# Sistemi e Architetture per Big Data - A.A. 2024/25

## Report Progetto 2: Analisi real-time di difetti nella produzione L-PBF con Apache Flink

Simone Niro
Matricola: 0350119
Università di Roma "Tor Vergata"

Lorenzo Brunori
Matricola: 0353481
Università di Roma "Tor Vergata"

Luigi Ciuffreda
Matricola: 0351898
Università di Roma "Tor Vergata"

***Abstract*—Questo report descrive un'architettura distribuita per l'analisi in tempo reale di difetti nella produzione manifatturiera basata su tecnologia L-PBF. Il sistema, basato su Apache Flink, processa in streaming immagini di tomografia ottica per rispondere a specifiche query, simulando il monitoraggio degli oggetti durante la loro creazione. Vengono discusse le principali scelte progettuali, le tecnologie adottate e le prestazioni ottenute.**

***Index Terms*—Stream processing, Apache Kafka, Apache Flink, Prometheus, Grafana, Kafka Streams, Docker Compose, Big Data, DEBS2025, L-PBF, Optical Tomography Image**

## I. Introduzione

Lo scopo del progetto è sviluppare, usando Apache Flink, una pipeline di analisi streaming per il monitoraggio in tempo reale dei difetti nella produzione manifatturiera tramite tecnologia L-PBF (*Laser Powder Bed Fusion*). In questo processo, che prevede la fusione di polveri metalliche strato su strato mediante un laser per creare componenti tridimensionali, impurità nei materiali o errori di calibrazione possono generare difetti che, se individuati troppo tardi, comportano sprechi di tempo e materiale. L'analisi in tempo reale dello stream di OTIs (*Optical Tomography Images*) prodotte durante la stampa consente di identificare le aree a rischio difetto e interrompere anticipatamente il processo, risparmiando così tempo e risorse.

## II. Dataset

Il dataset utilizzato simula il flusso di dati generato durante un processo reale di produzione L-PBF, fornendo uno stream di OTIs ricevute un layer alla volta, ciascuno suddiviso in blocchi detti *tiles*. Ogni immagine fornisce, per ciascun punto $P = (x, y)$, la temperatura $T(P)$ a quel punto, misurata layer per layer.

Ogni elemento dello stream, che rappresenta l'unità minima di processamento per la pipeline, corrisponde a un singolo tile e contiene:

- `seq_id`: numero di sequenza univoco per l'elemento in input.
- `print_id`: identificatore dell'oggetto in stampa.
- `tile_id`: identificatore del tile all'interno del layer.
- `layer`: indice del layer (i.e. la coordinata $z$).
- `tiff`: dati binari dell'immagine del tile in formato TIFF (temperatura ad alta risoluzione, codificata su 16 bit).

Il dataset è fornito dal `LOCAL-CHALLENGER`, un container Docker che espone le seguenti REST API:

- `/create`: crea un nuovo benchmark.
- `/start`: avvia il benchmark.
- `/next_batch`: restituisce il prossimo batch di dati.
- `/result`: riceve i risultati prodotti dal sistema.
- `/end`: termina il benchmark.

## III. Query

La pipeline di processamento implementata coinvolge le seguenti query:

1) ***Saturation analysis* (Q1)**. Per ogni tile, identificare i punti vuoti (temperatura inferiore a 5000) e i punti saturati (temperatura superiore a 65000). Entrambi sono da escludere dalle query successive, ma i punti saturati potrebbero indicare difetti e devono anche essere riportati.

2) ***Windowing & Outlier analysis* (Q2)**. Per ogni tile, mantenere una *sliding window* degli ultimi 3 layer, abilitando un'analisi dell'evoluzione della temperatura tra layer consecutivi. Per ogni punto $P$ sui tile del layer più recente della finestra, calcolare la deviazione di temperatura locale, definita come la differenza assoluta tra:
   - la temperatura media dei vicini prossimi di $P$ (i.e. tutti i punti con una distanza Manhattan $0 \leq d \leq 2$ da $P$), considerando i 3 layer;
   - la temperatura media dei vicini lontani di $P$ (i.e. tutti i punti con una distanza Manhattan $3 \leq d \leq 4$ da $P$), considerando i 3 layer.

   Dopodiché, classificare i punti come outlier se la loro deviazione di temperatura locale è superiore a 6000.

3) ***Outlier clustering* (Q3)**. Raggruppare in cluster, usando l'algoritmo DBSCAN e la distanza euclidea come metrica, i punti outlier identificati nella query precedente.

## IV. Architettura

Il sistema è stato sviluppato su un nodo standalone, utilizzando Docker per ogni componente ed orchestrando il tutto tramite Docker Compose.

### A. Schema Architetturale

La Figura 1 mostra lo schema architetturale del sistema, evidenziando le interazioni tra i vari servizi e il percorso seguito dai dati. Nel dettaglio, i componenti principali che lo compongono sono:

- **Challenger**: simula la produzione di immagini tomografiche e le rende disponibili tramite una REST API.
- **Producer**: legge i dati dal Challenger e li pubblica su Apache Kafka.
- **Apache Kafka**: gestisce la comunicazione tra i vari componenti tramite messaggi.
- **Zookeeper**: gestisce lo stato del cluster Kafka e coordina i nodi.
- **Apache Flink**: elabora i dati in tempo reale, eseguendo le query richieste e pubblicando i risultati su Kafka.
- **Prometheus**: raccoglie metriche dai componenti Flink per il monitoraggio delle prestazioni.
- **Grafana**: visualizza le metriche raccolte da Prometheus tramite dashboard personalizzate.
- **Consumer**: legge da Kafka i risultati elaborati da Flink, li salva su file in locale e li invia al Challenger per la validazione.
- **Locale**: permette di eseguire l'intero flusso in modo automatizzato e scegliere l'engine di elaborazione da utilizzare.

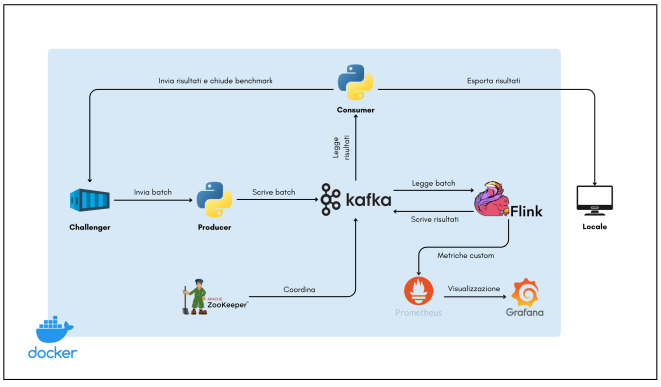


**Fig. 1:** Architettura del sistema

### B. Clusters

Tra i vari container Docker che compongono il sistema, quelli relativi a Flink sono quelli che meritano una vista più dettagliata, in quanto costituiscono un cluster di elaborazione distribuito. Come mostrato in Figura 2, il cluster Flink è composto da un *JobManager*, responsabile della coordinazione e della gestione dei job, due *TaskManager*, che eseguono fisicamente le operazioni, e un *Flink Client*, che si occupa di inviare i job al JobManager. Ogni componente è eseguito all'interno di un container dedicato.


**Fig. 2:** Cluster di Flink

Per quanto riguarda i restanti componenti, essi non presentano una struttura particolare da evidenziare.

### C. Deployment e Modularità

Tra i principali punti di forza del sistema ci sono la grande facilità di deployment e la modularità dell'architettura. Essendo tutti i componenti containerizzati tramite Docker e orchestrati con Docker Compose, è possibile avviare l'intero sistema con pochi comandi, senza necessità di configurazioni manuali o installazioni complesse. Questo approccio garantisce portabilità, riproducibilità e semplicità di setup anche su macchine diverse da quella di sviluppo.

L'applicazione è progettata in modo modulare: ogni componente è isolato in un proprio container e comunica con gli altri esclusivamente tramite interfacce ben definite. Questa separazione consente di sostituire, aggiornare o estendere singoli moduli senza impattare sul resto del sistema. Ad esempio, è possibile utilizzare Flink o Kafka Streams semplicemente scegliendo quale dei due engine attivare con il `docker-compose up`, senza dover intervenire sul codice degli altri componenti.

Un ulteriore elemento distintivo è la presenza di un `main.py` che consente di eseguire automaticamente l'intera analisi attraverso il passaggio di pochi parametri a riga di comando. Questo ha facilitato la ripetizione di esperimenti in modo controllato e automatizzato, ed è stato mantenuto per semplificare ulteriormente l'utilizzo dell'applicazione subito dopo il deploy.

### D. Apache Kafka & Apache Zookeeper

Nella configurazione di Kafka, l'utilizzo della variabile d'ambiente `KAFKA_ADVERTISED_LISTENERS` garantisce comunicazione interna tra Kafka e gli altri container. La variabile `KAFKA_ZOOKEEPER_CONNECT`, invece, permette a Kafka di stabilire una connessione con Zookeeper, che ne gestisce il coordinamento. È presente una dipendenza tra i due container, in quanto Kafka non può essere avviato senza che Zookeeper sia già in esecuzione.

### E. Producer & Consumer

Sia il *Producer* che il *Consumer* sono implementati come container dediti all'esecuzione di script Python. Il primo è responsabile della lettura dei dati generati dal Challenger e della loro pubblicazione su Apache Kafka. Il secondo, invece, si occupa di leggere i risultati elaborati da Flink dai topic Kafka di output, di salvarli in locale e di inviarli al Challenger per la validazione finale.

### F. Apache Flink

Il JobManager, che funge da nodo master, specifica il proprio indirizzo per le chiamate RPC (`jobmanager.rpc.address`) utilizzate per comunicare con gli altri nodi del cluster. Inoltre, definisce una variabile d'ambiente che abilita l'esportazione delle metriche tramite Prometheus, esponendo la porta `9249` per consentire la raccolta delle metriche.

I due TaskManager seguono la configurazione del master, specificando anch'essi l'indirizzo RPC del JobManager. In ciascun TaskManager, inoltre, è presente la variabile `taskmanager.numberOfTaskSlots`, che definisce il numero di slot disponibili per i job, che in questa configurazione è pari a 2 per ciascun TaskManager.

Il Client è configurato per comunicare con il JobManager tramite il solito indirizzo RPC. Monta un volume condiviso contenente il file JAR del job da inviare al master. Al suo avvio, tramite il comando specificato nella configurazione Docker, invia automaticamente il job al JobManager.

Questo cluster presenta una struttura gerarchica di dipendenze tra i container: i TaskManager dipendono dal JobManager, mentre il Client dipende sia dal JobManager che dai TaskManager, in quanto sono necessari per eseguire il job.

### G. Kafka Streams

Il sistema permette di utilizzare, come alternativa a Flink, **Kafka Streams** per l'esecuzione delle prime due query. Non è stato incluso nel diagramma sia per semplicità, sia perché si tratta di una componente opzionale.

La sua implementazione è stata dettata dalla volontà di confrontarne le prestazioni con Flink, come richiesto dalla parte opzionale del progetto. Infatti, il suo ruolo è praticamente identico e, come per il client di Flink, monta un volume condiviso contenente il file JAR del job da eseguire.

## V. Data Ingestion and Preprocessing

### A. Data Ingestion

Come anticipato, la fase di ingestione dei dati è gestita dal Producer, che si occupa di:

- Leggere i dati grezzi dal Challenger, comunicando tramite richieste HTTP all'API REST esposta.
- Pubblicare i dati su Kafka. I dati ricevuti sono, infatti, già serializzati in formato binario e vengono inviati, senza ulteriori trasformazioni, direttamente a Kafka al topic `gc-batches`.

### B. Data Preprocessing

La fase di preprocessamento nella pipeline non è gestita da una componente separata, ma è integrata direttamente nel job di Flink (o di Kafka Streams). Questa fase ha il compito di trasformare i dati letti da Kafka in strutture dati pronte per le analisi successive. Infatti, dopo che il Producer pubblica un messaggio su `gc-batches`, questo viene consumato dal motore di stream processing e sottoposto a una serie di trasformazioni.

Il primo passo consiste nella deserializzazione e nell'estrazione dei campi rilevanti tramite la classe `TileLayerExtractor`, che si occupa di:

- Estrarre i parametri identificativi del tile (`batchId`, `printId`, `tileId`, `layerId`) dal record Kafka.
- Decodificare l'immagine TIFF contenuta nel messaggio e convertirla in una matrice di interi rappresentante la temperatura.
- Creare l'oggetto `TileLayerData`, ossia una rappresentazione strutturata di un singolo tile, includendo attributi fondamentali per le analisi successive (e.g. timestamp utilizzati per il calcolo di metriche di performance personalizzate).

Durante questa fase, vengono effettuati controlli di validità sui campi obbligatori: se un campo essenziale è mancante o non valido, viene sollevata un'eccezione e il dato viene scartato, garantendo la qualità dei dati in ingresso.

Questo preprocessamento è fondamentale per assicurare che ogni tile sia rappresentato in modo coerente e completo, rendendolo pronto per essere analizzato dalle query successive.

## VI. Data Processing

### A. Apache Flink

Il file `StreamingJob.java` rappresenta il punto d'ingresso dell'applicazione Flink. Questo job gestisce l'intera pipeline di elaborazione dei dati: legge i messaggi da Kafka, li trasforma in oggetti specifici, applica le query di analisi e pubblica i risultati su Kafka. Ogni query agisce sul flusso di dati modificando gli oggetti `TileLayerData` e passandoli come input alla fase successiva.

Prima di avviare il job, l'applicazione attende che il broker Kafka e il topic di input (`gc-batches`) siano disponibili. Una volta verificata la disponibilità, viene creato un

KafkaSource per leggere i messaggi serializzati in formato MessagePack. Questi messaggi vengono deserializzati in mappe chiave-valore (Map<String, Object>) grazie a uno schema di deserializzazione personalizzato (MsgPackDeserializationSchema). Il risultato è un DataStream, l'astrazione principale di Flink per rappresentare i flussi di dati.

Attraverso un operatore map, i dati vengono trasformati in oggetti TileLayerData che, come già discusso, rappresentano i dati di un singolo tile all'interno di un layer. Questi oggetti costituiscono l'input per le query successive:

- *Saturazione*. La prima query analizza un tile alla volta per identificare i punti saturati. Il risultato dell'analisi viene incluso direttamente nell'oggetto TileLayerData tramite il campo saturatedCount, che tiene traccia del numero di punti saturati in uno specifico tile.
- *Outlier*. I dati elaborati da Q1 vengono raggruppati per chiave printId_tileId e raccolti in una sliding window count-based (countWindow(3, 1)), che consente di analizzare tre layer consecutivi con un passo di 1. La Q2 calcola gli outlier ed i risultati vengono inclusi nell'oggetto TileLayerData tramite campi specifici.
- *Clustering*. Infine, sul DataStream in uscita da Q2 viene applicata la Q3. Utilizzando l'algoritmo di clustering DBSCAN (per il quale è stata adottata un'implementazione basata su scikit-learn, per garantire coerenza con la baseline), gli outlier vengono raggruppati in cluster e vengono calcolati i centroidi dei cluster risultanti, anch'essi aggiunti all'oggetto TileLayerData.

I risultati di ciascuna query vengono pubblicati, al loro "termine", su un topic dedicato queryX-results, dove X rappresenta il numero della query.

### B. Ottimizzazione Flink

L'applicazione era già stata progettata in modo ottimizzato, sfruttando il parallelismo di Flink nelle fasi successive alla Q1 grazie all'uso di keyBy. Tuttavia, nella fase di elaborazione della Q1, mancava questa partizione logica dello stream: di conseguenza, anche impostando un grado di parallelismo superiore a 1, tutti i dati venivano comunque processati da un unico task, creando un potenziale collo di bottiglia.

Per raggiungere l'obiettivo di ottimizzazione è stato quindi sufficiente aggiungere un'operazione di keyBy basata sulla combinazione di tileId e printId, analogamente a quanto già avviene dalla Q2, visto che nella Q1 i tile sono indipendenti tra loro. Inoltre, la chiave scelta garantisce che anche in presenza di nuove stampe (i.e. nuovi printId), i dati vengano correttamente partizionati e distribuiti tra i task.

Grazie a questa modifica, tutti gli stadi della pipeline lavorano così su stream già partizionati, evitando colli di bottiglia e sfruttando il parallelismo offerto da Flink su tutta la pipeline.

### C. Alternativa con Kafka Streams

La pipeline descritta è stata implementata anche utilizzando Kafka Streams, riutilizzando gran parte delle componenti già sviluppate per Flink ma fermandosi alla Q2. In particolare, classi come TileLayerData e le query sono state integrate senza modifiche significative.

Tuttavia, l'assenza di supporto nativo per una sliding window count-based ha richiesto l'implementazione di una soluzione personalizzata: lo SlidingWindowProcessor, che utilizza uno state store per mantenere la finestra e applicare la logica di aggregazione.

## VII. Data Visualization

La fase di visualizzazione dei dati è stata implementata utilizzando Prometheus e Grafana, più per curiosità e diletto che per una reale necessità operativa. L'obiettivo iniziale era monitorare le prestazioni delle prime due query in Flink per poi confrontarle con quelle di Kafka Streams (per questo motivo la visualizzazione si limita a Q1 e Q2). Tuttavia, per la valutazione finale delle metriche di latenza e throughput, si è preferito affidarsi direttamente al Challenger, rendendo questa fase non essenziale ai fini del report.

Nonostante ciò, trattandosi di un aspetto potenzialmente interessante e ormai implementato, è stato comunque documentato.

Prometheus raccoglie le metriche esposte dal JobManager e dai TaskManager di Flink, configurati tramite il file prometheus.yml. È inoltre possibile abilitare la raccolta di queste metriche personalizzate in Flink impostando la costante ENABLE_PROFILING a true nel codice StreamingJob.java.

Grafana, invece, utilizza tali metriche per generare dashboard interattive, come definito nei file datasource.yml e flink-metrics.json. I grafici mostrano:

- Latenza mediana per query, che misura il tempo di elaborazione per Q1 e Q2.
- Throughput, che indica il numero di elementi elaborati al secondo per Q1 e Q2.

È importante notare che i grafici mostrano gli ultimi valori misurati anche quando il sistema non elabora dati. Questo comportamento è dovuto a Prometheus, che continua a riportare l'ultimo valore registrato in assenza di nuove misurazioni, senza scendere automaticamente a zero come ci si potrebbe aspettare.

## VIII. Risultati

Il Consumer è il componente responsabile della gestione dei risultati. I risultati di ciascuna query vengono esportati in formato .csv e, per l'ultima query, vengono salvati anche in formato .jsonl per l'invio al Challenger e la validazione finale del benchmark.

La logica di funzionamento del Consumer segue un ordine preciso nella lettura dei topic Kafka per garantire prestazioni quanto più reali possibili, evitando ritardi inutili dovuti, ad esempio, alla scrittura dei risultati delle prime query in file .csv prima di aver completato il benchmark. In particolare:

- Vengono prima letti i risultati della terza query dal topic query3-results e salvati in .jsonl per essere

immediatamente inviati al Challenger. Questo permette di chiudere il benchmark nel minor tempo possibile. Solo dopo la chiusura, i dati della terza query vengono salvati anche in formato `.csv`.

- Successivamente, vengono letti i risultati delle prime due query dai rispettivi topic e salvati in `.csv`.

## IX. Prestazioni

### A. Misurazione delle prestazioni

La valutazione sperimentale delle prestazioni della pipeline è stata effettuata facendo inviare al Consumer l'output della Q3 al LOCAL-CHALLENGER al termine del benchmark. Tali prestazioni sono, però, da considerare come un *upper bound* per il tempo di processamento vero e proprio, in quanto il coinvolgimento di altri attori (e.g. Producer e Consumer) introduce un inevitabile overhead di comunicazione.

Le specifiche tecniche della macchina utilizzata per la misura delle prestazioni sono riportate nella Tabella I. Per una maggiore leggibilità, oltre le tabelle II, III e IV, sono stati creati dei grafici che mostrano l'andamento delle metriche di throughput e latenza per ogni configurazione.

### B. Versione "short"

La versione "short" del benchmark è stata introdotta per confrontare le prestazioni tra Flink e Kafka Streams, concentrandosi sulle prime due query. In questa modalità, il Consumer si ferma alla lettura e gestione dei risultati delle prime due query, che vengono esportati esclusivamente in formato `.csv`. Per garantire un confronto equo, vengono generati e inviati al Challenger risultati fittizi per la terza query, simulando un output vuoto di Q3 e permettendo di completare il benchmark senza influenzare le metriche di prestazione.

### C. Baseline

La baseline rappresenta il processamento in locale, tramite codice Python, dello stream dei dati generato dal Challenger senza l'utilizzo di altri sistemi. Questo script, messo a disposizione dal DEBS, ha lo scopo di fornire un punto di partenza rispetto al quale confrontare le prestazioni delle soluzioni distribuite.

### D. Flink vs. baseline

Come mostrato nelle Figure 6 e 9, la baseline presenta una latenza significativamente inferiore rispetto a Flink, circa la metà. Questo è direttamente attribuibile al fatto che Flink, essendo un sistema distribuito che si interfaccia con Kafka, introduce un overhead di latenza rispetto a un'elaborazione locale.

Tuttavia, il throughput di Flink è decisamente superiore rispetto alla baseline, come evidenziato in Figura 3 dove Flink risulta essere 16 volte più performante. Questo confronto dimostra come un sistema distribuito consenta di elaborare grandi volumi di dati in modo efficiente, anche se a scapito della latenza.

### E. Parallelismo in Flink

La Figura 4 mostra come il throughput diminuisca leggermente all'aumentare del grado di parallelismo. Tuttavia, analizzando la Tabella II, si osserva che il grado di parallelismo più elevato si avvicina alle prestazioni delle configurazioni con parallelismo 2 o senza parallelismo, mantenendo però una deviazione standard più contenuta garantendo un throughput più stabile nel tempo.

Come previsto, però, le Figure 7 e 10 mostrano un aumento della latenza con l'aumentare del grado di parallelismo. Questo è dovuto ai meccanismi di sincronizzazione richiesti per coordinare l'esecuzione delle diverse task. In questo contesto, un maggiore grado di parallelismo non porta alcun vantaggio in termini di throughput e comporta anche un incremento della latenza.

Per un confronto immediato tra throughput e latenza tra le configurazioni, è stato creato un grafico osservabile nella Figura 12.

### F. Flink vs. Kafka Streams

Il confronto riportato nella Figura 5 mostra come Flink, nella versione "short", raggiunga un throughput notevolmente superiore rispetto a Kafka Streams, risultando circa quattro volte più performante. Questo vantaggio è principalmente dovuto alle differenze architetturali tra i due sistemi. Kafka Streams, nella configurazione adottata, non utilizza più nodi worker, centralizzando completamente il carico, e non fornisce un meccanismo esplicito per le sliding window count-based. Flink, al contrario, fa forza sulla sua natura distribuita, garantendo un'elaborazione delle tuple più efficiente grazie all'uso di DAG ottimizzati e di finestre specializzate.

Un'ulteriore differenza può risiedere nella gestione dello stato: Kafka Streams lo salva su disco, il che implica un overhead maggiore rispetto a Flink che, invece, lo mantiene in memoria, riducendo i tempi di accesso e migliorando le prestazioni complessive.

Per quanto riguarda la latenza, come mostrato nelle Figure 8 e 11, Flink mantiene comunque un vantaggio, dimostrando la sua capacità di sostenere un elevato throughput senza compromettere eccessivamente la latenza.

### G. Flink Optimized

L'ottimizzazione applicata, come si può dedurre dalle Tabelle II, III e IV, ha comportato un peggioramento delle prestazioni, sia in termini di throughput che di latenza.

Sebbene l'ottimizzazione, a livello logico, dovrebbe migliorare le performance, Flink sembra gestire meglio la versione non ottimizzata, costruendo un DAG più efficiente che utilizza solo due task rispetto ai tre della versione ottimizzata, come illustrato nelle Figure 13 e 14.

Questo comportamento, in controtendenza rispetto alle aspettative, potrebbe essere dovuto al fatto che la parallelizzazione introdotta non viene sfruttata appieno dato il numero limitato di batch analizzati (160 su 3600 disponibili). Inoltre, evidenzia come in tal caso l'ottimizzazione interna di Flink sia già in grado di produrre una soluzione performante.

**Fig. 3:** Confronto throughput tra la baseline e Flink senza parallelismo.

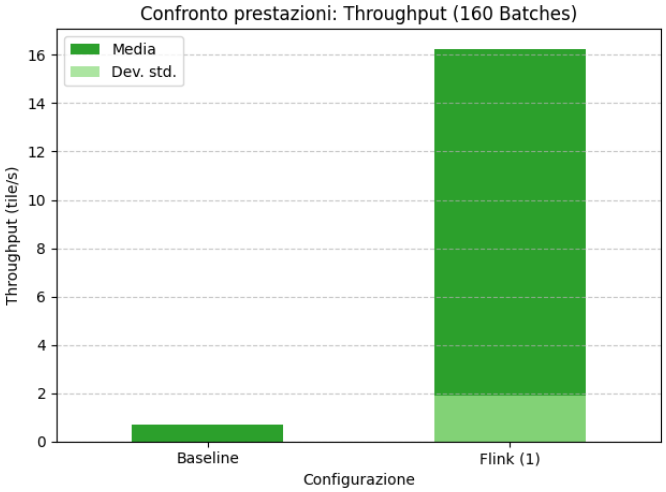


**Fig. 4:** Confronto throughput di Flink al variare del grado di parallelismo.

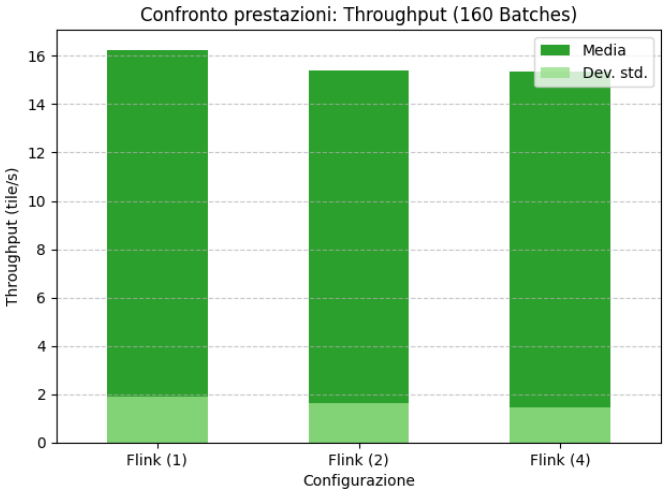


**Fig. 5:** Confronto throughput tra Flink (short) e Kafka Streams.

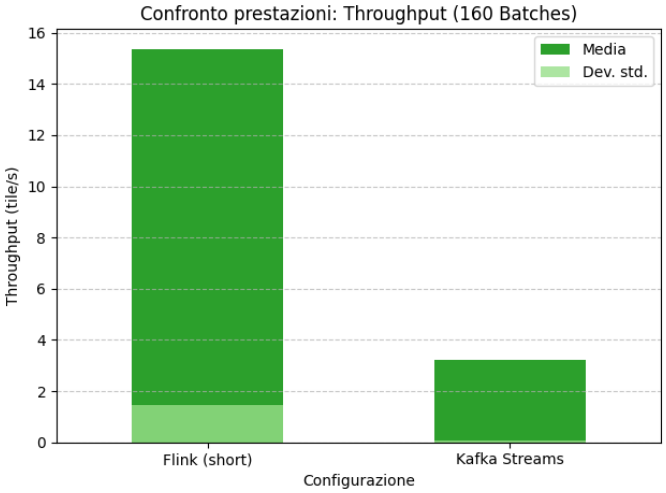

**Fig. 6:** Confronto latenza tra la baseline e Flink senza parallelismo.

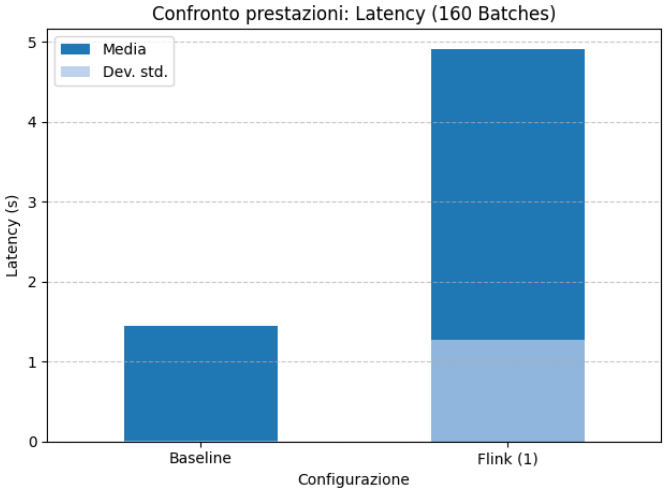


**Fig. 7:** Confronto latenza di Flink al variare del grado di parallelismo.

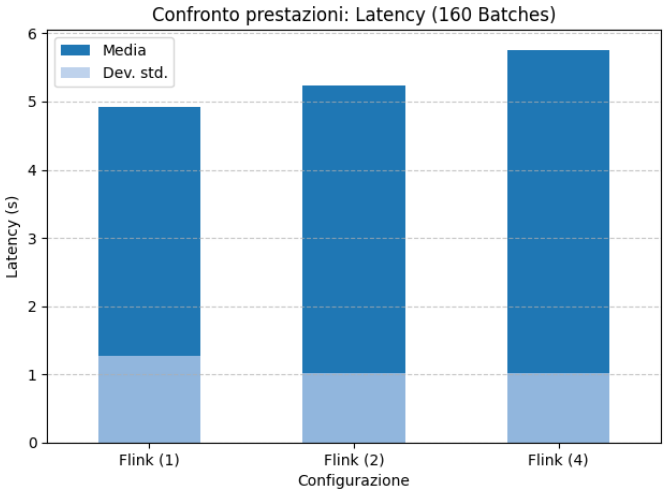


**Fig. 8:** Confronto latenza tra Flink (short) e Kafka Streams.

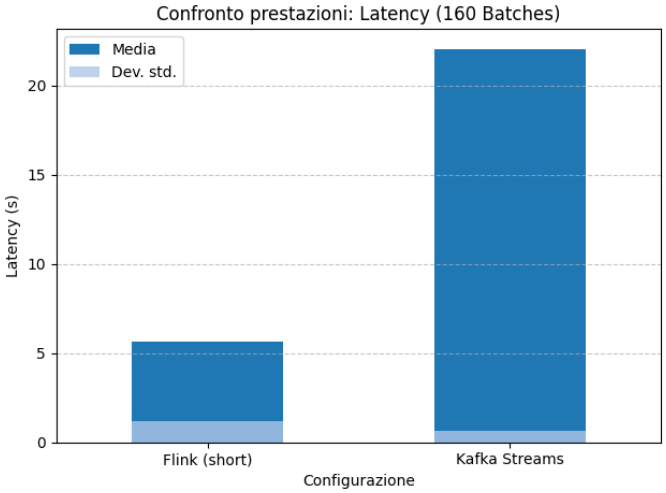

**Fig. 9:** Confronto 99esimo percentile della latenza tra la baseline e Flink senza parallelismo.

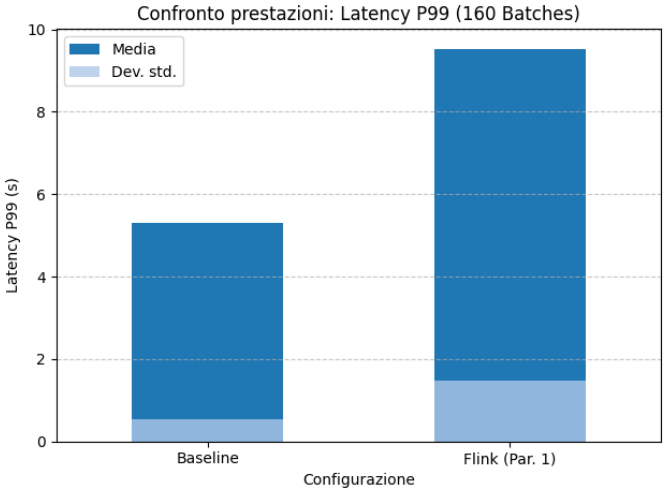


**Fig. 10:** Confronto 99esimo percentile della latenza di Flink al variare del grado di parallelismo.

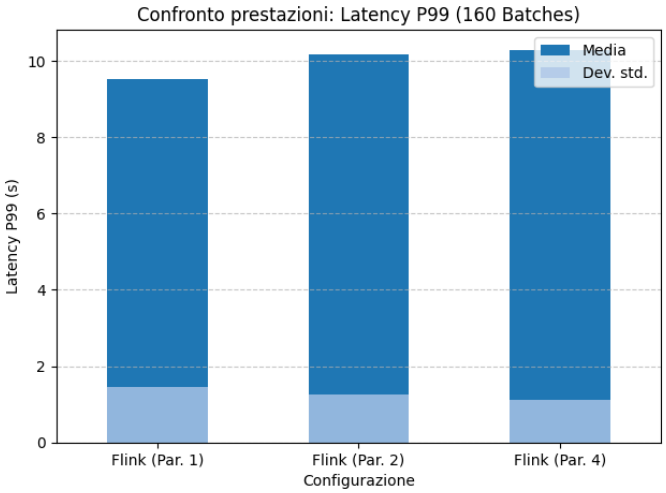


**Fig. 11:** Confronto 99esimo percentile della latenza tra Flink (short) e Kafka Streams.

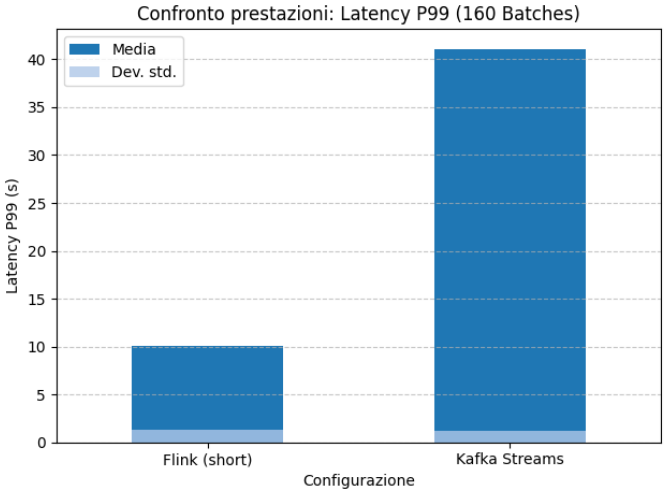

**Fig. 12:** Confronto throughput e latenza tra le configurazioni.

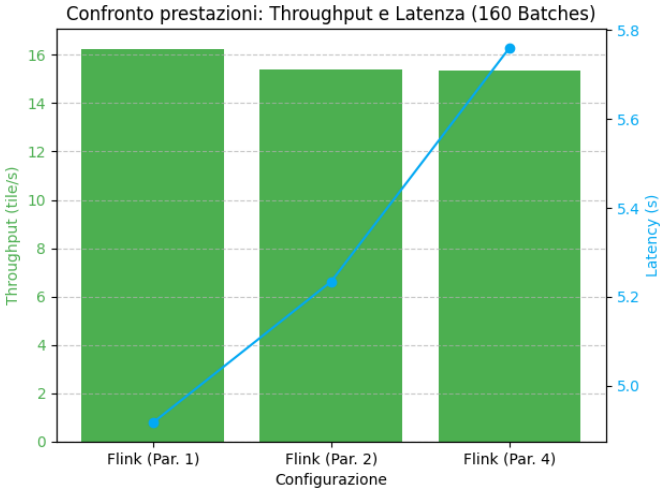


**Fig. 13:** DAG di Flink per la pipeline non ottimizzata.

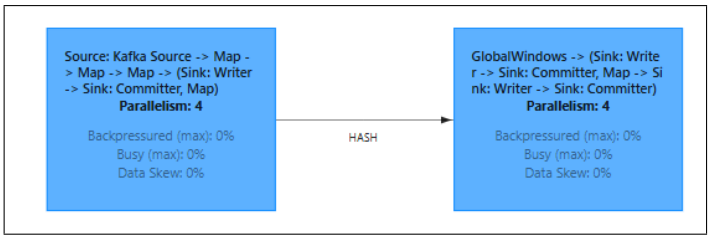


**Fig. 14:** DAG di Flink per la pipeline ottimizzata.

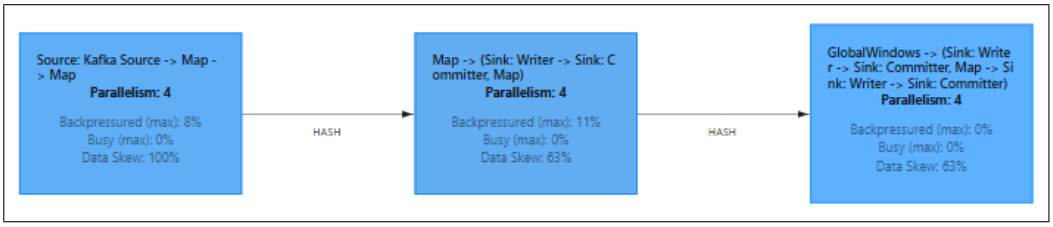

**TABLE I:** Specifiche tecniche della macchina utilizzata per i test.

| RAM | CPU |
|---|---|
| 32 GB DDR4 | Ryzen 7 5700x3D 8c-16t |

**TABLE II:** Metriche di throughput con 160 batches analizzati. 10 run per configurazione.

| Configuration | Avg Thr (tile/s) | Min Thr (tile/s) | Max Thr (tile/s) | Std Dev (tile/s) |
|---|---|---|---|---|
| Baseline | 0.6899 | 0.6823 | 0.6975 | 0.0050 |
| Flink (par. 1) | 16.2556 | 13.1517 | 18.3428 | 1.8908 |
| Flink (par. 2) | 15.3864 | 13.0926 | 18.0242 | 1.6523 |
| Flink (par. 4) | 15.3525 | 12.6544 | 17.8985 | 1.4554 |
| Flink Opt (par. 2) | 15.1204 | 12.1014 | 18.4769 | 2.4161 |
| Flink Opt (par. 4) | 14.9698 | 12.2903 | 18.3606 | 2.1903 |
| Flink (short) | 15.3694 | 12.3847 | 18.0485 | 1.4497 |
| Kafka Streams | 3.2339 | 3.1212 | 3.3190 | 0.0828 |

**TABLE III:** Metriche di latenza con 160 batches analizzati. 10 run per configurazione.

| Configuration | Avg Latency (s) | Min Latency (s) | Max Latency (s) | Std Dev (s) |
|---|---|---|---|---|
| Baseline | 1.4487 | 1.4327 | 1.4647 | 0.0106 |
| Flink (par. 1) | 4.9169 | 3.6712 | 7.0558 | 1.2683 |
| Flink (par. 2) | 5.2347 | 3.9548 | 6.4026 | 1.0155 |
| Flink (par. 4) | 5.7604 | 3.9733 | 7.3819 | 1.0137 |
| Flink Opt (par. 2) | 5.8377 | 4.1117 | 8.8437 | 1.6942 |
| Flink Opt (par. 4) | 6.1952 | 4.0972 | 8.5302 | 1.4445 |
| Flink (short) | 5.6726 | 4.0018 | 7.8372 | 1.1809 |
| Kafka Streams | 22.0466 | 21.1692 | 23.1880 | 0.6186 |

**TABLE IV:** Metriche del 99esimo percentile della latenza con 160 batches analizzati. 10 run per configurazione.

| Configuration | Avg P99 Lat (s) | Min P99 Lat (s) | Max P99 Lat (s) | Std Dev (s) |
|---|---|---|---|---|
| Baseline | 5.2950 | 4.2025 | 5.8904 | 0.5438 |
| Flink (par. 1) | 9.5290 | 8.4013 | 12.0297 | 1.4661 |
| Flink (par. 2) | 10.1718 | 8.3963 | 12.0768 | 1.2452 |
| Flink (par. 4) | 10.2958 | 8.5503 | 12.4891 | 1.1091 |
| Flink Opt (par. 2) | 10.5436 | 8.3949 | 13.0780 | 1.6998 |
| Flink Opt (par. 4) | 10.7250 | 8.4873 | 12.8777 | 1.6480 |
| Flink (short) | 10.0220 | 8.3202 | 12.7697 | 1.3497 |
| Kafka Streams | 41.0932 | 39.9739 | 42.8928 | 1.2239 |