Termal Defect Detection

Systems and Architectures for Big Data, A.Y. 2024-2025

Alessandro Cortese
Computer Engineering
University of Rome
Tor Vergata
Roma, Roma
alessandro.cortese@students.uniroma2.eu

Chiara Iurato
Computer Engineering
University of Rome
Tor Vergata
Vittoria, Ragusa
chiara.iurato@students.uniroma2.eu

Luca Martorelli
Computer Engineering
University of Rome
Tor Vergata
Firenze, Firenze
luca.martorelli@students.uniroma2.eu

***Abstract*—This project develops a real-time monitoring system for defects during Laser Powder Bed Fusion (L-PBF) additive manufacturing, leveraging stream processing frameworks such as Apache Kafka and Apache Flink.**

The system performs on-the-fly defect detection by analyzing local temperature deviations and applying clustering techniques. It demonstrates the integration of big data tools to support environmental data analysis and performance benchmarking in a fully containerized architecture.

I. Introduction

Laser Powder Bed Fusion (L-PBF) is a 3D printing technique for metals that fuses layers of powder with a laser, allows for greater design flexibility and reduces material waste compared to traditional subtractive methods that obtain the object by removing material from a solid block. However, the process can generate critical defects, such as porosity, which can compromise the quality of the parts produced. Traditional inspection is only carried out at the end of production, which wastes time, energy and resources as defective parts have to be discarded.

To reduce inefficiencies, real-time monitoring techniques using optical tomography (OT) are used. This technique allows thermal images to be acquired layer by layer, showing the temperature distribution. Correct analysis of the images makes it possible to detect irregularities and defects at an early stage, allowing immediate action to be taken to ensure product quality.

Each data batch consists of a set of optical tomography (OT) images, delivered as a stream in which each image corresponds to a single printing layer. Each OT image contains, for each point `P=(x, y)`, the temperature `T(P)` at that point, measured layer by layer `(z)`. The images are divided into multiple tiles, or blocks, representing portions of the layer. Each tile is encoded in TIFF format, providing high-resolution temperature data in 16-bit precision. Each stream element contains the following fields:

- `seq_id`: a unique sequence number for the input element;
- `print_id`: the identifier of the object currently being printed;
- `tile_id`: the identifier of the tile within the current layer;
- `layer`: the z-coordinate (i.e., the layer number);
- `tiff`: the raw binary data representing the tile image.

This structure allows for real-time analysis of the printing process, as we receive temperature data layer-by-layer, tile-by-tile.

Our pipeline uses this input to detect abnormal thermal behavior and infer the presence of possible defects during the build.

II. System Architecture - Flink Version

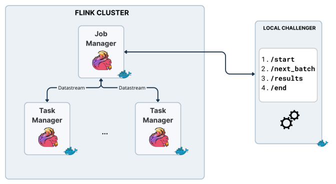

Fig. 1: System Architecture - Flink Version

The diagram above illustrates the system architecture, highlighting the interaction between the JobManager and the `Local Challenger`.

- **Local Challenger**[1]: is a REST server that hosts the OT dataset and manages benchmark execution. When active, it exposes the following API endpoints:
  - `/create`: Initializes a new benchmark
  - `/start`: Begins metric collection for the benchmark;
  - `/next_batch`: Retrieves the next batch of OT image data for processing;
  - `/result`: Submits query results back to the challenger;
  - `/end`: Terminates the benchmark and finalizes evaluation;
- **Apache Flink**[2]: is the main processing engine of the project. It is configured with a cluster of one *JobManager* and eight *TaskManager*. Horizontal scaling is programmatically controlled through Flink's parallelism

API - the `setParallelism()` method dynamically allocates processing resources by specifying the number of concurrent TaskManager instances.

III. Pipeline - Flink Version

Once the containers have been launched, we trigger the execution of our Flink job. This initializes the execution environment within Flink, preparing it to receive and process data.

The system uses a custom Flink `SourceFunction` to fetch processing batches via HTTP from the Local Challenger service. It starts by initializing a new benchmark session, then begins streaming data by repeatedly requesting the next available batch.

Each incoming batch is mapped to a `Batch` object within the internal data model and forwarded into the Flink pipeline for processing. This process loops continuously until the Challenger responds with a `404 Not Found` at the `/next_batch` endpoint, signaling that no more data is available.

Within the pipeline, each batch is processed sequentially by the three queries, where each stage takes as input the results produced by the previous one.

A. Processing: Queries

Query 1. *Within each tile, detect all points that are below a lower threshold value of 5000 or surpass an upper threshold value of 65000. Points below the lower threshold represent empty areas that should be excluded from subsequent queries. Points above the upper threshold are considered saturated and may indicate defects. Defects should be reported, but not included in the analysis of subsequent queries.*

The class `Q1SaturationMapFunction` implements a transformation within the Flink pipeline, used for Query 1 of the project, which aims to detect thermal saturations in optical data.

This function receives as input a Batch object, which represents a portion of a thermal image (a tile) encoded in TIFF format. The first operation performed is the decoding of the image: through the `decodeTIFF()` method, the binary data is converted into a matrix of integers (pixels), where each value represents the temperature recorded at a specific point on the tile. Once the matrix of pixels is obtained, the function performs a full scan to count how many points exceed a thermal saturation threshold, set at 65000.

This value is considered critical and the presence of numerous pixels exceeding it may indicate a potential defect in the print. The result of this count is saved in the appropriate attribute of the Batch object.

Query 2. *For each tile, maintain a sliding window of the last 3 layers, enabling the analysis of temperature evolution between consecutive layers. For each point P in the most recent layer of the window on the tiles, calculate the local temperature deviation, defined as the absolute difference between:*

- *The average temperature of the nearest neighbors of P, i.e., all points with a Manhattan distance $0 < d \leq 2$ from P, considering the 3 layers*
- *The average temperature of the external neighbors of P, i.e., all points with a Manhattan distance $2 < d \leq 4$ from P, considering the 3 layers.*

Classify the point as an outlier if its local temperature deviation exceeds the threshold of 6000. The query output must report, for each tile and window, the ranking of the 5 points with the highest deviation.

The second query is implemented through the `Q2OutlierDetection` class, which extends Flink's `KeyedProcessFunction`. Each key corresponds to a specific tile over time, identified by the combination of `print_id` and `tile_id`. This setup allows the operator to maintain a separate sliding window for each tile, enabling localized temporal outlier detection

A central aspect of this implementation is the handling of the time window, which does not rely on Flink's native windowing system, but is done manually through a `ListState` structure. This state maintains, for each key, a moving window of three consecutive images (corresponding to three successive layers) of the same tile. With each new incoming element, the image is added to the window, and eventually the oldest layer is removed to keep the size to three.

Once the window has accumulated three images, outlier analysis logic is activated. The goal of this analysis is to identify the points (`pixels`) in the most recent image that have an anomalous thermal deviation from the surrounding context. For each point (x, y) in the current image, the calculation is excluded if the temperature value is too low (lower than `EMPTY_THRESHOLD`) or too high (higher than `SATURATION_THRESHOLD`), as they are considered as empty or saturated, respectively.

The core of the analysis is to compare the local temperature of the point with two different contexts:

- Nearest Neighbors (`Close Neighbors`), which are the pixels that are within a certain distance (≤ 2) in the three-dimensional volume formed by the three window images;
- Outer Neighbors (`Outer Neighbors`), i.e., the pixels between distance 3 and 4.

For each point, two averages are computed: one over the values of the close neighbors and another over the outer neighbors. If the absolute deviation between these two averages exceeds a certain threshold (`OUTLIER_THRESHOLD`), the point is considered a thermal outlier.

All points identified as outliers are saved in `batch.q2_all_outliers`, while the five with the highest deviation are selected and saved in `batch.q2_top5_outliers` in the form of a map, containing the location (x, y) and deviation value δ.

To ensure robustness in processing at the edges of the image or in case of incomplete windows, the `getPaddedValue` method provides a zero-padding mechanism: if a point required for computation is outside the spatial or temporal bounds, it simply returns zero, avoiding access errors and maintaining the consistency of the computation.

At the end of the process, the batch is enriched with outlier information and output for the next steps in the pipeline.

Query 3. *The outlier points identified in Q2 must be clustered using DBSCAN4 algorithm, using Euclidean distance as metric.*

The last query is handled by the `Q3ClusteringMapFunction` class, which analyzes the points stored in the `q2_all_outliers` field of each batch. If this list is empty or missing, it means there are no outliers to process, so the batch is simply returned with an empty `q3_clusters` field. When outliers are present, the first step is to convert their coordinates: each point, originally stored as a list of Number objects (representing the [x, y] values), is transformed into an array of doubles. This conversion is required to feed the data into the clustering algorithm DBSCAN (Density-Based Spatial Clustering of Applications with Noise).

Before clustering is started, a check is made on the minimum number of available points. If there are less than 5 elements, DBSCAN is not executed, since this number represents the minimum threshold (`MIN_POINTS` = 5) for a set of points to be considered a valid cluster. In such a case, again the `q3_clusters` field is left blank.

Instead, when there are enough points, the DBSCAN algorithm is executed with `EPSILON` = 20, which represents the maximum radius within which two points can be considered neighbors;

The algorithm returns an array of labels, that associates each point with a cluster ID, or -1 if the point is considered noise and therefore ignored.

At this stage, the points are grouped based on their assigned cluster labels. For each cluster, the centroid is computed as the average of the X and Y coordinates of its points. Along with the centroid's position, the cluster size (the number of points it contains) is also determined. Finally, each cluster is serialized into a JSON string representation and saved in the `q3_clusters` field of the batch. The batch, updated with cluster information, is then returned and can later be sent to the Local Challenger.

B. Optimization Queries

Query 2.
We then focused on optimizing the application logic by introducing the `Q2OutlierDetectionOpt` class, as an alternative to the original implementation presented in Query 2.

The central aspect of the optimization concerns the spatial symmetry present in the local deviation calculation step. Specifically, the analysis distinguishes between close neighbors and outer neighbors, based on the manhattan distance between pixels.

In the original version, the two sets were processed separately, iterating twice over the entire three-dimensional window. In the optimized implementation, we instead exploit the symmetry of the manhattan distance to run a single unified loop on all possible *dx*, *dy*, and *d*, and dynamically compute which set each neighbor belongs to, based on the distance value.

This modification makes it possible to:

- reduce the total number of iterations performed in the calculation;
- limit repeated memory access for pixels that have already been evaluated;

Another small but important optimization involves the introduction of a constant `MAX_COORD_OFFSET`, which defines the maximum range for dx and dy, equal to 2 * `DISTANCE_THRESHOLD`. This approach eliminates redundant computations in cycle bounds and improves efficiency in handling neighbors in the relevant range.

To exploit the spatial symmetry of neighbourhood analysis, the basic logic of outlier detection was optimised using a 3D discrete convolutional kernel. Instead of iterating explicitly on the spatial and temporal neighbours for each point, the implementation builds a three-dimensional kernel that encodes the weights for neighbouring and outlying neighbours.

This kernel, defined on a cube of size 3×5 (depth×height×width), assigns:

- a positive weight of $\frac{1}{|C_p|}$ to each point q in the *close neighborhood* C_p, i.e., where the Manhattan distance $\delta(p, q) \leq 2$;
- a negative weight of $-\frac{1}{|O_p|}$ to each point q in the *outer neighborhood* O_p, where $2 < \delta(p, q) \leq 4$;
- a weight of 0 elsewhere.

The 3D convolution is then applied to the window tensor W (of dimension $3 \times H \times W$), centred on the most recent layer. The result is a 2D deviation map in which each value represents the local deviation δ_p at pixel (x, y), calculated as the difference between the mean values of the two neighbourhoods.

This transformation allows the deviation map to be calculated in a single pass on the image stack, improving both readability and performance compared to the manual summation approach. In addition, as the kernel remains constant in all batches, it can be precompiled once and reused efficiently during stream processing.

However, these performance benefits are most significant when the computation is offloaded to a GPU, (see results in Section IV).

The complete dag of the execution flow of the three queries is shown below.

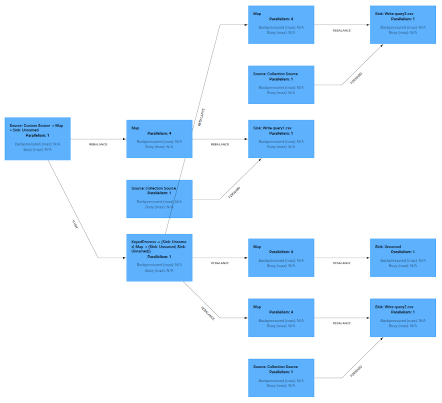

Fig. 2: Complete Flink Execution Dag

IV. EMPIRICAL RESULTS - FLINK VERSION

For the execution of the pipeline, each task manager has the default container configuration provided by the flink docker image, so with:

- `CPU`: 1;
- `Heap RAM`: 1 GB;
- `slots per TaskManager`: 1.

As mentioned above, the cluster is initialised directly with 8 task manager and the value of the parallelism used for query execution is varied.

For experimental purposes, both implementations, kernel-based and loop-based, were tested and compared to assess which one delivered better performance under the available hardware conditions. This comparative evaluation aimed to determine whether the added complexity of the kernel-based approach translated into tangible performance benefits in a CPU-only environment.

The system metrics presented in Figures 3, 4, and 5 were collected from the Local Challenger via its REST API, providing measurements of average latency, maximum latency, and throughput for each pipeline configuration.

As expected in an environment without GPU acceleration, the kernel-based implementation resulted in reduced throughput and increased latency. This result is not surprising, considering that the application of a convolutional kernel, especially in three dimensions, is a computationally intensive operation when performed solely by the CPU.

As shown by the latency and throughput graphs, the optimisation without kernel applied to Query 2 yields clear benefits, resulting in lower latencies and higher throughput across different levels of parallelism.

When analyzing throughput with respect to the number of task managers used, both the optimised and non, optimised versions of the query exhibit similar trends. Specifically, throughput increases as the number of task managers grows. Up to 4 task managers, the increase is nearly linear, an indication that the workload is efficiently distributed and the system effectively benefits from horizontal scalability.

Beyond this point, from 5 to 8 task managers, throughput begins to fluctuate, revealing the impact of I/O contention and synchronisation overhead. Nevertheless, no drastic drops are observed, which confirms that the system maintains good scalability even beyond the ideal parallelism threshold.

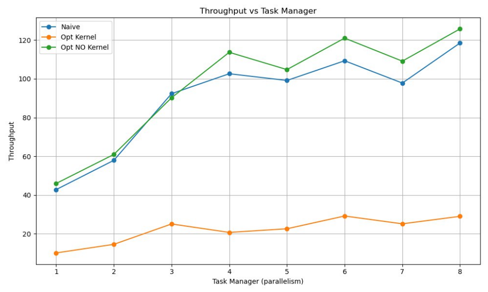

Fig. 3: System Throughput

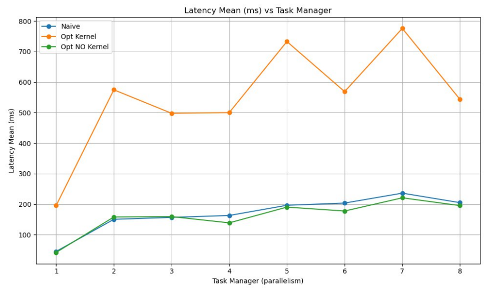

Fig. 4: System Latency Mean

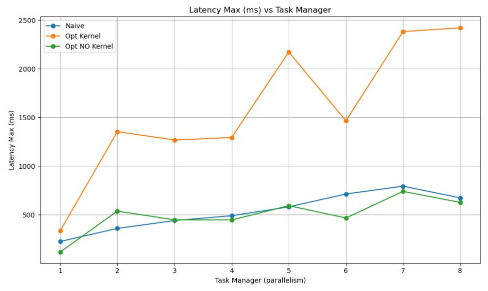

Fig. 5: System Latency Max

Latency behaviour is more nuanced. While the average latency generally increases with parallelism, due to higher communication and coordination between distributed operators, this trend is not uniform across all configurations.

Finally, thanks to Flink's native backpressure mechanism, the system maintains a relatively stable processing rate even under varying configurations.

Backpressure guarantees a dynamic balancing between the different pipeline phases: if a downstream operator slows down, Flink limits the throughput of upstream data, keeping the overall processing speed constant. This implies that, regardless of the type of optimisation applied, the observed performance is very similar.

This behaviour is reflected in the graphs below, which illustrate the performance of the three queries executed individually under varying degrees of parallelism.

To obtain these performance metrics, the system leverages a custom Flink sink operator, `QueryMetricsSink`, integrated at the end of each query pipeline. This component tracks, for each batch processed, the time elapsed since ingestion.

Specifically, the sink maintains internal counters to measure:

- the total number of processed events (`count`);
- the cumulative latency for computing the average;
- the maximum observed latency;
- the time span between the arrival of the first and last batch (to calculate throughput).

When the pipeline completes, the `close()` method is triggered, and the collected statistics are written to a CSV.

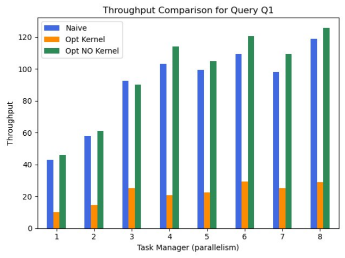

Fig. 6: Q1 Throughput

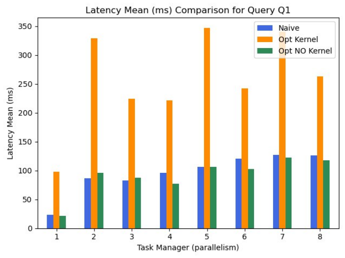

Fig. 7: Q1 Latency Mean

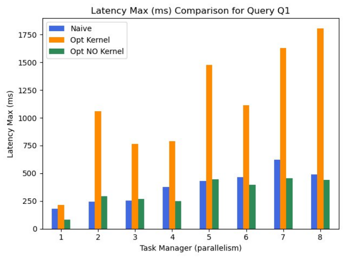

Fig. 8: Q1 Latency Max

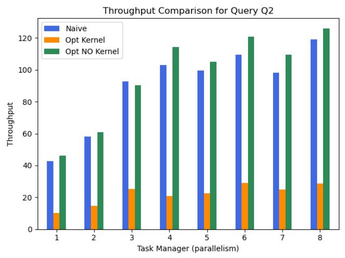

Fig. 9: Q2 Throughput

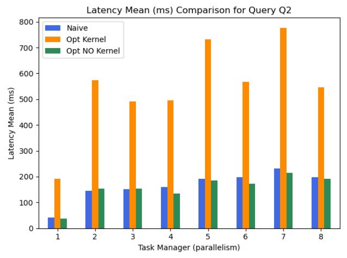

Fig. 10: Q2 Latency Mean

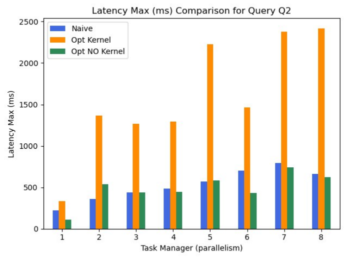

Fig. 11: Q2 Latency Max

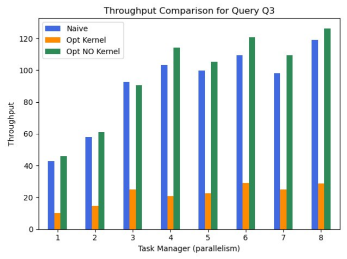

Fig. 12: Q3 Throughput

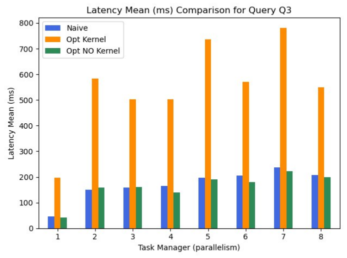

Fig. 13: Q3 Latency Mean

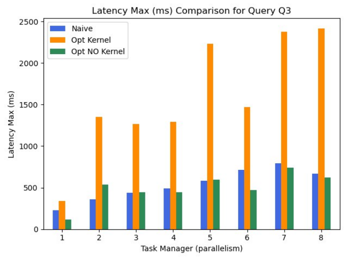

Fig. 14: Q3 Latency Max

V. System Architecture - Kafka Streams Version

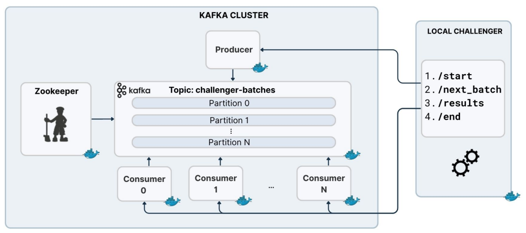

Fig. 15: System Architecture - Kafka Streams Version

The diagram above illustrates the system architecture, highlighting the interaction between the Kafka Cluster and the Local Challenger.

- **Local Challenger**: as previously described, it interfaces with the system through REST calls.
- **ZooKeeper**: coordinates metadata across distributed Kafka brokers. It manages topic configurations, partitions, and offsets, ensuring proper synchronization within the Kafka cluster;
- **Kafka**[3]: is responsible for storing and replicating batches on topics, managing topic partitions and providing durability and order within each partition;
- **Kafka-producer-app**: this component retrieves batches from the Local Challenger and publishes them to the appropriate Kafka topic. It initializes the topic and creates the required number of partitions at startup, distributing batches across them;
- **Kafka-consumer-app**: is the Kafka Stream application, consumes the messages from the topic and applies the processing of the three queries. Parallelism is handled automatically by Kafka Streams, with one thread assigned to each topic partition.

VI. Pipeline - Kafka Streams Version

A. Data Ingestion

The ingestion process starts with the Kafka producer fetching batches from the Challenger via RestAPI. Each batch is deserialized and published to a Kafka topic. The producer dynamically creates the topic with a configurable number of partitions, which determines the maximum parallelism of downstream processing. Batches are assigned to topic partitions by the producer using a round-robin strategy based on their `batch_id`, specifically, each partition is mapped using `batch_id % partitions`. This ensures a balanced distribution of load across all consumers.

B. Processing: Queries

On the consumer side, the `kafka-consumer-app` leverages Kafka Streams to subscribe to the topic and independently process each partition. Each batch is deserialized and passed through a streaming pipeline that mirrors the Flink logic, applying the Q1, Q2, and Q3 analytical stages sequentially.

Unlike Flink, Kafka does not manage the lifecycle and coordination of consumers at the framework level. This means the responsibility for orchestrating execution and termination is delegated entirely to the user. In our setup, this includes determining when the benchmark should be concluded and which consumer should trigger the finalization. This coordination challenge is addressed by monitoring the batch with `batch_id = 3599`, which represents the last batch in the dataset. The consumer that processes this final batch waits for an additional 10 seconds to allow all other partitions to complete their processing. Only then, it invokes the `/end_benchmark` endpoint on the Local Challenger, ensuring a synchronized and clean termination. Once the benchmark is closed, the system exports the performance metrics, such as latency and throughput for each query.

VII. EMPIRICAL RESULTS - Kafka Streams Version

To evaluate system performance in terms of throughput and latency, the Kafka topic is configured with a replication factor of 1, ensuring that each batch is consumed exactly once. Every message is read by a single consumer instance and immediately processed through the entire query pipeline.

The analysis of the results collected via the Local Challenger clearly demonstrates that the Kafka-based architecture exhibits good horizontal scalability in both throughput and latency. As the number of consumers increases, a notable improvement in overall throughput is observed, indicating effective exploitation of parallel resources.

The following graphs report the metrics gathered via the Local Challenger. Compared to the Flink pipeline, the Kafka-based solution delivers improved metrics, reflecting the benefits of its simpler execution model and fine-grained control over parallelism.

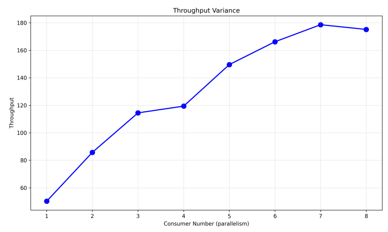

Fig. 16: Throughput Variance in Normal run

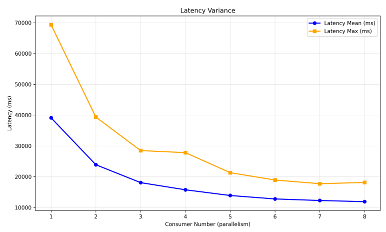

Fig. 17: Throughput Variance in Optimized run

Since Kafka Streams lacks a native backpressure mechanism, the latency and throughput observed for individual queries may differ significantly from those obtained with Flink. In this setup, latency is measured exclusively over the query processing time, with no additional network overhead.

For Query 1, which involves a simple saturation check, the results align with expectations: low latency and high throughput. Query 2, on the other hand, is more computationally intensive due to the sliding window logic and spatial analysis, resulting in higher latency and lower throughput.

Additionally, the performance of Query 2 and Query 3 is influenced by how batches are distributed across Kafka partitions. If a consumer receives a partition with batches that contain no outliers, the processing time is reduced, leading to lower latency and higher throughput for that specific consumer. These data distribution imbalances can affect the overall evaluation of system performance and should be carefully considered when interpreting the results.

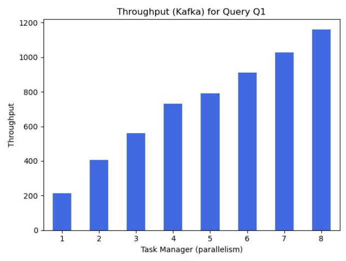

Fig. 18: Q1 Throughput

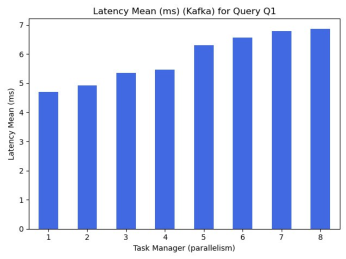

Fig. 19: Q1 Latency Mean

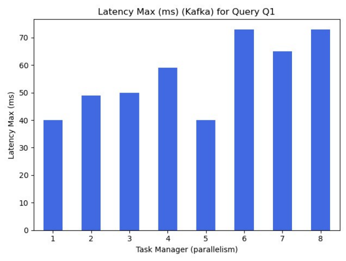

Fig. 20: Q1 Latency Max

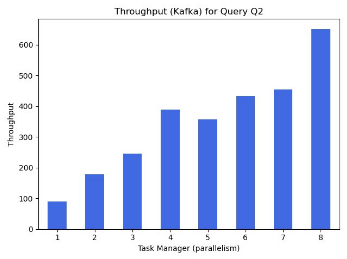

Fig. 21: Q2 Throughput

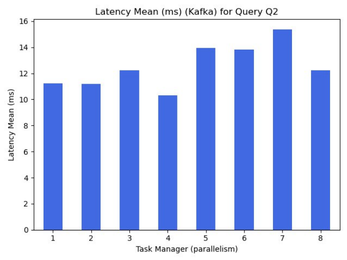

Fig. 22: Q2 Latency Mean

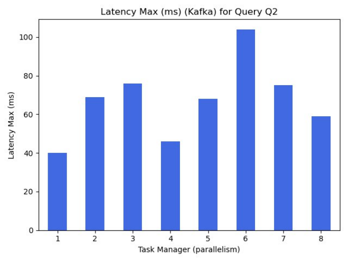

Fig. 23: Q2 Latency Max

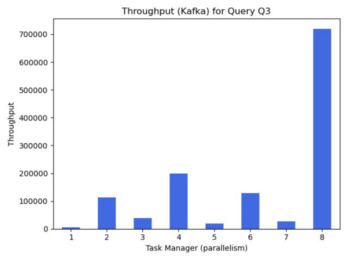

Fig. 24: Q3 Throughput

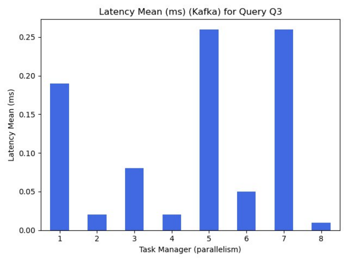

Fig. 25: Q3 Latency Mean

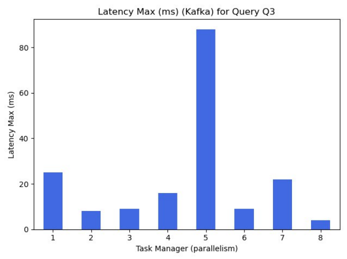

Fig. 26: Q3 Latency Max

References

[1] LOCAL-CHALLENGER Client. URL: https://dl.acm.org/doi/full/10.1145/3701717.3735578.

[2] Apache Software Foundation. *Apache Flink*. URL: https://flink.apache.org/.

[3] Apache Software Foundation. *Apache Kafka*. URL: https://kafka.apache.org/.