THE INTERSECTION OF FUNCTION-AS-A-SERVICE AND STREAM COMPUTING

An Undergraduate Research Scholars Thesis

by

TREVOR BOLTON

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dilma Da Silva

May 2021

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Trevor Bolton, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

Page

ABSTRACT 1

DEDICATION 3

ACKNOWLEDGEMENTS 4

NOMENCLATURE 5

SECTION

1. INTRODUCTION 6

   1.1 Dataflow Programming 6
   1.2 Cloud Computing 9

2. METHODS 13

   2.1 Shortcomings of FaaS for Stream Processing 13
   2.2 Building Apache Kafka Streams versus AWS Lambda 15

3. RESULTS 20

4. CONCLUSION 22

REFERENCES 23

ABSTRACT

The Intersection of Function-as-a-Service and Stream Computing

Trevor Bolton
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dilma Da Silva
Department of Computer Science and Engineering
Texas A&M University

With recent advancements in the field of computing including the emergence of cloud computing, the consumption and accessibility of computational resources have increased drastically. Although there have been significant movements towards more sustainable computing, there are many more steps to be taken to decrease the amount of energy consumed and greenhouse gases released from the computing sector.

Historically, the switch from on-premises computing to cloud computing has led to less energy consumption through the design of efficient data centers. By releasing direct control of the hardware that their software is run on, an organization can also increase efficiency and reduce costs. A new development in cloud computing has been serverless computing. Even though the term "serverless" is a misnomer because all applications are still executed on servers, serverless lets an organization resign another level of control, managing instances of virtual machines, to their cloud provider in order to reduce their cost. The cloud provider then provisions resources on-demand enabling less idle time. This reduction of idle time is a direct reduction of computing resources used, therefore resulting in a decrease in energy consumption.

One form of serverless computing, Function-as-a-Service (FaaS), may have a promising future replacing some stream computing applications in order to increase efficiency and reduce waste. To explore these possibilities, the development of a stream processing application using traditional methods through Kafka Streams and FaaS through AWS Lambda was completed in order to demonstrate that FaaS can be used for stateless stream processing.

DEDICATION

To my friends, family, instructors, and peers who supported me throughout the research process.

ACKNOWLEDGEMENTS

Contributors

I would like to thank my faculty advisor, Dr. Da Silva, for her guidance and support throughout the course of this research.

Thanks also go to my friends and colleagues and the department faculty and staff for making my time at Texas A&M University a great experience.

Finally, thanks to my friends and family for their encouragement and to my significant other for her patience and love.

The cloud computing resources used for the Intersection of Function-as-a-Service and Stream Computing were provided by Dr. Da Silva.

All work conducted for the thesis was completed by myself independently.

Funding Sources

Undergraduate research was supported by the College of Engineering at Texas A&M University by enabling Dr. Da Silva to hire me as her research assistant.

This work was also made possible in part by the National Science Foundation under Grant Number NSF-SPX-1919181. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the National Science Foundation.

NOMENCLATURE

API	Application Programmer Interface
AWS	Amazon Web Services
Azure	Microsoft Azure
BaaS	Backend-as-a-Service
CAPEX	Capital Expenditures
EC2	Amazon Elastic Compute Cloud
FaaS	Function-as-a-Service
IP	Internet Protocol
OPEX	Operational Expenditures
PaaS	Platform-as-a-Service
uuid	Universally unique identifier
VPC	Virtual Private Cloud

1. INTRODUCTION

Big Data has been a growing interest for organizations to leverage helpful insights from large quantities of collected data. With advancements in Machine Learning and Artificial Intelligence, taking this data and transforming it into value for the organization have become increasingly easier. This has led to the prominence and increased collection of data. Recent studies [1, pp. 98–115], [2], [3] have discussed the exponential growth of data collection and the importance of extracting value through computationally heavy processes. This increase in necessary computational resources should be a signal that the ways in which we process this data must be analyzed and optimized to ensure sustainability and feasibility. In this work, I investigate how current streaming applications can be replaced by Function-as-a-Service (FaaS), which has the potential to be a more sustainable solution. First, we will look at some history of how data has been processed.

1.1 Dataflow Programming

Parallel computing is the simultaneous use of computational resources in order to increase efficiency and decrease computation time. Dataflow Programming is a programming paradigm derived from parallel computing, in which the user designs a directed acyclic graph of operations to transform the data as it passes through. One could think of this as designing an assembly line where items must go through distinct stages of processing. Each of these stages is called either an operator or function that computes on an incoming stream or batch of data. Dataflow programming is a useful paradigm for processing large quantities of data that is continuously generated and manipulated through clearly defined operations. Further descriptions of specific dataflow types follow in the next subsections.

1.1.1 Batch Processing

The simplest way to process any data would be to gather all of it ahead of time, then feed that information into some computer program that will analyze and extract value from that data. This is where batch processing originates. Data is gathered into one batch that is then processed all at once. Once the processing starts, one must wait for it to run to completion to retrieve your results. For example, in an account management system where transactions happen throughout the day, the system would hold all the transactions until the end of the day. Only then the system would generate a final balance for the day. All the transaction records would be processed in order, exposing any overdrafts that occurred throughout the day, even if the day still ended with a positive balance.

The power of batch processing lies in its simplicity and its potential efficiency. There have been many advancements in batch processing that result in powerful frameworks that developers can utilize to quickly and correctly process large amounts of data utilizing a distributed system, such as Hadoop and Apache Spark. These are industry standard software that are run on computer clusters and have the ability to recover from any type of hardware failure. Once a framework is setup on a cluster of computational resources, it can be fed extremely large quantities of data that will then be processed in parallel to decrease the processing time. This type of computing is very effective when you have large, finite datasets that need to be analyzed. Although, this leads to a question that batch processing cannot answer: what if your data is being continuously generated, i.e., not finite?

1.1.2 Stream Processing

In order to answer this question, engineers and computer scientists developed stream processing. A stream is a theoretically unbound, ordered collection of data elements that can be consumed from data sources or produced to data sinks. These elements can be made available at any time and then be consumed soon after. Since the stream is not a bound set of data, the functions that are being applied, must either be applied to a single instance of a data element or a small local group of data. For example, one could not find the total average of a stream, but one could find the moving average of the last 20 elements received to infer something about the current data.

To understand this form of Dataflow programming, recall the example of managing account balances. With traditional processing, all the transactions were collected into a single batch at the end of the day, then processed. One problem such an approach is that the actual real-time balance of an account is unavailable except for right after the daily batches have been processed. Account management where up-to-date balances are required is clearly a stream processing problem and should not be handled using batch processing since it is a time-dependent application of continuously generated account transactions. Instead, we can define a few operators upon a stream of account transactions to solve the problem. First, there must be a way to fetch and publish the current balance. This can be in some database or handled by some other stream/service. Next, there will be a stream named “transactions” that will be populated with transaction data whenever they occur (think swiping a credit card or cashing a check). Finally, there needs to be an operator that consumes these transactions and updates the balances in real-time. This more accurately mimics how customers imagine an account to be managed: transactions are processed immediately, and balances are kept up to date with the most recent information.

Stream processing also typically builds in redundancy to ensure that these systems will be as reliable as batch processing. The real-time aspect adds complexity but brings the system to be more continuous and more reflective of the real world. Stream processing was an important development in dataflow programming in order to allow organizations to develop more complex and responsive systems. Some examples of stream processing in modern computing are fraud detection, analytics, and managing smart systems.

Many research projects investigate streaming programming models [4, 5] and streaming programming languages [6, 7] in the past. With the rise of big data, there has been a resurgence of interest in streaming applications in the last decade.

1.2 Cloud Computing

Within the last two decades, there has been a revolution in computing: the creation of “the cloud”. Pioneered by AWS in 2006, cloud computing began as a set of three computer resource infrastructure pieces: databases, storage, and compute. The idea behind this service was to free engineers and organizations from worrying about not having access to these services or maxing out their computational capacity. Cloud computing is a technique where IT services, such as databases and servers, are provided by massive low-cost computing units connected to the internet protocol network layer [8, pp. 626–631, 9].

The development of cloud providers has allowed organizations to shift away from large IT teams supporting their computer infrastructure to directing funds towards their major goals. These services can be significantly cheaper, more efficient, and easier to manage than hosting these services on-premises. Another benefit of cloud computing is scalability. In the past, if an organization were nearing capacity or reached its limit, it would be required to purchase and install more physical hardware into its infrastructure requiring more capital expenditure (CAPEX). On the other hand, if their requirements were below their capacity, it is unlikely that they could return their unused hardware, therefore wasting CAPEX. Cloud computing addresses both problems by providing as many computing resources as the organization needs through pay-per-usage operational expenditures (OPEX). If there are not enough resources, they can rent more from their cloud provider with the click of a button or sometimes even automatically. If they have an excess of resources, they can stop renting certain services in order to decrease their costs. By shifting costs from CAPEX to lower OPEX, organizations can remain more agile and require less startup capital.

Cloud providers can provide these services to their customers by leveraging the five major technical characteristics of cloud computing and the benefits from economy of scale. These five characteristics are large-scale data centers, shared resource pool (both virtualized and physical resources), dynamic resource scheduling, high scalability and elasticity, and general-purpose usage. Cloud providers manage large-scale computing resources by owning and maintaining a large quantity of customized low-power hardware. These resources can also be deployed on satellite locations, such as datacenters near solar or wind farms, to reduce the cost of electricity. This hardware contributes to the shared resource pool which is further optimized using virtualization, or the replication of systems creating a virtual representation. The sharing typically performed by virtualizing the operating system in order to provide the illusion of exclusivity. Many individuals or organizations may be utilizing the same hardware. Resources are distributed to their customers using dynamic resource scheduling designed by the cloud provider to reduce resource waste. Due to these capabilities, a cloud provider can grant highly scalable, general-purpose computation resources to their customers. Cloud providers are further able to decrease costs by utilizing the economy of scale, in which they can increase their efficiency by aggregating their resources and allocating them efficiently to many customers.

The transition from on-premises computing or private data centers to cloud computing has not only decreased the cost, but also decreased the environmental impact of the organization by greatly reducing waste and outsourcing to more energy-efficient locations. This is not a one-time improvement, like buying more energy-efficient computers. Cloud providers are constantly working to make their systems more efficient and reduce energy costs. A reduction in energy consumption by the cloud provider is also beneficial for the environment. By consolidating and aggregating computing resources into a cloud provider, which is greatly incentivized to reduce their energy consumption in order to increase profits, the environmental impacts of computing should become more sustainable. For example, Microsoft has a plan to become carbon negative by 2030 [10].

1.2.1 Serverless, the Future of the Cloud

Even though original cloud computing relieved organizations of physical infrastructure management and IT services, it still left them with an increase of virtual resources to manage. Initially, this was advantageous as organizations could easily port their enterprise applications to equivalent machines on the cloud without needing to rebuild applications. As cloud computing progressed, many organizations started building "cloud-first" applications that rely on serverless computing. Serverless computing is a subset of cloud computing in which all resources are scaled automatically with no need for explicit provisioning and are billed on usage [11].

Serverless will likely be the future for many cloud applications, in particular for designs based on the micro-service architecture [12]. Just like how cloud computing reduced the need for managing an infrastructure IT team, serverless computing will reduce the need for system administrators since the cloud provider will handle this for them. Similarly, this new service allows for an even more fine-grain level of control for the cloud provider, allowing them to optimize away waste even more. This efficiency, again, may benefit the customer by making serverless even less expensive than traditional cloud computing. This is a trend similar to the transfer from on-premises to cloud computing, and we will likely see a similar impact on the computing industry as more organizations adopt “cloud-first” and serverless applications. There are two primary forms of serverless which are Backend-as-a-Service (BaaS) and Function-as-a-Service (FaaS). For the scope of this research, the focus is on FaaS.

1.2.2 Function-as-a-Service

Function-as-a-Service, or commonly shortened to FaaS, is a category of serverless computing that provides a system with the capability to activate some functionality on demand. This breaks down the cloud computing paradigm further by not requiring a single server instance to remain running to accept requests, as with previous types of cloud computing services such as Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS). Function-as-a-Service limits the scope of the application to individual functions that can be invoked or triggered by individual events. Use cases for FaaS include creating Application Programming Interfaces (APIs) services or stateless data processing. As software development evolves, there might be a trend towards utilizing FaaS more often since it offers simplicity and affordability. Another benefit of FaaS is the flexibility and automatic management of scalability. Any application can scale up and down to the demand at any time. For example, if an application suddenly spikes in usage, the cloud provider will handle allocating more Virtual Private Server (VPS) instances to the application. There can be large downsides to the usage of FaaS, such as increased latency requests due to runtime preparation (called “cold starts”) or the difficulty in integrating it with stateful systems, but as more organizations adopt FaaS, these problems are likely to be addressed by cloud providers through improvements in their FaaS implementations.

2. METHODS

After introducing the necessary background information, this document discusses how Function-as-a-Service can be applied to Stream Computing and the current shortcomings of this application. We also describe how the research could benchmark these applications to evaluate their viability of implementing streaming applications through FaaS.

Function-as-a-Service is a promising prospect in augmenting Stream Computing due to their structural similarities. By helping to reduce the complexity of building such applications, FaaS can bring to streaming computing the benefits of native elastic scaling, simplified deployment, and a reduction in cost.

2.1 Shortcomings of FaaS for Stream Processing

Before investigating use cases and possible benchmarking methodologies, the research cataloged known shortcomings and limitations of the application of FaaS to Stream Computing.

There are two significant problems with FaaS that limit its compatibility with streaming applications: the stateless nature of FaaS and the lack of guarantees in preserving the order of events. These are definitive downsides that must be considered when trying to use FaaS for this type of application. Depending on the specifications for the application, an organization may accept these limitations in order to achieve the benefits offered by a FaaS-based implementation.

2.1.1 The Stateless Problem

A stateless process is one that does not store knowledge about past transactions, processing each individual request in isolation, not dependent on other requests in any way. The function takes a given input and returns a corresponding output. This is contrary to the nature of stream computing which relies on locality of data, e.g., the context withing a window of time.

Since a stream of data is an ordered collection of data produced over time, many of its algorithms rely on storing information about recently occurring events. For example, the rolling average algorithm uses a sliding window to calculate the average of some data flowing through a stream. The last *N* elements will be summed together and divided by N, then that average would be published to another stream. When one value is added to the stream, the oldest value in the window is removed, and replaced with the new value. This algorithm would be required to save some state about recent events in order to do this calculation.

These incongruencies are sometimes solved by adding stateful features to FaaS by utilizing other cloud services. For example, one may add state to AWS Lambda [13] by using Amazon DynamoDB [14] for key-value pair storage. This approach has its own flaws: it can lead to vendor lock, or a reliance on features of a specific cloud provider. It can also reduce performance by requiring a remote call to fetch the state. One solution to the stateless nature of FaaS is building a stateful FaaS platform like Cloudburst [15] which combines the function services and key-value storage onto the same platform. This result suggests the feasibility of creating a stateful FaaS system without introducing prohibitive latency for these data accesses. This could be a very promising step to realizing streaming applications with FaaS.

2.1.2 The Stream Correctness Problem

The other significant problem of the application of FaaS to Stream Computing is the challenge in preserving stream correctness. When utilizing a traditional stream computing platform, there is a guarantee of preserving the order of events within a stream. This is important for reasons discussed in the previous section. Even when multiple instances of an operator are running to process a higher workload, the streaming platform manages each event and ensures that they are inserted into the stream data structure in the proper order. This type of operation is not managed in a FaaS platform. Even though FaaS platforms guarantee a response for each request, the order in which these responses are given is not guaranteed to be the same order in which the requests were issued. This is due to how FaaS schedules these function calls. In order to maximize the throughput and efficiency of these systems, they may sacrifice order preservation. This is one clear challenge in the application of FaaS to Stream Computing.

2.2 Building Apache Kafka Streams versus AWS Lambda

This section explores the pros and cons of building a streaming application on a FaaS platform as opposed to Stream Processing platform. As a base, this work uses Apache Kafka as the message broker for both the Stream Processing and FaaS applications to ensure an equivalent environment and interface for these applications. These processes can then be run on AWS EC2 instances or in a local development environment.

2.2.1 Test Application: PrimeStreamApp

In order to assess the differences in building streaming applications in the two versions, we chose a basic application to implement in the two different methods. The functionality offered by this application is testing the primality of a number. This is an interesting application because for most input values, the processing is quick, but with certain inputs (such as large primes or products of large primes) the task is computationally intensive. Another benefit of experimenting with this application is that generating input data is very simple.

This application first does a simple transformation on a “numbers-input” stream to label the number as prime or not. Then the application splits the stream into “prime-numbers-output” and “composite-numbers-output” streams. This application does not necessarily have any practical uses, but could be used as a component of cryptography application that heavily rely on identifying prime numbers. The code for all three implementations of the application can be found in the Stream Processing vs. FaaS GitHub repository [16].

2.2.2 Implementing PrimeStreamApp in Apache Kafka Streams using Java

The first application implementation utilized the Kafka Streams API [17], which is the programming interface for developing stream processing applications on a Kafka broker. First, I started with the tutorial application from the Kafka Streams documentation [18]. This helped by creating a Maven [19] project to automatically compile the streams application.

Next, was the definition of a data format in the PrimeData.java file. In order to keep this project simple, one data structure was used for the entire project. This class contains several member variables: `uuid`, `timestampCreatedAt`, `timestampProcessStart`, `timestampProcessEnd`, `number`, and `isPrime`. The `uuid` is a unique identifier for each record in the stream and is also used as the key in the Kafka broker. Each of the timestamps designates a point in time of the processing of each element, for ease of benchmarking. The number is the actual payload of each datum that is tested for primality. Finally, we have a boolean for noting if the number is prime after the primality check.

For the Kafka broker to transfer messages with complex data, a data serializer and deserializer must be implemented. I tried three different approaches for this: serialize the object into a byte array, serialize the object using the Apache Avro [20] format, and serialize the object into a JSON [21] string then pass that value into the basic string serializer. These operations were mirrored for implementing a deserializer as well. The first two approaches achieve the greatest object compression; however, they did not work with AWS Lambda’s JSON-based events. For this reason, I chose to use the least efficient method of serializing to a JSON string. This is much more inefficient because the data is sent with the labels. Finding a way that is guaranteed to work over several different systems is challenging. In order to reduce complexity and allow for cross platform-support, I chose the last serializer.

Finally, I defined the topology of the stream processing application. In order to do this, Kafka Streams has its own Domain Specific Language (DSL) [22] to declare how streams should be processed. For my application, I used two stateless stream transformations that I could also implement in FaaS: `mapValue` and `branch`. The `mapValue` transformation passes the value of each event into a function, resulting in the output of a new event. The `branch` transformation is an operator that allows the stream to be split into one or more streams based on the supplied predicates. I combined these two operators to first `mapValue` from each incoming event to an identical event with `isPrime` set to true if the number is prime, then these events are then branched depending on whether the number is prime or not. The result of this topology is a "numbers-input" stream coming in and "composite-numbers-output" and "prime-numbers-output" streams as the result.

I compiled the code as an uber-jar file containing all the dependencies using Maven (mvn package). This resulting jar file can then be run on any system that has access to the Kafka broker. Users must run the main function in the `tbolton.PrimeStreamApp` class and make sure to pass the IP address of the Kafka broker as a command-line argument. Additionally, this jar file contains a simple producer that can be used to create `PrimeData` events into the corresponding Kafka broker on topic "numbers-input".

2.2.3 Implementing PrimeStreamApp in AWS Lambda using Java

After completing the PrimeStreamApp in Kafka Streams, the next step in this work was to implement PrimeStreamApp in AWS Lambda still using Java. This proved to be much more challenging than expected. Again, I started with a tutorial application from the AWS Lambda developer guide [23]. This project also uses Maven to compile and has some code structure for writing a Lambda event handler function. One big hurdle with using Lambda and Java is handling the JSON data and record datatypes. The code for writing Lambda functions in Java is quite complicated and adding the Kafka events and extra PrimeData class lead me to search for a better approach than writing the Lambda functions in Java.

2.2.4 Implementing PrimeStreamApp in AWS Lambda using Node.js

After identifying the difficulties in working in Java, I decided to explore a language with better support for the JSON data structure that is also supported by Lambda functions. I chose to use Node.js. Writing lambda functions in Node.js is simpler than in Java. This is evident in the starter code for each language. The blank-java [23] starter project is over 65 lines of code with several external dependencies including Gson [24] and slf4j [25], while the starter code in Node.js is 4 lines of code with no dependencies. Node.js was a much more appropriate tool for this work.

After a little preprocessing for the Lambda KafkaEvent, to transform the small batches of data into an array of key-value pairs, I converted these arrays into my own custom “Stream” data structure. In order to make mimic the operators for Kafka Streams DSL, I chose to implement the two features that I used in Streams DSL in my separate Stream class. I implemented the `mapValues`, `branch`, and `to` operators with practically the same interfaces that are used in the DSL. These simple functions were easy to write, and one could add the rest of the stateless DSL functions with relative ease to the Stream class. I then replicated the DSL code from PrimeStreamApp in Kafka Streams with some small modifications. Finally, using KafkaJS [26], the Node.js Lambda function can connect back to the Kafka broker to publish the results from the stream processing.

To run this application as a Lambda function, I simply ensured any necessary dependencies were installed using the Node Package Manager [27], compressed the file as a .zip, and uploaded the zip file to AWS Lambda. Using Node.js was the simplest, fastest, and most user-friendly language for developing the PrimeStreamApp.

In order to invoke functions on changes from a Kafka topic, there needs to be a Kafka broker that AWS Lambda can monitor for changes. I attempted to host a self-managed Kafka cluster [28] on AWS VPC [29] but was never able to connect this to AWS Lambda or another EC2 instance on the VPC to run the Kafka Streams application for benchmarking. In the future, using the Amazon MSK [30], might be an easier approach for configuring a Kafka cluster on a cloud platform.

3. RESULTS

Without the use of external storage, stateful stream processing through FaaS was impossible. Since FaaS invocations are by nature stateless, it was impossible to recreate a stateful operation without relying on external storage, such as Amazon DynamoDB or a FaaS platform that supports state management, like Cloudburst. If using the former, vendor lock-in is inevitable, which can greatly limit an organization's options. For the latter, no large-scale stateful FaaS platforms currently exist.

The implementation of stateless stream processing with FaaS was possible. With the PrimeStreamApp, I showed that multiple common stateless operators can easily be implemented in a FaaS application. Further work may be done here for fully implementing every stateless DSL transformation in FaaS.

When it came to development, there were pros and cons to each option. By using Apache Kafka Streams and the Streams DSL, connecting to the Kafka Broker was extremely simple, and the stream processing functionality is already implemented. This was balanced out by the verbose nature and non-compatibility with the JSON format of Java. This application was scalable by running the program on multiple machines but was not elastic.

Using Java to write Lambda functions was not justifiable, since all the stream processing functionality would have to be developed. This option also inherited the faults of the Kafka Streams application since they both use Java. I found that this option was the least desirable for stream processing.

Finally, using Node.js to write Lambda functions was surprisingly successful. Due to the compatibility with the JSON format that was used through AWS Lambda and the simple package manager and deployment scheme for Node.js functions, implementing stateless stream processing with this system was easily accomplished. By moving to a higher-level of abstraction programming language, writing Lambda functions was much more accessible to someone without extensive cloud computing or systems engineering experience.

4. CONCLUSION

As computing moves to higher levels of abstraction, it is imperative that software developers learn how to best use the tools given to them. The move away from classical stream computing to a FaaS-based stream computing will likely lead to changes in how software is created and what elements must be considered. This move will also provide other benefits such as decreased cost, increased efficiency, and faster development. The move to serverless computing seems to be inevitable, and we should embrace it and experiment with ways to utilize it and improve it.

Through developing a Kafka Streams application and replicating it using AWS Lambda functions written in Node.js, I showed that FaaS can replace some stateless stream processing applications. Even though many real-world streaming applications are not stateless, stateless parts of these applications can be rewritten as FaaS functions in order to reap the benefits of serverless computing.

There are many unexplored ideas that may make replacing current stream computing applications more viable. Experimentation with the overhead of managing state from external systems or stateful FaaS platforms may lead to the discovery of novel ways to achieve stateful stream transformations are possible on a FaaS platform. Another possibility is combining FaaS with edge computing. In this exploration path, one could preprocess the data coming into a streaming application from IoT devices or even run machine learning algorithms on the edge nodes to reduce network latency. The future of FaaS as a substitute platform for stream processing is very promising.

REFERENCES

[1] Hashem, Ibrahim Abaker Targio, Ibrar Yaqoob, Nor Badrul Anuar, Salimah Mokhtar, Abdullah Gani, and Samee Ullah Khan. "The rise of “big data” on cloud computing: Review and open research issues." Information systems 47 (2015): 98-115.

[2] Jagadish, Hosagrahar V., Johannes Gehrke, Alexandros Labrinidis, Yannis Papakonstantinou, Jignesh M. Patel, Raghu Ramakrishnan, and Cyrus Shahabi. "Big data and its technical challenges." *Communications of the ACM* 57, no. 7 (2014): 86-94.

[3] Miller, H. Gilbert, and Peter Mork. "From data to decisions: a value chain for big data." *It Professional* 15, no. 1 (2013): 57-59.

[4] Neumeyer, Leonardo, Bruce Robbins, Anish Nair, and Anand Kesari. "S4: Distributed stream computing platform." In *2010 IEEE International Conference on Data Mining Workshops*, pp. 170-177. IEEE, 2010.

[5] Hirzel, Martin, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. "A catalog of stream processing optimizations." *ACM Computing Surveys (CSUR)* 46, no. 4 (2014): 1-34.

[6] Thies, William, Michal Karczmarek, and Saman Amarasinghe. "StreamIt: A language for streaming applications." In International Conference on Compiler Construction, pp. 179-196. Springer, Berlin, Heidelberg, 2002.

[7] Gedik, Bugra, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. "SPADE: The System S declarative stream processing engine." In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1123-1134. 2008.

[8] Qian, Ling, Zhiguo Luo, Yujian Du, and Leitao Guo. "Cloud computing: An overview." In *IEEE International Conference on Cloud Computing*, pp. 626-631. Springer, Berlin, Heidelberg, 2009.

[9] Armbrust, Michael, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee et al. "A view of cloud computing." *Communications of the ACM* 53, no. 4 (2010): 50-58.

[10] “Microsoft will be carbon negative by 2030.” https://blogs.microsoft.com/blog/2020/01/16/microsoft-will-be-carbon-negative-by-2030/. Visited in April 2021.

[11] Jonas, Eric, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu,Vaishaal Shankar et al. "Cloud programming simplified: A berkeley view on serverless computing." *arXiv preprint arXiv:1902.03383* (2019).

[12] Thönes, Johannes. "Microservices." *IEEE software* 32, no. 1 (2015): 116-116.

[13] “AWS Lambda.” https://aws.amazon.com/lambda/. Visited in March 2021.

[14] “Amazon DynamoDB.” https://aws.amazon.com/dynamodb/. Visited in March 2021.

[15] Sreekanti, Vikram, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. "Cloudburst: Stateful Functions-as-a-Service." *Proceedings of the VLDB Endowment* 13, no. 11.

[16] “Stream Processing vs. FaaS.” https://github.com/TBolton2000/stream-processing-vs-faas/. Visited in April 2021.

[17] “Apache Kafka.” https://kafka.apache.org/. Visited in November 2020.

[18] “Tutorial: Write a Kafka Streams Application.” https://kafka.apache.org/27/documentation/streams/tutorial/. Visited in March 2021.

[19] “Apache Maven.” https://maven.apache.org/. Visited in April 2021.

[20] “Apache Avro.” http://avro.apache.org/. Visited in March 2021.

[21] “Introducing JSON.” https://www.json.org/json-en.html/. Visited in April 2021.

[22] “Streams DSL.” https://kafka.apache.org/27/documentation/streams/developer-guide/dsl-api.html/. Visited in March 2021.

[23] “blank-java AWS Lambda Developer Guide.” https://github.com/awsdocs/aws-lambda-developer-guide/tree/main/sample-apps/blank-java/. Visited in March 2021.

[24] “Gson.” https://github.com/google/gson/. Visited in April 2021.

[25] “Simple Logging Façade for Java.” http://www.slf4j.org/. Visited in April 2021.

[26] “KafkaJS.” https://kafka.js.org/. Visited in March 2021.

[27] “Node Package Manager.” https://www.npmjs.com/. Visited in April 2021.

[28] “Using self-hosted Apache Kafka as an event source for AWS Lambda.” https://aws.amazon.com/blogs/compute/using-self-hosted-apache-kafka-as-an-event-source-for-aws-lambda/. Visited in March 2021.

[29] “Amazon Virtual Private Cloud.” https://aws.amazon.com/vpc/. Visited in March 2021.

[30] “Amazon Managed Streaming for Apache Kafka (Amazon MSK).” https://aws.amazon.com/msk/. Visited in April 2021.