A QoS-Latency Aware Event Stream Processing with Elastic-FaaS

Jagadheeswaran Kathirvel, Elango Parasuraman

Abstract: ***Stream processing systems need to be elastically scalable to process and respond the unpredictable massive load spike in real-time with high throughput and low latency. Though the modern cloud technologies can help in elastically provisioning the required computing resources on-the-fly, finding out the right point-in-time varies among systems based on their expected QoS characteristics. The latency sensitivity of the stream processing applications varies based on their nature and pre-set requirements. For few applications, even a little latency in the response will have huge impact, whereas for others the little latency will not have that much impact. For the former ones, the processing systems are expected to be highly available, elastically scalable, and fast enough to perform, whenever there is a spike. The time required to elasticity provision the systems under FaaS is very high, comparing to provisioning the Virtual Machines and Containers. However, the current FaaS systems have some limitations that need to be overcome to handle the unexpected spike in real-time. This paper proposes a new algorithm called Elastic-FaaS on top of the existing FaaS to overcome this QoS latency issue. Our proposed algorithm will provision required number of FaaS container instances than any typical FaaS can provision normally, whenever there is a demand to avoid the latency issue. We have experimented our algorithm with an event stream processing system and the result shows that our proposed Elastic-FaaS algorithm performs better than typical FaaS by improving the throughput that meets the high accuracy and low latency requirements.***

Keywords: ***Data Stream Processing, Serverless, Function-as-a-Service, Elastic FaaS.***

I. INTRODUCTION

With ever growing technologies and devices, the data stream is produced everywhere with ever increasing speed. However, the processing of those streams has some latency in real time stream processing systems. Sometimes few events are getting discarded due to the non-availability of systems. Since each event in the stream might have precious information, these discarded events will have impact on the final accuracy. So, all events need to be given equal importance for a precise result. Hence stream processing applications are always in need of highly available systems to process the streams on arrival [1]. Also, since the stream cannot be persisted completely because of its unbounded nature, most of the stream processing systems are expected to work with exactly-once processing guarantee on-the-fly.

Manuscript published on 30 August 2019.
***Correspondence Author(s)**

Jagadheeswaran Kathirvel*, Research Scholar, Department of Computer Science, Bharathiar University, Coimbatore, INDIA. Email: jagpro@gmail.com
Elango Parasuraman, Assistant Professor, Department of Information Technology, Perunthalaivar Kamarajar Institute of Engineering and Technology, Karaikal, INDIA. Email: elanalin_74@yahoo.com

The effective utilization of modern technologies such as cloud computing, fog computing, edge computing, and containerization technologies can help us achieve these QoS characteristics. This paper introduces an algorithm that focuses on how serverless [11] computing can be used effectively for stream processing with the above objective. Serverless computing is a method of providing the functional services on-demand basis. With the help of Serverless provider, the users can write and deploy the business code without thinking much about the underlying infrastructure [11]. Also, the underlying services will auto-scale whenever there is a spike and demand. Though there are many flavors in Serverless, it is mostly known for Function-as-a-Service, which is commonly known as FaaS. The underlying systems of FaaS is mostly built to work with low CPU and memory. The availability in FaaS systems will be high [20], comparing to the other computing systems mentioned above, hence the throughput and accuracy will also be high. It is known that event stream processing requires less CPU and memory comparing with the batch processing [21]. This naturally matches with the native characteristics of FaaS. So, when a FaaS stream processors are readily available, whenever a new event is arrived or a window of events are arrived, those events can be processed without delay and the result can be responded immediately. Also, it is known that in IaaS, the provisioning of virtual machines will take few minutes. In PaaS, it will take few seconds to deliver the required platform. But in FaaS, the underlying containers will be provisioned in milliseconds. Also, there is a difference between VMs, Containers, and FaaS. The provisioning and deprovisioning of VMs and Containers are in end user's control, but in case of FaaS, only the provisioning part is in end user control, the deprovisioning is taken care by the provider. It has both advantages and disadvantages. The advantage is that there is no overhead of carefully disposing them by the end user, and the disadvantage is that the end user cannot change the behavior of the system based on their usage. The following is going to be the flow of this paper. In section II, we will walk through the background and related work, and in section III, we will provide the solution overview, and in section IV, we will derive an algorithm based on the formulae introduced in section III, and the implementation will be explained in section V, and the experimental results will be shown in section VI. Section VII will summarize the paper with the future path.

II. BACKGROUND AND RELATED WORK

The study on elastic stream processing has been consistently growing in the recent decades. Some existing literature papers have explored and shared the various approaches for elastic steam processing.

Bugra et al [1] proposed an auto-parallelization solution that will dynamically change the number of parallel channels to achieve the best throughput based on work load changes. In this paper, the algorithm proposed was based on the deployment of individual nodes. Cervino et al [2] proposed an adaptive algorithm for provisioning virtual machines for data stream processing systems in the cloud, based on their benchmark tests across performance metrics such as network latency and jitter. Similarly, our earlier work [3] proposed a fixed number of local machines and adaptive virtual machines to process the stream with high throughput. All these papers were targeting the elastic scalability of the stream processing system based on the individual nodes. Heinze, T et al [4] [6] presented an elastic allocator for Complex Event Processing systems with the help of bin packing in which the system deploys or redeploys the stream processing operator in available nodes based on the input rate and computing power required. B. Lohrmann et al [5] presented a reactive strategy to enforce latency guarantees in data flows running on scalable stream processing engines, while minimizing the resource consumption. Lorido-Botran et al [7] discussed about all cloud technologies that can be used for stream processing system except FaaS, and Brogi A. et al [8] discussed about stream processing with Docker on Fog where the deployment is carried over by the docker containers. Again, all these papers were based on the individual machines, nodes, or containers which cannot be directly applied for the FaaS based stream processing systems.

Each cloud FaaS providers maintain their own standards, and also since FaaS technology itself is evolving day-by-day, the provider's documentation has only the partial information. The FaaS systems are controlled by their proprietary internal kernels which cannot be estimated by the end-users. Also, there is still scope to improvise the provisioning and usage patterns. Though FaaS can be used for elastically provisioning the underlying infrastructure whenever there is a load, it has some provider preconfigured limitations in scaling. But in real life applications, these limitations will become impediments and need to be overcome. So, in this aspect there were few research papers already written for Stream processing with FaaS. The paper [9] and [10] propose predictive controllers that dynamically allocate resources in FaaS provider platform. There are few other serverless related papers which are given in the reference section. From the study, it seems that none of the existing works attempted to resolve the fast scalability limitation problem from outside the FaaS provider. This paper attempts to do that. The algorithm proposed in this paper, will provision the sufficient number of FaaS container instances than any existing FaaS can provide normally, whenever there is a load and demand with low latency requirement.

III. SOLUTION OVERVIEW

We will use the term "Processor" to mean "Stream Processor" or specifically "FaaS Function", and "Sub-Processor" to mean "Sub-Stream Processor" or specifically "FaaS Function Container Instance". Also the below abbreviations will be used in rest of this paper for better readability and understanding:

– Qt - Event Stream Queue length at time (t)
– Pc - Processor capacity
– NPt - Number of Processors at time (t)
– NPt' - Number of expected Processors at time (t')
– NSPt - Number of Sub-Processors at time (t)
– NSPt' - Number of expected Sub-Processors at time (t')
– SPc - Sub-Processor capacity
– MNSP - Maximum number of Sub-Processors allowed per Processor
– TNSPt - Total number of Sub-Processors for all Processors at time (t)
– TNSPt' - Total number of expected Sub-Processors for all Processors at time (t')
– L - QoS SLA for latency allowed in unit time
– PPT - Processor provisioning time
– n - Expected number of iterations to provision the required Processors.
– c - Number of Sub-Processors deployed per iteration.
– tf - Timer Frequency

We form the below elastic provisioning formula to find out the number of Processors expected to process all items from a queue in time t'.

$$\mathrm{NPt'} = \mathrm{NPt} + \frac{(\mathrm{Qt} - \mathrm{NPt.Pc})}{\mathrm{Pc}} \tag{1}$$

The above formula can be directly applied for adaptively provisioning the individual systems such as VMs, Containers, and so on. However, in systems like FaaS, this formula cannot be directly applied, since in FaaS under each Function, there can be multiple containers. These FaaS containers are incrementally deployed with certain time interval based on the load, invocation type, and other parameters [23][24]. For this kind of FaaS environment, a few changes are required in the above formula. So, the number of Sub-Processors (Container instances) deployed at the rate of (c) in the interval of (n) after time (t') can be formed as:

$$\mathrm{NSPt'} = \mathrm{c}\sum_{\mathrm{n}=1}^{\mathrm{k}} \mathrm{n} \tag{2}$$

The number of Sub-Processors instantiated vary from time-to-time based on scale-out, scale-in, and max concurrent limit. Also, it will vary from one Processor to another, since each Processor would have been instantiated at different timings. So, the total number of instantiated Sub-Processors for all currently running Processors at time (t) can be obtained by summing up the Sub-Processors created at each Processor as:

$$\mathrm{TNSPt} = \sum_{\mathrm{p}=1}^{\mathrm{NPt}} \mathrm{NSPt}\,[\mathrm{p}] \tag{3}$$

Similarly, the total number of allowed Sub-Processors for all currently running Processors at time (t) can be obtained by multiplying the number of Processors with their allowed Sub-Processors.

$$\mathrm{TMNSP} = \mathrm{NPt.MNSP} \tag{4}$$

The expected number of Sub-Processors for the given stream load can be arrived at time (t') by substituting (3) in (1):

$$\mathrm{TNSPt}' = \mathrm{TNSPt} + \frac{(\mathrm{Qt} - \mathrm{TNSPt.Pc})}{\mathrm{Pc}} \tag{5}$$

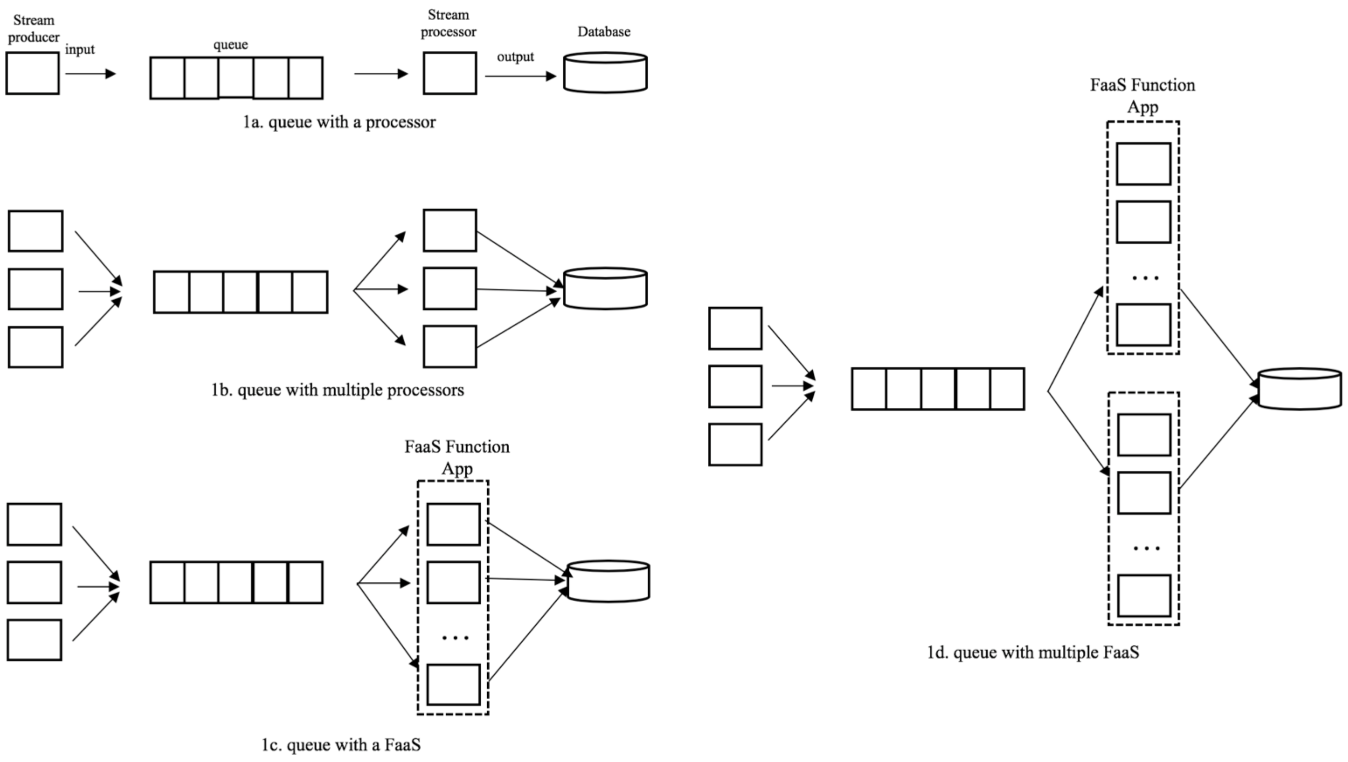


Fig. 1. Stream processing systems

Expected number of iterations or time units to reach the expected Sub-Processors found in (5) can be formed from (2) as:

$$n = \sqrt{\frac{2.\mathrm{NSPt}' - c}{c}} \tag{6}$$

If the expected iterations (n) is greater than both QoS latency and the time taken to provision a new Processor (FaaS Function), then the system can scale-out for new Processor(s). The number of new Processors required can be obtained by dividing NSPt' by TMNSP.

$$\mathrm{NPt}' = \frac{\mathrm{NSPt}'}{\mathrm{TMNSP}} \;\; \{n > L; n > \mathrm{PPT}\} \tag{7}$$

Here, the system will scale based on NPt'. If it is greater than NPt, then scale-out will happen. If it is less than NPt the scale-in can happen; Otherwise, no action is required, since the system is stable already to process the incoming load.

The conceptual design is depicted in the Figure 1. Figure 1a shows a simple stream processing system with a producer, a queue, a stream processor, and a database. This kind of design is enough for simple event stream processing system. Figure 1b shows multiple stream producers and multiple stream processors along with a queue and a database. It suits for the stream processing systems where the latency is tolerated. Figure 1c also has multiple stream producers, a queue, and a database. But instead of having multiple individually spun up stream processors, these stream processors were spun up elastically by the FaaS Function App. Here, FaaS provider takes care of automatically provisioning and deprovisioning the underlying stream processing containers. Figure 1d is almost similar to Figure 1c, but with multiple FaaS Function Apps that are elastically spun up by our Elastic-FaaS algorithm. This algorithm provisions more containers than a normal FaaS can do.

Generally, in FaaS environment, the scale-in or the deprovisioning is taken care by the providers. Hence, we are not going to focus much about the deprovisioning part in the upcoming sections and the rest of this paper as well. However the algorithm has been designed to take care of that. The above conceptual design can be applied in other scenarios as well, such as Cluster to Virtual Machine, Virtual Machine to Containers, and so on.

IV. CONTROL ALGORITHM

The Algorithm 1 shows the formulae derived in Section III in programmatic flow. The algorithm starts with setting up the infrastructure such as queue and FaaS through scripts. The input to the algorithm is initialized and the default values are set. The maximum number of FaaS concurrent container instances (MNSP) can be retrieved from the respective FaaS provider SDKs/Spec. This varies from one provider to another, and also from one plan to another. The capacity of the container instance (SPc) will vary based on the end user application's functional logic and provider infrastructure load. The former can be identified by the end user by looking at the functional code and its performance metrics, whereas the later cannot be done by the end user. Also, the expected SLA latency (L) of QoS can be set by the end user. This will control the entire algorithm. Also, the time taken to provision a FaaS Function (PPT) can be either retrieved from the provider spec or can be set by the end user based on the previous experience with the provider.

The timer frequency (tf) sets how often the monitor algorithm should run so that the elastic-scaling can happen automatically. The timer invokes the monitor functionality at pre-set frequency interval. The monitor is core part of this Elastic-FaaS algorithm. It takes care of finding out number of required FaaS Functions to be created, so that the streams can be processed with high throughput and low latency. FaaS itself is known for its elastic nature. However, the reason why we named this algorithm as Elastic-FaaS is that this algorithm will elastically scale the number of FaaS Functions which in-turn will have the multiplication effect on the number of FaaS Function container instances provisioned.

Algorithm 1: Elastic-FaaS

```
Environment:
        Queue - Event Stream Queue
        FaaS - Function as a Service

Initializer:
        NPt = 1
        NPt' = 1
        TNSPt = 0
        c = Set by FaaS provider
        MNSP = Set by FaaS provider
        PPT = Set by FaaS provider
        SPc = Get from recent performance
        metrics
        L = Set by Elastic-FaaS consumer
        tf = Set by Elastic-FaaS consumer

Output:
        Elastically Scaling system.
        High throughput with low latency
        result.

Timer:
        1. Monitor()

Monitor:
        2. NSPt' = (c(n(n+1)))/2

        3. For (p = 1; p < NPt; p++)
        4.    TNSPt += NSPs[p]
        5. End For

        6. TMNSP = NPt.MNSP

        7. TNSPt' = TNSPt + (Qt-TNSPt.Pc)/Pc

        8. n = SQRT ((2.NSPt' - c)/c)

        9. If (n > L) then
        10.    If (n > PPT) then
        11.      NPt' = NSPt'/TMNSP
        12.    End If
        13. End If

        14. Scaler(NPt')

Scaler:
        15. If NPt' > NPt then
        16.   ScaleOutProcessor(NPt')

        17. If NPt' < NPt then
        18.   ScaleInProcessor(NPt')

        19. NPt = NPt'

Processor:
        20. If (load increase)
        21.   ScaleOutSubProcessor()

        22. If (load decrease)
        23.   ScaleInSubProcessor()

Sub-Processor:
    On arrival of an event,
        24. Process it
        25. Store the result
        26. Acknowledge to Queue
```

V. IMPLEMENTATION

The algorithm can be simulated with any FaaS provider. However, there will be slight change in the behavior of each service provider. We have used Azure Function for our experiment. The reference architecture of this implementation is given in Figure 2. If AWS Lambda, Google Cloud Function, Apache OpenWhisk, OpenFaaS, or any other FaaS would have been chosen, its implementation would have been little different, since each FaaS provider's elastic characteristics and implementation steps are different from each other [26]. We created a direct acyclic graph for experimenting our algorithm with data producer, queue, a stream processing function app, and the Azure Table Storage for persisting the final result. The first part was synthetic data producer which produces the bank transaction synthetic data for this experiment. This data producer adds the data into the Azure Service Bus Queue. There were multiple instances of producer running to generate and add the massive synthetic data into the Queue. This data was in JSON format and has transaction id, timestamp, account number, currency type, and amount value attributes in it.

The Azure Function App was created to classify whether a transaction is a high-value or low-value transaction. The amount which is greater than certain value is considered as high-value, and anything below that value is considered as low-value transactions. At the rear-end, the classification result is stored into the Azure Table along with the time it took for processing, Function App id, instance id, and transaction id. This table in-turn will be used to identify how many high-value transactions were done between certain time limits.

The programmatically produced synthetic data are added into the Queue, and a Function App was created via the Azure CLI [24]. It started processing the transactions one-by-one, and since there was huge transactions already piled up in queue, and was continuously getting added, the first Function App started adding new stream processing instances one after other with certain time interval. At certain stage, we could notice that the number of outgoing queue items were very less than the incoming queue items. And at one stage the queue was intermittently not accepting the incoming messages. i.e., load shedding started occurring. That means some of the events were getting discarded, so our end result will not be accurate.

Then we ran our Elastic-FaaS algorithm that was scripted with Azure CLI. In next few minutes, the algorithm created a new Function App. The containers from both the existing Function App and newly created Function App started processing the queue items. Subsequently few more Function Apps were also created by the algorithm, since there was still huge load on the queue. This means the number of Function App Container instances that got spun up were increased with multiplier effect. After few seconds, we could see the number of outgoing messages were increased and Queue started accepting more incoming messages. i.e., load shedding was gone because of the high availability of the containers. The results are provided in the next section. Also, as mentioned earlier, the deprovisioning of container instances are taken care by the FaaS provider based on the load and other parameters. The new FaaS Functions that were created by the algorithm are deprovisioned by this algorithm itself as part of the scale-in flow.

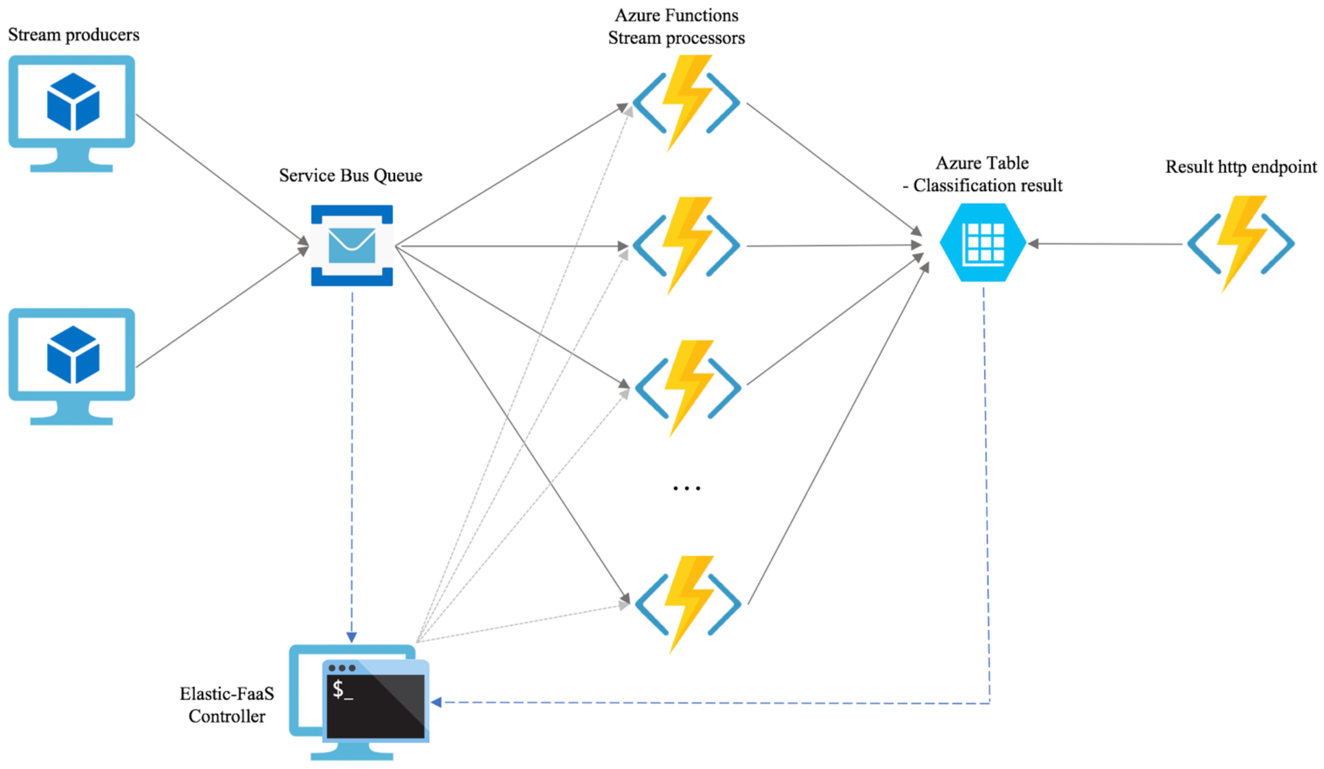


Fig. 2. Stream processing system with Elastic-FaaS

VI. EXPERIMENTAL RESULT

The experimental results are shown in the charts of Figure 3. The figure 3a shows the timing of the new FaaS Functions and their containers created by our Elastic-FaaS algorithm. First FaaS was initially created by us. Second FaaS was created by the algorithm after few minutes, and similarly third FaaS. Figure 3b shows overall FaaS Function container instances that were created by this algorithm.

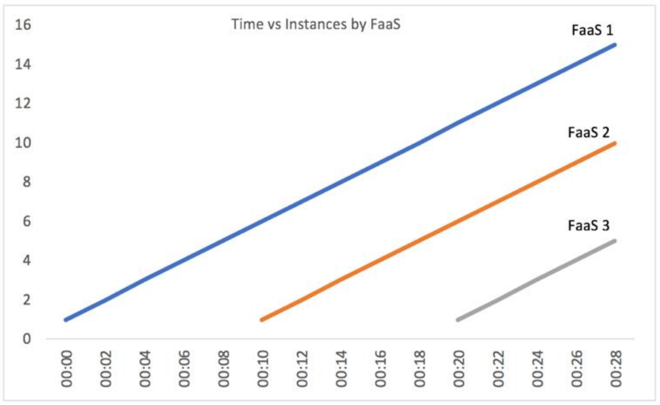
Table 1. Instances provisioned by time

Time	FaaS 1	FaaS 2	FaaS 3	Total
00:00	1	-	-	1
00:02	2	-	-	2
00:04	3	-	-	3
00:06	4	-	-	4
00:08	5	-	-	5
00:10	6	1	-	7
00:12	7	2	-	9
00:14	8	3	-	11
00:16	9	4	-	13
00:18	10	5	-	15
00:20	11	6	1	18
00:22	12	7	2	21
00:24	13	8	3	24
00:26	14	9	4	27
00:28	15	10	5	30

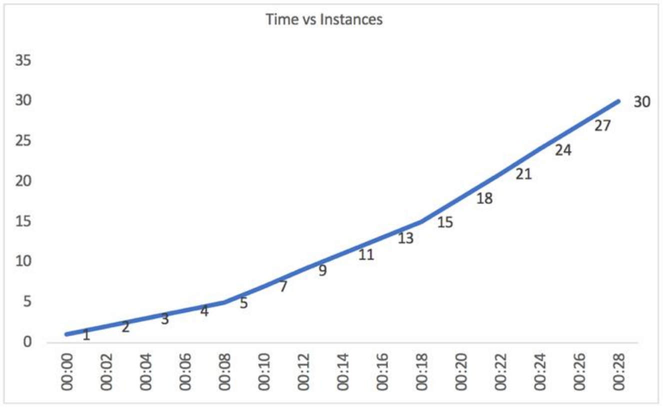
Figure 3c shows how long it would have taken for a stream processing system with 1 FaaS over 3 FaaS to reach 30 FaaS Function Container instances. Table 1 shows the number of instances provisioned by each FaaS at certain time intervals.

The single FaaS would have taken nearly sixty time units comparing to the three FaaS which took less than thirty time units. Figure 3d shows how the ingestion and throughput are balanced by our algorithm. When the system was started, the input rate was higher, and there was less throughput that caused the load shedding.

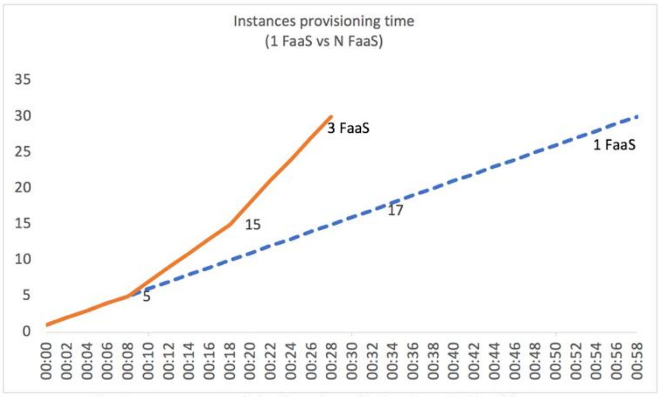
After the Elastic-FaaS started adding the Functions one-by-one the throughput increased gradually, and at certain stage the system was able to process the input load without much latency and with high throughput. The total number of high-value and low-value transactions became accurate after our algorithm's elastic provisioning, since the load-shedding was avoided by the high availability of containers.



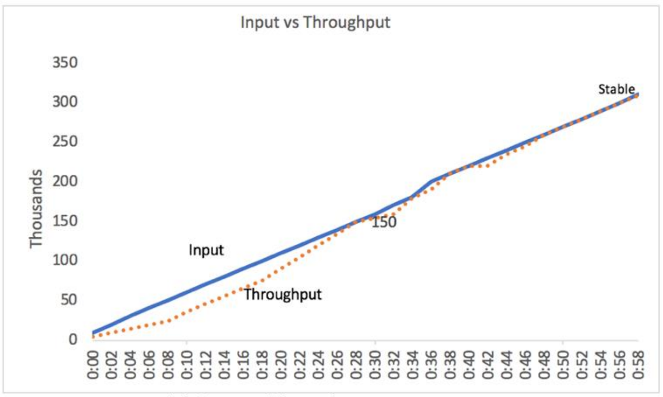
3a. Time vs Instances by FaaS



3b. Time vs Instances



3c. Instances provisioning time (1 FaaS vs N FaaS)



3d. Input vs Throughput

Fig. 3. Experimental Results

VII. CONCLUSION AND FUTURE WORK

We presented a novel elastically scalable Elastic-FaaS algorithm that increases the availability of the stream processing systems whenever there is a load. It is able to control the number of FaaS containers provisioned by adjusting QoS-latency requirement. The experimental result shows that our algorithm gets rid of the load shedding issue by improving the systems availability which in turn increases the throughput that meets the high accuracy and low latency requirements. This makes our Elastic-FaaS algorithm to perform better than typical FaaS. This algorithm can be explored in many real-time on-demand deployment scenarios where elastic scalability is highly required in short span of time. In future, we will focus on integrating this algorithm with real life applications and also with other widely used recent stream processing systems. Also, since FaaS has been changing the way the stream processing and mining systems worked in the past, we will further work on increasing the availability of the FaaS containers by resolving the cold-start issues [19].

REFERENCES

1. B. Gedik, S. Schneider, M. Hirzel and K. Wu, "Elastic Scaling for Data Stream Processing," in IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 6, pp. 1447-1463, June 2014. doi: 10.1109/TPDS.2013.295
2. Cervino, J.; Kalyvianaki, E.; Salvachua, J.; Pietzuch, P., "Adaptive Provisioning of Stream Processing Systems in the Cloud," Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on, vol., no., pp.295,301, 1-5 April 2012
3. J. Kathirvel and E. Parasuraman, "Effective data stream mining using ensemble on cloud with load balancing (E2CL)," 2015 International Conference on Computing and Communications Technologies (ICCCT), Chennai, 2015, pp. 383-386. doi: 10.1109/ICCCT2.2015.7292780
4. Heinze, T., Ji, Y., Pan, Y., Grüneberger, F.J., Jerzak, Z., & Fetzer, C. (2013). Elastic Complex Event Processing under Varying Query Load. BD3@VLDB.
5. B. Lohrmann, P. Janacik and O. Kao, "Elastic Stream Processing with Latency Guarantees," 2015 IEEE 35th International Conference on Distributed Computing Systems, Columbus, OH, 2015, pp. 399-410. doi: 10.1109/ICDCS.2015.48
6. T. Heinze, V. Pappalardo, Z. Jerzak and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," 2014 IEEE 30th International Conference on Data Engineering Workshops, Chicago, IL, 2014, pp. 296-302. doi: 10.1109/ICDEW.2014.6818344
7. Lorido-Botran, Tania et al. "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments." Journal of Grid Computing 12 (2012): 559-592.

8. Brogi A., Mencagli G., Neri D., Soldani J., Torquati M. (2018) Container-Based Support for Autonomic Data Stream Processing Through the Fog. In: Heras D. et al. (eds) Euro-Par 2017: Parallel Processing Workshops. Euro-Par 2017. Lecture Notes in Computer Science, vol 10659. Springer, Cham
9. M. R. HoseinyFarahabady, A. Y. Zomaya and Z. Tari, "A Model Predictive Controller for Managing QoS Enforcements and Microarchitecture-Level Interferences in a Lambda Platform," in IEEE Transactions on Parallel and Distributed Systems, vol. 29, no. 7, pp. 1442-1455, 1 July 2018. doi: 10.1109/TPDS.2017.2779502
10. HoseinyFarahabady M., Lee Y.C., Zomaya A.Y., Tari Z. (2017) A QoS-Aware Resource Allocation Controller for Function as a Service (FaaS) Platform. In: Maximilien M., Vallecillo A., Wang J., Oriol M. (eds) Service-Oriented Computing. ICSOC 2017. Lecture Notes in Computer Science, vol 10601. Springer, Cham
11. Ali Kanso and Alaa Youssef. 2017. Serverless: beyond the cloud. In Proceedings of the 2nd International Workshop on Serverless Computing (WoSC '17). ACM, New York, NY, USA, 6-10. DOI: https://doi.org/10.1145/3154847.3154854
12. Stefan Brenner and Rüdiger Kapitza. 2019. Trust more, serverless. In Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR '19). ACM, New York, NY, USA, 33-43. DOI: https://doi.org/10.1145/3319647.3325825
13. J. Kuhlenkamp and S. Werner, "Benchmarking FaaS Platforms: Call for Community Participation," 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, 2018, pp. 189-194. doi: 10.1109/UCC-Companion.2018.00055
14. C. Pahl, "Containerization and the PaaS Cloud," in IEEE Cloud Computing, vol. 2, no. 3, pp. 24-31, May-June 2015. doi: 10.1109/MCC.2015.51
15. W. Felter, A. Ferreira, R. Rajamony and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, 2015, pp. 171-172. doi: 10.1109/ISPASS.2015.7095802
16. Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. 2018. Package-Aware Scheduling of FaaS Functions. In Companion of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18). ACM, New York, NY, USA, 101-106. DOI: https://doi.org/10.1145/3185768.3186294
17. M. Sewak and S. Singh, "Winning in the Era of Serverless Computing and Function as a Service," 2018 3rd International Conference for Convergence in Technology (I2CT), Pune, 2018, pp. 1-5. doi: 10.1109/I2CT.2018.8529465
18. Baldini I. et al. (2017) Serverless Computing: Current Trends and Open Problems. In: Chaudhary S., Somani G., Buyya R. (eds) Research Advances in Cloud Computing. Springer, Singapore
19. J. Manner, M. Endreß, T. Heckel and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, 2018, pp. 181-188. doi: 10.1109/UCC-Companion.2018.00054
20. Serverless Architectures - Martin Fowler, https://martinfowler.com/articles/serverless.html
21. Batch Processing vs Real Time Processing – Comparison, https://data-flair.training/blogs/batch-processing-vs-real-time-processing/
22. Azure Functions scale and hosting, https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale
23. Azure Functions Trigger – scaling, https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-event-hubs#trigger---scaling
24. Azure Function CLI, https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-azure-function-azure-cli
25. AWS scaling, https://docs.aws.amazon.com/lambda/latest/dg/scaling.html
26. AWS concurrent executions, https://docs.aws.amazon.com/lambda/latest/dg/concurrent-executions.html

AUTHORS PROFILE

Jagadheeswaran Kathirvel is pursuing his doctorate in Department of Computer Science at Bharathiar University, India. His area of interests includes data stream processing, data mining, artificial intelligence, along with event driven software architecture, design, and engineering. He completed his master's degree in computer applications in 2007 at Bharathiar University, and bachelor's degree in computer science at Periyar University, India, in 2003.

Elango Parasuraman is working as an Assistant Professor in Department of Information Technology at Perunthalaivar Kamarajar Institute of Engineering and Technology, Karaikal, India. His area of interests includes image processing, data mining, and web mining. He completed his Ph.D., at National Institute of Technology Tiruchirappalli, India, in 2011, and his M.Tech., at National Institute of Technology Karnataka, India, in 2005.