
DIPARTIMENTO DI INGEGNERIA ELETTRICA, ELETTRONICA E INFORMATICA

CORSO DI LAUREA MAGISTRALE IN: INGEGNERIA INFORMATICA

# DISTRIBUTED SYSTEMS AND BIG DATA

Homework 3

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 trasformare un'architettura a microservizi già consolidata in una soluzione **cloud-native**, eseguibile su **Kubernetes** e dotata di un sottosistema di **white-box monitoring** basato su **Prometheus** e metriche applicative esposte tramite **Micrometer**.

Gli obiettivi specifici possono essere articolati come segue:

- **Industrializzare il deployment del sistema su Kubernetes.**
  Il sistema, in precedenza eseguito tramite containerizzazione e orchestrazione locale, viene ora descritto e gestito attraverso manifest Kubernetes (Deployment, Service, ConfigMap, Secret, ecc.), rendendo più esplicita la separazione tra *codice applicativo* e *configurazione di runtime* e abilitando scenari di deployment ripetibili, riproducibili e automatizzabili.
- **Introdurre un sottosistema di monitoring cross-cutting.**
  Tutti i microservizi applicativi critici (in particolare **Data Collector**, **Alert System** e **Alert Notifier**) vengono dotati di **metriche applicative** e **metriche tecniche** esposte sull'endpoint `/actuator/prometheus`.
  L'obiettivo è consentire a un operatore tecnico di analizzare in modo oggettivo:
  - il comportamento del client verso il servizio esterno OpenSky (numero di richieste, errori, fallback, tempi di risposta);
  - il funzionamento della pipeline di notifica asincrona (eventi elaborati, notifiche generate, tempi di valutazione e di processing);
  - l'affidabilità e le prestazioni del sottosistema di invio email (email inviate, errori, eventuale rate-limiting da parte del server SMTP).
- **Mantenere inalterata la semantica funzionale del sistema.**
  La versione corrente non introduce nuove funzionalità di dominio rispetto alla versione precedente, ma **preserva integralmente**:
  - il modello concettuale di utenti, aeroporti e interessi;
  - il flusso di raccolta periodica dei voli tramite OpenSky;
  - la logica di valutazione delle soglie di interesse e di notifica tramite email.
    L'evoluzione avviene quindi sul piano dell'**osservabilità** e del **deployment**, minimizzando l'impatto sulla logica business già validata.
- **Abilitare analisi operative e diagnosi guidate dai dati.**
  Le metriche introdotte consentono di:
  - monitorare l'affidabilità del servizio esterno OpenSky;
  - individuare colli di bottiglia nella pipeline Kafka-based;

    - misurare l’effetto di eventuali problemi infrastrutturali (latenze, errori di rete, indisponibilità del mail server) sui tempi di elaborazione e notifica.
      Ciò rende il sistema più adatto a contesti in cui sono richieste **osservabilità continua**, **capacità di troubleshooting** e **monitoraggio proattivo**.

- **Preparare il sistema a futuri scenari di scalabilità e gestione operativa.**
  La combinazione di Kubernetes, Prometheus e metriche applicative fornisce la base per:
    - scalare selettivamente i microservizi più critici;
    - integrare in futuro ulteriori strumenti di observability (es. sistemi di alerting, dashboard, log aggregation);
    - supportare pratiche operative moderne, come *continuous delivery*, *canary release* e *capacity planning*.

La versione corrente del sistema si propone quindi come una **evoluzione infrastrutturale e di osservabilità** di una piattaforma di flight monitoring già completa dal punto di vista funzionale, con l’obiettivo di avvicinarla agli standard architetturali tipici delle soluzioni distribuite moderne.

## 1.2. PERIMETRO DELLA DOCUMENTAZIONE E RELAZIONE CON LE VERSIONI PRECEDENTE

Il presente documento descrive le scelte progettuali, architetturali e tecnologiche relative alla **versione corrente** del sistema di flight monitoring, con particolare attenzione all’adozione di **Kubernetes** come piattaforma di esecuzione e di **Prometheus** come componente centrale del sottosistema di monitoring.

Il perimetro coperto comprende in particolare:

- la descrizione ad alto livello dell’architettura nella versione corrente, con evidenza dei microservizi applicativi, dei componenti infrastrutturali e dei nuovi elementi introdotti per il monitoring (*Prometheus, metriche Micrometer, endpoint /actuator/prometheus*);
- il modello delle metriche applicative e tecniche esposte dai microservizi, nonché gli obiettivi di osservabilità perseguiti;
- le principali interazioni tra i componenti del sistema, con particolare attenzione agli scenari monitorati (pipeline di notifica asincrona e scraping delle metriche);
- le API esposte (REST, gRPC, endpoint di metrics) e i relativi punti di osservazione;
- le scelte di deployment su Kubernetes e le modalità di integrazione del sottosistema di monitoring;
- le implicazioni non funzionali di tali scelte, in termini di scalabilità, resilienza, diagnosi dei problemi e controllo operativo.

La descrizione **non ripropone in dettaglio**:

- l’intero dominio applicativo;

- il modello dati completo e lo schema logico dei database;
- la definizione esaustiva di tutte le operazioni REST e gRPC già consolidate nelle versioni precedenti.

Tali aspetti si considerano **già documentati** e vengono, quando necessario, richiamati come *contesto* o *baseline*:

- la **prima versione** descrive l'architettura iniziale del sistema di flight monitoring, centrata sui microservizi *User Manager* e *Data Collector* e sui database relazionali per la gestione degli utenti, degli aeroporti e dei voli;
- la **seconda versione** estende tale architettura introducendo i microservizi *Alert System* e *Alert Notifier*, l'infrastruttura Kafka-based per la pipeline di notifica asincrona, l'API Gateway NGINX e il mail server per l'invio delle notifiche.

Il presente documento assume quindi la **seconda versione** come **baseline architetturale** di riferimento, ovvero come ultima fotografia completa del sistema dal punto di vista funzionale, e si concentra sull'analisi delle **estensioni introdotte dalla versione corrente**:

- porting e razionalizzazione del deployment su Kubernetes;
- introduzione di Prometheus e delle metriche Micrometer nei microservizi critici;
- definizione degli scenari di monitoring e delle metriche a supporto dell'analisi operativa.

Quando opportuno, vengono forniti rimandi alle sezioni e ai diagrammi delle versioni precedenti, al fine di evitare duplicazioni e mantenere una chiara distinzione tra le diverse fasi evolutive del sistema.

## 1.3. CONTESTO TECNOLOGICO E VINCOLI PROGETTUALI

La versione corrente del sistema si colloca in un contesto tecnologico caratterizzato da:

- **architetture a microservizi** per la suddivisione del dominio in componenti indipendenti, scalabili e distribuibili separatamente;
- **comunicazioni sincrone e asincrone** (REST/HTTP, gRPC, messaging Kafka) per integrare microservizi applicativi e servizi esterni;
- **containerizzazione e orchestrazione** come standard de facto per il deployment di sistemi distribuiti;
- necessità di **white-box monitoring** e *observability* per garantire controllo operativo e diagnosi tempestiva in ambienti distribuiti.

In questo quadro, le principali tecnologie adottate e i vincoli progettuali sono i seguenti.

**Architettura applicativa e stack tecnologico.**
I microservizi applicativi (*User Manager*, *Data Collector*, *Alert System*, *Alert Notifier*) sono sviluppati con **Spring Boot**, sfruttando:

- il supporto nativo per la creazione di servizi REST;

- l'integrazione con database relazionali tramite driver JDBC e repository di astrazione;
- il supporto per **gRPC** nella validazione degli utenti;
- l'integrazione con **Kafka** per la gestione della pipeline di notifica asincrona;
- l'integrazione con **Actuator** e **Micrometer** per l'esposizione delle metriche in formato Prometheus.

La persistenza dei dati è affidata a **PostgreSQL**, organizzato in due database logici (*User DB* e *Data DB*), mentre la messaggistica asincrona è gestita da un **Kafka broker** supportato da **ZooKeeper** e affiancato da **Kafka UI** per scopi di ispezione e amministrazione.

**Piattaforma di deployment.**
L'intera soluzione è progettata per essere eseguita su un cluster **Kubernetes**, con l'obiettivo di:

- standardizzare il modello di deployment tramite risorse dichiarative (Deployment, Service, ConfigMap, Secret, ecc.);
- isolare logicamente i componenti in un **namespace dedicato**;
- preparare il sistema a scenari di scalabilità e gestione operativa tipici di ambienti cloud-native.

Come ambiente di riferimento viene utilizzato un cluster **kind** (Kubernetes in Docker), che introduce vincoli di:

- esecuzione tipicamente *single-node*;
- risorse computazionali limitate rispetto a cluster multi-nodo;
- necessità di una configurazione delle risorse Kubernetes che tenga conto di un equilibrio tra complessità e reali esigenze del sistema.

**Sottosistema di monitoring e metriche.**
Per il monitoring viene adottata la coppia **Micrometer + Prometheus**, con i seguenti vincoli e linee guida:

- le metriche vengono esposte dai microservizi tramite l'endpoint `/actuator/prometheus`, in formato testuale compatibile con Prometheus;
- l'attività di raccolta delle metriche è realizzata tramite **Prometheus** in modalità *pull*, utilizzando lo scraping periodico dei target configurati;
- le metriche sono organizzate in:
    - **metriche applicative**, specifiche del dominio del flight monitoring (ad esempio, richieste verso OpenSky, eventi di notifica generati, email inviate o fallite);

    - **metriche tecniche**, esposte dagli actuator (stato di health, metriche JVM, thread, memory, HTTP) e da Micrometer (eventuali metriche di sistema e runtime);
- l'overhead introdotto dalla raccolta delle metriche deve rimanere **contenuto** e compatibile con l'ambiente di esecuzione, evitando un eccessivo impatto sulle prestazioni dei microservizi.

**Vincoli di progettazione architetturale.**
Le scelte progettuali sono guidate dai seguenti vincoli:

- **separazione netta tra concern funzionali e concern di monitoring**: la logica di dominio non deve dipendere dal dettaglio di Prometheus, che rimane un consumatore di metriche esposte in modo standard;
- **mantenimento del disaccoppiamento tra microservizi**: l'introduzione delle metriche non deve introdurre accoppiamento stretto tra componenti o violare la natura asincrona della pipeline Kafka-based;
- **aderenza a pratiche cloud-native**: configurazione esterna del sistema tramite variabili d'ambiente, ConfigMap e Secret; definizione completa dei componenti applicativi e infrastrutturali tramite manifest Kubernetes; osservabilità realizzata tramite strumenti diffusi nell'ecosistema (Prometheus, actuator, Micrometer);
- **portabilità** della soluzione verso altri ambienti Kubernetes, evitando dipendenze non necessarie da caratteristiche specifiche dell'ambiente di sviluppo.

All'interno di questo contesto, la versione corrente del sistema di flight monitoring rappresenta un passo verso una gestione più matura di deployment e osservabilità, pur preservando l'impianto logico e funzionale consolidato nelle versioni precedenti.

# 2. VISIONE D'INSIEME DELL'ARCHITETTURA CON MONITORING SU KUBERNETES

## 2.1. DESCRIZIONE SINTETICA DEL SISTEMA NELLA VERSIONE CORRENTE

Nella versione corrente, il sistema di *flight monitoring* è organizzato come un insieme di **microservizi Spring Boot** e **componenti infrastrutturali** eseguiti all'interno di un **cluster Kubernetes** dedicato (namespace applicativo). L'obiettivo principale è monitorare in modo continuativo il traffico aereo relativo a specifici aeroporti di interesse, valutare eventuali situazioni di superamento soglia e notificare gli utenti tramite email, mantenendo al contempo un elevato livello di osservabilità interna.

L'architettura applicativa è centrata su quattro microservizi:

- **User Manager Service**, responsabile della gestione degli utenti registrati e delle informazioni ad essi associate;
- **Data Collector Service**, incaricato di raccogliere periodicamente i voli dagli endpoint dell'API OpenSky per gli aeroporti di interesse, persistere i dati nel database e alimentare la pipeline event-driven;
- **Alert System Service**, che consuma gli eventi generati dal Data Collector, applica la logica di valutazione delle soglie di interesse e produce eventi di notifica quando vengono rilevate condizioni di superamento soglia;
- **Alert Notifier Service**, che consuma le notifiche generate dall'Alert System e coordina l'invio di email agli utenti interessati.

Questi microservizi sono esposti verso l'esterno in maniera controllata attraverso un **API Gateway NGINX**, che funge da reverse proxy e punto di ingresso unificato per i client HTTP. Le comunicazioni interne sono realizzate tramite:

- chiamate **REST/HTTP** per le API applicative;
- chiamate **gRPC** per la validazione degli utenti da parte del Data Collector;
- messaggistica **event-driven su Kafka** per la propagazione asincrona degli eventi di aggiornamento finestra voli e delle notifiche di superamento soglia.

Dal punto di vista dei dati, il sistema si appoggia a un'istanza **PostgreSQL** che ospita due database logici distinti: uno dedicato ai dati utente (*User DB*) e uno ai dati di dominio relativi a voli, aeroporti e interessi (*Data DB*).

Il **sottosistema di monitoring** è strutturato attorno a **Prometheus**, configurato per effettuare lo scraping periodico delle metriche esposte dai microservizi tramite **Micrometer** e **Spring Boot Actuator**. In particolare, i servizi **Data Collector**, **Alert System** e **Alert Notifier** espongono l'endpoint `/actuator/prometheus`, che fornisce metriche sia tecniche sia applicative. L'insieme di queste metriche consente a un operatore di analizzare:

- il comportamento del client verso l'API esterna OpenSky;
- la salute e le prestazioni della pipeline Kafka-based;
- la qualità e l'affidabilità del sottosistema di notifica via email.

Completano l'architettura i servizi esterni:

- **OpenSky Network API**, utilizzata come fonte dati per i voli in arrivo e in partenza dagli aeroporti monitorati;
- **Mail server SMTP**, utilizzato dall'Alert Notifier per l'invio delle email di alert;
- **Kafka UI**, impiegato come interfaccia di ispezione e amministrazione dei topic Kafka.

Nel complesso, la versione corrente mantiene invariata la logica di dominio definita nelle versioni precedenti, introducendo però un livello superiore di **cloud-native deployment** e **osservabilità operativa** grazie alla combinazione di Kubernetes, Prometheus e metriche applicative dedicate.

## 2.2. MICROSERVIZI E COMPONENTI INFRASTRUTTURALI

L'architettura logica del sistema nella versione corrente può essere vista come la composizione di due insiemi di componenti:

- **microservizi applicativi**, che implementano la logica di dominio e le API esposte ai client;
- **componenti infrastrutturali**, che forniscono servizi di supporto indispensabili per l'esecuzione, la persistenza, la messaggistica e il monitoring.

Tutti questi elementi sono eseguiti come *pod* e *service* Kubernetes nel medesimo namespace applicativo, con i microservizi che comunicano tra loro tramite i corrispondenti Service Kubernetes e gli elementi infrastrutturali che fungono da backbone per dati, eventi e osservabilità.

Di seguito si riportano i principali componenti e il loro ruolo nell'architettura.

### 2.2.1. USER MANAGER SERVICE

Lo **User Manager Service** è il microservizio incaricato della **gestione degli utenti** del sistema di flight monitoring. Le sue responsabilità comprendono:

- registrazione e aggiornamento dei dati degli utenti;
- eventuale gestione delle credenziali o degli identificativi necessari per l'accesso alle funzionalità applicative;
- messa a disposizione di un **endpoint gRPC** per la validazione dell'esistenza di un utente, utilizzato dal Data Collector.

Il servizio espone API **REST/HTTP** per le operazioni di gestione utente, tipicamente accessibili tramite l'API Gateway. A livello di persistenza, utilizza il database **User DB** su PostgreSQL, dove memorizza le entità utente e le informazioni correlate.

Nella versione corrente, lo User Manager **non espone metriche applicative verso Prometheus** e non è stato strumentato con Micrometer per la raccolta di indicatori di dominio. Il suo corretto funzionamento viene monitorato principalmente tramite:

- i **log applicativi** generati dal microservizio;

- le **probe Kubernetes** (liveness/readiness) associate al pod, che consentono di verificare lo stato di salute del processo;
- l'osservazione indiretta del traffico attraverso l'API Gateway e del carico sul database **User DB**.

In questo modo lo User Manager mantiene un profilo funzionale stabile e ben delimitato, rimanendo comunque integrato nel perimetro di controllo operativo del cluster, pur senza introdurre metriche Prometheus dedicate come negli altri microservizi critici di dominio.

### 2.2.2. DATA COLLECTOR SERVICE E OpenSky CLIENT

Il **Data Collector Service** costituisce il cuore del flusso di raccolta dati. Le sue responsabilità principali sono:

- memorizzare le **preferenze di interesse** degli utenti in termini di aeroporti da monitorare;
- eseguire una **raccolta periodica** dei voli in arrivo e in partenza dagli aeroporti di interesse, interrogando l'API esterna OpenSky entro finestre temporali definite;
- persistere i voli raccolti nel database **Data DB**;
- aggregare i dati per finestra temporale (snapshot per aeroporto) e generare eventi destinati alla pipeline di notifica.

Al suo interno, il servizio integra:

- uno **scheduler** che innesca periodicamente il processo di raccolta;
- un **OpenSky Client** HTTP che comunica con l'API OpenSky, tipicamente protetto da meccanismi di **Circuit Breaker** per gestire fault temporanei o degradazioni del servizio esterno;
- un **gRPC client** verso lo User Manager per validare l'esistenza di un utente associato a un interesse;
- un **Kafka producer** che pubblica, sul topic dedicato, gli eventi di tipo *FlightCollectionWindowUpdateEvent* destinati all'Alert System.

Il Data Collector utilizza metriche applicative esposte tramite Micrometer per misurare:

- il numero di richieste inviate all'API OpenSky;
- il numero di errori e fallback attivati in caso di problemi con il servizio esterno;
- la durata delle chiamate verso OpenSky;
- il numero di eventi di aggiornamento finestra prodotti e pubblicati su Kafka.

Queste metriche, insieme a quelle tecniche di base, sono rese disponibili tramite `/actuator/prometheus` e consentono di analizzare in dettaglio il comportamento del flusso di raccolta dati.

### 2.2.3. ALERT SYSTEM SERVICE

L'**Alert System Service** è il microservizio responsabile della **valutazione delle soglie di interesse** definite dagli utenti sugli aeroporti monitorati. Riceve in ingresso gli eventi generati dal Data Collector tramite **Kafka** e, per ciascuna finestra temporale, esegue:

- l'elaborazione degli snapshot aggregati per aeroporto;
- il calcolo del numero totale di voli (arrivi + partenze) in ciascuna finestra;
- il confronto tra il valore osservato e le soglie *high*/*low* configurate per gli interessi degli utenti.

Quando viene rilevata una **violazione di soglia**, il servizio costruisce uno o più eventi di tipo *ThresholdBreachNotificationEvent* e li pubblica sul topic Kafka dedicato alle notifiche, destinato all'Alert Notifier.

Il servizio accede al database **Data DB** per recuperare gli interessi configurati con soglie, sfruttando repository dedicati. Sul piano delle metriche, l'Alert System espone indicatori come:

- il numero di eventi di finestra elaborati;
- il numero di notifiche generate;
- la durata media e l'ultima durata delle fasi di valutazione.

Questi dati permettono di valutare la **responsività** e la **stabilità** del processo di valutazione delle soglie, oltre a fornire un'indicazione del volume di notifiche generate in diversi scenari operativi.

### 2.2.4. ALERT NOTIFIER SERVICE

L'**Alert Notifier Service** rappresenta l'ultimo anello della pipeline di notifica. Le sue funzioni principali sono:

- consumare dal topic Kafka le notifiche di superamento soglia generate dall'Alert System;
- trasformare ciascuna notifica in una **email** indirizzata all'utente interessato;
- interagire con il **server SMTP** configurato per l'invio dei messaggi.

Il servizio utilizza un componente dedicato, l'**Email Notification Service**, responsabile della costruzione del contenuto dell'email (oggetto e corpo) e dell'eventuale applicazione di politiche di *throttling* o rate limiting lato applicativo, per evitare un eccessivo numero di invii in un intervallo temporale ristretto.

Dal punto di vista del monitoring, l'Alert Notifier espone metriche sia a livello di listener Kafka sia a livello di invio email, tra cui:

- numero di notifiche consumate;
- errori in fase di deserializzazione o processing dell'evento;
- numero di email inviate con successo;

- errori di invio (es. problemi SMTP o rate limiting lato provider);
- durata degli invii email.

Queste metriche consentono di monitorare l'affidabilità del processo di notifica e di individuare rapidamente eventuali criticità nella comunicazione con il mail server.

### 2.2.5. API GATEWAY (NGINX)

L'**API Gateway**, realizzato tramite **NGINX**, svolge il ruolo di **reverse proxy** e punto di ingresso unico per i client esterni che accedono alle API del sistema. Le sue responsabilità includono:

- l'instradamento delle richieste HTTP verso i microservizi appropriati (ad esempio, User Manager per le operazioni sugli utenti, Data Collector per la gestione degli interessi e l'interrogazione dei voli);
- la normalizzazione dei path esposti, presentando un set coerente di endpoint al client;
- l'eventuale applicazione di regole di base per la gestione del traffico (ad esempio, configurazioni di timeout lato proxy).

L'API Gateway è esposto come Service Kubernetes accessibile dall'esterno del cluster e costituisce l'interfaccia principale utilizzata da strumenti di test, client REST o altre applicazioni che intendono interagire con il sistema di flight monitoring.

### 2.2.6. KAFKA BROKER, ZOOKEEPER E KAKFA UI

L'infrastruttura di **messaggistica asincrona** è basata su **Apache Kafka**, supportato da un'istanza **ZooKeeper** per la gestione della configurazione e del coordinamento del cluster Kafka.

In particolare:

- il **Kafka broker** ospita i topic utilizzati dal sistema:
    - un topic per la trasmissione degli eventi di aggiornamento finestra voli dal Data Collector all'Alert System;
    - un topic per le notifiche di superamento soglia dall'Alert System all'Alert Notifier;
- **ZooKeeper** gestisce le informazioni di coordinamento necessarie al broker Kafka;
- **Kafka UI** fornisce una **interfaccia web** che consente agli operatori di ispezionare i topic, visualizzare i messaggi, monitorare i consumer group e verificare lo stato dei broker.

Sebbene Kafka UI non faccia parte del percorso applicativo di produzione, risulta estremamente utile in fase di validazione, test e diagnosi dei problemi, permettendo di verificare in tempo reale il corretto funzionamento della pipeline event-driven.

### 2.2.7. DATABASE POSTGRESQL (USER DB E DATA DB)

La persistenza dei dati è affidata a un'istanza **PostgreSQL** eseguita come pod all'interno del cluster Kubernetes. Tale istanza ospita due database logici distinti:

- **User DB**, utilizzato dallo User Manager per memorizzare le informazioni relative agli utenti;

- **Data DB**, utilizzato dal Data Collector e dall'Alert System per gestire:
    - aeroporti,
    - interessi degli utenti verso specifici aeroporti, eventualmente con soglie di interesse,
    - record di volo raccolti dall'API OpenSky.

I microservizi accedono a PostgreSQL tramite driver JDBC e layer di persistenza basati su repository, mantenendo una netta separazione tra logica applicativa e dettaglio di storage. La configurazione di connessione (host, porta, nome del database, credenziali) è esternalizzata tramite configurazioni Kubernetes (ConfigMap e Secret) per favorire la portabilità tra ambienti e facilitare la gestione operativa.

### 2.2.8. PROMETHEUS E SOTTOSISTEMA DI MONITORING

**Prometheus** costituisce il componente centrale del sottosistema di **white-box monitoring**. È eseguito come Deployment Kubernetes e configurato per effettuare lo **scraping periodico** delle metriche esposte dai microservizi e, ove opportuno, da altri componenti rilevanti.

Le metriche sono esposte in formato compatibile Prometheus tramite l'endpoint `/actuator/prometheus` dei microservizi Spring Boot, reso disponibile da **Micrometer** in combinazione con Spring Boot Actuator. In particolare, vengono raccolte:

- metriche **applicative**, che descrivono il comportamento specifico del dominio:
    - richieste verso OpenSky e relative durate;
    - numero di eventi elaborati nella pipeline;
    - numero di notifiche generate e email inviate o fallite;
- metriche **tecniche**, come:
    - metriche JVM (heap, garbage collection, thread);
    - metriche HTTP (conteggio richieste, codici di risposta, tempi di risposta);
    - metriche di health e stato di availability dei microservizi.

Prometheus memorizza le serie temporali generate dallo scraping e le rende interrogabili tramite il proprio motore di query, permettendo a un operatore di analizzare in dettaglio l'andamento del sistema nel tempo e di definire eventualmente regole di alerting esterne basate sui valori delle metriche raccolte.

## 2.3. DIAGRAMMA ARCHITETTURALE – ARCHITETTURA COMPLESSIVA DEL SISTEMA SU KUBERNETES CON SOTTOSISTEMA DI MONITORING PROMETHEUS

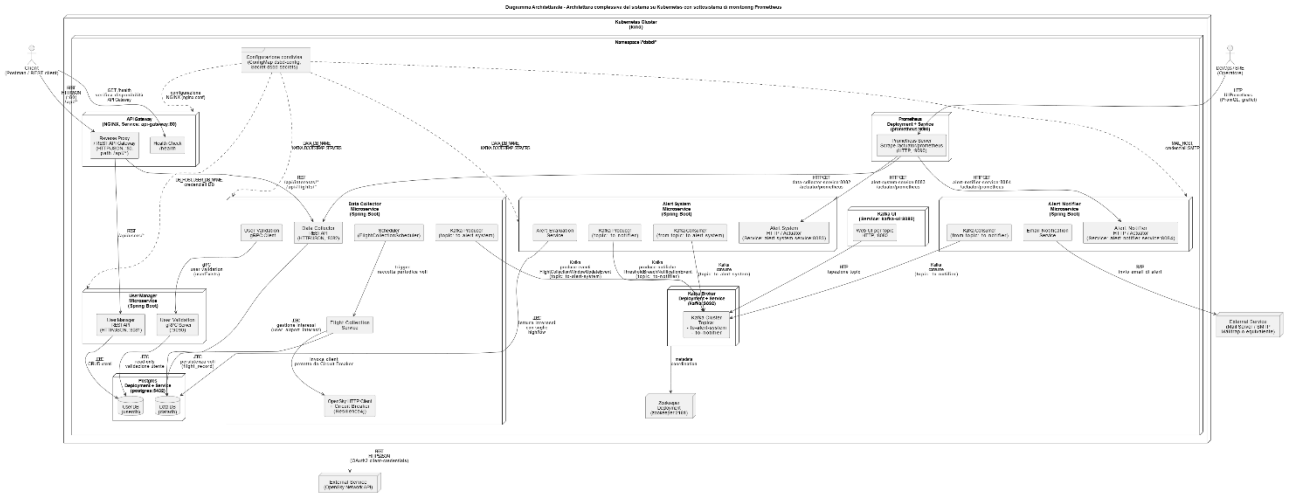
Il diagramma architetturale associato a questa sezione rappresenta la **vista complessiva** del sistema di flight monitoring nella versione corrente, mostrando:

- i **microservizi applicativi** (*User Manager*, *Data Collector*, *Alert System*, *Alert Notifier*), eseguiti come pod nel namespace applicativo del cluster Kubernetes;
- i **componenti infrastrutturali** interni al cluster:
  - API Gateway NGINX;
  - Kafka broker, ZooKeeper e Kafka UI;
  - PostgreSQL con i database *User DB* e *Data DB*;
  - Prometheus come componente di monitoring;
- i **servizi esterni**:
  - OpenSky Network API, come sorgente dati per i voli;
  - mail server SMTP, come endpoint di consegna delle notifiche email;
  - client esterni (ad esempio strumenti REST) che accedono al sistema tramite l'API Gateway.

Nel diagramma sono evidenziati:

- i **confini logici** del cluster Kubernetes e del namespace dell'applicazione;
- i **flussi di comunicazione** principali:
  - REST/HTTP tra client esterni e API Gateway, e tra API Gateway e microservizi;
  - gRPC tra Data Collector e User Manager per la validazione utenti;
  - flussi Kafka tra Data Collector, Alert System e Alert Notifier;
  - interazioni JDBC tra i microservizi e PostgreSQL;
  - chiamate HTTP verso OpenSky;
  - invio di email tramite SMTP verso il mail server;
- i **flussi di metriche**, in cui Prometheus effettua lo scraping degli endpoint `/actuator/prometheus` esposti dai microservizi, raccogliendo sia metriche tecniche sia metriche applicative.

La presenza congiunta dei microservizi, dell'infrastruttura di messaging, del database relazionale, del gateway e di Prometheus fornisce una visione chiara della **stratificazione architetturale**: livello di dominio, livello di integrazione/eventi, livello di persistenza e livello di osservabilità.



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

Rispetto alla versione precedente, l'architettura della versione corrente mantiene **invariata** la struttura dei microservizi e dei principali flussi funzionali, mentre introduce una serie di **evoluzioni architetturali** orientate al deployment e al monitoring.

Le principali differenze possono essere sintetizzate come segue:

- **Transizione dal deployment su host Docker al deployment su Kubernetes.** Nella versione precedente, i microservizi e i componenti infrastrutturali erano eseguiti tramite container in un singolo host orchestrato con strumenti di livello inferiore. Nella versione corrente, tali elementi sono definiti come risorse Kubernetes (Deployment, Service, ConfigMap, Secret, ecc.), eseguite in un namespace dedicato. Questo consente di:
    - gestire in maniera dichiarativa lo stato desiderato del sistema;
    - separare in modo netto la configurazione applicativa dall'infrastruttura sottostante;
    - predisporre il sistema a scenari di scalabilità orizzontale e gestione operativa più avanzata.
- **Introduzione di Prometheus come componente di monitoring centralizzato.** La versione precedente non disponeva di un sottosistema strutturato di white-box monitoring; il controllo del comportamento dei microservizi era affidato principalmente a log e verifiche manuali. Nella versione corrente, Prometheus diventa un componente di prima classe dell'architettura, incaricato di:
    - raccogliere metriche tecniche e applicative dai microservizi;
    - consentire interrogazioni analitiche sulle serie temporali generate;
    - fornire una base per eventuali regole di alerting esterne.

- **Strumentazione dei microservizi con metriche applicative dedicate.**
  I microservizi critici sono stati modificati per integrare metriche specifiche tramite Micrometer, in particolare:
    - nel Data Collector, per la raccolta delle statistiche sulle chiamate ad OpenSky e sulla pubblicazione degli eventi su Kafka;
    - nell'Alert System, per tracciare l'elaborazione degli eventi di finestra e la generazione delle notifiche;
    - nell'Alert Notifier e nell'Email Notification Service, per monitorare il consumo delle notifiche, l'invio delle email e gli eventuali errori SMTP.
      Queste metriche si aggiungono alle metriche tecniche esposte dagli actuator, fornendo una visione approfondita del comportamento interno del sistema.
- **Riorganizzazione della configurazione applicativa tramite risorse Kubernetes.**
  La configurazione dei microservizi (variabili d'ambiente, endpoint esterni, parametri di connessione a database e broker) viene ora gestita tramite **ConfigMap** e **Secret**, in coerenza con le pratiche cloud-native. Questo approccio rende più semplice la gestione di diversi ambienti di esecuzione e riduce il rischio di configurazioni incoerenti tra componenti.

Dal punto di vista del dominio, il sistema conserva la stessa semantica funzionale delle versioni precedenti: monitoraggio di aeroporti di interesse, raccolta periodica dei voli, valutazione delle soglie e invio di notifiche agli utenti. L'evoluzione introdotta dalla versione corrente è quindi focalizzata sulla **maturazione architetturale** del sistema, rendendolo più vicino agli standard operativi richiesti per sistemi distribuiti osservabili e gestibili in ambienti Kubernetes.

# 3. MODELLO DELLE METRICHE E OBIETTIVI DI OSSERVABILITÀ

## 3.1. REQUISITI DI WHITE-BOX MOTORING E AMBITO DI COPERTURA DEI MICROSERVIZI

L'obiettivo del *white-box monitoring* in questa versione del sistema è fornire una visibilità puntuale sul comportamento interno della **pipeline di raccolta, valutazione e notifica** degli eventi di traffico aereo, mantenendo al contempo un livello di complessità controllato.

Gli obiettivi principali possono essere sintetizzati come segue:

- **Monitorare in modo esplicito le dipendenze esterne critiche**, in particolare:
  - le chiamate HTTP verso l'API di OpenSky (endpoint di arrivi e partenze);
  - l'invio delle email verso il Mail Server SMTP.
- **Osservare la pipeline event-driven end-to-end**:
  - produzione degli eventi di aggiornamento finestra nel *Data Collector*;
  - consumo ed elaborazione degli eventi nel *Alert System*;
  - consumo delle notifiche e invio email nel *Alert Notifier*.
- **Correlare i comportamenti dei microservizi nel tempo** tramite:
  - indicatori di throughput (quante richieste, quante valutazioni, quante notifiche);
  - indicatori di latenza (durata delle chiamate a OpenSky, durata delle valutazioni, durata degli invii email);
  - indicatori di errore (errori verso OpenSky, errori di processing notifiche, errori SMTP, eventi limitati dal rate-limit).

Il perimetro di copertura del monitoring applicativo è focalizzato sui microservizi direttamente coinvolti nella pipeline di notifica:

- **Data Collector Service**, con un'attenzione specifica all'*OpenSky Client* e al Circuit Breaker;
- **Alert System Service**, come punto di valutazione delle soglie sulle finestre temporali di traffico;
- **Alert Notifier Service**, che consuma le notifiche da Kafka e delega l'invio delle email;
- **Email Notification Service**, considerato come sottocomponente logico dell'Alert Notifier ma dotato di metriche dedicate.

Per questi componenti sono stati introdotti **metriche applicative esplicite** tramite Micrometer e integrate nel flusso esposto su `/actuator/prometheus`.
Gli altri microservizi (ad esempio *User Manager* o l'API Gateway NGINX) restano nel perimetro del monitoring principalmente attraverso le **metriche tecniche** (HTTP, JVM,

sistema) e tramite le funzionalità standard fornite dagli actuator o dal layer infrastrutturale, senza introdurre metriche applicative aggiuntive in questa versione.

L'insieme dei requisiti copre sia l'esigenza di individuare rapidamente malfunzionamenti locali (per esempio una degradazione strutturale delle chiamate a OpenSky o problemi SMTP) sia quella di seguire il comportamento complessivo della pipeline di business lungo il tempo, supportando scenari di diagnosi e analisi a posteriori.

### 3.2. TIPOLOGIE DI METRICHE ESPOSTE (COUNTER, GAUGE, LABEL DI SERVIZIO E DI NODO)

Il modello di osservabilità adottato si basa su **Micrometer** come *facade* di metriche dentro i microservizi Spring Boot e su **Prometheus** come backend di raccolta. Le tipologie principali di metriche utilizzate sono:

- **Counter**:
    - metriche monotone che aumentano nel tempo, utilizzate per conteggiare eventi:
        - numero di richieste verso OpenSky;
        - numero di errori di chiamata verso OpenSky;
        - numero di attivazioni del fallback del Circuit Breaker;
        - numero di valutazioni di finestre eseguite dall'Alert System;
        - numero di notifiche generate verso Kafka;
        - numero di notifiche consumate dall'Alert Notifier;
        - numero di errori nel processing delle notifiche;
        - numero di email inviate con successo;
        - numero di errori SMTP o eccezioni inattese;
        - numero di invii bloccati dal rate-limit del provider SMTP.
- **Gauge**:
    - metriche che rappresentano un **valore istantaneo** aggiornato nel tempo, utilizzate in particolare per:
        - durata in millisecondi dell'ultima chiamata OpenSky effettuata dal Data Collector;
        - durata in millisecondi dell'ultima valutazione effettuata dall'Alert System;
        - durata in millisecondi dell'ultimo processing di una notifica da parte dell'Alert Notifier;
        - durata in millisecondi dell'ultimo invio email da parte dell'Email Notification Service.

Oltre alla distinzione *counter/gauge*, le metriche sono arricchite da un set coerente di **etichette (labels)**, gestite in parte da Micrometer e in parte dalla configurazione di Prometheus e Kubernetes:

- **Label applicative configurate negli actuator**:
    - `service`: identifica il microservizio che espone la metrica (ad esempio `data-collector-service`, `alert-system-service`, `alert-notifier-service`), impostata tramite `management.metrics.tags.service`;
    - `node`: identifica il nodo logico o fisico su cui gira il pod (valorizzato tramite variabile d'ambiente `NODE_NAME`), utile per distinguere performance per nodo in scenari con più repliche.
- **Label infrastrutturali fornite da Prometheus/Kubernetes**:
    - `job`: identifica lo *scrape job* (ad esempio `data-collector`, `alert-system`, `alert-notifier`);
    - `instance`: identifica l'endpoint concreto (`<pod-ip>:8082`, `<pod-ip>:8083`, ecc.);
    - ulteriori label di contesto come `namespace`, `pod`, `service` derivano dal discovery Kubernetes e permettono di filtrare e aggregare le metriche per componente infrastrutturale.

Le metriche applicative introdotte in questa versione **non introducono label ad alta cardinalità** (come email utente, codice aeroporto o ID notifica), che rimangono confinati nel dominio dei log. Ciò consente di preservare un modello di osservabilità **stabile e scalabile**, evitando esplosioni di cardinalità nella serie temporali di Prometheus.

### 3.3. METRICHE APPLICATIVE DEL DATA COLLECTOR E DELL'OpenSky CLIENT

Nel *Data Collector Service* il focus del monitoring applicativo è l'interazione con l'API di OpenSky, incapsulata nel componente `OpenSkyClient`. Le metriche sono raccolte dalla classe `OpenSkyClientMetrics` e rappresentano il comportamento combinato di **Circuit Breaker**, **client HTTP** e finestra temporale di raccolta voli.

Le metriche principali sono:

- `opensky_requests_total` (*Counter*)
  Conta il numero totale di richieste effettuate verso OpenSky (`getArrivals` e `getDepartures`), indipendentemente dall'esito.
  Viene incrementata *all'inizio* di ogni invocazione di `fetchFlights`, prima dell'esecuzione della chiamata HTTP vera e propria.
- `opensky_request_errors_total` (*Counter*)
  Conta il numero di richieste verso OpenSky terminate in errore "reale", ovvero:
    - eccezioni HTTP diverse da `404 Not Found` (errori 4xx/5xx che indicano un malfunzionamento della controparte o problemi di business imprevisti);
    - eccezioni di tipo `RestClientException` (timeout, errori di I/O, problemi di rete).

Il caso `404` viene esplicitamente trattato come *assenza di voli* e non incrementa questo contatore, evitando la confusione fra "nessun dato disponibile" ed errore di servizio.

- `opensky_fallback_total` (*Counter*)
  Registra il numero di attivazioni del **fallback** del Circuit Breaker (`fallbackFetchFlights`).
  È incrementata ogni volta che la chiamata principale viene cortocircuitata da Resilience4j per stato *OPEN* del circuito o per failure che sfociano nel fallback.
  Questa metrica permette di distinguere tra:
  - errori di chiamata "normali" (registrati da `opensky_request_errors_total`);
  - casi in cui il sistema si affida esplicitamente al meccanismo di protezione del Circuit Breaker.
- `opensky_last_fetch_duration_ms` (*Gauge*)
  Memorizza la durata (in millisecondi) **dell'ultima invocazione di** `fetchFlights`, calcolata come differenza di tempo tra inizio e fine della chiamata (inclusa la gestione delle eccezioni).
  La durata viene aggiornata nel blocco `finally`, garantendo che la metrica venga popolata anche in presenza di errori.
  L'uso di un gauge consente di osservare trend di latenza e di correlare eventuali aumenti di `opensky_request_errors_total` con un rallentamento sistematico del servizio esterno.

Queste metriche, combinate con i log del Circuit Breaker, consentono di identificare in modo chiaro problemi di disponibilità o performance dell'API OpenSky e di valutare l'efficacia della protezione offerta da Resilience4j nell'ambito del flusso di raccolta dei voli.

### 3.4. METRICHE APPLICATIVE DELL'ALERT SYSTEM

Nel *Alert System Service* il monitoring applicativo è orientato a misurare il **throughput** e la **latenza** della fase di valutazione delle soglie, nonché il volume di notifiche generate verso il topic Kafka di output. Le metriche sono gestite dalla classe `AlertSystemMetrics` e aggiornate in due punti distinti: il listener Kafka in ingresso (`AlertEventsListener`) e il producer Kafka in uscita (`AlertNotificationProducer`).

Le metriche principali sono:

- `alert_system_evaluations_total` (*Counter*)
  Conta il numero totale di valutazioni di finestre di raccolta eseguite dall'Alert System.
  È incrementata ogni volta che il listener Kafka deserializza con successo un `FlightCollectionWindowUpdateEvent` e invoca `processWindowUpdate` sul servizio di valutazione.
  Questo valore fornisce un indicatore diretto del **volume di finestre elaborate** nel tempo.
- `alert_system_notifications_total` (*Counter*)
  Conta il numero totale di notifiche di superamento soglia

(`ThresholdBreachNotificationEvent`) generate e pubblicate su Kafka dall'Alert System.
È incrementata nel producer `AlertNotificationProducer` dopo la serializzazione e l'invio del messaggio al topic configurato.
Il rapporto fra notifiche generate e valutazioni eseguite permette di stimare, a livello macro, la probabilità di violazione delle soglie configurate.

- `alert_system_last_eval_duration_ms` (*Gauge*)
  Rappresenta la durata (in millisecondi) dell'**ultima valutazione** eseguita dall'Alert System, misurata dalla ricezione del messaggio Kafka alla conclusione del processo di valutazione.
  Il listener registra l'istante iniziale (`startNanos`) e, nel blocco `finally`, aggiorna il gauge con il tempo trascorso.
  Questa metrica consente di:
  - individuare rallentamenti nella logica di valutazione (ad esempio per un numero crescente di interessi attivi o per query più pesanti su `Data DB`);
  - correlare eventuali aumenti di latenza con fluttuazioni nei tempi di risposta del database o del broker Kafka.

In combinazione, queste metriche forniscono una visione chiara della **capacità elaborativa** dell'Alert System e del suo contributo alla latenza complessiva della pipeline di notifica.

### 3.5. METRICHE APPLICATIVE DELL'ALERT NOTIFIER

Nel *Alert Notifier Service* il focus è la fase di **consumo delle notifiche da Kafka** e di orchestrazione dell'invio delle email tramite l'`EmailNotificationService`. Le metriche sono gestite dalla classe `AlertNotifierMetrics` e aggiornate nel listener Kafka `AlertNotificationsListener`.

Le metriche principali sono:

- `alert_notifier_notifications_consumed_total` (*Counter*)
  Conta il numero di notifiche consumate dal topic Kafka di input (to-notifier).
  È incrementata all'inizio della gestione di ogni `ConsumerRecord`, prima della deserializzazione del payload.
  Questo contatore rappresenta il **throughput di input** del servizio, indipendentemente dall'esito della successiva elaborazione.

- `alert_notifier_notifications_processing_errors_total` (*Counter*)
  Conta il numero di errori occorsi durante il processing delle notifiche, inclusi:
  - errori di deserializzazione del payload JSON;
  - eccezioni nella fase di delega al servizio di invio email.
    È incrementata nel blocco `catch` del listener Kafka, consentendo di monitorare la qualità dell'elaborazione a valle del broker.

- `alert_notifier_last_processing_duration_ms` (*Gauge*)
  Memorizza la durata (in millisecondi) dell'**ultimo processing end-to-end** della

notifica, misurata dall'istante di inizio dell'elaborazione nel listener fino alla conclusione del blocco `try/catch/finally` (inclusa la chiamata al servizio di invio email).
Tale durata comprende:

- deserializzazione dell'evento;
- chiamata a `EmailNotificationService` (con eventuale throttling);
- gestione di eventuali eccezioni.

Queste metriche consentono di distinguere tra problemi di **consumo Kafka** (es. errori di deserializzazione) e problemi legati al **downstream SMTP**, che sono invece tracciati in dettaglio tramite le metriche specifiche dell'Email Notification Service.

### 3.6. METRICHE APPLICATIVE DELL'EMAIL NOTIFICATION SERVICE

L'`EmailNotificationService` è responsabile della costruzione e dell'invio delle email di alert verso il Mail Server, includendo una logica di **throttling** per rispettare il rate-limit del provider (ad esempio Mailtrap). La classe `EmailNotificationMetrics` espone un set di metriche orientate a misurare l'efficacia e l'affidabilità di questa fase.

Le metriche principali sono:

- `email_sent_total` (*Counter*)
  Conta il numero totale di email inviate con successo.
  È incrementata dopo che la chiamata a `JavaMailSender.send(...)` è andata a buon fine e che la durata dell'invio è stata misurata.
  Rappresenta un indicatore diretto del **volume di notifiche consegnate** al Mail Server.

- `email_send_errors_total` (*Counter*)
  Conta il numero totale di errori verificatisi durante l'invio delle email, includendo:
  - errori SMTP (`MailException`) come problemi di autenticazione, temporanei 4xx o errori definitivi `5xx`;
  - eccezioni inattese (non `MailException`) legate a problemi runtime, configurazioni errate o altri malfunzionamenti.
    È incrementata sia nel blocco `catch` (`MailException`) sia nel blocco `catch` (`Exception`), fornendo una visione aggregata della *failure rate* del sottosistema di notifica email.

- `email_rate_limited_total` (*Counter*)
  Conta il numero di invii bloccati dal **rate-limit** del provider SMTP.
  Viene incrementata quando il messaggio d'errore associato alla `MailException` contiene un'indicazione esplicita di *rate limiting* (ad esempio "Too many emails per second").
  Questa metrica consente di distinguere problemi di capacità del sistema interno da vincoli di politica esterna imposti dal provider.

- `email_last_send_duration_ms` (*Gauge*)
  Memorizza la durata (in millisecondi) dell'ultimo invio email, calcolata tra l'istante

precedente alla chiamata a `mailSender.send(...)` e l'istante immediatamente successivo, anche nel caso di errore.
Il gauge viene aggiornato sia nei percorsi di successo sia in quelli di errore, a condizione che l'invio sia stato effettivamente tentato.

La combinazione di questi indicatori permette di individuare rapidamente:

- rallentamenti del Mail Server o del canale SMTP (tramite la durata media osservata);
- errori sistematici in fase di invio (ad esempio configurazioni errate delle credenziali o dei parametri di connessione);
- violazioni ricorrenti dei limiti di throughput consentiti dal provider, che richiedono eventualmente un adeguamento della logica di throttling o del piano di servizio.

### 3.7. METRICHE TECNICHE ESPOSTE DAGLI ACTUATOR E DA MICROMETER

Oltre alle metriche applicative definite ad hoc, ogni microservizio strumentato espone, tramite Spring Boot Actuator e Micrometer, un insieme di **metriche tecniche** standard, raccolte da Prometheus tramite l'endpoint `/actuator/prometheus`.

La configurazione degli actuator prevede:

- esposizione degli endpoint `health`, `info` e `prometheus` tramite `management.endpoints.web.exposure.include`;
- abilitazione delle *health probes* dedicate (`management.endpoint.health.probes.enabled`) per supportare **readiness** e **liveness** in Kubernetes, utilizzate dalle probe HTTP definite nei manifest dei Deployment;
- configurazione di **tag globali** per le metriche (`management.metrics.tags.service` e `management.metrics.tags.node`), garantendo un'identificazione coerente del servizio e del nodo sui quali sono raccolte le metriche.

Fra le principali famiglie di metriche tecniche disponibili rientrano:

- **Metriche HTTP server** (`http.server.requests`)
  Riguardano il numero di richieste HTTP gestite, la latenza per endpoint, l'esito (codici di stato) e, ove applicabile, l'eventuale presenza di errori.
  Queste metriche forniscono una prospettiva *black-box* sul comportamento delle API REST esposte dai microservizi.
- **Metriche JVM** (`jvm.*`)
  Includono utilizzo di heap e non-heap, conteggio dei thread, attività del garbage collector, dimensione dei pool di buffer e altri aspetti legati alla runtime Java.
  Sono fondamentali per individuare problemi quali memory leak, eccessiva frammentazione o blocchi del GC.

- **Metriche di sistema** (`system.*`, `process.*`)
  Riguardano utilizzo della CPU, file descriptor, uptime del processo, carico medio e altre informazioni ambientali.
- **Metriche di logging e infrastruttura interna** (`logback.*`, metriche dei pool di connessione JDBC ecc.)
  Consentono di monitorare aspetti come il livello di logging attivo, la dimensione dei pool di connessione al database, i tempi di attesa per ottenere connessioni, ecc.

Tutte queste metriche confluiscono nello stesso flusso Prometheus, arricchite dai tag `service`, `node`, `job` e `instance`, e possono essere interrogate congiuntamente alle metriche applicative per effettuare **analisi di correlazione** (ad esempio, correlare un aumento della latenza di valutazione con un aumento del tempo di risposta del database o con un eccessivo carico CPU).

### 3.8. CONSIDERAZIONI SU GRANULARITÀ, OVERHEAD E RIUSO DELLE METRICHE

Il modello di metriche adottato è stato progettato con particolare attenzione a tre aspetti: **granularità**, **overhead** e **riuso**.

Sul piano della **granularità**, le scelte effettuate mirano a:

- concentrarsi su **punti di osservazione chiave**:
  - confine con i servizi esterni (OpenSky e Mail Server);
  - ingressi e uscite dei componenti Kafka-based;
  - durata complessiva delle fasi critiche (valutazione delle soglie, processing delle notifiche, invio email);
- evitare metriche a **granularità troppo fine**, come:
  - contatori distinti per singolo aeroporto;
  - metriche con label utente o combinazioni di parametri di business.
    Questi aspetti di dettaglio rimangono gestiti tramite logging applicativo, lasciando alle metriche il ruolo di rappresentare indicatori sintetici ma significativi.

Per quanto riguarda l'**overhead**, il modello privilegiato comporta:

- operazioni di aggiornamento delle metriche **a costo costante** (incrementi di counter e aggiornamenti di gauge basati su `AtomicLong`), che risultano trascurabili rispetto al costo delle chiamate esterne o delle operazioni su database;
- un numero limitato di metodi strumentati, posizionati nei punti in cui la semantica dell'operazione è chiara e facilmente interpretabile (ad esempio all'ingresso del listener Kafka, immediatamente prima e dopo la chiamata al Mail Server o alla API OpenSky);

- un intervallo di scraping di Prometheus configurato in modo equilibrato (15 secondi), che consente di ottenere una risoluzione temporale adeguata senza generare carico eccessivo sulla rete o sui microservizi.

Infine, sul fronte del **riuso**, il modello è stato impostato per essere:

- **coerente tra i diversi microservizi**: nomenclatura omogenea dei prefissi (`opensky_*`, `alert_system_*`, `alert_notifier_*`, `email_*`), tipologie di metriche ricorrenti (counter per volumi e errori, gauge per durate) e tag comuni (`service`, `node`);
- **indipendente dall'ambiente di esecuzione**: le metriche applicative sono definite nel codice dei microservizi e rimangono valide sia in esecuzione *stand-alone* che in ambiente Docker o Kubernetes;
- **estensibile**: l'introduzione di nuovi microservizi o di ulteriori viste di monitoring (ad esempio l'aggiunta di un dashboard Grafana) può riutilizzare lo stesso schema di naming, gli stessi tag e lo stesso meccanismo di esposizione su `/actuator/prometheus`, riducendo il costo complessivo di evoluzione del sottosistema di osservabilità.

# 4. INTERAZIONI TRA COMPONENTI E SCENARI MONITORATI

## 4.1. SCENARIO DI PIPELINE DI NOTIFICA ASINCRONA: DESCRIZIONE FUNZIONALE

Lo scenario di riferimento è la **pipeline asincrona** che parte dalla raccolta periodica dei voli sugli aeroporti di interesse e termina con l'invio di email di notifica agli utenti in caso di superamento delle soglie configurate. Tale pipeline è realizzata attraverso una combinazione di:

- microservizi applicativi (*Data Collector*, *Alert System*, *Alert Notifier*);
- messaggistica **event-driven** basata su Kafka;
- interazioni con servizi esterni (OpenSky e Mail Server);
- metriche applicative aggiornate lungo le varie fasi per supportare il monitoring end-to-end.

Il flusso funzionale può essere descritto in fasi.

1. **Raccolta periodica dei voli (Data Collector)**
    - Uno **Scheduler** interno al Data Collector attiva periodicamente l'operazione di raccolta, invocando il servizio di *Flight Collection* con una finestra temporale `[windowBegin, windowEnd]`.
    - Per ciascun aeroporto per cui esiste almeno un interesse attivo, il *Flight Collection Service*:
        - interroga l'API OpenSky, tramite l'`OpenSkyClient`, per ottenere voli in arrivo e in partenza;
        - persiste i record di volo in **Data DB**;
        - costruisce uno snapshot aggregato per aeroporto, contenente il numero di arrivi e partenze nella finestra.
    - Durante ciascuna invocazione dell'`OpenSkyClient`, vengono aggiornate le metriche applicative:
        - contatori di richieste totali, errori e fallback;
        - gauge con la durata dell'ultima chiamata.
2. **Pubblicazione dell'evento finestra su Kafka**
    - Terminata la raccolta, il Data Collector costruisce un evento di tipo *FlightCollectionWindowUpdateEvent*, che contiene:
        - la finestra temporale elaborata;
        - la lista degli snapshot aeroportuali costruiti.
    - L'`AlertUpdateProducer` serializza tale evento e lo pubblica sul topic Kafka dedicato alla comunicazione verso l'Alert System.

- L’operazione è asincrona: il Data Collector non attende alcuna conferma da parte dell’Alert System e rimane disaccoppiato dal consumer.

3. **Consumo dell’evento e valutazione delle soglie (Alert System)**
   - Un listener Kafka nel microservizio **Alert System** riceve gli eventi pubblicati sul topic e li deserializza in oggetti *FlightCollectionWindowUpdateEvent*.
   - Per ciascun evento:
     - viene incrementato il contatore di valutazioni eseguite;
     - viene avviata una misurazione della durata complessiva della valutazione.
   - L’AlertEvaluationService:
     - calcola il numero totale di voli per ciascun aeroporto nella finestra;
     - recupera da **Data DB** gli interessi degli utenti verso gli aeroporti coinvolti, filtrando quelli che possiedono soglie *high*/*low*;
     - per ogni combinazione *snapshot–interesse*, valuta le condizioni di superamento soglia e, se necessario, crea eventi *ThresholdBreachNotificationEvent* (distinguendo tra violazioni *HIGH* e *LOW*).
   - Ogni notifica generata viene inviata a un **producer Kafka** dedicato, che:
     - serializza l’evento;
     - lo pubblica sul topic riservato alle notifiche verso l’Alert Notifier;
     - incrementa il contatore di notifiche prodotte.
   - Al termine della valutazione, il gauge di durata viene aggiornato con il tempo impiegato per l’elaborazione della finestra.
4. **Consumo delle notifiche e delega all’invio email (Alert Notifier)**
   - Un listener Kafka nel microservizio **Alert Notifier** consuma gli eventi *ThresholdBreachNotificationEvent* dal topic corrispondente.
   - Per ciascun messaggio:
     - viene incrementato il contatore di notifiche consumate;
     - viene avviata la misurazione del tempo di processing end-to-end;
     - viene eseguita la deserializzazione del payload e viene invocato l’`EmailNotificationService`.
   - In caso di errori nella deserializzazione o nell’invocazione del servizio di invio email, viene incrementato il contatore di errori di processing e la durata dell’elaborazione viene comunque registrata.

5. **Costruzione e invio dell'email (Email Notification Service)**
    - L'`EmailNotificationService` costruisce il contenuto dell'email (oggetto e corpo) a partire dalle informazioni contenute nell'evento:
        - indirizzo email dell'utente;
        - aeroporto interessato;
        - tipo di violazione (HIGH/LOW) e valori soglia/valore osservato;
        - finestra temporale di riferimento.
    - Prima dell'invio, può applicare politiche di *throttling* per rispettare i limiti del provider SMTP.
    - L'invio avviene tramite il `JavaMailSender` verso il **Mail Server** configurato:
        - se l'invio ha successo, vengono aggiornate le metriche di successo e la durata dell'ultima operazione di invio;
        - se si verifica un errore (SMTP o eccezione imprevista), vengono aggiornate le metriche di errore e, nel caso di messaggi di rate-limit, il contatore specifico di invii bloccati.
    - Al termine dell'operazione, il controllo ritorna al listener Kafka, che completa il processing della notifica e consente il *commit* dell'offset.

L'intera pipeline è quindi **completamente asincrona**, con due confini principali gestiti tramite Kafka (Data Collector → Alert System, Alert System → Alert Notifier) e una forte attenzione alla raccolta di metriche nei punti critici del flusso.

### 4.2. DIAGRAMMA DI SEQUENZA – PIPELINE DI NOTIFICA ASINCRONA CON RACCOLTA DELLE METRICHE APPLICATIVE (DATA COLLECTOR → ALERT SYSTEM → ALERT NOTIFIER → MAIL SERVER)

Il diagramma di sequenza associato a questo scenario descrive, in modo dettagliato e temporale, le interazioni tra i componenti coinvolti nella pipeline di notifica, integrando sia il **profilo funzionale** sia il **profilo di monitoring**.

Gli elementi rappresentati includono:

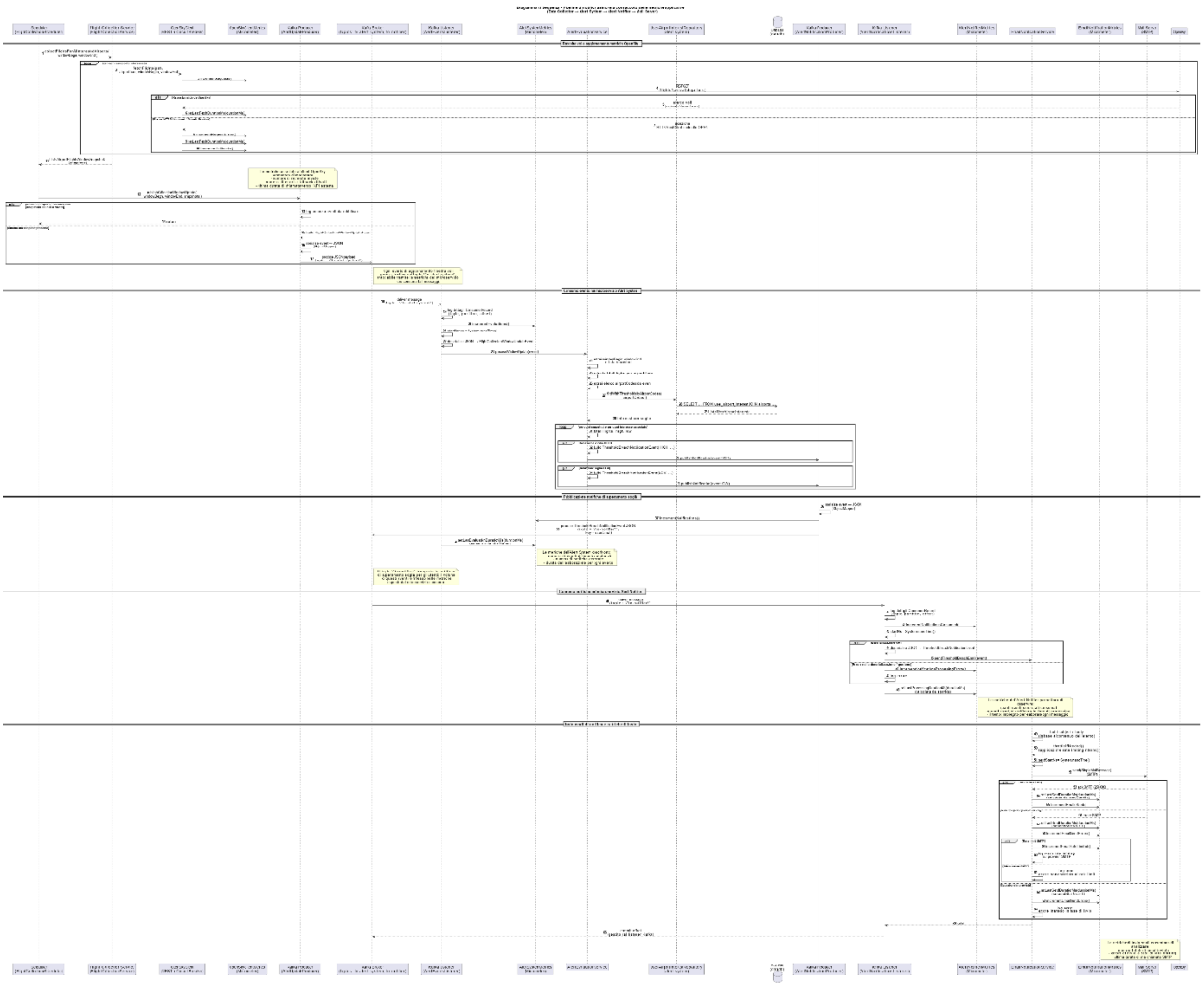
- nel **Data Collector**:
    - lo Scheduler che attiva il flusso di raccolta;
    - il *Flight Collection Service* che coordina le chiamate a OpenSky e la costruzione degli snapshot;
    - l'`OpenSkyClient` che effettua le richieste HTTP verso gli endpoint esterni;
    - il componente di metriche `OpenSkyClientMetrics`;
    - il Kafka producer responsabile della pubblicazione degli eventi di finestra;

- nel **Kafka broker**:
    - i topic utilizzati per la comunicazione tra Data Collector e Alert System e tra Alert System e Alert Notifier;
- nell'**Alert System**:
    - il listener Kafka che consuma gli eventi di finestra;
    - il componente di metriche `AlertSystemMetrics`;
    - l'`AlertEvaluationService` che implementa la logica di soglia;
    - il repository per gli interessi utenti (`UserAirportInterestRepository`) e l'accesso a *Data DB*;
    - il Kafka producer che pubblica le notifiche di superamento soglia;
- nell'**Alert Notifier**:
    - il listener Kafka che consuma le notifiche;
    - il componente di metriche `AlertNotifierMetrics`;
    - l'`EmailNotificationService` che si occupa della costruzione e dell'invio delle email;
    - il componente di metriche `EmailNotificationMetrics`;
    - il Mail Server esterno che riceve le richieste SMTP.

Il diagramma evidenzia in particolare:

- la sequenza delle chiamate **sincrone** all'interno di ciascun microservizio (dal listener Kafka al servizio di business, dai servizi di business ai componenti di persistenza o di integrazione esterna);
- i punti in cui vengono **aggiornate le metriche applicative**:
    - incrementi dei contatori di richieste, errori e fallback nel client OpenSky;
    - conteggio delle valutazioni e delle notifiche nell'Alert System;
    - conteggio delle notifiche consumate e degli errori di processing nell'Alert Notifier;
    - conteggio delle email inviate, delle email fallite e degli invii rate-limit nell'Email Notification Service;
    - aggiornamento dei gauge di durata per le chiamate critiche (OpenSky, valutazione, processing, invio email);
- i confini **asincroni** dove l'interazione tra componenti avviene tramite Kafka, con pubblicazione e consumo dei messaggi da topic dedicati, mantenendo il disaccoppiamento tra i microservizi.

Questa rappresentazione fornisce una vista completa del flusso di business e dell'impianto di monitoring, permettendo di comprendere come i diversi componenti contribuiscano alle metriche esposte a Prometheus.



## 4.3. SCENARIO DI SCRAPING PROMETHEUS E CONSULTAZIONE DELLE METRICHE DA PARTE DEGLI OPERATORI

Lo scenario di scraping e consultazione delle metriche descrive il flusso che consente a un **operatore tecnico** di osservare lo stato del sistema a partire dai dati raccolti da Prometheus.

La sequenza funzionale è articolata in tre livelli:

1. **Esposizione delle metriche da parte dei microservizi**
    - Ogni microservizio strumentato espone un endpoint HTTP `/actuator/prometheus`, che produce l'output in formato testuale conforme allo *plain text format* di Prometheus.

- L’endpoint aggrega:
  - metriche applicative definite nel codice (ad esempio le metriche di OpenSky, di valutazione e di invio email);
  - metriche tecniche raccolte da Micrometer e dagli actuator (HTTP, JVM, sistema).
- Le metriche sono arricchite con i tag globali configurati (`service`, `node`) e con i tag aggiunti automaticamente dal contesto (ad esempio l’endpoint HTTP specifico, il codice di stato, ecc.).

2. **Scraping da parte del server Prometheus**
   - Il server **Prometheus** è configurato con uno o più *scrape job* dedicati all’applicazione, nei quali vengono specificati:
     - l’intervallo di scraping (ad esempio, 15 secondi);
     - i target da interrogare (Service Kubernetes o endpoint diretti esposti nel namespace applicativo);
     - eventuali label aggiuntive a livello di job.
   - A ogni intervallo di scraping, Prometheus:
     - effettua una richiesta HTTP GET verso l’endpoint `/actuator/prometheus` del target;
     - riceve il payload testuale contenente tutte le metriche attualmente esposte;
     - aggiorna il proprio database di serie temporali con i nuovi campioni.
   - Grazie all’integrazione con Kubernetes, le label `job`, `instance`, `namespace`, `pod` e `service` permettono di identificare con precisione la sorgente di ciascuna metrica.
3. **Consultazione delle metriche da parte dell’operatore**
   - Un operatore accede all’interfaccia di Prometheus (o a strumenti che si appoggiano alla stessa API, come eventuali dashboard) e formula query basate su **PromQL**, ad esempio:
     - tassi di errore di OpenSky (`rate(opensky_request_errors_total[5m])`);
     - throughput delle notifiche (`rate(alert_system_notifications_total[10m])`, `rate(alert_notifier_notifications_consumed_total[10m])`);
     - distribuzione delle durate (`opensky_last_fetch_duration_ms`, `alert_system_last_eval_duration_ms`, `email_last_send_duration_ms`).

- Le query possono filtrare per `service`, `node`, `job` o altre label, ottenendo viste differenziate:
  - per microservizio (ad esempio solo `alert-system-service`);
  - per nodo del cluster (utile in scenari con più repliche);
  - per job di scraping specifico.
- I risultati possono essere visualizzati sotto forma di grafici temporali, tabelle o valutati programmaticamente, fungendo da base per analisi di capacità, troubleshooting o definizione di soglie di alerting.

Questo scenario rende evidente come il modello di metriche definito nei microservizi sia effettivamente *azionabile* lato operatore, consentendo un'osservazione continua e strutturata del comportamento del sistema.

### 4.4. DIAGRAMMA DI SEQUENZA – SCRAPING PROMETHEUS E CONSULTAZIONE DELLE METRICHE APPLICATIVE

Il diagramma di sequenza correlato allo scenario di scraping rappresenta in forma temporale le interazioni tra:

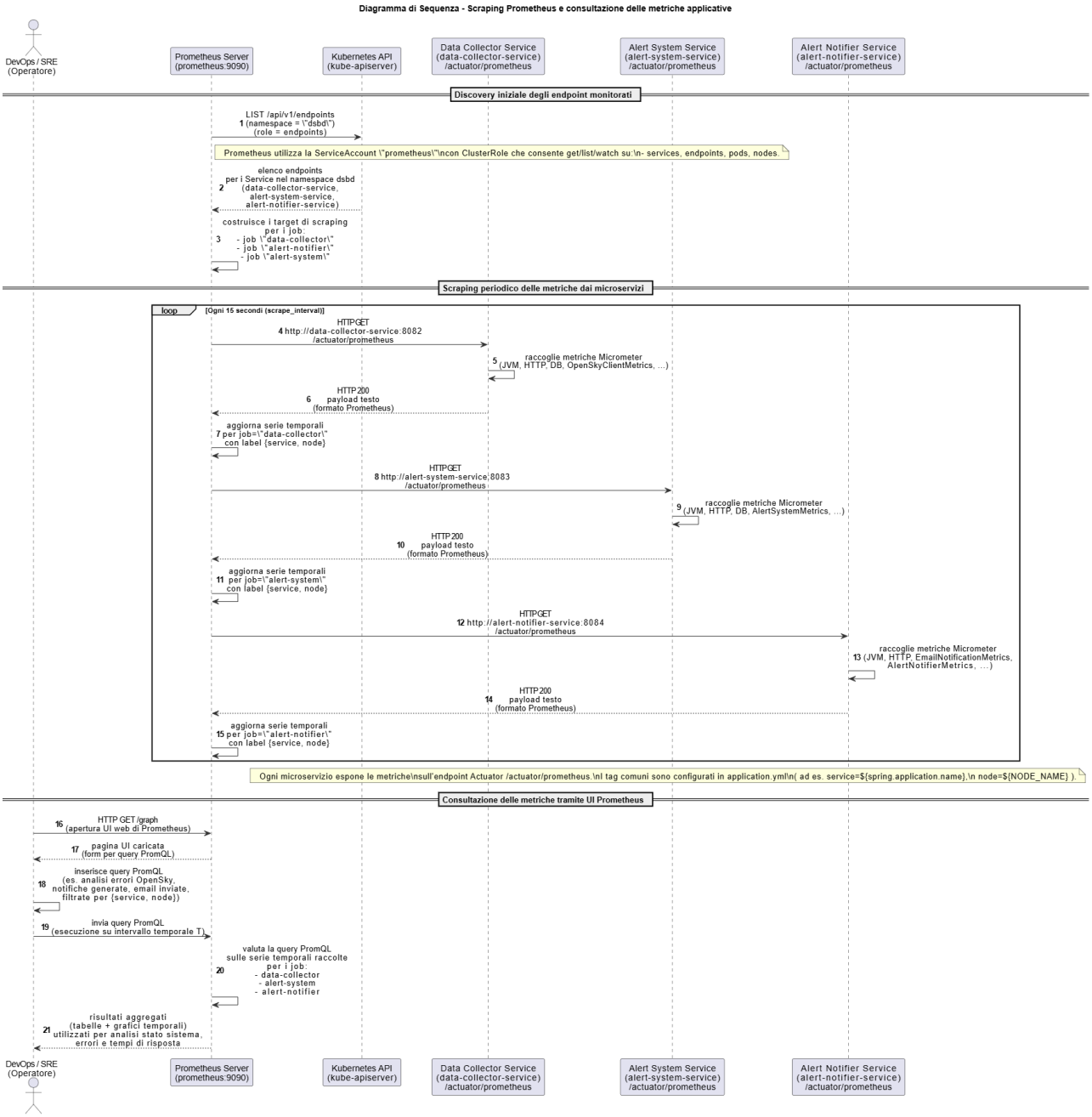
- il **server Prometheus** come componente centrale di monitoring;
- i **microservizi target** (Data Collector, Alert System, Alert Notifier e, dove rilevante, altri servizi) che espongono l'endpoint `/actuator/prometheus`;
- l'**operatore** che interroga Prometheus e analizza i risultati.

Gli elementi evidenziati sono in particolare:

- il ciclo periodico di **scraping**:
  - Prometheus avvia, a intervalli configurati, le richieste HTTP GET verso i target registrati;
  - ciascun microservizio risponde con il payload delle metriche, generato dinamicamente da Micrometer;
  - Prometheus aggiorna il proprio database di serie temporali con i nuovi campioni;
- l'associazione delle metriche ai target tramite **label**:
  - ogni risposta è arricchita dal contesto di scraping (job, instance);
  - le metriche contengono i tag `service` e `node` configurati a livello di microservizio;
- la fase di **consultazione**:
  - l'operatore invia una query PromQL all'interfaccia di Prometheus;
  - Prometheus valuta la query sul proprio database e restituisce i risultati;

- l’operatore interpreta i valori e, se necessario, approfondisce tramite query aggiuntive.

Il diagramma mostra il legame tra aggiornamento delle metriche nei microservizi (derivante dall’esecuzione della pipeline di business) e la raccolta periodica operata da Prometheus, mettendo in luce il fatto che le misurazioni inserite nel codice applicativo diventano osservabili e interrogabili in modo centralizzato.



### 4.5. FLUSSI DI INTERAZIONE CON SERVIZI ESTERNI (OpenSky, MAIL SERVER, CLIENT DI MONITORING)

Oltre alle interazioni interne tra microservizi, l'architettura prevede flussi strutturati verso alcuni **servizi esterni** fondamentali:

- **OpenSky Network API**
    - Il Data Collector interagisce con OpenSky come sorgente autorevole dei dati di volo, utilizzando l'`OpenSkyClient` per invocare endpoint dedicati a arrivi e partenze per specifici aeroporti e finestre temporali.
    - Le chiamate sono effettuate tramite HTTP/HTTPS, con autenticazione secondo il modello previsto dal servizio esterno, e sono protette da un Circuit Breaker per evitare che malfunzionamenti prolungati compromettano la stabilità del sistema interno.
    - Le metriche di OpenSky rappresentano un punto di osservazione privilegiato per misurare:
        - disponibilità e performance del servizio esterno;
        - impatto di tali caratteristiche sulla tempestività di aggiornamento dei dati nel sistema di flight monitoring.
- **Mail Server SMTP**
    - L'Alert Notifier, tramite l'`EmailNotificationService`, interagisce con il Mail Server per l'invio delle email di notifica.
    - Le comunicazioni avvengono in protocollo **SMTP**, con eventuale autenticazione e cifratura in base alla configurazione.
    - Gli errori restituiti dal Mail Server (ad esempio errori di autenticazione, limiti di throughput, problemi temporanei) vengono mappati su metriche applicative, rendendo possibile:
        - distinguere problemi di rete o configurazione da limitazioni esplicite del provider;
        - stimare l'affidabilità del canale di notifica e la necessità di adeguamenti nella logica di retry o throttling.
- **Client di monitoring e strumenti di ispezione**
    - Gli operatori accedono al sistema tramite diversi strumenti:
        - l'interfaccia web di **Prometheus** per la consultazione delle metriche;
        - eventuali strumenti esterni che si appoggiano alle API HTTP di Prometheus per costruire dashboard o report;
        - la **Kafka UI** per l'ispezione dei topic, dei messaggi e dei consumer group;

        - strumenti REST (ad esempio client HTTP) che interrogano l'API Gateway per testare selettivamente determinati endpoint applicativi.
    - Questi client non partecipano direttamente alla logica di business, ma svolgono un ruolo essenziale nelle attività di:
        - osservabilità e diagnosi;
        - validazione funzionale e non funzionale;
        - analisi a posteriori di incidenti o anomalie operative.

La combinazione dei flussi verso OpenSky, Mail Server e client di monitoring completa la vista delle interazioni del sistema con il mondo esterno, mostrando come il **dominio applicativo**, la **messaggistica interna** e il **monitoring** si appoggino a servizi esterni in modo controllato e misurabile tramite le metriche esposte.

# 5. API ESPOSTE DAI MICROSERVIZI E PUNTI DI OSSERVAZIONE

## 5.1. ENDPOINT REST PUBBLICATI TRAMITE API GATEWAY

L'accesso esterno al sistema avviene unicamente tramite l'**API Gateway NGINX**, che svolge la funzione di **reverse proxy** verso i microservizi interni. Il gateway espone un set di endpoint REST *stabilizzato* e indipendente dai dettagli di deployment dei singoli pod, presentando al client una superficie uniforme.

Gli endpoint principali pubblicati verso l'esterno sono organizzati per **ambito funzionale**:

- **Gestione utenti** (instradati verso *User Manager*):
  - `POST /api/users` – creazione di un nuovo utente;
  - `GET /api/users/{userId}` – interrogazione dei dati di un utente;
  - `GET /api/users` – elenco o ricerca di utenti secondo criteri di filtro di base.
- **Gestione degli interessi aeroportuali** (instradati verso *Data Collector*):
  - `POST /api/interests` – registrazione di un interesse verso un aeroporto (associato a un utente e a un codice ICAO);
  - `GET /api/interests` – interrogazione degli interessi configurati, con possibilità di filtro per utente;
  - `DELETE /api/interests/{interestId}` – rimozione di un interesse precedentemente registrato;
  - eventuali endpoint di aggiornamento per l'inserimento/modifica delle soglie *high*/*low* associate all'interesse.
- **Interrogazione dei voli** (instradati verso *Data Collector*):
  - `GET /api/flights` – interrogazione dei voli registrati nel sistema, filtrabili per aeroporto, finestra temporale e/o utente;
  - eventuali varianti per ottenere viste aggregate o dettagliate del traffico in una certa finestra.

L'API Gateway effettua il **routing** verso i Service Kubernetes interni (`user-manager-service`, `data-collector-service`, ecc.), mantenendo il path logico `/api/...` stabile anche in presenza di modifiche infrastrutturali. Dal punto di vista del monitoring:

- tutte le richieste transitano da un unico *entry point*, permettendo di controllare globalmente il traffico in ingresso;
- i microservizi interni espongono metriche HTTP (tramite Actuator/Micrometer) che consentono di correlare *tasso di richieste* e *comportamento per endpoint* con i flussi rilevati a livello di gateway.

## 5.2. ENDPOINT REST DEI MICROSERVIZI APPLICATIVI

Gli endpoint REST effettivi sono implementati nei microservizi applicativi, che espongono API **HTTP/JSON** sui rispettivi port interni (ad esempio 8081 per User Manager, 8082 per

Data Collector). Nella configurazione di produzione, tali endpoint sono normalmente raggiunti solo tramite il gateway, ma restano comunque accessibili come Service Kubernetes interni per scopi di test o amministrazione.

Di seguito si riportano i principali endpoint e il relativo ruolo, raggruppati per microservizio.

### 5.2.1. USER MANAGER: GESTIONE E INTERROGAZIONE DEGLI UTENTI

Lo **User Manager Service** implementa le API di **gestione dell'anagrafica utenti**. La superficie tipica comprende:

- `POST /users`
  Crea un nuovo utente a partire da un payload JSON contenente i dati identificativi necessari (ad esempio identificativo logico, email, eventuali altri attributi di profilo). Costituisce il punto di ingresso per l'onboarding degli utenti nel sistema di flight monitoring.
- `GET /users/{userId}`
  Restituisce i dati di un utente identificato da `userId`.
  È utilizzato sia dai client esterni, sia indirettamente come riferimento logico per la validazione gRPC effettuata dal Data Collector.
- `GET /users`
  Fornisce l'elenco degli utenti, con possibilità di filtro e paginazione di base.
  Può essere impiegato per attività di amministrazione o verifica della corretta registrazione degli utenti.

In aggiunta alle API funzionali, il microservizio espone gli **endpoint di servizio** Actuator (`/actuator/health`, `/actuator/info`, `/actuator/prometheus`), che costituiscono i principali punti di osservazione tecnica. Le metriche HTTP relative a questi endpoint includono:

- conteggio delle richieste per metodo e path;
- tempi medi/percentili di risposta;
- distribuzione dei codici di stato (2xx, 4xx, 5xx).

Queste informazioni consentono di rilevare anomalie nel carico o nel comportamento del servizio (ad esempio un aumento di errori 5xx dopo modifiche alla configurazione).

### 5.2.2. DATA COLLECTOR: GESTIONE DEGLI INTERESSI E INTERROGAZIONE DEI VOLI

Il **Data Collector Service** espone API per la **gestione degli interessi aeroportuali** e per l'**interrogazione dei dati di volo** raccolti:

- `POST /interests`
  Registra un nuovo interesse associato a:
  - un utente esistente (identificato da `userId` o da un campo equivalente);
  - un aeroporto (codice ICAO);

    - eventuali soglie *high* e *low* per il numero di voli nella finestra.
      L'operazione può attivare, a livello logico, la partecipazione dell'aeroporto alla successiva raccolta periodica dei voli.

- `GET /interests`
  Restituisce l'elenco degli interessi registrati, con possibilità di filtro per utente o altri criteri.
  È spesso utilizzato per verificare lo stato delle configurazioni applicate dagli utenti e per testare la corretta propagazione delle informazioni in *Data DB*.

- `DELETE /interests/{interestId}`
  Rimuove un interesse precedentemente registrato, impedendo che l'aeroporto corrispondente continui a essere considerato nelle raccolte future per quell'utente.

- `PUT o PATCH /interests/{interestId}` (se previsto)
  Consente l'aggiornamento delle soglie di interesse per un record esistente, permettendo di modificare limiti *HIGH*/*LOW* senza dover creare un nuovo interesse.

- `GET /flights`
  Interroga i voli presenti in *Data DB*, consentendo filtri per:

    - aeroporto (codice ICAO);
    - intervallo temporale;
    - utente o interesse associato.
      Questo endpoint è utilizzato per verificare che i dati raccolti da OpenSky siano stati correttamente persistiti e correlati agli interessi configurati.

Anche il Data Collector espone gli endpoint Actuator, con un ruolo fondamentale per il monitoring. In particolare, le richieste verso `/interests` e `/flights` vengono riflesse nelle metriche HTTP di Micrometer, mentre le interazioni periodiche verso OpenSky e la pubblicazione su Kafka sono tracciate dalle metriche applicative personalizzate (`opensky_*`).

### 5.2.3. ENDPOINT DI SERVIZIO E HEALTH CHECK NEGLI ALTRI MICROSERVIZI

Gli altri microservizi applicativi, **Alert System** e **Alert Notifier**, espongono principalmente **endpoint di servizio** a uso interno o amministrativo:

- `GET /actuator/health`
  Fornisce un'indicazione sintetica sullo stato del microservizio, utilizzata dalle **liveness** e **readiness probe** di Kubernetes per stabilire se il pod è in grado di accettare traffico e di eseguire correttamente le proprie responsabilità.

- `GET /actuator/health/liveness` e `GET /actuator/health/readiness` (se abilitati)
  Discriminano fra problemi di disponibilità generali e problemi che impediscono al servizio di gestire correttamente le richieste.

- `GET /actuator/info`
  Espone metadati sull'applicazione (versione, informazioni di build), utili per verificare la versione effettivamente deployata.

In questa versione, Alert System e Alert Notifier non espongono endpoint REST di business accessibili ai client esterni: la loro funzionalità è attivata esclusivamente tramite **Kafka**, a partire dagli eventi generati dal Data Collector. Dal punto di vista dell'osservabilità, gli endpoint Actuator costituiscono il principale punto di ispezione diretto, mentre l'andamento della pipeline è osservato tramite:

- metriche applicative proprie (`alert_system_*`, `alert_notifier_*`, `email_*`);
- ispezione dei topic Kafka tramite Kafka UI.

### 5.3. ENDPOINT gRPC PER LA VALIDAZIONE UTENTE

La **validazione dell'esistenza dell'utente** è realizzata tramite un canale **gRPC** tra il Data Collector e lo User Manager. Questo pattern consente di:

- evitare duplicazioni della logica di validazione;
- ottenere una comunicazione binaria efficiente e tipizzata fra i due microservizi.

Sul lato **User Manager** è esposto un server gRPC che implementa un servizio di validazione, concettualmente simile a:

- `service UserValidationService { rpc userExists(UserIdRequest) returns (UserExistsResponse); }`

Il Data Collector utilizza un **client gRPC** per interrogare lo User Manager prima di:

- registrare un interesse associato a un `userId`;
- eseguire operazioni che presuppongano l'esistenza di un utente.

Dal punto di vista del monitoring:

- il canale gRPC non è direttamente visibile tramite un endpoint HTTP specifico, ma il suo utilizzo si riflette:
    - nelle metriche tecniche di rete e di thread dello User Manager;
    - nella *health* complessiva del servizio.
- il comportamento della validazione può essere inferito indirettamente dal successo o fallimento delle operazioni REST nel Data Collector (ad esempio tramite codici di errore quando la validazione fallisce).

L'uso di gRPC mantiene il **dominio applicativo** centralizzato nello User Manager, mentre la Data Collector conserva una dipendenza chiara e confinata sul servizio di validazione, senza dover accedere direttamente al suo database.

### 5.4. ENDPOINT DI ESPOSIZIONE DELLE METRICHE PROMETHEUS (/actuator/prometheus)

Il fulcro del sottosistema di monitoring è l'endpoint `/actuator/prometheus`, esposto dai microservizi Spring Boot strumentati. Questo endpoint:

- è reso disponibile da **Spring Boot Actuator** in combinazione con **Micrometer**;

- fornisce tutte le metriche registrate nel registry Micrometer del microservizio, sia tecniche sia applicative, in formato **testuale Prometheus**.

Per ciascun microservizio critico (Data Collector, Alert System, Alert Notifier, User Manager), l'endpoint:

- rappresenta il **target di scraping** configurato in Prometheus (`metrics_path: /actuator/prometheus`);
- è arricchito con tag globali configurati tramite `management.metrics.tags.service` e `management.metrics.tags.node`, che identificano:
    - il nome logico del servizio;
    - il nodo su cui gira il pod, valorizzato tramite variabili d'ambiente (ad esempio NODE_NAME).

Esempi di famiglie di metrica esposte includono:

- `opensky_requests_total`, `opensky_request_errors_total`, `opensky_fallback_total`, `opensky_last_fetch_duration_ms` per il Data Collector;
- `alert_system_evaluations_total`, `alert_system_notifications_total`, `alert_system_last_eval_duration_ms` per l'Alert System;
- `alert_notifier_notifications_consumed_total`, `alert_notifier_notifications_processing_errors_total`, `alert_notifier_last_processing_duration_ms` per l'Alert Notifier;
- `email_sent_total`, `email_send_errors_total`, `email_rate_limited_total`, `email_last_send_duration_ms` per l'Email Notification Service;
- metriche tecniche standard (`http.server.requests`, `jvm.memory.used`, `jvm.gc.pause`, `process.uptime`, ecc.) generate automaticamente.

L'endpoint `/actuator/prometheus` costituisce quindi il principale **punto di osservazione unificato** per le metriche del sistema, centralizzando in un unico flusso interrogabile da Prometheus tutte le informazioni raccolte in fase di esecuzione.

### 5.5. PATTERN DI COMUNICAZIONE (REST, gRPC ED EVENT-DRIVEN CON KAFKA) E LORO IMPATTO SUL MONITORING

L'architettura della versione corrente adotta tre principali **pattern di comunicazione**:

1. **REST/HTTP**
    - Utilizzato per:
        - API di dominio esposte agli utenti (via API Gateway) per gestione utenti, interessi e interrogazione voli;
        - endpoint di servizio e di health esposti dai microservizi.

- Impatto sul monitoring:
  - le richieste REST sono tracciate da Micrometer tramite le metriche `http.server.requests`, consentendo di analizzare:
    - throughput per endpoint;
    - tempi medi di risposta;
    - distribuzione dei codici di stato;
  - eventuali anomalie nei flussi applicativi sono immediatamente visibili come picchi di errori 4xx/5xx o come aumenti di latenza sugli endpoint critici.

2. **gRPC**
   - Utilizzato per la **validazione utente** tra Data Collector e User Manager.
   - Impatto sul monitoring:
     - il traffico gRPC resta interno al cluster e non è esposto come endpoint HTTP dedicato;
     - l'effetto del suo corretto o errato funzionamento si riflette nei log e nelle metriche HTTP/REST del Data Collector (ad esempio errori in operazioni che presuppongono la validazione) e nelle metriche tecniche dello User Manager (carico CPU, tempi di risposta generali).
   - Il pattern gRPC riduce l'overhead di serializzazione rispetto a REST e mantiene la coerenza dei contratti tramite il file di definizione del servizio, a fronte di una minore visibilità diretta delle singole chiamate nel layer di monitoring general-purpose.
3. **Event-driven con Kafka**
   - Utilizzato per:
     - propagare in modo asincrono gli eventi di aggiornamento finestra dal Data Collector all'Alert System;
     - propagare le notifiche di superamento soglia dall'Alert System all'Alert Notifier.
   - Impatto sul monitoring:
     - la pipeline Kafka-based introduce **confini asincroni** in cui non esiste una singola *request/response* end-to-end;
     - il monitoring deve perciò concentrarsi su:
       - **metriche di throughput** (eventi pubblicati, eventi consumati, notifiche generate);
       - **metriche di latenza** misurate all'interno dei microservizi (durata della valutazione, durata del processing nel notifier);

        - **metriche di errore** (errori di deserializzazione, errori di processing, errori SMTP).
    - la *Kafka UI* fornisce un supporto complementare per osservare:
        - stato dei topic;
        - lag dei consumer group;
        - eventuali anomalie di consumo/produttività non catturate direttamente dalle metriche applicative.

La combinazione di questi pattern realizza un equilibrio tra:

- **visibilità diretta**, garantita dal monitoring HTTP e dai log associati alle API REST;
- **efficienza e disaccoppiamento**, ottenuti con gRPC e Kafka;
- **osservabilità end-to-end**, raggiunta attraverso l'inserimento di metriche applicative nei punti chiave della pipeline e la raccolta centralizzata tramite l'endpoint `/actuator/prometheus` scrappato da Prometheus.

Questo disegno consente di valutare non solo lo stato dei singoli microservizi, ma anche il comportamento complessivo dei flussi di business che li attraversano, supportando sia il controllo operativo sia l'analisi dettagliata degli scenari monitorati.

# 6. DEPLOYMENT SU KUBERNETES E INTEGRAZIONE DEL MONITORING

## 6.1. OBIETTIVI DEL DEPLOYMENT SU PIATTAFORMA KUBERNETES

Il deployment su **Kubernetes** ha l'obiettivo di portare l'architettura a microservizi del sistema di flight monitoring in un ambiente di esecuzione **coerente, isolato e riproducibile**, mantenendo una chiara separazione tra:

- **servizi applicativi** (User Manager, Data Collector, Alert System, Alert Notifier, API Gateway);
- **servizi di infrastruttura** (PostgreSQL, Kafka, ZooKeeper, Kafka UI, Prometheus).

Gli obiettivi principali possono essere sintetizzati come segue:

- garantire una **mappatura 1:1** tra l'architettura logica (microservizi, broker, database, monitoring) e gli artefatti di deployment Kubernetes (Deployment, Service, ConfigMap, Secret, ecc.);
- rendere **esplicite le dipendenze** fra i componenti (ad esempio Alert Notifier che dipende da Kafka e dal Mail Server, Data Collector che dipende da OpenSky e da PostgreSQL);
- sfruttare le **primitive native di Kubernetes** (probe di liveness/readiness, Service discovery, configurazione tramite ConfigMap/Secret) per migliorare robustezza e gestibilità del sistema;
- integrare in modo naturale il sottosistema di monitoring basato su **Prometheus**, facendo sì che ogni microservizio diventi un target di scraping standardizzato;
- supportare un ciclo di sviluppo locale su **kind** (Kubernetes in Docker), in cui le immagini Docker costruite localmente possono essere caricate nel cluster e referenziate dai Deployment con `imagePullPolicy: IfNotPresent`.

In questa configurazione, Kubernetes non viene utilizzato per funzionalità avanzate di autoscaling o multi-tenant, ma come **orchestratore coerente** di tutti i container necessari al funzionamento della piattaforma di flight monitoring e del suo sottosistema di osservabilità.

## 6.2. RISORSE KUBERNETES E MANIFEST UTILIZZATI

La descrizione del deployment è organizzata per categorie di risorse Kubernetes, con riferimento ai manifest che definiscono namespace, Deployment, Service, ConfigMap, Secret e componenti di monitoring.

### 6.2.1. NAMESPACE E ORGANIZZAZIONE LOGICA DEI COMPONENTI

Tutti i componenti del sistema sono deployati all'interno di un **namespace dedicato** (ad esempio `dsbd`), che svolge le seguenti funzioni:

- isolare i microservizi e i servizi infrastrutturali rispetto ad altri carichi di lavoro presenti nel cluster;
- consentire una gestione omogenea delle risorse (quota, policy, configurazione) a livello di namespace;

- semplificare la risoluzione DNS interna tramite nomi del tipo `service-name.dsbd.svc.cluster.local`.

All'interno del namespace, l'organizzazione logica segue i due insiemi principali:

- **microservizi applicativi**:
  - `user-manager-service`;
  - `data-collector-service`;
  - `alert-system-service`;
  - `alert-notifier-service`;
  - `api-gateway` (NGINX).
- **componenti infrastrutturali**:
  - `postgres` (User DB e Data DB);
  - `zookeeper` e `kafka`;
  - `kafka-ui`;
  - `prometheus`.

Questa struttura facilita sia la lettura dei manifest sia la navigazione delle risorse tramite `kubectl`, mantenendo chiara la distinzione tra responsabilità applicative e di supporto.

### 6.2.2. DEPLOYMENT E SERVICE PER I MICROSERVIZI APPLICATIVI

Per ciascun microservizio applicativo è definito un **Deployment** che specifica:

- l'immagine Docker da utilizzare (ad esempio `dsbd/user-manager-service:dev`, `dsbd/data-collector-service:dev`, `dsbd/alert-system-service:dev`, `dsbd/alert-notifier-service:dev`, `dsbd/api-gateway:dev`);
- il numero di repliche (*replicas*, inizialmente fissato a 1 per ambiente di sviluppo);
- le **label** che identificano i pod (ad esempio `app: user-manager-service`), utilizzate dai Service per il *service discovery*;
- i **container port** esposti internamente (8081 per User Manager, 8082 per Data Collector, 8083 per Alert System, 8084 per Alert Notifier, 80 per API Gateway);
- le variabili d'ambiente di base (ad esempio `SPRING_PROFILES_ACTIVE=docker` o equivalente), necessarie all'avvio dell'applicazione Spring Boot;
- eventuali **initContainer** per gestire dipendenze hard (ad esempio un initContainer `wait-for-kafka` che utilizza `busybox` per verificare la disponibilità del broker Kafka prima di avviare Alert System o Alert Notifier).

Accanto ai Deployment, per ogni microservizio è definito un **Service** di tipo **ClusterIP** che:

- espone la porta logica del microservizio all'interno del namespace (ad esempio `user-manager-service:8081`);

- permette al resto dell'architettura (gateway, altri servizi) di comunicare con il microservizio senza dipendere dall'IP del pod.

Nel caso dell'**API Gateway NGINX**, il Service è configurato in modo da:

- essere raggiungibile come entry-point logico del sistema, con una porta (ad esempio `80`) esposta internamente;
- poter essere eventualmente associato a una modalità di esposizione esterna (NodePort, Ingress o port-forward) descritta nel paragrafo 6.2.7.

Le **liveness** e **readiness probe** sono configurate utilizzando gli endpoint Actuator /actuator/health (e, se definiti, `/actuator/health/liveness` e `/actuator/health/readiness`), consentendo a Kubernetes di:

- riavviare i pod che non risultano più sani;
- evitare di inviare traffico a pod non ancora completamente inizializzati.

### 6.2.3. DEPLOYMENT E SERVICE PER KAFKA, ZOOKEEPER E KAFKA UI

La componente di messaggistica è implementata tramite **Kafka** e **ZooKeeper**, accompagnati da una **Kafka UI** per l'ispezione dei topic e dei consumer group.

- **ZooKeeper**
  - è deployato come Deployment dedicato o come pod singolo, con un Service associato (ad esempio `zookeeper:2181`);
  - fornisce il servizio di coordinamento necessario al broker Kafka in configurazione single-node.
- **Kafka Broker**
  - è deployato come Deployment che utilizza l'immagine Kafka (ad esempio `confluentinc/cp-kafka` o equivalente);
  - riceve via variabili d'ambiente i parametri di connessione a ZooKeeper e le porte di ascolto;
  - espone un Service (ad esempio `kafka:9092`) di tipo ClusterIP, utilizzato dai microservizi per connettersi al broker tramite `bootstrap.servers=kafka:9092`.
- **Kafka UI**
  - è deployata come Deployment separato che si connette al broker Kafka tramite la configurazione fornita in variabili d'ambiente o ConfigMap;
  - espone un Service (tipicamente NodePort o ClusterIP) che consente agli operatori di accedere all'interfaccia grafica per:
    - ispezionare i topic (`to-alert-system`, `to-notifier`, ecc.);
    - verificare il consumo da parte dei microservizi;
    - analizzare eventuali lag nei consumer group.

La configurazione è orientata a un **cluster single-broker** adatto ad ambienti di sviluppo e test, con dipendenze esplicite fra initContainer dei microservizi e disponibilità del broker Kafka.

### 6.2.4. DEPLOYMENT E SERVICE PER POSTGRESQL

Il database relazionale è gestito da un'istanza **PostgreSQL** che ospita sia **User DB** sia **Data DB** come database logici distinti.

- Il Deployment (o StatefulSet, se previsto) specifica:
  - l'immagine PostgreSQL da utilizzare;
  - le variabili d'ambiente per configurare utente, password e database principale (`POSTGRES_USER`, `POSTGRES_PASSWORD`, `POSTGRES_DB`);
  - eventuali mount di volume per la persistenza dei dati (volume locale o hostPath per ambiente kind).
- Il Service associato (ad esempio `postgres:5432`) è di tipo ClusterIP e rappresenta l'endpoint utilizzato dai microservizi Spring Boot per connettersi al database tramite JDBC, con URL del tipo:
  `jdbc:postgresql://postgres:5432/userdb` e
  `jdbc:postgresql://postgres:5432/datadb`.

I parametri di connessione (URL, user, password) sono forniti ai microservizi tramite variabili d'ambiente, tipicamente configurate in ConfigMap e Secret (si veda il paragrafo 6.2.6).

### 6.2.5. DEPLOYMENT E SERVICE PER PROMETHEUS

**Prometheus** è deployato come componente dedicato di monitoring e definito attraverso:

- un Deployment che utilizza l'immagine Prometheus, con:
  - un volume (configMapVolume) montato per caricare il file di configurazione `prometheus.yml`;
  - eventuali volumi per la persistenza del database di serie temporali (in ambiente di sviluppo può essere sufficiente lo storage effimero);
- un Service (ad esempio `prometheus:9090`) che espone l'interfaccia web e l'endpoint HTTP per le query PromQL.

La ConfigMap che contiene `prometheus.yml` definisce:

- uno `scrape_interval` coerente con le esigenze di osservabilità (ad esempio `15s`);
- uno o più `scrape_configs` dedicati ai microservizi applicativi (Data Collector, Alert System, Alert Notifier, User Manager), con `metrics_path: /actuator/prometheus` e `static_configs` che referenziano i Service Kubernetes nel namespace (ad esempio `data-collector-service:8082`, `alert-system-service:8083`, `alert-notifier-service:8084`, `user-manager-service:8081`);

- eventuali job di scraping per componenti infrastrutturali (Prometheus stesso, Kafka o altri target).

In questo modo Prometheus diventa il **collector centrale** delle metriche esposte dai microservizi, utilizzando la service discovery di Kubernetes per raggiungere gli endpoint `/actuator/prometheus` nei pod applicativi.

### 6.2.6. CONFIGMAP, SECRET E PARAMETRIZZAZIONE DELL'AMBIENTE

La parametrizzazione dell'ambiente è gestita tramite una combinazione di **ConfigMap** e **Secret**, che consentono di separare le informazioni di configurazione dal codice applicativo.

Esempi tipici includono:

- **ConfigMap applicative**:
  - URL dei servizi esterni (OpenSky base URL, Mail Server host);
  - configurazioni Kafka (`bootstrap.servers`, nomi dei topic, configurazioni di consumer/producer);
  - parametri di scheduling (intervallo di raccolta voli, timeout, finestra temporale di default);
  - variabili comuni per gli actuator e per il tagging delle metriche (ad esempio `MANAGEMENT_METRICS_TAGS_SERVICE`, `NODE_NAME`).
- **ConfigMap per Prometheus**:
  - definizione completa di `prometheus.yml`, con job di scraping, label di job e target da monitorare;
  - eventuali regole di alerting o configurazioni aggiuntive per l'osservabilità.
- **Secret**:
  - credenziali di accesso a PostgreSQL (`POSTGRES_PASSWORD`, password per gli utenti applicativi);
  - credenziali SMTP (`MAIL_USERNAME`, `MAIL_PASSWORD`) per il Mail Server;
  - eventuali token o client secret necessari per l'accesso ad API esterne (se previsti dalla configurazione effettiva).

I Deployment dei microservizi referenziano tali risorse tramite:

- `envFrom` (per importare interi ConfigMap/Secret come variabili d'ambiente);
- `env.valueFrom.secretKeyRef` o `env.valueFrom.configMapKeyRef` per singoli valori sensibili.

Questa impostazione consente di:

- cambiare configurazioni tra ambienti diversi (sviluppo, test, produzione) senza modificare le immagini Docker;

- gestire in modo separato i **segreti** e i parametri non sensibili, in linea con le buone pratiche di sicurezza e manutenibilità.

### 6.2.7. ESPOSIZIONE ESTERNA DI API GATEWAY E INTERFACCE DI AMMINISTRAZIONE

L'esposizione verso l'esterno è concentrata sui seguenti punti:

- **API Gateway (NGINX)**
  - può essere esposto:
    - tramite un Service di tipo **NodePort**, che apre una porta sul nodo per inoltrare il traffico verso il Service del gateway;
    - oppure tramite meccanismi di port-forward (`kubectl port-forward`) in ambiente di sviluppo;
  - rappresenta l'unico ingresso ufficiale per il traffico applicativo dei client verso `/api/....`
- **Kafka UI**
  - può essere esposta via NodePort o port-forward, consentendo agli operatori di:
    - ispezionare topic e consumer group;
    - verificare in tempo reale il flusso degli eventi nella pipeline Kafka.
- **Prometheus**
  - espone l'interfaccia web (porta 9090) tramite Service, tipicamente accessibile via port-forward in ambiente kind;
  - l'accesso consente di eseguire query PromQL sulle metriche raccolte e di analizzare il comportamento dei microservizi nel tempo.

Eventuali altre interfacce (ad esempio una dashboard per PostgreSQL o altre UI amministrative) possono essere esposte puntualmente nello stesso modo, mantenendo il principio per cui il *perimetro* dell'applicazione è mediato da endpoint controllati e facilmente identificabili nel namespace.

## 6.3. STRATEGIA DI BUILD E DISTRIBUZIONE DELLE IMMAGINI NEL CLUSTER (KIND)

L'ambiente di esecuzione è basato su **kind**, che esegue un cluster Kubernetes all'interno di container Docker. In questo contesto, la strategia di build e distribuzione delle immagini dei microservizi è progettata per:

- riutilizzare le **Docker image** già definite per l'ambiente Docker Compose;
- garantire che i nodi del cluster kind abbiano a disposizione le immagini con i tag attesi dai Deployment Kubernetes.

La pipeline operativa tipica è la seguente:

1. **Build degli artefatti applicativi**
    - per ciascun microservizio Spring Boot, esecuzione di build Maven/Gradle (ad esempio `mvn clean package -DskipTests`) per generare i JAR esecutivi;
2. **Build delle immagini Docker locali**
    - per ognuno dei microservizi:
        - costruzione dell'immagine con il tag atteso dai Deployment (ad esempio dsbd/user-manager-service:dev, dsbd/data-collector-service:dev, dsbd/alert-system-service:dev, dsbd/alert-notifier-service:dev, dsbd/api-gateway:dev);
    - le immagini vengono registrate nel Docker daemon locale, non in un registry remoto.
3. **Caricamento delle immagini nel cluster kind**
    - utilizzo del comando kind load docker-image per trasferire le immagini all'interno dei nodi del cluster kind, ad esempio:
        - `kind load docker-image dsbd/user-manager-service:dev --name <nome-cluster>`
        - `kind load docker-image dsbd/data-collector-service:dev --name <nome-cluster>`
        - e così via per gli altri microservizi;
    - questa operazione è coerente con l'uso di `imagePullPolicy: IfNotPresent` nei manifest: i pod utilizzeranno l'immagine locale se presente, senza tentare il pull da un registry esterno.
4. **Applicazione dei manifest Kubernetes**
    - una volta che le immagini sono state caricate nel cluster, viene eseguito `kubectl apply -f` sulla directory contenente i manifest (namespace, Deployment, Service, ConfigMap, Secret, Prometheus, Kafka, PostgreSQL, Kafka UI, API Gateway);
    - Kubernetes crea i pod utilizzando le immagini pre-caricate e configura automaticamente la rete interna e il service discovery.

Questa strategia consente di sviluppare e testare l'architettura completa su un cluster locale **self-contained**, senza dipendere da un registry Docker esterno, mantenendo coerenza tra i tag delle immagini utilizzati in Docker Compose e quelli richiesti dal deployment su Kubernetes.

## 6.4. CONFIGURAZIONE DI PROMETHEUS PER LO SCRAPING DEI TARGET APPLICATIVI

La configurazione di **Prometheus** è gestita tramite una **ConfigMap** che contiene il file `prometheus.yml`. L'obiettivo è rendere espliciti:

- i target di scraping corrispondenti ai microservizi che espongono metriche applicative;
- la periodicità con cui le metriche vengono raccolte;
- l'associazione logica fra job di scraping e componenti del sistema.

Un esempio di struttura della configurazione prevede:

- **Impostazioni globali**
  - `global.scrape_interval: 15s` per ottenere una risoluzione temporale adeguata senza introdurre overhead eccessivo;
  - eventuali parametri globali di timeout e valutazione.
- **Scrape job per i microservizi applicativi**
  - un job per il **Data Collector**:
    - `job_name: "data-collector"`
    - `metrics_path: "/actuator/prometheus"`
    - `static_configs` con target `data-collector-service.dsbd.svc.cluster.local:8082` o, in forma abbreviata, `data-collector-service:8082`;
  - un job per l'**Alert System**:
    - `job_name: "alert-system"`
    - target `alert-system-service:8083`;
  - un job per l'**Alert Notifier**:
    - `job_name: "alert-notifier"`
    - target `alert-notifier-service:8084`;
  - un job per lo **User Manager**:
    - `job_name: "user-manager"`
    - target `user-manager-service:8081`.
- **Eventuali job aggiuntivi**
  - un job per scrappare **Prometheus stesso** (`localhost:9090` all'interno del pod), utile per monitorare lo stato del server di monitoring;

    - job ulteriori per servizi infrastrutturali (ad esempio Kafka) se anch'essi espongono metriche compatibili.

Ogni job di scraping aggiunge label predefinite (`job`, `instance`) alle metriche raccolte. In combinazione con i tag configurati negli actuator (`service`, `node`), questo consente di:

- distinguere chiaramente i flussi metrici provenienti dai vari microservizi;
- filtrare e aggregare i dati in base al servizio (Data Collector, Alert System, Alert Notifier, User Manager) e al nodo del cluster;
- formulare query PromQL mirate per analizzare:
    - l'andamento delle metriche applicative (`opensky_*`, `alert_system_*`, `alert_notifier_*`, `email_*`);
    - la dinamica delle metriche tecniche (HTTP, JVM, sistema) dei singoli componenti.

La ConfigMap viene montata nel Pod di Prometheus come volume in **read-only**, e il container viene avviato con il path al file `prometheus.yml` fornito come parametro di configurazione. In questo modo, l'intero comportamento di scraping può essere versionato a livello di manifest e facilmente modificato o esteso in caso di introduzione di nuovi microservizi o target da monitorare.

# 7. ASPETTI NON FUNZIONALI E SCELTE PROGETTUALI

## 7.1. IMPATTO DEL MONITORING SU SCALABILITÀ, RESILIENZA E FAULT-TOLERANCE

L'introduzione sistematica del **monitoring applicativo e tecnico** incide direttamente sugli aspetti di *scalabilità*, *resilienza* e *fault-tolerance* dell'architettura. L'obiettivo non è soltanto misurare lo stato corrente del sistema, ma creare le condizioni per una evoluzione controllata e informata delle politiche di scaling e gestione dei guasti.

Dal punto di vista della **scalabilità**, il disaccoppiamento realizzato tramite Kafka, combinato con la visibilità sulle metriche, permette di:

- individuare i **colli di bottiglia** lungo la pipeline asincrona (ad esempio un Alert System sovraccarico rispetto al volume di eventi prodotti dal Data Collector), osservando:
    - tassi di produzione e consumo delle notifiche;
    - lag implicito fra pubblicazione su topic e elaborazione da parte dei consumer;
- dimensionare in modo più accurato il numero di repliche dei singoli microservizi, grazie a metriche di:
    - throughput per servizio (`service` tag nelle metriche Prometheus);
    - utilizzo delle risorse (CPU, memoria) e durata media delle operazioni critiche;
- distinguere problemi di **scalabilità locale** (un singolo servizio saturato) da problemi sistemici (ad esempio un database che non scala con il volume di richieste).

La **resilienza** trae beneficio dal monitoring in quanto:

- le metriche del client OpenSky (richieste totali, errori, fallback) rendono osservabile l'effetto del Circuit Breaker e permettono di verificare che la logica di *degradazione controllata* stia funzionando come atteso;
- le metriche del sistema di notifica (numero di notifiche consumate, errori di processing, errori SMTP, eventi rate-limited) consentono di individuare rapidamente:
    - problemi temporanei del Mail Server;
    - condizioni di saturazione o misconfigurazione del canale di notifica;
- gli endpoint di *health* esposti dai microservizi permettono a Kubernetes di effettuare automaticamente azioni di **self-healing** (ad esempio riavvio di pod non sani), riducendo l'impatto di errori transitori o memory leak.

In termini di **fault-tolerance**, la combinazione tra:

- architettura event-driven con **persistenza degli eventi** in Kafka;
- storage transazionale in PostgreSQL per utenti, interessi e record di volo;

- metriche di stato e di errore centralizzate in Prometheus

consente di gestire i guasti lungo la pipeline in modo non distruttivo. Ad esempio, un temporaneo malfunzionamento dell'Alert Notifier:

- non comporta la perdita delle notifiche già pubblicate sul topic Kafka;
- è visibile attraverso l'aumento del lag e delle metriche di errore;
- può essere mitigato tramite riavvio del servizio, scaling orizzontale o intervento mirato sulla configurazione SMTP.

La strumentazione di monitoring non crea la resilienza di per sé, ma rende possibile **misurarla, verificarla e migliorarla** in modo iterativo, trasformando l'infrastruttura Kubernetes in una piattaforma non solo esecutiva, ma anche osservabile e governabile.

## 7.2. OSSERVABILITÀ E DIAGNOSI DEI PROBLEMI SUPPORTATE DALLE NUOVE METRICHE

Le metriche applicative e tecniche introdotte abilitano un livello di **osservabilità** che va oltre il semplice controllo del *liveness* dei microservizi, rendendo possibile una diagnosi più precisa e veloce dei problemi.

Dal punto di vista **applicativo**, alcune famiglie di metriche giocano un ruolo centrale:

- le metriche di **interazione con OpenSky** permettono di separare:
  - errori dovuti al servizio esterno (time-out, errori HTTP, circuit breaker *open*);
  - errori interni al Data Collector (eccezioni applicative, problemi di parsing, errori di persistenza);
- le metriche dell'**Alert System** rendono osservabile:
  - il numero di valutazioni eseguite per finestra;
  - la quantità di notifiche generate per violazioni *HIGH* e *LOW*;
  - la durata media della valutazione;
- le metriche dell'**Alert Notifier** e dell'**Email Notification Service** consentono di correlare:
  - volume di notifiche consumate;
  - tasso di invii email riusciti/falliti;
  - effetto delle politiche di *rate limiting* del provider SMTP.

Dal punto di vista **tecnico**, le metriche esposte dagli actuator (HTTP, JVM, sistema) integrano la visibilità applicativa:

- le metriche `http.server.requests` permettono di identificare endpoint con latenza anomala o con tassi elevati di errori;

- le metriche JVM (heap usage, GC pause) consentono di individuare potenziali problemi di memoria o di garbage collection che potrebbero degradare le prestazioni;
- le metriche di uptime dei processi, combinate con le probe di health, chiariscono se eventuali errori osservati siano correlati a riavvii frequenti dei pod.

Questa combinazione di metriche è fondamentale per la **diagnosi dei problemi**. Alcuni esempi tipici:

- un aumento dei tempi di valutazione nell'Alert System, associato a una crescita del volume di snapshot per finestra, può suggerire la necessità di ottimizzare gli accessi al database o di introdurre meccanismi di batching più efficienti;
- un incremento degli errori SMTP accompagnato da un aumento dei tentativi di invio per utente può indicare:
    - un problema con il provider di posta;
    - una configurazione troppo aggressiva delle soglie che genera un numero eccessivo di notifiche;
- valori crescenti di `opensky_request_errors_total` uniti a un numero stabile di richieste possono segnalare una regressione lato servizio esterno oppure modifiche non compatibili alla configurazione del client.

L'osservabilità ottenuta non si limita alla fase di troubleshooting reattivo, ma supporta anche attività di **analisi preventiva** (ad esempio individuazione di trend di degrado) e di verifica degli effetti di modifiche rilasciate in produzione.

### 7.3. IMPATTO DEL SOTTOSISTEMA DI MONITORING SU PERFORMANCE E UTILIZZO DI RISORSE

L'introduzione di un sottosistema di monitoring comporta inevitabilmente un **costo in termini di performance e di risorse**, che deve essere gestito in modo consapevole. Le principali fonti di overhead sono:

- la **registrazione delle metriche** nei microservizi:
    - incrementi di contatori e aggiornamento di gauge a ogni operazione critica;
    - gestione del registry interno Micrometer;
- l'**esposizione delle metriche** tramite l'endpoint `/actuator/prometheus`, che comporta la serializzazione dei valori in formato testuale;
- lo **scraping periodico** da parte di Prometheus, che genera richieste HTTP aggiuntive verso ogni microservizio target;
- la **memorizzazione delle serie temporali** nel database interno di Prometheus, con conseguente consumo di CPU, memoria e storage.

Dal punto di vista dei microservizi, le scelte progettuali adottate mirano a mantenere l'overhead accettabile:

- l'aggiornamento delle metriche è concentrato in pochi punti chiave della pipeline (chiamate OpenSky, valutazioni, processing notifiche, invio email), evitando una strumentazione troppo fine-granulare su ogni singolo passaggio interno;
- l'uso di **counter** e **gauge** semplici riduce il costo di aggiornamento;
- lo scraping viene effettuato con un intervallo stabile e moderato (ad esempio 15 secondi), che rappresenta un buon compromesso tra freschezza dell'informazione e carico aggiuntivo sul sistema.

Il consumo di risorse da parte di **Prometheus** stesso viene gestito:

- limitando il numero di target (principalmente i microservizi di dominio critici);
- evitando configurazioni di scraping troppo aggressive;
- mantenendo l'orizzonte temporale dei dati in linea con le esigenze di osservazione (in ambiente di sviluppo sono sufficienti retention limitate).

In termini di **latenza percepita dal client**, il monitoring è progettato per avere un impatto minimo:

- le operazioni di aggiornamento delle metriche sono integrate all'interno dei flussi di business, ma non introducono I/O di rete aggiuntivo oltre allo scraping periodico;
- le richieste degli utenti continuano a passare attraverso API Gateway e microservizi senza essere rallentate da chiamate sincrone verso il sistema di monitoring.

Nel complesso, il **trade-off** tra overhead e valore informativo delle metriche è gestito privilegiando:

- la raccolta di indicatori realmente utili per il controllo operativo;
- l'adozione di granularità coerenti con gli scenari di diagnostica previsti, evitando di introdurre metriche ridondanti o a bassa utilità.

## 7.4. TRADE-OFF TRA COMPLESSITÀ ARCHITETTURALE E BENEFICI IN TERMINI DI CONTROLLO OPERATIVO

L'estensione del sistema con un sottosistema di monitoring e con una pipeline completa di deployment su Kubernetes introduce un livello di **complessità architetturale** superiore rispetto a una soluzione minimale. Tale complessità si manifesta principalmente in:

- numero maggiore di componenti da gestire (Prometheus, Kafka UI, configurazioni aggiuntive per i microservizi);
- maggior quantità di manifest Kubernetes (Deployment, Service, ConfigMap, Secret, configurazione Prometheus);
- necessità di definire e mantenere **metriche applicative** coerenti con il dominio e con gli obiettivi di osservabilità.

Questa complessità è però compensata da benefici significativi in termini di **controllo operativo**:

- possibilità di **diagnosticare rapidamente** problemi che, in assenza di metriche, richiederebbero analisi manuali estese sui log o riproduzioni complesse;
- capacità di valutare l'impatto di modifiche applicative o infrastrutturali (ad esempio aggiornamenti delle dipendenze, modifiche agli scheduler, variazione dei parametri di notifica) osservando l'evoluzione delle metriche nel tempo;
- supporto a decisioni di **scaling** basate su dati oggettivi, anziché su ipotesi o osservazioni aneddotiche;
- possibilità di definire in futuro politiche di **alerting automatizzato**, basate su soglie, trend o regole di anomalia applicate alle metriche raccolte.

Il disegno adottato mantiene la complessità entro limiti gestibili grazie ad alcune scelte:

- focalizzare il monitoring su una **sotto-base di microservizi** ritenuti critici (Data Collector, Alert System, Alert Notifier, User Manager), evitando di strumentare indiscriminatamente ogni componente secondario;
- utilizzare un unico **stack di monitoring** (Prometheus + Micrometer) ben integrato con Spring Boot, riducendo la frammentazione tecnologica;
- centralizzare la configurazione di scraping in un solo file `prometheus.yml`, versionato tramite ConfigMap.

L'effetto complessivo è un sistema più articolato ma anche **più trasparente**, in cui ogni servizio rilevante è osservabile in modo coerente e integrato, e in cui le decisioni di gestione operativa possono essere prese su base quantitativa anziché puramente qualitativa.

# 8. INQUADRAMENTO NEL CICLO EVOLUTIVO DEL SISTEMA

## 8.1. COMPONENTI E SEZIONI DELLE VERSIONI PRECEDENTI ANCORA PIENAMENTE VALIDE

L'evoluzione che porta alla versione corrente non stravolge il **dominio applicativo** né le responsabilità fondamentali dei microservizi introdotti nelle versioni precedenti. Gran parte delle decisioni architetturali e delle descrizioni già formalizzate rimangono **pienamente valide** e continuano a costituire il riferimento principale per comprendere il comportamento funzionale del sistema.

In particolare, restano immutati i seguenti elementi:

- **Modello concettuale del dominio**
    - Il ruolo dell'utente come soggetto che registra interessi verso specifici aeroporti.
    - Il concetto di *interesse aeroporto–utente* come entità logica che collega l'utente a un aeroporto, con eventuali attributi aggiuntivi (come le soglie di interesse).
    - La rappresentazione dei voli come eventi associati a un aeroporto, con informazioni su arrivi e partenze, persistenza in *Data DB* e interrogazione tramite API.
- **Responsabilità dei microservizi applicativi**
    - *User Manager* continua a essere il **single source of truth** per la gestione e l'interrogazione degli utenti, con API REST stabili e servizio di validazione gRPC.
    - *Data Collector* rimane il responsabile della **raccolta periodica** dei voli da OpenSky, del popolamento del database dei dati e della gestione degli interessi verso gli aeroporti.
    - *Alert System* mantiene il ruolo di motore di **valutazione delle soglie**, consumando gli eventi di aggiornamento finestra e producendo notifiche di violazione.
    - *Alert Notifier* conserva la responsabilità di trasformare le notifiche in **email** verso gli utenti, fungendo da terminazione della pipeline di alerting.
- **Architettura logica e flussi di base**
    - La suddivisione tra microservizi, database relazionali e broker Kafka, già descritta nelle versioni precedenti, resta valida come **architettura logica complessiva**.
    - I flussi di interazione **REST** per la gestione utenti/interessi e per l'interrogazione dei voli seguono lo stesso pattern, con API Gateway come punto di ingresso e microservizi applicativi come backend logici.

    - I flussi **event-driven** che legano Data Collector, Alert System e Alert Notifier tramite Kafka mantengono la stessa semantica funzionale: produzione di eventi di finestra, valutazione delle soglie, generazione di notifiche e invio email.

- **Modellazione dei dati e schema dei database**
    - La distinzione tra **User DB** (anagrafica utenti) e **Data DB** (interessi, aeroporti, record di volo) rimane invariata.
    - Le relazioni fondamentali tra tabelle (ad esempio tra utenti, interessi e aeroporti) e le modalità di utilizzo da parte dei microservizi restano aderenti alla descrizione esistente.
- **Documentazione delle API di dominio**
    - Le sezioni dedicate alla descrizione puntuale degli endpoint REST e dei relativi payload, definite nelle versioni precedenti, rimangono il riferimento principale per chi deve integrare client applicativi o effettuare test funzionali.

In questa prospettiva, la nuova documentazione non sostituisce la descrizione del **core funzionale** già formalizzato, ma vi si appoggia, estendendolo e completandolo sotto il profilo del deployment su Kubernetes e del monitoring applicativo e tecnico.

### 8.2. COMPONENTI E SEZIONI INTEGRATE O SOSTITUITE DALLA VERSIONE CORRENTE

La versione corrente introduce un **salto di livello architetturale** concentrato soprattutto su:

- modalità di deployment (passaggio da orchestrazioni basate esclusivamente su Docker alla gestione tramite Kubernetes su cluster kind);
- integrazione di un sottosistema di monitoring basato su **Prometheus** e sulle metriche esposte dai microservizi tramite Actuator/Micrometer;
- aggiornamento e arricchimento dei diagrammi architetturali e di sequenza, ora esplicitamente allineati al contesto Kubernetes e al modello di osservabilità.

Di conseguenza, alcune componenti e sezioni della documentazione precedente risultano:

- **integrate** all'interno di una vista più ampia;
- **superate** per quanto riguarda la rappresentazione dell'infrastruttura e delle modalità di esecuzione, pur rimanendo utili sul piano storico e funzionale.

In particolare:

- **Architettura infrastrutturale e diagrammi architetturali**
    - I diagrammi architetturali delle versioni precedenti, centrati su un deployment principalmente Docker/ Docker Compose, restano corretti rispetto al contesto originario, ma vengono **superati** dalla nuova rappresentazione architetturale su Kubernetes con sottosistema di monitoring Prometheus.

- La nuova architettura mantiene gli stessi microservizi e servizi infrastrutturali, ma li colloca in un ambiente orchestrato da Kubernetes, con introduzione esplicita di namespace, Deployment, Service, ConfigMap, Secret e componenti dedicati al monitoring.

- **Sezioni di deployment e run-time environment**
  - Le sezioni che descrivono il deployment tramite Docker Compose e i relativi file di configurazione continuano a essere valide per la riproduzione dell'ambiente precedente.
  - Ai fini della versione corrente, però, il riferimento principale per l'esecuzione della piattaforma è costituito dalle nuove sezioni dedicate al **deployment su Kubernetes**, che descrivono la configurazione del cluster kind, le strategie di build e distribuzione delle immagini, la definizione dei manifest e l'integrazione di Prometheus.
- **Diagrammi di sequenza della pipeline di notifica**
  - I diagrammi di sequenza precedenti che illustrano la pipeline di notifica asincrona (Data Collector → Alert System → Alert Notifier → Mail Server) rimangono validi sul piano funzionale.
  - La versione corrente introduce una **nuova variante** dei diagrammi, che integra le metriche applicative e la loro raccolta da parte di Prometheus, rendendo esplicite:
    - le chiamate ai componenti di metriche Micrometer;
    - l'effetto delle operazioni di business sulle misure esposte per il monitoring.
  - Per la comprensione della logica di dominio è sufficiente fare riferimento ai diagrammi originari; per la comprensione del comportamento osservabile e del ciclo di monitoring è invece preferibile utilizzare i nuovi diagrammi arricchiti.
- **Aspetti di configurazione e gestione dei parametri**
  - Le sezioni che descrivono la configurazione tramite file `.env` e variabili d'ambiente per Docker sono ora **integrate** da una descrizione più ampia che include ConfigMap e Secret Kubernetes.
  - I concetti alla base della parametrizzazione (isolamento delle credenziali, separazione tra configurazione e codice) restano invariati, ma i dettagli operativi sono aggiornati e ampliati nella versione corrente per tenere conto del modello Kubernetes.

Nel complesso, la versione corrente **non invalida** la documentazione esistente, ma la **ricontestualizza**: i documenti precedenti costituiscono il riferimento per la logica applicativa e l'architettura su Docker, mentre la nuova relazione si pone come strato superiore che descrive l'evoluzione verso Kubernetes e l'integrazione del monitoring,

sostituendo le sezioni infrastrutturali e di deployment laddove esse confliggano con le scelte aggiornate.

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

Per facilitare la consultazione congiunta della documentazione, è opportuno esplicitare alcuni **riferimenti incrociati** fra i capitoli e i diagrammi delle versioni precedenti e le sezioni della versione corrente.

A livello concettuale, si può tracciare la seguente mappa di corrispondenza:

- **Architettura logica e microservizi**
    - La descrizione dell'architettura a microservizi, con separazione tra User Manager, Data Collector, Alert System, Alert Notifier, database relazionali e broker Kafka, è documentata nelle sezioni architetturali delle versioni precedenti.
    - Tali sezioni rimangono il riferimento per comprendere:
        - il perimetro funzionale di ciascun microservizio;
        - le interazioni principali tramite REST, gRPC e Kafka.
    - Il capitolo 2 della versione corrente ("Visione d'insieme dell'architettura con monitoring su Kubernetes") si collega direttamente a quelle sezioni, aggiornandone la **prospettiva infrastrutturale** e introducendo il contesto Kubernetes e Prometheus.
- **Modello dei dati e schema dei database**
    - Le parti dedicate alla modellazione dei database (diagrammi ER, descrizione delle tabelle e dei vincoli) nelle versioni precedenti restano l'origine della verità per il modello persistente.
    - Nel capitolo 3 della versione corrente, il modello delle metriche si appoggia implicitamente a quelle strutture dati, in quanto le metriche di valutazione delle soglie e di generazione delle notifiche riflettono il contenuto e l'evoluzione delle entità memorizzate in **User DB** e **Data DB**.
- **Pipeline di raccolta e notifica**
    - I diagrammi di sequenza originari relativi a:
        - registrazione interessi;
        - raccolta periodica dei voli;
        - pipeline di notifica asincrona

        costituiscono la base per comprendere il **flusso funzionale** end-to-end.
    - Il capitolo 4 della versione corrente, insieme ai nuovi diagrammi di sequenza, aggiunge una vista **ortogonale** focalizzata sui punti in cui il flusso aggiorna

le metriche esposte a Prometheus e sui meccanismi di scraping e consultazione da parte degli operatori.

- **Sezioni di deploy e run-time precedente vs. corrente**
    - Le sezioni dedicate al deployment su Docker e Docker Compose nelle versioni precedenti sono utili come riferimento per ambienti di esecuzione più semplici o per una comprensione graduale dell'evoluzione infrastrutturale.
    - Il capitolo 6 della versione corrente, che descrive il deployment su Kubernetes e l'integrazione del monitoring, ne rappresenta la **naturale prosecuzione**, mantenendo un legame ideale con le scelte di base (immagini Docker, configurazioni applicative) ma riorganizzandole in manifest Kubernetes.
- **Aspetti non funzionali e valutazioni complessive**
    - Le valutazioni già espresse in merito a robustezza, separazione delle responsabilità, chiarezza dei contratti API e coerenza del modello di dati mantengono la loro validità e vengono ora arricchite dal punto di vista dell'osservabilità.
    - Il capitolo 7 della versione corrente si colloca direttamente in continuità con tali valutazioni, introducendo l'analisi dell'impatto del monitoring su scalabilità, resilienza, fault-tolerance e controllo operativo.

Questa rete di riferimenti permette a chi legge di:

- utilizzare la documentazione delle versioni precedenti come **base funzionale e concettuale**;
- impiegare la presente relazione come **strato evolutivo** focalizzato su Kubernetes e monitoring;
- ricostruire il percorso di crescita del sistema dalla configurazione iniziale a quella corrente, mantenendo coerenza tra architettura logica, modello dei dati, flussi di interazione e scelte infrastrutturali.

# 9. CONCLUSIONI E SVILUPPI FUTURI

## 9.1. SINTESI DELLE ESTENSIONI FUNZIONALI E ARCHITETTURALI INTRODOTTE

La versione corrente del sistema di **flight monitoring** consolida il percorso evolutivo intrapreso nelle versioni precedenti, introducendo due direttrici principali di estensione:

- il passaggio a un **deployment orchestrato su Kubernetes**;
- l'integrazione di un **sottosistema di monitoring applicativo e tecnico** basato su Prometheus e Micrometer.

Dal punto di vista architetturale, i microservizi già consolidati (User Manager, Data Collector, Alert System, Alert Notifier, API Gateway) vengono collocati all'interno di un **namespace Kubernetes dedicato**, gestiti tramite Deployment e Service specifici e affiancati dalle componenti infrastrutturali (PostgreSQL, Kafka, ZooKeeper, Kafka UI, Prometheus). Questo spostamento dall'orchestrazione esclusivamente basata su Docker verso Kubernetes consente di:

- formalizzare in modo esplicito i confini tra componenti applicativi e infrastrutturali;
- sfruttare i meccanismi nativi di **scheduling**, **scaling** e **self-healing** di Kubernetes;
- uniformare la modalità di configurazione tramite ConfigMap e Secret, mantenendo netta la separazione tra codice e parametri di ambiente.

Sul piano del monitoring, la versione corrente introduce un modello coerente di **metriche applicative** esposte dai microservizi tramite l'endpoint `/actuator/prometheus`, successivamente raccolte da Prometheus. In particolare:

- il **Data Collector** e l'**OpenSky Client** espongono contatori e misure di latenza relativi alle chiamate verso il servizio esterno, alla frequenza di errore e all'utilizzo dei meccanismi di fallback;
- l'**Alert System** espone metriche legate al numero di valutazioni effettuate, alle notifiche generate e ai tempi di elaborazione per finestra;
- l'**Alert Notifier** espone indicatori sul consumo delle notifiche e sui tempi di processing dei messaggi Kafka;
- l'**Email Notification Service** espone metriche specifiche per l'invio delle email, inclusi tempi di consegna, esito degli invii e impatto di eventuali politiche di throttling.

Queste metriche si affiancano a quelle tecniche fornite dagli actuator (HTTP, JVM, sistema), completando una **vista osservabile** del comportamento della pipeline asincrona di alerting, dalla raccolta dei voli sino alla consegna delle notifiche via email.

L'insieme di queste estensioni mantiene intatti i principi fondanti delle versioni precedenti (separazione dei domini, microservizi autonomi, contratti chiari, modello relazionale coerente), collocandoli in un contesto infrastrutturale più maturo e dotato di strumenti nativi per il controllo operativo.

## 9.2. VALUTAZIONE CRITICA DELLE SCELTE PROGETTUALI ADOTTATE

Le scelte progettuali adottate nella versione corrente si distinguono per **continuità** con le versioni precedenti e per **incrementalità controllata** dell'evoluzione architetturale.

In primo luogo, la migrazione verso Kubernetes è stata impostata in modo da **preservare l'architettura logica esistente**, evitando stravolgimenti del dominio applicativo e dei contratti API già definiti. I microservizi mantengono le stesse responsabilità e gli stessi flussi (REST, gRPC, Kafka), mentre la piattaforma sottostante viene arricchita con:

- definizione esplicita di Deployment e Service per ciascuna componente;
- gestione coerente della configurazione tramite ConfigMap e Secret;
- isolamento logico mediante namespace dedicato.

Questa impostazione favorisce la **leggibilità** della soluzione e rende più agevole il passaggio da ambienti di sviluppo locali a contesti più strutturati, grazie alla naturalità con cui le risorse Kubernetes descrivono lo stato desiderato del sistema.

La progettazione del sottosistema di monitoring mostra una particolare attenzione alla **coerenza semantica delle metriche**. Ogni microservizio espone:

- metriche strettamente allineate al proprio ruolo di dominio (richieste verso OpenSky, valutazioni di soglia, consumi di notifiche, invio di email);
- metriche tecniche che permettono di correlare lo stato applicativo con il comportamento dell'infrastruttura.

Questa impostazione evita una strumentazione dispersiva e concentra l'osservabilità su ciò che è realmente significativo per il **controllo operativo** del sistema.

Il disegno complessivo valorizza inoltre:

- la **separazione delle responsabilità** tra microservizi, che rimane chiara e ben documentata;
- l'uso combinato di pattern consolidati (REST per l'accesso sincrono, gRPC per la validazione tipizzata, Kafka per la pipeline asincrona, circuit breaker verso il servizio esterno);
- la possibilità di **estendere ulteriormente** il sistema senza dover rimettere in discussione le fondamenta: il passaggio a Kubernetes e l'introduzione di Prometheus preparano il terreno per future integrazioni di strumenti di osservabilità aggiuntivi, politiche di scaling più sofisticate e automazioni tipiche degli ambienti cloud-native.

Nel complesso, le scelte effettuate dimostrano un equilibrio consapevole fra **modernizzazione infrastrutturale**, **mantenimento della chiarezza architetturale** e **attenzione alla manutenibilità** nel medio-lungo periodo.

## 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 nelle versioni precedenti sono state ormai recepite nella versione corrente – in particolare l'introduzione di un **message broker** per la pipeline event–driven di alerting, l'adozione di un **API Gateway**, i meccanismi di **resilienza** verso OpenSky e il **deployment su Kubernetes** con **esposizione delle metriche applicative verso Prometheus** – mentre altre sono state intenzionalmente lasciate fuori dallo scope attuale e rappresentano il naturale proseguimento della roadmap. Il fuoco di questo paragrafo è pertanto sugli sviluppi che rimangono aperti, ma che risultano già allineati con le scelte architetturali e di monitoring effettuate.

Un **primo asse di evoluzione** riguarda il *rafforzamento del livello funzionale e analitico*, facendo leva sulle strutture di dominio e sulle pipeline di raccolta dati già disponibili:

- **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), così da esporre endpoint orientati a reporting e dashboarding senza sovraccaricare i percorsi transazionali.
- **Adozione del pattern CQRS**, separando in modo netto il modello di scrittura (registrazione utenti, gestione degli interessi, ingestione dei voli) dal modello di lettura, servito da *read model* specifici per interrogazioni analitiche, aggregazioni su finestre temporali e viste sintetiche per utenti avanzati.
- **Utilizzo sistematico dell'*outbox pattern*** per propagare in modo affidabile gli eventi di dominio dal perimetro transazionale verso canali asincroni o sistemi analitici, garantendo che ogni cambiamento rilevante (creazione utente, registrazione interesse, raccolta voli) possa essere riflesso in modo consistente nei *read model* o nelle piattaforme esterne.
- **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 condizioni composite; tale evoluzione consentirebbe di trasformare il sistema da semplice motore di soglie statiche a componente in grado di intercettare pattern operativi più sofisticati.

Un **secondo asse di evoluzione** è di natura *architetturale–infrastrutturale* ed è orientato ad aumentare ulteriormente disaccoppiamento, resilienza e osservabilità:

- **Generalizzazione del paradigma event–driven oltre la sola pipeline di alerting**, con la pubblicazione sistematica di eventi di dominio (ad esempio *UserRegistered*, *InterestCreated*, *FlightsCollected*) e l'eventuale introduzione di uno *schema registry* per governare in modo rigoroso l'evoluzione dei payload e la compatibilità fra produttori e consumatori.
- **Definizione di *saga* per processi distribuiti complessi**, in grado di orchestrare workflow multi–servizio (gestione utenti, interessi, moduli analitici, eventuali

componenti di billing o compliance) tramite step coordinati e azioni compensative, riducendo il rischio di inconsistenze in presenza di fallimenti parziali.

- **Evoluzione del ruolo dell'API Gateway in vero punto di enforcement delle policy**, incaricato non solo del semplice *reverse proxying*, ma anche di:
    - gestire autenticazione e autorizzazione;
    - applicare politiche di *rate limiting* e protezione;
    - centralizzare *logging*, *routing* e *versioning* delle API;
    - veicolare *header* tecnici a supporto di tracciamento e correlazione delle richieste.
- **Completamento della *observability stack* attorno alle metriche già esposte verso Prometheus**, introducendo ad esempio dashboard dedicate (Grafana), log strutturati centralizzati e meccanismi di *tracing distribuito*; ciò consentirebbe di affiancare alle metriche tecniche e applicative già disponibili una visibilità end–to–end sulle chiamate, sulle interazioni fra microservizi, sulla pipeline di ingestione e sulla pipeline di notifica.

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, così da mantenere reattive le query operative anche in presenza di volumi crescenti.
- **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 e *capacity planning*.
- **Valutazione di database e motori specializzati per serie temporali e stream**, in grado di gestire con maggiore efficienza query temporali complesse, aggregazioni su finestre scorrevoli o *tumbling* e richieste ad alto throughput tipiche di scenari *near real–time*.
- **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; gli output di tali componenti potrebbero alimentare l'Alert System con segnali più evoluti rispetto alle soglie statiche, abilitando forme di *alerting proattivo* basate su indicatori predittivi e *anomaly scores*.