Living on the Edge: Serverless Computing and the Cost of Failure Resiliency

Sameer G Kulkarni[†], Guyue Liu[‡], K. K. Ramakrishnan[†], and Timothy Wood[‡]

[†]University of California, Riverside, [‡]George Washington University.

Abstract—Serverless computing platforms have gained popularity because they allow easy deployment of services in a highly scalable and cost-effective manner. By enabling just-in-time startup of container-based services, these platforms can achieve good multiplexing and automatically respond to traffic growth, making them particularly desirable for edge cloud data centers where resources are scarce. Edge cloud data centers are also gaining attention because of their promise to provide responsive, low-latency shared computing and storage resources. Bringing serverless capabilities to edge cloud data centers must continue to achieve the goals of low latency and reliability. The reliability guarantees provided by serverless computing however are weak, with node failures causing requests to be dropped or executed multiple times. Thus serverless computing only provides a best effort infrastructure, leaving application developers responsible for implementing stronger reliability guarantees at a higher level. Current approaches for providing stronger semantics such as "exactly once" guarantees could be integrated into serverless platforms, but they come at high cost in terms of both latency and resource consumption. As edge cloud services move towards applications such as autonomous vehicle control that require strong guarantees for both reliability and performance, these approaches may no longer be sufficient. In this paper we evaluate the latency, throughput, and resource costs of providing different reliability guarantees, with a focus on these emerging edge cloud platforms and applications.

I. INTRODUCTION

Edge computing [1] seeks to provide more responsive service to users, with the promise of access to shared computing and storage with low latency, thus enabling and encouraging offloading processing from end-devices. It also reduces core bandwidth costs by moving cloud-resident applications closer to users. Reducing the latency costs from tens of milliseconds to one millisecond opens a new range of cloud application types related to Cyber Physical Systems (CPS) and the Internet of Things (IoT) [1]. Applications such as autonomous vehicle control and live sensor feed analysis require both strong performance and reliability guarantees, *e.g.,* mission critical communication IoT use cases such as factory automation and autonomous vehicles demand latency in the range of 250μs-10ms and a reliability of 10^{-9} packet loss rate [2].

Cloud computing's fundamental appeal arises from the exploitation of multiplexing computing and storage by sharing resources and thus lowering costs to end-users. Serverless computing takes this multiplexing to the next level by reducing the time resources are committed to a particular user or application to just the time needed to execute an invoked function. Serverless computing thus provides a new approach to managing cloud resources by deploying applications in dynamically instantiated containers. For instance, Amazon provides AWS-Lambda [3], an event-driven, serverless computing

platform that enables to implement and deploy application code in any of the supported languages (Python, NodeJS, Java, C#, Go, Ruby, and Powershell), and execute on-demand as docker-containers. The serverless infrastructure manages the queuing of requests and can automatically scale backend service containers to meet fluctuating demands. The elastic nature of serverless computing makes it an ideal match for edge computing data centers, where the small scale of each site places tight restrictions on the number of servers that can be deployed there. Unlike mega-data-centers, the limited capacity of an edge data center requires efficient use of resources with applications only consuming the resources they need. This minimizes waste and fragmentation compared to the traditional approach of deploying fixed size virtual machines with dedicated resources for time periods of hours or days.

Unfortunately, the combination of serverless and edge computing is not as perfect a fit as it initially appears. It is well known that serverless computing can incur high overhead if requests reach a "cold" service, causing delays on the order of hundreds of milliseconds to instantiate a new container. In this paper we focus on a second, less explored challenge: the difficulty of providing reliable processing with a serverless-based infrastructure. Just as people have come to depend on network-based services and cloud computing facilities to be as reliable as their own dedicated computing systems, serverless computing needs to evolve and mature to achieve the same reliability expectations. We believe that the limited reliability semantics of today's serverless platforms (*e.g.,* AWS Lambda and Microsoft Azure functions support only at-least once semantics) poses high risk for emerging CPS and IoT applications, leaving them truly "living on the edge" in terms of the risk of failure and its consequences.

A number of the edge computing applications mentioned above depend on high throughput and low latency processing of streaming applications. While 'traditional' stream processing frameworks are designed for applications with dedicated resources (at least for significant time periods), their functionality is likely to be needed in the serverless context. Moreover, they typically reside on large-scale data centers that are built and managed for being resilient to failures with redundant resources, we see them having to also evolve to be supported on edge data centers. Furthermore, the edge computing applications such as for IoT and CPS as well as even some vehicular applications may have varying reliability requirements. Some, by the nature of disseminating and processing data in an idempotent manner may require little or no reliability guarantees, and 'only' require good responsiveness.

On the other hand, there are many applications, such as vehicular safety, processing medical data etc., that require high reliability, and the environment needs to be robust to failures. Thus, it is useful to understand the costs of providing failure resiliency of these stream processing frameworks so that we can examine their utility for serverless computing in an edge computing environment, and also how this failure resiliency can be configured appropriately on an as-needed basis.

In this paper we systematically evaluate the costs associated with providing different reliability guarantees. We use Apache-Storm/Trident [4] and Apache-Kafka [5] stream processing frameworks to evaluate the throughput, latency, and resource cost associated with request processing ranging from best effort service to exactly once guarantees.

II. BACKGROUND

We look at a canonical serverless architecture and examine its applicability for an Edge Cloud computing platform supporting stream based applications like IoT, data analytics, and (responsive) web applications.

A. Serverless Computing

Serverless computing is a system architecture paradigm that incorporates the "Backend as a Service" (BaaS) *i.e.,* services like storage (object store and/or key-value databases), messaging (push event notifications), user management (authentication, authorization) *etc.* , and "Functions as a Service (FaaS) *i.e.,* provide the capability to deploy and run user-custom code in ephemeral containers on a compute platform. Figure 1 describes the typical architecture of Serverless, where the user requests (Rest API) can be processed by any of the executors (function containers). Thus serverless computing removes the need for a traditional always-on server components.
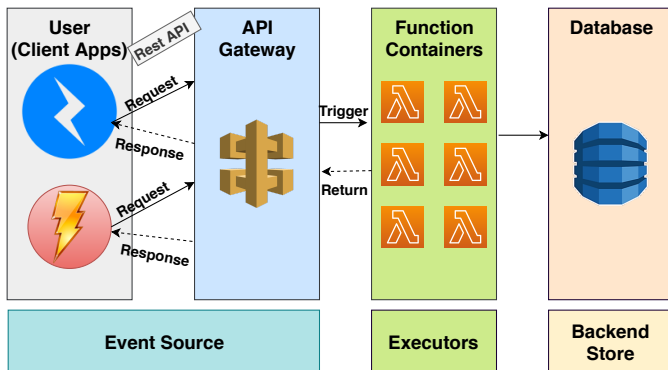

Fig. 1: Serverless Architecture: Users generate requests through the Restful interface. API gateway triggers the execution of functions based on the requested events. Backend database is used to store the configuration and processing state.

Serverless architectures seek to significantly reduce the operational cost and complexity, and greatly improve service scalability and availability [6]. However, supporting low-latency stateful services and providing reliable services with guaranteed stream processing semantics are key challenges affecting the applicability of serverless computing paradigm to many sensitive web applications.

B. Stream Processing

Stream processing, also referred to as event processing, refers to continuous processing of unbounded series of data or events [7]. Stream processing can be described using the directed acyclic graph (DAG), where the edges represent the flow of data/events and the vertices represent the operations/functions that are application-specific logic to process data.

Stream processing allows applications to readily exploit a limited form of parallel processing in-terms of both task parallelism (*e.g.,* Storm) and data parallelism (*e.g.,* Kafka, Spark) [8]. Note that stream processing includes the event-stream processing and reactive programming models that enable processing of functions and generation of new events in reaction to processed data/events, elements which are commonly employed in Serverless computing as well.

Stream processing engines execute the processing graph of streams, and allow injection of events into the processing graph. Users write code to create stream processing operations and chain them together into a processing graph, which can then be run exploiting parallelism in a cluster of computing devices. Stream processing engines tend to be natively parallel and distributed and may span multiple data centers [9]. Thus, stream processing platforms can form the basis of serverless computing. In addition, Stream processing can be either stateless or stateful. In the latter case, the state is usually stored in some external data-store (database), and this externalized state-store concept is extensively employed in serverless architectures to persist state across invocations. Furthermore, works [10], [11] have demonstrated the feasibility of using serverless computing for big-data stream processing engines like Spark and Flint.

C. Stream or Event Processing Reliability Semantics

Different types of failures, *e.g.,* of network links, machines (server nodes) and software process crashes *etc.*, can result in loss of data and inconsistent stream processing across the data center. Stream/event processing platforms provide different reliability modes or the processing semantics for the data stream to provide failure resiliency for streaming applications. Accordingly, there are three different modes (at-most once, at-least once, and exactly once), provided by the streaming platforms as described below:

- **At-most once**: In this approach, sending a message/event from sender to receiver provides no guarantee that a given message will be delivered and executed upon. Any given message may be delivered once or it may not be delivered at all. In a nutshell this is a 'Best-effort' approach where a data loss is possible.
- **At-least once**: This approach ensures that sending a message/event guarantees the delivery of message at least once. However, in the event of anomalies, the message could be delivered more than once. Thus, it is a 'No-loss model' where the messages are guaranteed to be delivered at-least once, and they may be redelivered (possible duplicate that are processed again). This approach requires the sender and receiver to actively participate and coordinate on message request or delivery in the events of failure.

- **Exactly once**: This is a 'No-loss, no-duplicates model' where messages are guaranteed to be delivered only-once. This approach also requires the sender and receiver to actively participate and coordinate on message request or delivery in the event of a failure and additionally they need to provide the transaction semantics for state update and state roll-back for recovery.

D. Evaluation Platforms: Apache Kafka and Storm

We consider several aspects as the key decision factors for different streaming platforms, namely: Reliability, Latency, Scale, and Throughput. Storm [4] is considered to be the highest performance streaming engine with low-latency, while Kakfa [9] is the most widely deployed platform that is leveraged as either a distributed messaging platform in several streaming platforms (*e.g.,* Spark, Storm, Flink, *etc*) or can be deployed entirely as a stream processing platform. Further, Kafka and Storm/Trident are two of the streaming platforms that can support and guarantee exactly-once processing semantics, while most of the other streaming platforms tend to be best-effort which can guarantee either only-once (at-most once) or at-least once processing semantics [12], [13].
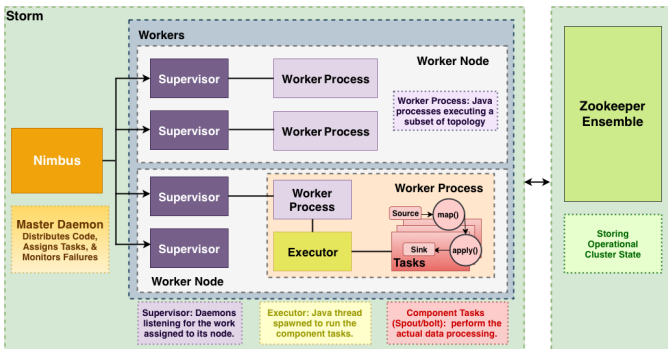

Fig. 2: Architecture of Apache-Storm: A low-latency stream processing platform. Storm cluster constitutes of the Nimbus (master) node and supervisor (worker) nodes. Zookeeper hosts the storm cluster, executor and topology configurations.

III. STREAMING PLATFORMS: STORM AND KAFKA

Today's cloud environments support several stream processing platforms, such as Apache Spark [14], Samza [13], Flink [15], *etc.* We briefly describe the streaming platforms used in this work, Apache Storm [4] and Apache Kafka [5].
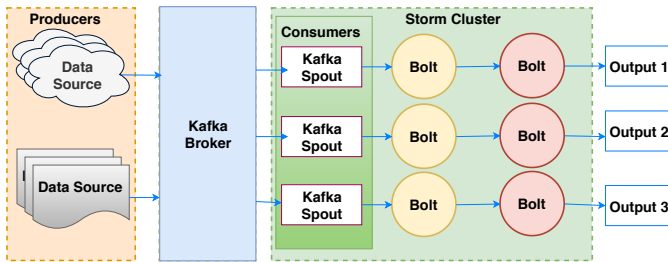

Fig. 3: Storm Topology with Kafka messaging broker. Producers publish the results to Kafka Broker; Storm's Kafka Spouts consume the messages from Kafka and emit to the Bolts, which process on these messages and emit the output.

A. Apache Storm

Apache Storm [4] is an open source distributed real-time stream processing platform. Storm provides a scalable architecture for processing unbounded data streams at scale and provides strong fault tolerance mechanisms. The key components of the Apache Storm stream processing framework are:

- **Tuples**: correspond to a unit of data in the data stream. It is a named list of values, where each value can be of any type and can be dynamically typed *i.e.,* the field types do not need to be declared.
- **Spouts**: represent the source of the data stream. Spouts can read data from different data sources such as a database, distributed file system, messaging framework like Kafka, etc. Spouts generate a stream of tuples that can be fed into a network of bolts (see below) to carry out the required data processing. Spouts can broadly be classified into two types:

1) *Unreliable*: They provide 'at-most once message processing' semantics, and do not have the capability to replay the tuples. Once a tuple is emitted, it can not be replayed irrespective of whether the tuple got processed successfully or not.
2) *Reliable*: These can provide 'at-least once or exactly once message processing' and have the capability to replay tuples in the event of any failures.

- **Bolts**: represent the processing logic or the functions in Storm. Different operations such as filtering, aggregating, joining, *etc.* can be realized with Bolts. They process the tuples and can also anchor and emit a new stream of tuples.

Figure 2 shows an Apache Storm cluster architecture's components. It is made up of two types of processes, namely i) Nimbus and ii) Supervisor. Nimbus is a process running on a master node that is responsible for tracking the progress of data processing while the Supervisor process runs on worker nodes and is responsible for executing the data processing logic. Nimbus is responsible for scheduling, monitoring, and distributing the stream processing tasks. Zookeeper is used to manage the Storm cluster and monitor heartbeats from the workers and supervisors.

Reliability modes: Storm can at-best guarantee that every event will be processed "at-least once". Trident is a high-level abstraction built on top of Storm, and provides the primitives for performing stateful processing through the use of a database or persistence store to provide 'exactly-once' processing semantics. Another notable difference is that unlike Storm which performs tuple-by-tuple processing, Trident processes the stream of tuples in 'mini-batches'. This allows to provide the notion of transaction, by assigning a transactionID for each of the processed batch of tuples. Trident performs `beginCommit()` at the beginning of each batch and once all processing for the batch of tuples completes successfully, the transaction is considered to be successful and Trident will call `Commit()` at the end to update the state. However, in the event of any failure in processing for any of the tuples in the batch, Trident requests for the entire batch to be re-transmitted.

B. Apache Kafka

Kafka, as a distributed messaging system, is one of the common component included in other stream processing engines like Storm [4], Spark [14], *etc.* for scalability purposes. Figure 3 shows the usage of Kafka as a distributed messaging broker in the Storm streaming platform. This enables the persistence of messages in the Kafka partitioned topics (can be across multiple nodes in the case of cluster) and improves on the reliability of processing the messages in the event of any failures with Storm cluster. There are three key concepts in Kafka as described below:

- **Record & Topic**: Kafka uses a cluster of servers to store streams of records, and each record has a key, a value, and a timestamp. These records are classified into topics.
- **Producer**: publishes the records to one or more topics. From Kafka 0.11 release on-wards, the producer can be configured to function in an idempotent and transactional mode, which strengthens the delivery semantics from providing at-least once to exactly once semantics.
- **Consumer**: subscribes to different topics of their interest and receive the records produced in them. Each consumer belongs to a 'consumer group' and cooperatively works with other consumers to achieve load balancing and resiliency.

Reliability modes: Kafka supports both 'at-least once' and exactly-once' delivery semantics. Kafka's exactly once semantics is built on top of idempotency and transaction features. Idempotency is achieved by using a sequence number to de-duplicate the duplicate (multiple) writes, and transaction allows for atomic writes across multiple topics. These two features together offer end-to-end 'exactly-once' guarantees for Kafka Streams. Also, for the stream processing applications like Storm, Spark, that extend from Kafka can ensure exactly-once provided the streams can rewind the offsets and rollback the corresponding external state updates.

IV. EVALUATION

We describe our evaluation testbed and demonstrate the overheads incurred for supporting different reliability semantics with two different streaming platforms Apache Storm and Apache Kafka.

A. Evaluation setup

1) Hardware: Our experimental testbed used the Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz dual-socket server with 20 cores each with hyper-threading enabled and 252GB RAM, running Ubuntu SMP Linux kernel 4.4.0-142-generic.

2) Software: We deployed the following software packages: i) Apache-Storm ver. 1.2.2, ii) Zookeeper ver. 3.4.13 and iii) Apache Kafka ver. 2.1.0.

3) Workload: Due to the lack of standard benchmarks for real-world streaming platforms, we make use of the available benchmark tools deployed in earlier works [8], [12] and extend them to fit our evaluation needs. We evaluated for both reliable and unreliable stream processing for the three different reliability semantics discussed in Section II-C. The evaluation platform configurations for the three reliability variants for message processing are shown in Table I.

Reliability Semantics	Test Platforms & Topology Description
at-most once (Unreliable)	Apache Storm without Kafka Broker Storm Spouts without acknowledgement (ack)
at-least once (Reliable)	i) Apache Storm + Kafka Broker, ii) Kafka Streams Kafka producer & Storm Spouts with ack
exactly-once (Reliable)	i) Trident + Kafka Broker, ii) Kafka Streams Kafka producer & Trident Spouts with ack

TABLE I: Streaming platforms used for evaluation.

B. Analysis with Trident/Storm

For Apache Storm and Trident experiments, we extend Intel's Storm-benchmark package [16] and implemented variants of WordCount topology, which provides a good measure of the overhead introduced by Storm as the Spouts and Bolts in this topology do minimal work [12]. This WordCount topology demonstrates a very simple streaming example consisting of spouts and two bolts. Spouts read sentences from a given source file (input stream of data) and emit them to the 'Split' bolts. The split bolts in turn split these sentences into words and send to the 'Count' bolt where the count of words is updated.

We configure the Storm, Trident stream and topology parameters as shown in the Table II. Also, we configure the Kafka stream producers to use the same number (4) of producer spouts. In order to keep the replication overhead minimal, we set the Kafka message replication factor to 1.

Configuration parameter	Value
topology.workers	4
topology.acker.executors	4
topology.max.spout.pending	200
component.spout_num	4
component.split_bolt_num	8
component.count_bolt_num	8

TABLE II: WordCount Topology configurations used for the Storm and Trident experiments.

Impact on Throughput and Latency: Figure 4 compares the throughput (*i.e.,* the number of messages processed per second) for different reliability modes (note the log scale for the Y-axis). We observe that the at-most once processing mode provides ∼850K messages per second (mps), while the throughput drops by 4X with at-least once semantics resulting in ∼150-200Kmps. Further, exactly-once semantics drastically degrades the throughput even more, providing at-best ∼85Kmps and intermittently not processing any messages for brief periods. The corresponding impact on latency is shown in Figure 5. We observe an average processing latency of 5ms for both at-most and at-least once semantics, while there is an order of magnitude increase in latency ∼8X-20X for the exactly-once (transactional) mode[1]

Impact on Resource utilization: We also collect the CPU and disk usage statistics (averaged over a 10 seconds interval) using the 'dstat' tool [17]. Figure 6 shows the CPU utilization observed across the three modes. we can observe that the CPU utilization increases by ∼4X for both at-least and exactly-once semantics in comparison to the at-most once mode. We also

---

[1]We only plot the first 10 minutes of the results, which indicate that the latency levels off. But, we observed that latency rises back up intermittently to 100ms when the tests were run for a longer interval (30+ minutes).
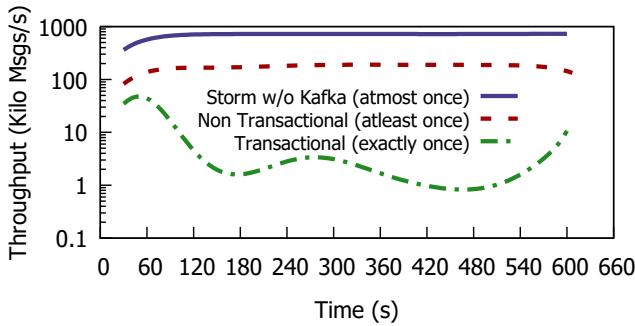
Fig. 4: Message processing throughput for different messaging semantics with Storm and Trident.
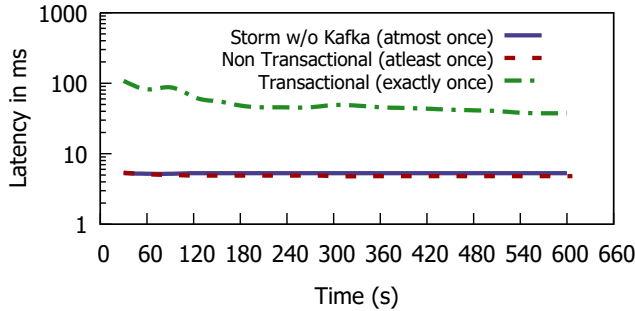

Fig. 5: Latency (in milliseconds) for different reliability semantics with Storm and Trident.

profile the disk read and write usage statistics and observe ~10X increase in the disk usage for the at-least and exactly-once semantic experiments as shown in Figure 7. We noticed most of disk usage to be due to the disk writes performed by the Kafka producer and commit state updates by Trident. Note, the CPU and disk usage per work done (throughput) is far worse (2X-20X) for exactly-once mode than the at-least once mode.
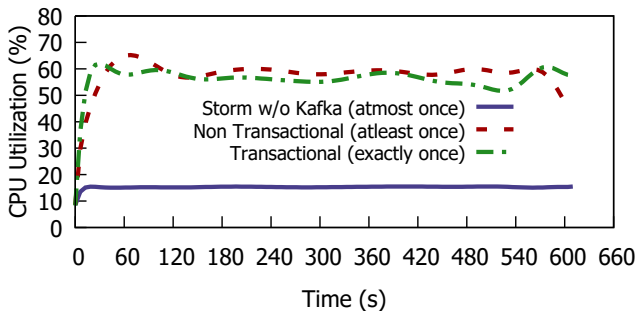

Fig. 6: Avg. CPU usage for different reliability semantics.

C. Analysis with Kafka Streams

We measure the performance of Kafka by running the Word-Count streaming application. For these experiments, we switch the producer into the non-transactional mode and transactional mode with a commit interval of 1ms, 10ms, and 100ms. As shown in Figure 8, compared to the nontransactional mode, a large 100ms commit interval makes the throughput drop by 2X. But, even more importantly, a small 1ms commit interval
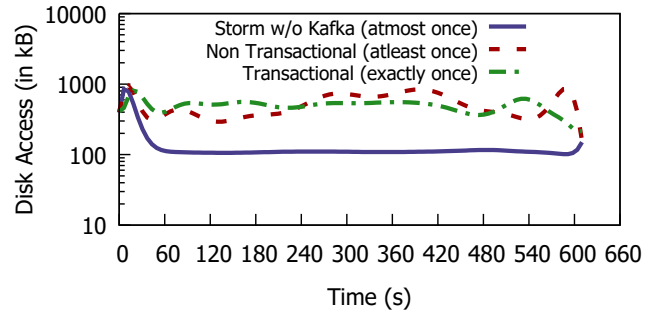

Fig. 7: Avg. Disk usage (bytes written) for different reliability semantics with Storm and Trident.

leads to 58X degradation for 64B records and 11X degradation for 1024B records. To evaluate the latency overheads, we throttle the sending rate to 1MB/s to avoid large, unnecessary queuing latency. As shown in Figure 9, a 100ms commit interval has an average 3.3ms latency, but a 1ms commit interval incurs an average 21ms latency which is 12X and 83X larger than the non-transactional mode.
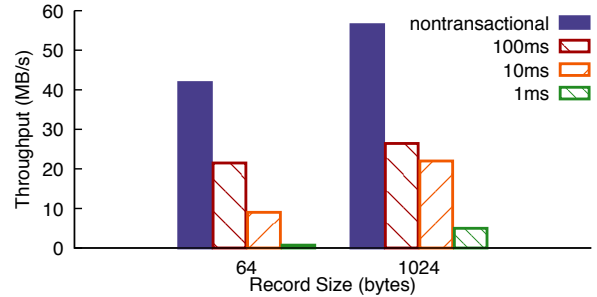

Fig. 8: Throughput for Kafka with non-transactional and transactional models with different commit intervals
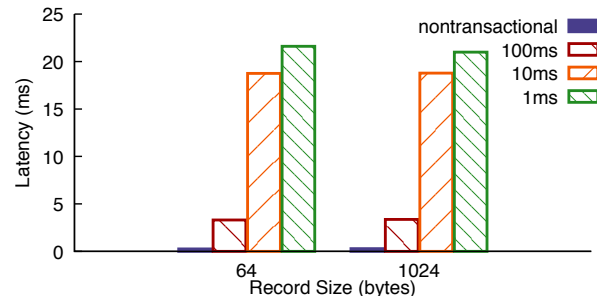

Fig. 9: Latency for Kafka with non-transactional and transactional models with different commit intervals

To summarize, these non-negligible overheads pose challenges when using the streaming platforms like Storm, Kafka for emerging edge and serverless based applications that have strict reliability, correctness and latency requirements, and requires us to re-examine existing solutions.

V. RELATED WORK

Serverless and Edge Computing: Previous works have explored various problems associated with serverless computing.

Notably, with the focus on large-scale data centers, [18], [19], [20], [21] have studied issues pertaining to the pricing model, programming model, autoscaling, and startup delay associated with the serverless computing. Some of the projects [10], [11] have looked at how to apply the serverless model into big data stream processing engines. However, the latency and reliability issues of using the serverless computing model at the edge haven't yet drawn much attention.

Streaming Engine Analysis: Works [8], [22], [23], [24] have compared different stream processing platforms. However, they focus primarily on the comparison of supported features and evaluate throughput, latency through tests over different distances to the data center. In contrast, our work focuses on the overheads and impact on throughput, latency and resource utilization for providing different reliability semantics for stream processing.

VI. CONCLUSION AND FUTURE WORK

To summarize, in this work, we have looked at Serverless Computing - an emerging paradigm replacing many of the stream/event processing based web applications. We highlighted the difficulty of current Serverless applications in achieving reliable event/data processing.

We evaluated the overheads associated with providing 'exactly-once' reliable message processing semantics with the state-of-the-art streaming platforms like Apache Storm/Trident and Apache Kafka. Because of the significant overheads, there is a large performance degradation both in terms of latency (increases almost 20X-80X), and throughput (drops by 4X-50X). In addition, the resource overheads, specifically the CPU and disk usage, are around 4X-10X increase. Hence, we reason that the direct adoption of such approaches with serverless computing at the edge may be problematic, bordering on being infeasible. These mechanisms as implemented would greatly hinder the efficient operation of edge computing resources. We believe it is necessary to revisit these mechanisms for providing reliable processing and failure resiliency approaches for edge computing.

REFERENCES

[1] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computinga key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[2] P. Schulz, M. Matthe, H. Klessig, M. Simsek, G. Fettweis, J. Ansari, S. A. Ashraf, B. Almeroth, J. Voigt, I. Riedel *et al.*, "Latency critical iot applications in 5g: Perspective on the design of radio interface and network architecture," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 70–78, 2017.

[3] Amazon, "Aws lambda," https://aws.amazon.com/lambda, [ONLINE]. [Online]. Available: https://aws.amazon.com/lambda/

[4] "Apache storm," http://storm.apache.org, [ONLINE]. [Online]. Available: http://storm.apache.org

[5] N. Garg, *Apache Kafka*. Packt Publishing, 2013.

[6] F. Martin, "Serverless architectures," url=https://martinfowler.com/articles/serverless.html, May 2018.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable distributed stream processing." in *CIDR*, vol. 3, 2003, pp. 257–268.

[8] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1789–1792.

[9] "Apache kafka," https://kafka.apache.org, [ONLINE]. [Online]. Available: https://kafka.apache.org

[10] G. Owen, E. Liang, P. Chockalingam, and S. Shankar, "Databricks serverless: Next generation resource management for apache spark," 2017. [Online]. Available: https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html

[11] Y. Kim and J. Lin, "Serverless data analytics with flint," *arXiv preprint arXiv:1803.06354*, 2018.

[12] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 239–250. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742788

[13] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, pp. 1634–1645, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137770

[14] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[15] E. Friedman and K. Tzoumas, *Introduction to Apache Flink: Stream Processing for Real Time and Beyond*, 1st ed. O'Reilly Media, Inc., 2016.

[16] Z. Manu and Z. Sean, "Storm benchmark," url=https://github.com/intel-hadoop/storm-benchmark, 2016.

[17] D. Wieers, "Dstat: Versatile resource statistics tool," 2016. [Online]. Available: http://dag.wiee.rs/home-made/dstat/

[18] Z. Al-Ali, S. Goodarzy, E. Hunter, S. Ha, R. Han, E. Keller, and E. Rozner, "Making serverless computing more serverless," in *11th IEEE International Conference on Cloud Computing, CLOUD 2018, San Francisco, CA, USA, July 2-7, 2018*, 2018, pp. 456–459.

[19] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," in *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[20] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[21] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "{SOCK}: Rapid task provisioning with serverless-optimized containers," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 57–70.

[22] A. Carr and A. Aspell-Clark, "Comparing apache spark, storm, flink and samza stream processing engines - part 1," 2018. [Online]. Available: https://blog.scottlogic.com/2018/07/06/comparing-streaming-frameworks-pt1.html

[23] M. A. Lopez, A. G. P. Lobato, and O. C. M. Duarte, "A performance comparison of open-source stream processing platforms," in *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2016, pp. 1–6.

[24] S. Qian, G. Wu, J. Huang, and T. Das, "Benchmarking modern distributed streaming platforms," in *2016 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 2016, pp. 592–598.