Relazione Progetto CCT

Architettura Microservizi Event-Driven per Monitoraggio IoT su Kubernetes

Studente: Enea Manzi **Corso:** Cloud computing technologies
Anno Accademico: 2025/2026 **Repository:** GitHub Link

1. Scenario e Obiettivi Architetturali

Il progetto propone un'architettura Event-Driven su Kubernetes per il monitoraggio real-time di una rete di sensori IoT industriali. L'implementazione è progettata per soddisfare i requisiti combinati delle tracce d'esame relative a **Apache Kafka (Project 1)**, garantendo *fault tolerance*, *alta disponibilità* e *sicurezza avanzata* (TLS/SASL), e **Kong API Gateway (Project 3)**, centralizzando *autenticazione* e *routing all'edge*.

Lo scenario di riferimento simula un ambiente industriale eterogeneo che genera flussi continui di dati. Le tipologie di eventi gestiti — **Telemetria** (dati ambientali), **Boot** (avvio dispositivo), **Firmware Updates** e **Alerts** — sono state classificate in due macro-categorie logiche per ottimizzare la *Qualità del Servizio (QoS)*:

1. **Flussi Operativi ad Alta Frequenza:** Dati massivi che richiedono efficienza e bassa latenza (Telemetria, Boot, Update).
2. **Eventi Critici:** Dati sporadici (Alerts) che impongono la massima garanzia di consegna e durabilità.

L'infrastruttura supera i limiti delle architetture monolitiche garantendo **Scalabilità, Resilienza e Sicurezza** tramite un design a microservizi che sfrutta **Apache Kafka** per il disaccoppiamento asincrono e **Kong** per la gestione del traffico, orchestrando l'intero ciclo di vita dei componenti su **Kubernetes**.

2. Architettura del Sistema

La soluzione adotta un pattern **Event-Driven** per disaccoppiare l'ingestione dati dal processamento tramite messaggistica asincrona. L'infrastruttura è segmentata in namespace logici per garantire la *Separation of Concerns*: `kong` (Ingress/Gateway), `kafka` (Core logic, Broker, Storage) e `metrics` (Analytics).

2.1 Flusso dei Dati (Data Pipeline)

La pipeline *(qui l'immagine)* è progettata per gestire picchi di carico e garantire resilienza tramite meccanismi di *Backpressure*:

1. **Ingress & Security (Kong):** Il traffico entra esclusivamente attraverso **Kong**, che agisce da Reverse Proxy intelligente validando l'identità del sensore (API Key) prima di inoltrare le richieste ai servizi backend.

2. **Ingestion (Producer):** Un microservizio *stateless* arricchisce i payload HTTP con metadati operativi (UUID, timestamp) e li instrada verso Kafka. Questo approccio di disaccoppiamento evita scritture dirette sul database, prevenendo colli di bottiglia.
3. **Streaming & Routing (Kafka):** Il broker garantisce la durabilità su disco e applica un routing intelligente: i dati operativi confluiscono su `sensor-telemetry` (ottimizzato con compressione LZ4), mentre le criticità su `sensor-alerts`, permettendo l'applicazione di policy di retention differenziate.
4. **Processing & Storage:** Il Consumer sottoscrive i topic, esegue la normalizzazione dei dati e li storicizza su **MongoDB** all'interno di collezioni *Time Series* ottimizzate.
5. **Analytics:** Il *Metrics Service* calcola aggregazioni on-demand direttamente sul database (es. medie temperature per zona), esponendo i risultati via API REST.

3. Apache Kafka: il Core Event Streaming

L'implementazione su Kubernetes è stata realizzata tramite **Strimzi**, che adotta il pattern *Operator*. Questo permette di gestire il cluster Kafka come una risorsa nativa Kubernetes (`Kind: Kafka`), automatizzando compiti complessi come il rolling update dei nodi, la gestione dei certificati TLS e la configurazione dei listener.

3.1 Orchestrazione KRaft (Senza ZooKeeper)

L'architettura adotta l'innovativa modalità **KRaft** (Kafka Raft Metadata), configurata esplicitamente tramite `KafkaNodePool` distinti per ruoli di *Controller* e *Broker*. Questa scelta sostituisce la dipendenza da **ZooKeeper**, internalizzando la gestione dei metadati.

- **Vantaggio Architetturale:** L'eliminazione del coordinatore esterno riduce drasticamente il footprint di risorse (CPU/RAM) e la superficie di attacco, semplificando la gestione operativa in favore di un'unica piattaforma unificata.

3.2 Strategia dei Topic e Trade-off

Topic `sensor-telemetry` (Alta Efficienza): Destinato ai flussi massivi (Telemetria, Boot, Update), configurato con **3 partizioni** per massimizzare il **parallelismo** di lettura (fino a 3 consumer concorrenti).

- **Strategia di Compressione Ibrida (LZ4):** È stato scelto **LZ4** per il *Transport Layer* grazie al suo bassissimo overhead sulla CPU e alta velocità di decompressione. Questo, per i flussi IoT in tempo reale, riduce l'occupazione di banda (fino al 60% per payload JSON ripetitivi) e previene colli di bottiglia sul Producer, integrandosi perfettamente con lo *Storage Layer* (MongoDB) che applica successivamente la compressione **Zstd** per l'efficienza dello storage a lungo termine.
- **Retention:** Limitata a 7 giorni per mantenere disponibili i dati operativi "caldi" (replay/analisi immediata), delegando lo storico profondo a MongoDB.

Topic `sensor-alerts` (Alta Affidabilità): Destinato esclusivamente agli allarmi.

- **Durabilità (Zero Data Loss):** Configurato con `min.insync.replicas: 2` e `replicas: 2`. Questo garantisce che **nessun allarme venga perso** anche in caso di crash improvviso di un broker, ma richiede che entrambe le repliche siano online per accettare scritture (trade-off: disponibilità ridotta durante manutenzioni).
- **Retention:** Estesa a 30 giorni per permettere analisi post-incidente e audit.

4. Kong API Gateway: Edge Computing & Security

L'esposizione diretta dei microservizi è considerata un anti-pattern in ambienti distribuiti. Si è scelto di utilizzare **Kong** come Ingress Controller per implementare il pattern *Gateway Offloading*: le responsabilità trasversali (autenticazione, rate limiting, routing) sono spostate dal codice applicativo all'infrastruttura.

4.1 Sicurezza Dichiarativa "As Code"

La sicurezza è definita tramite risorse Kubernetes (CRD - Custom Resource Definitions).

Autenticazione (Key-Auth):
Tramite il plugin `KongPlugin` di tipo `key-auth`, Kong intercetta ogni richiesta in ingresso. Verifica la presenza dell'header `apikey` e lo confronta con i *Secret* Kubernetes (`iot-devices-apikey`). Solo se la chiave è valida, la richiesta viene inoltrata al backend. Questo approccio permette di revocare l'accesso a dispositivi compromessi semplicemente aggiornando il Secret, senza dover ridistribuire o riavviare i microservizi.

Protezione (Rate Limiting):
L'architettura supporta l'applicazione dinamica di policy di Rate Limiting per mitigare attacchi Denial of Service (DoS), proteggendo i servizi backend da picchi anomali di traffico.

4.2 Routing e Load Balancing

Kong gestisce il routing basato su Host (es. `producer.nip.io`) tramite **Ingress** e instrada il traffico verso i *Service* Kubernetes sottostanti. Il bilanciamento del carico tra le varie repliche dei pod applicativi avviene automaticamente sfruttando il meccanismo nativo dei Service, garantendo una distribuzione uniforme delle richieste.

5. MongoDB: Persistenza e Ottimizzazione Dati

Per lo storage è stato selezionato **MongoDB**, configurato per gestire carichi IoT *write-heavy* e letture analitiche tramite le **Time Series Collections** native.

5.1 Vantaggi delle Time Series

L'utilizzo di collezioni ottimizzate rispetto ai documenti standard garantisce:

- **Compressione Zstd e Storage:** I dati sono organizzati fisicamente in "bucket" colonnari compressi per intervallo temporale. Questa struttura permette l'applicazione dell'algoritmo **Zstd**, che riduce drasticamente l'occupazione su disco rispetto al JSON tradizionale.
- **Efficienza I/O:** La struttura a bucket riduce i blocchi letti durante le query su range temporali (es. *"media ultima ora"*), massimizzando il throughput e l'efficienza.
- **Performance Analitiche:** Le aggregazioni del *Metrics Service* beneficiano di indici *clustered* automatici sul campo temporale, garantendo risposte rapide anche su storici profondi.

5.2 Configurazione Cloud Native

In linea con le best practice, la configurazione è disaccoppiata:

- **Parametri:** Endpoint, porte e nomi dei database sono iniettati via `ConfigMap`.
- **Sicurezza:** Le credenziali sensibili (utente/password di Mongo, credenziali SASL Kafka) sono gestite esclusivamente tramite `Secrets` e montate come variabili d'ambiente, senza esporre password in chiaro.

6. Analisi delle Proprietà Non Funzionali (NFP)

L'architettura è stata validata rispetto a tre pilastri fondamentali per sistemi cloud-native mission-critical, verificati tramite scenari di test specifici.

6.1 Security & Secrets Management (Defense in Depth)

La sicurezza è stata implementata seguendo il principio della *Defense in Depth*, proteggendo ogni livello dello stack.

6.1.1 Data in Transit & Auth

La comunicazione interna tra i microservizi (Producer/Consumer) e il cluster Kafka è protetta da **TLS** (porta 9093) per prevenire *Man-in-the-Middle*. L'accesso al broker non è anonimo ma autenticato via **SASL/SCRAM-SHA-512**, con utenze dedicate (`producer-user`, `consumer-user`) per ogni componente (`KafkaUser`).

6.1.2 Edge Protection (Kong)

Kong agisce da *Policy Enforcement Point* centralizzato, evitando di disperdere logica di auth nel codice applicativo. L'accesso richiede un header `apikey` valido (gestito via Secret). È stata verificata la mitigazione di attacchi DoS/Flood tramite policy (`KongPlugin`) di **Rate Limiting** (5 req/sec): il traffico in eccesso viene respinto all'edge (`429 Too Many Requests`) senza saturare il backend con traffico malevolo/anomalo.

6.1.3 Gestione dei Segreti

Nessuna credenziale è hardcodata. Le password di MongoDB e Kafka sono iniettate nei Pod esclusivamente tramite **Kubernetes Secrets**, mantenendo la separazione dalla configurazione non sensibile gestita tramite **ConfigMaps**.

6.2 Resilience, Fault Tolerance & High Availability

Il sistema è progettato per sopravvivere a guasti parziali senza perdita di dati o interruzione del servizio.

6.2.1 Fault Tolerance (Disaccoppiamento e Buffering)

La natura asincrona di Kafka garantisce che un crash del *Consumer* non impatti il *Producer*. Durante un disservizio del worker, i messaggi si accumulano nei topic Kafka (che agiscono da buffer persistente) e vengono elaborati (*drained*) non appena il Consumer torna online, garantendo **zero perdita di dati (Zero Data Loss)**.

6.2.2 Self-Healing & High Availability (Kubernetes)

I `Deployment` Kubernetes assicurano che il numero desiderato di repliche sia sempre attivo. In caso di crash del processo Python, il **Kubelet** rileva l'errore e riavvia automaticamente il container, garantendo continuità di servizio e alta disponibilità.

6.3 Scalabilità & Load Balancing

La validazione delle capacità di scaling è stata condotta in due fasi distinte: una verifica manuale per confermare i meccanismi di distribuzione del carico e una verifica automatica per testare l'elasticità del sistema.

6.3.1 Verifica dei Meccanismi (Scaling Manuale)

Incremento manuale delle repliche dei microservizi per validare il comportamento dell'architettura.

Ingress Load Balancing (HTTP Layer): Scalando il *Producer* a 2 repliche e iniettando un carico di richieste rapide (Burst), l'analisi dei log ha confermato la corretta distribuzione del traffico tra i pod (Round-Robin) implicando quindi il corretto bilanciamento attuato da **Kong Ingress** e dal Service Kubernetes

Consumer Parallelism (Stream Layer): Scalando il *Consumer* a 3 repliche (corrispondenti alle 3 partizioni del topic `sensor-telemetry`), è stato verificato il processamento parallelo dei messaggi. Questo dimostra che il protocollo di *Consumer Group* di Kafka assegna correttamente le partizioni esclusive ai nuovi worker, massimizzando il throughput di lettura.

6.3.2 Elasticità Automatica (HPA)

È stato attivato l'**Horizontal Pod Autoscaler (HPA)** configurato su una soglia di CPU del 50%.

- **Scale Out:** Durante uno stress test prolungato, il sistema ha rilevato la saturazione della CPU e ha avviato automaticamente nuove repliche (fino a un massimo di 4) per assorbire il carico.

- **Scale Down:** Terminato il picco, l'HPA ha ridotto il numero di Pod rilasciando le risorse eccedenti, riportando il cluster allo stato operativo standard e garantendo efficienza dei costi.

7. Sfide Tecniche e Soluzioni

7.1 Networking e Reachability degli Ingress

Problema: L'uso di **Docker Desktop** come driver Kubernetes creava un isolamento di rete interno, bloccando l'accesso diretto ai servizi esposti tramite Ingress (endpoint `producer` e `metrics`). Il "tunnel" verso l'IP del cluster risultava inaccessibile.

Soluzione: Dopo un tentativo fallito di usare `nip.io` per la mappatura dinamica dei nomi host, la risoluzione definitiva è stata la migrazione a un **driver nativo**. L'utilizzo di Minikube con un driver nativo ha permesso l'esposizione corretta dell'IP del cluster, ripristinando la connettività esterna necessaria per il testing.

7.2 Garanzia di Consegna (At-Least-Once) e Idempotenza

Problema: La semantica di consegna **At-Least-Once** di Kafka genera il rischio di duplicazione: se il Consumer crasha dopo la scrittura su MongoDB ma prima del *commit* dell'offset, Kafka ritrasmette il messaggio. Questo può causare l'inserimento multiplo dello stesso dato di telemetria nel database.

Soluzione: È stato implementato l'**Idempotency Token** a livello di Producer, arricchendo ogni payload con un **UUID univoco** (`event_id`). Attualmente, l'**implementazione accetta la duplicazione** (tramite `insert_one`) per **massimizzare il *throughput*** in uno scenario IoT massivo**.** Tuttavia, l'architettura è predisposta per l'idempotenza stretta: usando `l'event_id` come chiave univoca in MongoDB, il database scarterebbe automaticamente le scritture ridondanti, garantendo la consistenza dei dati a discapito di una leggera riduzione della performance di scrittura.

7.3. Migrazione MongoDB a StatefulSet per Stabilità e Persistenza

Problema: Gestire MongoDB (un'applicazione stateful) con un **Deployment** stateless ha causato instabilità. L'identità **instabile e casuale** dei Pod nei Deployment non permetteva al Pod sostitutivo, in caso di *failure*, di reclamare la sua **Persistent Volume Claim (PVC)** precedentemente associata. Dato che Kubernetes permette un solo *mount* della PVC, il Pod rimaneva bloccato nello stato `Pending`, interrompendo la continuità del servizio.

Soluzione: La migrazione a un **StatefulSet** ha risolto il problema. Questo costrutto garantisce un'**identità Pod stabile e prevedibile** (es. `mongo-mongodb-0`) univocamente associata a una specifica PVC. In caso di riavvio o fallimento, il Pod mantiene la stessa identità, potendo così **riacquisire il disco immediatamente e senza conflitti**. Questo ha richiesto solo un aggiornamento del campo `MONGO_HOST` nella `ConfigMap` per riflettere il nuovo servizio di rete interno (`mongo-mongodb-headless`).

8. Conclusioni e Raggiungimento Obiettivi

Il progetto ha soddisfatto i requisiti architetturali richiesti, validando l'efficacia del modello a microservizi su Kubernetes:

1. **Kafka (Project 1):** È stata garantita la **Fault Tolerance** e l'**High Availability** (validate tramite crash test e buffer persistente) e implementata la **Sicurezza Avanzata** (TLS/SASL). L'adozione di **KRaft** ha inoltre eliminato la complessità di ZooKeeper, rendendo il cluster più leggero e gestibile.
2. **API Gateway (Project 3):** L'uso di Kong ha evidenziato i vantaggi del pattern *Gateway Offloading* rispetto all'esposizione diretta dei servizi: la centralizzazione di **Load Balancing**, **Autenticazione** e **Rate Limiting** all'edge ha ridotto la complessità dei microservizi backend.