# Model-based Stream Processing Auto-scaling in Geo-Distributed Environments

Hamidreza Arkian, Guillaume Pierre, Johan Tordsson, Erik Elmroth

# Model-based Stream Processing Auto-scaling in Geo-Distributed Environments

HamidReza Arkian
*Univ Rennes, Inria, CNRS, IRISA*
hamidreza.arkian@irisa.fr

Guillaume Pierre
*Univ Rennes, Inria, CNRS, IRISA*
guillaume.pierre@irisa.fr

Johan Tordsson
*Elastisys and Umeå University*
johan.tordsson@elastisys.com

Erik Elmroth
*Elastisys and Umeå University*
erik.elmroth@elastisys.com

***Abstract*—Data stream processing is an attractive paradigm for analyzing IoT data at the edge of the Internet before transmitting processed results to a cloud. However, the relative scarcity of fog computing resources combined with the workloads' non-stationary properties make it impossible to allocate a static set of resources for each application. We propose *Gesscale*, a resource auto-scaler which guarantees that a stream processing application maintains a sufficient Maximum Sustainable Throughput to process its incoming data with no undue delay, while not using more resources than strictly necessary. Gesscale derives its decisions about when to rescale and which geo-distributed resource(s) to add or remove on a performance model that gives precise predictions about the future maximum sustainable throughput after reconfiguration. We show that this auto-scaler uses 17% less resources, generates 52% fewer reconfigurations, and processes more input data than baseline auto-scalers based on threshold triggers or a simpler performance model.**

***Index Terms*—Stream processing, auto-scaling, fog computing.**

## I. Introduction

The volume of data produced by Internet of Things (IoT) devices and end users is rapidly increasing. It is expected that, by 2025, 75% of all enterprise data will be produced far from the data centers [1]. These data are often generated as uninterrupted streams that must be analyzed as quickly as possible after being produced [2]. Data Stream Processing (DSP) frameworks are often used as a middleware to process such streams of data [3].

When input data are produced at the edge of the Internet, transferring them to a cloud data center where they can be processed is becoming increasingly impractical or infeasible [4]. To reduce the pressure on long-distance network links, geo-distributed platforms such as fog computing platforms are therefore being designed to extend traditional cloud systems with additional compute resources located close to the main sources of data [5]. However, managing data stream processing frameworks in geo-distributed environments remains a difficult challenge [6], [7].

A difficult and important issue faced by geo-distributed stream processing systems is that long-running IoT applications produce variable amounts of data, with significant fluctuations occurring over time [8]. Statically configuring the DSP frameworks according to their expected peak load would essentially bring these services back to a pre-cloud era where each application had to be provisioned individually with its own dedicated hardware. However, DSP frameworks were not originally designed with the necessary elasticity to dynamically adjust their resource usage to the current workload conditions [9]. As a result, any runtime DSP resource reconfiguration remains a costly operation [10].

A second issue is that stream processing applications are designed as complex workflows of data stream processing operators. Each logical operator may be replicated and distributed over multiple servers in different locations. As a consequence, stream processing applications are not monolithic entities that must be scaled up and down as a single unit, but a set of individual components that should be controlled individually.

Last but not least, fog computing networks are known to be highly heterogeneous [5]. The performance and efficiency of a fog platform is thus strongly influenced by the choice of the fog computing servers to execute any stream processing operator's replicas, and their specific locations within the fog computing platform [11].

This paper presents Gesscale (GEo-distributed Stream autoSCALEr), an auto-scaler for stream processing applications in geo-distributed environments such as fogs. Gesscale continuously monitors the workload and performance of the running system, and dynamically adds or removes replicas to/from individual stream processing operators, to maintain a sufficient Maximum Sustainable Throughput (MST) while using no more resources than necessary. MST is a standard measure of the stream processing system's capacity to process incoming data with no undue queuing delay [12]–[14]. Gesscale relies on an experimentally-validated performance model that gives precise estimates of the resulting performance from any poten-

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

tial reconfiguration [11]. This allows Gesscale to reduce the number of reconfigurations compared to a simple threshold-based auto-scaler. This is particularly important when scaling the system down, as a good performance model is the only way to accurately identify the moment when resources may be removed without violating the MST requirement.

We base our experiments on a real fog computing testbed, and the popular Apache Flink DSP engine [15]. Our evaluations show that Gesscale produces 52% fewer reconfigurations and processes more data than a baseline threshold-based auto-scaler, while using 17% fewer resources.

The remainder of paper is organized as follows. Section II presents technical background. Section III discusses related works. Then Section IV details the design of Gesscale, and Section V evaluates it. Finally, Section VI concludes.

## II. Background

### A. Data stream processing in Fog platforms

Data stream processing systems were created to continuously process unbounded incoming data streams with low end-to-end latency [16]. In recent years, numerous stream processing engines (SPEs) have been proposed, including Apache Storm [17], Apache Spark [18] and Apache Flink [15]. DSP applications are designed as directed acyclic graphs of data transformations (*operators*) that are connected to each other by *data streams*, and that together form a data processing workflow. The inputs are usually composed of data stream items (*tuples*) that every operator consumes by applying a predefined computation and which produces new tuples that are forwarded to the next operator(s).

To guarantee a satisfactory quality of service, DSPs use a variety of techniques to parallelize the execution of their operators [19]. In most cases, replicating a stream processing operator increases its processing capacity and improves its quality of service (QoS) [20]. Therefore, many DSP performance improvement approaches focus on parallelization and elasticity strategies [3].

SPEs were initially designed for big-data analytics applications running in a cluster or a data center. Despite their distributed computation model they still lack the necessary elasticity to handle the requirements that derive from IoT/fog computing scenarios [21]. The main characteristic of fog computing environments is the geographically distribution of computing nodes, which in turn causes heterogeneous inter-node network latencies [22]. In such environments neither the DSP engines nor their users can easily predict the performance that would result from a reconfiguration of the DSP system using geo-distributed resources.

To handle non-stationary workloads in IoT/fog computing environments, *self-adaptive DSP systems* are being developed [23], [24]. They are characterized by their ability to adapt themselves to changes in their execution environment and internal dynamics to continue to achieve their objectives.

In this paper, we propose an auto-scaling system based on the MAPE (Monitor, Analyze, Plan, Execute) loop model from Autonomic Computing [25]. Using this pattern, Gesscale can locally control the adaptation of single DSP operators via a feedback controlling loop. As shown in Figure 1, Gesscale continuously monitors the workload and performance of the running application and its execution environment. When the current set of resources is insufficient to process the incoming data without delay, it selects additional nodes in the fog computing environment and rescales Apache Flink to extend to these new resources. Conversely, it releases the unnecessary resources as soon as the decreasing workload may be processed with fewer replicas.

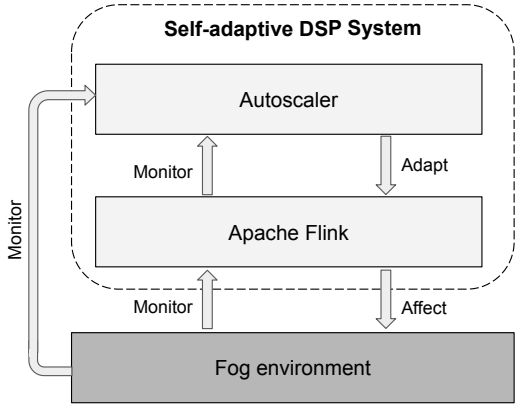



Fig. 1. Self-adaptive data stream processing in fog.

### B. Flink's behavior in overload situations

An important aspect of any auto-scaler is to accurately detect and analyze overload situations of its controlled system. Unlike many Web-based systems that can be accurately modeled with queuing theory [26], Flink automatically slows down its operations to handle overload situations.

Flink applications are directed acyclic graphs of operators through which data are streamed and processed. When one of these operators is unable to sustain the same throughput as the others, it may compromise the entire workflow as potentially large amounts of data may need to be queued until they can be processed by the overloaded operator. As illustrated in Figure 2, when operator $Op_{i-1}$ produces data faster than its downstream operator $Op_i$ can consume, it locally buffers the undelivered tuples and starts building *back pressure* to slow itself down, as well as its own upstream operators. The presence of back pressure therefore indicates that at least one operator in the workflow has reached its maximum processing capacity.

Although this mechanism effectively avoids operator overload and ensures that no intermediate data is discarded for lack of queuing space, it also means that simple resource utilization metrics such as CPU utilization cannot be used to detect operator overload. Similarly, the presence of back pressure in one operator is not a sufficient indication to pinpoint operator overload, as this back pressure may either have been caused by the operator itself or propagated from downstream operators.

We therefore identify that a specific operator is experiencing overload if it does not experience back pressure itself, but all

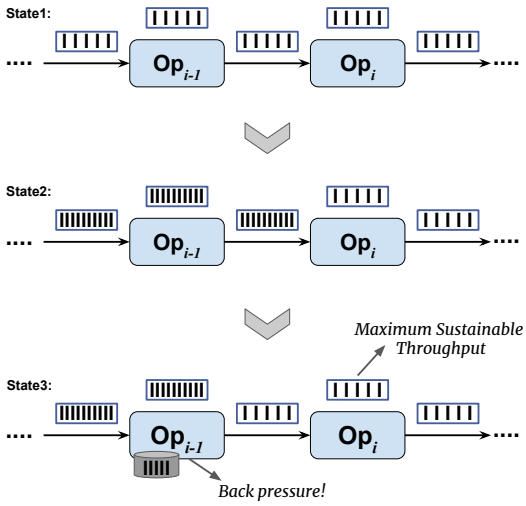



Fig. 2. Back pressure in a stream processing workflow. Initially operator $Op_i$ processes data at the same rate as $Op_{i-1}$ produces it (State 1). When operator $Op_{i-1}$ outputs data faster than operator $Op_i$ can consume (State 2), it locally buffers the undelivered tuples and starts building *back pressure* (State 3).

of its upstream operators do. Note that the actual throughput of the overloaded operator is a good measure of the operator's Maximum Sustainable Throughput (MST). Back pressure is measured as a ratio $BP \in [0; 1]$. In this work we define "high level of back pressure" as $BP > 0.5$.

### C. Runtime Flink reconfigurations

Apache Flink was unfortunately not designed as an elastic platform capable of dynamically adding or removing compute resources [3]. As a consequence, when Gesscale decides to add or remove compute resources to/from a running application, it needs to stop Flink and rely of Apache Kafka to reliably buffer incoming data before entering Flink. After restarting Flink with a different resource configuration, processing may resume without any data loss. This operation however takes time during which no data can be processed.

Figure 3 depicts a Flink operator's throughput before, during and after a resource reconfiguration. Initially the system processes incoming data as soon as it has been produced. From time $t = 650\,s$ the workload increases beyond the system's Maximum Sustainable Throughput, which triggers a resource reconfiguration at $t = 800\,s$. We observe that the system throughput drops to zero during about 120 seconds before restarting with a significantly greater throughput. This illustrates the importance of reducing the number of resource reconfigurations as much as possible.

Note that Flink's long reconfiguration times are not a fundamental limitation of this system. A declarative resource management capability and the corresponding scheduler are planned to be integrated in Flink-1.13, bringing concrete perspectives of significant reconfiguration time reductions in the near future [27], [28]. This new feature does not solve the problem we are addressing, but it rather makes it even more urgent to solve the auto-scaling problem.

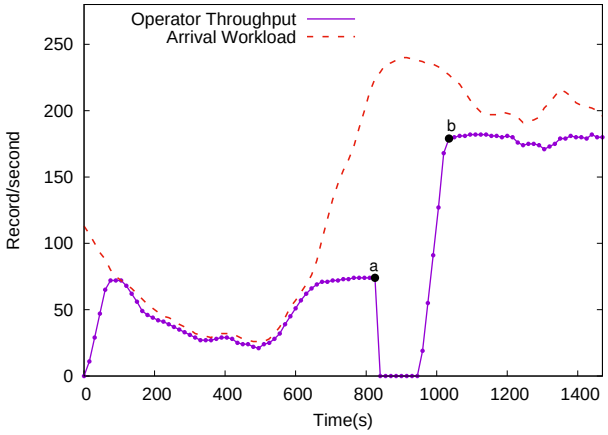



Fig. 3. Flink's throughput upon a resource reconfiguration.

### D. Deploying Flink in geo-distributed environments

Deploying Flink in a geo-distributed fog environment requires the usage of a suitable resource orchestrator. We rely on the popular open-source Kubernetes (k8s) container orchestrator. Although it was initially designed for cluster and cloud environments, it is now being adapted to handle fog computing scenarios as well [29]–[32]. Kubernetes provides a variety of mechanisms to simplify the deployment, scaling and management of containerized applications in distributed environments by managing their complete life-cycle.

A Flink system is composed of one *JobManager* (in charge of managing the entire system) and a number of *TaskManagers* (in charge of processing data). To select in which Fog server to deploy a specific *TaskManager*, we maintain a map of inter-node latencies (i.e., a table of peer-to-peer network latencies between the nodes which can be obtained dynamically or provided statically), and attach the deployment requests given to Flink with Kubernetes' *nodeSelector* and affinity/anti-affinity rules which constrain their deployment in the chosen server.

## III. State of the Art

Numerous approaches have been proposed to control the elasticity of DSP systems in virtualized infrastructures. They differ in the environment (cluster, cloud, fog) where the DSP is deployed, the type of data that are monitored, the quality-of-service objective which is targeted, and the optimization method which is utilized [3].

Elasticity of stream processing systems has been well studied in the domain of Cloud computing where compute resources are homogeneous and connected with one another with almost negligible network latency [33]–[39]. However, working in geo-distributed environments like fog computing platforms exposes very different features where the environment is heterogeneous in terms of network latency. Heterogeneous network latency has important effects on stream processing performance which cannot be safely ignored [11].

Most works on geo-distributed stream processing auto-scaling aim to minimize the end-to-end workflow latency [40]–[44]. They use different monitored metrics and methodologies,

TABLE I
STATE OF THE ART OF GEO-DISTRIBUTED DSP AUTO-SCALING.

| | | QoS objective | | |
|---|---|---|---|---|
| | | **Latency** | **Throughput** | **Cost** |
| **Method** | **Threshold** | — | [33], [45]–[47] | [34] |
| | **Reinf. learning** | [40]–[42] | — | — |
| | **Control theory** | — | [48] | — |
| | **Model** | [35], [36], [43], [44] | ***Gesscale*** | — |

either at workflow-level or operator-level. A cost model is also usually considered as a side condition which describes the system cost in terms of used resources or adaptation cost [34].

Conversely, some other solutions aim to ensure sufficient throughput as their QoS objective. Threshold-based approaches compare the current throughput against a set of thresholds [33], [45]–[47]. These thresholds are usually experimentally determined using profiling methods [3]. While thresholds facilitate scaling decisions, finding appropriate threshold values for a wide range of applications and different states of a geo-distributed infrastructure can be very difficult. Conversely, in this work we aim to design an auto-scaler which does not depend on such arbitrary parameters.

Another class of solutions exploits machine learning techniques to recognize patterns from measured or profiled data [40]–[42]. Patterns are usually refined at runtime to improve precision. However, these systems must be trained for potentially very long periods of time before they can deliver accurate scaling decisions.

Finally, some auto-scalers base their decisions on a performance model to calculate the configuration that can best satisfy the defined objective. To our best knowledge, all systems in this category aim to optimize the end-to-end workflow latency [35], [36], [43], [44]. Model-based auto-scaling has the potential for delivering the most accurate decisions, depending on the prediction accuracy of the used model.

Table I summarizes and classifies these different works according to their QoS objective and the method which is used to reach this objective. As we can see, the Gesscale auto-scaler proposed in this paper is the only one which aims to optimize system throughput using a model-based approach.

## IV. SYSTEM DESIGN

### A. Design principles

Although a stream processing application typically consists of a complex workflow with multiple operators, the problem of auto-scaling such complex workflows can be split in a number of independent sub-problems. Because our QoS objective is to optimize throughput rather than other metrics such as end-to-end processing latency, the auto-scaling decisions can be made individually for each operator in isolation from the rest of the application. It is indeed sufficient that each operator provides sufficient throughput to process its own workload without creating undue delays which may indirectly affect other operators. We can therefore focus our attention to the auto-scaling of a single operator, with the assumption that all other operators will be auto-scaled using the same techniques.

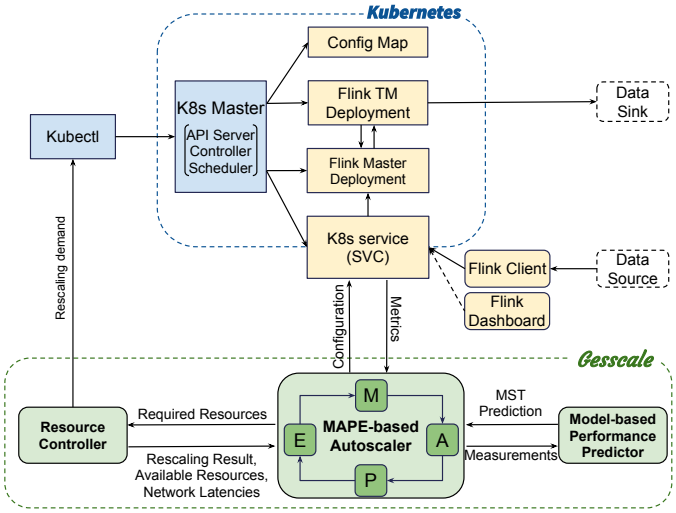



Fig. 4. System architecture.

We design the single-operator auto-scaler following the classical *Monitor, Analyze, Plan, Execute (MAPE)* loop architectural pattern which is the design basis for large numbers of self-adaptive systems [25].

Figure 4 shows the main architectural components of Gesscale. The core element is a MAPE-based auto-scaler which continuously monitors the performance of the running Apache Flink system, and which collaborates with a model-based performance predictor to accurately determine the required throughput and sufficient resources at operator-level. Once rescaling decisions have been made, they are implemented by a resource controller which triggers Kubernetes by sending a rescaling demand to issue the requested adaptation. The following sections respectively describe the four phases of this MAPE-based auto-scaler.

### B. Monitor

Any auto-scaler bases its decisions on up-to-date information about the current system performance. In our case, we need to know the incoming data rate and current operator throughput, as well as information about the system's computing resources in terms of their availability and network latencies with each other.

The system starts monitoring different metrics as soon as Apache Flink starts to run an application. In the initialization step, the system fetches general information such as JobID, all operators IDs, and the parallelism level of every operator. Then, it periodically monitors the current throughput and back pressure level of every operator at a 1-minute granularity. We found this periodicity implements a good trade-off between reactiveness and stability of the system.

### C. Analyze

The Analyze phase is in charge of determining when a scale-up or a scale-down decision is necessary, and of automatically calibrating the parameters of the operator performance model.

TABLE II
NOTATIONS USED IN THE PERFORMANCE MODEL.

| Symbol | Description |
|---|---|
| $MST_i^n$ | Maximum sustainable throughput of operator $i$ with $n$ replicas. |
| $BP_i$ | Back pressure level of operator $i$. |
| $CT_i$ | Current throughput of operator $i$. |
| $DT_i$ | The difference between current throughput and MST of operator $i$. |
| $Trshld_{DT}$ | Predefined threshold for $DT$. |
| $RCF_i$ | Reconfiguration state of operator $i$. |
| $IR_i$ | Input rate of operator $i$. |
| $\alpha$ | maximum sustainable throughput of a single node. |
| $\beta$ | SPE (Apache Flink) parallelization inefficiency. |
| $\gamma$ | Effect of network delays. |
| $ND_{max}$ | Maximum network delay between nodes. |

*1) Performance model and its calibration:* Gesscale relies on our previously-published experimentally-validated performance model to predict the throughput of stream processing operators in a wide range of configurations, including varying the number of replicas used to run them and the network latency between them [11]. Without entering into full details, the performance model for a single operator is as follows:

$$MST = \alpha \times n^{\beta} - \gamma \times \mathrm{ND}_{max} \tag{1}$$

where *MST* is the operator's *maximum sustainable throughput*, $n$ is the number of resource units used to execute the operator, and $\mathrm{ND}_{max}$ is the greatest network round-trip latency among these resource units. The model is fully parameterized with three parameters $\alpha$, $\beta$ and $\gamma$ which respectively characterize the *MST* of a single node, Flink parallelization inefficiency, and the effect of network latency on the operator's throughput, respectively. The model's notation are summarized in Table II.

As extensively discussed in [11], the values of $\alpha$, $\beta$ and $\gamma$ can be derived from at least three experimental measurements of the operator's *MST* in different resource configurations. These measurements must be made during time periods where the operator is working at its maximum capacity. In other terms, a parameter calibration measurement can be made only at the time the operator is about to be scaled up.

When fewer than three *MST* measurements are available to calibrate the performance model, we can use simplified, yet less accurate, versions of the model. With a single measurement we can fit the value of $\alpha$ (the most important model parameter) to the experimental data, and keep default values $\beta = 1$ and $\gamma = 0$. This simplifies the model as follows:

$$MST_n = \alpha \times n \tag{2}$$

When a second measurement is available we can fit the $\alpha$ and $\gamma$ parameters, and keep the default value $\beta = 1$:

$$MST_n = \alpha \times n - \gamma \times \mathrm{ND}_{max} \tag{3}$$

As soon as at least three *MST* measurements are available, we obtain a fully-calibrated performance model which delivers *MST* predictions for a wide range of resource configurations with less than 2% error.

**Algorithm 1** Scale-up analysis.

**Input1:** Back pressure levels of operators ($BP_i$ & $BP_{i-1}$).
**Input2:** Current throughput of $Op_i$ ($CT_i$).
**Input3:** Number of replicas of $Op_i$ ($n_i$).
**Output1:** Maximum sustainable throughput of $Op_i$ with parallelism $n$ ($MST_i^n$).
**Output2:** A Boolean variable which shows if $Op_i$ is the root of back pressure or not ($Op_i^{root}$).
**Output3:** A Boolean variable which shows if $Op_i$ needs reconfiguration or not ($RCF_i$).
**if** $BP_{i-1} > 0.5$ **and** $BP_i \leqslant 0.5$ **then**
    $Op_i^{root} \leftarrow true$
    $RCF_i \leftarrow true$
    $MST_i^n \leftarrow CT_i$
**else**
    $Op_i^{root} \leftarrow false$
    $RCF_i \leftarrow false$
    $MST_i^n \leftarrow null$
**end if**
**return** $Op_i^{root}$, $RCF_i$, $MST_i^n$

*2) Deciding when to scale up:* When an operator incurs high back pressure, we know that one of its downstream operators is overloaded. We identify the overloaded operator as the one that does not have high back pressure itself whereas all its upstream operators do. Scaling this operator up should increase its capacity. Scale-up analysis is presented as Algorithm 1.

*3) Deciding when to scale down:* An operator should be scaled down when it under-utilizes its resources and it would be able to sustain the current data throughput with fewer resources. However, there is no simple metric which unambiguously indicates this condition. In particular, low CPU utilization is a poor predictor. Instead, we use the performance model to determine the *MST* that the operator *would have* if it used fewer resources than the current configuration. If this *MST* is greater or equal than the current operator's throughput, then at least one resource unit can be removed without violating the QoS objective. Note that this method also allows us to determine how many resource units may be safely removed (which may be more than one in case the traffic intensity is decreasing quickly). Scale-down analysis is presented as Algorithm 2.

### D. Plan

The Plan phase is in charge of identifying the new resource configuration which should be used to maintain the QoS objective. In a fog computing platform, the heterogeneous network performance between nodes implies that the choice of specific nodes to execute an operator influences the resulting system performance. Such decisions are taken with the help of the performance model as well as the resource controller.

*1) Scaling up:* Scale-up planning is presented as Algorithm 3. When a stream processing operator experiences overload as detected by the Analyze phase, we know that at least one additional resource is necessary to increase its processing capacity. We however have no way to accurately determine the throughput objective that should be reached

**Algorithm 2** Scale-down analysis.

**Input1:** Current throughput of $Op_i$ ($CT_i$).
**Input2:** Number of replicas of $Op_i$ ($n_i$).
**Input3:** Threshold for the difference between current throughput and MST of $Op_i$ ($Trshld_{DT}$).
**Output:** A Boolean variable which shows if $Op_i$ needs reconfiguration or not ($RCF_i$).
$MST_i^n \leftarrow PerfModel(n, ND_{max})$
$DT_i \leftarrow 100 \times \frac{MST_i^n - CT_i}{MST_i^n}$
**if** $DT_i \geqslant Trshld_{DT}$ **then**
    $RCF_i \leftarrow true$
**else**
    $RCF_i \leftarrow false$
**end if**
**return** $RCF_i$

**Algorithm 3** Scale-up planning.

**Input:** Maximum acceptable number of replicas ($n_{max}$).
**Output:** Next number of replicas for $Op_i$ ($n_{next}$).
**if** $n + 1 \leqslant n_{max}$ **then**
    $n_{next} \leftarrow n + 1$
**end if**
**return** $n_{next}$

to resolve the situation. We can therefore only increase the number of resource units by one, and observe whether this is sufficient to reduce the operator's overload situation. In case the problem is not solved yet, then we can repeat the operation and add other resource units one by one until the operator reaches the required throughput.

During the scale-up operations, the performance model is useful only to select which resource should be added in case more than one is currently available. In particular, in case the cost of new resource units is taken into account in the QoS objectives, the performance model can deliver accurate estimates of the future throughput with the chosen resource, which allows the system to decide whether the cost/benefit ratio is sufficient to trigger this reconfiguration.

*2) Scaling down:* Scale-down planning is presented as Algorithm 4. When scaling down, the goal of the auto-scaler is to remove as many resource units as possible without reducing the *MST* below the current system throughput value. The performance model delivers these estimates so the auto-scaler can obtain accurate information about the performance consequences of executing the operator with fewer resources.

Note that, in case of a large drop in the throughput demand, Gesscale may remove more than one resource at a time if the remaining ones are sufficient to handle the current workload. In contrast, any threshold-based auto-scaler would have to issue multiple scale-down operations one by one, thereby producing a greater number of system reconfigurations.

### E. Execute

The Execute phase is in charge of executing the resource reconfiguration decisions taken by the Plan phase. It must address two challenges: first it needs to obtain an up-to-date list of available resources; this list may be fetched from

**Algorithm 4** Scale-down planning.

**Input1:** Minimum acceptable number of replicas ($n_{min}$).
**Input2:** Threshold for the difference between current throughput and MST of $Op_i$ ($Trshld_{DT}$).
**Input3:** Input rate of operator $Op_i$ ($IR_i$).
**Output:** Next number of replicas for operator $Op_i$ ($n_{next}$).
**while** $MST_i^n \leqslant IR_i$ **and** $DT_i^n \geqslant Trshld_{DT}$ **and** $n \geqslant n_{min}$ **do**
    $n \leftarrow n - 1$
    $MST_i^n \leftarrow PerfModel(n, ND_{max})$
    $DT_i^n \leftarrow 100 \times \frac{MST_i^n - CT_i}{MST_i^n}$
**end while**
$n_{next} \leftarrow n$
**return** $n_{next}$

Kubernetes. The resource controller then sorts the available resources based on the location of the operator $i$ that needs to be rescaled and according to their network latencies to the existing resources. When scaling up we want to choose the resource with lowest network latency to the other resources, whereas when scaling down we want to choose the resource with greatest network latency to the other resources.

Second, the auto-scaler needs to trigger a change in the resource configuration used by Apache Flink. Unfortunately, Flink is currently not capable of dynamically integrating additional resources nor of removing some of its resources seamlessly. We therefore need to stop the Flink system and restart it with a different configuration.

The auto-scaler first stops Flink's running job with a *savepoint* which checkpoints the current system state. The resource controller triggers kubectl to stop all TaskManagers by rescaling the deployment to zero replica. It then rescales again the TaskManagers deployment based on the new requested scale and choice of resources. Finally the auto-scaler reconfigures the Flink application execution model and restarts Flink's job from the *savepoint* with a different set of resources.

This operation maintains the workflow processing correctness (i.e., "it does not lose data"), but it temporarily interrupts the processing workflows. This means that resource reconfigurations are expensive operations which should be issued as rarely as possible. As previously discussed, note that future versions of Flink are expected to have this capability which should significantly reduce the resource reconfiguration times. We show in the next section the importance of basing the auto-scaling decisions on an accurate performance model.

## V. Evaluation

### A. Experimental Setup

We conduct evaluations using a cluster of ten RaspberryPi4 single-board computers equipped with 2 GB of RAM each, quad-core ARMv7 processor, and Raspbian GNU/Linux v10. This type of machines is frequently utilized to prototype fog computing systems [31], [49], [50]. The cluster is powerful enough as a testbed and it is also horizontally and vertically scalable.

We use Kubernetes across this cluster to deploy variable numbers of Flink *TaskManagers* as well as other required

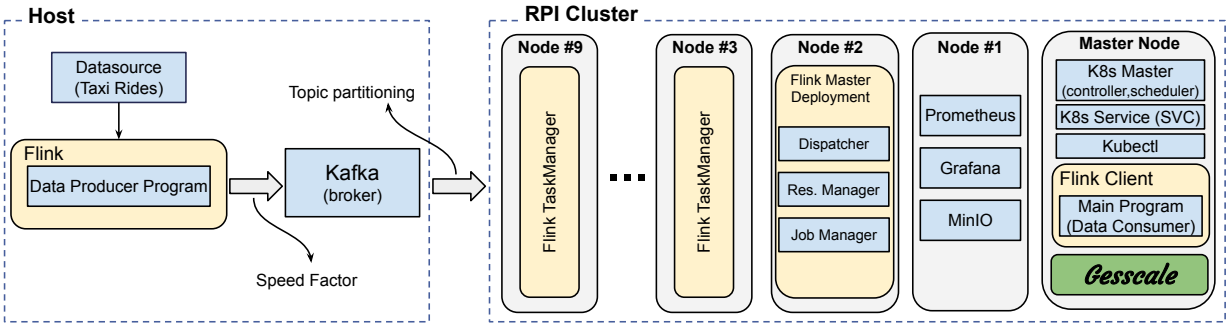



Fig. 5. Organization of the experimental testbed.

tools. One node in the cluster acts as the Master node where we deploy Kubernetes management services as well as Flink client and the auto-scaler. A second node is used to run the MinIO data storage service and the Prometheus and Grafana monitoring tools. The remaining nodes act a separate servers of the fog computing platform.

We emulate heterogeneous network latencies between 0 and 300 ms across the fog nodes using the Linux tc (traffic control) command.

*a) Flink Integration:* We use Apache Flink 1.12.0. As shown in Figure 5, we deploy Flink's *Jobmanager* and *TaskManagers* in separate nodes of the cluster. Without loss of generality, we configure Flink to execute a single stream processing operator in each *TaskSlot*, and each *TaskManager* is configured with a single *TaskSlot*.

*b) Application and workload:* We evaluate Gesscale using a real-world non-stationary workload derived from a dataset with records of four years of taxi operations in New York City [51]. Each record in the dataset represents one taxi trip including the date, time, and coordinate of each pickup or drop-off as well as the driver, taxi, and ride IDs. We used the first two days of May 2013, and cleaned up the data by removing duplicate and invalid entries. The resulting dataset includes roughly one million taxi rides.

We process this data set with a Data Producer and a Data Consumer. The Data Producer program (which is located out of the RPI cluster) generates a data stream of taxi ride records which are read from the dataset file and then writes them into an Apache Kafka broker. The Data Producer serves events according to their timestamps, with an adjustable speed factor. We use *SpeedFactor=10* in our experiments which means that 10 s of real-world events are replayed in 1 s. To generate sufficient workload without reducing the experiment duration too much, we inject three identical records in the Flink system for each record in the dataset.

The Data Consumer, implemented as an Apache Flink operator, extracts the pickup location (latitude, longitude), calculates the Euclidean distance between this pickup location and three major touristic hotspots in New York City, and returns a record which contains the original trip information extended with the name and distance of the closest attraction.

*c) Auto-scaling baselines:* We evaluate the effectiveness of Gesscale and compare its performance to three other baseline algorithms which resemble systems proposed in the state of the art.

- `THR-NLUnaware` is a simple threshold-based auto-scaler. The scale-up threshold is defined as a CPU utilization greater than 90%, and the scale-down threshold as a CPU utilization lower than 50%. Every time one threshold is reached, `THR-NLUnaware` adds or removes one resource to/from the system. `THR-NLUnaware` is latency-unaware so it chooses resources randomly out of an unsorted list.
- `THR-NLAware` is another threshold-based auto-scaler which uses the same thresholds as `THR-NLUnaware`. In contrast, `THR-NLAware` is aware of the network latencies between available resources. Hence, upon a scale-up operation it chooses the available resource with lowest latency with the other nodes, and upon scale-down it removes the resource with greatest latency to the other nodes.
- `MDL-NLUnaware` is a simplified version of Gesscale which relies on the latency-*unaware* performance model presented in Equation 2. `MDL-NLUnaware` therefore chooses resources to be added to or removed from the system out of an unsorted list of available resources.

*d) Evaluation metrics:* We evaluate Gesscale and compare its performance to the three aforementioned baseline algorithms using a variety of metrics which highlight their respective performance and costs.

**Accuracy** is a standard metric proposed by the SPEC RG Cloud Working Group which computes the difference between the parallelism levels chosen by the evaluated algorithms and that of an "ideal" auto-scaler [52]. Figure 6 shows the behavior of our reference "ideal" autocaler: it instantly adjusts the number of resources assigned to the considered stream processing operator to the incoming workload. As a consequence, the processed number of records/s and the number of replicas closely follow the incoming ride events' workload. Note that this auto-scaler cannot be implemented in practice because it assumes that any change of configuration is applied immediately with no period of unavailability. According to SPEC: *"The accuracy metric is divided in two sub-metrics: the under-provisioning accuracy*

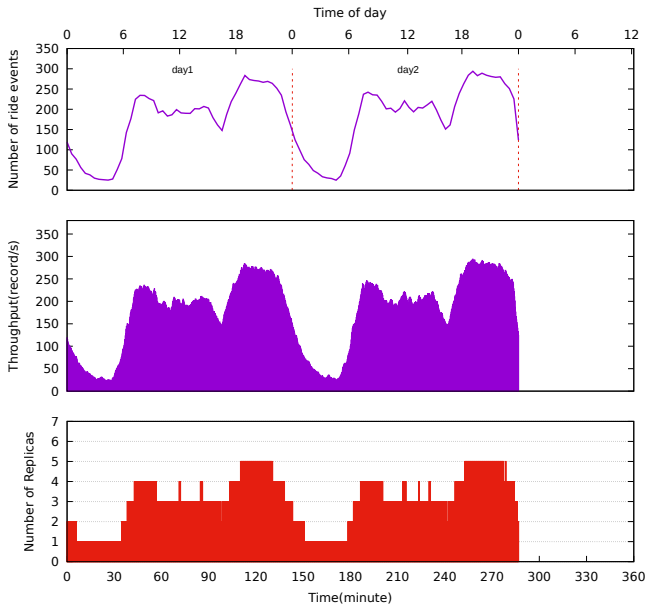



Fig. 6. "Ideal" auto-scaling strategy.

*metric* accuracy$_U$ *is calculated as the sum of areas* ($\Sigma_U$) *where the resource demand exceeds the supply normalized by the duration of the measurement period* $T$. *Accordingly, the over-provisioning accuracy metric* accuracy$_O$ *bases on the sum of areas* ($\Sigma_O$) *where the resource supply exceeds the demand:"*

$$\begin{cases} accuracy_U = \frac{\Sigma_U}{T} \\ accuracy_O = \frac{\Sigma_O}{T} \end{cases}$$

The lower these two metrics are, the closer the evaluated auto-scaling algorithm is from the "ideal" strategy.

**Provisioning Timeshare:** According to SPEC: *"the two accuracy metrics allow no reasoning whether the average amount of under-/over-provisioned resources results from a few big deviations between demand and supply or if it is rather caused by a constant small deviation. The* Provisioning timeshare *metrics* $timeshare_U$ *and* $timeshare_O$ *are standard SPEC metrics computed by summing up the total amount of time spent in an under-*($\Sigma_A$) *or over-provisioned* ($\Sigma_B$) *state normalized by the duration of the measurement period. Thus, they measure the overall timeshare spent in under- or over-provisioned states."*

$$\begin{cases} timeshare_U = \frac{\Sigma_A}{T} \\ timeshare_O = \frac{\Sigma_B}{T} \end{cases}$$

**Excess computation time:** An important characteristic of data stream processing is that every input record will be processed eventually, assuming that the system has sufficient data buffering capacity. However, records may have to be buffered for extended durations in case the stream processing system was underprovisioned and/or it spent too much time being reconfigured. The Excess computation time measures this effect by evaluating the necessary amount of time to finished processing the remaining buffered records after the incoming workload dropped to zero at the end of the trace:

$$Excess\ time = \frac{P_t - I_t}{T}$$

where $I_t$ is the ideal execution time, and $P_t$ is actual execution time. The resulting value is also normalized by the duration of the measurement period $T$ and the lower the excess reconfiguration time, the better.

**Number of reconfigurations:** considering the cost of any resource reconfiguration, a good auto-scaler should aim to reconfigure only when this is absolutely necessary. The lower the number of reconfigurations, the smaller the amount of time during which the system cannot process incoming data.

**Cost** evaluates the amount of used resources. We use a very simple cost model where each replica is charged one cost unit per minute of execution.

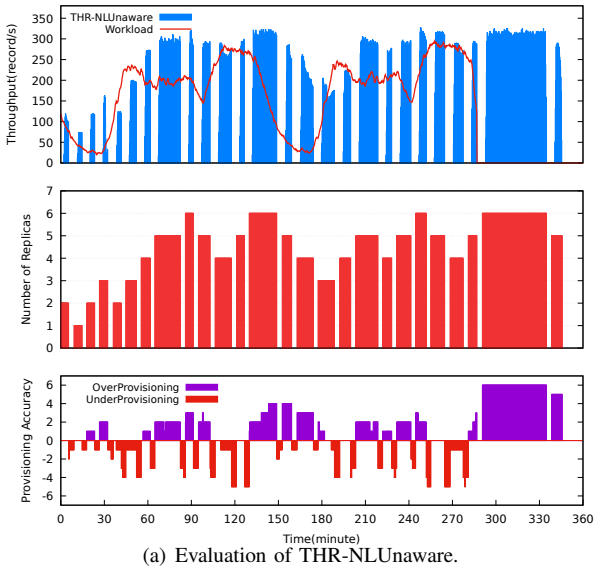
### *B. Auto-scaling effectiveness*

Figure 7 compares the behavior of the four auto-scaling algorithms. For each algorithm, the first graph depicts the incoming workload as well as the system's processing throughput. The throughput drops are caused by reconfiguration operations. The second graph shows the number of replicas chosen by the algorithms, and the third graph shows the provisioning accuracy compared to the "ideal" auto-scaling strategy (zero values in this chart represent perfect provisioning accuracy).

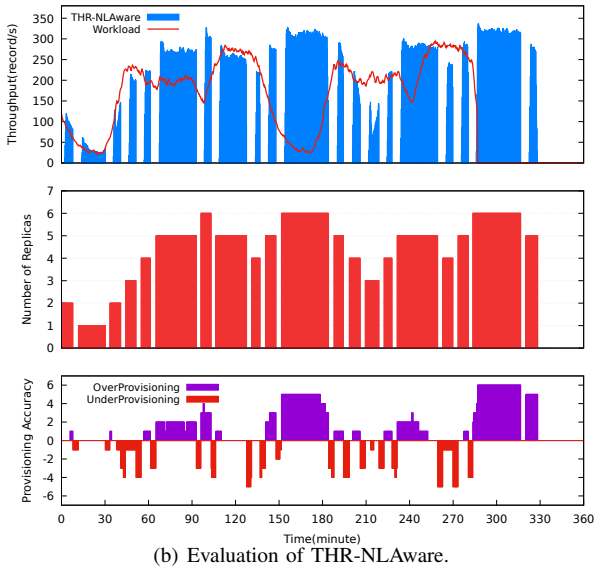
We observe two periods at time $t = 30$ and $t = 180$ where the workload strongly increases before starting to oscillate. All auto-scalers react by gradually increasing the number of replicas. We can however see that the latency-aware auto-scalers eventually create fewer replicas than their latency-unaware counterparts. This is because they choose the resources that are going to provide the greatest performance gains, whereas the latency-unaware auto-scalers select resources randomly. Also, the latency-aware algorithms generate fewer reconfigurations during these phases, which results in smaller amount of incoming data being buffered during reconfiguration, and eventually lower amounts of excess computation time.

We also observe periods of workload decrease: small decreases at time $t = 100$ and $t = 220$, and much stronger decreases at $t = 120$ and $t = 270$. Here the main difference is between the threshold-based and the model-based algorithms. Threshold-based algorithms have no way to identify the new correct number of replicas so they remove replicas one by one. Conversely, the performance model gives precise indications about the necessary number of replicas, so the model-based algorithms may remove more than one replica at a time, thereby reducing the number of reconfigurations and reducing the excess computation time.

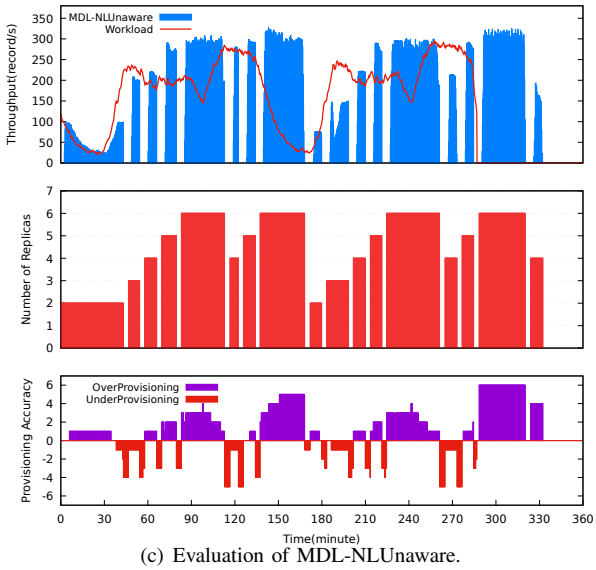
We also observe that the model-based auto-scalers identify the time when they should remove replicas, whereas threshold-based auto-scalers often trigger their reconfiguration too early, too late, or at a time when no reconfiguration is necessary. This is particularly visible in THR-NLaware at time $t = 130$ where a decision to scale down was inaccurate and had to be



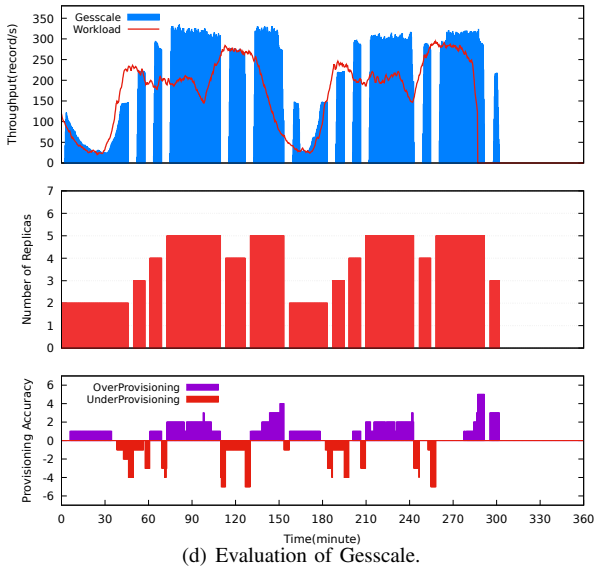

(a) Evaluation of THR-NLUnaware.




(b) Evaluation of THR-NLAware.




(c) Evaluation of MDL-NLUnaware.




(d) Evaluation of Gesscale.

Fig. 7. Comparative evaluation of THR-NLUnaware, THR-NLAware, MDL-NLUnaware and Gesscale.

immediately compensated by two scale-up decisions. This is a fundamental difficulty faced by all threshold-based algorithms, where choosing the best set of thresholds is extremely difficult.

In short, latency-aware strategies are better during scale-up periods whereas model-based ones are better during scale-down periods. Gesscale combines latency-awareness and being model-based, and is clearly the winner of this comparison.

Table III reports the evaluation metrics for the same set of experiments. Gesscale outperforms the baseline algorithms according to all metrics except one. For instance, it generates 37% fewer reconfigurations than THR-NLAware and 52% fewer than THR-NLUnaware. Its provisioning accuracy is also 38% better in average (for both over-/under-provisioning situations), and consequently its resource usage cost is 16% lower than THR-NLAware. Eventually, although it does not manage to process all incoming data in real-time, its excess time is still 63.5% lower than THR-NLAware.

## VI. Conclusion

Auto-scaling is an important feature for stream processing engines in geo-distributed infrastructures such as fog computing platforms. We presented Gesscale, an auto-scaler designed to allow Apache Flink to maintain a sufficient Maximum Sustainable Throughput while using no more resources than strictly necessary. Gesscale bases its decisions on a performance model which allows it to correctly anticipate the consequences of any potential reconfiguration action. Our evaluations show that Gesscale produces 52% fewer reconfigurations and processes more data than the baseline THR-NLUnaware auto-scaler, while using 17% fewer resources.

TABLE III
EVALUATION METRICS OF THE DIFFERENT AUTO-SCALING ALGORITHMS.

| **Algorithm** | $accuracy_O$ | $accuracy_U$ | $timeshare_O$ | $timeshare_U$ | $excesstime$ | $\#reconf.$ | $cost$ |
|---|---|---|---|---|---|---|---|
| THR-NLUnaware | 1.636 | 0.742 | 51.11 | 30.28 | 0.163 | 25 | 1199.5 |
| THR-NLAware | 1.517 | 0.640 | **46.67** | 22.99 | 0.115 | 19 | 1193.75 |
| MDL-NLUnaware | 1.713 | 0.622 | 60.42 | 25.00 | 0.126 | 16 | 1270.5 |
| **Gesscale** | **0.838** | **0.499** | 49.31 | **21.74** | **0.042** | **12** | **999.5** |

## REFERENCES


[1] R. van der Meulen, "What edge computing means for infrastructure and operations leaders," Smarter with Gartner, 2018, https://gtnr.it/3euQbFh.

[2] K. Yasumoto, H. Yamaguchi, and H. Shigeno, "Survey of real-time processing technologies of IoT data streams," *Journal of Information Processing*, vol. 24, no. 2, 2016.

[3] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Computing Surveys*, vol. 52, no. 2, 2019.

[4] S. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, "The emerging landscape of edge computing," *GetMobile: Mobile Computing and Communications*, vol. 23, no. 4, 2020.

[5] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalalid, A. Niakanlahiji, and J. K. J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, 2019.

[6] V. Cardellini, G. Mencagli, and D. T. M. Torquati, "New landscapes of the data stream processing in the era of fog computing," *Future Generation Computer Systems*, vol. 99, 2019.

[7] T. Hießl, C. Hochreiner, and S. Schulte, "Towards a framework for data stream processing in the fog," *Informatik-Spektrum*, vol. 42, no. 4, 2019.

[8] U. Tadakamalla and D. A. Menascé, "Characterization of IoT workloads," *Lecture Notes in Computer Science*, vol. 11520, 2019.

[9] X. Liu and R. Buyya, "Resource management and scheduling in distributed stream processing systems: A taxonomy, review and future directions," *ACM Computing Surveys*, vol. 53, no. 3, 2018.

[10] S. Vanneste, J. de Hoog, T. Huybrechts, S. Bosmans, R. Eyckerman, M. Sharif, S. Mercelis, and P. Hellinckx, "Distributed uniform streaming framework: An elastic fog computing platform for event stream processing and platform transparency," *Future Internet*, vol. 11, no. 7, 2019.

[11] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth, "An experiment-driven performance model of stream processing operators in fog computing environments," in *Proc. ACM SAC*, 2020.

[12] S. Imai, S. Patterson, and C. A. Varela, "Maximum sustainable throughput prediction for data stream processing over public clouds," in *Proc. IEEE/ACM CCGrid*, 2017.

[13] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking distributed stream processing engines," in *Proc. ICDE*, 2018.

[14] Z. Chu, J. Yu, and A. Hamdull, "Maximum sustainable throughput evaluation using an adaptive method for stream processing platforms," *IEEE Access*, vol. 8, 2020.

[15] P. Carbone, A. Katsifodimos, S. Ewen, and V. Markl, "Apache Flink : Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[16] H. C. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*. Cambridge University Press, 2014.

[17] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@Twitter," in *Proc. SIGMOD*, 2014.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. NSDI*, 2012.

[19] A. Shukla and Y. Simmhan, "Model-driven scheduling for distributed stream processing systems," *Journal of Parallel and Distributed Computing*, vol. 117, 2018.

[20] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys*, vol. 46, no. 4, 2014.

[21] W. Hummer, B. Satzger, and S. Dustdar, "Elastic stream processing in the cloud," *WIREs Data Mining and Knowledge Discovery*, vol. 3, no. 5, 2013.

[22] M. Dias de Assunção, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *Journal of Network and Computer Applications*, vol. 103, 2018.

[23] G. R. Russo, "Self-adaptive data stream processing in geo-distributed computing environments," in *Proc. ACM DEBS*, 2019.

[24] A. Jonathan, A. Chandra, and J. Weissman, "Multi-query optimization in wide-area streaming analytics," in *Proc. ACM SoCC*, 2018.

[25] J. O. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, 2003.

[26] J. Dejun, G. Pierre, and C.-H. Chi, "Autonomous resource provisioning for multi-service Web applications," in *Proc. WWW Conference*, Apr. 2010.

[27] Apache Flink, "FLIP-159: Reactive mode," Feb. 2021, https://t.co/hHVWKcpr6S.

[28] ——, "FLIP-160: Declarative scheduler," Feb. 2021, https://t.co/a0kZKwG63C.

[29] A. Fahs, G. Pierre, and E. Elmroth, "Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler," in *Proc. IEEE MASCOTS*, Nov. 2020.

[30] M. Chima Ogbuachi, A. Reale, P. Suskovics, and B. Kovács, "Context-aware Kubernetes scheduler for edge-native applications on 5G," *Journal of Communications Software and Systems*, vol. 16, no. 1, 2020.

[31] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *Proc. IEEE/IFIP NOMS*, 2018.

[32] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *Proc. ICCCN*, 2021.

[33] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez, "StreamCloud: An elastic and scalable data streaming system," *IEEE TPDS*, vol. 23, no. 12, 2012.

[34] T. Heinze, L. Roediger, A. Meister, Y. Ji, Z. Jerzak, and C. Fetzer, "Online parameter optimization for elastic data stream processing," in *Proc. ACM SoCC*, 2015.

[35] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proc. ACM DEBS*, 2014.

[36] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD*, 2013.

[37] R. Preet Singh, B. Kumarasubramanian, P. Maheshwari, and S. Shetty, "Auto-sizing for stream processing applications at LinkedIn," in *Proc. HotCloud20*. USENIX Association, Jul. 2020.

[38] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang, "Elasticutor: Rapid elasticity for realtime stateful stream processing," in *Proc. SIGMOD '19*, 2019.

[39] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proc. OSDI*. USENIX Association, 2018.

[40] T. Li, Z. Xu, J. Tang, and Y. Wang, "Model-free control for distributed stream data processing using deep reinforcement learning," in *Proc. VLDB Endow.*, 2018.

[41] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Generation Computer Systems*, vol. 87, 2018.

[42] A. Da Silva Veith, m. D. de Assunção, and L. Lefèvre, "Monte-carlo tree search and reinforcement learning for reconfiguring data stream processing on edge computing," in *Proc. SBAC-PAD*, 2019.

[43] T. De Matteis and G. Mencagli, "Elastic scaling for distributed latency-sensitive data stream operators," *Proc. PDP*, 2017.
[44] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 29, no. 3, 2018.
[45] V. Marangozova-Martin *et al.*, "Multi-level elasticity for data stream processing," *IEEE Trans. on Parallel and Distributed Systems*, vol. 30, no. 10, 2019.
[46] X. Ni, S. Schneider, R. Pavuluri, J. Kaus, and K.-L. Wu, "Automating multi-level performance elastic components for IBM streams," in *Proc. ACM/IFIP Middleware*, 2019.
[47] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *J. of Systems and Software*, vol. 127, no. C, 2017.
[48] G. Mencagli, M. Torquati, and M. Danelutto, "Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams," *Future Generation Computer Systems*, vol. 79, 2018.
[49] H. Arkian, D. Giouroukis, P. Souza Jr, and G. Pierre, "Potable water management with integrated fog computing and LoRaWAN technologies," *IEEE IoT Newsletter*, 2020.
[50] A. van Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, "MEC-ConPaaS: An experimental single-board based mobile edge cloud," in *Proc. IEEE Mobile Cloud*, Apr. 2017.
[51] B. Donovan and D. Work, "New York City Taxi Trip Data (2010-2013)," 2016.
[52] N. Herbst, R. Krebs, G. Oikonomou, G. Kousiouris, A. Evangelinou, A. Iosup, and S. Kounev, "Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics," 2016. [Online]. Available: https://arxiv.org/abs/1604.03470