A Comprehensive Performance Analysis of Stream Processing with Kafka in Cloud Native Deployments for IoT Use-cases

István Pelle[1,2,3], Bence Szőke[1], Abdulhalim Fayad[1,2], Tibor Cinkler[1,2], László Toka[1,2,3]
[1]*HSN Lab, Department of Telecommunications and Media Informatics, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics*, Műegyetem rkp. 3., H-1111 Budapest, Hungary;
[2]*ELKH-BME Cloud Applications Research Group*, Műegyetem rkp. 3., H-1111 Budapest, Hungary;
[3]*MTA-BME Network Softwarization Research Group*, Műegyetem rkp. 3., H-1111 Budapest, Hungary
pelle.istvan@vik.bme.hu, szoekebence@edu.bme.hu, fayad@tmit.bme.hu, cinkler@tmit.bme.hu, toka.laszlo@vik.bme.hu

***Abstract*—The constant growth of the number of Internet of Things devices drives a huge increase in data that needs to be analyzed, at times in real time. Multiple platforms are available for delivering such data to analytics engines that can perform various operations on the data with low processing latency. These platforms can find their home in cloud native environments where high availability and scaling to the actual workload can be easily achieved. While the deployment environment is elastic, clusters still need to be adequately dimensioned to accommodate the components of the platforms even under high load.**

In this paper, we provide an analysis in this regard: we discuss key performance indicators of the popular Kafka message bus and the related Kafka Streams processing engine. Namely, we analyze latency, throughput, CPU and memory resource footprint aspects of these services under varying load and processing tasks that appear in Internet of Things applications. We find subsecond processing latency and linear but heavily task-dependent scaling behavior in the other performance indicators' case.

***Index Terms*—Kafka, Kafka Streams, stream processing, performance analysis, IoT**

I. Introduction

In the "always-on" world we find more and more devices getting interconnected which induces a rise in the demand for developing and maintaining highly available systems that can store, analyze and manipulate extremely large and continuously expanding amounts of data safely with low latency. With the advance of cloud computing, many of these systems are transitioning into a cloud native environment where they can be operated in a robust and easily scalable fashion. Such systems are indispensable for many areas, as Internet of Things (IoT), e-commerce, financial and investment services.

The abundance of sensors used in IoT ecosystems creates astronomical amounts of data and analyzing it has vast benefits in controlling complex environments via enhanced decision-making [2]. By looking at data snippets in real time, stream analytics can create control decisions, patterns, and statistics assisting IoT applications in decision-making in a range of use cases encompassing societal, environmental, and economic aspects [3]. Among others, Amazon Kinesis and Apache Kafka [4] are present for stream delivery, while Apache Flink, Spark and Kafka Streams exist for processing tasks [5]. Due to Kafka being one of the most widespread platforms and having a tight integration with Kafka Streams, here we are setting out to examine their conjoined processing capabilities in cloud native deployments that grant adaptation to actual load. We argue that understanding the implications on resource footprint still remains crucial for designing adequately-sized data processing clusters that can handle the expected load while still consuming as few resources (CPU, memory, energy) as possible to keep expenses low and satisfy green aspects. Our contributions are thus threefold. First, we identify IoT use-cases that can benefit from stream analytics and provide the background for the cloud native deployment. Second, we define a measurement framework for capturing throughput, processing latency and (CPU and memory) resource footprint metrics. Third, we evaluate these metrics using various scenarios under different processing operations deployed on a Kubernetes-managed [6] environment.

Consequently, we organize our paper in the following structure. In §II, while also underlining the motivation behind our investigations and selection of stream delivery and analytics tools, we give a comparison with related work. In §III, we continue this by looking into various IoT use-cases that showcase the importance of data analytics. We also summarize Kafka and Kafka Streams capabilities and briefly review cloud native deployment aspects here. Later, we discuss the considerations of our measurement framework's setup and selected scenarios in §IV. We provide a detailed description of our measurement

This work was supported by: i) Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund through projects a) no. 135074 under the FK_20 funding scheme, b) 2019-2.1.13-TÉT_IN-2020-00021 under the 2019-2.1.13-TÉT-IN funding scheme, c) 2019-2.1.11-TÉT-2020-00183 under the 2019-2.1.11-TÉT funding scheme, d) 2021-1.2.4-TÉT-2021-00058 under the 2021-1.2.4-TÉT funding scheme, ii) ÚNKP-22-5-BME-317 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund. L. Toka was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. On behalf of the Performance Analysis of Stream Processing project we are grateful for the possibility to use ELKH Cloud (see Héder et al. 2022 [1]; https://science-cloud.hu/) which helped us achieve the results published in this paper.

environment in §V and also evaluate measurement results there. Finally, we conclude our paper in §VI.

II. Related Work

The IoT stream processing technology using the Apache Kafka framework has gained attention in several papers in the literature. A comprehensive survey [5] of several distributed data stream processing and analytics frameworks studies open source (Storm, Spark Streaming, Flink, Kafka Streams) and commercially available (IBM Streams) distributed data stream processing frameworks from multiple angles, however, it does not supply quantitative resource footprint, throughput or latency analyses. Other authors [7] provide an evaluation and comparison of Kafka with other available technologies. According to their results, Kafka proved to be one of the best options to achieve high performance while staying under budget to stream data in real-time. Theodolite [8] is a benchmarking framework that assesses the scalability of Flink and Kafka Streams. While its authors provide an evaluation of different processing operations in terms of number of data sources and used processing instances, they do not deliver actual CPU and memory usage statistics that can be easily used for dimensioning purposes. Other papers [9], [10] also prepare comparative evaluations of different stream processing engines, however, they do not cover Kafka Streams. A further study [11] gives a detailed comparison of three major stream processing frameworks (Flink, Kafka Streams and Spark) from the perspective of different processing operations and input characteristics of sustained load, single burst load and periodic bursts. It offers CPU, memory and latency analyses of the frameworks, but does not include dimensioning aspects specific to the Kafka event bus. Other authors [12] also investigate Spark, Flink and Storm latency and scaling aspects but do not include Kafka Streams.

In terms of discussing resource utilization metrics, the work of Shahverdi *et al* [13] is the closest to ours. They present a comparison of Storm, Flink, Spark (DStream and Structured Streaming), Kafka Streams and Hazelcast Jet. They provide latency and resource usage metrics both for the stream analytics engines and the Kafka event bus. CPU can be easily used for dimensioning purposes, while reported memory loads can also be leveraged after additional calculations. However, in their case, Kafka is used only for delivering events to the processing engines, their output is pushed to an external Redis in-memory cache instead of back to Kafka. This separation hinders the results' applicability for cases where Kafka is used as a unique, general event delivery mechanism that forwards derived data as well.

III. Background

A. IoT data analytics use cases

The vast amount of data with different velocities generated by various IoT devices makes IoT data analytics an important and challenging topic. From *smart cities* [14] and *buildings* through *environmental monitoring*, *agriculture*, *healthcare* and *education* [2], many IoT settings profit from data analytics. Analytics of data generated by IoT devices in *smart industry* environments is employed for predictive maintenance, in the *retail sector* for tracking and improving customer experience. Analytics is also essential for *smart metering* which measures and tracks power usage in *smart grids* to help predict future electricity requirements [15]. In such situations, the demand exists for sending data to scalable systems providing proper collection methods, persistent storage with high throughput. Apache Kafka emerged as a viable option for streaming data coming from various IoT devices and then distributing streams of data to different applications. It has become an essential option for IoT data analytics due to its reliability and stability in processing real-time streaming IoT data with minimum latency, and high throughput [5], [16]. Various works suggest its use in cloud native settings for scaling to real-time data streams coming from IoT devices [17]–[19], they also leverage compatible monitoring tools, such as Prometheus.

B. Event stream delivery: Kafka

Apache Kafka is a popular open-source, low-latency, high-performance, fault-tolerant distributed messaging system used for collecting, processing, storing, and analysing event streams (replayable, ordered, unbounded, continuously updating data sets of immutable events, records, key-value pairs) at scale. Kafka clusters consist of (optionally geographically distributed) broker entities that are coordinated by ZooKeeper [20] nodes and durably store event streams in multi-producer, multi-subscriber replicable topics partitioned across (a subset of) the cluster's brokers. I.e, clients can read or write events in a topic frequently accessing many brokers at the same time. Upon publishing a new event to a topic, it is appended to one of the topic's partitions and it will be replicated to other brokers for the sake of fault-tolerance. Events are discarded only after the configured per-topic retention time and the cluster's performance is promised to be constant with respect to stored data size. Distributed applications and microservices writing, reading, and processing streams of events can be developed with Kafka clients using the Producer, Consumer and Streams APIs [21], respectively.

C. Event stream processing: Kafka Streams

Leveraging this last API, Kafka Streams applications act as stream consumer, processor/analyzer and producer applications. Their instructions are interpreted as processor topologies, logical abstractions of graphs consisting of processor nodes connected by stream edges. Nodes get events from their upstream nodes, perform their operation and forward one or more (modified) events to downstream processors. Two special node types are present: sources for reading Kafka topics (i.e., having no upstream nodes) and sinks for writing (i.e., without downstream nodes). Different processing operations require different aspects of the same data. On the one hand, a stream can be considered a changelog of a table, where the stream's each data record captures a state change of the table. On the other hand, a table can be considered a snapshot, at a point in time, of the latest value for each key in a stream.

Kafka Streams provides support for both aspects, handling the so-called stream-table duality via the KStream and KTable interfaces that support a variety of transformation operations specific for each aspect. Transformations are categorized as being stateless (requiring no knowledge of other events, e.g., filter and map operations) or stateful (that use a state depending on data from multiple events, e.g., a windowing state in aggregation or join).

Processing is scaled by launching multiple tasks created from the processor topology. In layman's terms, each task is assigned to at least one partition and can be executed in different application instances or threads (without shared state) of a single instance providing similar performance but different fault-tolerance. Partition to task assignment and load balancing between parallel tasks are handled transparently.

D. The cloud native environment

Docker [22] is the de facto standard for running standardized units of software isolated from each other and their environment. Kubernetes is an open-source container orchestration platform for automating deployment, management and scaling of applications (packaged as Docker images) making it ideal for hosting cloud native applications that require rapid scaling, like real-time data streaming through Apache Kafka. Strimzi [23] provides an option for doing just that: it is an open-source Cloud Native Computing Foundation project providing an operator for running Kafka clusters. It is able to set up ZooKeeper nodes, Kafka brokers and topics with TLS encryption enabled between the components. It also provides integration with the Prometheus [24] monitoring and Grafana [25] observability tools for exposing, collecting and displaying various performance metrics of the cluster, while Netdata [26] can be used for collecting (among others) pod-level CPU and memory load in a Kubernetes environment.

IV. Measurement layout and scenarios

Fig. 1 depicts the layout of the components and their interactions that we used to measure the performance and resource footprint of Kafka and Kafka Streams. The figure shows four main components (from top to bottom): the Kafka Streams Processor, the Kafka cluster, a Producer, and a Consumer, all running as pods within a Kubernetes cluster. We used the Strimzi operator for setting up a Kafka cluster consisting of N brokers in Kubernetes. Strimzi created additional operator pods (not depicted in the figure for the sake of simplicity), certificates for TLS encryption and ZooKeeper pods for coordinating the Kafka cluster as well. These latter pods supply the available broker IDs and consumer offsets (which event to consume next) to the Processor and Consumer components. We also relied on Strimzi for configuring two topics in Kafka: T_{in} is used for feeding the Kafka Streams processor application while T_{out} stores its output.

We implemented the Producer as a configurable Java application using the Producer API packaged into a Docker image. It has access to 8760 event files which come from an industry use-case and contain JSON data with multiple nested fields

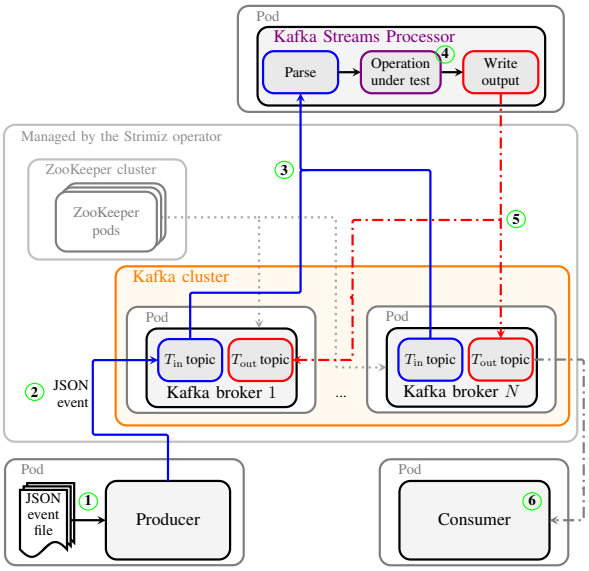

Fig. 1: Measurement setup.

relaying IDs, measurement and status data, as well as duration related information using timestamps. The Producer randomly selects a file (see step 1 in Fig. 1 and sends it to the T_{in} topic of the Kafka cluster (step 2). As additional configuration arguments, we used a target throughput rate for the Producer, various batching and TLS configuration options, as well as the Kafka cluster's address for sending out events. The producer tries to commit the events to the Kafka brokers at the given rate and signals when it cannot match the defined rate.

From T_{in} the Kafka Streams Processor application reads the events (step 3). We also implemented the Kafka Streams Processor as a configurable Java application using the Kafka Streams API packed into a Docker image. The application first parses the incoming data: corresponding to a source node in its processor topology, it deserializes events as Java entities to have access to the various fields within the JSON records. Then, in step 4, it performs a selected operation on the event (see §IV-A for a discussion on the various operations used) before writing back the operation's results to the T_{out} Kafka topic by serializing them again (creating a sink node in the processor topology), in step 5. To configure communication with the Kafka cluster, we supplied its address and TLS setup options (certificates) to the Processor. To test parallelism, we made the number of processing threads configurable and defined the maximum commit interval in which the Producer needs to write data to the Kafka cluster.

The optional Consumer Java application (also packaged as a Docker image) reads data from T_{out} in step 6 (by connecting to the configured cluster address). Using the Kafka Consumer API, we implemented this component to check the validity of the Processor's output in a human readable format. We did not use the component during the measurements but it proved to be extremely helpful while developing, testing and validating the stream processor application and the various operations.

For setting up the TLS encryption between the Producer

and Kafka, as well as between Kafka and the Processor, we used the certificates created by Strimzi. For encrypted and unencrypted communications, we used the dedicated listener port provided by Strimzi and the respective service where it granted communication access to the cluster.

During our measurements, we targeted the crucial metrics of maximum throughput, CPU and memory footprint of Kafka and Kafka Streams, as well as processing latency. Using Strimzi, we exposed Kafka metrics via its Exporter [23] and enabled a similar Java Management Extensions Exporter [27] in our own Kafka Streams Processor application. We scraped the exposed metrics with Prometheus and analyzed the data through Grafana dashboards. Using these, we were able to observe the input rate of the T_{in} topic form the Kafka metrics and the processing rate and latency of the Processor from the Kafka Streams metrics. To gauge the maximum throughput, we increased the Producer output until the arrival of the signal warning us that the specified event output rate cannot be sustained. We captured CPU and memory load metrics of the deployed components using Netdata.

A. Scenarios

We created three scenarios with subcases in order to define different processor topologies and investigated their effects on performance and resource footprint based on simple but frequently used use-cases. Two scenarios focus on different stateless operations while one examines stateful operations.

The first stateless scenario is a filtering use-case, where we define conditions to filter out different amounts of events from the input stream. By matching a certain ID field with the filter operation in each of the events, we keep 0.05%, 1% or 10% of the original data without any modifications on any of the fields that we write to T_{out}.

In the second stateless scenario, we investigate the effects of changing fields in input events. We perform anonymization to mask sensitive data by replacing all the occurrences of certain field values in events with *X* characters. We selected fields and values to keep the number of fields where the operation takes effect under 1% of the total number of fields present in each event. To achieve this, we use the map method of the Kafka Streams library before writing each input event to T_{out}.

The third scenario experiments with stateful aggregation operations using three subcases. In the *i*) *basic block aggregation* subcase, the aggregation is performed on a group of events when a specific criteria is satisfied. Here we target the scenario of counting the occurrences of different values of 11 keys in the events. The aggregated result is a key-value pair containing 11 fields which is written to T_{out}. With the *ii*) *grouped block aggregation* case, we aggregate a group of events in a specific granularity window grouped by all possible values of one or more fields. The targeted task is to count the number of various distinct event codes in the input events. In the output written to T_{out}, the keys are the occurring event codes, and the values are the number of event occurrences. In the *iii*) *averaging block aggregation* case, we perform the aggregation on all events of a specific granularity window unconditionally. The aggregated result written to T_{out} is a simple float number supplying the average difference between each event's start and end timestamps.

V. Measurements and evaluation

To conduct our measurements, we used different environments. Results shown in the following were obtained from a Kubernetes cluster set up in the ELKH Cloud [1], a compute cloud service specialized for research purposes. Our cluster consisted of a single master and 6 worker nodes running as virtual machines (VM) managed by OpenStack. Due to the nature of the provider, it was impossible to select the hardware nodes where our VMs were set up by the provided automation. To increase the possibility of VMs being deployed to different hardware nodes and to provide ample resources for tests even with high loads, we selected VMs with high resource flavors. Thus, 32 vCPUs (Intel Cascadelake Xeon Processor), 64 GiB of memory and 1 TB of SSD storage were assigned to each worker node. As shown in the results discussed later, these resources were never fully utilized, however. Results gained on the cluster were validated by repeating the measurements in a Kubernetes cluster deployed on a single bare metal server having 40 physical CPU cores (Intel Xeon Gold 6230), 188 GB of memory and 2 TB of HDD storage. Both setups ran Ubuntu 20.04 and Kubernetes v1.24. We configured Strimzi to set up 3 ZooKeeper nodes in every scenario as those proved to be sufficient for handling our measurement cases. We varied the number of deployed Kafka brokers and Kafka Streams processing threads to reach the highest throughput achievable on our cluster. In each measurement, we used a Kafka replication factor of 3 for both topics, i.e., each event was stored at 3 brokers at any given time which is the recommended setting. We always used twice as many partitions in both topics as brokers and a corresponding number of parallel threads in the Producer to push data to the Kafka cluster. In order to give ample resources to the Processor, we set it up with limits of 15 vCPUs and 30 GB of memory and specified a maximum commit interval of 1 s.

A. Baseline measurements

To gauge the performance of our Kafka cluster, we first set up a scenario without the Processor application. Employing 12 brokers, we reached a maximum throughput of 32 kEvent/s. We also found that the highest throughput per vCPU can be reached at 2 vCPUs and 4 GB of memory in each Kafka broker. We used these settings for the rest of our measurements. We also analyzed the effects of Kafka retention and measurement time coupling Kafka with the Processor application. According to our measurements, CPU and memory loads do not show observable changes in the cases of setting retention time to 1 h or 15 min. We observed the same for measurement times of 1 h and 30 min. Thus, we ran all of our further measurements for 30 min with a Kafka retention time of 15 min and evaluated average performance metrics gained during this time. We also found that due to their Java implementation, at high loads, Kafka brokers fill up $\sim 90\%$

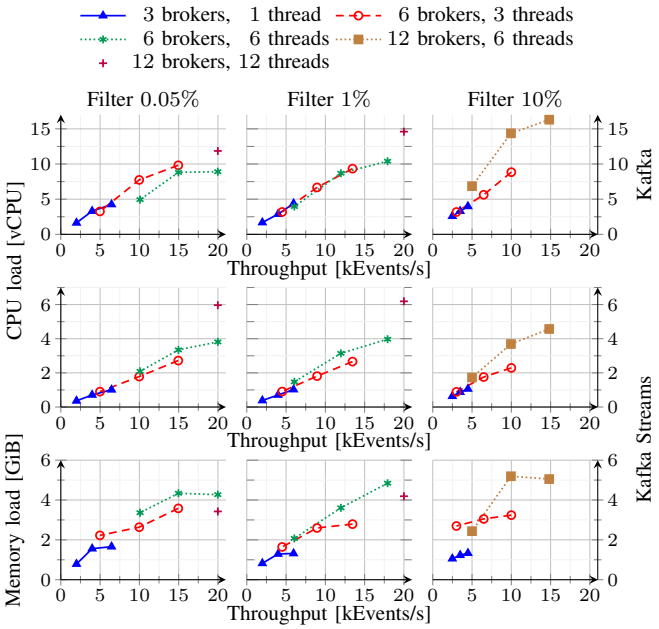

Fig. 2: Resource footprint of the filtering case.

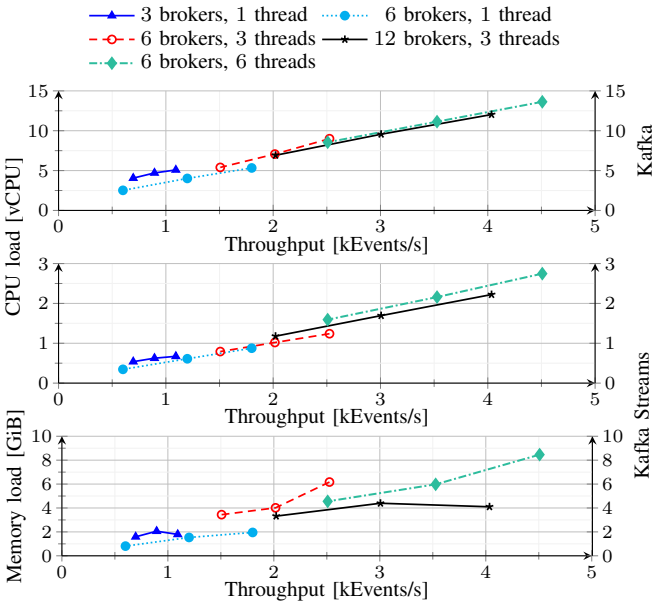

Fig. 3: Resource footprint of the anonymization case.

of the memory allotted to them during this measurement time which they retain for extended periods. Due to this effect, we do not discuss Kafka memory usage in the following, as it can be calculated from the number of the used brokers.

B. Filtering scenario

Fig. 2 shows the measurement results of the filtering scenario. The columns of the figure display the different subcases (keeping 0.05%, 1% or 10% of incoming events) and rows display Kafka and Kafka Streams CPU load as well as Kafka Streams memory load. Different lines on each subfigure represent different parallelism settings of Kafka brokers and Kafka Streams processing threads. The rightmost endpoint of each line shows the highest achievable throughput with the given configuration. According to the results, we do not find observable differences between the maximum throughput of the first two filter cases, however they are already much lower than what we reached in our baseline measurements, and the 10% filter further loses ∼5 kEvents/s throughput. Kafka and Kafka Streams CPU load characteristics of the two smaller cases are extremely similar with the exception of Kafka CPU load at the maximum throughput, where it is significantly higher compared to the 0.05% case. For lower parallelism, the 10% filter case causes similar CPU load albeit at a lower throughput, while for high parallelism, CPU is also significantly higher both for Kafka and Kafka Streams. The evolution of Kafka Streams memory load is similar in every case. Processing latency was always under 0.4 s.

C. Anonymization task

As depicted by Fig. 3, the anonymization task shows similar minimum and maximum Kafka CPU loads, however, at the much lower maximum throughput of ∼4.5 kEvents/s compared to the filtering scenario. Maximum Kafka Streams CPU load is also singificantly lower, reaching 2.75 vCPU compared to the 4–4.5 vCPUs of the filtering case. This is most probably due to the increased load caused by writing each input event back to the T_{out} topic thus causing Kafka to become a bottleneck sooner. Kafka Streams memory load is significantly higher than in the previous case due to the more taxing operation type. Processing latency is still below 0.4 s.

D. Aggregation tasks

In all of the subcases, we aggregate the events using a 1 s time window. According to our measurements (shown in Fig. 4), this scenario performs the worst as all subcases can go only slightly above 1 kEvents/s. As we increase parallelization, only an increasingly lower ratio of the available Kafka CPU resources is leveraged. The evolution of Kafka Streams CPU load is similar in each case while memory load is the highest at the basic subcase (among every measured case) and significantly decreases for the other subcases. Maximum processing latency is ∼0.6 s.

E. TLS

When enabling TLS in our setup, we did not experience a decrease in maximum achievable throughput or an increase in processing latency. While memory usage was similar to previous cases, we observed a 6%–9% increase in Kafka CPU load and another 7–10% increase in Kafka Streams CPU load. Table I details these changes for the measured filter cases.

F. Applicability of results

We have observed various performance differences that can be attributed to the different execution environments. We executed comparative measurements on OpenStack-managed

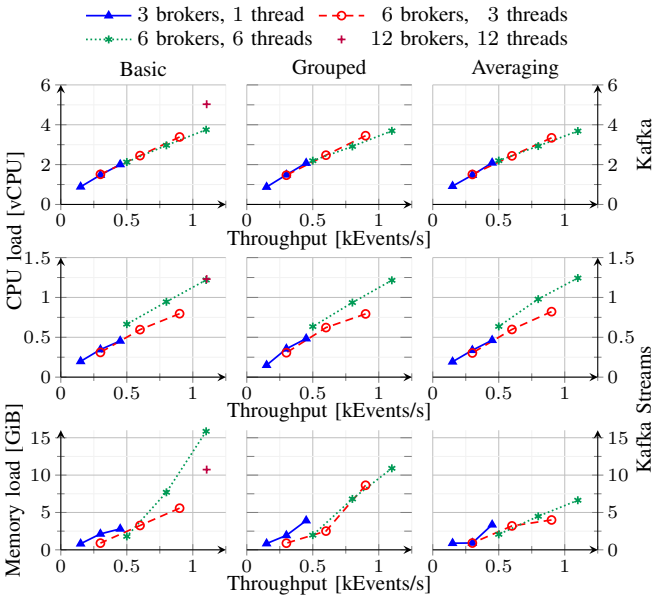

Fig. 4: Resource footprint of the aggregation case.

TABLE I: Footprint differences with enabled TLS.

Filter operation	Change [%]				
	Kafka	Kafka Streams		Total	
	CPU	CPU	Memory	CPU	Memory
0.05%	6	10	−3	7	0
1%	9	10	−4	9	3
10%	8	7	−1	8	2

VMs with Intel Xeon E5-2620 v3 CPUs. Comparing the results to CPU benchmark scores, we found that the performance difference ratio was corresponding to the ratio of benchmark scores. We also found that running Kubernetes over a cluster of VMs of the ELKH cloud severely limits achievable network bandwidth. There, the throughput between Kubernetes pods was around 30% of the throughput achievable between the worker VMs which significantly limited our Kafka cluster's ingest capabilities. We attribute this to setup and configuration specifics (undisclosed to us) of the used deployment and provider, thus general conclusions cannot be confidently made on this.

VI. Conclusion

We presented a measurement framework for gauging key performance indicators of the Kafka event bus and the Kafka Streams processing engine. We evaluated the indicators using different operations and found that while both platforms scale approximately linearly with low latency, their capability to process data is heavily dependent on the performed operation. Evaluating the platforms in different hardware and virtualization environments also revealed that the cloud native environment can significantly affect performance, sometimes through surprising interactions. Extending measurements with new configurations and operations (e.g., working with multiple topics, e.g., through join operations), could reveal further insights into processing and data delivery performance.

References

[1] M. Héder *et al.*, "The past, present and future of the ELKH cloud," *Információs Társadalom*, vol. 22, p. 128, aug 2022.

[2] T. J. Saleem and M. A. Chishti, "Data analytics in the internet of things: a survey," *SCPE*, vol. 20, no. 4, pp. 607–630, 2019.

[3] E. Siow, T. Tiropanis, and W. Hall, "Analytics for the internet of things: A survey," *ACM computing surveys (CSUR)*, vol. 51, no. 4, 2018.

[4] J. Kreps, N. Narkhede, J. Rao, *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, 2011.

[5] H. Isah *et al.*, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154300–154316, 2019.

[6] The Kubernetes Authors, "Kubernetes: Production-grade container orchestration." Available: https://kubernetes.io/, 2022. [Online] Accessed: 2022-12-09.

[7] S. Vyas, R. K. Tyagi, C. Jain, and S. Sahu, "Literature review: A comparative study of real time streaming technologies and apache kafka," in *4th CCICT*, pp. 146–153, 2021.

[8] S. Henning and W. Hasselbring, "Theodolite: Scalability benchmarking of distributed stream processing engines in microservice architectures," *Big Data Research*, vol. 25, p. 100209, 2021.

[9] J. Karimov *et al.*, "Benchmarking distributed stream data processing systems," in *34th IEEE ICDE*, pp. 1507–1518, 2018.

[10] Sanket Chintapalli et al, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *IPDPSW*, pp. 1789–1792, 2016.

[11] G. van Dongen and D. Van den Poel, "Evaluation of stream processing frameworks," *IEEE TPDS*, vol. 31, no. 8, pp. 1845–1858, 2020.

[12] H. Nasiri, S. Nasehi, and M. Goudarzi, "Evaluation of distributed stream processing frameworks for iot applications in smart cities," *Journal of Big Data*, vol. 6, no. 1, pp. 1–24, 2019.

[13] E. Shahverdi, A. Awad, and S. Sakr, "Big stream processing systems: An experimental evaluation," in *35th IEEE ICDEW*, pp. 53–60, 2019.

[14] H. Rajab and T. Cinkelr, "Iot based smart cities," in *ISNCC 2018*, 2018.

[15] H. Hejazi, H. Rajab, T. Cinkler, and L. Lengyel, "Survey of platforms for massive iot," in *Future IoT*, pp. 1–8, 2018.

[16] J. Perez, "Using Apache Kafka for Stream Processing: Common Use Cases | OpenLogic by Perforce." https://www.openlogic.com/blog/kafka-stream-processing. [Online] Accessed: 2023-01-14.

[17] A. Chawla *et al.*, "Intelligent monitoring of iot devices using neural networks," in *24th ICIN*, pp. 137–139, IEEE, 2021.

[18] K. Ogawa *et al.*, "Iot device virtualization for efficient resource utilization in smart city iot platform," in *PerCom Workshops*, IEEE, 2019.

[19] G. M. D'silva, A. Khan, S. Bari, *et al.*, "Real-time processing of iot events with historic data using apache kafka and apache spark with dashing framework," in *2nd RTEICT*, pp. 1804–1809, IEEE, 2017.

[20] P. Hunt *et al.*, "ZooKeeper: Wait-free Coordination for Internet-scale Systems," in *USENIX ATC 10*, 2010.

[21] Apache Software Foundation, "Apache kafka documentation." https://kafka.apache.org/documentation, 2022. [Online] Accessed: 2022-12-09.

[22] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[23] Strimzi Authors, "Strimzi overview guide." https://strimzi.io/docs/operators/0.30.0/overview.html, 2023. [Online] Accessed: 2023-01-13.

[24] B. Rabenstein and J. Volz, "Prometheus: A next-generation monitoring system (talk)," (Dublin), USENIX Association, May 2015.

[25] Grafana Labs, "Grafana: The open observability platform." https://grafana.com/, 2023. [Online] Accessed: 2023-01-13.

[26] Netdata Inc., "Netdata: Monitoring and troubleshooting transformed." https://www.netdata.cloud/, 2022. [Online] Accessed: 2023-01-13.

[27] Various authors, "JMX Exporter." https://github.com/prometheus/jmx_exporter, 2022. [Online] Accessed: 2023-01-13.