DIPARTIMENTO DI INGEGNERIA ELETTRICA, ELETTRONICA E INFORMATICA

CORSO DI LAUREA MAGISTRALE IN: INGEGNERIA INFORMATICA

DISTRIBUTED SYSTEMS AND BIG DATA

Homework 2

Documentazione

STUDENTI:
STEFANO CARAMAGNO
FEDERICO CALABRESE

DOCENTI:
PROF.SSA ANTONELLA DI STEFANO
PROF. GIOVANNI MORANA

ANNO ACCADEMICO 2025-2026

SOMMARIO

1. INTRODUZIONE E PERIMENTO DEL DOCUMENTO

1.1. OBIETTIVI DELLA VERSIONE CORRENTE DEL SISTEMA DI FLIGHT MONITORING

La versione corrente del sistema di *flight monitoring* ha l'obiettivo principale di evolvere un'architettura già esistente, orientata alla raccolta e interrogazione dei dati di volo, verso una piattaforma capace di gestire in modo **proattivo** le condizioni di traffico aereo di interesse per gli utenti, introducendo meccanismi di **notifica asincrona** e di **resilienza avanzata** verso servizi esterni potenzialmente instabili.

In particolare, gli obiettivi funzionali e architetturali della versione corrente possono essere sintetizzati nei seguenti punti:

- **Gestione di soglie personalizzate per aeroporto**: estendere il modello dei dati e i servizi applicativi in modo che ogni utente possa associare a ciascun aeroporto di interesse una coppia di soglie (*high_value* e *low_value*), parametrizzate sul numero di voli registrati in una determinata finestra temporale.
- **Introduzione di una pipeline di notifica asincrona**: decouplare la fase di raccolta e aggregazione dei dati di volo dalla fase di valutazione delle soglie e dalla generazione delle notifiche, adottando un **message broker** (Kafka) per veicolare eventi tra microservizi distinti, con responsabilità ben separate (raccolta dati, valutazione soglie, invio email).
- **Incremento della resilienza verso il servizio esterno OpenSky**: proteggere le chiamate HTTP verso l'API OpenSky mediante un **Circuit Breaker** basato su Resilience4j, in grado di:
    - limitare l'impatto di errori ripetuti o instabilità del servizio esterno;
    - applicare strategie di *fallback* controllate, restituendo risultati coerenti con il dominio applicativo anche in presenza di indisponibilità temporanee.
- **Esposizione unificata delle API tramite API Gateway**: introdurre un layer di **API Gateway** (basato su NGINX) per convogliare il traffico in ingresso verso i microservizi interni, standardizzando i punti d'accesso, semplificando la configurazione dei client e predisponendo il sistema a futuri requisiti di osservabilità e sicurezza a livello di perimetro.
- **Mantenimento della compatibilità funzionale con la versione precedente**: preservare le funzionalità di base già disponibili (registrazione utenti, registrazione interessi sugli aeroporti, raccolta periodica dei voli, interrogazione dei dati di volo), integrandole con le nuove capacità senza introdurre regressioni o modifiche incompatibili ai contratti esistenti.

La versione corrente non si limita, quindi, ad aggiungere nuove funzionalità, ma mira a consolidare il sistema come **piattaforma estensibile**, predisposta a ulteriori evoluzioni, mantenendo un equilibrio tra complessità architetturale, manutenibilità e chiarezza dei flussi applicativi.

1.2. PERIMETRO DELLA DOCUMENTAZIONE E RELAZIONE CON LA VERSIONE PRECEDENTE

Il presente documento descrive lo stato del sistema nella **versione corrente**, con particolare attenzione alle **estensioni architetturali e funzionali** introdotte rispetto alla versione precedente. L'obiettivo non è riprodurre integralmente la documentazione di base, ma fornire una vista focalizzata sulle componenti e sui flussi che sono stati aggiunti, modificati o specializzati.

Nel perimetro di questa documentazione rientrano in modo esplicito:

- la descrizione dell'**architettura estesa**, con l'introduzione dei microservizi *Alert System* e *Alert Notifier*, del **Kafka Broker** e dell'**API Gateway**;
- l'analisi dell'**evoluzione del modello dei dati**, in particolare l'estensione della tabella degli interessi utente sugli aeroporti con gli attributi di soglia (*high_value* e *low_value*) e l'impatto sui flussi di raccolta e aggregazione dei voli;
- la formalizzazione dei **nuovi flussi applicativi**:
    - configurazione e aggiornamento delle soglie di interesse,
    - pipeline asincrona di notifica (Data Collector → Alert System → Alert Notifier → Email),
    - gestione delle failure verso OpenSky tramite Circuit Breaker;
- la **lista delle API esposte** e dei **contratti di servizio** rilevanti per la versione corrente, includendo tanto gli endpoint REST quanto i messaggi scambiati su Kafka;
- la discussione delle **scelte progettuali** legate all'introduzione di broker di messaggistica, Circuit Breaker, API Gateway e soglie di interesse, con una valutazione del loro impatto su resilienza, scalabilità e manutenibilità;
- gli **aspetti non funzionali e operativi** collegati alle nuove componenti, con particolare riferimento a logging, tracciamento, gestione degli errori e possibili colli di bottiglia.

Sono invece considerati **fuori dal perimetro** di questo documento, se non per eventuali richiami puntuali:

- la descrizione di dettaglio del dominio applicativo e delle funzionalità di base (registrazione utenti, registrazione interessi senza soglie, raccolta periodica e interrogazione dei voli) già illustrata nella documentazione della **versione precedente**;
- la descrizione approfondita di aspetti infrastrutturali generali non modificati rispetto alla versione precedente (ad esempio, configurazioni di base di Docker, dettagli delle immagini utilizzate, aspetti sistemistici non direttamente influenzati dalle nuove estensioni);

- la trattazione di eventuali interfacce client esterne non specificamente rilevanti per le nuove funzionalità (ad esempio, eventuali front-end grafici o strumenti esterni di analisi che consumano le API REST esposte).

Quando necessario, il documento farà esplicito riferimento alla **documentazione della versione precedente** per rinviare a descrizioni già consolidate, evitando duplicazioni e mantenendo una chiara separazione tra:

- **stato di base** del sistema (versione precedente);
- **estensioni introdotte** nella versione corrente.

1.3. CONTESTO TECNOLOGICO E VINCOLI PROGETTUALI

La versione corrente del sistema si inserisce in un contesto tecnologico caratterizzato da un'architettura a **microservizi** eseguiti in un ambiente containerizzato orchestrato tramite **Docker Compose**. Ogni microservizio è implementato in **Java** utilizzando il framework **Spring Boot**, con un'adozione pervasiva di componenti della famiglia Spring per la realizzazione di REST API, integrazione con database relazionali, schedulazione di job periodici e interazione con sistemi esterni.

La persistenza dei dati è affidata a un'istanza di **PostgreSQL**, che ospita due database logici distinti:

- **User DB**, responsabile della gestione degli utenti registrati;
- **Data DB**, deputato alla gestione:
    - degli aeroporti monitorati,
    - degli interessi utente (ora arricchiti con le soglie *high_value* e *low_value*),
    - dei dati di volo raccolti periodicamente dall'API OpenSky.

Per la comunicazione interna tra microservizi vengono adottati **pattern eterogenei**, selezionati in funzione dei requisiti di accoppiamento e latenza:

- **REST/HTTP** per l'esposizione di API verso l'esterno e per alcune interazioni sincrone tra servizi;
- **gRPC** per la validazione dell'esistenza dell'utente da parte del *Data Collector* verso il *User Manager*, in modo da mantenere un accoppiamento debole a livello logico ma efficiente a livello di comunicazione;
- **Kafka** come message broker per la gestione delle comunicazioni asincrone:
    - dal *Data Collector* all'*Alert System* (topic di aggiornamento delle finestre di raccolta dei voli),
    - dall'*Alert System* all'*Alert Notifier* (topic di notifica dei superamenti soglia).

Le chiamate verso il servizio esterno **OpenSky Network API** sono realizzate mediante un client HTTP basato su RestTemplate, protetto da un **Circuit Breaker** configurato tramite **Resilience4j**. Il Circuit Breaker ha il compito di:

- monitorare il tasso di errore delle chiamate,
- aprire il circuito in caso di failure rate elevato, evitando di sovraccaricare il servizio esterno e il sistema interno,
- applicare una logica di *fallback* che restituisce una lista vuota di voli in caso di indisponibilità prolungata, preservando la coerenza dei flussi applicativi interni.

L'accesso esterno alle funzionalità del sistema è mediato da un **API Gateway** basato su **NGINX**, configurato come *reverse proxy* per:

- instradare le richieste HTTP verso i microservizi interni corretti (User Manager, Data Collector, ed eventualmente altri);
- centralizzare i punti d'ingresso, predisponendo il sistema a future evoluzioni in termini di autenticazione, autorizzazione, rate limiting e osservabilità a livello di perimetro.

Il sistema di notifica email si appoggia a un **server SMTP** configurato a livello applicativo, raggiunto tramite il componente `JavaMailSender`. Il microservizio *Alert Notifier* utilizza questo canale per recapitare all'utente messaggi strutturati che descrivono il superamento delle soglie configurate, includendo informazioni sul numero di voli rilevati, sul valore della soglia e sull'intervallo temporale considerato.

Dal punto di vista dei **vincoli progettuali**, la versione corrente assume:

- un ambiente di esecuzione **single-node**, orchestrato da Docker Compose, nel quale tutti i microservizi, il broker Kafka, il database PostgreSQL, l'API Gateway e il server SMTP di test coesistono;
- requisiti di carico e scalabilità **compatibili con un ambiente di dimostrazione e validazione** delle scelte architetturali, pur mantenendo un disegno che può essere esteso verso scenari di deploy più complessi (ad esempio, orchestrazione tramite Kubernetes, scalabilità orizzontale dei microservizi, distribuzione geografica);
- l'assenza, nella versione corrente, di meccanismi avanzati di sicurezza (autenticazione/autorìzzazione fine-grained, cifratura end-to-end), che possono essere introdotti in versioni successive senza alterare le scelte architetturali fondamentali qui descritte.

All'interno di questi vincoli, la progettazione si propone di massimizzare **chiarezza, separazione delle responsabilità e osservabilità dei flussi**, costruendo un sistema che rappresenti una base solida e coerente per eventuali evoluzioni future.

2. VISIONE D'INSIEME DELL'ARCHITETTURA ESTESA

2.1. DESCRIZIONE SINTETICA DEL SISTEMA NELLA RELEASE CORRENTE

Nella versione corrente il sistema di *flight monitoring* assume la forma di un'architettura a **microservizi** containerizzati, orientata a:

- raccogliere periodicamente dati di volo da **OpenSky Network API**;
- persistere tali dati in un **database relazionale**;
- permettere agli utenti di configurare **interessi sugli aeroporti** con soglie quantitative (*high_value*, *low_value*);
- generare **notifiche asincrone via email** quando i dati raccolti violano le soglie configurate;
- proteggere la dipendenza dal servizio esterno OpenSky mediante un **Circuit Breaker** applicato a livello di client HTTP.

Il sistema è composto da cinque microservizi principali:

- **User Manager Service**, responsabile della gestione degli utenti e della validazione delle identità;
- **Data Collector Service**, responsabile della raccolta periodica dei voli, della gestione degli interessi sugli aeroporti e dell'emissione di eventi verso la pipeline di allerta;
- **Alert System Service**, responsabile dell'elaborazione degli eventi di raccolta voli e della valutazione delle soglie configurate;
- **Alert Notifier Service**, responsabile della generazione e dell'invio delle notifiche email agli utenti;
- **API Gateway** basato su **NGINX**, che espone un punto di ingresso unico verso le API dei microservizi interni.

A supporto dei microservizi sono presenti:

- un **Kafka Broker** (accompagnato da ZooKeeper e da un'interfaccia di amministrazione) per la gestione dei **topic** utilizzati nella pipeline asincrona di notifica;
- un'istanza di **PostgreSQL**, che ospita sia il database utenti (User DB) sia il database dati (Data DB);
- un **server SMTP** (ad esempio Mailtrap in ambiente di sviluppo) utilizzato dall'Alert Notifier per il recapito delle email;
- il servizio esterno **OpenSky Network API**, integrato tramite client HTTP con **Circuit Breaker Resilience4j**.

Tutti questi componenti sono eseguiti come **container Docker** su un unico host, orchestrati tramite **Docker Compose** e collegati da una rete virtuale comune. I client

esterni interagiscono esclusivamente con l'API Gateway, che si occupa di instradare le richieste verso il microservizio appropriato all'interno della rete applicativa.

2.2. MICROSERVIZI E COMPONENTI INFRASTRUTTURALI

L'architettura della versione corrente è organizzata attorno a un insieme di **microservizi specializzati**, affiancati da una serie di **componenti infrastrutturali condivisi** che ne abilitano il funzionamento in un contesto distribuito. I microservizi implementano la logica di dominio e le API esposte verso l'esterno, mentre i componenti infrastrutturali forniscono servizi trasversali di persistenza, messaggistica, instradamento del traffico e integrazione con sistemi esterni.

Ogni microservizio è eseguito in un container dedicato, con propri endpoint di comunicazione e proprie dipendenze, e interagisce con gli altri esclusivamente tramite protocolli ben definiti (HTTP/REST, gRPC, Kafka). I componenti infrastrutturali, a loro volta, sono progettati per essere **riutilizzabili e indipendenti** rispetto alla logica applicativa specifica, in modo da poter supportare anche future estensioni del sistema senza richiedere modifiche invasive.

2.2.1. USER MANAGER SERVICE

Lo **User Manager Service** è un microservizio **Spring Boot** che implementa le funzionalità di gestione degli utenti. Le sue responsabilità principali sono:

- registrazione di nuovi utenti, identificati in maniera univoca tramite **email**;
- gestione dei metadati associati all'utente (ad esempio *name*);
- fornitura di un servizio di **validazione dell'esistenza dell'utente** esposto tramite **gRPC**, utilizzato dagli altri microservizi.

Dal punto di vista delle interfacce, il servizio espone:

- una **API REST** per la registrazione e la gestione degli utenti, accessibile tramite l'API Gateway;
- un **gRPC server** che fornisce un metodo di validazione (`userExists`), consumato dal Data Collector.

Lo User Manager persiste i dati nel database logico **User DB** (schema `userdb` in PostgreSQL), tipicamente tramite Spring Data JPA. Le operazioni esposte in questa versione rimangono coerenti con la versione precedente: il servizio non viene modificato in modo sostanziale, ma assume un ruolo ancora più critico come *source of truth* per la validazione degli utenti.

2.2.2. DATA COLLECTOR SERVICE

Il **Data Collector Service** è il microservizio responsabile di tutti i flussi legati:

- alla gestione degli **interessi utente sugli aeroporti** (inclusa, nella versione corrente, la configurazione delle soglie *high_value* e *low_value*);
- alla **raccolta periodica** dei voli dai servizi OpenSky per gli aeroporti di interesse;
- alla persistenza dei **flight records** nel **Data DB**;

- alla **pubblicazione di eventi Kafka** che alimentano la pipeline di notifica.

Le sue principali componenti logiche includono:

- una **REST API** (accessibile via API Gateway) che permette di:
    - registrare nuovi interessi sugli aeroporti per un utente,
    - aggiornare o rimuovere interessi esistenti,
    - configurare e modificare le soglie associate;
- un **scheduler** che, a intervalli regolari, attiva la raccolta dei voli per gli aeroporti configurati;
- un **OpenSky HTTP Client**, arricchito da un **Circuit Breaker Resilience4j**, che si occupa di:
    - interrogare le *OpenSky Network API* per ottenere le informazioni di volo,
    - gestire eventuali errori di rete, timeout o risposte non valide tramite fallback appropriati;
- un **client gRPC** per la validazione dell'utente presso lo User Manager;
- un **Kafka producer** che pubblica sul topic `to-alert-system` eventi contenenti il riepilogo dei voli raccolti e il contesto necessario alla valutazione delle soglie (ad esempio identificativo utente, aeroporto, finestra temporale, numero di voli riscontrati).

Il Data Collector utilizza il database logico **Data DB** (schema `datadb` in PostgreSQL) per persistere gli **interests** (ora arricchiti con le soglie) e i **flight_records**.

2.2.3. ALERT SYSTEM SERVICE

L'**Alert System Service** è un microservizio **Spring Boot** introdotto nella versione corrente per disaccoppiare la **valutazione delle soglie** dalla **raccolta dei dati**. Opera come un **consumer Kafka** del topic `to-alert-system`, sul quale il Data Collector pubblica eventi relativi alle finestre di raccolta dei voli.

Le responsabilità principali dell'Alert System sono:

- consumare gli eventi `ThresholdBreachEvaluationRequest` (o equivalente) prodotti dal Data Collector;
- recuperare, se necessario, le informazioni complementari dal Data DB (ad esempio le soglie associate a un interesse);
- calcolare, per ciascun evento, se il numero di voli raccolti:
    - supera la soglia *high_value*,
    - scende al di sotto della soglia *low_value*,
    - o si colloca all'interno dell'intervallo ammesso;

- generare, in caso di violazione delle soglie, un evento di notifica da pubblicare sul topic `to-notifier`, destinato all'Alert Notifier.

L'Alert System, quindi, realizza una **logica di dominio concentrata**, interamente dedicata alla gestione delle **policy di soglia**. In questo modo, il Data Collector rimane focalizzato sulla raccolta dei dati e sulla loro persistenza, mentre la responsabilità di interpretare tali dati in termini di *alerting* è delegata a un componente autonomo.

2.2.4. ALERT NOTIFIER SERVICE

L'**Alert Notifier Service** è un microservizio **Spring Boot** che si occupa della **generazione e invio delle notifiche** agli utenti. È configurato come **consumer Kafka** del topic `to-notifier`, sul quale l'Alert System pubblica gli eventi relativi ai superamenti delle soglie.

Le sue principali responsabilità includono:

- consumare i messaggi che descrivono una violazione di soglia (ad esempio, superamento di *high_value* o discesa al di sotto di *low_value*);
- formattare un contenuto email comprensibile per l'utente, includendo:
    - identificativo dell'aeroporto,
    - finestra temporale considerata,
    - valore della soglia,
    - numero di voli effettivamente conteggiati;
- utilizzare il componente `JavaMailSender` per inviare un'email all'indirizzo associato all'utente, tramite un **server SMTP** configurato (in ambiente di sviluppo, un servizio come Mailtrap).

L'Alert Notifier non accede direttamente ai database applicativi: consuma esclusivamente eventi *self-contained* (o quasi) provenienti dall'Alert System, secondo un approccio coerente con i principi di **event-driven architecture** e di separazione delle responsabilità.

2.2.5. API GATEWAY (NGINX)

L'**API Gateway** è implementato tramite **NGINX** configurato come *reverse proxy*. Esso rappresenta l'unico punto di contatto tra i **client esterni** e i **microservizi interni** del sistema.

Le sue funzioni principali sono:

- esporre un **endpoint HTTP unificato** (ad esempio sulla porta `:8080`) verso l'esterno;
- instradare le richieste in ingresso verso:
    - il **User Manager Service** per tutte le operazioni relative agli utenti;
    - il **Data Collector Service** per tutte le operazioni relative agli interessi sugli aeroporti e all'interrogazione dei dati di volo;

- nascondere la topologia interna dei microservizi, permettendo di modificare la configurazione (ad esempio porte e nomi dei container) senza impattare i client;
- costituire un punto naturale per l'eventuale introduzione futura di:
    - autenticazione e autorizzazione,
    - *rate limiting*,
    - logging centralizzato delle richieste,
    - ulteriori funzionalità trasversali.

In questa versione, l'API Gateway si concentra principalmente sulla **funzione di routing**, garantendo una chiara separazione tra perimetro esterno e rete interna dei servizi.

2.2.6. KAFKA BROKER E COMPONENTI DI SUPPORTO

Il sistema utilizza **Apache Kafka** come **message broker** per implementare la pipeline asincrona tra Data Collector, Alert System e Alert Notifier. Nel contesto della versione corrente, l'infrastruttura Kafka comprende:

- un **Kafka Broker**, responsabile della gestione dei topic e della memorizzazione dei messaggi;
- un'istanza di **ZooKeeper**, utilizzata per il coordinamento del broker (in linea con le versioni tradizionali di Kafka che richiedono ZooKeeper);
- una **Kafka UI** (o analogo strumento di amministrazione), impiegata per l'ispezione dei topic, la verifica dei messaggi e il monitoraggio dell'attività del cluster durante lo sviluppo e il test.

I topic principali gestiti includono:

- `to-alert-system`, sul quale il Data Collector pubblica gli eventi che descrivono il risultato della raccolta dei voli per una determinata finestra temporale e un determinato interesse utente;
- `to-notifier`, sul quale l'Alert System pubblica gli eventi di *threshold breach* destinati all'Alert Notifier.

L'adozione di Kafka consente di:

- ottenere un **disaccoppiamento temporale** tra la raccolta dei dati e la generazione delle notifiche;
- migliorare la **scalabilità** del sistema, potendo scalare i consumer per topic in base al carico;
- facilitare l'eventuale introduzione futura di ulteriori consumer (ad esempio servizi di analytics o dashboard in tempo quasi reale) senza modificare i produttori esistenti.

2.2.7. DATABASE POSTGRESQL (USER DB E DATA DB)

La persistenza dei dati è centralizzata su un'istanza di **PostgreSQL**, che ospita due database logici distinti:

- **User DB (`userdb`)**, utilizzato dallo User Manager per memorizzare:
  - gli utenti registrati, identificati dall'email,
  - eventuali metadati associati (nome, timestamp di creazione, ecc.);
- **Data DB (`datadb`)**, utilizzato principalmente dal Data Collector per gestire:
  - la tabella airports, che contiene le informazioni sugli aeroporti monitorabili;
  - la tabella `user_airport_interest`, che rappresenta gli interessi utente sugli aeroporti e che, nella versione corrente, è stata estesa con gli attributi `high_value` e `low_value` per codificare le soglie quantitative;
  - la tabella `flight_records`, che contiene i dati dei voli raccolti dalle OpenSky API, associati agli aeroporti di interesse.

L'accesso ai database è realizzato tramite **JPA/Spring Data**, con un mapping coerente al **diagramma ER aggiornato**. L'introduzione delle soglie nel modello dati è stata progettata in modo da:

- non alterare le chiavi primarie esistenti;
- preservare la compatibilità con i flussi di raccolta e interrogazione dei dati già implementati;
- consentire una valutazione delle soglie completamente gestita a livello applicativo (soprattutto nell'Alert System).

2.3. DIAGRAMMA ARCHITETTURALE – ARCHITETTURA COMPLESSIVA DEL SISTEMA ESTESO

L'architettura descritta è rappresentata da un **diagramma architetturale** che raffigura:

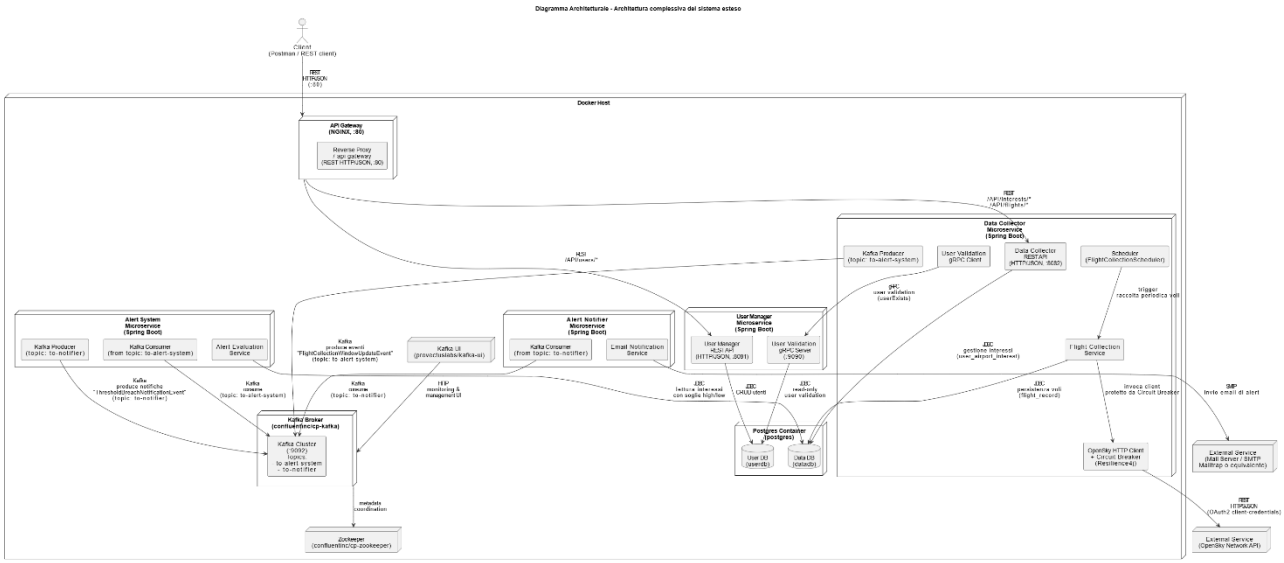
- il **client esterno** (ad esempio un REST client o Postman) che interagisce con l'**API Gateway NGINX**;
- i microservizi **User Manager**, **Data Collector**, **Alert System** e **Alert Notifier**, eseguiti come container separati all'interno dello stesso host Docker;
- il **Kafka Broker** (con ZooKeeper e Kafka UI) e il **database PostgreSQL** (con gli schemi logici User DB e Data DB) come componenti infrastrutturali condivisi;
- il **server SMTP** utilizzato per l'invio delle email dal microservizio Alert Notifier;
- il servizio esterno **OpenSky Network API**, contattato dal Data Collector tramite il client HTTP protetto da **Circuit Breaker Resilience4j**.

Il diagramma evidenzia i **protocolli di comunicazione** utilizzati:

- HTTP/JSON tra client esterno e API Gateway, e tra API Gateway e i microservizi REST interni;

- gRPC tra Data Collector e User Manager per la validazione dell'utente;
- Kafka come canale asincrono di comunicazione tra Data Collector, Alert System e Alert Notifier;
- JDBC tra i microservizi che accedono ai database e l'istanza PostgreSQL;
- SMTP tra l'Alert Notifier e il server di posta;
- HTTPS/JSON tra l'OpenSky client e l'API OpenSky.

Il diagramma costituisce il riferimento visivo principale per la comprensione complessiva della **topologia del sistema**, dei suoi confini e delle interazioni tra componenti applicativi e infrastrutturali.



2.4. EVOLUZIONE AD ALTO LIVELLO DELL'ARCHITETTURA RISPETTO ALLA VERSIONE PRECEDENTE

Rispetto alla versione precedente, nella quale l'architettura era centrata sui soli **User Manager** e **Data Collector** con un'interazione sincrona verso OpenSky e un accesso diretto ai database, la versione corrente introduce una serie di estensioni architetturali di rilievo.

Le principali evoluzioni possono essere riassunte lungo tre assi:

- **Estensione funzionale del dominio**:
  - introduzione del concetto di **soglia di interesse** per ciascun utente e aeroporto (`high_value`, `low_value`);
  - trasformazione del sistema da semplice piattaforma di raccolta e interrogazione dati in un sistema capace di generare **allarmi proattivi** mediante notifiche email;

- **Transizione verso un'architettura event-driven**:
    - introduzione del **Kafka Broker** e dei topic `to-alert-system` e `to-notifier`;
    - creazione del microservizio **Alert System**, che si occupa di valutare in modo asincrono le soglie sulla base degli eventi pubblicati dal Data Collector;
    - creazione del microservizio **Alert Notifier**, che reagisce agli eventi di violazione di soglia pubblicati dall'Alert System e invia email agli utenti;
    - disaccoppiamento temporale tra raccolta dati e notifica, con maggiore flessibilità in termini di scalabilità e robustezza;
- **Rafforzamento del perimetro e della resilienza**:
    - introduzione di un **API Gateway NGINX**, che centralizza l'ingresso delle richieste e isola la rete interna dei microservizi dai client esterni;
    - integrazione di un **Circuit Breaker Resilience4j** nel client HTTP verso OpenSky, in modo da:
        - prevenire l'accumulo di errori in caso di malfunzionamento del servizio esterno;
        - garantire comportamenti di fallback consistenti (ad esempio, trattare certe condizioni come "assenza di voli" anziché propagare eccezioni a catena).

Dal punto di vista della **topologia complessiva**, i microservizi già esistenti (User Manager, Data Collector) mantengono le proprie responsabilità principali, ma vengono inseriti in un ecosistema più articolato, nel quale:

- i **flussi sincroni** (REST, gRPC) continuano a coprire le interazioni di configurazione e interrogazione;
- i **flussi asincroni** (Kafka) estendono il sistema con capacità di *alerting* reattivo basato su eventi;
- le **dipendenze esterne** (OpenSky, SMTP) sono gestite con maggiore attenzione agli aspetti di resilienza e isolamento.

Questa evoluzione rende l'architettura più vicina ai pattern adottati nei sistemi distribuiti moderni, pur mantenendo continuità con la struttura logica e con i contratti funzionali definiti nella versione precedente.

3. MODELLO DEI DATI E PERSISTENZA

3.1. ESTENSIONE DEL MODELLO CONCETTUALE PER LA GESTIONE DELLE SOGLIE

Nel modello concettuale di partenza il dominio era strutturato attorno a poche entità fondamentali:

- **Utente**, identificato univocamente dall'email;
- **Aeroporto**, identificato da un codice (ad esempio ICAO/IATA) e caratterizzato da metadati descrittivi;
- **Interesse utente–aeroporto**, che rappresenta l'associazione tra un utente e un aeroporto di cui desidera monitorare il traffico;
- **Flight record**, che descrive un singolo volo raccolto dal sistema per uno specifico aeroporto e in una determinata finestra temporale.

L'"interesse" era modellato come una relazione *binaria*: un utente o è interessato a un aeroporto oppure no, senza una gradazione esplicita in termini di intensità del traffico ritenuta "accettabile" o "critica".

La versione corrente introduce il concetto di **soglia di interesse**, trasformando l'interesse da semplice flag di iscrizione a **policy parametrizzabile**. Per ciascuna coppia *(utente, aeroporto)* è ora possibile configurare:

- una **soglia alta** (*high_value*), che rappresenta il numero massimo di voli oltre il quale il traffico è considerato anomalo o meritevole di notifica;
- una **soglia bassa** (*low_value*), che rappresenta il numero minimo di voli al di sotto del quale si intende rilevare una situazione di traffico insolitamente scarso.

Le soglie sono concettualmente definite rispetto a una **finestra temporale di osservazione**, determinata dal processo di raccolta periodica del Data Collector (ad esempio un certo intervallo tra due esecuzioni dello scheduler). Il sistema valuta, per ogni finestra e per ogni interesse attivo, il **conteggio dei flight record** associati all'aeroporto e confronta tale valore con le soglie configurate.

La scelta progettuale è stata quella di **ancorare le soglie direttamente alla relazione di interesse** (entità `user_airport_interest`) anziché introdurre un'entità separata (ad esempio una "AlertPolicy" sganciata dall'interesse). Questa decisione è motivata da diversi fattori:

- le soglie sono semanticamente parte della configurazione dell'interesse stesso;
- l'associazione *uno-a-uno* tra interesse e policy di soglia semplifica il modello dei dati e le operazioni CRUD;
- la logica di dominio rimane più leggibile: un interesse rappresenta contemporaneamente:
    - l'"iscrizione" dell'utente a un aeroporto,

- la configurazione delle condizioni in base alle quali generare eventuali notifiche.

Dal punto di vista concettuale, le soglie sono **opzionali**: è possibile avere interessi per i quali *high_value* e/o *low_value* non sono impostate (valori null), interpretando tale condizione come “nessuna policy di alert attiva per quel tipo di soglia” e lasciando al servizio di valutazione la responsabilità di ignorare tali casi.

3.2. SCHEMA LOGICO AGGIORNATO DEI DATABASE

Lo schema logico mantiene la separazione in due database logici distinti, all’interno della stessa istanza PostgreSQL:

- **User DB** (`userdb`), dedicato alla persistenza degli utenti;
- **Data DB** (`datadb`), dedicato alla persistenza degli aeroporti, degli interessi e dei dati di volo.

L’estensione per la gestione delle soglie è stata realizzata in modo **localizzato**, concentrando le modifiche sulla tabella che rappresenta la relazione di interesse (`user_airport_interest`), senza introdurre nuove tabelle e senza modificare la struttura base delle altre entità. Il **diagramma ER aggiornato** riflette quindi una continuità rispetto alla versione precedente, con un arricchimento mirato del modello esistente.

3.2.1. USER DB: ENTITÀ E VINCOLI INVARIATI

Nel database logico **User DB** rimane centrale l’entità:

- `users (userdb.users)`, con:
  - `email : VARCHAR(255)` come **chiave primaria** (*PK*),
  - `name : VARCHAR(255)` come attributo descrittivo,
  - `created_at : TIMESTAMP NOT NULL` come timestamp di creazione.

I vincoli logici e fisici non sono stati modificati:

- l’email costituisce l’identificativo univoco dell’utente;
- eventuali vincoli di unicità o indicizzazione sono finalizzati a garantire:
  - ricerche efficienti per email,
  - impossibilità di duplicare un utente con la stessa email.

La relazione tra `users` e `user_airport_interest` rimane una **relazione logica** (utente ↔ interessi), non implementata tramite vincolo di chiave esterna a livello di database. L’integrità referenziale è **garantita a livello applicativo** attraverso il servizio di validazione gRPC esposto dallo User Manager, che viene invocato dal Data Collector prima di creare o aggiornare un interesse.

3.2.2. DATA DB: ESTENSIONE DI `user_airport_interest` E IMPATTO SU `flight_records`

Nel database logico **Data DB**, lo schema aggiornatamente rilevante comprende:

- `airports (datadb.airports)`, che elenca gli aeroporti gestiti dal sistema;
- `user_airport_interest (datadb.user_airport_interest)`, che rappresenta gli interessi utente–aeroporto;
- `flight_records (datadb.flight_records)`, che contiene i dati di volo raccolti.

La tabella `airports` mantiene la struttura già definita:

- `id : BIGSERIAL` **PK**;
- `code : VARCHAR(10) NOT NULL` con vincolo di **unicità** (codice aeroporto);
- `name : VARCHAR(255)`, `city : VARCHAR(255)`, `country : VARCHAR(255)`, `timezone : VARCHAR(64)`.

La tabella user_airport_interest è stata estesa nel modo seguente:

- `id : BIGSERIAL` **PK**;
- `user_email : VARCHAR(255) NOT NULL`;
- `airport_id : BIGINT NOT NULL`;
- `created_at : TIMESTAMP NOT NULL`;
- `high_value : INTEGER NULL`;
- `low_value : INTEGER NULL`;
- vincolo di **unicità composita** `UNIQUE(user_email, airport_id)`.

Il vincolo di unicità garantisce che, per una data coppia *(user_email, airport_id)*, esista **al massimo un interesse** attivo. Questo si traduce, a livello di dominio, nell'idea che un utente possa definire **una sola configurazione di soglie** per ciascun aeroporto, evitando situazioni ambigue in cui più interessi con soglie diverse convivano per la stessa coppia utente–aeroporto.

Gli attributi `high_value` e `low_value` sono definiti come `INTEGER NULL`, con le seguenti implicazioni:

- il database non impone limiti sul valore (ad esempio non garantisce direttamente la non negatività), demandando al livello applicativo le validazioni di dominio;
- la **nullabilità** consente di esprimere in modo naturale la condizione "soglia non configurata", evitando l'uso di valori sentinella o codifiche artificiali.

La tabella `flight_records` mantiene la struttura orientata alla registrazione dei dati di volo associati a ciascun aeroporto, tipicamente con campi come:

- `id : BIGSERIAL` **PK**;
- `airport_id : BIGINT NOT NULL` (collegamento all'aeroporto di riferimento);
- `external_flight_id : VARCHAR(64)` (identificativo esterno, quando disponibile);
- `flight_number : VARCHAR(32)`;
- `direction : VARCHAR(16)` (ad esempio "arrival" o "departure");
- `scheduled_time : TIMESTAMP`;
- `actual_time : TIMESTAMP`;
- `status : VARCHAR(32)`;
- `delay_minutes : INTEGER`;
- `collected_at : TIMESTAMP NOT NULL`.

Le relazioni tra le entità sono implementate tramite vincoli di **chiave esterna** (FK):

- `airports.id` → `user_airport_interest.airport_id`;
- `airports.id` → `flight_records.airport_id`.

La tabella `flight_records` non viene estesa direttamente per ospitare informazioni sulle soglie; il suo ruolo rimane quello di **registro storico dei voli**, mentre la logica di confronto con le soglie è demandata al livello applicativo (Alert System), che utilizza `flight_records` come base dati per i propri calcoli.

3.3. VINCOLI DI INTEGRITÀ, CONSISTENZA E IDEMPOTENZA NELLE NUOVE OPERAZIONI

L'introduzione delle soglie e della pipeline di notifica ha reso necessario esplicitare con maggiore precisione alcuni aspetti legati a **integrità**, **consistenza** e **idempotenza** delle operazioni sui dati.

Sul piano dell'**integrità referenziale**, le principali scelte sono:

- mantenere la relazione formale tra `airports` e:
  - `user_airport_interest` tramite `airport_id`,
  - `flight_records` tramite `airport_id`,

  assicurando che ogni interesse e ogni flight record punti a un aeroporto esistente;
- mantenere la relazione tra `user_airport_interest.user_email` e `users.email` come **vincolo logico**, non espresso a livello di database, ma garantito dall'applicazione tramite:
  - validazione preventiva via gRPC verso lo User Manager prima di inserire o aggiornare un interesse.

Per quanto riguarda l'**integrità del modello di interesse con soglie**, i vincoli principali sono:

- unicità della coppia *(user_email, airport_id)*, che impedisce la creazione di configurazioni multiple e incoerenti per la stessa coppia;
- possibilità di validare, a livello applicativo, condizioni di dominio quali:
    - non negatività delle soglie,
    - eventuale relazione tra *low_value* e *high_value* (ad esempio `low_value ≤ high_value`, quando entrambe sono valorizzate).

Sul piano della **consistenza** dei dati tra i diversi componenti:

- la creazione o l'aggiornamento di un interesse con soglie avviene tipicamente all'interno di una singola operazione applicativa, che aggiorna in modo atomico la riga in `user_airport_interest`;
- il Data Collector utilizza `user_airport_interest` e `flight_records` in maniera coerente: gli eventi pubblicati sul topic `to-alert-system` sono costruiti a partire da un insieme di flight record raccolti in una specifica finestra temporale e associati a interessi definiti in quel momento;
- l'Alert System legge le soglie dal Data DB e applica i calcoli su una base dati che rispecchia lo stato "committato" della persistenza, evitando dipendenze da stati intermedi non confermati.

L'**idempotenza** delle nuove operazioni è considerata su due livelli:

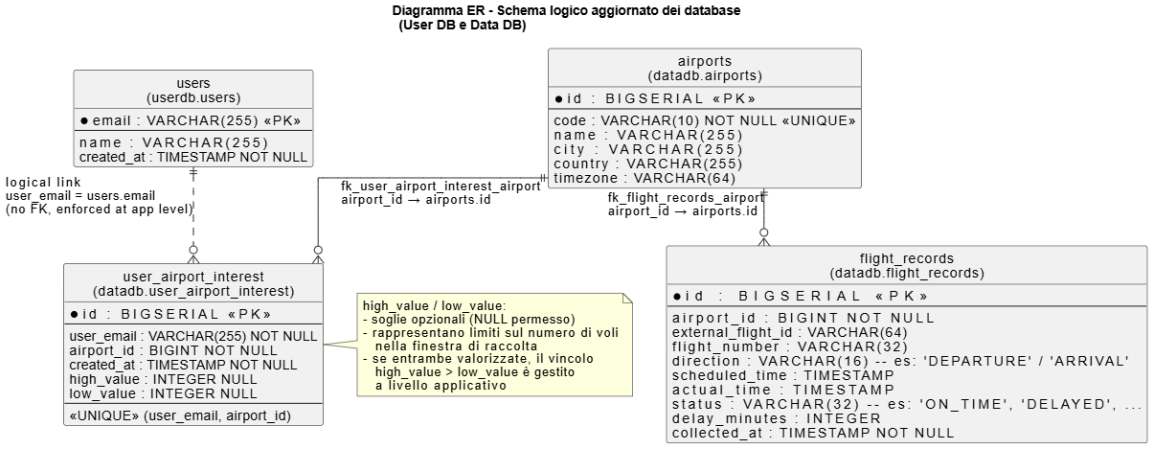
- **Operazioni sincrone di configurazione delle soglie**:
    - le API del Data Collector per la gestione degli interessi e delle soglie sono progettate in modo che richieste ripetute con lo stesso payload conducano allo stesso stato persistito;
    - la presenza del vincolo `UNIQUE(user_email, airport_id)` e l'uso di operazioni di *update* o *upsert* consente di evitare duplicazioni involontarie di interessi.
- **Operazioni asincrone nella pipeline di notifica**:
    - la persistenza delle soglie rimane concentrata in user_airport_interest, mentre il passaggio tramite Kafka consiste nello scambio di eventi derivati che non modificano direttamente lo stato dei database applicativi;
    - l'eventuale trattamento di duplicati a livello di messaggi (tipico dei sistemi *at-least-once delivery*) può essere gestito a livello applicativo, ad esempio limitando la generazione di notifiche multiple per lo stesso evento logico o introducendo meccanismi di deduplicazione, senza richiedere modifiche al modello fisico dei dati.

L’insieme di questi vincoli e accorgimenti consente di mantenere un **modello dati coerente** con le esigenze dell’architettura event-driven introdotta nella versione corrente, garantendo allo stesso tempo una base solida per successive estensioni.

3.4. DIAGRAMMA ER AGGIORNATO E MAPPATURA VERSO IL MODELLO JPA

Il **diagramma ER aggiornato** rappresenta la sintesi visuale del modello dati descritto, includendo:

- le entità:
  - `users` (User DB),
  - `airports`, `user_airport_interest`, `flight_records` (Data DB);
- le relazioni:
  - `airports` ↔ `user_airport_interest` (uno-a-molti),
  - `airports` ↔ `flight_records` (uno-a-molti),
  - legame logico `users` ↔ `user_airport_interest` attraverso `user_email`;
- gli attributi aggiuntivi `high_value` e `low_value` all’interno di `user_airport_interest`, evidenziati come estensione del modello esistente.



Il mapping verso il modello di persistenza applicativo è realizzato tramite **JPA** (Java Persistence API) e l’ecosistema **Spring Data**. Ogni tabella è associata a una **entità JPA** annotata con `@Entity`, con:

- una property corrispondente alla chiave primaria (`@Id` e, ove necessario, `@GeneratedValue`);
- proprietà che mappano gli attributi di colonna (`@Column`), includendo:
  - per `user_airport_interest`, campi come `userEmail`, `airportId`, `createdAt`, `highValue`, `lowValue`;

    - per `flight_records`, campi come `airportId`, `externalFlightId`, `flightNumber`, `direction`, `scheduledTime`, `actualTime`, `status`, `delayMinutes`, `collectedAt`;
- eventuali relazioni esplicite (`@ManyToOne`, `@OneToMany`) o, in alternativa, mapping basati su colonne di chiave esterna gestite come semplici attributi scalar, a seconda del livello di accoppiamento desiderato tra le entità.

Gli attributi `high_value` e `low_value` sono tipicamente mappati su campi di tipo `Integer` (e non `int`) nel modello JPA, in modo da rispecchiare la **nullabilità** consentita a livello di database e rappresentare in modo naturale la condizione di soglia non configurata.

Il diagramma ER e il mapping JPA risultano allineati, garantendo:

- corrispondenza univoca tra entità logiche, tabelle fisiche e classi di dominio;
- possibilità di sfruttare appieno le astrazioni offerte da Spring Data per le operazioni CRUD, senza perdere il controllo sui vincoli critici (chiavi primarie, vincoli di unicità, foreign key);
- una chiara separazione tra:
    - dati di **configurazione** (utenti, aeroporti, interessi con soglie),
    - dati **operativi** e storici (flight records raccolti nel tempo).

4. MODELLO DEI DATI E PERSISTENZA

4.1. CONFIGURAZIONE E AGGIORNAMENTO DELLE SOGLIE DI INTERESSE

La gestione delle **soglie di interesse** costituisce il primo tassello dei flussi introdotti nella versione corrente. Essa è realizzata mediante una serie di operazioni esposte dal **Data Collector Service** attraverso API REST, raggiungibili dai client esterni tramite l'**API Gateway**. I flussi considerano sia la creazione iniziale dell'interesse con soglie valorizzate, sia l'aggiornamento di un interesse già esistente, mantenendo coerenza con il modello dati e con le politiche di validazione applicativa.

4.1.1. CASO D'USO: CREAZIONE DELL'INTERESSE CON SOGLIE `high_value` / `low_value`

Nel caso d'uso di **creazione di un interesse** con soglie, il flusso applicativo può essere descritto come segue:

1. **Richiesta del client**
   Un client esterno invia una richiesta HTTP all'API Gateway per creare un interesse utente–aeroporto. Il payload include:
   - l'email dell'utente (`user_email`);
   - il codice dell'aeroporto o l'identificativo interno (`airport_id` o `airport_code`);
   - i valori delle soglie `high_value` e/o `low_value`, eventualmente omessi se non si desidera configurare una delle due soglie.
2. **Instradamento tramite API Gateway**
   L'API Gateway instrada la richiesta verso il **Data Collector Service**, esponendo un endpoint logico stabile verso l'esterno e traducendolo in un endpoint interno per il microservizio.
3. **Validazione dell'utente via gRPC**
   Il Data Collector, prima di registrare l'interesse, invoca il servizio di validazione esposto dallo **User Manager Service** tramite gRPC (`userExists`).
   - Se l'utente non esiste, la creazione dell'interesse viene rifiutata con una risposta di errore coerente (ad esempio HTTP 400/404);
   - Se l'utente esiste, il flusso prosegue.
4. **Risoluzione dell'aeroporto e controlli di dominio**
   Il Data Collector verifica l'esistenza dell'aeroporto nel **Data DB** (`airports`) e valida i valori di `high_value` / `low_value` secondo le regole di dominio (non negatività, eventuale rapporto fra le due soglie se entrambe sono impostate).
5. **Persistenza dell'interesse con soglie**
   Il microservizio crea una nuova riga nella tabella `user_airport_interest`, valorizzando:
   - `user_email`, `airport_id`, `created_at`;

- `high_value` e `low_value` secondo i valori forniti (o `NULL` se non specificati). Il vincolo `UNIQUE(user_email, airport_id)` impedisce la creazione di interessi duplicati per la stessa coppia.

6. **Risposta al client**
   Il Data Collector restituisce una risposta positiva al client, tramite l'API Gateway, includendo i dettagli dell'interesse appena creato, comprensivi delle soglie configurate.

Questo flusso rende atomica, dal punto di vista applicativo, la creazione dell'interesse e la definizione delle politiche di soglia, evitando la necessità di passi separati per la sola configurazione dei valori di `high_value` e `low_value`.

4.1.2. CASO D'USO: AGGIORNAMENTO DELLE SOGLIE PER UN INTERESSE ESISTENTE

Nel caso d'uso di **aggiornamento** delle soglie per un interesse già esistente, il flusso mantiene la stessa architettura di base, ma opera su un record preesistente in `user_airport_interest`:

1. **Richiesta del client**
   Il client invia una richiesta HTTP (solitamente `PUT` o `PATCH`) all'API Gateway, indicando:
   - l'utente e l'aeroporto di riferimento (tramite user_email e `airport_id`/`airport_code`);
   - i nuovi valori di `high_value` e/o `low_value` da impostare.
2. **Instradamento verso il Data Collector**
   L'API Gateway inoltra la richiesta al Data Collector, che interpreta l'operazione come una modifica di un interesse esistente.
3. **Validazione di esistenza**
   Il Data Collector:
   - verifica l'esistenza dell'utente tramite gRPC verso lo User Manager (opzionale se garantita a monte dal contesto, ma coerente con il modello di validazione);
   - verifica l'esistenza dell'aeroporto nel Data DB;
   - verifica l'esistenza di un record `user_airport_interest` per la coppia *(user_email, airport_id)*.
4. **Aggiornamento delle soglie**
   Se l'interesse esiste, il servizio aggiorna i campi `high_value` e `low_value` secondo la richiesta:
   - un valore esplicito imposta la soglia corrispondente;
   - l'omissione o un valore nullo può essere interpretato come "rimozione" della soglia, impostando la colonna a `NULL`.

5. **Salvataggio e risposta**
   Il record aggiornato viene persistito nel Data DB. Il Data Collector restituisce al client, tramite l'API Gateway, una rappresentazione aggiornata dell'interesse, riflettendo le nuove soglie.

Questo flusso consente di adattare dinamicamente le politiche di alert per un utente e un aeroporto, senza creare duplicati e nel rispetto del vincolo di unicità sulla coppia *(user_email, airport_id)*.

4.1.3. DIAGRAMMA DI SEQUENZA – CONFIGURAZIONE E AGGIORNAMENTO DELLE SOGLIE

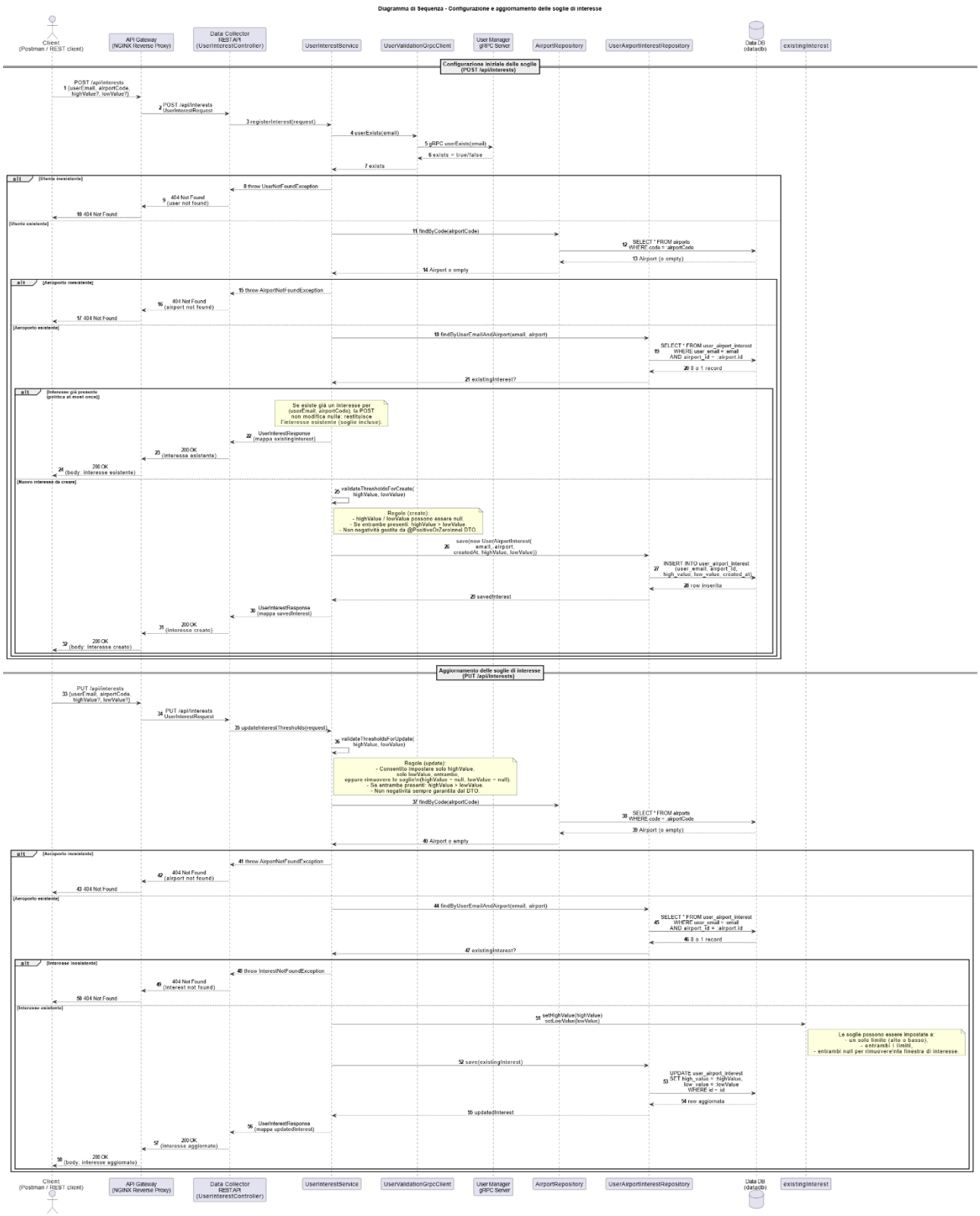
Il **diagramma di sequenza** dedicato alla configurazione e all'aggiornamento delle soglie rappresenta le interazioni tra:

- **Client → API Gateway → Data Collector Service → User Manager Service → Data DB**.

Il diagramma evidenzia:

- il passaggio della richiesta dal client all'API Gateway e da quest'ultimo al Data Collector;
- le invocazioni gRPC di validazione dell'utente verso lo User Manager;
- le operazioni di lettura/scrittura su `user_airport_interest` nel Data DB;
- le possibili varianti del flusso (creazione di un nuovo interesse, aggiornamento di uno esistente, errore in caso di utente o aeroporto inesistenti).

L'uso del diagramma permette di visualizzare con chiarezza il **percorso dell'informazione** e i punti di controllo principali per garantire la consistenza del modello dati.



4.2. RACCOLTA PERIODICA DEI VOLI CON CIRCUIT BREAKER VERSO OpenSky

La **raccolta periodica dei voli** costituisce il cuore operativo del sistema ed è realizzata dal Data Collector tramite uno scheduler che interroga ciclicamente le **OpenSky Network API**. Nella versione corrente questo flusso è stato arricchito dall'adozione di un **Circuit**

Breaker basato su *Resilience4j*, che protegge il sistema da condizioni di errore ripetuto o di degrado del servizio esterno.

4.2.1. INTERAZIONI CON OPENSKY E PROTEZIONE TRAMITE Resilience4j

Il flusso principale di raccolta dei voli è articolato come segue:

1. **Attivazione dello scheduler**
   Un componente di schedulazione, configurato nel Data Collector, si attiva a intervalli regolari. Per ciascuna finestra temporale di interesse:
   - individua gli aeroporti da monitorare, basandosi sulle informazioni contenute in `user_airport_interest` e sugli aeroporti configurati in `airports`;
   - calcola gli intervalli temporali da passare alle API OpenSky.
2. **Invocazione del client OpenSky con Circuit Breaker**
   Per ciascun aeroporto e finestra temporale, il Data Collector utilizza un **client HTTP** integrato con **Resilience4j**:
   - le chiamate verso l'endpoint OpenSky sono avvolte da un **Circuit Breaker** che monitora:
     - il tasso di errore (HTTP 5xx, timeout, eccezioni di rete);
     - il tempo di risposta;
   - lo stato del Circuit Breaker può essere:
     - *CLOSED*: le chiamate sono permesse e monitorate;
     - *OPEN*: le chiamate vengono bloccate immediatamente, attivando il fallback;
     - *HALF_OPEN*: vengono consentite un numero limitato di chiamate di prova per verificare il ripristino del servizio esterno.
3. **Interpretazione della risposta OpenSky**
   Quando la chiamata viene eseguita con successo, il client:
   - deserializza la risposta JSON in strutture di dominio;
   - filtra i voli pertinenti alla finestra temporale e all'aeroporto considerato;
   - costruisce una lista di flight records da persistere.
4. **Persistenza dei dati di volo**
   Il Data Collector salva i flight records nel Data DB (`flight_records`), associandoli all'aeroporto tramite `airport_id` e annotando il timestamp di raccolta (`collected_at`).
5. **Passaggio alla fase di valutazione delle soglie**
   A valle della persistenza, il Data Collector aggrega i dati per interesse utente–aeroporto e prepara gli eventi da inviare alla pipeline Kafka, come descritto nel paragrafo 4.3.

L’integrazione di Resilience4j permette di evitare che problemi ripetuti nel servizio OpenSky:

- degradino significativamente le prestazioni del Data Collector;
- causino catene di eccezioni propagate all’intero sistema;
- generino un eccessivo numero di tentativi verso un servizio esterno in stato di errore.

4.2.2. STRATEGIE DI FALLBACK, LOGGING E METRICHE DI ERRORE

Quando il Circuit Breaker si trova in stato *OPEN* o quando una singola chiamata verso OpenSky fallisce in modo non recuperabile, viene attivata una **strategia di fallback** definita a livello applicativo. Tipicamente, il fallback consiste in:

- restituire una **lista vuota** di voli per la richiesta corrente;
- trattare la condizione come “assenza di dati disponibili” per la finestra temporale considerata.

Questa scelta garantisce che:

- i flussi interni del Data Collector e dell’Alert System continuino a funzionare, anche in presenza di indisponibilità temporanea di OpenSky;
- le pipeline verso Kafka e le logiche di notifica non vengano interrotte da eccezioni non gestite.

Contestualmente, il sistema effettua un **logging strutturato** degli errori e degli eventi significativi, includendo informazioni quali:

- endpoint OpenSky chiamato,
- codice di risposta HTTP o tipo di eccezione,
- stato del Circuit Breaker,
- istante temporale e contesto dell’operazione.

Questi log sono fondamentali per:

- diagnosticare problemi di integrazione con OpenSky;
- analizzare il comportamento del Circuit Breaker nel tempo;
- correlare eventuali mancate notifiche con periodi di indisponibilità del servizio esterno.

Le **metriche di errore** e di stato del Circuit Breaker (numero di failure, tempo medio di risposta, transizioni tra stati) possono essere esposte a sistemi di monitoraggio esterni o consultate in fase di analisi, supportando attività di tuning dei parametri di Resilience4j (soglie di errore, durate delle finestre di osservazione, tempi di *wait* in stato OPEN).

4.2.3. DIAGRAMMA DI SEQUENZA – GESTIONE DELLE FAILURE VERSO OpenSky

Il **diagramma di sequenza** dedicato alla gestione delle failure verso OpenSky rappresenta le interazioni tra:

- **Scheduler → Data Collector → OpenSky HTTP Client + Circuit Breaker → OpenSky API → Data DB**.

Nel diagramma vengono evidenziati:

- il flusso normale in cui il Circuit Breaker è in stato *CLOSED* e la risposta viene elaborata correttamente;
- il flusso in cui la chiamata fallisce e viene attivato il fallback, con conseguente utilizzo di una lista vuota di voli;
- il flusso in cui il Circuit Breaker si trova in stato *OPEN*: in questo caso il client non tenta neppure la chiamata verso OpenSky e utilizza direttamente il fallback, registrando log specifici che indicano la causa (circuito aperto).

Questa rappresentazione consente di analizzare il comportamento del sistema nelle varie condizioni operative, distinguendo chiaramente tra problemi puntuali di rete e situazioni di degrado prolungato del servizio esterno.

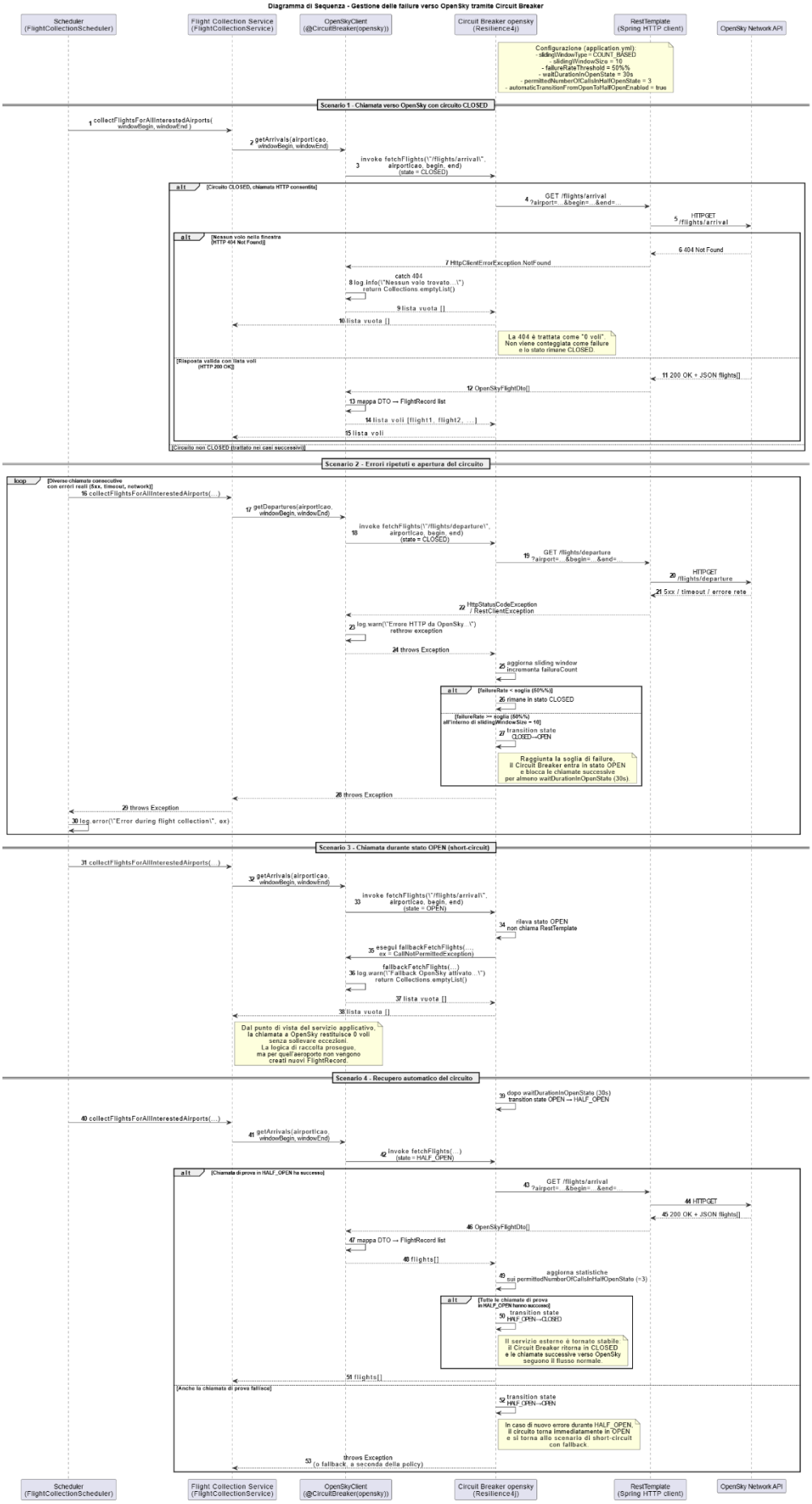


Diagramma di Sequenza - Gestione delle failure verso OpenSky tramite Circuit Breaker
Scheduler (FlightCollectionScheduler)
Flight Collection Service (FlightCollectionService)
OpenSkyClient (@CircuitBreaker(opensky))
Circuit Breaker opensky (Resilience4j)
RestTemplate (Spring HTTP client)
OpenSky Network API
Configurazione (application.yml):
- slidingWindowType = COUNT_BASED
- slidingWindowSize = 10
- failureRateThreshold = 50%
- waitDurationInOpenState = 30s
- permittedNumberOfCallsInHalfOpenState = 3
- automaticTransitionFromOpenToHalfOpenEnabled = true
Scenario 1 - Chiamata verso OpenSky con circuito CLOSED
1 collectFlightsForAllInterestedAirports(windowBegin, windowEnd)
2 getArrivals(airportIcao, windowBegin, windowEnd)
3 invoke fetchFlights(\"/flights/arrival\", airportIcao, begin, end) (state = CLOSED)
alt [Circuito CLOSED, chiamata HTTP consentita]
4 GET /flights/arrival ?airport=...&begin=...&end=...
5 HTTP GET /flights/arrival
alt [Nessun volo nella finestra (HTTP 404 Not Found)]
6 404 Not Found
7 HttpClientErrorException.NotFound
catch 404
8 log.info(\"Nessun volo trovato...\")
return Collections.emptyList()
9 lista vuota []
10 lista vuota []
La 404 è trattata come "0 voli". Non viene conteggiata come failure e lo stato rimane CLOSED
[Risposta valida con lista voli (HTTP 200 OK)]
11 200 OK + JSON flights[]
12 OpenSkyFlightDto[]
13 mappa DTO → FlightRecord list
14 lista voli [flight1, flight2, ...]
15 lista voli
[Circuito non CLOSED (trattato nei casi successivi)]
Scenario 2 - Errori ripetuti e apertura del circuito
loop [Diverse chiamate consecutive con errori reali (5xx, timeout, network)]
16 collectFlightsForAllInterestedAirports(...)
17 getDepartures(airportIcao, windowBegin, windowEnd)
18 invoke fetchFlights(\"/flights/departure\", airportIcao, begin, end) (state = CLOSED)
19 GET /flights/departure ?airport=...&begin=...&end=...
20 HTTP GET /flights/departure
21 5xx / timeout / errore rete
22 HttpStatusCodeException / RestClientException
23 log.warn(\"Errore HTTP da OpenSky...\") rethrow exception
24 throws Exception
25 aggiorna sliding window incrementa failureCount
alt [failureRate < soglia (50%%)]
26 rimane in stato CLOSED
[failureRate >= soglia (50%%) all'interno di slidingWindowSize = 10]
27 transition state CLOSED→OPEN
Raggiunta la soglia di failure, il Circuit Breaker entra in stato OPEN e blocca le chiamate successive per almeno waitDurationInOpenState (30s)
28 throws Exception
29 throws Exception
30 log.error(\"Error during flight collection\", ex)
Scenario 3 - Chiamata durante stato OPEN (short-circuit)
31 collectFlightsForAllInterestedAirports(...)
32 getArrivals(airportIcao, windowBegin, windowEnd)
33 invoke fetchFlights(\"/flights/arrival\", airportIcao, begin, end) (state = OPEN)
34 rileva stato OPEN non chiama RestTemplate
35 esegui fallbackFetchFlights(..., ex = CallNotPermittedException)
fallbackFetchFlights(...)
36 log.warn(\"Fallback OpenSky attivato...\")
return Collections.emptyList()
37 lista vuota []
38 lista vuota []
Dal punto di vista del servizio applicativo, la chiamata a OpenSky restituisce 0 voli senza sollevare eccezioni. La logica di raccolta prosegue, ma per quell'aeroporto non vengono creati nuovi FlightRecord.
Scenario 4 - Recupero automatico del circuito
39 dopo waitDurationInOpenState (30s) transition state OPEN → HALF_OPEN
40 collectFlightsForAllInterestedAirports(...)
41 getArrivals(airportIcao, windowBegin, windowEnd)
42 invoke fetchFlights(...) (state = HALF_OPEN)
alt [Chiamata di prova in HALF_OPEN ha successo]
43 GET /flights/arrival ?airport=...&begin=...&end=...
44 HTTP GET
45 200 OK + JSON flights[]
46 OpenSkyFlightDto[]
47 mappa DTO → FlightRecord list
48 flights[]
49 aggiorna statistiche sul permittedNumberOfCallsInHalfOpenState (=3)
alt [Tutte le chiamate di prova in HALF_OPEN hanno successo]
50 transition state HALF_OPEN→CLOSED
Il servizio esterno è tornato stabile. Il Circuit Breaker ritorna in CLOSED e le chiamate successive verso OpenSky seguono il flusso normale.
51 flights[]
[Anche la chiamata di prova fallisce]
52 transition state HALF_OPEN→OPEN
In caso di nuovo errore durante HALF_OPEN, il circuito torna immediatamente in OPEN e si torna allo scenario di short-circuit con fallback.
53 throws Exception (o fallback, a seconda della policy)

4.3. PIPELINE ASINCRONA DI NOTIFICA

La **pipeline asincrona di notifica** realizza la transizione da un modello puramente *request–response* a un modello **event-driven**, in cui i cambiamenti rilevanti (superamenti di soglia) vengono propagati mediante eventi su Kafka. Essa coinvolge il Data Collector, l'Alert System e l'Alert Notifier, con ruoli distinti e chiaramente separati.

4.3.1. PUBBLICAZIONE DEGLI EVENTI SUL TOPIC `to-alert-system`

A valle della raccolta e persistenza dei voli, il Data Collector:

1. **Aggrega i dati per finestra e interesse**
   Per ciascun interesse utente–aeroporto, calcola:
   - il numero di voli registrati nella finestra temporale corrente;
   - le informazioni necessarie a identificare l'utente e l'aeroporto, oltre alla finestra stessa.
2. **Costruisce un evento di valutazione soglie**
   Viene creato un oggetto di dominio (ad esempio una struttura `ThresholdEvaluationEvent` o equivalente) contenente:
   - identificativo dell'utente;
   - identificativo o codice dell'aeroporto;
   - intervallo temporale di riferimento;
   - numero di voli osservati (`observed_count`);
   - eventuali ulteriori metadati utili alla valutazione.
3. **Pubblica l'evento su Kafka**
   L'evento viene serializzato (tipicamente in JSON) e pubblicato sul topic `to-alert-system` tramite un **Kafka producer** configurato nel Data Collector.

Da questo momento, la responsabilità passa alla componente consumer (Alert System), senza che il Data Collector debba attendere la conclusione delle elaborazioni di soglia o dell'invio delle notifiche.

4.3.2. VALUTAZIONE DELLE SOGLIE NEL SERVIZIO ALERT SYSTEM

L'**Alert System Service** è configurato come **consumer** del topic `to-alert-system`. Il suo flusso operativo è il seguente:

1. **Consumo degli eventi**
   Il servizio legge i messaggi in ingresso dal topic, deserializzando gli eventi di valutazione.
2. **Recupero delle soglie**
   Per ciascun evento, l'Alert System:
   - identifica la coppia *(utente, aeroporto)*;

- recupera dal Data DB (`user_airport_interest`) i valori di `high_value` e `low_value` eventualmente configurati.

3. **Calcolo delle condizioni di soglia**
   Sulla base di `observed_count` e delle soglie configurate, l'Alert System valuta:
   - se `observed_count` > `high_value` (quando `high_value` è valorizzata), generando un evento di superamento soglia alta;
   - se `observed_count` < `low_value` (quando `low_value` è valorizzata), generando un evento di superamento soglia bassa;
   - se nessuna delle condizioni di soglia è soddisfatta, nessun evento di notifica viene generato.
4. **Preparazione dell'evento per il notifier**
   In caso di violazione di soglia, il servizio costruisce un evento di notifica che include:
   - email dell'utente;
   - codice o nome dell'aeroporto;
   - tipo di soglia violata (alta o bassa);
   - valore della soglia;
   - numero di voli osservati;
   - intervallo temporale considerato.
5. **Pubblicazione sul topic `to-notifier`**
   L'evento di notifica viene pubblicato sul topic `to-notifier`, pronto per essere consumato dall'Alert Notifier.

Questo approccio consente di concentrare nella componente Alert System tutta la logica di valutazione delle soglie, mantenendo il Data Collector focalizzato sulla raccolta dei dati.

4.3.3. INVIO DELLE NOTIFICHE NEL SERVIZIO ALERT NOTIFIER

L'**Alert Notifier Service** agisce come **consumer** del topic `to-notifier`. Il suo flusso può essere schematizzato come segue:

1. **Consumo degli eventi di notifica**
   Il servizio legge i messaggi dal topic, deserializzando le informazioni necessarie alla composizione dell'email.
2. **Composizione del contenuto della notifica**
   Viene costruito il corpo dell'email, tipicamente includendo:
   - saluto e identificazione dell'utente destinatario;
   - descrizione dell'evento (superamento soglia alta o bassa);
   - dettaglio dei parametri:
     - aeroporto coinvolto;

- numero di voli osservati;
- valore della soglia;
- intervallo temporale di riferimento.

3. **Invio dell'email via SMTP**
   L'Alert Notifier utilizza `JavaMailSender` per inviare l'email tramite il **server SMTP** configurato.
   Eventuali eccezioni (es. problemi nel recapito) vengono tracciate tramite logging, senza impattare i flussi interni di consumo degli eventi.
4. **Gestione di eventuali casi limite**
   In presenza di problemi temporanei nel servizio SMTP, possono essere previste strategie di retry o logging avanzato, in modo da non perdere la traccia delle notifiche non recapitate.

L'Alert Notifier non accede direttamente al database: opera esclusivamente sugli eventi ricevuti, secondo un approccio coerente con i principi di *event-driven microservices*.

4.3.4. DIAGRAMMA DI SEQUENZA – PIPELINE DI NOTIFICA ASINCRONA

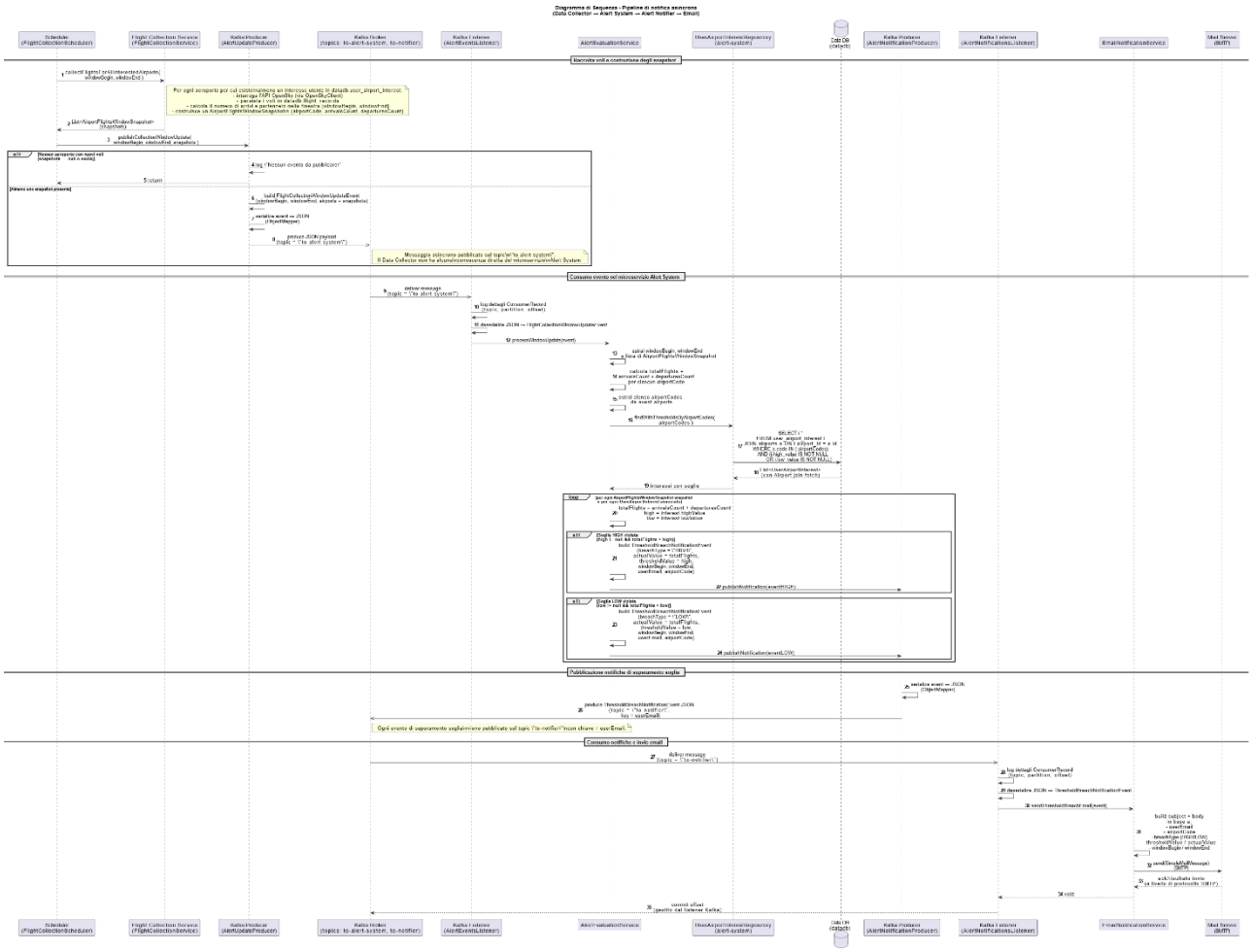
Il **diagramma di sequenza** della pipeline di notifica asincrona raffigura i passaggi chiave tra:

- **Data Collector** → Kafka (`to-alert-system`) → **Alert System** → Kafka (`to-notifier`) → **Alert Notifier** → **SMTP server** → **utente**.

Nel diagramma sono messi in evidenza:

- il momento in cui il Data Collector pubblica gli eventi di valutazione sul topic `to-alert-system`;
- la fase in cui l'Alert System consuma tali eventi, interroga il Data DB per le soglie e genera, se necessario, eventi sul topic `to-notifier`;
- l'attività dell'Alert Notifier nel consumare gli eventi, comporre le email e interagire con il server SMTP.

Questa rappresentazione chiarisce come la pipeline permetta di **disaccoppiare** temporalmente e logicamente la raccolta dei dati dall'invio delle notifiche, migliorando la scalabilità e la robustezza complessiva del sistema.



4.4. INGRESSO CENTRALIZZATO TRAMITE API GATEWAY

L'**API Gateway** svolge un ruolo cruciale nel modellare le interazioni tra i client esterni e i microservizi interni, fungendo da **punto di ingresso unificato** e da elemento di isolamento tra perimetro esterno e rete interna del sistema.

4.4.1. ROUTING DELLE RICHIESTE VERSO I MICROSERVIZI INTERNI

Il flusso di routing tramite API Gateway segue la logica:

1. **Ricezione della richiesta HTTP esterna**
   Il client invia una richiesta verso l'endpoint esposto dall'API Gateway, che può includere:
   - path specifici per le operazioni utente (`/api/users/...`);
   - path per la gestione degli interessi e dei voli (`/api/flights/...`, `/api/interests/...`).
2. **Instradamento verso il servizio appropriato**
   NGINX, configurato come *reverse proxy*, utilizza regole di routing basate sul path (e, se necessario, su altri criteri) per:
   - inoltrare le richieste user-related verso il container del **User Manager Service**;

- inoltrare le richieste relative ad aeroporti, interessi e interrogazioni sui voli verso il **Data Collector Service**.

3. **Trasparenza della topologia interna**
   I client non sono a conoscenza dei dettagli interni:
   - nomi dei container,
   - porte interne dei microservizi,
   - eventuali modifiche di deployment.
     L'API Gateway funge da strato di astrazione, consentendo di modificare la configurazione interna senza impatti sui consumatori esterni.
4. **Restituzione della risposta**
   Le risposte generate dai microservizi vengono inoltrate dall'API Gateway al client, mantenendo la trasparenza del flusso a livello HTTP.

Questo schema permette di mantenere un **contratto stabile** verso l'esterno, pur consentendo una maggiore flessibilità nella gestione interna dei microservizi.

4.4.2. CONSIDERAZIONI SU LOGGING, TRACCIAMENTO E SICUREZZA DEL PERIMETRO

Dal punto di vista **osservazionale e di sicurezza**, l'API Gateway rappresenta una posizione privilegiata per:

- centralizzare il **logging delle richieste**, registrando:
  - URL richiesto,
  - metodo HTTP,
  - codice di risposta,
  - tempi di risposta globali;
- introdurre, quando necessario, meccanismi di **tracciamento delle richieste** (ad esempio mediante *correlation id* propagati ai microservizi interni), facilitando l'analisi end-to-end dei flussi;
- delimitare il **perimetro di esposizione**:
  - solo le porte dell'API Gateway e dei servizi strettamente necessari (ad esempio SMTP, se richiesto) sono esposte verso l'esterno;
  - i microservizi interni e i componenti infrastrutturali (Kafka, PostgreSQL) restano accessibili solo all'interno della rete privata Docker.

Sebbene nella versione corrente l'API Gateway sia focalizzato principalmente sul routing, la sua presenza rende il sistema predisposto all'introduzione futura di:

- autenticazione e autorizzazione centralizzate;
- limitazione del traffico (rate limiting, throttling);
- filtri di sicurezza aggiuntivi (ad esempio IP whitelisting o header validation).

4.5. SINTESI DEI FLUSSI END-TO-END NEL SISTEMA ESTESO

Nel sistema esteso i flussi applicativi si articolano in una **catena di elaborazione** che collega configurazione, raccolta dati, valutazione di soglie e notifica all'utente:

- un utente viene gestito dal **User Manager Service** e, tramite il **Data Collector Service**, può definire interessi sugli aeroporti con soglie `high_value` / `low_value`;
- lo scheduler del Data Collector interroga periodicamente le **OpenSky Network API**, passando attraverso il **Circuit Breaker Resilience4j** per proteggere il sistema da condizioni di errore ripetuto;
- i dati di volo vengono salvati nel **Data DB**, e per ogni interesse vengono costruiti eventi che descrivono il numero di voli osservati, pubblicati sul topic `to-alert-system`;
- l'**Alert System Service** consuma tali eventi, recupera le soglie configurate e determina se siano state violate, pubblicando in caso positivo eventi sul topic `to-notifier`;
- l'**Alert Notifier Service** consuma gli eventi di notifica, compone email informative e le invia agli utenti tramite il server **SMTP**;
- tutte le operazioni di configurazione e interrogazione delle API avvengono tramite l'**API Gateway**, che fornisce un ingresso centralizzato al sistema.

La combinazione di questi flussi produce un sistema in grado di trasformare **dati grezzi di volo** in **informazioni operative** per l'utente finale, con un'architettura che sfrutta in modo coordinato pattern sincroni (REST, gRPC) e asincroni (Kafka), preservando separazione delle responsabilità, manutenibilità e possibilità di evoluzione futura.

5. API ESPOSTE E CONTRATTI DI SERVIZIO

5.1. CONVENZIONI GENERALI PER LE API REST ESPOSTE TRAMITE API GATEWAY

Tutte le API esposte verso l'esterno sono oggi raggiungibili *esclusivamente* tramite l'**API Gateway** basato su **NGINX**, che svolge il ruolo di reverse proxy e di punto di ingresso unico al sistema.

Le principali convenzioni adottate sono le seguenti:

- **Protocollo e formato dati**
  - tutte le API seguono lo stile **RESTful** su **HTTP/HTTPS**;
  - il formato di scambio è **JSON** UTF-8, sia in richiesta sia in risposta;
  - le strutture dati sono versionate implicitamente tramite l'evoluzione del contratto e documentate nel presente documento.
- **Base path e routing**
  - il gateway espone un **endpoint unico** del tipo `http://<host_gateway>:<port>/api/...`;
  - il routing verso i microservizi interni è **path-based**, con mapping logici del tipo:
    - `/api/users/**` → microservizio *User Manager*;
    - `/api/interests/**` e `/api/airports/**` → microservizio *Data Collector*;
    - `/api/flights/**` → microservizio *Data Collector* (interrogazione dei voli);
  - il client non interagisce mai direttamente con i microservizi, ma sempre e solo con il gateway.
- **Codici di stato HTTP**
  - `2xx` per operazioni concluse con successo (ad es. `200 OK`, `201 Created`, `204 No Content`);
  - `4xx` per errori lato client (ad es. `400 Bad Request`, `404 Not Found`, `409 Conflict`);
  - `5xx` per errori lato server o condizioni di malfunzionamento non imputabili al client;
  - gli errori applicativi sono rappresentati tramite un **modello di errore omogeneo** già definito nella versione precedente della documentazione, mantenuto invariato nella release corrente.

- **Gestione degli errori e idempotenza**
    - le operazioni di **creazione/aggiornamento degli interessi** implementano una semantica di tipo *upsert* basata sulla coppia `(userEmail, airportCode)`, garantendo l'assenza di duplicati e un comportamento deterministico in presenza di retry;
    - le operazioni di **lettura** sono **idempotenti**;
    - le operazioni di **cancellazione** sono idempotenti per costruzione (`DELETE` su una risorsa già assente restituisce una risposta coerente con il modello di errore).

L'introduzione dell'API Gateway non modifica la semantica delle API già esistenti nella versione precedente, ma ne **centralizza l'accesso**, permettendo in prospettiva di applicare in un unico punto politiche di *logging*, tracciamento, rate limiting e, in release successive, meccanismi di sicurezza più avanzati (autenticazione/authorization).

5.2. ESTENSIONI DELLE API DEL DATA COLLECTOR PER LA GESTIONE DELLE SOGLIE

Il microservizio **Data Collector** mantiene la responsabilità sul dominio *aeroporti–interessi–voli* e sulle API di interrogazione dei dati di volo. Nella release corrente le sue API REST sono state estese per supportare la gestione delle **soglie di interesse** associate alla coppia *(utente, aeroporto)* tramite i parametri `highValue` e `lowValue`, preservando le interfacce esistenti e introducendo un arricchimento dei payload.

Le principali caratteristiche dell'estensione sono:

- la risorsa logica **"interesse utente–aeroporto"** continua a essere identificata dalla coppia `(userEmail, airportCode)` e memorizzata nel Data DB, con un vincolo di unicità a livello di schema;
- i nuovi attributi `highValue` e `lowValue` sono **facoltativi** e possono essere:
    - forniti contestualmente alla creazione dell'interesse;
    - aggiunti in un secondo momento su un interesse privo di soglie;
    - aggiornati su un interesse esistente che già possiede soglie;
- sono enforce-ate le seguenti regole:
    - può essere presente **anche un solo valore** tra `highValue` e `lowValue`;
    - se entrambi sono presenti, deve valere `highValue > lowValue`;
    - in caso di violazione delle regole, l'API risponde con `400 Bad Request` e payload di errore descrittivo.

Le estensioni sono state progettate per **non rompere** i client esistenti:

- i campi di soglia sono opzionali in input;
- gli endpoint originari continuano a funzionare anche se il client non è consapevole dei nuovi attributi.

5.2.1. ENDPOINT DI CREAZIONE E AGGIORNAMENTO DEGLI INTERESSI CON SOGLIE

La gestione delle soglie sfrutta l'endpoint già esistente per la creazione degli interessi, estendendone il contratto in modo da supportare anche lo scenario di **aggiornamento**:

- **Endpoint**:
  `POST /api/interests`
- **Body di richiesta (schema logico)**:

```
{
  "userEmail": "mario.rossi@example.com",
  "airportCode": "LIMC",
  "highValue": 120,
  "lowValue": 20
}
```

  - `userEmail` (string, obbligatorio): e-mail dell'utente, utilizzata come chiave logica verso il *User DB* e validata via gRPC dal *User Manager*;
  - `airportCode` (string, obbligatorio): codice IATA/ICAO dell'aeroporto, validato rispetto al catalogo interno `airports`;
  - `highValue` (integer, opzionale): soglia superiore per il numero di voli;
  - `lowValue` (integer, opzionale): soglia inferiore per il numero di voli.
- **Semantica applicativa**:
  1. Il Data Collector inoltra una richiesta di validazione dell'utente al *User Manager* tramite gRPC, mantenendo il confine dei domini applicativi come nella release precedente.
  2. Verifica l'esistenza dell'aeroporto nel catalogo `airports`.
  3. Se **non esiste** ancora un interesse per `(userEmail, airportCode)`, viene creato un nuovo record in `user_airport_interest` con eventuali soglie valorizzate.
  4. Se **esiste già** un interesse per `(userEmail, airportCode)`:
     - i valori `highValue` e `lowValue` presenti nel payload **aggiornano** le soglie esistenti;
     - i campi omessi nel payload possono essere lasciati invariati o azzerati, in base alla strategia progettuale adottata (ad es. *full replace* oppure *partial update* mediante convenzione applicativa esplicitata nella documentazione dettagliata).
- **Codici di risposta tipici**:
  - `201 Created` in caso di creazione di un nuovo interesse;
  - `200 OK` in caso di aggiornamento di un interesse esistente;

- `400 Bad Request` per violazioni di vincoli (soglie non coerenti, parametri mancanti);
- `404 Not Found` se l'utente o l'aeroporto non risultano validi;
- `409 Conflict` in caso di conflitti non risolvibili sul vincolo di unicità.

Questo approccio consente di mantenere un **contratto compatto** e di implementare una semantica di *upsert* sulla risorsa "interesse con soglie", semplificando i flussi client e preservando la coerenza con il modello dati.

5.2.2. ENDPOINT DI LETTURA E CANCELLAZIONE DEGLI INTERESSI AGGIORNATI

Gli endpoint di **lettura** e **cancellazione** degli interessi mantengono i path originari, ma i payload restituiti in risposta includono ora, ove presenti, i valori di `highValue` e `lowValue`.

- **Lettura degli interessi di un utente**
  - **Endpoint**:
    `GET /api/interests?userEmail=<email>`
  - **Risposta (esempio)**:

```
[
  {
    "airportCode": "LIMC",
    "highValue": 120,
    "lowValue": 20,
    "createdAt": "2025-12-10T09:15:30Z"
  },
  {
    "airportCode": "LIRN",
    "highValue": null,
    "lowValue": 10,
    "createdAt": "2025-12-10T09:16:00Z"
  }
]
```

  - I record restituiti comprendono la **fotografia attuale** delle soglie associate a ciascun aeroporto di interesse, e costituiscono il riferimento per i client che desiderano verificare o presentare all'utente la configurazione delle notifiche.

- **Cancellazione di un interesse**
  - **Endpoint**:
    `DELETE /api/interests?userEmail=<email>&airportCode=<code>`
  - La cancellazione rimuove l'associazione `(userEmail, airportCode)` dal Data DB, eliminando contestualmente i valori di soglia. Eventuali notifiche in transito non vengono duplicate, mentre le successive elaborazioni non considereranno più quell'interesse.

- L’operazione è **idempotente**: una richiesta di cancellazione ripetuta produce sempre lo stesso stato finale (assenza dell’interesse) e una risposta coerente rispetto al modello di errore adottato

5.3. API PER L’INTERROGAZIONE DEI VOLI: RIEPILOGO E RIMANDO ALLA DOCUMENTAZIONE DI BASE

Le API per l’interrogazione dei voli, già definite nella versione precedente, **non subiscono modifiche strutturali** nella release corrente. Rimangono disponibili, tramite API Gateway, gli endpoint che permettono al client di:

- ottenere l’**ultimo volo** registrato per aeroporto e direzione (arrivo/partenza);
- calcolare la **media dei voli** su un certo numero di giorni per un aeroporto;
- recuperare l’**elenco completo dei voli** registrati in un intervallo temporale.

Gli schemi di richiesta e risposta, i filtri disponibili (ad es. per codice aeroporto, direzione, intervallo temporale) e le considerazioni di performance e indicizzazione restano quelli documentati nella relazione della versione precedente, alla quale si rimanda per i dettagli di dominio.

Nell’architettura estesa, queste API continuano a rappresentare il **punto di accesso principale** per la consultazione dei dati di volo, mentre i nuovi servizi *Alert System* e *Alert Notifier* operano in maniera asincrona su base batch, facendo leva sullo stesso **Data DB** per la valutazione delle soglie e la generazione delle notifiche, senza esporre ulteriori API pubbliche.

5.4. CONTRATTI DEI MESSAGGI KAFKA

L’introduzione di **Kafka** come message broker abilita una pipeline asincrona composta da due topic:

- `to-alert-system`, sul quale il **Data Collector** pubblica un evento al termine di ogni ciclo di raccolta dei voli;
- `to-notifier`, sul quale l’**Alert System** pubblica le condizioni di superamento soglia destinate all’**Alert Notifier**.

Entrambi i topic trasportano messaggi **JSON** con schema stabilizzato, per favorire l’evolvibilità dei servizi e la possibile introduzione futura di *schema registry* o validazioni formali (ad es. JSON Schema, Avro).

5.4.1. TOPIC `to-alert-system`: SCHEMA DEL MESSAGGIO E SEMANTICA

Il topic `to-alert-system` veicola la notifica di completamento di un **batch di raccolta** da parte del Data Collector. Il messaggio rappresenta un *trigger* per l’Alert System, mettendolo in condizione di valutare le soglie per tutti gli interessi registrati.

Uno schema logico del messaggio può essere rappresentato come segue:

```
{
  "batchId": "7a9d5c4e-1234-4b56-9f00-abc123def456",
  "windowStart": "2025-12-10T08:00:00Z",
```

```
  "windowEnd": "2025-12-10T09:00:00Z",
  "airports": [
    "LIMC",
    "LIRN",
    "LICC"
  ],
  "collectedAt": "2025-12-10T09:01:15Z"
}
```

- `batchId` (string): identificatore univoco del batch di raccolta;
- `windowStart` / `windowEnd` (datetime ISO-8601): estremi temporali dell'intervallo per cui sono stati raccolti i voli;
- `airports` (array di string): elenco dei codici aeroporto per i quali è stata effettuata la raccolta in questo batch;
- `collectedAt` (datetime ISO-8601): timestamp di completamento del processo di ingestione e persistenza dei voli.

Semantica:

- il messaggio non replica necessariamente l'intero dettaglio dei voli (già persistito nel Data DB), ma fornisce all'Alert System tutte le informazioni necessarie per:
    - identificare il *time window* di riferimento;
    - identificare gli aeroporti coinvolti;
    - interrogare il Data DB per derivare il **numero di voli** pertinenti a ciascun interesse utente–aeroporto, distinguendo arrivi e partenze;
- la responsabilità di calcolare le metriche da confrontare con le soglie (`highValue`, `lowValue`) è demandata all'**Alert System**, che può applicare logiche di aggregazione e filtri coerenti con i requisiti funzionali.

Questa scelta separa in modo netto il **ruolo di ingestion** (Data Collector) dal **ruolo di valutazione e alerting** (Alert System), evitando duplicazioni di dati sul broker e mantenendo il Data DB come *single source of truth* per i flight records.

5.4.2. TOPIC `to-notifier`: SCHEMA DEL MESSAGGIO E SEMANTICA

Il topic `to-notifier` trasporta i messaggi che rappresentano una **violazione di soglia** per una coppia *(utente, aeroporto)*. Il payload è aderente alle esigenze dell'Alert Notifier, che deve costruire e inviare la mail di notifica.

Uno schema logico del messaggio può essere il seguente:

```
{
  "userEmail": "mario.rossi@example.com",
  "airportCode": "LIMC",
  "breachType": "HIGH",
  "direction": "ARRIVAL",
  "thresholdValue": 120,
```

```
  "currentFlightsCount": 135,
  "evaluationWindow": {
    "windowStart": "2025-12-10T08:00:00Z",
    "windowEnd": "2025-12-10T09:00:00Z"
  },
  "generatedAt": "2025-12-10T09:02:10Z"
}
```

- `userEmail (string)`: indirizzo e-mail del destinatario della notifica;
- `airportCode` (string): codice dell'aeroporto per cui è stata rilevata la violazione;
- `breachType` (string): tipo di superamento soglia, ad esempio:
  - `"HIGH"`: numero di voli > `highValue`;
  - `"LOW"`: numero di voli < `lowValue`;
- `direction` (string): direzione considerata, ad esempio `"ARRIVAL"`, `"DEPARTURE"` oppure eventuali combinazioni consentite dall'implementazione;
- `thresholdValue` (integer): valore di soglia che è stato superato;
- `currentFlightsCount` (integer): numero di voli calcolato nell'intervallo di valutazione;
- `evaluationWindow` (oggetto): finestra temporale di riferimento (stessi estremi temporali del batch che ha originato la valutazione);
- `generatedAt` (datetime ISO-8601): timestamp di generazione del messaggio da parte dell'Alert System.

Semantica:

- ogni messaggio su `to-notifier` rappresenta un **evento di alert già filtrato**, pronto per essere tradotto in una notifica verso l'utente;
- l'Alert Notifier non applica ulteriori logiche di dominio (controllo soglie o aggregazioni), ma si limita a:
  - interpretare il payload;
  - costruire una mail con:
    - `to` = `userEmail`;
    - `subject` = riferimento all'`airportCode` e alla condizione di superamento;
    - `body` = testo che descrive il tipo di violazione (soglia superiore o inferiore) e il numero di voli rilevato;
  - inviare la mail tramite il **Mail Sender** configurato.

La separazione tra to-alert-system e to-notifier consente di **scalare** indipendentemente la fase di valutazione delle soglie e quella di notifica, mantenendo contratti di messaggistica chiari e specifici per ciascun ruolo

5.5. INTERFACCE INTERNE E CONFIGURAZIONE DEL CIRCUIT BREAKER

L'introduzione del **Circuit Breaker** riguarda le interazioni verso il servizio esterno **OpenSky Network** e si basa sull'integrazione di **Resilience4j** all'interno del microservizio **Data Collector**.

Le scelte progettuali principali sono:

- **Interfaccia applicativa invariata**
  - il componente `OpenSkyClient` continua a esporre i metodi:
    - `getArrivals(airportCode, begin, end);`
    - `getDepartures(airportCode, begin, end);`
  - il chiamante principale, `FlightCollectionService`, utilizza la stessa interfaccia descritta nella documentazione precedente, senza dover conoscere i dettagli di Resilience4j.
- **Wrapping tramite Circuit Breaker**
  - le chiamate verso OpenSky vengono eseguite all'interno di un **decorator** di Resilience4j, configurato tramite proprietà applicative (ad es. in `application.yml`);
  - il Circuit Breaker monitora:
    - il **tasso di errore** delle chiamate HTTP verso OpenSky;
    - eventuali **timeout** o eccezioni di rete;
  - al superamento di soglie predefinite (es. `failureRateThreshold`, `slidingWindowSize`), il Circuit Breaker entra in stato **OPEN**, bloccando temporaneamente nuove chiamate verso il servizio esterno.
- **Gestione delle eccezioni e comportamento del sistema**
  - quando il Circuit Breaker è in stato OPEN, le chiamate verso OpenSkyClient generano una `CallNotPermittedException` intercettata dal `FlightCollectionService`;
  - il servizio:
    - registra log dettagliati dell'evento, comprensivi di aeroporto, intervallo temporale e stato del Circuit Breaker;
    - evita di propagare l'errore verso l'esterno (si tratta di un processo schedulato, non associated a una richiesta sincrona del client);

        - può applicare logiche di *fallback* (ad esempio, saltare la raccolta per l'intervallo corrente, lasciando la responsabilità di recupero ai run successivi dello scheduler).

- **Configurazione parametrica**
    - i parametri del Circuit Breaker (soglie di errore, dimensione della finestra, tempi di *wait* nello stato HALF-OPEN) sono configurati tramite proprietà esterne, in modo da poter essere:
        - differenziati tra ambiente locale e ambiente Docker;
        - adeguati in funzione di eventuali limiti o politiche del provider OpenSky;
    - l'uso di configurazione esterna consente di adattare il comportamento del Circuit Breaker senza modifiche al codice, mantenendo invariate le interfacce interne.
- **Compatibilità con l'evoluzione futura**
    - la scelta di incapsulare il Circuit Breaker all'interno del layer di integrazione con OpenSky rende possibile l'eventuale introduzione futura di:
        - **retry** con *backoff* esponenziale;
        - **bulkhead** e altri pattern di resilienza;
        - integrazione con **metriche** e sistemi di osservabilità;
    - senza dover alterare le API pubbliche né i contratti REST esposti tramite API Gateway.

In questo modo, il sistema ottiene una protezione strutturata rispetto a malfunzionamenti o degradi del servizio esterno, mantenendo **stabili** sia i contratti REST verso i client, sia i contratti interni verso la pipeline Kafka e gli altri microservizi.

6. SCELTE PROGETTUALI E VALUTAZIONE ARCHITETTURALE

6.1. DECOMPOSIZIONE IN MICROSERVIZI E RESPONSABILITÀ DEI NUOVI COMPONENTI

La decomposizione adottata segue un principio di **separazione delle responsabilità** e di **allineamento ai sotto-domini funzionali** individuati nel sistema di flight monitoring.

Il sistema è articolato in cinque microservizi principali:

- **User Manager Service**
  Costituisce il *sistema di record* per gli utenti. La responsabilità è circoscritta a:
  - gestione del ciclo di vita degli utenti (registrazione, lettura, eventuali estensioni future);
  - esposizione di un servizio di **validazione dell'esistenza dell'utente** tramite gRPC.
    Non contiene logica di dominio legata ai voli o alle soglie, evitando accoppiamenti impropri.
- **Data Collector Service**
  È il microservizio che governa il sotto-dominio *raccolta dati e interessi sugli aeroporti*. Le sue responsabilità includono:
  - gestione degli **interessi utente–aeroporto**, comprensivi delle soglie `high_value` e `low_value`;
  - interrogazione periodica delle **OpenSky Network API** e persistenza dei flight records;
  - pubblicazione degli eventi di raccolta sul topic Kafka `to-alert-system`.

  La logica di *alerting* è intenzionalmente esclusa da questo servizio, che si limita a produrre dati consistenti e completi.
- **Alert System Service**
  Si colloca sul sotto-dominio *valutazione soglie e generazione di alert*. È progettato per:
  - consumare gli eventi relativi ai batch di raccolta voli;
  - calcolare le **metriche rilevanti** (numero di voli per finestra, direzione, interesse);
  - confrontare tali metriche con le soglie configurate in `user_airport_interest`;
  - generare eventi di violazione soglia sul topic `to-notifier`.
    Questo servizio incapsula la *business logic* più sensibile alla crescita del dominio (nuovi tipi di soglia, nuove regole di valutazione).

- **Alert Notifier Service**
  Opera nel sotto-dominio *notifica e comunicazione verso l'utente*. Le responsabilità sono:
  - consumare gli eventi di violazione soglia dal topic `to-notifier`;
  - comporre il contenuto delle email;
  - interagire con il **Mail Sender** (SMTP) per l'invio delle notifiche.

  È deliberatamente privo di logica di dominio sui voli; riceve eventi già "decisi" come degni di notifica.
- **API Gateway (NGINX)**
  Rappresenta il **perimetro di ingresso** del sistema. È responsabile di:
  - esporre un'unica superficie REST verso l'esterno;
  - instradare le richieste ai microservizi interni in base al path;
  - mascherare la topologia interna e le porte dei servizi.

Questa decomposizione presenta diversi vantaggi architetturali:

- ogni microservizio ha una **responsabilità ben delimitata**, evitando domini "onnicomprensivi" difficili da mantenere;
- l'evoluzione funzionale può avvenire in modo mirato, ad esempio:
  - estendendo l'Alert System con nuovi tipi di regole;
  - sostituendo il Notifier con un servizio che supporti canali multipli (email, SMS, push);
- la scalabilità può essere **differenziata per servizio**, ad esempio scalando il Data Collector in base al numero di aeroporti monitorati e l'Alert System/Notifier in base al numero di utenti e di notifiche generate.

Il rovescio della medaglia è un aumento di **complessità operativa** (più container, più configurazioni, dipendenza da Kafka, ecc.), gestito tramite Docker Compose e una chiara suddivisione degli artefatti di configurazione.

6.2. PATTERN DI COMUNICAZIONE SINCRONI E ASINCRONI (REST, GRPC, KAFKA)

La progettazione della comunicazione tra componenti segue un criterio di **adeguatezza al contesto**:

- *sincrono* quando è necessario un feedback immediato e deterministico;
- *asincrono* quando prevalgono requisiti di disaccoppiamento, scalabilità e tolleranza ai ritardi.

Nel sistema sono stati adottati tre principali pattern/protocolli:

- **REST/HTTP**
  Utilizzato per:
  - tutte le interazioni tra client esterni e sistema, tramite API Gateway;
  - alcune interazioni interne di configurazione e interrogazione, limitate a chiamate occasionali.

  È la scelta naturale per API pubbliche, grazie alla semplicità di utilizzo da parte di client eterogenei (browser, Postman, script, ecc.) e alla facile integrazione con tool esistenti.

- **gRPC**
  Riservato alla comunicazione **interna** tra Data Collector e User Manager per la **validazione dell'utente**.
  Le motivazioni principali sono:
  - necessità di una chiamata sincrona, ma stabile e ad *overhead* contenuto;
  - interfaccia fortemente tipizzata, con contratto chiaro (`userExists(email)` → boolean/risultato), che si presta a un'evoluzione controllata;
  - performance e efficienza maggiori rispetto a REST per chiamate frequenti o in scenari di crescita futura.

  L'uso di gRPC è confinato a un punto ben preciso del sistema, riducendo la complessità percepita dai consumatori esterni.

- **Kafka (messaggistica asincrona)**
  Rappresenta il **bus eventi** utilizzato per collegare Data Collector, Alert System e Alert Notifier.
  Il pattern adottato è quello degli **event-driven microservices**:
  - il Data Collector pubblica eventi su `to-alert-system` al termine dei batch di raccolta;
  - l'Alert System consuma tali eventi e, quando necessario, pubblica eventi di violazione soglia su `to-notifier`;
  - l'Alert Notifier consuma `to-notifier` e interagisce con l'infrastruttura SMTP.
    I benefici principali sono:
  - **disaccoppiamento temporale**: la raccolta dati non è bloccata dai tempi di valutazione delle soglie né dalla consegna delle email;
  - **resilienza**: in caso di momentanea indisponibilità di Alert System o Notifier, i messaggi rimangono nel broker e possono essere elaborati successivamente;
  - **scalabilità**: i consumer possono essere scalati orizzontalmente, suddividendo le partizioni dei topic.

La combinazione di questi pattern consente di allineare il **modello di interazione** alle caratteristiche di ciascun flusso:

- REST + gRPC per le **operazioni transazionali** e di configurazione;
- Kafka per i **flussi di dominio ad alto volume** e per l'alerting asincrono.

6.3. MOTIVAZIONI DELL'INTRODUZIONE DI API GATEWAY E MESSAGE BROKER

L'introduzione di un **API Gateway** e di un **message broker** non è un mero esercizio tecnologico, ma risponde a due esigenze architetturali distinte.

API Gateway (NGINX)

Le motivazioni principali sono:

- **Unificazione del punto di ingresso**
  Esporre un solo endpoint verso l'esterno consente:
  - di isolare la rete interna dei microservizi;
  - di nascondere nomi host e porte interne, permettendo modifiche trasparenti lato backend.
- **Separazione tra client e topologia interna**
  I client esterni ragionano in termini di risorse logiche (`/api/users`, `/api/interests`, `/api/flights`) senza doversi preoccupare di:
  - dove risiedono fisicamente i servizi;
  - come vengono orchestrati i container.
    Ciò facilita l'evoluzione futura (aggiunta di nuovi servizi, refactoring, migrazione verso orchestratori diversi da Docker Compose).
- **Punto naturale per cross-cutting concerns**
  Anche se nella versione corrente l'API Gateway è focalizzato sul routing, esso costituisce un **punto privilegiato** per:
  - logging centralizzato delle richieste;
  - introduzione di autenticazione e autorizzazione;
  - rate limiting e protezione da abusi;
  - eventuali trasformazioni di payload o header.

Message broker (Kafka)

L'introduzione di Kafka mira a:

- **Decouplare la pipeline di alerting** dal flusso di raccolta:
  - il Data Collector termina il proprio lavoro una volta pubblicato l'evento su `to-alert-system`;
  - l'Alert System può elaborare gli eventi con tempi e risorse proprie, senza influenzare la periodicità dello scheduler;

    - l’Alert Notifier può essere dimensionato indipendentemente, in funzione del volume di notifiche.

- **Gestire carichi variabili e picchi**
  La presenza di un broker consente di gestire in modo più controllato:
    - picchi di traffico verso OpenSky;
    - picchi nel numero di alert da generare in seguito a condizioni particolari (ad esempio, forte congestione o totale assenza di voli per alcuni aeroporti).
- **Abilitare estensioni future**
  Kafka fornisce un backbone naturale per:
    - aggiungere nuovi consumer (dashboard, servizi di analytics, storage a lungo termine) senza modificare i produttori;
    - introdurre pattern di elaborazione streaming avanzata (windowing, aggregazioni, ecc.) su pipeline dedicate.

La combinazione di API Gateway e Kafka produce un’architettura che è allo stesso tempo **ben protetta ai bordi** e **flessibile al centro**, predisposta a crescere in complessità senza perdere controllo sulla topologia e sul flusso dei dati.

6.4. IMPATTO DELLE ESTENSIONI SU RESILIENZA, SCALABILITÀ E MANUTENIBILITÀ

Le estensioni introdotte nella release corrente hanno un impatto significativo su diversi aspetti non funzionali dell’architettura.

Resilienza

- L’integrazione di **Resilience4j** come Circuit Breaker sulle chiamate a OpenSky:
    - limita il numero di tentativi contro un servizio esterno instabile;
    - protegge il Data Collector da condizioni di *fail-fast* ripetute;
    - consente l’adozione di **fallback controllati** (ad esempio, lista vuota di voli) che preservano la coerenza interna del sistema.
- L’uso di **Kafka** come buffer asincrono tra Data Collector, Alert System e Alert Notifier:
    - permette di assorbire temporanee indisponibilità dei consumer;
    - riduce la probabilità che un singolo componente degradi l’intera catena di elaborazione;
    - rende possibile riprocessare eventi in caso di failure, entro i limiti del modello di consegna adottato.

Scalabilità

- La decomposizione in microservizi permette di **scalare selettivamente**:
    - il Data Collector può essere replicato per gestire un numero maggiore di aeroporti o frequenze di polling più serrate;
    - l'Alert System e l'Alert Notifier possono essere scalati in base al numero di utenti, di interessi e di notifiche generate;
    - Kafka può essere configurato con più partizioni per distribuire il carico di consumo.
- L'API Gateway consente di aggiungere **nuovi endpoint e nuovi servizi** senza impattare il modo in cui i client raggiungono il sistema, semplificando l'introduzione di capacità verticali aggiuntive (nuove aree funzionali, nuove versioni di API).

Manutenibilità

- La chiara **assegnazione delle responsabilità** per microservizio facilita:
    - la localizzazione dei problemi (ad esempio, bug nella valutazione soglie confinati all'Alert System);
    - l'evoluzione del codice con impatto limitato sul resto del sistema;
    - la possibilità di ruotare team diversi su servizi differenti, evitando conflitti sull'intero codebase.
- Il modello dati mantiene una **struttura minimale e coerente**:
    - l'estensione di `user_airport_interest` con `high_value` e `low_value` non introduce nuove entità, ma arricchisce un concetto già esistente;
    - la logica di alerting è esterna al modello di persistenza, riducendo il rischio di vincoli rigidi e difficili da modificare in futuro.
- L'adozione di contratti chiari (REST, gRPC, JSON su Kafka) e di un API Gateway centralizzato contribuisce a migliorare la **comprensibilità globale** dell'architettura, rendendo più agevole:
    - l'onboarding di nuovi sviluppatori;
    - la stesura di documentazione tecnica allineata al comportamento reale del sistema.

Il costo principale di queste scelte è l'incremento della **complessità distribuita** (più servizi, più protocolli, più configurazioni), mitigato attraverso:

- l'uso sistematico di Docker Compose per orchestrare l'ambiente;
- la strutturazione rigorosa dei file di configurazione (application properties, configurazioni NGINX, configurazioni Kafka);
- la chiara documentazione dei flussi e dei contratti, che rende espliciti i punti di integrazione e le dipendenze tra i componenti.

7. ASPETTI NON FUNZIONALI E OPERATIVI

7.1. OSSERVABILITÀ, LOGGING E TRACCIAMENTO LUNGO LA PIPELINE DI NOTIFICA

L'osservabilità del sistema si basa principalmente su **logging strutturato** nei microservizi applicativi e, dove opportuno, sull'utilizzo di **identificatori di correlazione** che permettono di tracciare i flussi end-to-end lungo la pipeline asincrona.

Nel **Data Collector Service** i punti di log principali riguardano:

- avvio e terminazione dei job schedulati di raccolta:
  - identificazione della finestra temporale (`windowStart`, windowEnd`);`
  - elenco degli aeroporti oggetto della raccolta;
- invocazioni verso le API di OpenSky:
  - URL chiamato, parametri principali, stato del **Circuit Breaker** (CLOSED/OPEN/HALF_OPEN);
  - codice di risposta HTTP o eccezione riscontrata;
- esito della fase di persistenza:
  - numero di flight records inseriti per ogni aeroporto e finestra;
  - eventuali errori di scrittura sul Data DB;
- pubblicazione su Kafka:
  - identificativo del batch (`batchId`);
  - topic di destinazione (`to-alert-system`);
  - dimensione logica del messaggio (numero di aeroporti, intervallo di riferimento).

Questi log permettono di verificare rapidamente se la catena *raccolta → persistenza → pubblicazione evento* si è chiusa correttamente o si è interrotta in uno dei passaggi.

Nel **Alert System Service** i log sono focalizzati su:

- consumo dei messaggi dal topic `to-alert-system`:
  - `batchId`, finestra temporale, aeroporti coinvolti;
  - offset e partizione, utili per diagnosi e re-processing controllato;
- interrogazioni del Data DB per il recupero delle soglie:
  - numero di interessi trovati per ciascun aeroporto;
  - presenza o assenza di soglie (`high_value`, `low_value`) per ciascun interesse;

- valutazione delle soglie:
    - valori calcolati delle metriche (ad esempio `currentFlightsCount`);
    - condizioni di superamento (`HIGH`/`LOW`) effettivamente verificate;
- pubblicazione su `to-notifier`:
    - numero di eventi di notifica generati per batch;
    - email destinatario e codice aeroporto (anonimizzati o parzialmente mascherati, se necessario).

Nel **Alert Notifier Service** l'attenzione è rivolta a:

- consumo dei messaggi da `to-notifier`:
    - email destinatario, aeroporto, tipo di violazione (`HIGH`/`LOW`);
- interazione con il server SMTP:
    - esito della chiamata (`success`, `temporary failure`, `permanent failure`);
    - eventuali codici di errore restituiti dal server di posta;
- gestione di eventuali eccezioni nell'invio:
    - dettaglio dell'errore;
    - presenza o meno di eventuali retry (qualora previsti).

L'uso di un **formato coerente di logging** (ad esempio JSON strutturato) e la presenza di campi comuni (come `batchId`, `userEmail`, `airportCode`, `evaluationWindow`) consente, anche in assenza di un sistema di tracciamento distribuito, di correlare gli eventi tra microservizi analizzando i log.

Dal lato **API Gateway**, NGINX può essere configurato per:

- registrare ogni richiesta in ingresso con:
    - path, metodo HTTP, codice di risposta, tempo di risposta complessivo;
    - indirizzo IP del client ed eventuali header di tracciamento (es. `X-Correlation-Id`);
- fornire una vista aggregata del traffico verso i microservizi interni, utile per individuare pattern anomali (es. picchi di richieste verso le API di configurazione delle soglie o di interrogazione dei voli).

La combinazione di logging nei microservizi, log dell'API Gateway e informazioni di partizione/offset su Kafka fornisce una base solida per l'osservabilità del sistema e per l'analisi puntuale di fault o comportamenti anomali lungo l'intera pipeline di notifica.

7.2. GESTIONE DEGLI ERRORI E DEI CASI LIMITE (SOGLIE MANCANTI, EMAIL NON RECAPITABILE, ECC.)

La gestione degli errori è strutturata lungo tre dimensioni principali: **validazione dei dati in ingresso**, **comportamento in presenza di dipendenze esterne non disponibili** e **trattamento dei casi limite funzionali**.

Per quanto riguarda la **validazione dei dati in ingresso**:

- le API del **Data Collector** che gestiscono gli interessi e le soglie verificano:
    - la presenza dei campi obbligatori (`userEmail`, `airportCode`);
    - la validità formale dell'email e del codice aeroporto;
    - la coerenza dei valori di soglia:
        - highValue e lowValue non negativi;
        - in caso di presenza di entrambe le soglie, vincolo highValue > lowValue;
- in caso di violazione dei vincoli, viene restituito un errore 400 Bad Request con payload descrittivo, e nessuna modifica viene applicata al Data DB;
- il vincolo `UNIQUE(user_email, airport_id)` nel Data DB previene la creazione di interessi duplicati; eventuali violazioni vengono intercettate e mappate su errori applicativi (`409 Conflict` o analoghi).

Dal punto di vista delle **dipendenze esterne**, i casi principali sono:

- **OpenSky Network API**:
    - errori di rete, timeout o errori HTTP 5xx sono gestiti dal **Circuit Breaker Resilience4j**, che può portare all'attivazione del fallback (lista di voli vuota);
    - in stato di Circuit Breaker *OPEN*, le chiamate verso OpenSky vengono bloccate, evitando di sovraccaricare il servizio esterno e di bloccare il thread dello scheduler;
    - il Data Collector registra log esaustivi e prosegue con il flusso interno, trattando l'assenza di dati come "nessun volo disponibile" per l'intervallo considerato;
- **Server SMTP** utilizzato dall'Alert Notifier:
    - errori di connessione o di autenticazione vengono intercettati e loggati con livello adeguato (`WARN`/`ERROR`);
    - in caso di errori permanenti (es. indirizzo email non valido o rifiutato dal server), il messaggio di notifica viene considerato non recapitabile; il sistema non tenta di reinviarne una copia indefinita, per evitare loop;
    - in scenari evolutivi, si può introdurre una coda di retry o un *dead-letter queue* per le notifiche fallite, mantenendo inalterato il modello dati corrente.

Sul piano dei **casi limite funzionali**:

- **soglie mancanti**:
  - è possibile creare un interesse senza soglie (`highValue = null`, `lowValue = null`);
  - in questo caso l'interesse partecipa alla raccolta dati ma non genera mai alert, in quanto l'Alert System interpreta la mancanza di soglia come "nessuna regola attiva" per quel tipo di condizione;
  - se è configurata una sola soglia (ad esempio solo `highValue`), viene valutata esclusivamente la condizione corrispondente;
- **concorrenza nella modifica degli interessi**:
  - se un interesse viene cancellato o modificato tra la fase di raccolta e la fase di valutazione delle soglie, l'Alert System può non trovare più il record atteso o trovarlo aggiornato;
  - la logica di valutazione può gestire in modo robusto il caso di interesse assente (ad esempio, ignorando l'evento e loggando la condizione), evitando inconsistenze o errori;
- **assenza completa di voli in un intervallo valido**:
  - l'assenza di flight records per una certa finestra temporale è trattata come un caso normale;
  - la metrica `currentFlightsCount` assume valore 0 e viene confrontata con eventuali soglie `lowValue`, determinando eventualmente una violazione di soglia bassa.

Per gli errori interni non direttamente riconducibili a dipendenze esterne (ad esempio eccezioni inattese nella logica di valutazione delle soglie o nella composizione del contenuto dell'email), i microservizi catturano le eccezioni e producono log dettagliati, evitando, per quanto possibile, che un singolo evento malformato comprometta l'intero flusso.

7.3. IMPATTO PRESTAZIONALE DELLE NUOVE COMPONENTI E POSSIBILI COLLI DI BOTTIGLIA

L'estensione architetturale con **Alert System**, **Alert Notifier**, **Kafka** e **Circuit Breaker** introduce nuove opportunità, ma anche nuovi potenziali **colli di bottiglia** che devono essere considerati in un'ottica prestazionale.

Dal punto di vista del **Data Collector**:

- l'introduzione del Circuit Breaker non aggiunge overhead significativo per singola chiamata verso OpenSky, in quanto la logica di monitoraggio è leggera rispetto al costo delle chiamate HTTP esterne;

- il numero di chiamate verso OpenSky cresce con:
    - il numero di aeroporti monitorati;
    - la frequenza di esecuzione dello scheduler;
- in scenari di carico elevato è opportuno dimensionare:
    - il numero di thread dedicati alla raccolta;
    - le politiche di timeout del client HTTP;
    - gli indici nel Data DB (in particolare su `flight_records.airport_id`, `flight_records.collected_at`) per mantenere efficienti le operazioni di inserimento e aggregazione.

Per quanto riguarda **Kafka**:

- il volume di messaggi su `to-alert-system` e `to-notifier` dipende:
    - dal numero di batch di raccolta prodotti in unità di tempo;
    - dal numero di interessi attivi e dalla frequenza con cui vengono violate le soglie;
- se il numero di eventi cresce in modo significativo, possibili colli di bottiglia possono manifestarsi:
    - nel throughput del broker Kafka (numero di messaggi al secondo gestibili);
    - nella capacità di consumo dell'Alert System e dell'Alert Notifier;
- la progettazione attuale consente di intervenire su:
    - **partizionamento dei topic**, per distribuire il carico su più istanze dei consumer;
    - **parallelismo interno** dei microservizi consumer, in modo da elaborare più messaggi in parallelo mantenendo l'ordine quando necessario.

Nel **Alert System**, il costo principale è legato a:

- interrogazioni del Data DB per recuperare interessi e soglie;
- calcolo delle metriche (conteggio dei voli) per ogni finestra e per ogni interesse.

Per evitare che queste operazioni diventino critiche, è utile:

- assicurare la presenza di indici sulle colonne più utilizzate in WHERE e JOIN (ad esempio `user_airport_interest.airport_id`, `user_airport_interest.user_email`, `flight_records.airport_id`, `flight_records.collected_at`);
- limitare dimensione e numerosità delle finestre temporali, bilanciando granularità e costo di aggregazione.

Nel **Alert Notifier**, il collo di bottiglia potenziale è l'**invio delle email**:

- ogni notifica comporta una chiamata sincrona al server SMTP;
- in caso di volumi molto elevati, questa fase può rallentare l'elaborazione complessiva;
- possibili mitigazioni, anche in evoluzione futura, includono:
    - l'uso di invii asincroni o *bulk*;
    - la configurazione di un pool di connessioni verso il server SMTP;
    - l'introduzione di meccanismi di rate limiting per evitare di saturare il servizio di posta.

L'**API Gateway** introduce un overhead minimo in termini di latenza, soprattutto se configurato in modo leggero (reverse proxy semplice senza funzionalità di ispezione profonda del traffico). Tuttavia, essendo il punto di ingresso unico, può diventare un **single point of congestion** in caso di:

- picchi intensi di richieste in lettura sui dati di volo;
- uso improprio delle API (ad esempio poll eccessivamente frequenti da parte di client mal configurati).

In questi scenari, la scalabilità orizzontale dell'API Gateway (ad esempio tramite più istanze dietro un load balancer) e una corretta configurazione delle timeouts e delle connessioni keep-alive consentono di mantenere livelli di servizio adeguati.

Nel complesso, le nuove componenti sono state introdotte con la massima attenzione a preservare:

- **isolamento dei colli di bottiglia** (ad esempio confinando il peso dell'invio email all'Alert Notifier);
- **scalabilità indipendente** dei microservizi latentemente più critici (Data Collector, Alert System, Alert Notifier);
- **capacità di intervento mirato** su Kafka, database e sottosistema di posta, senza richiedere modifiche invasive al modello dei dati o ai contratti esposti.

8. INQUADRAMENTO NEL CICLO EVOLUTIVO DEL SISTEMA

8.1. COMPONENTI E SEZIONI DELLA VERSIONE PRECEDENTE ANCORA PIENAMENTE VALIDE

La versione precedente della documentazione rimane **pienamente valida** per tutti gli aspetti che descrivono il *nucleo funzionale* e il *contesto di dominio* del sistema di flight monitoring, in particolare per:

- **Obiettivi di business e scenario d'uso**
  La descrizione delle esigenze informative (monitoraggio dei voli per aeroporti selezionati dagli utenti, analisi di arrivi/partenze, storicizzazione dei dati di volo) continua a rappresentare correttamente le finalità del sistema. Le estensioni introdotte (soglie, pipeline di notifica) non modificano il *perimetro concettuale* del dominio, ma lo arricchiscono con capacità reattive.
- **Modello concettuale di base**
  Il modello che individua le entità fondamentali:
  - *Utente* (identificato da email),
  - *Aeroporto* (codice, metadati descrittivi),
  - *Interesse utente–aeroporto*,
  - *Flight record* (volo associato a un aeroporto in un certo istante),
    rimane valido come fondamento del dominio. Le soglie di interesse sono una *specializzazione* della relazione di interesse, non un cambio di paradigma.
- **Responsabilità core di User Manager e Data Collector**
  La documentazione preesistente che inquadra:
  - lo **User Manager** come sistema di record degli utenti, con API REST per la gestione delle anagrafiche e servizio gRPC di validazione;
  - il **Data Collector** come servizio di raccolta periodica dei voli da OpenSky e di gestione degli interessi utente–aeroporto;
    resta corretta. Le nuove funzionalità si sovrappongono a queste responsabilità senza stravolgerle.
- **Schema dei database per utenti e dati di volo**
  Le sezioni che descrivono:
  - il database logico **User DB** con la tabella `users`;
  - il database logico **Data DB** con le tabelle `airports` e `flight_records`, non richiedono correzioni; l'estensione riguarda esclusivamente l'arricchimento di `user_airport_interest` con gli attributi di soglia, mantenendo invariata la struttura delle altre entità.

- **Flussi di base già documentati**
  I flussi principali già descritti nella versione precedente rimangono **funzionalmente validi**:
    - registrazione utente tramite User Manager;
    - registrazione di un interesse utente–aeroporto con validazione via gRPC;
    - raccolta periodica dei voli (scheduler + OpenSky) e persistenza dei flight records;
    - interrogazione dei dati di volo tramite le API del Data Collector.

  Le estensioni introdotte agiscono *a valle* o *in parallelo* a questi flussi, senza modificarne la semantica di base.

L'intera parte della documentazione precedente che presenta il **sistema come soluzione minimale** (due microservizi, integrazione con OpenSky, persistenza in PostgreSQL) continua quindi a costituire un riferimento corretto per comprendere il funzionamento del "core" dell'applicazione.

8.2. COMPONENTI E SEZIONI INTEGRATE O SOSTITUITE DALLA VERSIONE CORRENTE

Alcune sezioni della documentazione precedente devono essere considerate **integrate** o **superate** in virtù delle estensioni architetturali e funzionali introdotte nella versione corrente.

I principali punti di impatto sono:

- **Architettura complessiva del sistema**
  La descrizione che presenta un'architettura composta esclusivamente da:
    - User Manager;
    - Data Collector;
    - PostgreSQL;
    - OpenSky Network,

  deve essere interpretata come un **baseline architetturale**. La versione corrente introduce:
    - l'**API Gateway (NGINX)** come punto di ingresso centralizzato;
    - il **Kafka Broker** come message backbone;
    - l'**Alert System** e l'**Alert Notifier** come nuovi microservizi dedicati alla valutazione delle soglie e all'invio delle notifiche;
    - il **Mail Sender (SMTP)** come infrastruttura di consegna.
      Il nuovo diagramma architetturale esteso sostituisce, per l'analisi globale del sistema, la precedente rappresentazione, che rimane comunque utile per comprendere il primo stadio evolutivo.

- **Descrizione della tabella `user_airport_interest`**
  Le sezioni che descrivono `user_airport_interest` come semplice relazione binaria utente–aeroporto risultano **incomplete** rispetto alla versione corrente.
  Rimane valido:
    - il concetto di associazione tra utente e aeroporto;
    - il vincolo di unicità sulla coppia.
      Risulta invece aggiornata:
    - la struttura logica, arricchita dagli attributi `high_value` e `low_value`;
    - la semantica applicativa, che ora include la configurazione di politiche di alert per ogni interesse.
- **Flusso di raccolta e uso dei dati di volo**
  La descrizione precedente si fermava alla:
    - raccolta dei voli da OpenSky;
    - persistenza dei dati nel Data DB;
    - interrogazione sincrona da parte dei client.

  Con la versione corrente:
    - la raccolta innesca la pubblicazione di un evento su `to-alert-system`;
    - i dati persistiti vengono utilizzati dall'**Alert System** per calcolare metriche rispetto alle soglie;
    - gli alert vengono propagati su `to-notifier` per l'invio email.
      Le sezioni preesistenti relative alla *sola* interrogazione sincrona rimangono corrette, ma la vista complessiva sul **ciclo di vita dei dati** deve ora includere la pipeline event-driven.
- **Aspetti non funzionali iniziali**
  Eventuali valutazioni su resilienza, scalabilità e manutenibilità basate unicamente su due microservizi e un servizio esterno devono essere considerate **parziali** rispetto allo stato attuale.
  La versione corrente:
    - introduce pattern di resilienza (Circuit Breaker con Resilience4j);
    - riposiziona alcuni potenziali colli di bottiglia (ad esempio sull'invio massivo di email o sulla capacità del broker Kafka);
    - abilita nuove strategie di scaling selettivo.
- **Diagrammi di sequenza centrati esclusivamente su User Manager e Data Collector**
  I diagrammi che descrivono:
    - registrazione utente;
    - registrazione interesse con validazione gRPC;

- raccolta periodica dei voli;
- interrogazione dei dati di volo;

restano validi *nel perimetro che rappresentano*, ma non coprono:

- la valutazione delle soglie;
- la propagazione degli eventi su Kafka;
- la generazione e invio delle notifiche.

I nuovi diagrammi di sequenza dedicati alla configurazione delle soglie, alla pipeline asincrona e alla gestione delle failure verso OpenSky completano questa prospettiva.

La documentazione corrente deve quindi essere letta come **strato evolutivo** che integra e, dove necessario, sostituisce specifiche porzioni della descrizione originaria, preservando la coerenza dell'insieme.

8.3. RIFERIMENTI INCROCIATI A DIAGRAMMI E CAPITOLI DELLA DOCUMENTAZIONE DELLA VERSIONE PRECEDENTE

Per disporre di una visione completa e stratificata del sistema, è opportuno utilizzare la presente documentazione **in combinazione** con la versione precedente, stabilendo alcuni riferimenti incrociati espliciti.

- **Descrizione del dominio e degli attori principali**
  La parte della documentazione precedente che introduce:
  - il contesto del monitoraggio voli;
  - le figure di utente finale e client applicativo;
  - il ruolo di OpenSky Network come fonte dati esterna;

  costituisce il riferimento primario per la comprensione del *problema* e del *perimetro funzionale di base*. Le sezioni introduttive lì presenti possono essere richiamate per accompagnare la lettura dell'odierna visione architetturale estesa.
- **Diagramma architetturale di base**
  Il **diagramma architetturale originario** (User Manager + Data Collector + PostgreSQL + OpenSky) può essere utilizzato:
  - come *snapshot* della prima release;
  - come base di confronto per leggere il nuovo diagramma architetturale esteso, che aggiunge API Gateway, Kafka, Alert System, Alert Notifier e server SMTP.

  L'affiancamento dei due diagrammi evidenzia visivamente:
  - quali componenti sono rimasti invariati;
  - quali sono stati introdotti in un secondo momento;

- come si è passati da un modello essenzialmente sincrono a un modello ibrido sincrono/asincrono.

- **Diagramma ER originario**
  Il **diagramma ER della versione precedente** fornisce la vista di base su:
  - `users` nel User DB;
  - `airports`, `user_airport_interest`, `flight_records` nel Data DB, prima dell'introduzione degli attributi di soglia.

  L'ER aggiornato della presente documentazione può essere letto come **refinement** del precedente, con `user_airport_interest` arricchito da `high_value` e `low_value`. Il confronto tra i due diagrammi facilita la comprensione dell'impatto limitato (ma significativo) della modifica sul modello fisico.

- **Diagrammi di sequenza dei flussi core**
  I diagrammi che illustrano:
  - la registrazione utente;
  - la registrazione di un interesse;
  - la raccolta periodica dei voli;
  - l'interrogazione dei dati di volo,

  rimangono il riferimento primario per i **flussi core sincroni**.

  I nuovi diagrammi introdotti nella versione corrente (configurazione/aggiornamento soglie, pipeline asincrona, gestione delle failure verso OpenSky) vanno letti **in continuità** con quelli preesistenti, completando il quadro delle interazioni:
  - i diagrammi di configurazione delle soglie estendono il flusso di registrazione dell'interesse;
  - il diagramma della pipeline Kafka completa il ciclo *raccolta* → *valutazione* → *notifica*;
  - il diagramma sul Circuit Breaker dettaglia la robustezza della fase di integrazione con OpenSky.

- **Capitoli dedicati ad API e modello dati**
  Le sezioni della documentazione precedente che descrivono:
  - il modello delle API per la registrazione utente, la gestione degli interessi e l'interrogazione dei voli;
  - la struttura delle tabelle `users`, `airports`, `user_airport_interest`, `flight_records` prima dell'estensione;

  sono da considerarsi *documentazione di base* alla quale la presente versione aggiunge:
  - l'estensione dei payload con `highValue` e `lowValue`;

- la descrizione dei contratti di messaggio Kafka (`to-alert-system`, `to-notifier`);
- la mappatura aggiornata verso il modello JPA.

Nel complesso, la documentazione precedente e quella corrente costituiscono **due livelli di dettaglio complementari**: la prima definisce le fondamenta del sistema, la seconda ne descrive l'evoluzione verso un'architettura più ricca, resiliente e reattiva, preservando la leggibilità e la tracciabilità delle scelte nel tempo.

9. CONCLUSIONI E SVILUPPI FUTURI

9.1. SINTESI DELLE ESTENSIONI FUNZIONALI E ARCHITETTURALI INTRODOTTE

La versione corrente del sistema di *flight monitoring* rappresenta un'evoluzione significativa rispetto alla precedente, sia sul piano funzionale sia su quello architetturale.

Sul piano **funzionale** sono stati introdotti i seguenti elementi chiave:

- **Gestione delle soglie di interesse**
  L'entità *user–airport interest* è stata estesa con le soglie `high_value` e `low_value`, permettendo di associare a ciascun interesse una *politica di monitoraggio quantitativa* sul numero di voli rilevati.
  Questo abilita scenari quali:
  - notifica quando il numero di voli supera una soglia alta (situazione di congestione);
  - notifica quando il numero di voli scende sotto una soglia bassa (situazione di scarsa operatività).
- **Pipeline di notifica asincrona basata su eventi**
  Il sistema non è più limitato alla raccolta e interrogazione sincrona dei dati di volo, ma è in grado di:
  - valutare le soglie in maniera automatica e periodica;
  - generare eventi di *threshold breach*;
  - notificare gli utenti via email quando si verificano condizioni critiche sul traffico aereo degli aeroporti monitorati.
- **Arricchimento del modello dati**
  Il *Data DB* è stato esteso in modo mirato:
  - la tabella `user_airport_interest` integra le colonne `high_value` e `low_value`, mantenendo il vincolo di unicità sulla coppia `(user_email, airport_id)` e preservando la compatibilità con il modello precedente;
  - la semantica del collegamento logico con gli utenti rimane invariata, ma è ora funzionalmente più ricca.

Sul piano **architetturale** le principali estensioni sono:

- **Introduzione dell'API Gateway (NGINX)**
  Tutte le API sono ora esposte attraverso un punto di ingresso unificato, che:
  - centralizza l'accesso ai microservizi `User Manager` e `Data Collector`;
  - maschera la topologia interna del sistema rispetto ai client;
  - apre la strada all'introduzione di ulteriori *cross-cutting concerns* (autenticazione, rate limiting, logging avanzato).

- **Introduzione del message broker Kafka**
  La comunicazione tra Data Collector, Alert System e Alert Notifier è ora mediata da:
  - topic `to-alert-system`, che rappresenta il completamento di un batch di raccolta voli;
  - topic `to-notifier`, che trasferisce eventi di *breach* già valutati

  Questo passaggio codifica un vero e proprio paradigma **event-driven**, disaccoppiando temporalmente e strutturalmente i microservizi.
- **Introduzione dei microservizi Alert System e Alert Notifier**
  La logica di alerting è stata estratta in due componenti dedicati:
  - **Alert System**: si occupa di recuperare le soglie, calcolare le metriche e determinare se le condizioni di superamento sono verificate;
  - **Alert Notifier**: si occupa di tradurre gli eventi di violazione soglia in notifiche email concrete, integrandosi con un server SMTP.
- **Adozione del Circuit Breaker verso OpenSky tramite Resilience4j**
  Le chiamate alle OpenSky Network API sono ora protette da un Circuit Breaker che:
  - monitora il tasso di errore e i timeout;
  - previene richieste ripetute verso un servizio in stato di degrado;
  - abilita fallback controllati, preservando la stabilità del Data Collector.

Nel complesso, il sistema passa da una soluzione *minimal* centrata su due microservizi e un solo flusso sincrono a una piattaforma più articolata, capace di trasformare i dati raccolti in **alert operativi** con proprietà migliorate in termini di resilienza, scalabilità e modularità.

9.2. VALUTAZIONE CRITICA DELLE SCELTE PROGETTUALI ADOTTATE

Le scelte progettuali adottate nella versione corrente presentano diversi punti di forza, accompagnati da alcuni aspetti che richiedono attenzione in ottica evolutiva e operativa.

Dal punto di vista **architetturale**, la decomposizione in microservizi con responsabilità ben definite può essere valutata positivamente:

- la separazione tra *raccolta dati* (Data Collector), *gestione utenti* (User Manager), *valutazione soglie* (Alert System) e *notifica* (Alert Notifier) riduce l'accoppiamento e rende più chiaro il perimetro di ciascun servizio;
- l'introduzione dell'API Gateway e del message broker conferma una direzione verso un'architettura *cloud-native ready*, dove ingressi, comunicazioni interne e integrazioni esterne sono trattati con pattern consolidati.

Sul piano delle **tecnologie di comunicazione**, la combinazione REST + gRPC + Kafka è coerente con i casi d'uso:

- REST è utilizzato nel ruolo per cui è maggiormente adatto: interfaccia pubblica verso client eterogenei;

- gRPC è confinato a una funzione specifica (validazione utente), mantenendo il vantaggio di un'interfaccia tipizzata e performante senza diffondere complessità non necessaria;
- Kafka viene impiegato dove il disaccoppiamento e la resilienza al carico sono più critici: la pipeline di alerting.

Queste scelte migliorano **scalabilità e resilienza**, al costo di un aumento della complessità complessiva:

- occorre gestire più processi (microservizi), più canali di comunicazione (HTTP, gRPC, Kafka) e una maggiore quantità di configurazioni (Docker Compose, application properties, NGINX, Kafka/ZooKeeper);
- la diagnosi dei problemi richiede una maggiore disciplina nella raccolta e correlazione dei log, oltre a strumenti di osservabilità più strutturati nelle release successive.

Dal punto di vista del **modello dati**, l'estensione di `user_airport_interest` con `high_value` e `low_value` è una scelta mirata e poco invasiva:

- evita l'introduzione di nuove entità dedicate alle policy di soglia, mantenendo la configurazione direttamente associata alla relazione utente–aeroporto;
- permette una gestione relativamente semplice delle operazioni CRUD, anche in un contesto di evoluzione successiva del dominio.

Al tempo stesso, la concentrazione di logiche di alerting sul solo asse "numero di voli per finestra temporale" rappresenta un **compromesso consapevole**:

- l'implementazione risulta più semplice e lineare;
- l'estensibilità verso altre metriche (ritardi, cancellazioni, trend su lunghe finestre) richiederà, in futuro, una raffinata evoluzione dell'Alert System e del modello dati.

L'adozione del **Circuit Breaker** verso OpenSky rappresenta una scelta allineata alle buone pratiche di resilienza, ma introduce alcuni aspetti da monitorare:

- la corretta taratura dei parametri (soglie, window size, tempo di attesa in stato OPEN) è cruciale per evitare sia attivazioni troppo frequenti sia ritardi eccessivi nel ripristino;
- l'assenza di un sistema strutturato di metriche e dashboard dedicate rende più complessa, allo stato attuale, la verifica empirica del comportamento del Circuit Breaker in scenari reali.

Nel complesso, le scelte effettuate privilegiano **chiarezza del disegno**, **allineamento ai pattern moderni** e **predisposizione all'evoluzione**, a fronte di un aumento controllato della complessità infrastrutturale che potrà essere governato con l'introduzione di strumenti di orchestrazione, osservabilità e automazione più avanzati nelle versioni successive.

9.3. POSSIBILI DIREZIONI DI EVOLUZIONE DEL SISTEMA DI FLIGHT MONITORING

Le possibili evoluzioni possono essere organizzate lungo tre direttrici complementari: **potenziamento funzionale e analitico**, **consolidamento architetturale–infrastrutturale** e **scalabilità dei dati e dei carichi**. Alcune direttrici evolutive delineate in precedenza sono già state recepite in questa release (in particolare l'introduzione di un message broker, di un API Gateway, della pipeline event–driven per l'alerting e dei meccanismi di resilienza verso OpenSky), mentre altre sono state intenzionalmente lasciate fuori dallo scope corrente e costituiscono il naturale proseguimento della roadmap. Il fuoco di questo paragrafo è quindi sugli sviluppi che rimangono aperti, sia fra quelli già identificati in precedenza, sia fra quelli resi oggi possibili dalle nuove scelte architetturali.

Un primo asse di evoluzione riguarda il **rafforzamento del livello funzionale e analitico**, facendo leva sulle strutture già presenti:

- introduzione di **API analitiche dedicate**, supportate da viste o proiezioni ottimizzate per la lettura (ad esempio statistiche di ritardo, indicatori di performance per aeroporto, confronti tra gruppi di aeroporti);
- adozione del pattern **CQRS** per separare in modo netto il modello di scrittura (registrazione utenti, interessi, ingestione voli) dal modello di lettura, servito da *read model* specifici per reporting e dashboard;
- utilizzo sistematico dell'**outbox pattern** per propagare in modo affidabile gli eventi di dominio dal perimetro transazionale ai componenti di lettura o ai sistemi analitici, mantenendo allineato lo stato tra database e flussi di eventi;
- arricchimento del **modello di alerting**, estendendo le soglie oltre il semplice numero di voli per finestra temporale per includere, ad esempio, ritardi medi, tassi di cancellazione, variazioni anomale dei volumi o combinazioni di condizioni articolate.

Un secondo asse di evoluzione è di natura **architetturale–infrastrutturale**, orientato a incrementare disaccoppiamento, resilienza e osservabilità:

- generalizzazione del paradigma **event–driven** oltre la sola pipeline di alerting, con la pubblicazione sistematica di eventi di dominio (`UserRegistered`, `InterestCreated`, `FlightsCollected`, ecc.) e l'eventuale introduzione di uno *schema registry* per governare l'evoluzione dei payload;
- definizione di **saga** per modellare processi distribuiti complessi, con step multipli e azioni compensative, ad esempio per orchestrare workflow che coinvolgano gestione utenti, interessi, moduli analitici o futuri componenti di billing e compliance;
- evoluzione del ruolo dell'**API Gateway** in vero punto di enforcement delle policy, incaricato di gestire autenticazione e autorizzazione, applicare politiche di rate limiting e protezione, centralizzare logging, routing e versioning delle API;
- consolidamento di una **stack di osservabilità completa**, ad esempio **Prometheus** per la raccolta e l'esposizione delle metriche tecniche, integrato con dashboard dedicate (es. Grafana), log strutturati e meccanismi di *tracing* distribuito per

ottenere visibilità end–to–end sulle chiamate, sulle interazioni fra microservizi, sulla pipeline di ingestione e sulla pipeline di notifica;

- adozione di un **orchestratore di container come Kubernetes**, in grado di abilitare scaling automatico, strategie di rilascio evolute (rolling update, canary release) e una gestione più strutturata di configurazioni e segreti.

Una terza direttrice riguarda infine la **scalabilità dei dati, la gestione di carichi elevati e la capacità analitica avanzata**:

- ottimizzazione della **persistence layer** e introduzione di politiche di *retention* differenziate, con indici mirati, partizionamento delle tabelle dei flight records e meccanismi di archiviazione per i dati storici meno consultati;
- integrazione con **piattaforme analitiche esterne** (data warehouse, data lake, motori di stream processing) alimentate tramite esportazioni periodiche o flussi di eventi, per supportare analisi storiche profonde, reporting direzionale e studi sul traffico aereo in ottica di pianificazione;
- valutazione di database e motori specializzati per **serie temporali e stream**, in grado di gestire con maggiore efficienza query temporali complesse e aggregazioni su finestre scorrevoli o tumbling;
- integrazione di **modelli predittivi e meccanismi di rilevazione anomalie** (ad esempio per la previsione dei ritardi, dei volumi di traffico o per l'individuazione di pattern anomali), i cui output possano alimentare l'Alert System con segnali più sofisticati rispetto alle sole soglie statiche, abilitando forme di alerting proattivo.

Queste direttrici si innestano in modo naturale sulle fondamenta già consolidate, sfruttando l'architettura event–driven, l'API Gateway, il message broker e i meccanismi di resilienza esistenti per trasformare progressivamente il sistema in una piattaforma completa di monitoring, analisi e integrazione con ecosistemi distribuiti complessi.