Run-time Adaptation of Stream Processing Spanning the Cloud and the Edge

Adam Cattermole
a.cattermole@newcastle.ac.uk
School of Computing
Newcastle University
Newcastle upon Tyne, UK

Jonathan Dowland
jdowland@redhat.com
Red Hat, Inc.
Newcastle upon Tyne, UK

Paul Watson
paul.watson@newcastle.ac.uk
School of Computing
Newcastle University
Newcastle upon Tyne, UK
& The Alan Turing Institute
UK

ABSTRACT

Applications that process streams of events generated by sensors are important in a wide range of domains. It is now common to distribute stream processing across edge devices and the cloud. This exploits processing power near the sensors, reducing the load on the cloud and often the required network bandwidth. In this paper we focus on one challenge in distributed stream processing: automatically adapting the partitioning of the processing between the edge and the cloud without a loss of service. An example is when the event arrival rate increases and the edge processor can no longer meet performance requirements. Re-partitioning without loss of service involves moving computations between the edge and the cloud while events are still being processed. In this paper we describe *StrIoT* – a stream processing system that supports automatic re-partitioning. It is based on a set of functional stream operators, and the paper describes how the run-time system can automatically adapt applications that use them. A key feature is support for the fission and fusion of operators during adaptations. Performance evaluation shows that *StrIoT* can move parts of a stream processing application between the cloud and edge with only a low, temporary impact on performance.

CCS CONCEPTS

• **Computer systems organization** → **Real-time system architecture**; • **Information systems** → *Computing platforms.*

ACM Reference Format:
Adam Cattermole, Jonathan Dowland, and Paul Watson. 2021. Run-time Adaptation of Stream Processing Spanning the Cloud and the Edge. In *2021 IEEE/ACM 14th International Conference on Utility and Cloud Computing (UCC '21) Companion (UCC '21 Companion), December 6–9, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3492323.3495627

UCC '21 Companion, December 6–9, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-9163-4/21/12…$15.00
https://doi.org/10.1145/3492323.3495627

1 INTRODUCTION

Applications that process streams of events generated by sensors are becoming increasingly important in a wide range of domains. A common application structure is to distribute the processing across edge devices and the cloud. This has the advantage of exploiting processing power near the sensors to reduce both the load on the cloud, and the required network bandwidth between the edge and the cloud. This approach does however raise issues for developers as designing a distributed stream processing system that meets performance and other non-functional requirements is complex.

In this paper we focus on one key challenge: adapting the partitioning of the processing between the edge and the cloud at run-time without loss of service. Figure 1a shows a very simple stream processing system in which a stream of events generated by a sensor S is first processed by operator F before being processed by operator G. For example, F might filter out uninteresting events generated by S, before G performs a computation on each of those events. A developer might decide to deploy F at the edge (so reducing the bandwidth needed between the edge and the cloud) and G on the cloud (to exploit its greater processing power). This is shown in Figure 1b. To ensure that this configuration will succeed, the developer could estimate the load on the edge device by combining estimates of the rate at which S will generate messages, and the processing power required to process each message. If processing this rate of messages overloads the edge device then the system will fail. In this case, the only solution is to run F on the cloud, rather than at the edge (Figure 1c) providing the bandwidth of the network between them is large enough to handle the extra traffic.

The developer should also consider whether both F and G could both run on the edge device (Figure 1d). This would reduce the cloud resources needed, and so could reduce the hosting costs paid to the cloud provider. By exploring these various options, the application developer could make the choice of the optimal partitioning of the stream computation between the edge and the cloud. However, while this may be the best partitioning for the initial deployment of the application, what happens if some of the underlying assumptions change? In particular, what if the rate of events arriving at F for processing increases? This may be due to the installation of more sensors, all sending data to F, or it could be because the sensor is replaced with a newer model that can sample at a higher frequency. In this case, the edge device may not have the power to process all the events, and so F would have to be re-deployed onto the cloud. The opposite scenario is where it is discovered that a lower sampling rate is sufficient for the sensor,

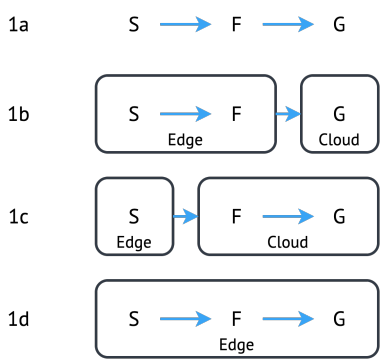

Figure 1: Partitioning Options

and so F now requires lower computational resources, enabling G to also be run at the edge, so reducing cloud costs.

Moving operators between the edge and cloud is not straightforward. Taking the application down for a period of time to make the change may not be possible (for example if it is a safety-critical application), or may be at least undesirable. Any manual reconfiguration also introduces the possibility of human error.

In this paper we describe an alternative: *StrIoT* enables the automatic adaptation of streaming applications, including automatically re-partitioning processing between the edge and the cloud. In order to achieve this, we designed a set of high-level functional stream processing operators from which developers can build streaming applications. These operators are defined and implemented in a pure functional programming language, and so have simple, clear semantics (especially in terms of state management) that we can exploit when implementing a set of run-time adaptations.

A particular advantage of *StrIoT* is that it supports partition fission (splitting the operators from one partition across more than one partition) and fusion (combining the operators from multiple partitions into one). This opens up more options of adaptation than is possible with simple partition migration. As will be discussed, fusion can also improve performance. Fusion and fission are enabled in *StrIoT* because the partitions are automatically created by transforming the original program provided by the programmer.

The rest of the paper is as follows. We define the operators in Section 2 and then describe the run-time system that executes distributed stream processing applications built from these applications (Section 3). Section 4 focuses on how *StrIoT* implements automatic, run-time adaptivity without loss of service. We then describe a performance evaluation of two types of adaptivity involving fusion and fission between the edge and the cloud respectively. This shows that this can be done with only a low, temporary impact on performance. After a comparison with related systems we drawn conclusions from the results, and point to future directions. The *StrIoT* system, and the example used in the evaluation, are available as open source [21].

2 THE FUNCTIONAL STREAM PROCESSING OPERATORS

We model a stream in *Haskell* (a pure functional language) as a (possibly infinite) list of events:

```
data Event a =
    Event  { time    :: Maybe Timestamp
           , value   :: Maybe a }
type Timestamp   = UTCTime
type Stream a    = [Event a]
```

`Event` has been designed to be as general as possible: it can hold data of any type (e.g. integers, strings, tuples, lists, trees, graphs and even functions - in the definition, "a" represents any type). An `Event` can also optionally hold a timestamp (`Maybe` is a *Haskell* datatype that can contain a value, or `Nothing`).

Based on analysis of the literature on stream processing and Complex Event Processing [5], and by experimenting with the implementation of a range of applications, we have selected and implemented a library of operators an application developer can use to create streaming applications in *StrIoT*. An important criterion for choosing these is that both an optimizer and the run-time system that enables adaptivity can exploit knowledge of their semantics.

There are four key types of operations that any stream processing system must perform:

- `Filter`: select only those events that match a set of criteria
- `Map`: transform all events into another type of event
- `Window`: break a stream of events into a stream of windows, each containing zero or more events
- `Combine`: merge or combine events from different streams

A full description of all the operators provided by *StrIoT* is available at [20]. We now briefly describe the three operators that we will use in the examples of adaptivity later in the paper.

2.1 Filtering

The `streamFilter` function takes a stream as input, and outputs a stream containing only those events that meet the provided criteria. The type signature is:

```
streamFilter :: EventFilter a
             -> Stream a
             -> Stream a

type EventFilter a = a -> Bool
```

One advantage of *StrIoT* is that the application programmer does not need to know or understand the exact format of the events as seen by the infrastructure. They only need to know the type of the `value` field – in this case so that they can write a function of type `EventFilter` to filter out all events whose value does not meet a particular criterion.

For example, if a temperature sensor generates a stream of events of type `Int`:

```
tempSensor :: Stream Int
```

an application programmer interested only in temperatures of over *100* can easily write a program to filter out all other values:

```
over100 :: EventFilter Int
over100 temp = temp > 100
```

```
streamFilter over100 tempSensor
```

2.2 Mapping

The function `streamMap` is used to transform the values in a stream. The programmer supplies a function of type `EventMap` which is applied to the value within each event in the input stream to generate the output stream. The function `streamMap` has the signature:

```
streamMap :: EventMap a b
          -> Stream a
          -> Stream b

type EventMap a b = a -> b
```

If the programmer wants to extend the running example by representing all temperatures by their value over 100, they can code this as:

```
amountOver100 :: EventMap Int Int
amountOver100 temp = temp - 100

streamMap amountOver100
        $ streamFilter over100 tempSensor
```

(the `$` symbol is used in *Haskell* to chain functions together.)

Note that the application programmer does not have to understand the format of events, nor how map or filter are implemented. Instead they can focus solely on how the value held within the events is to be processed.

2.3 Windowing

Windowing provides a way to break a stream into a new stream whose events contain a subset of events from the original. Other operators (usually `streamMap`) can then be used to process the events in each window. The basic windowing function in *StrIoT* is:

```
streamWindow :: WindowMaker a
             -> Stream a
             -> Stream [a]

type WindowMaker a = Stream a -> [Stream a]
```

`WindowMaker` is a function that generates a list of windows. A pre-defined set of functions covering the common `WindowMaker` cases is provided. However, application programmers are free to define their own. An example of a pre-defined function is a sliding window of fixed time length:

```
slidingTime:: NominalDiffTime -> WindowMaker a
```

The first argument is the length of the window in milliseconds.

3 THE STRIOT DISTRIBUTED RUN TIME SYSTEM

The set of *StrIoT* operators that are to be deployed to a single node (a decision that can be made by the application programmer, or the *StrIoT* optimizer) is called a *partition*. The *Haskell* code for each partition is automatically written out to individual source code files which are then compiled into executable code by the GHC *Haskell* compiler. We therefore benefit from the optimizations that the compiler applies to the source code.

The compiled code for each partition is then automatically packaged in its own Docker container ready for deployment on the cloud or a field gateway/edge device such as a Raspberry Pi. The deployment system automatically connects up the partitions so that events generated as the output from one partition are sent as input to the next.

Events are serialised prior to transmission, and then deserialised and placed in a queue on arrival. Each node implements a multi-threaded connection handler, to allow multiple upstream nodes to connect to the downstream concurrently (e.g. when merging events from different sources). Figure 2 shows the node level design.

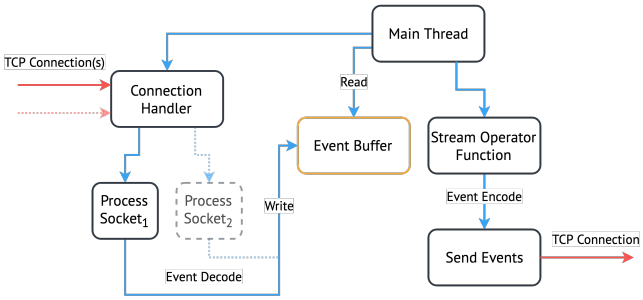

Figure 2: Node level design

4 ADAPTIVITY

Our work focuses on systems in which the operators in a stream processing system are distributed across a set of physical platforms such as cloud servers or edge nodes. We illustrate our work using an application taken from the Distributed Event Based Systems (DEBS) Conference Grand Challenge in 2015[10]. Its structure is shown in Figure 3. A stream of events, each representing one taxi journey (including start/end locations and times, fare and tip paid) are sent to a `Map toJourney` operator that performs pre-processing (e.g. turning GPS locations into grid squares). The output is then filtered (`Filter inRange`) to remove any journeys that are outside of the area of interest. These events are then assigned into 30 second sliding windows over which a top-k operation is performed to determine the most profitable routes.

Figure 3: Taxi Application Example

A range of adaptations is possible – we focus on two key ones in this article. In both cases the operators moved include one with state which complicates the adaptation. However, because *StrIoT* is based on a set of pure functional operators, their clear semantics makes it possible to perform the adaptation efficiently and automatically, without any human intervention.

4.1 Cloud to Edge Fusion

Figure 4 shows adaptation from an initial partitioning of the example application across the edge and cloud. Initially, the data is mapped and filtered at the edge (P_2) before being sent to the cloud for windowing and top-k calculation (P_3). If the monetary cost of running on the cloud becomes expensive, and there is sufficient spare computational resources at the edge, *StrIoT* adaptivity mechanisms can move the windowing and top-k to the edge without loss of service, or manual reconfiguration. This happens as follows.

Management messages are defined as a special kind of `Event` containing a unique identifier for each stateful partition (e.g. `streamWindow`) that must be shut down prior to migration to another node. These messages are passed directly into the stream operator functions, where they trigger the function to stop processing and instead store any partial state contained within the function. In this example, the management message is sent to P_2 and enters the stream as any other `Event`. As P_2 is stateless (operators such as `streamMap` and `streamFilter` operate independently on each `Event`), no state needs to be stored prior to migration, but the management message causes each operator to stop taking any further input. Once the management message arrives at P_3, it is passed to the stateful `streamWindow` operator, triggering it to stop processing further input, and to store its state – in this case a partial window. After this, the management message passes through the (stateless) `streamFilter` before being removed from the stream. By injecting the management message into the stream and passing it through the operators in order, we can ensure that both partitions are shut down at the same point in the computation. A single Redis [17] key/value store instance was used for all partitions as the external state store; the key is a unique identifier contained within the management message, and the value is the encoded state. Partition P_5 – a new partition containing the combined operators from both P_2 and P_3 – is then started on the edge infrastructure and initialised using the key to retrieve the state directly from Redis – in this case the `streamWindow` function is initialised with the partial window. The operators can then resume normal operation, processing the stream of events that have temporarily queued.

A feature of *StrIoT* is that the fusion of operators from one or more partition into a new single partition is possible because the partitions are created by transforming the original program provided by the programmer. The advantage of fusion (rather than just migrating partition P_2 onto the same edge node as P_3) is that the Haskell compiler can generate efficient code for the fused partition – here it removes the need to serialise and deserialise the events passing from `streamFilter` to `streamWindow`, as would be the case if they were in 2 separate partitions. As a result, when compared directly to migrating the partition, fusion results in increased throughput and lower latency.

4.2 Edge to Cloud Fission

The second example of adaptivity is the inverse of the first. Initially, all of the operators are running on the edge, but – perhaps because the event arrival rate has increased, causing the edge to run out of available processing power – the windowing and top-k operators are moved to the cloud (Figure 5). Fission increases the number of partitions by splitting a single partition into two. This

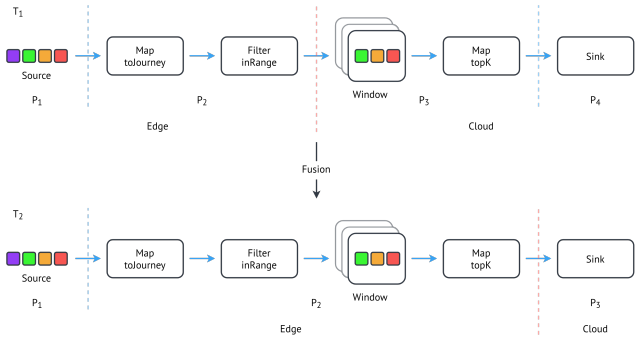

Figure 4: Cloud to Edge Fusion

is possible in *StrIoT* for the same reason that fusion is possible – the two partitions can be created by transforming the original program provided by the programmer. Fission occurs as follows. A management message is received by P_2 causing the partition to be shut down, with the state of the `streamWindow` operator stored in the key/value store. Two new partitions, P_4 and P_5, are then created from the original program provided by the programmer. These new partitions are deployed, with P_4 on the edge infrastructure, and P_5 on the cloud infrastructure. Once the state of the `streamWindow` has been restored, the partitions can resume event processing.

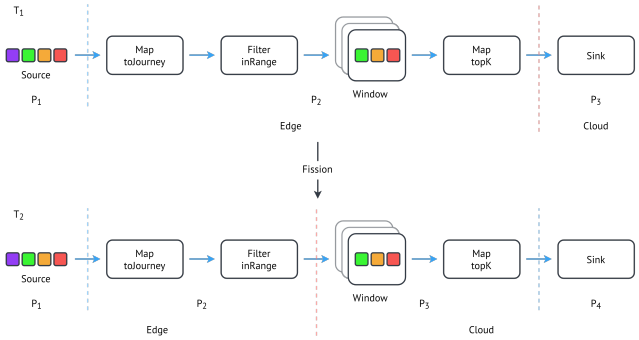

Figure 5: Edge to Cloud Fission

5 EVALUATION

In order to evaluate adaptivity, we created a distributed testbed. Cloud infrastructure from Microsoft Azure [15] was used to create a Kubernetes [9] cluster, in which there is a single control-plane node and three regular nodes. Standard D2s v3 virtual machines (VMs) were used, each with two Intel Xeon Platinum 2.60GHz CPU cores and 8GB memory. A single edge device, located out of the cloud (connected by a domestic broadband connection), was also provisioned as part of the Kubernetes cluster. An AAEON UP board was used at the edge, representing a field gateway between sensors and the cloud. This contained an Intel Atom x5-Z8350, providing 4 cores @ 1.92GHz and 4GB memory. By using Kubernetes node restriction labels we could mark the edge device, and use Kubernetes node affinity rules to ensure the scheduler deploys the required applications to the correct nodes.

Evaluation used the taxi application discussed in Section 4. Every experiment processed 25,000 events from the taxi data-set. These were streamed from the starting node, and it took approximately 300s for all the events to stream through the system. After starting the initial partition for the experiment, the system is given 30s to settle into a steady-state. After this, a management message to trigger the adaptation is injected into the system. There are then 150-210 seconds after the adaptation until the experiment ends.

5.1 Cloud to Edge Fusion Results

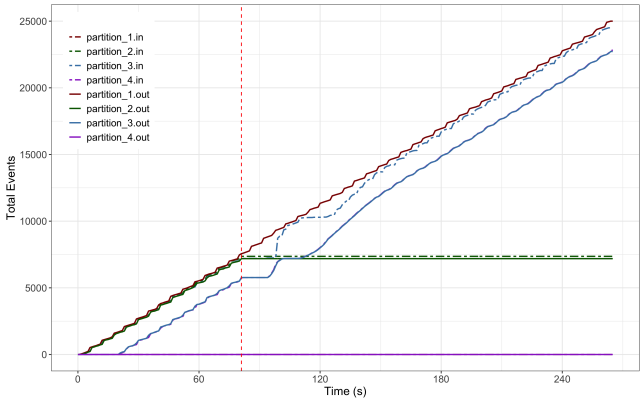

Figure 6: Fusion to Edge Results

Figure 6 shows the effect of fusion from the cloud to the edge. The red dashed vertical line represents the point at which the adaptivity process to fuse P_2 and P_3 begins. The new fused partition, P_5 takes over the P_3 graph after adaptation in the figure because P_3 was already running on the edge, and this enables us to more clearly see the effects of adaptivity.

We can see P_2 stops when the adaptivity is triggered, and does not produce any more data for the rest of the experiment, as expected. Looking at P_3/P_5 we can see that the new partition does not start generating events for around 11 seconds. This is the time to complete the new partition's deployment at the edge, load state from the key/value store, and then to process the partial window.

The new partition then follows expected patterns for input and output. The input data rate reduces after an initial rise. Given that the output throughput is limited by the ability for the edge device to process the CPU-intensive top-k function, this suggests that the internal queue filled to capacity, and so no more input could be requested until some processing had completed (the output event rate graph suggests this). Overall, the partition takes around 70 seconds to stabilize back to the pre-adaptation rate of event processing. Whilst this is a considerable period of time to catch back up, the data rate in this example has been deliberately set at a high level to show a worst-case scenario for the edge device.

5.2 Edge to Cloud Fission Results

The fission experiment is the inverse of the fusion experiment. We start with three partitions, where P_2 corresponds to the already fused partitions P_2 and P_3 from the fusion experiment's starting topology. Upon adaptation, we split P_2 into two new partitions. In this case, to see the behaviour more clearly within the figure, P_4 takes over the name of P_2 and starts at the same event count that P_2 finished at. P_3 starts from zero as a brand new partition. The effect of fission is shown in Figure 7.

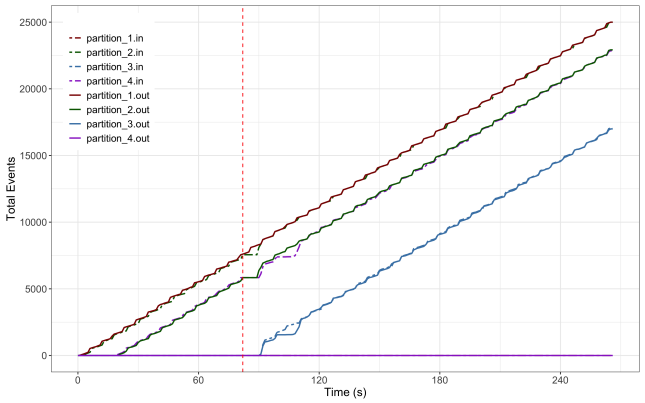

Figure 7: Fission to Cloud Results

Taking a closer look at P_2, once adaptivity is triggered the input message and output message count stops for around 8 seconds as the old partition is taken offline and the new one is started. We then see a spike in the input and output as the node catches up with backed up messages. P_3 starts at a similar time and follows the behaviour of P_2 as expected. It is clear in this example that the final topology stabilizes much faster than the previous fusion to edge experiment – from the point that adaptivity is triggered, approximately 30 seconds elapse before the system resumes normal operation. We attribute this to the greater processing power of the cloud node. This causes the new partition to start in less time, resulting in a smaller backlog of messages, and it also means that the backlog is processed more quickly.

6 RELATED WORK

There are many existing distributed real-time stream-processing systems, including Apache Flink [1], Apache Storm [23], Apache Heron [11], Apache Spark Streaming [24], Apache Samza [16], and Google Cloud Dataflow [13]. These typically follow a similar approach to *StrIoT*, in having high-level operators that perform stream transformations on the data. The implementation differs from system to system, usually to maximize some particular goal, such as correctness, throughput, latency, or some alternate heuristic. However, none have followed the distributed, pure functional approach explored by *StrIoT*.

Several purely functional streaming libraries already exist (e.g.[7, 12, 19, 22]) that typically focus on data ingestion and manipulation within a single machine, providing stream operators with similarities to those within *StrIoT*. However, *StrIoT* differs by supporting distributed stream-processing with a single definition of the overall stream processing graph as a program that is distributed and deployed across a set of nodes, and can then be automatically adapted at run-time. This is how fission and fusion can be supported.

Michalák and Watson demonstrated[14] that there can be advantages in automatically placing computation onto edge nodes, as

well as the cloud. The key difference for *StrIoT* is that it explores a functional, rather than relational, model for expressing, optimising and executing the computation. This results in many differences, particularly in the way the user expresses the computation and the design of the run-time system. We believe that the *StrIoT* functional approach offers a richer set of stream operations than are available in a relation streaming system such as *PATH2iot*.

With the popularity of Apache Spark, there are several areas of research focusing entirely on adapting the batch sizes of Apache Spark Streaming jobs in real-time [3, 4]. Whilst these are promising for workloads suited to batch processing, it is inevitable that extra latency is introduced as a result of batching. *StrIoT* differs in being a stream processing system that does not rely on batching, and that can distribute and adapt computations across distributed nodes.

There are also various investigations into the dynamic adaptation of streaming systems [2, 6, 8, 18] with an emphasis on resource utilization, through horizontal auto-scaling, as well as operator migration. However, *StrIoT* provides an additional set of real-time adaptations in the form of operator fission and fusion.

7 CONCLUSIONS

This paper has described how the *StrIoT* distributed stream processing system can enable efficient, dynamic adaptation at run-time without a loss of service. It has focused on adaptations that move computation between the cloud and the edge, combined with fission and fusion. However, others are also provided by *StrIoT*, including those that automatically parallelize operators in the cloud, including map and filter.

The main lessons from our work so far have been:

- the small set of purely functional operators we defined is sufficiently expressive to implement a wide range of stream processing applications
- a key advantage of *StrIoT* is that it supports partition fission and fusion. These are enabled because the partitions are automatically created by transforming the original program provided by the programmer
- the clear semantics of the set of purely functional operators enables the run-time system to adapt computations at run-time, without any knowledge of the specific application, and so without placing an additional burden on the programmer
- the overheads of performing an adaptation are low, generating temporary processing delays of only a few seconds for the two types of adaptations featured in the paper, both of which involve migrating computational state

8 FURTHER WORK

Work on *StrIoT* continues in a range of areas. There is continued development to support adaptivity across all combinations of the functional operators, enabling adaptation of complex topologies. There is the development of an optimizer that uses rewrite rules to generate alternative versions of a program, including different partitionings across cloud and edge nodes. This again exploits the advantage of having a small set of functional operators. A cost model enables *StrIoT* to automatically select the option that best meets non-functional requirements, with performance as the first focus. The cost model and a performance monitoring infrastructure will then form the basis for the automatic triggering of adaptations, such as those included in this paper, when the current deployment fails to meet requirements.

ACKNOWLEDGMENTS

Adam Cattermole is supported by the Engineering and Physical Sciences Research Council, Centre for Doctoral Training in Cloud Computing for Big Data [grant number EP/L015358/1]. Jonathan Dowland acknowledges Red Hat, Inc for supporting his contributions to this work.

REFERENCES

[1] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015), 28–38.

[2] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized self-adaptation for elastic data stream processing. *Future Generation Computer Systems* 87 (2018), 171–185.

[3] Dazhao Cheng, Xiaobo Zhou, Yu Wang, and Changjun Jiang. 2018. Adaptive scheduling parallel jobs with dynamic batching in spark streaming. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (2018), 2672–2685.

[4] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.

[5] Miyuru Dayarathna and Srinath Perera. 2018. Recent Advancements in Event Processing. *ACM Comput. Surv.* 51, 2, Article 33 (Feb. 2018), 36 pages. https://doi.org/10.1145/3170432

[6] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2013. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1447–1463.

[7] Gabriel Gonzalez. 2012. *pipes: Compositional pipelines.* Retrieved February 13, 2020 from https://hackage.haskell.org/package/pipes

[8] Muhammad Hanif, Eunsam Kim, Sumi Helal, and Choonhwa Lee. 2019. SLA-based adaptation schemes in distributed stream processing engines. *Applied Sciences* 9, 6 (2019), 1045.

[9] Kubernetes Inc. 2014. *Kubernetes.* Retrieved April 24, 2018 from https://kubernetes.io/

[10] Zbigniew Jerzak and Holger Ziekow. 2015. The DEBS 2015 Grand Challenge. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems* (Oslo, Norway) *(DEBS '15)*. ACM, New York, NY, USA, 266–268. https://doi.org/10.1145/2675743.2772598

[11] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream Processing at Scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2723372.2742788

[12] Harendra Kumar. 2017. *streamly: Beautiful Streaming, Concurrent and Reactive Composition.* Retrieved February 13, 2020 from https://hackage.haskell.org/package/streamly

[13] Google LLC. 2015. *Google Cloud Dataflow.* Retrieved Sept 17, 2019 from https://cloud.google.com/dataflow/

[14] Peter Michalák and Paul Watson. 2017. PATH2iot: A Holistic, Distributed Stream Processing System. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, IEEE, New Jersey, NJ, USA, 25–32.

[15] Microsoft. 2008. *Microsoft Azure: Cloud Computing Services.* Retrieved September 30, 2021 from https://azure.microsoft.com/

[16] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1634–1645. https://doi.org/10.14778/3137765.3137770

[17] Redis. 2009. *Redis.* Retrieved September 30, 2021 from https://redis.io/

[18] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kun-Lung Wu. 2009. Elastic scaling of data parallel operators in stream processing. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 1–12.

[19] Michael Snoyman. 2011. *conduit: Streaming data processing library.* Retrieved February 13, 2020 from https://hackage.haskell.org/package/conduit

[20] *StrIoT* authors. 2021. *StrIoT* — Stream Operators. Retrieved September 28, 2021 from https://github.com/striot/striot/blob/main/docs/Operators.md

[21] *StrIoT* authors. 2021. *StrIoT* — functional stream processing for IoT. Retrieved September 28, 2021 from https://github.com/striot/striot

[22] Michael Thompson. 2015. *streaming: an elementary streaming prelude and general stream type.* Retrieved February 13, 2020 from https://hackage.haskell.org/package/streaming

[23] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, and et al. 2014. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[24] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737