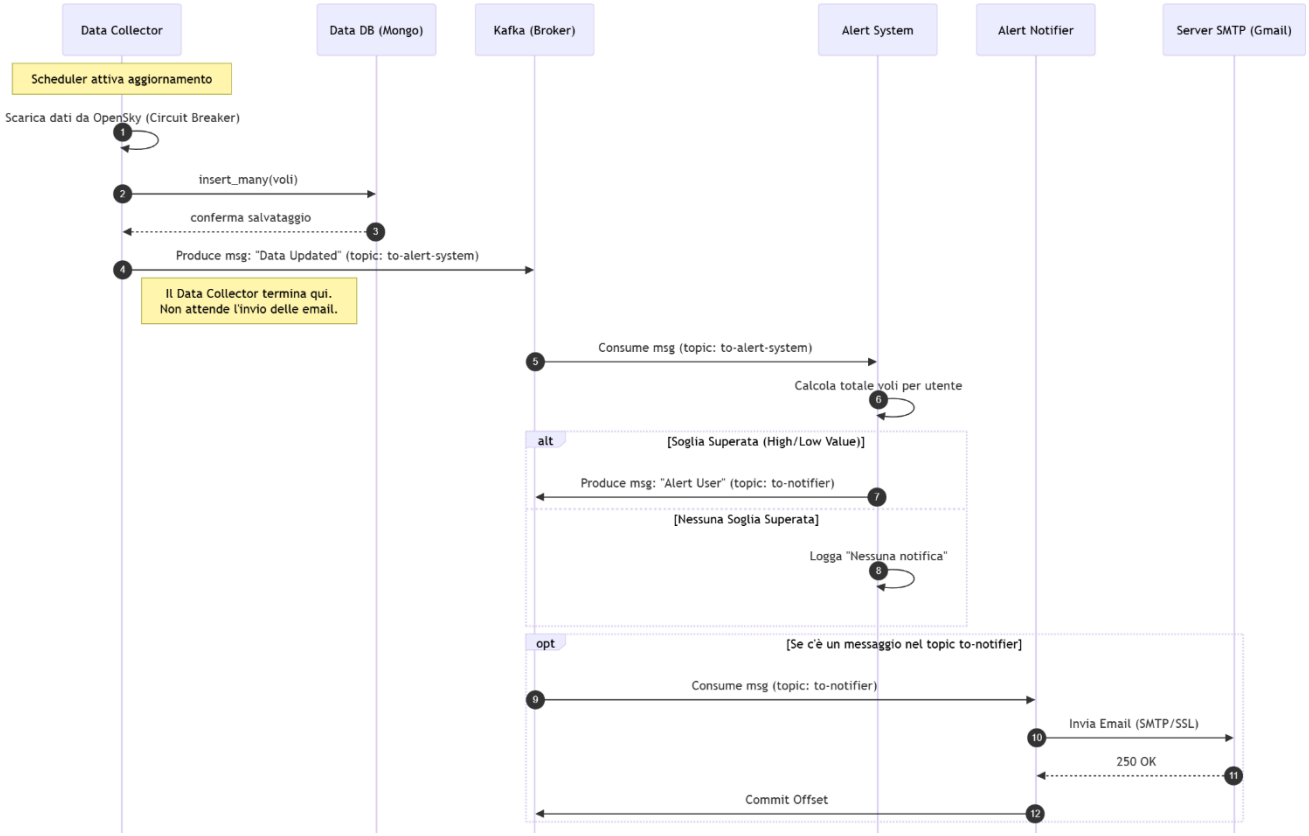
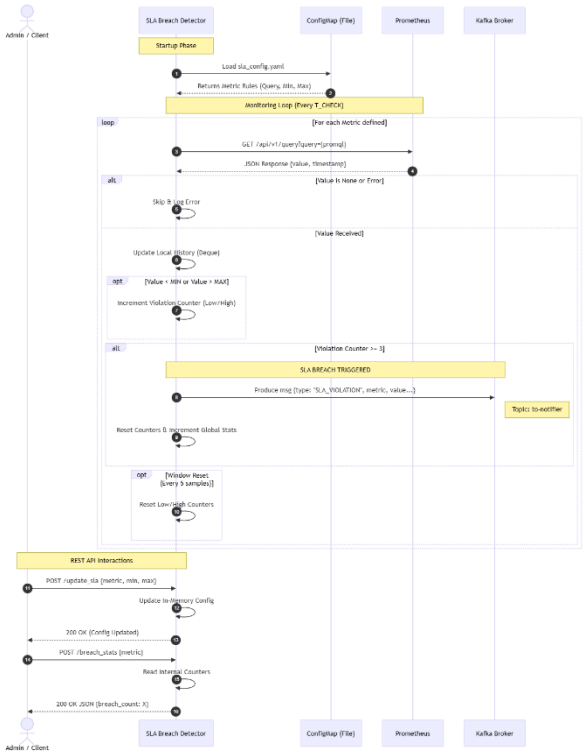


Diagramma delle interazioni di sla-detector



Conclusioni

Il progetto ha portato con successo l'architettura di monitoraggio voli da un ambiente di sviluppo basato su container isolati a un'infrastruttura orchestrata e osservabile, simulando uno scenario di produzione reale.

La migrazione su **Kubernetes** ha permesso di risolvere le criticità legate alla gestione manuale del ciclo di vita dei container. L'utilizzo di primitive native come *Deployment*, *StatefulSet* e *Ingress* ha garantito scalabilità, persistenza dei dati e una gestione centralizzata del traffico, eliminando la complessità della configurazione di rete punto-punto. Parallelamente, l'implementazione dello stack di monitoraggio **White-box** ha trasformato il sistema da una "scatola nera" a un'entità trasparente. L'integrazione di **Prometheus** e lo sviluppo del microservizio custom **SLA Breach Detector** hanno chiuso il ciclo di controllo, permettendo non solo di raccogliere dati, ma di reagire proattivamente alle degradazioni delle performance.

Lo sviluppo del sistema in ambiente Kubernetes ha evidenziato diverse sfide architetturali tipiche dei sistemi distribuiti. Le principali decisioni progettuali e le relative considerazioni tecniche sono riassunte di seguito:

- **Raffinamento della Resilienza (Smart Circuit Breaking):** L'implementazione del *Circuit Breaker* ha richiesto un approccio non banale alla gestione degli errori. È stato necessario implementare una logica di "filtraggio semantico" per distinguere i **fallimenti infrastrutturali** (es. timeout, HTTP 5xx) dagli **stati applicativi validi** (es. HTTP 404 da OpenSky). Trattare un "dato non trovato" come un guasto avrebbe compromesso la disponibilità del servizio; questa distinzione ha permesso di mantenere il circuito chiuso (operativo) anche in presenza di buchi nei dati di volo, isolando solo i reali disservizi di rete.
- **Trade-off tra Consistenza e Disaccoppiamento:** L'adozione di un'architettura ibrida (Sincrona/Asincrona) è stata dettata dalla necessità di bilanciare requisiti contrastanti:
    - Per la validazione degli utenti, è stata prioritaria la **Strong Consistency**: l'uso di gRPC garantisce che il Data Collector non operi mai su utenti inesistenti, accettando un accoppiamento temporale più stretto.
    - Per la pipeline di allerta, si è privilegiata la **Scalabilità**: l'uso di Kafka disaccoppia il produttore (Data Collector) dal consumatore (Alert System), permettendo al sistema di assorbire picchi di carico (backpressure) senza bloccare il ciclo di aggiornamento dei voli.
- **Affidabilità dell'Osservabilità:** Durante lo sviluppo dello *SLA Breach Detector*, è emerso come le metriche grezze (raw metrics) siano spesso soggette a volatilità. Per evitare il fenomeno dell'*alert fatigue* (sovraesposizione a falsi positivi), è stato implementato un algoritmo basato su finestre di osservazione. La richiesta di 3

campioni consecutivi fuori soglia funge da filtro passa-basso, trasformando semplici picchi transitori in segnali di allarme affidabili.

- **Gestione dello Stato in Kubernetes (StatefulSets vs Deployments):** La migrazione da Docker Compose a Kubernetes ha reso critica la gestione della persistenza. A differenza dei microservizi stateless, i database (MySQL, Mongo) hanno richiesto l'uso di **StatefulSets** per garantire identità di rete stabili (mysql-0) e l'associazione deterministica con i *PersistentVolumeClaims*. Questo ha evidenziato la complessità intrinseca nel mantenere lo stato in un ambiente orchestrato ed effimero.
- **Modernizzazione dell'Infrastruttura Kafka:** La scelta di configurare Kafka in modalità **KRaft** (rimuovendo la dipendenza da ZooKeeper) ha semplificato l'architettura, riducendo l'overhead di risorse nel cluster locale. Tuttavia, la configurazione del networking (tramite *Advertised Listeners*) si è rivelata cruciale per permettere la corretta comunicazione tra i pod producer/consumer e il broker all'interno della rete overlay di Kubernetes.