

# **REAL-TIME MONITORING OF DEFECTS IN L-PBF MANUFACTURING**

---

**SABD - 2024/25**

Massimo Buniy - 0350022

[massimo.buniy@students.uniroma2.eu](mailto:massimo.buniy@students.uniroma2.eu)



# Programma

- 1 Obiettivi del progetto
- 2 Architettura
- 3 Dettagli Middleware e Flink
- 4 Strategie di test
- 5 Soluzione Java

# OBIETTIVI DEL PROGETTO



**MONITORAGGIO IN  
TEMPO REALE DEI  
DIFETTI IN L-PBF**

**ELABORAZIONE DI  
IMMAGINI TERMICHE  
TOMOOGRAFICHE**

Rilevamento di:

- pixel saturi
- outlier termici
- cluster di anomalie

**ANALISI DELLE  
SOLUZIONI  
PROPOSTE**

# L-PBF

## TECNOLOGIA - ADDITIVE MANUFACTURING

Tramite la  **fusione laser di strati di polvere** metallica, permette la stampa di componenti complessi.

## VANTAGGI

Consente la produzione precisa di  **forme complesse e componenti altamente personalizzati**.

## DIFETTI

Durante il processo L-PBF possono insorgere difetti come  **porosità, piccole fratture e deformazioni**, che compromettono la qualità e le prestazioni dei componenti. Il monitoraggio in tempo reale è fondamentale per individuarli e ridurli.



**1**

**RILEVARE  
DIFETTI  
DURANTE IL  
PROCESSO**

**2**

**RIDURRE  
RILAVORAZIONI  
E SCARTI**

**3**

**SUPROTARE  
INTERVENTI  
RAPIDI E MIRATI**

---

**NECESSITÀ DI ELABORAZIONE  
STREAMING E DISTRIBUITA**

# Soluzione proposta

## Architettura basata su:

- Apache Flink
- Kafka
- Redis
- Middleware (in Python)

Middleware



Flink



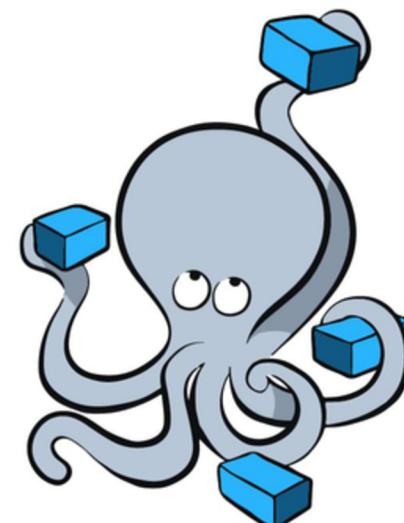
Redis



Kafka



## Tutto quanto containerizzato



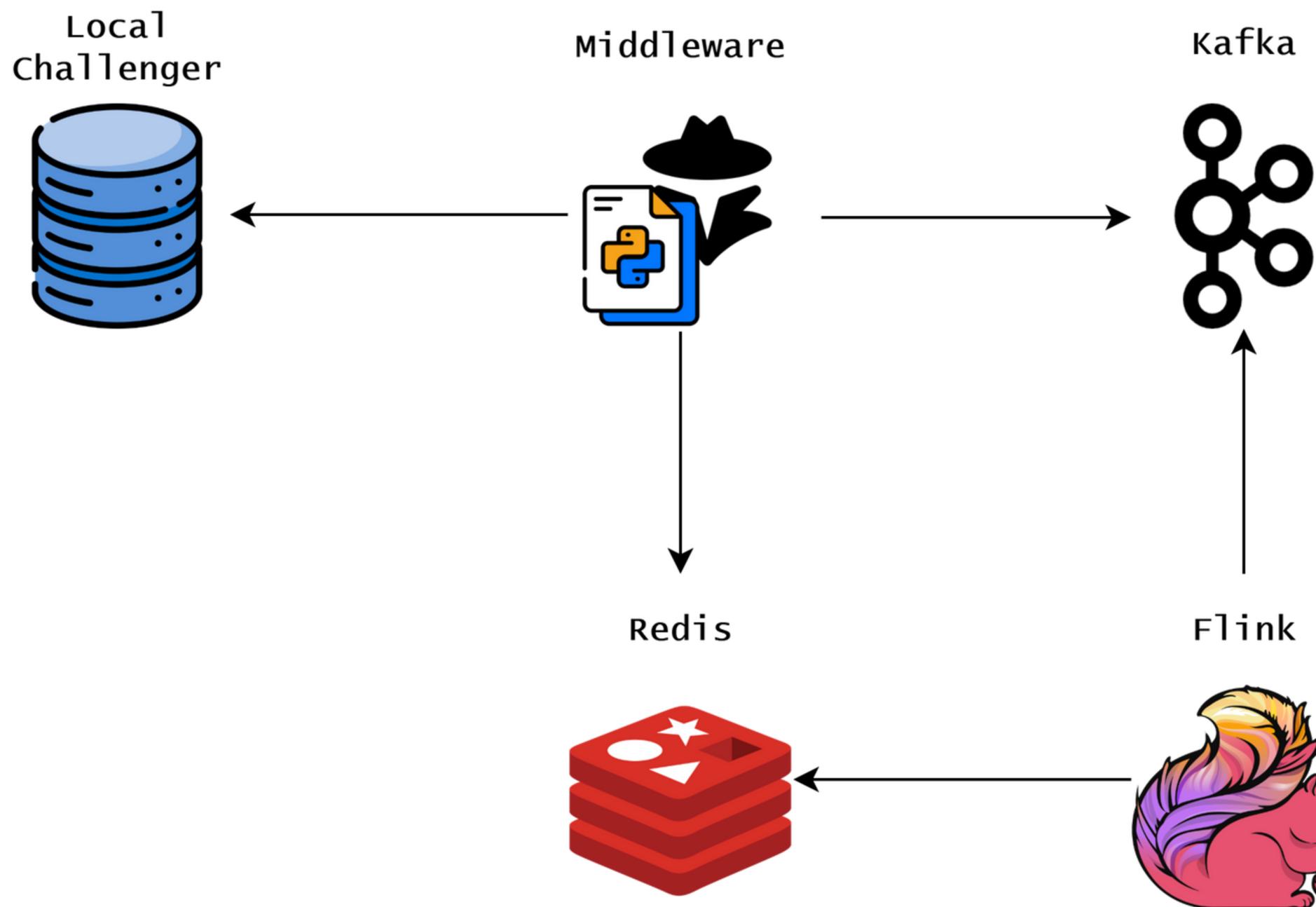
**docker**  
Compose

## Output

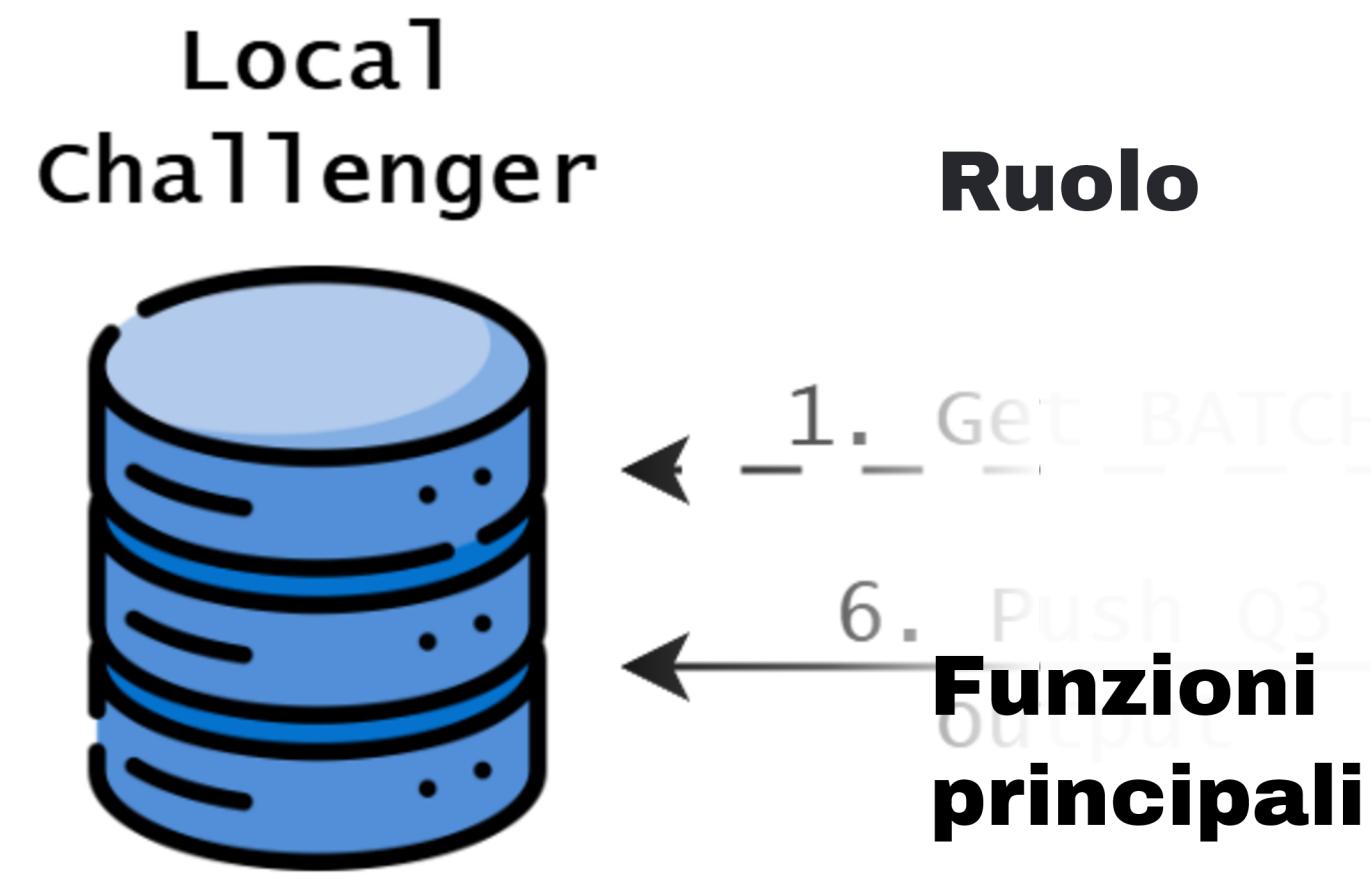


Comma  
Separated  
Value

# Architettura generale



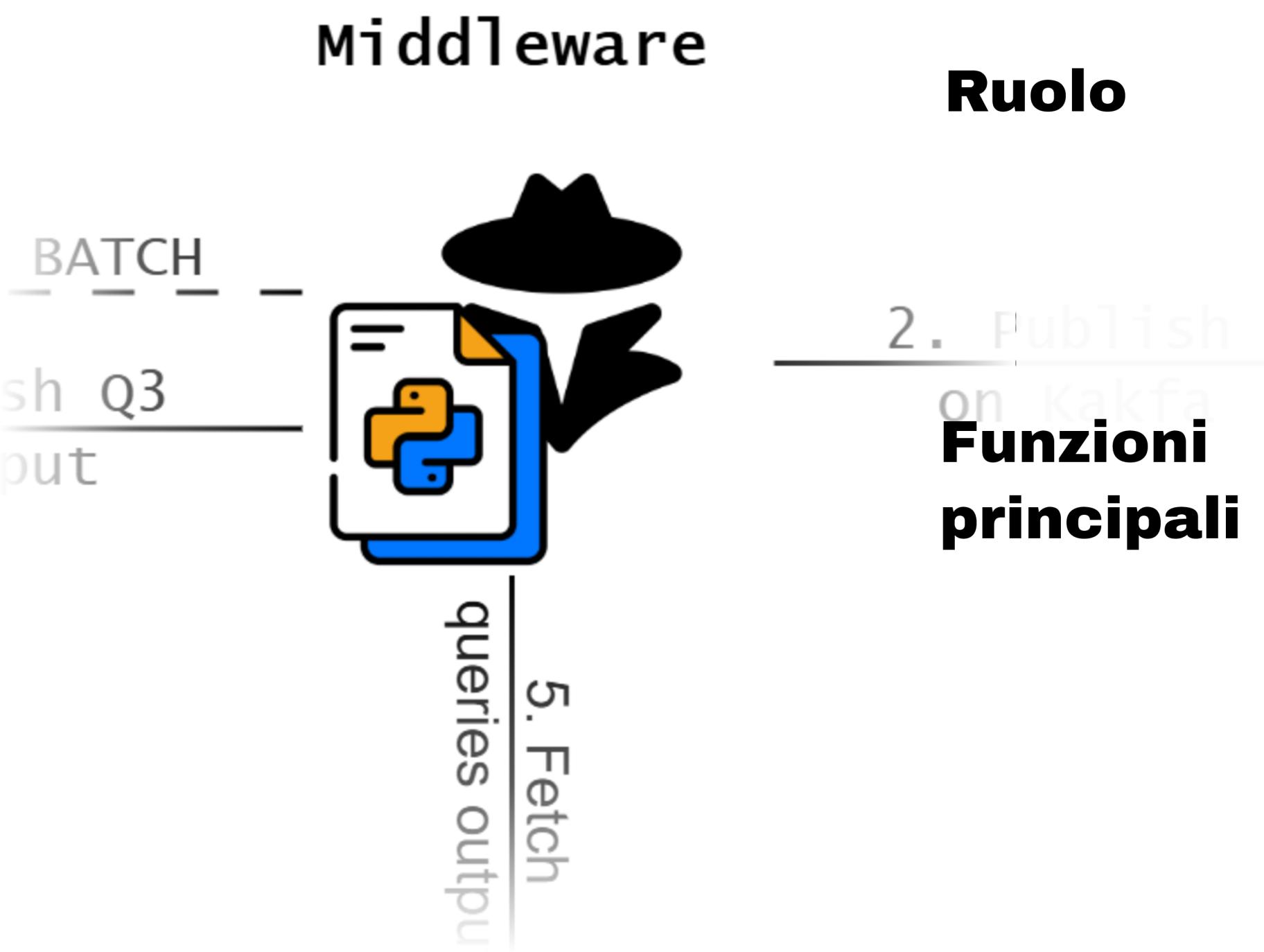
# Local Challenger



Componente centrale per la validazione dei risultati prodotti dal sistema di analisi real-time.

- Server REST
- Fornire le immagini OT
- Calcolare le metriche di performance:
  - Latenza
  - Throughput

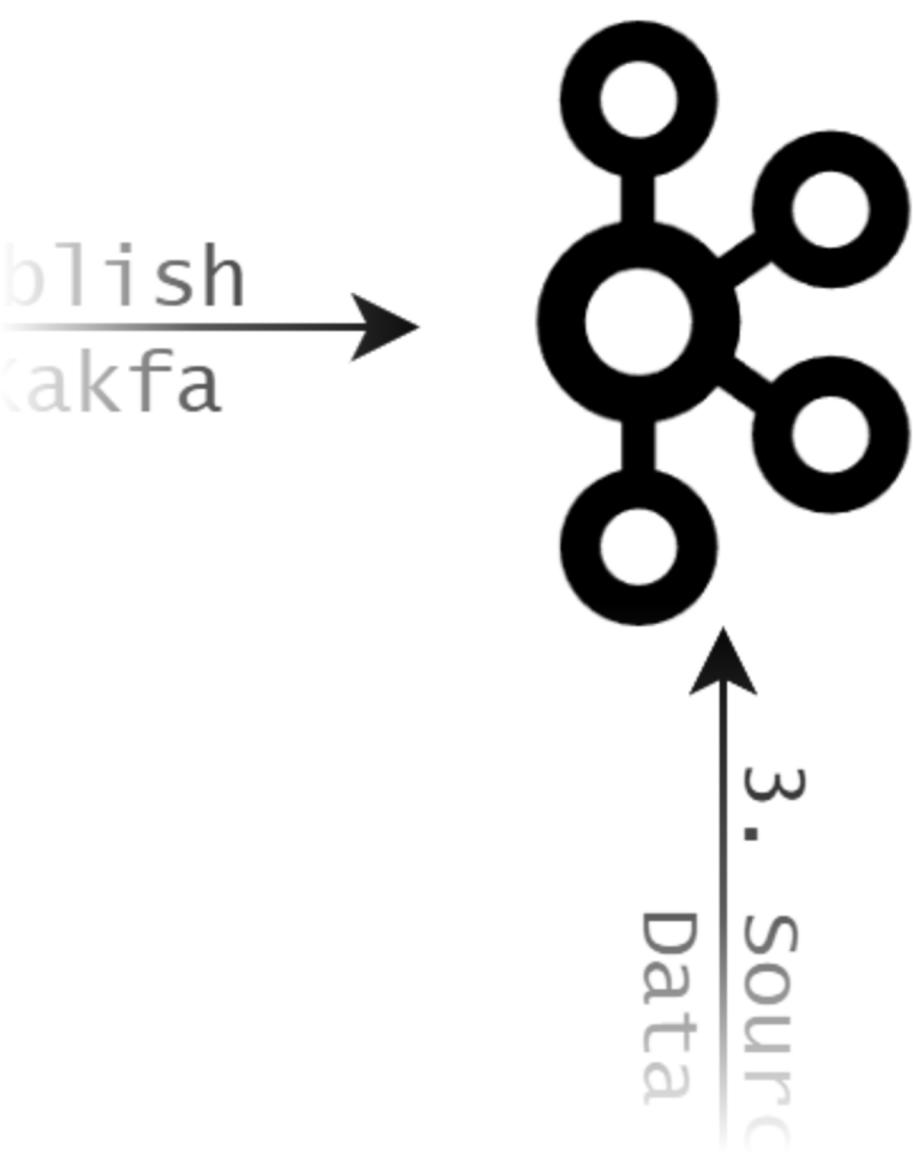
# Middleware



Ponte tra sistema di elaborazione (Flink/Kafka) ed il local-challenger

- Polling periodico del server per nuovi batch
- Push dei batch su Kafka
- In ascolto su Redis per i risultati del processamento di Flink
- Invio dei risultati al server via HTTP POST

# Kafka



## Ruolo

## Funzioni principali

Sistema di messaggistica che collega i componenti della pipeline di elaborazione dati

- Bufferizzazione e trasporto dei dati tra il produttore (middleware) ed il consumatore (Flink)
- Garantisce affidabilità, scalabilità e resilienza nella trasmissione dei messaggi
- Permette l'elaborazione asincrona e in tempo reale dei dati

# Flink

Source  
Data  
sink

Flink



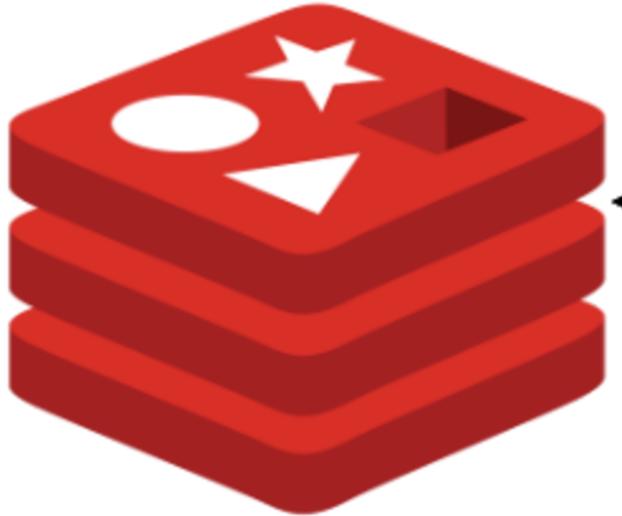
Ruolo

Funzioni  
principali

Motore di elaborazione dati in streaming per analisi e processamento in tempo reale

- Consumo dei dati da Kafka
- Applicazione di filtri, trasformazioni e analisi sui dati ricevuti
- Suddivisione del lavoro in operatori paralleli
- Scrittura dei risultati verso destinazioni esterne (Kafka, Redis, CVS sink)

# Redis



**Redis**

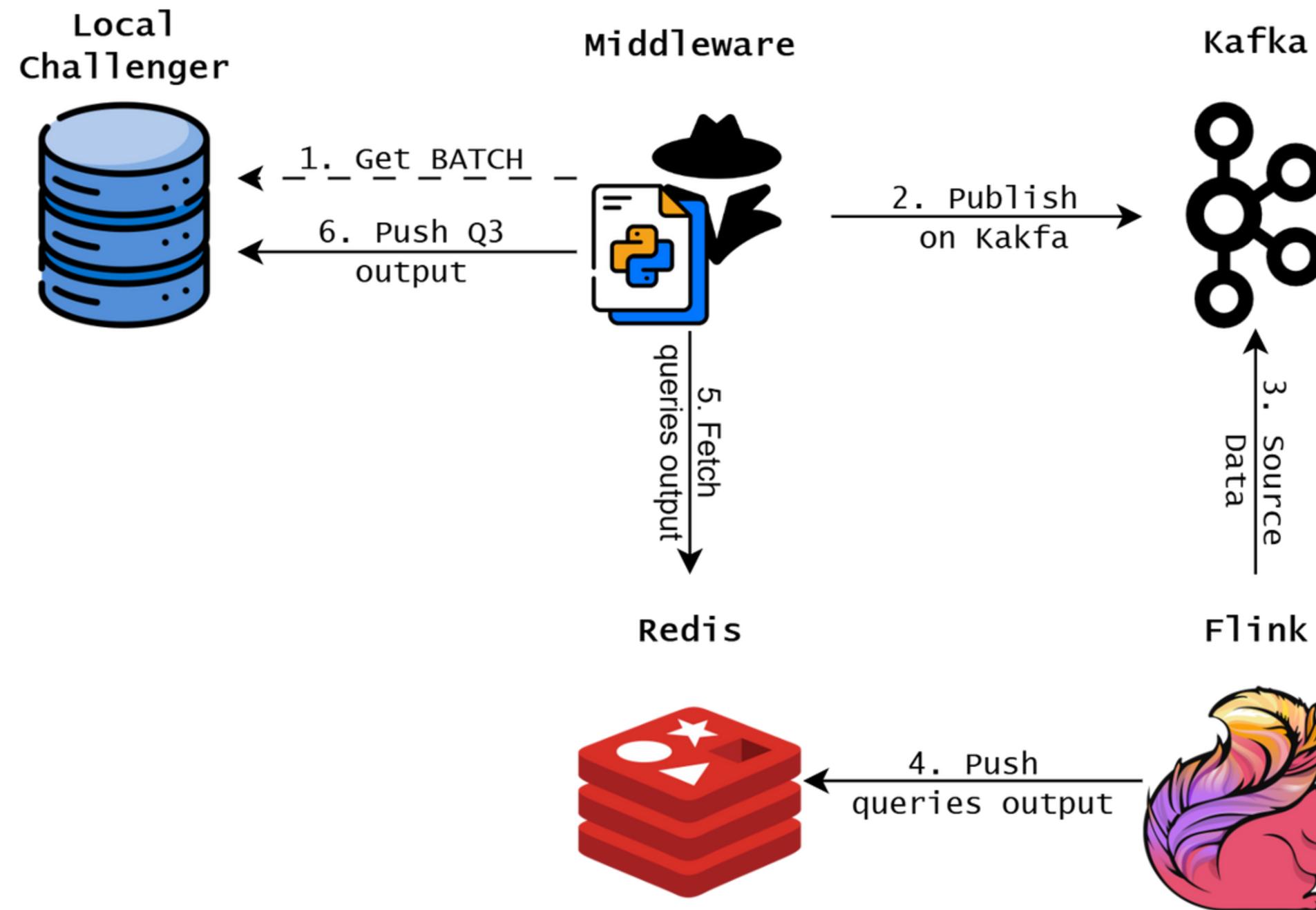
**Ruolo**

**Funzioni  
principali**

Database in-memory per la memorizzazione veloce dei risultati elaborati

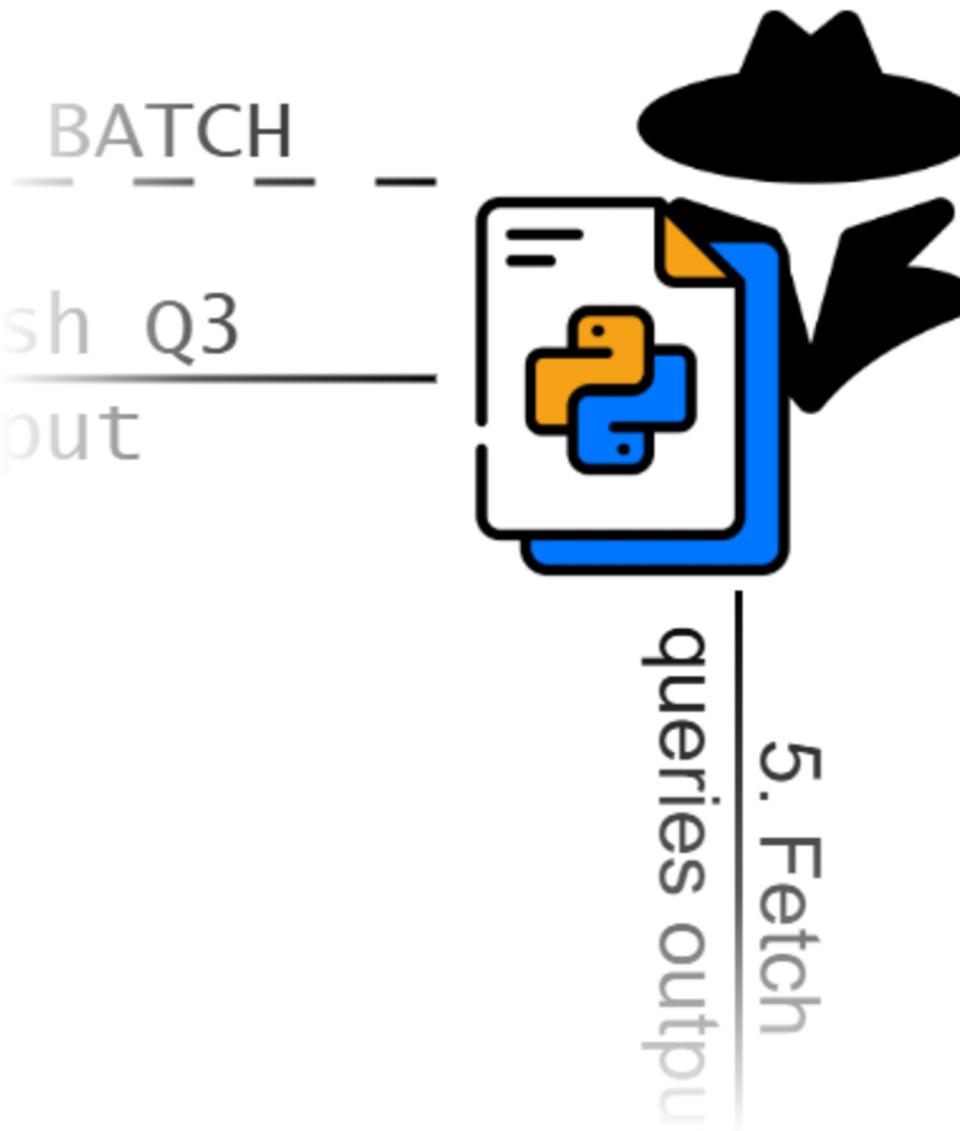
- Archiviazione temporanea e accesso rapido ai risultati
- Supporto a strutture dati semplici (chiave/valore, liste, set, ecc.)
- Condivisione dei dati tra diversi componenti del sistema
- Facilita la consultazione in tempo reale dei risultati

# Flusso dei dati dettagliato



# Middleware - Ingestion

Middleware



```
# Step 1: Create and start a benchmark  
bench_id = create_and_start_benchmark(...)  
  
# Step 2: Start threads  
batch_thread = threading.Thread(target=poll_batches, ...)
```

## Funzione: poll\_batches

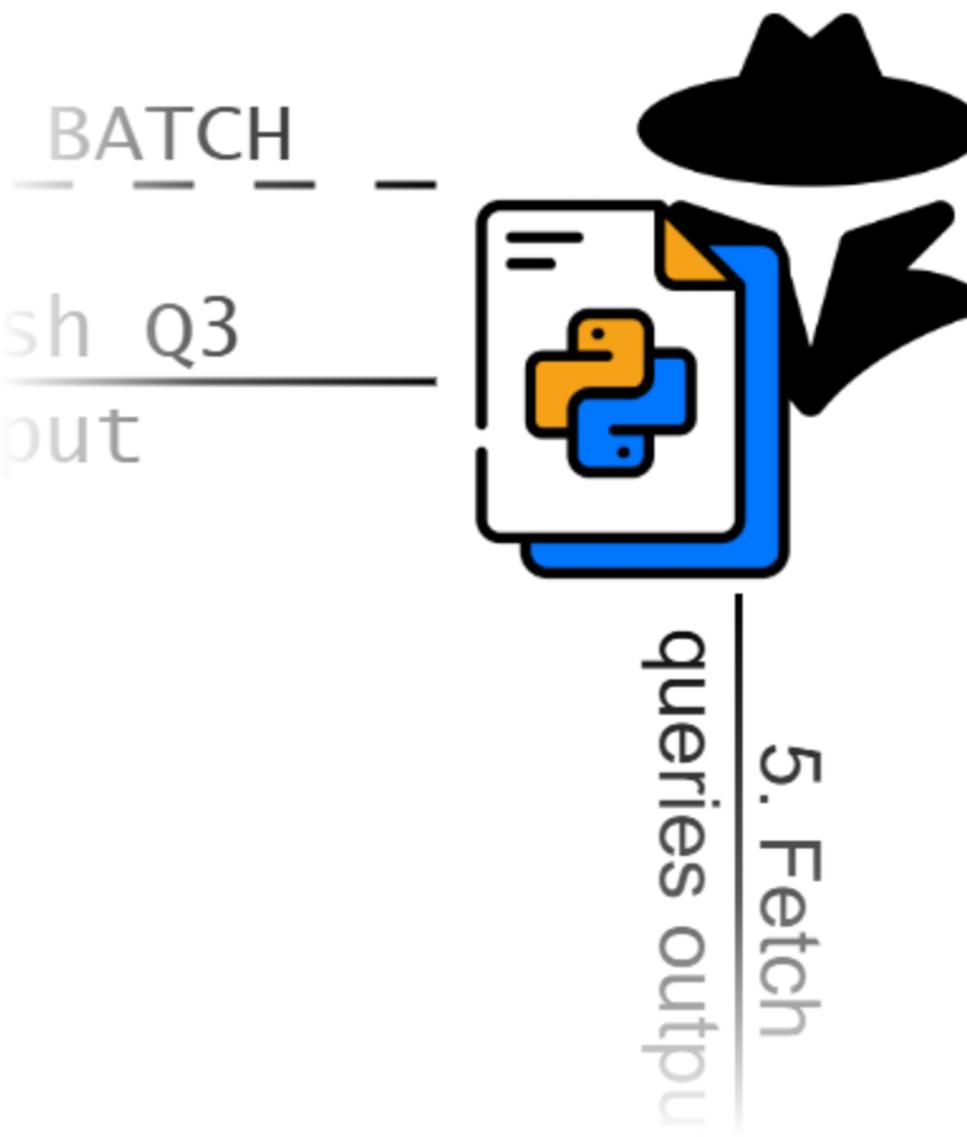
- Richieste periodiche a “/api/next\_batch”
- Pubblica sul topic Kafka “raw\_batch” i dati ricevuti dalla GET al local-challenger

## Parametri via ENV

- KAFKA\_BOOTSTRAP\_SERVERS=kafka:9092
- CHALLENGER\_URL=http://local-challenger:8866
- BURST\_SIZE=1
- INTERVAL=1
- LIMIT=500
- VERBOSE=1

# Middleware - Output

Middleware



```
# Step 2: Start threads
```

```
q1_thread = threading.Thread(target=consume_q1_redis, ...)  
q2_thread = threading.Thread(target=consume_q2_redis, ...)  
q3_thread = threading.Thread(target=consume_q3_redis, ...)
```

## Canali Redis

- Riceve da Redis su 3 canali distinti:
  - “saturated-pixels” (Q1)
  - “saturated-rank” (Q2)
  - “centroids” (Q3)

## Scrittura su file CSV

Redis è stato introdotto per un fatto di semplificazione della scrittura degli output su CSV. In Flink è presente un’implementazione via sink per la scrittura direttamente sul container

# Flink - Panoramica



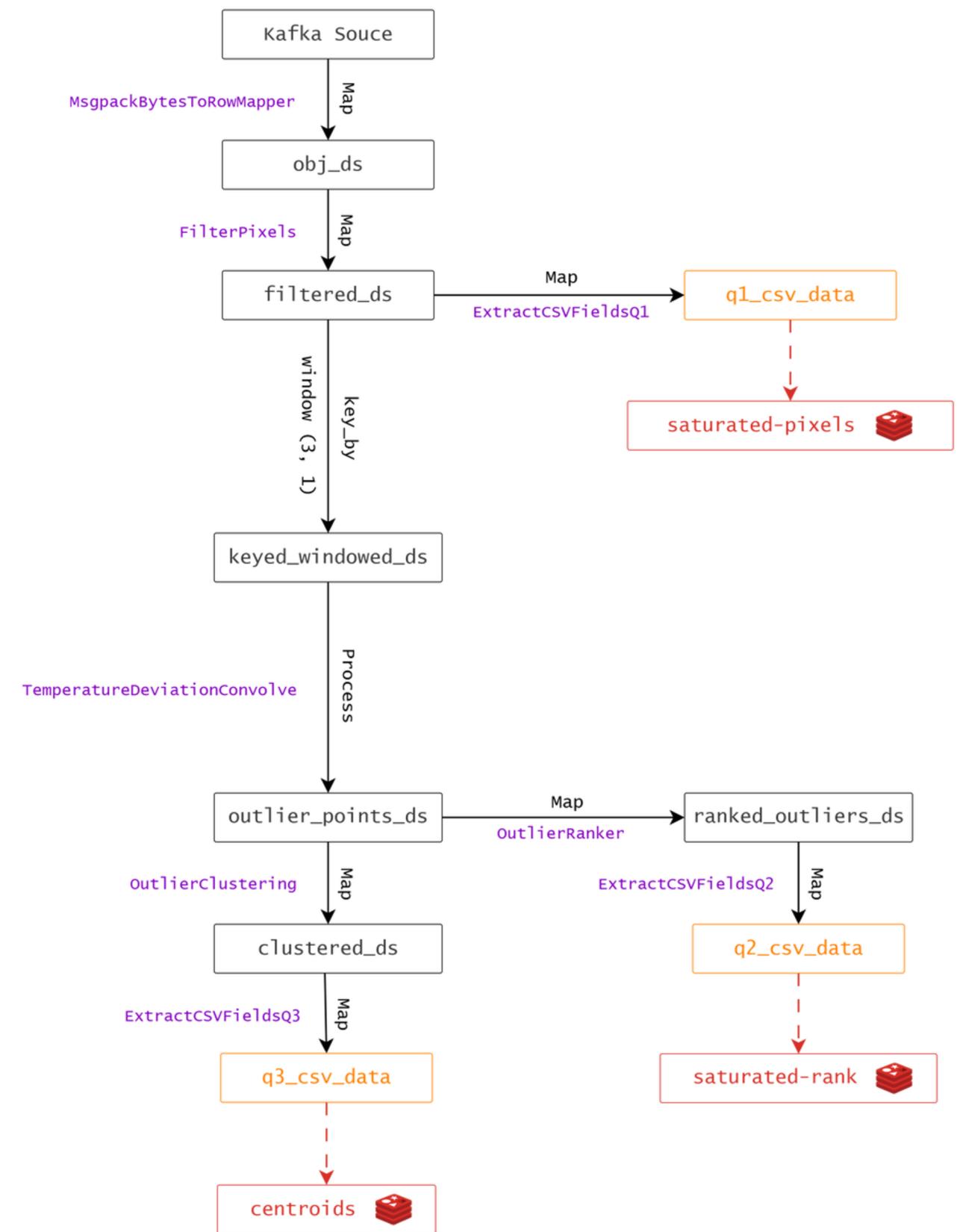
Flink

Source

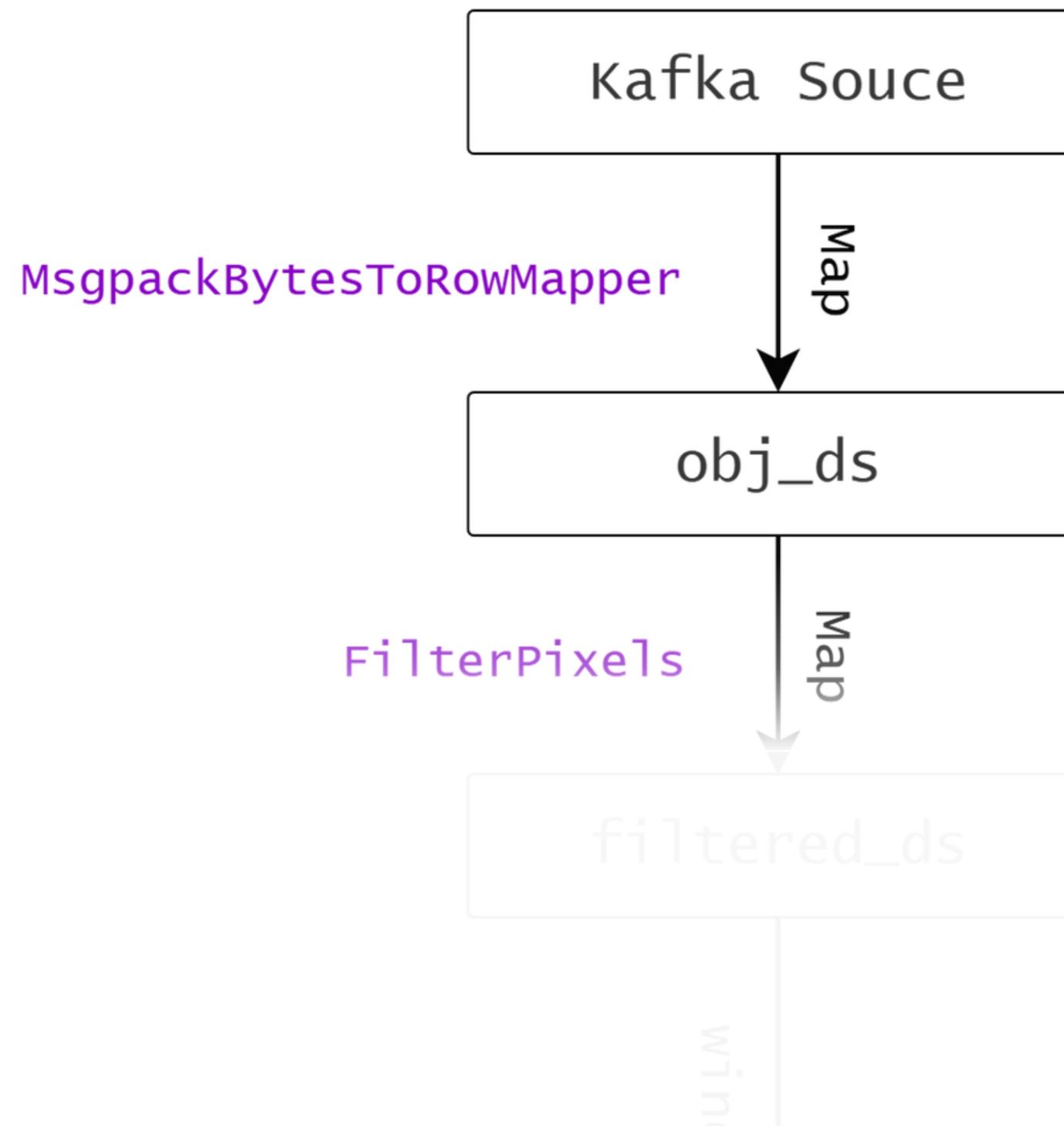
## Struttura del workflow di Flink

3 step analitici:

- Q1: pixel saturi
- Q2: outlier termici
- Q3: clustering



# Flink - Lettura e Mapping dei Dati



## Lettura dalla sorgente

- Consumo dei messaggi dal topic Kafka (raw-batch)
- I dati sono serializzati (msgpack)

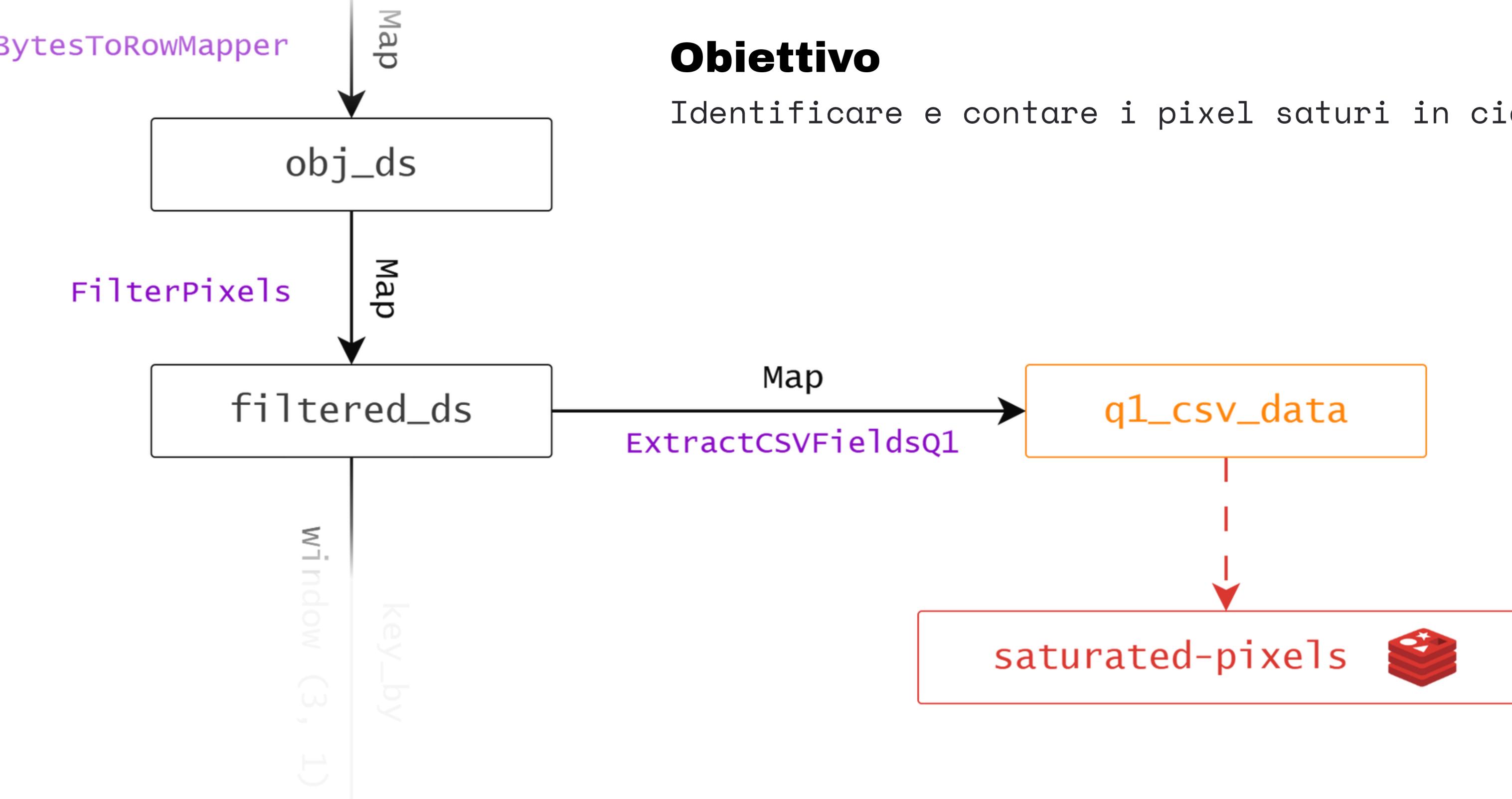
## Decodifica e parsing

- Deserializzazione dei messaggi (msgpack → Python dict)
- Estrazione dei campi (tif, print\_id, batch\_id, layer, tile\_id)

## Mapping in Row Flink

- Conversione dei dati in oggetti Row compatibili con PyFlink
- Definizione dello schema (tipi di dato e campi)

# Flink - Q1: Calcolo Pixel Saturi



## Obiettivo

Identificare e contare i pixel saturi in ciascuna tile TIFF

# Flink - Windowing

FilterPixels

Map

filtered\_ds

Map

ExtractCSVFieldsQ1

q1\_csv\_data

window (3, 1)  
key\_by

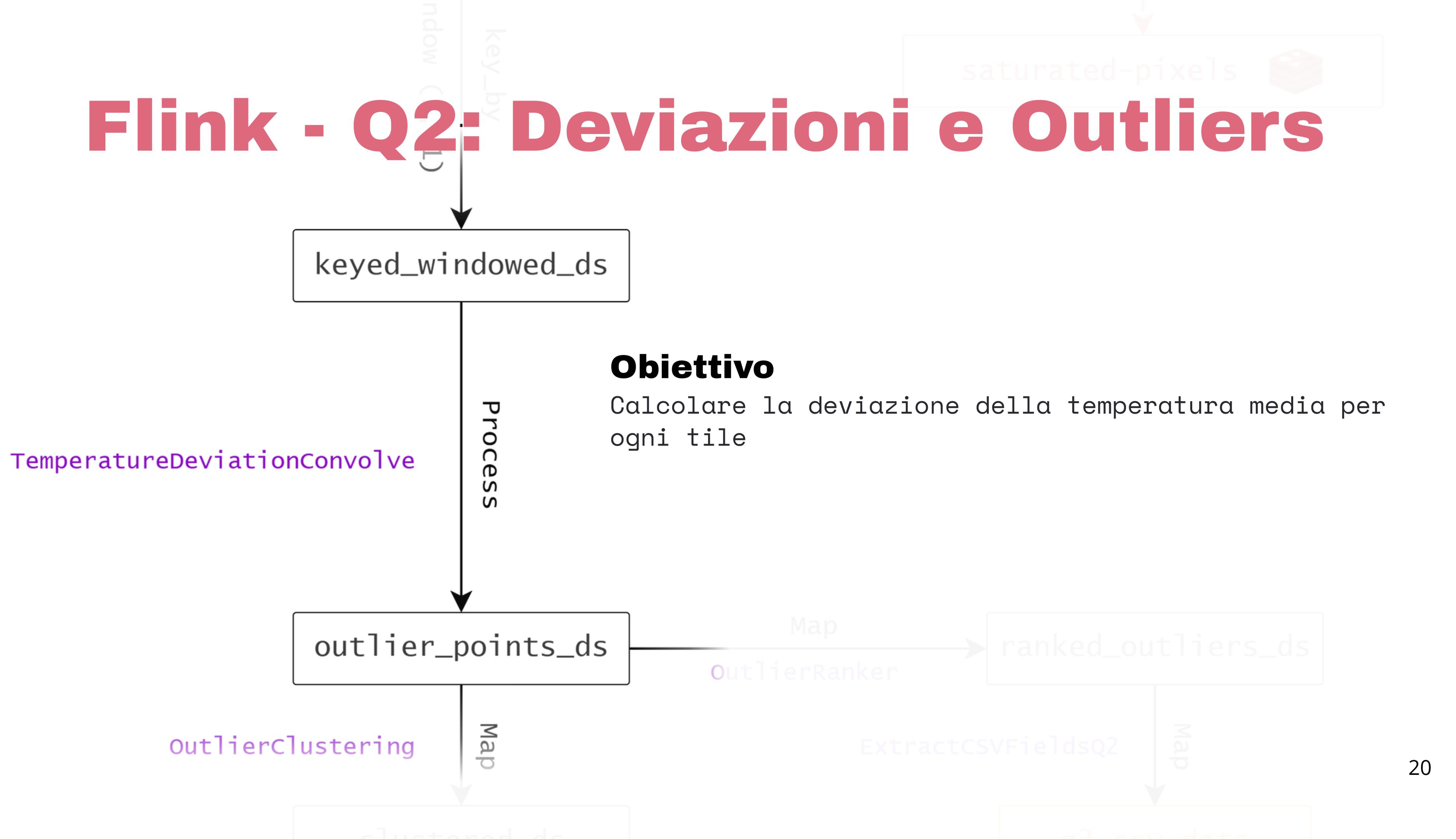
keyed\_windowed\_ds

## Obiettivo

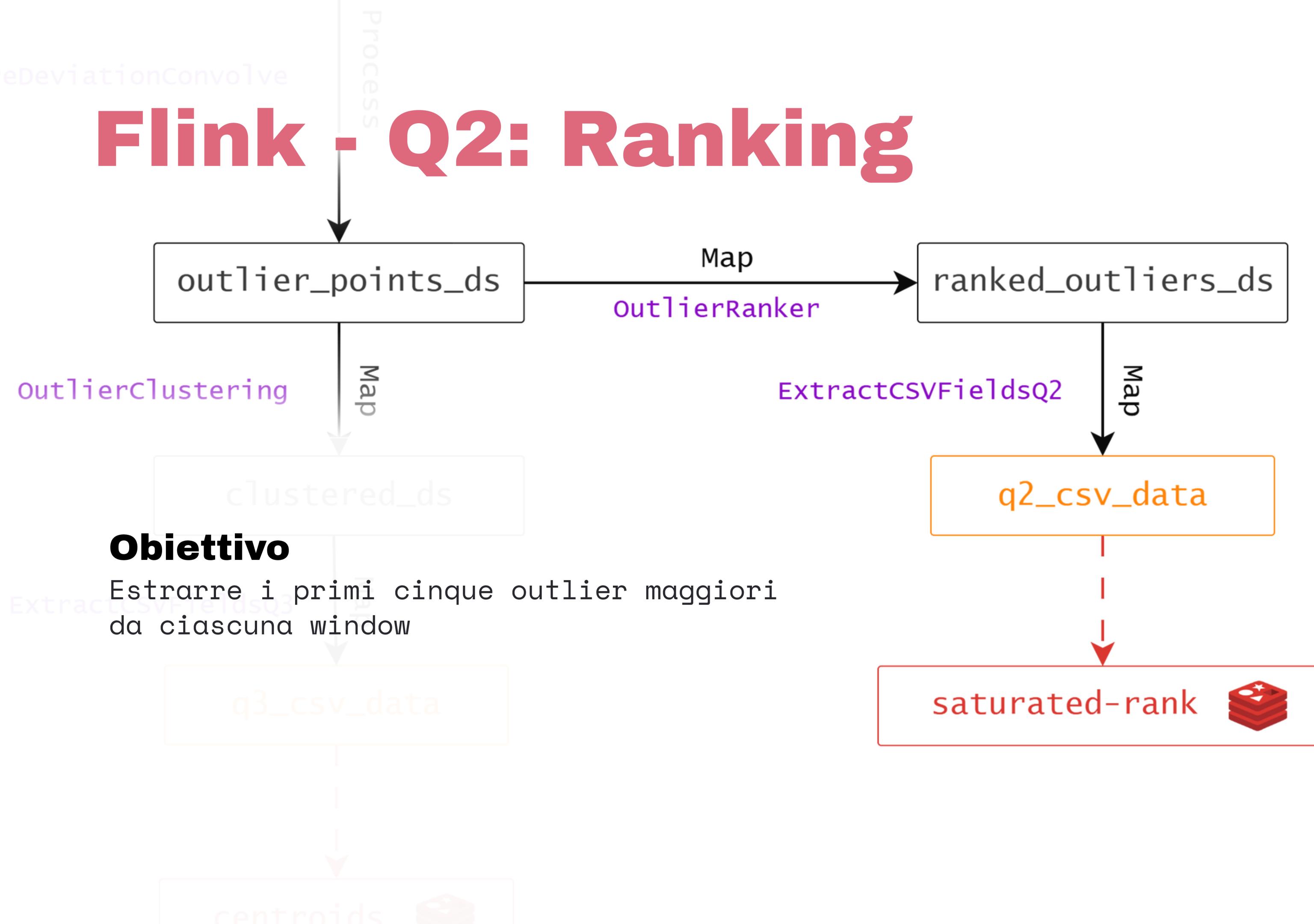
Raggruppare i dati in finestre temporali per analisi aggregate attraverso più layer

```
keyed_windowed_ds = filtered_ds  
.key_by(lambda row: row.tile_id, key_type=Types.INT())  
.count_window(3, 1)
```

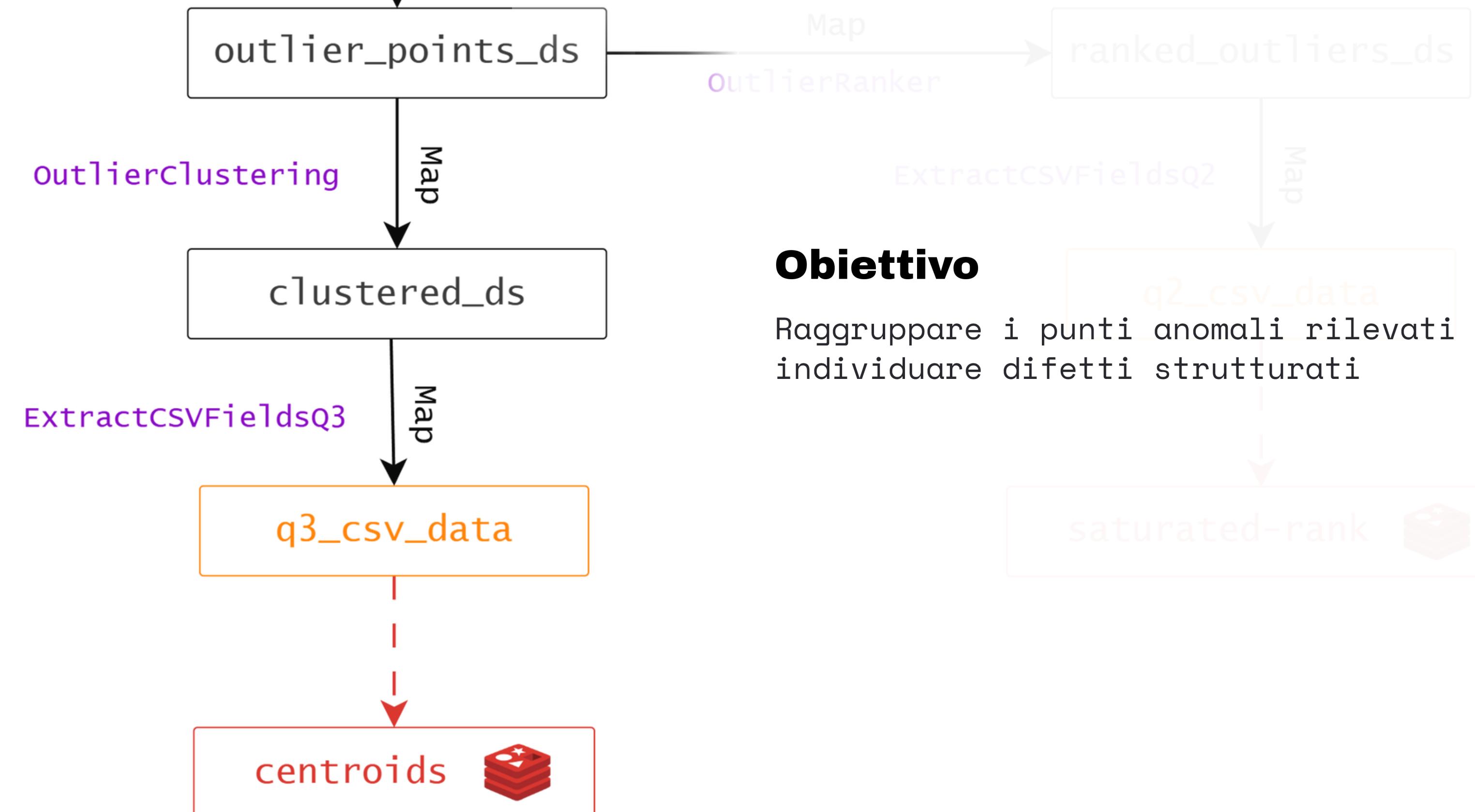
# Flink - Q2: Deviazioni e Outliers



# Flink - Q2: Ranking



# Flink - Q3: Clustering outliers



## Obiettivo

Raggruppare i punti anomali rilevati nelle tile per individuare difetti strutturati

# Strategia di Test

## Obiettivo

Valutare le prestazioni del sistema sotto diversi carichi.

---

### Tre scenari di invio batch al Local-Challenger

Invio continuo	Invio regolare	Invio burst
- BURST_SIZE = 1 - INTERVAL = 0	- BURST_SIZE = 1 - INTERVAL = 1	- BURST_SIZE = 16 - INTERVAL = 15

# Scenario 1 - Invio continuo

Il middleware invia richieste GET senza pausa



Il flusso di tile da elaborare è continuo

## Risultati

- Throughput: 2.01 batch/s
- Latenza media: 14 minuti e 47 secondi
- Latenza minima: 12 secondi
- Latenza massima: 27 minuti e 57 secondi
- Batch mancanti: 32 su 3600

Osservazioni:

La pipeline va in saturazione → la latenza cresce rapidamente

Tutti i batch vengono comunque elaborati

# Scenario 2 - Invio regolare

Delay fisso di 1 secondo tra ogni richiesta GET

## Risultati

- Throughput: 0.99 batch/s
- Latenza media: 4.36 secondi
- Latenza minima: 760 millisecondi
- Latenza massima: 40.6 secondi
- Batch mancanti: 32 su 3600

Osservazioni:

Latenza drasticamente ridotta

Comportamento stabile e prevedibile

# Scenario 3 - Invio a burst

16 richieste GET inviate ogni 15 secondi (simula arrivo a gruppi)

## Risultati

- Throughput: 1.04 batch/s
- Latenza media: 8.28 secondi
- Latenza minima: 1 secondo e 120 millisecondi
- Latenza massima: 1 minuto e 16 secondi
- Batch mancanti: 32 su 3600

Osservazioni:

Buon compromesso tra carico e latenza

Adatto a scenari realistici con arrivo a ondate

# Confronto tra Scenari

<b>Scenario</b>	<b>Throughput</b>	<b>Lat. Media</b>	<b>Lat. Max</b>	<b>Batch Mancanti</b>
Continuo	2.01	14m47s	27m57s	32
Regolare (1s)	0.99	4.36s	40.6s	32
Burst (16/15s)	1.04	8.28s	1m16s	32

Tabella I

CONFRONTO TRA I TRE SCENARI DI INVIO BATCH

# Perché 32 batch mancanti...

```
keyed_windowed_ds = filtered_ds  
    .key_by(lambda row: row.tile_id, key_type=Types.INT())  
    .count_window(3, 1)
```

- La finestra ha dimensione 3 e passo 1
- Serve almeno 1 finestra completa per generare output
- I primi 2 elementi (per ogni tile\_id) non completano la finestra

## Nessuna elaborazione per layer 1 e 2

$$2 \text{ layer} \times 16 \text{ tile} = 32 \text{ batch}$$



# **RIELABORAZIONE DEL CODICE DI PYFLINK IN JAVA**

# PERCHÉ

Parlando con i colleghi ho visto come il throughput che ottenevo sull'esecuzione cumulativa era fin troppo basso.

Per questa ragione ho deciso di provare a reimplementare quantomeno il primo algoritmo sperimento con PyFlink e vedere come si comportasse.

Sottolineo che questo non è stato il focus principale del progetto, ma principalmente una curiosità sorta durante la fase di testing.

# Java 1 - Invio continuo

Il middleware invia richieste GET senza pausa su un totale di 500 jobs

## Risultati (500)

- Throughput: 2.09 batch/s
- Latenza media: 1 minuti e 46 secondi
- Latenza minima: 557 millisecondi
- Latenza massima: 3 minuti 46 secondi
- Batch mancanti: 32 su 500

## Risultati - PyFlink (3600)

- Throughput: 2.01 batch/s
- Latenza media: 14 minuti e 47 secondi
- Latenza minima: 12 secondi
- Latenza massima: 27 minuti e 57 secondi
- Batch mancanti: 32 su 3600

# Java 2 - Invio regolare

Il middleware invia richieste GET con un secondo di pausa su un totale di 500 jobs

## Risultati (500)

- Throughput: 1.052 batch/s
- Latenza media: 576 millisecondi
- Latenza minima: 399 millisecondi
- Latenza massima: 3 secondi 134 millisecondi
- Batch mancanti: 32 su 500

## Risultati - PyFlink (3600)

- Throughput: 0.99 batch/s
- Latenza media: 4.36 secondi
- Latenza minima: 760 millisecondi
- Latenza massima: 40.6 secondi
- Batch mancanti: 32 su 3600

# Java 3 - Invio a burst

Il middleware invia richieste GET inviate ogni 15 secondi (simula arrivo a gruppi) su un totale di 500 jobs

## Risultati (500)

- Throughput: 1.11 batch/s
- Latenza media: 3 secondi e 472 millisecondi
- Latenza minima: 436 millisecondi
- Latenza massima: 9 secondi 324 millisecondi
- Batch mancanti: 32 su 500

## Risultati - PyFlink (3600)

- Throughput: 1.04 batch/s
- Latenza media: 8.28 secondi
- Latenza minima: 1 secondo e 120 millisecondi
- Latenza massima: 1 minuto e 16 secondi
- Batch mancanti: 32 su 3600