UNIVERSITÀ DEGLI STUDI DI CATANIA

Corso di Laurea Magistrale in Ingegneria Informatica LM-32

Relazione Homework 3

Corso di Distributed Systems and Big Data

Anno Accademico 2025-2026

Studente:
Riccardo Maria Villaggio
Matricola: 1000084767

Indice

1 Introduzione

Il presente elaborato illustra le scelte progettuali e implementative relative alla terza ed ultima iterazione del progetto realizzato per il corso di Distributed Systems and Big Data. Il sistema oggetto di studio, un'architettura a microservizi per il monitoraggio del traffico aereo e la gestione degli interessi degli utenti, è stato oggetto di una profonda rifattorizzazione infrastrutturale, passando da un deployment statico basato su Docker Compose a un'orchestrazione dinamica su piattaforma **Kubernetes** (utilizzando un cluster locale Kind). Altro obiettivo primario di questa fase è stato l'introduzione di un meccanismo di **White-box Monitoring** basato su **Prometheus**, in particolare: ogni microservizio soggetto a monitoraggio espone metriche specifiche (Counter e Gauge) arricchite con label che identificano il servizio e il nodo Kubernetes ospitante, permettendo un'analisi dettagliata delle performance e dello stato del sistema.

1.1 Stack Tecnologico Aggiornato

Per supportare i nuovi requisiti di orchestrazione e monitoraggio, lo stack tecnologico originale è stato esteso con nuovi componenti. Di seguito si riporta l'elenco completo delle principali tecnologie adottate:

- **Core Applicativo & Runtime:**
  - **Python 3.11 & Flask:** Framework per lo sviluppo dei microservizi RESTful.
  - **gRPC & Protobuf:** Protocollo ad alte prestazioni per la comunicazione inter-processo sincrona (User Manager $\leftrightarrow$ Data Collector).
  - **Docker:** Container Runtime per la pacchettizzazione dei servizi.
- **Orchestrazione & Infrastruttura:**
  - **Kubernetes (K8s):** Piattaforma di orchestrazione per la gestione automatizzata del deployment, scaling e management dei Pod.
  - **Kind (Kubernetes in Docker):** Strumento utilizzato per simulare un cluster Kubernetes multi-nodo in ambiente di sviluppo locale.
  - **NGINX Ingress Controller:** Componente per la gestione dell'accesso esterno (L7 Load Balancing) e terminazione TLS.
- **Persistenza & Messaging:**
  - **MySQL 8.0:** RDBMS per la persistenza relazionale, gestito tramite *StatefulSet* e *PersistentVolumeClaims*.
  - **Apache Kafka (KRaft mode):** Piattaforma per il messaging distribuito, operante senza ZooKeeper per la gestione asincrona delle notifiche.
  - **kafka-python:** Client Python per l'interazione Producer/Consumer con il broker.
- **Observability & Monitoring:**
  - **Prometheus:** Toolkit di monitoraggio e time-series database per la raccolta delle metriche (scraping).

  - **prometheus-client:** Libreria Python utilizzata per strumentare il codice e implementare metriche custom (Counter, Gauge) ed esporre l'endpoint `/metrics`.

- **Utilities:**
  - **APScheduler:** Libreria per la schedulazione in-process dei job di raccolta dati nel Data Collector.
  - **smtplib & ssl:** Moduli standard per l'invio sicuro di email di alert.

2 Ottimizzazioni e Miglioramenti

Prima di procedere con il deployment su Kubernetes, è stata effettuata una revisione approfondita del codice per migliorare il sistema e prepararlo all'ambiente distribuito.

2.1 Gestione della Consistenza Distribuita: Pattern Saga

Per garantire la consistenza dei dati tra *User Manager* e *Data Collector* durante l'operazione critica di eliminazione dell'account, è stato rifinito e adottato il **Pattern Saga basato su Orchestrazione**. L'operazione, implementata nell'endpoint `DELETE /users/<email>`, è strutturata identificando ruoli precisi per le transazioni coinvolte:

- **Pivot Transaction (Remote):** La chiamata gRPC `delete_interests` verso il Data Collector funge da punto di decisione ("Go/No-Go"). Il successo di questa operazione rappresenta il punto di non ritorno che impegna il sistema a completare la cancellazione. Se questa fase fallisce, la transazione locale non viene nemmeno avviata, garantendo l'atomicità senza effetti collaterali. In particolare, fintanto che non avviene con successo la rimozione degli interessi dal Data Collector, l'utente rimane integro nel sistema, senza effettuare la cancellazione sulla sessione del database MySQL.
- **Retryable Transaction (Local):** La successiva commit sul database MySQL dello User Manager è identificata come transazione *Retryable*, ovvero un'azione che, una volta superata la Pivot, deve essere garantita fino al successo.

Per gestire i fallimenti nella fase locale senza ricorrere a complesse transazioni di compensazione, è stata implementata una **Strategia di Recovery Ibrida** direttamente nel codice applicativo:

1. **Short-Term Retry (Backend):** L'orchestratore esegue un ciclo di tentativi automatici con backoff, per superare errori transitori del database locale.
2. **User-Driven Retry (End-to-End):** In caso di guasto persistente del database locale dopo i tentativi automatici, l'eccezione viene propagata al client (HTTP 500). Grazie alla proprietà di **Idempotenza** implementata nel servizio Data Collector (che restituisce successo anche se gli interessi sono già stati rimossi), il successivo tentativo manuale dell'utente agisce come meccanismo di *Forward Recovery*, permettendo al sistema di convergere allo stato consistente finale.

```python
# PHASE 1: PIVOT TRANSACTION (gRPC Remote Cleanup)
# Verifichiamo il successo della pulizia remota sul Data Collector.
# Questo funge da Point of No Return.
grpc_success, grpc_msg = data_collector_client.delete_interests(clean_email)

if not grpc_success:
    # Fallimento remoto -> Abort dell'operazione.
    # Non tocchiamo il DB locale, garantendo consistenza.
    db.session.rollback() # Safety Net (anche se non dovrebbe essere necessario, la sessione locale non e' stata modificata)
    return jsonify({
        "error": "Errore comunicazione Data Collector (Pivot Failed)",
        "details": grpc_msg
    }), 503

# PHASE 2: RETRYABLE TRANSACTION (Local Commit)
# La remota e' andata a buon fine. Procediamo con il commit locale.
MAX_RETRIES = 3
for attempt in range(MAX_RETRIES):
    try:
        # Tentativo di commit locale (Retryable Operation)
        user_to_delete = db.session.merge(user)
        db.session.delete(user_to_delete)
        db.session.commit()
        return jsonify({"message": "Utente eliminato con successo"}), 200

    except Exception as db_err:
        db.session.rollback()
        # Short-Term Retry logic
        if attempt < MAX_RETRIES - 1:
            time.sleep(0.5)
            continue
        else:
            # Fallimento persistente -> Delego al client (User-Driven Retry)
            raise db_err
```

Listing 1: Implementazione Pattern Saga nello User Manager

2.2 Evoluzione delle Strategie di Resilienza e Orchestrazione

La transizione dall'ambiente di sviluppo locale all'orchestrazione su Kubernetes ha richiesto un cambio di paradigma fondamentale: il passaggio da una filosofia *Fail-Fast*, utile per il debugging, a una architettura *Fault-Tolerant* orientata alla disponibilità in produzione. Di seguito vengono dettagliate le aree critiche oggetto di rifattorizzazione, valutate tramite analisi preliminare su Docker Compose, ma che hanno trovato piena espressione nell'ambiente Kubernetes.

2.2.1 Indurimento degli Healthchecks (Transizione L4 → L7)

Un'analisi critica del comportamento all'avvio ha evidenziato che i controlli di salute basati sulla semplice raggiungibilità di rete (TCP Check/Ping - Livello 4) generavano *Race Conditions*. Servizi complessi come i Database o i Message Broker aprivano la porta TCP prima di aver completato le procedure interne di inizializzazione, causando il crash immediato dei microservizi dipendenti che tentavano la connessione prematuramente.

Per mitigare questo rischio, gli healthcheck utilizzati su Docker Compose sono stati elevati al Livello Applicativo (L7), verificando l'effettiva operatività funzionale:

- **MySQL (Autenticazione):** Adozione del comando `mysqladmin ping` corredato esplicitamente delle credenziali di root (`-p${MYSQL_ROOT_PASSWORD}`). Questo garantisce che il DBMS sia non solo attivo, ma che il sottosistema di autenticazione sia pronto ad accettare e validare le connessioni dell'applicazione.
- **Kafka (Metadata):** Adozione del comando `kafka-topics -list -bootstrap-server localhost:9092` per verificare che il Broker sia attivo, che il Controller sia stato eletto e che i metadati del cluster siano accessibili.
- **Microservizi:** Utilizzo degli endpoint `/health` dedicati, che verificano lo stato interno dell'applicazione (es. thread attivi), e la validità delle connessioni verso le dipendenze per garantire il corretto ordine di avvio tra i servizi.
- **Gestione Dipendenze:** Nel `docker-compose.yml`, la direttiva `depends_on` con la condizione `service_healthy` è stata utilizzata per gestire tutte le dipendenze esistenti tra i componenti del sistema, creando una catena di avvio deterministica che previene il crash dei servizi all'avvio.

2.2.2 Resilienza Applicativa e Prevenzione del CrashLoopBackOff

A differenza di Docker Compose, Kubernetes non garantisce un ordine di avvio deterministico dei Pod. Se un microservizio Python tentasse di connettersi al Database immediatamente all'avvio e fallisse, il processo terminerebbe con un codice di errore. Kubernetes reagirebbe riavviando il Pod applicando un **Exponential Backoff** (ritardi crescenti: 10s, 20s, 40s...), ritardando in modo inaccettabile la convergenza del sistema (specialmente durante la fase di Cold Start).

Per risolvere questo problema strutturale, il codice applicativo è stato reso **Self-Reliant**:

1. **Bootstrap Resiliente (wait_for_db):** È stata implementata una funzione di polling attivo all'avvio, grazie alla quale il servizio entra in un ciclo di attesa finché il DB non risponde. In questo modo il Pod rimane nello stato *Running* consumando i tentativi della **Startup Probe** (configurata con soglie calibrate in base ai tempi di avvio dei DB, si faccia riferimento alla Sezione 4.2.3) invece di terminare. Questo permette al servizio di completare l'avvio ed entrare in stato *Ready* non appena il database diviene disponibile, senza subire le penalità di tempo imposte dai riavvii del Pod.
2. **Runtime Resilience (pool_pre_ping):** La configurazione dell'ORM SQLAlchemy è stata arricchita con l'opzione `pool_pre_ping: True`. Questo assicura che ogni connessione prelevata dal pool venga testata (tramite `SELECT 1`) prima dell'uso, gestendo in modo trasparente la caduta di connessioni inattive o i riavvii dei nodi DB, tipici degli ambienti containerizzati.

3. **Idempotenza della Schema Initialization (Gestione Race Conditions):** Con l'introduzione della scalabilità orizzontale (`replicas: 2`), l'esecuzione concorrente del comando `db.create_all()` da parte di più Pod generava una *Race Condition* critica: il secondo Pod falliva tentando di creare tabelle già esistenti. È stata quindi introdotta una gestione esplicita dell'eccezione `OperationalError` (codice MySQL 1050 "Table already exists"), rendendo l'operazione di inizializzazione **idempotente**. Questo permette a tutte le repliche di completare l'avvio con successo, indipendentemente dall'ordine di esecuzione, considerando l'esistenza delle tabelle non come un errore bloccante ma come uno stato desiderato già raggiunto.

```python
def wait_for_db(app):
    print("Verifica connessione al Database...", flush=True)
    with app.app_context():
        while True:
            try:
                # Polling attivo sulla connessione
                with db.engine.connect() as connection:
                    print("Database pronto! Connessione stabilita.", flush=True)
                    return
            except Exception as e:
                print(f"Database non pronto ({str(e)}). Riprovo...", flush=True)
                time.sleep(3)

# ...

with app.app_context():
    wait_for_db(app) # Bloccante fino a DB available
    try:
        db.create_all()
    except OperationalError as e:
        # Gestione Race Condition per avvio concorrente (Replicas > 1)
        orig = getattr(e, "orig", None)
        if orig and orig.args[0] == 1050: # MySQL Error: Table already exists
            print("Tabelle gia' esistenti (Race Condition gestita).", flush=True)
        else:
            raise
```

Listing 2: Bootstrap Resiliente e Gestione Concorrenza DB

2.2.3 Ottimizzazione delle Connessioni gRPC (Keepalive e Load Balancing)

L'adozione di gRPC per la comunicazione sincrona tra *User Manager* e *Data Collector* introduce la sfida della gestione delle connessioni persistenti (HTTP/2) in un ambiente dinamico come Kubernetes. Oltre al rischio di *Idle Timeout* imposto dai componenti di rete intermedi, una criticità frequente è rappresentata

dalla presenza di connessioni "zombie" (Half-Open) causate da riavvii o ri-schedulazioni dei Pod server. In assenza di traffico continuo, il client potrebbe non rilevare tempestivamente la caduta del server, ricevendo errori `UNAVAILABLE` o `StreamRemoved` solo al tentativo successivo di invocazione RPC.

Per mitigare questi problemi e garantire la stabilità del canale, è stata implementata una configurazione di **Keepalive** aggressiva sia lato Client che lato Server:

- **Lato Client:** È stato configurato l'invio proattivo di frame `PING` HTTP/2 ogni 10 secondi (`grpc.keepalive_time_ms`) con un timeout di 5 secondi.
  Fondamentale è l'opzione `grpc.keepalive_permit_without_calls: 1`, che autorizza l'invio dei ping anche quando non ci sono RPC in corso, mantenendo il canale "caldo" anche durante i periodi di inattività dell'utente.
- **Lato Server:** Il server è stato configurato per accettare e rispondere a questi ping frequenti (`grpc.http2.min_ping_interval_without_data_ms: 5000`), rilassando le protezioni anti-DDoS predefinite di gRPC che altrimenti chiuderebbero la connessione considerandola "troppo chiacchierona" (too many pings).

```
options = [
    # Definisce politica di retry automatica per codici UNAVAILABLE
    ('grpc.service_config', json.dumps(service_config)),

    # KEEPALIVE OPTIONS
    # Invia PING ogni 10 secondi per mantenere vivo il tunnel TCP
    ('grpc.keepalive_time_ms', 10000),
    # Timeout del PING prima di chiudere la connessione
    ('grpc.keepalive_timeout_ms', 5000),
    # Cruciale: permette i PING anche senza RPC attive (idle state)
    ('grpc.keepalive_permit_without_calls', 1),
    # Permette infiniti ping senza dati (client-side)
    ('grpc.http2.max_pings_without_data', 0),
]

self.channel = grpc.insecure_channel(target, options=options)
```

Listing 3: Configurazione Keepalive gRPC (Client-side)

Nota su Scalabilità e Load Balancing (Sticky Connections): Infine, è doveroso menzionare una considerazione architetturale relativa allo scaling orizzontale massivo. Sebbene non implementato nello scope attuale (dove il bilanciamento L4 del Service Kubernetes è sufficiente), in uno scenario di produzione ad alto traffico l'uso di connessioni persistenti HTTP/2 renderebbe inefficace il load balancing standard di Kubernetes (fenomeno delle *Sticky Connections*). Una configurazione ideale per tale scenario prevedrebbe l'adozione del **Client-Side Load Balancing**: configurando il target gRPC con schema `dns:///` e la policy `round_robin`, il client risolverebbe direttamente gli IP di tutti i Pod disponibili, distribuendo le richieste RPC a livello applicativo anziché di connessione, garantendo un utilizzo uniforme delle risorse server.

3 Architettura del Sistema e Componenti

3.1 Diagramma Architetturale

Il sistema mantiene la struttura logica a microservizi delle precedenti iterazioni, ma viene arricchito da componenti infrastrutturali specifici per l'orchestrazione su Kubernetes, nonché dall'aggiunta di Prometheus come componente di monitoraggio. Il seguente diagramma illustra l'architettura complessiva del sistema, evidenziando i componenti principali e la suddivisione in layer logici, ma vengono omesse alcune interazioni per chiarezza (come le comunicazioni gRPC tra User Manager e Data Collector).

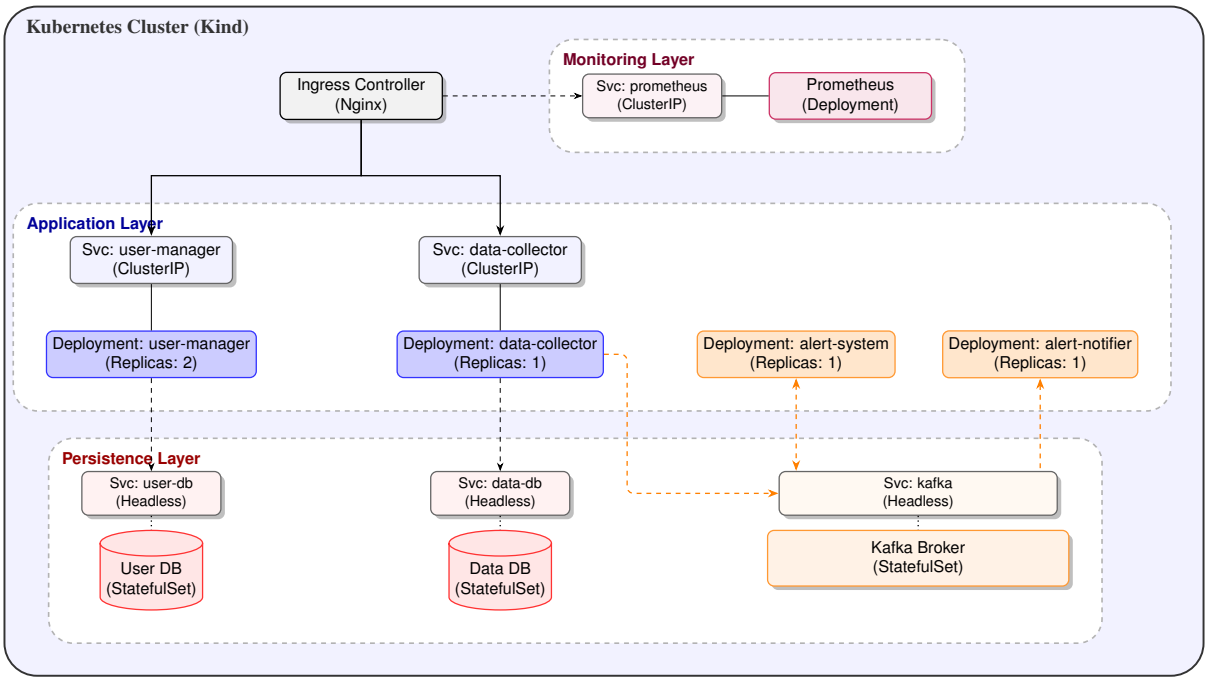

Figura 1: Architettura di Deployment su Kubernetes (Namespace: dsbd-ns)

3.2 Diagramma di interazione

Il seguente diagramma mostra, invece, le interazioni tra tutti i componenti del sistema, inclusi i flussi di monitoraggio di Prometheus e le chiamate verso servizi esterni come l'API OpenSky e Gmail SMTP.

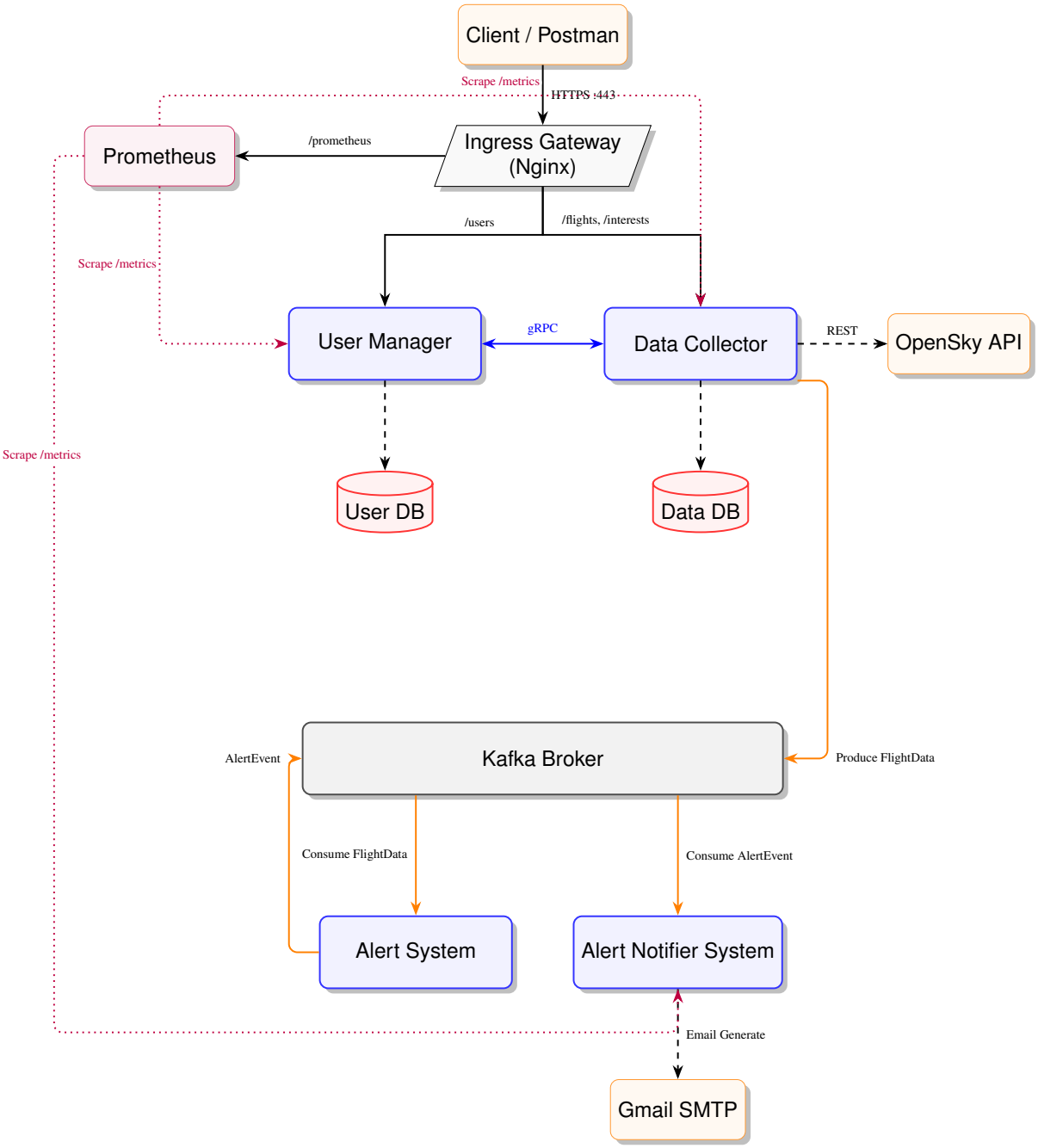

Figura 2: Diagramma Logico: Architettura e Flussi Dati

4 Deployment su Kubernetes

Con riferimento all'architettura illustrata in Figura 1, l'infrastruttura è stata definita seguendo il paradigma *Infrastructure as Code* (IaC) tramite manifesti YAML dichiarativi. Per simulare fedelmente un ambiente di produzione distribuito e soddisfare i requisiti di osservabilità sulla distribuzione dei carichi, il cluster Kind è stato inizializzato in una topologia **Multi-Node**, composta da:

- **1 Control Plane Node:** Responsabile della gestione del cluster e dello scheduling.
- **2 Worker Nodes:** Dedicati all'esecuzione dei workload applicativi. Questa configurazione permette di verificare, tramite le metriche di Prometheus (label `node`), l'effettiva distribuzione dei Pod su nodi fisici differenti.

4.1 Configurazione delle Risorse Kubernetes

La definizione delle risorse su Kubernetes è stata modellata sulle caratteristiche architetturali dei singoli microservizi. In particolare, sono stati adottati pattern di deployment specifici per rispecchiare la natura *stateless* o *stateful* dei componenti e i vincoli operativi associati.

4.1.1 Deployment e Scaling dei Servizi Stateless

Per il microservizio **User Manager**, caratterizzato da una natura puramente *stateless* (dato che delega lo stato al DB), è stato implementato un meccanismo di **Horizontal Scaling**, impostando il manifest di deployment con `replicas: 2`. Questo approccio consente di bilanciare il carico delle richieste HTTP in ingresso tra più istanze, migliorando la disponibilità e la capacità di gestione del traffico. Inoltre, come verrà discusso nella sezione relativa al layer logico di persistenza, avendo deciso di non replicare i database MySQL, le repliche di User Manager si connettono tutte alla stessa istanza primaria del database, evitando problemi di consistenza e continuando a garantire la corretta implementazione della politica **At-Most-Once** per la registrazione degli utenti.
Per i servizi event-driven **Alert System** e **Notifier**, la strategia di replicazione non dipende esclusivamente da Kubernetes, ma è intrinsecamente legata alla configurazione del broker tramite il pattern dei **Kafka Consumer Groups**. L'architettura delega interamente al broker la gestione della concorrenza e del coordinamento tra le istanze, permettendo di adottare due modelli operativi distinti senza modificare il codice applicativo (che definisce già i `group_id` specifici):

- **High Availability (Configurazione Attuale):** Allo stato attuale, il broker è configurato con una singola partizione per topic (`KAFKA_NUM_PARTITIONS: "1"`). In questo scenario, anche configurando il Deployment con `replicas: 2` (o superiore), il protocollo di Kafka impone che una sola istanza del gruppo ottenga l'assegnazione della partizione e processi i messaggi, mentre le altre rimangono in *idle*, pronte a subentrare istantaneamente solo in caso di failure del leader. Questa configurazione massimizza l'affidabilità del servizio garantendo il failover automatico, pur non parallelizzando il carico di lavoro (non c'è un Single Point of Failure, ma si sprecano risorse).
- **Scalabilità Orizzontale (Predisposizione):** Il sistema è progettato per essere scalabile orizzontalmente. Per incrementare la capacità di elaborazione, è sufficiente aumentare il numero di partizioni nel broker e, parallelamente, il numero di repliche dei Pod su Kubernetes. In tale configurazione,

Kafka distribuirebbe le partizioni tra le diverse istanze attive (Pattern *Competing Consumers*), permettendo l'elaborazione parallela dei messaggi e riducendo la latenza complessiva in presenza di elevati volumi di traffico.

4.1.2 Deployment e Scaling dei Servizi Stateful con Vincoli Attivi

Il **Data Collector** presenta un vincolo architetturale critico: pur scrivendo su DB relazionale, agisce come un componente *attivo* che mantiene uno stato volatile in memoria per la gestione dei job pianificati (tramite *APScheduler*). Ciò implica che una strategia di replicazione orizzontale sarebbe dannosa, in quanto causerebbe la duplicazione dello scheduler e, di conseguenza, l'esecuzione ridondante delle chiamate API (con esaurimento delle quote OpenSky) e l'aumento di eventuali conflitti di scrittura sul database (dato che più istanze scriverebbero contemporaneamente esattamente gli stessi dati). Il servizio è dunque configurato come **Singleton** (`replicas: 1`). La resilienza non è data dalla ridondanza, ma dal meccanismo di *Self-Healing* di Kubernetes che riavvia il Pod in caso di guasto, ripristinando l'unico scheduler autorizzato. Una possibile evoluzione futura potrebbe prevedere la separazione del componente di scheduling in un microservizio dedicato, consentendo così la scalabilità orizzontale del Data Collector puro; oppure l'utilizzo di un topic Kafka di coordinamento a **singola partizione** come meccanismo di *Leader Election*. In questo scenario, sarebbe possibile dispiegare N repliche del Data Collector appartenenti allo stesso *Consumer Group*: grazie al protocollo di Kafka, solo una replica alla volta (la leader) otterrebbe l'assegnazione della partizione e processerebbe i "trigger" di schedulazione (eventualmente prodotti da un componente esterno dedicato a questo scopo), mentre le altre rimarrebbero in *standby* pronte a subentrare istantaneamente in caso di guasto, eliminando il singolo punto di fallimento senza duplicare l'esecuzione dei job.

4.1.3 Gestione dei Workload Stateful

Per i componenti critici che richiedono persistenza stabile e identità di rete univoca, come **MySQL** e **Kafka**, la strategia di deployment si fonda sull'utilizzo di `StatefulSet`. A differenza dei Deployment standard, questa risorsa assegna a ciascun Pod un identificativo persistente e ordinato (es. `kafka-0`), supportato da servizi *Headless* (`ClusterIP: None`) che permettono la risoluzione DNS diretta delle singole istanze, fondamentale per la comunicazione interna del cluster (per un'analisi dettagliata, si faccia riferimento alla sezione 4.1.4). Un aspetto cruciale di questa configurazione è il binding stabile con lo storage: il sistema garantisce che ogni Pod, anche in caso di riavvio o ri-schedulazione su nodi diversi, si riconnetta invariabilmente al proprio `PersistentVolumeClaim` (PVC) dedicato, assicurando la durabilità e l'integrità dei dati. Un dettaglio implementativo critico per la stabilità del broker riguarda la configurazione del Service Headless con l'opzione `publishNotReadyAddresses: true`. Poiché le sonde di salute (*Startup/Readiness Probes*) sono state configurate per utilizzare i tool nativi di Kafka (es. `kafka-topics -list`) che richiedono risoluzione DNS, è necessario che il nome del Pod sia risolvibile anche quando lo stato non è ancora *Ready*. Senza questa direttiva, si creerebbe un deadlock in fase di avvio: la sonda fallirebbe perché non trova l'host, e il Pod non diventerebbe mai pronto perché la sonda fallisce. Infine, si è scelto deliberatamente di non adottare pattern di replicazione distribuita a livello applicativo, quali architetture Master-Slave per MySQL o cluster basati su Raft per Kafka. Tale decisione è motivata dalla necessità di evitare l'elevata complessità accidentale che questi meccanismi introdurrebbero: garantire la consistenza dei dati in un sistema distribuito (nel rispetto dei vincoli del

Teorema CAP), gestire la latenza di sincronizzazione e configurare meccanismi robusti di leader election e split-brain recovery comporterebbe, infatti, un overhead gestionale sproporzionato rispetto agli obiettivi del progetto. Pertanto, si è optato per delegare la disponibilità alla resilienza infrastrutturale nativa offerta da Kubernetes; attraverso l'uso di StatefulSet e volumi persistenti, il sistema assicura il riavvio automatico dei Pod vincolandoli ai rispettivi dati, garantendo così l'integrità dello storage e la Resource Isolation dai carichi stateless, senza la necessità di logiche di replica complesse.

4.1.4 Strategie di Networking e Service Exposure

La comunicazione inter-processo all'interno del cluster è governata da diverse tipologie di risorse `Service`, selezionate in base alla natura specifica del carico di lavoro e ai requisiti di indirizzamento:

- **ClusterIP (Standard Internal Communication):** Per i microservizi *stateless* (`user-manager`, `data-collector`) e per la dashboard di Prometheus, si è adottato il tipo `ClusterIP`. Questa risorsa fornisce un IP virtuale stabile (VIP) e un bilanciamento del carico interno (Layer 4) tra le repliche disponibili, astraendo la natura effimera dei Pod. L'accesso a questi servizi non avviene mai direttamente dall'esterno, ma è sempre mediato dall'Ingress Controller (pattern Reverse Proxy).
- **Headless Services (Stateful Identity):** Per i componenti *stateful* (`mysql`, `kafka`), è stato configurato un servizio di tipo `Headless` (impostando `ClusterIP: None`). A differenza del ClusterIP standard, questa configurazione non assegna un singolo IP virtuale al servizio, ma permette la risoluzione DNS diretta degli indirizzi IP dei singoli Pod (es. `kafka-0.kafka...`). Questo meccanismo è fondamentale per le architetture distribuite che necessitano di distinguere le istanze (es. per eleggere un Leader o gestire la replicazione dei dati) e per consentire ai client di connettersi a una specifica replica.
- **Architettura "Pure Worker" (Assenza di Service):** I componenti `alert-system` e `alert-notifier` agiscono esclusivamente come consumatori/produttori attivi verso Kafka o client verso server SMTP esterni. Non dovendo accettare connessioni in ingresso (né traffico HTTP né chiamate gRPC), per questi workload non è stata definita alcuna risorsa Service. Questa scelta progettuale aderisce al principio del *Least Privilege*, riducendo la superficie di attacco interna del cluster e risparmiando risorse di rete (IP virtuali e regole iptables) non necessarie.

4.2 Gestione Operativa, Sicurezza e Ottimizzazione delle Risorse

L'integrità e la sicurezza dell'infrastruttura sono garantite dall'adozione di primitive native di Kubernetes, configurate specificamente per bilanciare le esigenze di sicurezza con i vincoli prestazionali di un ambiente di sviluppo locale.

4.2.1 Gestione della Configurazione e Sicurezza

Per garantire la portabilità e la sicurezza del codice, è stata adottata una rigorosa separazione tra logica applicativa e configurazione. I parametri non sensibili, come endpoint dei servizi, porte e topic Kafka, sono stati disaccoppiati dal codice sorgente e iniettate nei Pod tramite oggetti **ConfigMap**. Parallelamente, la gestione delle informazioni critiche — quali password dei database MySQL, chiavi API OpenSky e credenziali SMTP — è affidata a oggetti **Secret**, montati come variabili d'ambiente nei container. In linea

con le best practices GitOps, i manifesti YAML contenenti tali segreti sono esclusi dal versionamento nel repository, prevenendo l'esposizione accidentale di credenziali.
L'accesso esterno al cluster è centralizzato e protetto mediante un **Ingress Controller** (basato su NGINX) configurato con *TLS Termination*. Questa risorsa funge da Reverse Proxy unico, esponendo i servizi tramite protocollo sicuro HTTPS e gestendo il routing del traffico in base ai path specifici (es. `/users`, `/flights`), mascherando così la complessità della topologia di rete interna. Per permettere la visualizzazione delle metriche di Prometheus, è stato configurato un path dedicato (`/prometheus`) sull'Ingress, garantendo l'accesso sicuro all'interfaccia di monitoraggio senza esporre direttamente il servizio all'esterno del cluster.

4.2.2 Affidabilità e Self-Healing

La resilienza dei servizi è monitorata attivamente tramite un sistema di sonde (*Probes*) configurate su tutti i componenti critici. Nello specifico, l'uso combinato di **Liveness Probe** e **Readiness Probe** permette a Kubernetes rispettivamente di riavviare i processi in stallo e di escludere temporaneamente dal bilanciamento del carico i Pod non pronti a ricevere traffico. Particolare attenzione è stata dedicata alla **Startup Probe**, essenziale per gestire le fasi di inizializzazione complesse (come le migrazioni dei database) inibendo i controlli di liveness fino al completamento dell'avvio.

4.2.3 Tuning Prestazionale in Ambiente Virtualizzato

In considerazione dell'ambiente di esecuzione locale basato su virtualizzazione (Kind su Docker/WSL), sono state implementate specifiche ottimizzazioni per mitigare i colli di bottiglia hardware tipici del *Cold Start*. Le soglie di tolleranza delle *Startup Probes* sono state configurate con valori molto ampi (es. `failureThreshold: 120-150`) per compensare i tempi di I/O su disco virtualizzato — stimati in 10-12 minuti in caso di primo avvio, nel caso peggiore — evitando così falsi positivi e cicli di riavvio (*CrashLoop*) durante l'inizializzazione. Si precisa che tali valori sono specifici per l'ambiente di sviluppo locale e andrebbero rivisti in un contesto di produzione, dove le performance hardware sarebbero significativamente superiori, e che sono stati impostati a valori pari circa al doppio del tempo misurato **Worst Case**. Inoltre, si è scelto di definire le risorse minime garantite (`requests`) disabilitando volutamente i limiti rigidi (`limits`) per CPU e memoria. Questa strategia previene la terminazione forzata dei Pod (OOMKilled o CPU throttling) durante i picchi di utilizzo all'avvio, garantendo la stabilità del sistema su hardware limitato, pur mantenendo la consapevolezza che, anche in questo caso, in un contesto di produzione (Staging/Prod) tali vincoli andrebbero riattivati e calibrati sui carichi effettivi.

5 Monitoraggio e Observability

Il sistema adotta un approccio di monitoraggio **White-box** completo, basato sulla strumentazione diretta del codice applicativo. Ogni microservizio critico integra la libreria client di Prometheus per esporre metriche applicative custom sull'endpoint `/metrics`. Tali endpoint sono configurati per essere accessibili esclusivamente all'interno del cluster (protetti da esposizione pubblica) e vengono interrogati periodicamente (scrape) dall'istanza di Prometheus.

Un aspetto fondamentale dell'implementazione è l'arricchimento delle metriche con label contestuali, quali `service` (label indicante il microservizio che ha generato la metrica) e `node`. Quest'ultima, in particolare, permette di correlare le prestazioni dei microservizi non solo al Pod effimero (tramite un'ulteriore label gestita nativamente da Prometheus), ma direttamente al **nodo fisico** del cluster Kubernetes su cui il processo è in esecuzione, facilitando l'analisi della distribuzione del carico sui worker.

5.1 Infrastruttura di Monitoraggio (Deployment Prometheus)

L'implementazione del monitoraggio non si limita al deployment del server Prometheus, ma richiede una configurazione articolata delle risorse Kubernetes per abilitare il **Service Discovery dinamico**:

- **RBAC e Permessi (ClusterRole):** Poiché Prometheus deve interrogare le API di Kubernetes per scoprire quali Pod monitorare, è stato necessario definire un `ClusterRole` specifico con permessi di lettura (`get, list, watch`) sulle risorse `pods`, `services` e `endpoints`. Questo ruolo è associato al Pod di Prometheus tramite un `ServiceAccount` dedicato, garantendo l'accesso sicuro al Control Plane.
- **Configurazione di Scraping (Service Discovery):** La logica di monitoraggio è definita all'interno del manifest `prometheus-config.yaml` e iniettata nel container tramite una **ConfigMap** che genera il file `prometheus.yml`. Tale configurazione definisce un job `kubernetes-pods` basato su `kubernetes_sd_configs`: invece di elencare staticamente gli IP dei target, il sistema utilizza regole di **Relabeling** per filtrare dinamicamente solo i Pod che possiedono la specifica annotation `prometheus.io/scrape: "true"`.
- **Persistenza (PVC):** Per garantire che lo storico delle metriche sopravviva ai riavvii o agli aggiornamenti del server di monitoraggio, il Deployment di Prometheus è vincolato a un `PersistentVolumeClaim` da 1Gi. A livello di strategia di deployment, è stato impostato `strategy: Recreate`: questo assicura che il vecchio Pod venga terminato (rilasciando il lock sul volume) prima che venga avviato il nuovo, prevenendo conflitti di montaggio tipici dei volumi ReadWriteOnce (RWO).

5.2 Dettaglio delle Metriche Applicative

Di seguito viene riportato il dettaglio delle metriche implementate per ogni servizio:

- **User Manager Service:**
  - `http_requests_total` (*Counter*): Metrica standard per il monitoraggio del traffico HTTP ricevuto, arricchita dalle label `method`, `endpoint` e `status` code.
  - `active_users_total` (*Gauge*): Rappresenta il numero totale di utenti registrati. Per garantire la consistenza del dato anche in uno scenario replicato (Deployment con `replicas: 2`), il valore non è mantenuto in memoria locale ma viene aggiornato periodicamente (ogni 10s) tramite un thread dedicato che interroga il database condiviso.
  - `cache_cleanup_duration_seconds` (*Gauge*): Monitora le performance del meccanismo di pulizia della cache di idempotenza.
  - `cache_cleaned_entries_total` (*Counter*): Traccia il numero di elementi scaduti e rimossi dalla cache di idempotenza.
- **Data Collector Service:**
  - `opensky_api_calls_total` (*Counter*): Metrica critica sia per il monitoraggio del consumo delle quote API esterne (OpenSky Network), sia per verificare la percentuale di richieste andate a buon fine rispetto al totale. Include la label `status` (*attempt, success, failure*) per rilevare tempestivamente errori, siano essi di rete o rate-limiting.
  - `flight_data_processing_seconds` (*Gauge*): Misura la latenza complessiva del ciclo di raccolta dati periodico, includendo le fasi di fetch API, salvataggio dei dati su MySQL e invio dei messaggi su Kafka.
  - `http_requests_total` (*Counter*): Analoga alla metrica del User Manager per il monitoraggio del traffico HTTP in ingresso (anch'essa presenta le stesse label di arricchimento).
- **Alert Notifier System:**
  - `emails_sent_total` (*Counter*): Conteggio cumulativo delle email inviate agli utenti, con label `status` per distinguere invii riusciti da fallimenti.
  - `last_email_sent_duration_seconds` (*Gauge*): Metrica di latenza che monitora il tempo impiegato per l'handshake e l'invio effettivo tramite il server SMTP esterno (Gmail), utile per isolare problemi di rete verso terze parti.