Real-Time Defect Detection in L-PBF Manufacturing Using Stream Processing with Apache Flink

Matteo Basili
Department of Civil Engineering and Computer Engineering
University of Rome Tor Vergata
Rome, Italy
matteo.basili@students.uniroma2.eu

Adriano Trani
Department of Civil Engineering and Computer Engineering
University of Rome Tor Vergata
Rome, Italy
adriano.trani@students.uniroma2.eu

***Abstract*—We present a real-time streaming pipeline for thermal anomaly detection in Laser Powder Bed Fusion (L-PBF) manufacturing, using Apache Flink. The system ingests thermal image tiles from a REST-based local server into Kafka, and processes them through a four-stage Flink pipeline: threshold filtering, sliding window analysis, outlier detection, and DBSCAN clustering. Intermediate and final results are published to separate Kafka topics, with the final output returned to the LOCAL-CHALLENGER for benchmarking. Performance was evaluated through log-based analysis of latency (average, max, percentiles) and throughput across multiple batch sizes. Results confirm the pipeline's efficiency and its suitability for real-time defect monitoring in industrial L-PBF scenarios.**

Keywords*—Real-time stream processing, Apache Flink, Additive manufacturing, L-PBF, Thermal image analysis, Anomaly detection, Outlier detection, Clustering, Kafka, DBSCAN, Industrial monitoring, Online analytics

I. Introduction

Additive manufacturing (AM) processes, such as **Laser Powder Bed Fusion (L-PBF)**, rely on precise thermal control to ensure structural integrity and part quality. Real-time monitoring of the temperature distribution across the powder bed is crucial to detect anomalies that may indicate defects during the build process. The increasing availability of high-resolution sensor data opens up opportunities for online analytics; however, it also poses significant challenges in terms of scalability, responsiveness, and data throughput.

This work addresses these challenges by implementing a real-time streaming architecture for thermal anomaly detection and defect analysis in L-PBF using **Apache Flink [1] with Python (PyFlink) [2]**, a distributed stream processing framework. The system is designed in response to the **ACM DEBS 2025 Grand Challenge**, which provides a realistic dataset of thermal images (OT data) acquired layer-by-layer during the L-PBF process. Each image is delivered through a REST-based server (the LOCAL-CHALLENGER), simulating a live production scenario.

Our solution ingests the tile stream through a **Kafka-based messaging pipeline**: a client component requests images from the LOCAL-CHALLENGER and publishes them to an input Kafka topic. Apache Flink then processes this stream through a four-stage pipeline:

1. **Threshold-based filtering**, to identify saturated or invalid points;
2. **Sliding window processing** across multiple layers to capture temporal trends;
3. **Local outlier detection** based on neighborhood temperature deviations;
4. **Clustering** of outliers using the DBSCAN algorithm.

The output of each query is written to dedicated Kafka topics. The result of the final stage is additionally forwarded to the LOCAL-CHALLENGER to compute official performance metrics.

To assess system behavior under load, we conducted an experimental evaluation for **each individual query** in the processing pipeline. Performance was measured separately in terms of **latency** and **throughput**, based on real execution logs captured during controlled runs.

II. Background and Theory

A. Laser Powder Bed Fusion and In-situ Thermal Monitoring

Laser Powder Bed Fusion (L-PBF) [3] is an additive manufacturing (AM) technique widely used for fabricating high-precision metal components. The process involves spreading thin layers of metallic powder and selectively fusing regions using a high-power laser, guided by a digital model (e.g., CAD file). L-PBF enables complex geometries and material efficiency, offering significant advantages over traditional subtractive manufacturing methods.

Despite its strengths, L-PBF is susceptible to process-induced defects such as porosity, cracking, and lack of fusion. These issues often stem from thermal instabilities, material impurities, or calibration errors. **Porosity**, in particular, can critically weaken mechanical performance, making real-time quality monitoring essential.

Traditionally, quality control in L-PBF is performed post-process through techniques like X-ray CT or destructive testing. However, these approaches are time-consuming and wasteful when defective parts are discovered after production. To mitigate this, **in-situ thermal monitoring** using **Optical Tomography (OT)** is employed. OT captures thermal images of the powder bed for each layer, revealing the temperature distribution across the surface. Anomalies in these distributions can be indicative of process deviations or incipient defects. Real-time analysis of OT data allows early detection and potential intervention, improving reliability and reducing scrap rates.

B. *The DEBS 2025 Challenge and Dataset Specification*

The ACM DEBS 2025 Grand Challenge [3], [4] provides a realistic benchmark scenario for stream processing in additive manufacturing. The challenge is based on a synthetic dataset simulating OT image acquisition during an L-PBF process. The dataset is embedded within a REST server, called the LOCAL-CHALLENGER, which implements an OpenAPI-compliant interface [5] for retrieving data and submitting results.

Each OT image represents the temperature distribution on the powder bed at a specific layer of the print. Every image is subdivided into smaller units called **tiles**. These tiles are streamed individually by the server, simulating a real-time acquisition scenario. Each stream element includes:

- *seq_id*: a unique sequence number;
- *print_id*: an identifier for the printed object;
- *tile_id*: the tile's position within the layer;
- *layer*: the z-coordinate (layer number);
- *tiff*: 16-bit binary image data in TIFF format.

These high-resolution images capture pixel-level temperatures on a scale from 0 to 65,536 (in grayscale). The subdivision into tiles improves parallelism and reflects actual industrial monitoring setups where different parts of the build area may be covered by different sensors or perspectives.

The LOCAL-CHALLENGER is designed for local benchmarking purposes and exposes a REST API to simulate online interaction. A reference Python implementation is provided to illustrate query semantics and facilitate output validation.

III. Solution Design

This section describes the architectural design of the real-time processing application developed for detecting defects during the L-PBF process using Apache Flink. The solution is designed to handle continuous streaming data from the LOCAL-CHALLENGER REST server, process the OT images in a multi-stage pipeline, and produce query results.

A. *System Architecture*

The architecture consists of three main components (Fig. 1):

- **Client Module:** Responsible for interacting with the LOCAL-CHALLENGER REST API to fetch OT image tiles. The client establishes a session, initiates the benchmark, and continuously requests image tiles (1). Retrieved tiles (2) are published into a Kafka [6] topic (3), which acts as the streaming data source for the processing pipeline.

- **Stream Processing Pipeline:** Implemented in Apache Flink, this module consumes the tile stream from Kafka (4) and performs the multi-stage analysis as defined by the queries:

  - **Stage 1: Threshold Analysis (Q1)** — Each tile is analyzed to identify points with temperature values below or above predefined thresholds. Saturated points above the upper threshold are counted, while points below the lower threshold are excluded from subsequent stages. Also the saturated points are excluded from subsequent stages.

  - **Stage 2: Sliding Window Aggregation (Q2)** — A sliding window of the latest three layers per tile is maintained to enable temporal analysis.

  - **Stage 3: Outlier Detection (Q2)** — For each tile and sliding window, local temperature deviations are computed by comparing neighborhoods defined by Manhattan distance. Points exceeding the deviation threshold are flagged as outliers.

  - **Stage 4: Outlier Clustering (Q3)** — The outliers identified in Stage 3 are clustered using the DBSCAN [7] algorithm based on Euclidean distance, and cluster centroids and sizes are computed.

- **Result Handling and Output:** Results of queries are written to separate Kafka output topics (5) for further inspection or offline analysis. The output of query Q3, which includes cluster centroids and saturated point

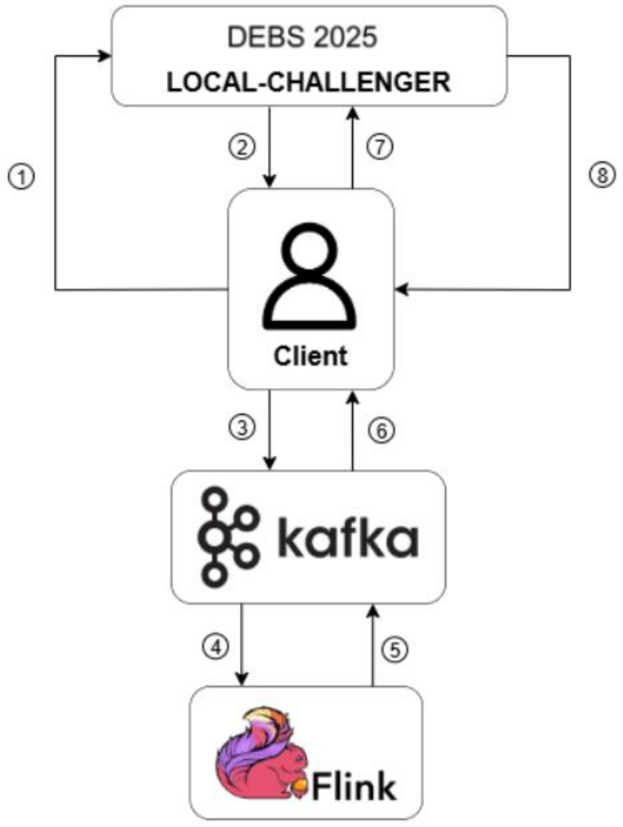


Fig. 1. Architecture of the system. *The arrows with numbers indicate the ordered data flow.*

counts, is sent back to the LOCAL-CHALLENGER server to complete the benchmark (6)-(7) and obtain performance metrics (8).

B. *Performance Measurement Strategy*

To evaluate the system's responsiveness and scalability, a performance evaluation strategy was devised focusing on the individual processing stages corresponding to each query. For each query (Q1, Q2, Q3), **latency** was measured as the time elapsed between the moment a tile enters that specific stage and the moment its corresponding result is emitted. This allows for a fine-grained analysis of computational cost per query, independently from other stages.

In parallel, **throughput** was measured as the rate of processed tiles per second, based on real-time logs generated during controlled executions. This metric provides insights into the system's capacity to sustain continuous ingestion without degradation in responsiveness.

By analyzing both latency and throughput at the query level, the system's performance can be characterized in detail, highlighting stage-specific bottlenecks and guiding future scalability improvements.

C. *Auxiliary Components*

In addition to the main streaming pipeline, the system includes two auxiliary services that support deployment, integration, and observability:

Kafka Topic Initializer

At system startup, a dedicated **Kafka Topic Initializer** service is responsible for creating all Kafka topics required by the application. This includes:

- The **input topic**, which receives the OT tiles from the client;
- The **output topics** for Query 1, Query 2, and Query 3.

The service ensures that all topics exist with the correct configuration (e.g., number of partitions, replication factor) before other components begin operation. This avoids race conditions and guarantees smooth deployment in containerized or automated environments.

CSV Writer Consumer

The **CSV Writer Consumer** is a standalone service that listens to the Kafka output topics corresponding to each query and writes the received results to local CSV files. This serves multiple purposes such as:

- **Debugging and Inspection**: Developers can manually inspect the outputs and correlate them with the expected results.
- **Offline Validation**: Persisted outputs allow post-execution comparison against the reference implementation.

Each query's results are logged in a separate file using a custom schema aligned with its output format.

IV. Solution Details

All the project's code is located in a Github repository [8]. All the folders, files and code calls mentioned in this section (and also in the next ones) are located within this repository.

A. *Client*

The client, implemented in Python, acts as the bridge between the LOCAL-CHALLENGER platform and the Kafka pipeline. It initializes the benchmark via REST API, then fetches image batches encoded with MessagePack, base64-encodes TIFF data for JSON compatibility, and publishes them to the Kafka input topic (*tiff-batches*).

Simultaneously, a consumer listens on the Kafka output topic (*l-pbf-output*), parses CSV results containing metadata and cluster centroids, repacks them with MessagePack, and posts them back to LOCAL-CHALLENGER for validation.

Producer and consumer run in parallel threads, synchronized via thread-safe counters to ensure all batches are processed before shutdown. The client includes retry loops to handle Kafka and API readiness, enabling robust and efficient asynchronous data flow management within the benchmarking architecture.

Related script: *client/l-pbf_client.py*.

B. *Pipeline*

The streaming data pipeline is implemented using **PyFlink** to process image batches from Kafka topics, applying three main queries (Q1, Q2, Q3) sequentially with stateful transformations and windowing.

Q1: Saturation Analysis

The first stage, Q1, starts by ingesting raw JSON strings from the Kafka topic *tiff-batches*. Each message contains metadata (such as *batch_id*, *print_id*, *tile_id*, and *layer*) along with a base64-encoded TIFF image. The core function responsible for handling this input is *process_batch*.

The process begins by parsing and validating the incoming JSON. If the structure is valid, the TIFF image is decoded from base64, loaded using the Python Imaging Library (PIL), and converted into a NumPy array. This allows the system to perform pixel-level analysis.

One of the key computations at this stage is saturation detection. Using a predefined threshold, the function counts the number of pixels in the image that exceed this limit, indicating potential overexposure.

From the processed data, two outputs are generated:

- A Flink *Row* containing batch metadata along with the saturation count, which is serialized to CSV and sent to the Kafka topic *q1-output*.
- A filtered dictionary that retains essential metadata, the original TIFF (as a base64 string), and the saturation count, all serialized as strings. This dictionary is forwarded downstream to feed into Q2.

Related script: *flink/jobs/queries/q1.py).*

Q2: Stateful Sliding Window Outlier Detection

Q2 takes the filtered JSON output from Q1 and performs a more advanced analysis to detect spatial outliers over time.

Each incoming JSON message is parsed and passed through a function called *decode_and_prepare*, which extracts the image, reconstructs it into a NumPy array, and attaches relevant metadata.

The core logic of Q2 lies in a Flink *KeyedProcessFunction*, where incoming data is grouped by *print_id* and *tile_id*. This allows the system to maintain a separate sliding window for each tile across time. Specifically, it keeps a stateful list of the most recent three batches for each key.

When fewer than three batches are available, the function simply emits placeholders to indicate insufficient data. Once the sliding window is full, it constructs a 3D volume from the batch stack and analyzes each pixel's deviation by comparing its value to surrounding neighbors (based on Manhattan distance).

Outlier pixels are identified where the deviation exceeds a predefined threshold. The system then selects the five most extreme outliers (by deviation magnitude), and produces two outputs:

- A Flink *Row* with batch metadata and the top 5 outlier points, which is serialized to CSV and sent to the Kafka topic *q2-output*.
- A JSON-encoded list of all detected outlier coordinates and their corresponding deviations, which serves as input for Q3.

Related script: *flink/jobs/queries/q2.py).*

Q3: Clustering

Q3 focuses on spatial clustering of the outliers detected in Q2.

It ingests the JSON output from Q2 and parses it to extract all outlier points. Using the DBSCAN algorithm, it identifies spatial clusters based on proximity parameters (*DBSCAN_EPS* and *DBSCAN_MIN_SAMPLES*).

For each non-noise cluster, the system computes a centroid (average coordinate) and cluster size. These centroids are then serialized into a compact JSON format.

The final output from Q3 is a Flink *Row* containing batch metadata, saturation count (carried forward), and the JSON-encoded cluster summary. This Row is serialized to CSV and written to the Kafka topic *l-pbf-output,* concluding the pipeline.

Related script: *flink/jobs/queries/q3.py).*

Query Coordination and Stream Flow

The entire streaming pipeline is orchestrated inside the **L-PBF Job** (*flink/jobs/l-pbf_job.py*), which manages how data flows through the three stages.

The Flink environment is set up in streaming mode with RocksDB for state management and periodic checkpointing for fault tolerance.

The pipeline starts by consuming raw JSON messages from the Kafka topic *tiff-batches*.

Each query writes its output independently to Kafka, allowing for decoupled processing, monitoring, and scaling. The pipeline is launched with *env.execute()* and runs continuously.

C. *Kafka Topic Initialization*

The Kafka topi initializer uses the ***kafka-python*** library's *KafkaAdminClient* to programmatically create Kafka topics required by the pipeline. Topics are defined in a configuration list specifying their name, partition count, and replication factor.

A retry loop ensures connection to Kafka brokers before proceeding, handling *NoBrokersAvailable* exceptions by waiting and retrying every two seconds. This approach guarantees the broker is ready, avoiding startup errors.

Topics are created via *create_topics* with a list of *NewTopic* objects. If a topic already exists, the script catches the *TopicAlreadyExistsError* and logs a warning without failing, ensuring idempotent execution.

Finally, the admin client is properly closed.

Related script: *kafka/init/create_topics.py*.

D. *CSV Writer*

The CSV writer consumes messages from multiple Kafka topics concurrently using the *KafkaConsumer* from *kafka-python*, each running in its own thread. It subscribes to specified topics and continuously polls for new messages.

Each message is expected to be a CSV-formatted string, which the script parses and writes as rows into corresponding CSV files. If the output file is empty, a header row is written first based on the provided field names.

Robustness is enhanced by logging all received messages, warnings for field count mismatches, and errors during consumption. Finally, the Kafka consumers close properly when the script terminates.

Related script: *csv-writer/ kafka_to_csv_stream_writer.py.*

E. *Query Performance Measurement*

To assess the runtime behavior of the streaming pipeline, we implemented a lightweight performance logging strategy combined with offline analysis in Python. Each query (Q1–Q3) emits custom log lines annotated with timestamps and latency measurements for each processed batch. These entries follow a structured format:

```
METRICS|Q1|batch=123|latency_ms=17.42
```

This format ensures the data is easy to parse and uniformly structured across the pipeline. Logging statements are placed around critical regions of each query function, allowing us to capture the full query end-to-end latency per batch.

The analysis is handled by the *scripts/analyze_latency.py* script, which performs five key steps:

1. **Log Parsing**:

The script reads the raw Flink logs and extracts structured entries using regular expressions. It collects four fields:

timestamp, query identifier (Q1, Q2, or Q3), batch ID, and measured latency in milliseconds.

2. **Aggregation**:

Since a batch may trigger multiple metrics entries within the same query (e.g., in Q2), the script groups log lines by *(query, batch_id)* and computes the total latency for that batch. It also records the first observed timestamp for throughput estimation.

3. **Latency Statistics**:

The script computes detailed latency metrics for each query:

- **Mean** and **maximum** latency
- **Percentiles**: 50th (median), 90th, and 99th percentiles
- **Count** of batches processed

These statistics provide insight into both the average-case and worst-case performance under load.

4. **Throughput Estimation**:

For each query, throughput is computed as the number of batches processed divided by the wall-clock time between the first and last logged batch. The result is expressed in batches per second.

5. **Export to Excel**:

The final results are written to an Excel file, with two sheets:

- *Latency*: Contains per-query latency statistics
- *Throughput*: Lists batch throughput for each query

This lightweight measurement framework allows for reproducible benchmarking of the streaming job. It is fully decoupled from Flink internals and does not rely on external monitoring tools, making it portable and easy to integrate into CI pipelines or large-scale batch evaluations.

V. Deployment

The entire system is containerized using Docker and orchestrated via *docker-compose.yml* [9], [10], which defines all required services for ingestion, stream processing, and output export. This setup allows for consistent and reproducible deployment across different environments.

Core Containers:

- **LOCAL-CHALLENGER** – Exposes port *8866* and mounts local data for input.
- **Apache Kafka Cluster** – Composed of:
    - ***zookeeper*** (Confluent 7.4.0)
    - Two Kafka brokers (***kafka1*** on port *9092*, ***kafka2*** on port *9093*)
- **Apache Flink Cluster** – Deployed with:
    - One ***jobmanager*** container with dashboard exposed on port *8081*
    - Two ***taskmanager*** containers sharing a common volume for checkpointing
- **Topic Initializer** – Responsible for programmatically creating all Kafka topics at system startup.
- **L-PBF Client** – The component that interacts with the LOCAL-CHALLENGER to simulate streaming ingestion and publish tiles to Kafka.
- **CSV Writer** – The consumer service that subscribes to the output topics of the queries and writes results to local CSV files for inspection, validation, and export.

Networking & Storage

- All containers communicate over a shared Docker bridge network: *flinknet*
- Volumes:
    - *./local-challenger/data* – Used by the LOCAL-CHALLENGER to expose OT images
    - *./flink/checkpoints* – Mounted by Flink for state checkpointing
    - *./Results/csv* – Shared output directory used by the CSV Writer

A. System Startup and Management

To launch the full system, the following command is used:

docker compose up --build –d

- **Flink Dashboard** – http://localhost:8081
- **LOCAL-CHALLENGER** – http://localhost:8866

B. Setup and Execution

Execution of the complete L-PBF processing pipeline is automated via the *run_all.py* script, located in the *scripts/* directory. This script must be executed from the project's root.

For quick testing or partial runs, the script supports an optional *--limit N* argument to restrict the number of image batches streamed through the system.

Further configuration, requirements and usage details are provided in the Github repository README.

VI. Results

This section presents the results produced by the system, structured in two parts: (A) the output CSV files generated for each query and (B) the quantitative performance metrics collected during experimental runs.

A. Query Output Files

Each query in the pipeline produces structured outputs that are exported to separate CSV files by the dedicated Kafka consumer. The files follow a custom schema tailored to the logic of each processing stage, as described below.

Query 1 (Threshold Filtering)

The Q1 output file reports saturated pixel counts per tile, based on temperature thresholds. The CSV format is:

seq_id, print_id, tile_id, saturated

- *seq_id*: Sequence number of the processed element
- *print_id*: Identifier of the printed object
- *tile_id*: Identifier of the tile
- *saturated*: Number of pixels with temperature above 65,000 (considered saturated)

These values serve as early indicators of potential thermal anomalies but are not propagated to the LOCAL-CHALLENGER.

Query 2 (Sliding Window + Outlier Detection)

Q2 computes local temperature deviations over a 3-layer sliding window and ranks the top 5 outlier points per tile. The CSV format is:

seq_id, print_id, tile_id, p_1, dp_1, p_2, dp_2, p_3, dp_3, p_4, dp_4, p_5, dp_5

- *p_X*: Position (coordinate) of the X-th most anomalous point
- *dp_X*: Local temperature deviation of point p_X

This output is not sent to the LOCAL-CHALLENGER.

Query 3 (Clustering of Outliers)

Q3 clusters outlier points using the DBSCAN algorithm and includes both the result of clustering and the saturated pixel count from Q1. The CSV format is:

seq_id, print_id, tile_id, saturated, centroids

- *saturated*: Number of saturated points in the tile
- *centroids*: A list of tuples containing cluster centroids (x, y) and cluster size

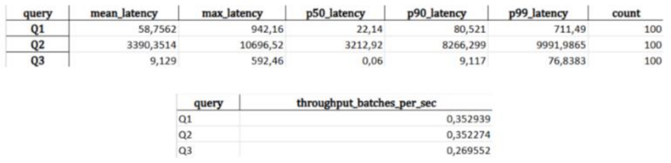
This is the only output stream transmitted back to the LOCAL-CHALLENGER server for benchmarking and evaluation purposes.

➞ **Example excerpts** of each CSV format can be found in the GitHub repository, under the *Results/csv* folder.

B. *Performance Metrics*

To evaluate the performance of the pipeline under varying load conditions, we conducted three separate runs of the full system using the *--limit* option in the pipeline launcher (*run_all.py*). Each run processed a different number of image batches: 100, 200, and 300 respectively.

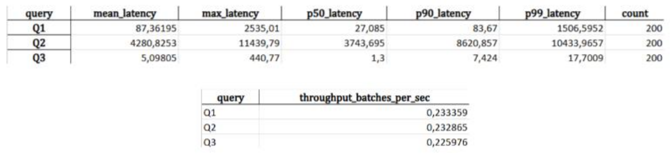
The following figures summarize the performance metrics gathered at the end of each run:



query	mean_latency	max_latency	p50_latency	p90_latency	p99_latency	count
Q1	58,7562	942,16	22,14	80,521	711,49	100
Q2	3390,3514	10696,52	3212,92	8266,299	9991,9865	100
Q3	9,129	592,46	0,06	9,117	76,8383	100

query	throughput_batches_per_sec
Q1	0,352939
Q2	0,352274
Q3	0,269552

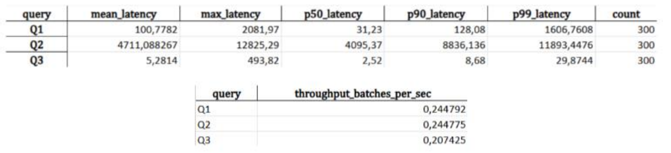
Fig. 2. Performance report for the 100-batch execution.



query	mean_latency	max_latency	p50_latency	p90_latency	p99_latency	count
Q1	87,36195	2535,01	27,085	83,67	1506,5952	200
Q2	4280,8253	11439,79	3743,695	8620,857	10433,9657	200
Q3	5,09805	440,77	1,3	7,424	17,7009	200

query	throughput_batches_per_sec
Q1	0,233359
Q2	0,232865
Q3	0,225976

Fig. 3. Performance report for the 200-batch execution.



query	mean_latency	max_latency	p50_latency	p90_latency	p99_latency	count
Q1	100,7782	2081,97	31,23	128,08	1606,7608	300
Q2	4711,088267	12825,29	4095,37	8836,136	11893,4476	300
Q3	5,2814	493,82	2,52	8,68	29,8744	300

query	throughput_batches_per_sec
Q1	0,244792
Q2	0,244775
Q3	0,207425

Fig. 4. Performance report for the 300-batch execution.

VII. DISCUSSION

This section evaluates the performance results collected over the three execution runs (Fig. 2, 3, 4). The analysis focuses on latency trends, scalability, and throughput for each query (Q1–Q3), drawing conclusions on computational efficiency and stability under increasing load

A. *Latency Analysis*

Q1

- **Mean latency** increases from ~58 ms (100 batches) to ~100 ms (300 batches), indicating moderate sensitivity to input volume.
- **Maximum latency** shows significant fluctuations: peaking at ~2535 ms (200 batches), likely due to outliers or temporary backpressure.
- **P50 (median)** latency remains low across runs (22–31 ms), confirming that the majority of batches are processed quickly.
- The tail latencies (p99) rise considerably with batch size (711 ms → 1606 ms), highlighting occasional slowdowns under higher load.
- **Conclusion**: Q1 is lightweight and efficient for most batches, but its tail performance may degrade under stress.

Q2

- This is the **most computationally expensive** query: mean latency grows from ~3390 ms (100) to ~4711 ms (300).
- The latency distribution is highly skewed, with **p99 values near 10–12 seconds** in all runs, and a **max latency of over 12.8 seconds**.
- The **median (p50)** is consistently high (~3–4 seconds), indicating sustained computational cost even for "typical" batches.
- **Conclusion**: Q2 introduces heavy stateful operations (e.g., windowed aggregations) and suffers from high and persistent latencies. Optimization or parallelism may be necessary for real-time scenarios.

Q3

- Q3 shows the **best overall performance**, with mean latency between **~5 ms (200–300 batches)** and a remarkably low **median** (<3 ms).
- However, a few high-latency outliers push the **max latency up to ~493 ms** (300 batches), seen also in the **p99 percentile** (~30 ms).
- Compared to Q2, Q3 is an order of magnitude faster across all metrics.
- **Conclusion**: Q3 is consistently efficient and scalable, making it suitable even in streaming systems with tight latency constraints.

B. *Throughput Analysis*

- **Q1 and Q2** exhibit similar throughput values in each run (e.g., ~0.35 batch/s for 100 batches, ~0.24 for 300).
- Despite its high latency, Q2 matches Q1 in throughput, possibly due to asynchronous operations or overlapping stages in Flink's pipeline.
- **Q3**, while faster per batch, exhibits slightly **lower throughput** (down to **0.207 batch/s** in the 300 batch run). This could be due to backpressure inherited from Q2's processing chain.
- Overall, throughput **declines slightly** as the input size grows, consistent with expectations in a fixed-resource, non-tuned streaming environment.

C. *Scalability and Observations*

- Q1 scales relatively well but is affected by sporadic slowdowns under load.
- Q2 remains the **most sensitive to scale**, both in terms of average and tail latency.
- Q3 remains performant across all runs, although it slightly suffers from upstream delays.
- The use of **Flink's default configuration** (e.g., no tuning, limited parallelism, RocksDB state backend) places a realistic lower bound on performance. Given these conditions, observed results highlight the robustness of the pipeline, even for Q2.

VIII. CONCLUSIONS

This project presented a baseline implementation of a streaming analytics pipeline for thermal image data, using Apache Flink with its default configuration. Specifically, the pipeline was executed with a **parallelism level set to 1**, meaning that each operator was executed as a single task. Moreover, **Kafka topics were configured with a single partition**, ensuring that only one Flink subtask could consume data per topic. While this setup limits the potential for parallel execution, it allowed for a clear and controlled evaluation of the system's core functionalities and performance behavior.

Despite these constraints, the project successfully implemented all required features, including ingestion, preprocessing, stateful analysis (sliding windows), and clustering-based summarization. The performance analysis revealed insightful trends across all queries (Q1–Q3), offering a strong reference point for understanding latency and throughput characteristics under increasing data volumes.

This work serves as a solid foundation for future developments. In particular, several optional and advanced optimization tasks can be explored:

- **Intra-layer tile parallelism**, where tiles belonging to the same print layer (i.e., *tile_id* per *print_id*) can be processed independently to reduce latency;
- **Application-level optimization** for Query 2, where the local temperature deviation calculation lends itself to a **discrete convolutional implementation [11]**. This would allow for efficient bulk computation of deviations using a custom 3D kernel over the window tensor, reducing the cost of nested iterations;
- **Platform benchmarking**, comparing the current Flink implementation with **Kafka Streams** or **Spark Streaming**, using the same dataset and infrastructure to assess performance differences in terms of latency and throughput.

These directions provide meaningful opportunities to enhance both the scalability and responsiveness of the system, paving the way for a production-grade version of this analytics pipeline.

REFERENCES

[1] Apache Flink, "Apache Flink®" Available online: https://flink.apache.org/

[2] Read the Docs, "PyFlink Docs." Available online: https://pyflink.readthedocs.io/en/main/index.html

[3] DEBS 2025. Call for Grand Challenge Solutions. https://2025.debs.org/call-for-grand-challenge-solutions/, 2025.

[4] L. De Martini, J. Tahir, A. Margara, C. Doblander, S. Frischbier, R. Mayer, and H.-A. Jacobsen. The debs 2025 grand challenge: Real-time monitoring of defects in laser powder bed fusion (l-pbf) manufacturing. In Proc. of 19th ACM Int'l Conf. on Distributed and Event-Based Systems, DEBS '25, page 223–228, 2025.

[5] openAPI. The OpenAPI Initiative. https://www.openapis.org/, 2025.

[6] Apache Kafka, "Apache Kafka." Available online: https://kafka.apache.org/

[7] Wikipedia, "DBSCAN." Available online: https://it.wikipedia.org/wiki/DBSCAN

[8] MatteoBasili, "sabd-progetto2-2024_25" [GitHub Repository], 2025. Available online: https://github.com/MatteoBasili/sabd-progetto2-2024_25

[9] Docker, “Dockerfile reference,” Docker Documentation. Available online: https://docs.docker.com/reference/dockerfile/

[10] Docker Compose, "Documentation." Available online: https://docs.docker.com/compose/

[11] Wikipedia, "Convolution – Discrete Convolution." Available online: https://en.wikipedia.org/wiki/Convolution#Discrete_convolution