Evaluating Stream Processing Autoscalers

George Siachamis George Christodoulou Kyriakos Psarakis Marios Fragkoulis
Arie van Deursen Asterios Katsifodimos
Delft University of Technology
{initial.surname}@tudelft.nl

ABSTRACT

While the concept of large-scale stream processing is very popular nowadays, efficient dynamic allocation of resources is still an open issue in the area. The database research community has yet to evaluate different autoscaling techniques for stream processing engines under a robust benchmarking setting and evaluation framework. As a result, no conclusions can be made about the current solutions and problems that remain unsolved. Therefore, we address this issue with a principled evaluation approach.

This paper evaluates the state-of-the-art control-based solutions in the autoscaling area with diverse, dynamic workloads, applying specific metrics. We investigate different aspects of the autoscaling problem as performance and convergence. Our experiments reveal that current control-based autoscaling techniques fail to account for generated lag cost by rescaling or underprovisioning and cannot efficiently handle practical scenarios of intensely dynamic workloads. Unexpectedly, we discovered that an autoscaling method not tailored for streaming can outperform others in certain scenarios.

CCS CONCEPTS

• **Information systems** → **Stream management**.

KEYWORDS

Data streams, Autoscaling, Reconfiguration, Elastic provisioning

ACM Reference Format:
George Siachamis, George Christodoulou, Kyriakos Psarakis, Marios Fragkoulis, Arie van Deursen, Asterios Katsifodimos. 2024. Evaluating Stream Processing Autoscalers . In *The 18th ACM International Conference on Distributed and Event-based Systems (DEBS '24), June 24–28, 2024, Villeurbanne, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3629104.3666036

1 INTRODUCTION

A plethora of applications utilize services provided by cloud computing vendors with a variable demand for resources over time. Thus, the vendors have to prepare their platforms for a highly dynamic allocation of resources, depending on the configurations set by users. Furthermore, cloud platforms offer pay-per-use pricing models so that applications only pay for the resources they actually consume. To tackle this multi-conditional problem, the resources should be able to upscale and downscale elastically, adapting to the dynamic demand of the applications.

Most of the widely adopted stream processing engines (SPEs) were originally developed for deployment on clusters of fixed resources. These SPEs provide limited autoscaling capabilities and require substantial operational effort to adapt to changes in needs and workloads. An operations team has to always monitor the performance of the deployed system or application, estimate the required resources, decide whether to scale, and perform a manual rescaling. This process is time-consuming and offers slow reactions to workload changes with serious performance implications.

To provide automated solutions, specialized autoscalers have been developed to equip SPEs with the missing self-managing capabilities. However, it remains unclear how these autoscalers perform in different practical scenarios due to the absence of a proper comparison framework. We argue that without a principled and configurable experimental analysis, it is doubtful that these autoscalers will have the desired impact on modern stream processing engines.

Identifying the difference in resource demand is a critical point of this problem. The shift in demand can be identified by monitoring the underlying infrastructure. Autoscaling methods can vary in terms of problem modeling, heuristics, parameters, provisioning metrics, granularity, and performance [27, 30]. Furthermore, the more fine-grained the rescaling actions can be, based on the operators employed in the pipeline, the better an autoscaler will adjust the resources to the workload patterns. The most prevalent categories of autoscalers include reactive, such as *threshold-based* [14, 18], *reinforcement learning* [8, 25], *queue-based* [10, 24], *control-based* [9, 19] and proactive solutions, like *time series forecasting* [4, 26].

Contributions. In this work, we thoroughly investigate the existing control-based autoscaling solutions for SPEs and provide a concrete set of metrics, queries, and workloads to evaluate them principally. We focus on control-based solutions due to their versatility, their simplicity and the lack of training requirements. In short, the contributions presented in this paper are the following:

- We stress the importance of extensive experimental evaluation of autoscalers for stream processing.
- We reproduce state-of-the-art autoscalers for stream processing under a common framework.
- We extend the autoscaling solutions operating on the deployment level to rescale on an operator level to ensure that the resource allocation will harmonize with the demand.
- We extend the experimental evaluation of the state-of-the-art control-based autoscaling solutions with heavily dynamic workloads, and we establish important metrics for evaluating autoscaling. We present our experimental results over diverse queries.
- We reach a series of interesting conclusions which, in our opinion, will spark additional research in the area:

- The design choices of each operator heavily influence their performance and their ability to adhere to different objectives.
- General-purpose autoscalers can perform better for stateless queries than the evaluated solutions specifically tailored to stream processing.
- The evaluated autoscalers struggle with complex stateful queries under dynamic workloads.
- The stop-and-restart state migration process of current SPEs hinders the performance of autoscalers that do not account for the lag generated during the rescaling action.

All the resources are publicly available:
https://github.com/delftdata/espa-autoscaling.git

Outline. In Section 2 we present preliminaries and necessary notation. Section 3 reviews autoscaling techniques and benchmarks relevant to our work. In Section 4, we present in detail autoscaling solutions, which we evaluate in this work. In Section 5, we describe the metrics, workloads, and queries used for this evaluation. Section 6 includes our experimental evaluation. In Sections 6.5 and 7, we discuss our key findings and limitations, highlight open challenges, and share the lessons we learned, concluding the paper.

2 BACKGROUND

In this section, we dive into the autoscaling process and discuss the necessary concepts to discuss the selected autoscalers.

2.1 Autoscaling Process

The process of autoscaling resembles the MAPE loop from control theory. As depicted in Figure 1, the first step includes *monitoring* of a stream processing job and acquiring all the metrics needed both for the evaluation of its performance and for the decision of performing rescaling actions. Then, the *analysis* step takes place, where we evaluate the job's current state and calculate the job's needs to adhere to the enforced agreements. The analysis outcome is then used from the *planning* step to decide on the proper rescaling actions. The goal is to satisfy the calculated needs while minimizing the resources employed. The last step is *executing* the devised plan. The monitoring API of the SPE or any applicable monitoring tool is usually responsible for retrieving the metrics, while execution usually falls on the SPE and its rescaling mechanism. The analysis and planning steps are handled from the *autoscaler*.

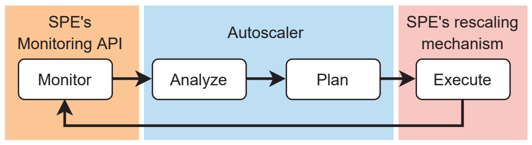


Figure 1: MAPE loop for stream processing autoscaling

2.2 Common notions

Workers & Operators. In this paper, we use Apache Flink as our SPE. We choose Flink among other SPEs since it is the current state-of-the-art and the most widely adopted system in production, while providing all the mechanisms expected by the autoscalers. Apache Flink refers to workers as task managers. By default, a task manager runs multiple operators that share its resources. However, we have configured Flink to isolate operators and assign a single operator to each task manager.

Back pressure. Back pressure is a rate control mechanism employed by many SPEs. When an operator cannot handle the input rate, the system uses the back pressure mechanism to regulate the output rate of the upstream operator. The backpressure can be propagated up to the source operator and the input queue.

Lag. Lag is defined as the number of unprocessed records waiting in the input queue or the operator buffers.

Elasticity. Elasticity in cloud computing is the system's ability to dynamically adjust the resource allocation to evolving workloads transparently. The system's cost is optimized by aligning resource allocation with actual demand.

3 RELATED WORK

In this section, we discuss the related work on autoscalers specifically designed for stream processing and the available stream processing benchmarks.

Threshold-based. In [14], threshold-based rules are shown to boost performance when applied on individual hosts but not on the entire system. The distributed stream processing engine in [18] supports system scaling at run-time. The proposed autoscaler uses threshold-based rules to test the scaling capabilities of the system.

Reinforcement Learning. Heinze et al. [14] also propose a reinforcement learning approach that can result in high performance while minimizing the initial configuration costs. This problem is further addressed in [15], where an online parameter optimization technique is proposed, which detects changes in the workload pattern and adapts the scaling policy accordingly. Lombardi et al. [25] propose ELYSIUM, an autoscaler that optimizes resource consumption considering the trade-off between horizontal and vertical scaling. In another work, Lombardi et al. [26] propose PASCAL, a general-purpose autoscaler based on reinforcement learning. A proactive approach forecasts incoming workloads, while a profiling system estimates the optimal provisioning. Doan et al. [8] propose a fuzzy deep reinforcement learning method for autoscaling streaming architectures. Although effective, the parameter tuning of the method is a non-trivial task. Cardellini et al. [6] propose an autoscaler for stream processing in a decentralized environment. It consists of two reinforcement-based learning approaches on a two-layered hierarchical structure for handling each operator in the system individually.

Queue-based. Lohrmann et al. [24] propose a generalized Jackson network, allowing for more precise performance estimations. The scaling decision is determined by comparing various resource allocations. Similarly, Fu et al. [10],[11] propose DRS to capture the impact of provisioned resources using a queuing-theory-based autoscaler. [4] propose an autoscaling method for distributed stream processing in geo-distributed environments. A performance model decides which geo-distributed servers need additional resources to optimize the maximal sustainable throughput of the system.

Control-based. Gedik et al. [13] proposed one of the first autoscalers specifically designed for distributed stream processing engines (SPEs) with stateful operations support. Floratou et al. [9]

propose a framework to create self-regulating streaming systems using scaling policies based on the back-pressure status of the system. A self-adaptive processing graph was introduced in [17], which divides the workload of overloaded operators over multiple replicas. Using a control algorithm, the topology can be reactively and proactively scaled, improving the performance and resource efficiency of the system. Kalavri et al. [19] propose DS2, a control-based autoscaler for distributed stream processing. The authors introduce true processing and output rate notions and estimate the optimal parallelism for every operator in a single iteration. Mencagli et al. [29] propose a two-fold autoscaling method based on adaptive scheduling techniques for the short-term spikiness, while a fuzzy logic controller handles the long-term rate variability. Liu et al. [23] develop a profiling model to capture the impact of provisioned resources on the performance and scale the application accordingly. Varga et al. [33] propose two custom metrics combined with Kubernetes' out-of-the-box autoscaler HPA [2] for scaling SPEs.

Benchmarking. NEXMark [31], later extended by Beam [1], is a streaming benchmark that includes a set of analytical queries on streaming data from an online auction platform. Linear Road benchmark [3] simulates a toll system for a fictional urban area. The system monitors traffic and supports operations on live and historical data. Another benchmark on traffic sensor analytics is OSPbench [32]. SmartBench [28] focuses on querying IoT data derived from a smart building monitoring system. The benchmark performs a diverse set of temporal and spatial queries. SparkBench [22], a benchmark focused on Apache Spark, emphasizes on popular Spark applications, including machine learning, graph computation, SQL query and streaming. Analogousely, ESPBench [16], use multiple types of data to test workloads of varying complexity (e.g. filtering, machine learning). DSPBench [5] covers multiple streaming scenarios with 15 different benchmark workloads. Yahoo Streaming Benchmark (YSB) [7] uses an advertisement campaign simulation focusing on relational algebra operations, including filtering, projections and joins. StreamBench [28] generated streams from real-time web log processing and network traffic monitoring seeds. The operational workload varies in complexity and scenarios (e.g. performance, fault-tolerance).

4 CONTROL-BASED AUTOSCALERS

In this section, we delve into the core concepts of the selected autoscalers and how we extended some of them. For our evaluation, we select the state-of-the-art DS2[19] and Dhalion[9]; these solutions are easily deployed and widely accepted by the community. We also consider Horizontal Pod Autoscaler [2], a solution applied to a commercial product. Finally, we employ the metrics suggested by Varga et al. [33] to extend HPA towards a solution more tailored to stream processing.

4.1 Dhalion

Dhalion [9] is a framework that provides self-regulating capabilities to underlying stream processing systems that employ a backpressure mechanism to perform rate control. It utilizes user-defined policies to handle performance issues related to different underlying causes, such as load skew, slow instances, and provisioning. In this work, we are only interested in its proposed policy for autoscaling. The policy distinguishes two cases: an overprovisioning and an underprovisioning case.

Overprovisioning. For an operator of a running job to be considered overprovisioned, two conditions must hold: (a) there is no backpressure anywhere in the pipeline, and (b) the input queue of the operator has a length of almost zero. For each operator considered overprovisioned, new parallelism is calculated using a provided *scale down factor*.

Underprovisioning. If there is any backpressure along the pipeline, the job is considered to be in an unhealthy state and underprovisioned. To resolve the issue, the first step is to identify the operator which is the root of the backpressure. Then, a *scale up factor* is calculated for this operator based on the amount of time the job used to process the input normally and the amount of time backpressure occurred over the monitoring window. The current monitoring window is denoted as w_i. More precisely, the *scale up factor* is provided by the following formula:

$$scaleUpFactor = \frac{backpressuredTime_{w_i}}{normalProcessingTime_{w_i}} \tag{1}$$

As we consider Kafka as our source, we need to scale up/down Flink's KafkaSource operators. Since there is no backpressure information available for these operators, we decided to use the increase of lag noticed in Kafka as an indicator of backpressure caused by the KafkaSource operators. We denote as *pendingRecordsRate*, the average lag increase per second, and the average number of records consumed per second as *consumedRecordsRate*. Finally, the *scale up factor* is calculated as:

$$scaleUpFactor_{KS} = \frac{pendingRecordsRate_{w_i}}{consumedRecordsRate_{w_i}} \tag{2}$$

We gather the needed metrics using the monitoring API of Flink and Prometheus. Since Flink does not report the input queue size of each individual operator, we use the percentage of input buffers used to decide on the lag in the input queues.

4.2 DS2

In contrast to Dhalion, which scales each operator independently, DS2 [19] attempts to combine the scaling of all operators in a single step by leveraging the topology of the streaming query. To do so, it introduces the notions of *useful time*, *true processing rate*, and *true output rate*. *Useful Time* is the time spent by an operator in (de)serializing and processing records. *True processing rate* is the number of records an operator processes per unit of useful time, while *true output rate* is the number of records an operator outputs per unit of *useful time*. Based on these notions, DS2 calculates progressively the optimal parallelism of each operator o_i as follows:

$$OP_{o_i} = \frac{\sum \text{true output rate of upstream operators}}{avg(\text{true processing rate})\text{ of } o_i} \tag{3}$$

In this work, to calculate the optimal parallelism for the KafkaSource operators, we use the rate at which records are written to Kafka as the *true output rate* of the upstream operators. In addition, we extend DS2 with a user-provided overprovisioning factor to help DS2 to handle noisy spikes and the lag accumulated due to scaling actions. This is the only tunable parameter of DS2.

4.3 HPA

The Horizontal Pod Autoscaler (HPA) [2] is the default autoscaling solution shipped with Kubernetes. HPA scales horizontally a deployment by adding or removing pods in order to match user-provided target values based on an observed metric. The observed metric can be either the standard average CPU/memory utilization or any custom user-defined metric, applied as shown in Equation (4).

$$desiredPods = \lceil currentPods \times \frac{currentMetricValue}{targetMetricValue} \rceil \tag{4}$$

When scaling down, HPA opts for a conservative approach. It records the scaling recommendations over a stabilization window and picks the highest recommendation as the desired amount of resources. Thus, it ensures a gradual scaledown that is not affected by fluctuations in the metric values.

HPA as a Streaming Topology Autoscaler. Since a given worker in the streaming topology runs on an individual pod, HPA can be used as a basis for building a streaming topology autoscaler that will add or remove workers when required. Although, HPA works over the workers' deployment of Flink and is agnostic of the underlying operators. Our version of HPA monitors the actual operators within a pod instead of the deployment of the workers. We employ the average CPU utilization as a metric. From now on, we will refer to this custom version of HPA as HPA-CPU. HPA-CPU has two tunable parameters: the CPU utilization target value and the length of the stabilization window.

4.4 HPA-Varga

Varga et al. [34] extend the HPA autoscaler to use metrics tailored for stream processing; relative lag change and utilization can be used in an ad-hoc fashion for HPA.

Utilization. Utilization provides additional system performance insights. It separates over-provisioning from optimal provisioning by analyzing the percentage of the available resources currently employed for stream processing tasks. To do so, utilization employs the idle-time-per-second metric that most modern stream processing engines provide out of the box. The utilization of the system is calculated using the following formula:

$$Utilization = 1 - avg(\text{idle time per second}) \tag{5}$$

Utilization can take values between 0 and 1. A value close to 1 means that the system is using the provided resources to their limits. Otherwise, the resources are underutilized. If the targeted value is close to 1, the autoscaler suggests intensive resource utilization, resulting to fewer scale-up and more aggressive scale-down actions. Although such a strategy might lower resource costs, it might also result in underprovisioning. When lowering the target value, the autoscaler issues scale-up actions more frequently. Such a setting has a higher chance of leading to overprovisioning.

Relative lag change rate. To mitigate the effects of a possible utilization's misconfiguration, Varga et al. [33] pair utilization with another metric. Relative lag change estimates the portion of the workload the system cannot handle. It uses the derivative of the system's lag and the application's input throughput recorded at the input queue. The following formula calculates the relative lag change rate:

$$RelativeLagChangeRate = 1 + \frac{deriv(\text{total lag})}{\text{input throughput}} \tag{6}$$

The relative-lag-change rate denotes the rate at which the lag in the input queue is increasing (> 1) or decreasing (< 1). When equal to 1, the lag is not changing, and the current resources are sufficient to handle the workload. Therefore, the relative lag change rate's target value is usually 1.0. When HPA is provided with two monitored metrics, it decides on a scaling actions based on the metrics resulting to the highest parallelism. To allow the autoscaler to scale down in cases of overprovisioning, the authors propose ignoring the relative lag change rate when the lag is below a user-provided threshold. As a result, the HPA will only consider utilization when the lag is below the threshold, allowing for scale-down actions when overprovisioning

As discussed, in Section 4.3, the original HPA autoscaler targets deployment-level autoscaling, which is insufficient for a stream processing engine. The extensions suggested by Varga et al. [33] operate similarly. To achieve operator-level autoscaling, we use the same procedure with HPA-CPU, monitoring the operators. While measuring the utilization is straightforward, measuring the relative lag change rate per operator is not trivial. We measure it at the input queue and propagate the result to the operator responsible based on the utilization metric and the backpressure mechanism.

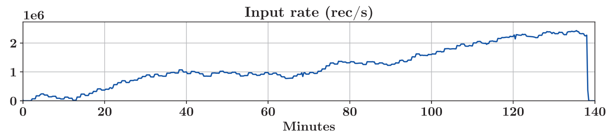
5 EVALUATION COMPONENTS

We now focus on establishing a principled evaluation framework for stream processing autoscaling. First, we establish the metrics that can provide feedback on the effectiveness of the autoscaling solutions. Then, we discuss the most interesting NEXMark queries, and finally, we propose a set of dynamic workloads that enable a meaningful evaluation of the autoscalers. We conclude this section with a discussion about the evaluation contributions in contrast to the original papers of the methods we cover.

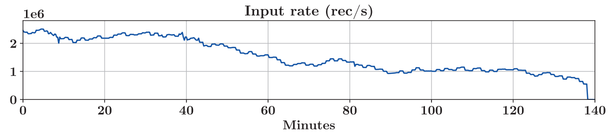
5.1 Performance Evaluation Metrics

Latency. Stream processing usually targets processing data and acquiring results in real-time. Therefore, the most important metric characterizing the performance of an SPE is latency [20]. Since the goal of every autoscaler is to provision just the optimal resources for an SPE to perform efficiently, the SPE's latency can also be used to evaluate the performance of an autoscaler [12, 23]. Typically, the latency is measured as the time it takes for a record to be processed and produce results from the moment it becomes available in the input queue [20]. However, this definition of latency is difficult to measure and depends significantly on the underlying query. Instead, we measure latency as the time a record stays in the input queue until the SPE processes it. We focus on the 50th and the 95th percentile of this latency.

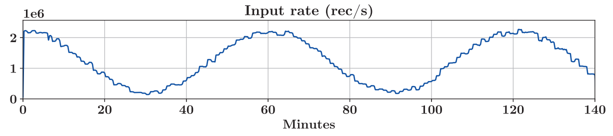
Throughput. Throughput is also one of the primary and most important metrics used to evaluate the performance of an SPE under the current deployment condition and, therefore, the performance



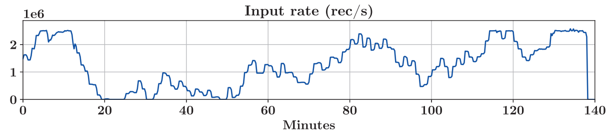
(a) Increasing



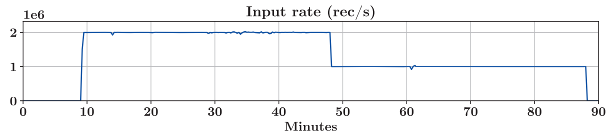
(b) Decreasing



(c) Cosine



(d) Random



(e) Steps

Figure 2: Workloads

of an autoscaler. We define throughput as the number of records the SPE ingests and processes over a second. By comparing the SPE's throughput with the input rate of our source, we can see whether the system can keep up with the input rate and evaluate whether an autoscaler has provided enough resources to the SPE.

Resource efficiency. To estimate the autoscalers' efficiency regarding resource consumption, we consider the number of workers deployed at any time during execution. This is equal to the sum of the operators' parallelism.

Number of scaling actions. We also consider the total number of scaling actions. Depending on the system's topology, scaling the application can induce a significant overhead. This is especially the case with stateful operators and stop and restart migration mechanisms such as the one of Apache Flink, where the state needs to be persistently stored, then migrated offline and reloaded according to the new topology when the system restarts.

Convergence time & steps. Finally, we also investigate the convergence time of the autoscaler, i.e., the total amount of time it takes for the autoscaler to converge to a specific deployment for a new throughput. We also measure the total number of scaling actions required before converging to a new configuration.

5.2 Queries

For the evaluation of the autoscalers, we employ queries from the original NEXMark benchmark [31] and the extended version provided by the Apache Beam project [1]. NEXMark simulates an e-commerce application, mainly featuring three types of records: people, auctions, and bids. NEXMark provides a set of different streaming queries with different properties and complexities. From these queries, we select the following subset that covers the most common types of streaming queries.

Map (*Map*). We first evaluate the autoscalers using an implementation of *Q1* of the original NEXMark benchmark [31]. We will refer to *Q1* as the *Map* query. The *Map* query transforms the values of auctions' bids between different currencies, for example, converting U.S. dollars to Euros. Thus, it performs a map over the stream of data. We choose *Q1* as a representative stateless query with a low-complexity topology and a low computational load.

Filter (*FQ*). *FQ* filters out bids that do not belong to a number of selected auctions. Similarly to *Q1*, *Q2* is also a stateless query with low computational complexity that employs a flatmap as a filter.

Incremental Join (*IJ*). *IJ* focuses on profiling user lifecycles within an online marketplace or an auction platform. It creates insights about user behavior and engagement inside the platforms. The query performs operations involving filtering and grouping. It is a stateful query with a constantly growing state and a non-linear to the input computational cost and, therefore, a complex and heavy workload-wise query. *IJ* has the highest number of operators among the selected NEXMark queries. We apply a time-to-live setting for the state's records, as a continuously expanding state is an unrealistic streaming scenario and will eventually strain our limited resources.

Sliding Window Aggregate (*SA*). *SA* targets items in the auction platform based on their popularity over a certain time period measured by the volume of bids. *SA* operates a sliding window aggregation query, i.e., it computes the total number of bids received for each auction over a sliding time window. The query needs filtering and grouping steps before performing the aggregation step. *SA* is also a computationally expensive query with a large state; however, in contrast to *IJ*, the computational cost is linear to the input rate.

Session Window Aggregate Query (*SWA*). *SWA* focuses on calculating the number of bids a user makes in each active session. To do so, it performs a windowed aggregate (count) over a session window. Therefore, it also comprises a stateful complex computation task, like the other windowed queries.

5.3 Workloads

We use a scalable generator that utilizes the NEXMark's entity generators to create dynamic workloads following specified patterns. We employ five different workload patterns.

Increasing. The increasing workload pattern (fig. 2a) starts from zero input rate and constantly increases over time. The underlying system starts from the minimum parallelism and must scale up following the increase of the input rate. The increasing workload allows for focusing only on scaling up actions and investigating how each autoscaler handles them.

Decreasing. The decreasing workload (fig. 2b) provides a symmetrically different scenario than the increasing workload. It starts at a maximum input rate and then constantly decreases towards zero input rate. The system starts from an appropriate for the high throughput configuration and scales down following the decrease in the input rate. Contrary to the increasing workload, the decreasing workload allows us to focus only on scaling down actions.

Cosine. The cosine workload (fig. 2c) combines the increasing and decreasing workloads following a cosine pattern. The cosine workload allows for evaluating both the scaling up and scaling down capabilities of an autoscaler, a more complex scaling scenario. It imitates a significant subset of real-world scenarios of dynamic workloads that show some periodical or seasonal behavior.

Random. Another workload that mimics a real-world scenario is the random workload (fig. 2d). The random workload starts at a specific input rate, which is randomly increased or decreased over time. Not following a predefined pattern makes it more difficult for the autoscalers to anticipate changes in the input rate, potentially uncovering unwanted behavior programmed into the autoscalers.

Steps. Finally, the steps workload (fig. 2e) simulates a workload that consists of fixed pulses of input rates. This workload allows for investigating the performance of autoscalers when the objective is to handle specific changes in the input rate efficiently. It resembles a real-world scenario of changes in the throughput SLAs between a provider and a client. It also enables us to investigate the time autoscalers take to converge to the optimal parallelism configuration after a change in the targeted input throughput.

5.4 Discussion

To the best of our knowledge, this work is the first to compare multiple autoscalers under a common framework and establish specific workloads and metrics for this evaluation. The original works presenting the evaluated autoscaling methods are limited, using metrics tailored to a specific goal, and do not include extensive comparisons with competitors under a variety of scenarios. Dhalion's evaluation includes a single wordcount query for throughput performance measurement, a convergence experiment measuring the provisioned resources, and an experiment with two input rate changes while omitting experiments with competitors. Varga HPA focuses on snapshot capturing duration and operator loading during rescaling without including experiments with competitors. DS2 performs the most comprehensive evaluation across three systems, employing six NEXMark queries and a wordcount query. However, it only compares against Dhalion, while the evaluation includes experiments with only two input rate changes and convergence experiments. DS2's primary focus lies in outlining the convergence steps required to reach a requested throughput. Finally, HPA has not been evaluated in a stream processing setting.

In this work, we establish the metrics that are relevant to autoscaling and should always be used for evaluating autoscaling solutions. Additionally, we propose four heavily dynamic workloads that constantly change their input following specific patterns. In this specific evaluation, we focus on latency performance, which none of the evaluated solutions have previously considered. Finally, we stress-test all methods in real-world conditions where data sources continuously produce items during rescaling, leading to potential lag. Notably, none of the evaluated solutions has been previously tested in this setting.

6 EXPERIMENTAL EVALUATION

We now present our detailed experimental analysis. First, we evaluate the performance of the autoscalers with queries from the NEXMark benchmark on different workloads. Then, we demonstrate the performance of the autoscaling solutions on additional queries. Next, we compare the convergence ability of each autoscaler. Finally, we discuss the results of our experimental evaluation.

6.1 Experimental Setup

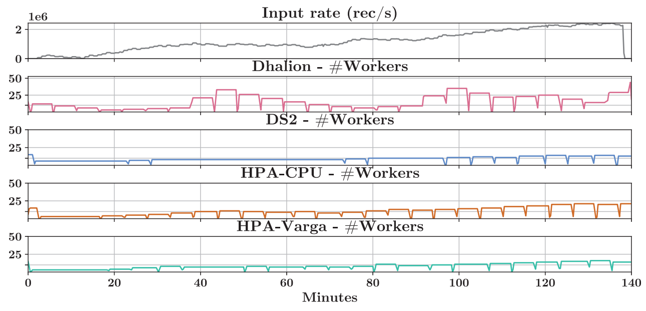
The experiments are conducted on a 3-node Kubernetes cluster with AMD EPYC 7H12 2.60GHz CPUs. On top of this Kubernetes cluster, we have configured an Apache Flink cluster in application mode. The JobManager (Flink's coordinator) instance is provided with 1 CPU and 8GB of memory, while each employed TaskManager (Flink's worker) consists of 1CPU and 4GB of memory. An NFS server is deployed as a persistence layer, Prometheus[1] is used for scraping and gathering all the metrics, and an Apache Kafka[2] deployment is used as a source for the experiments. We cap the available resources to 70 task managers, resulting in a maximum of 70 CPUs and 280GB of memory available for processing.

In our experiments, we set Dhalion's scale-down factor to 0.2, a value suggested in the original work. We use an overprovisioning factor of 0.2 for DS2, which we consider to be sufficient as the intention of DS2 is to avoid any overshooting of resources. We use the default stabilization window of 5 minutes for HPA-CPU, and we choose a target CPU utilization of 70% as the best performing among the values tested. For HPA-Varga, we also consider a CPU utilization target of 70%, the same with HPA-CPU. In addition, we employ a cooldown window of 5 minutes after every scaling action to allow time for the system to reach a stable state and avoid back-to-back scaling actions due to a slow restart of the system or the lag produced by the scaling action.

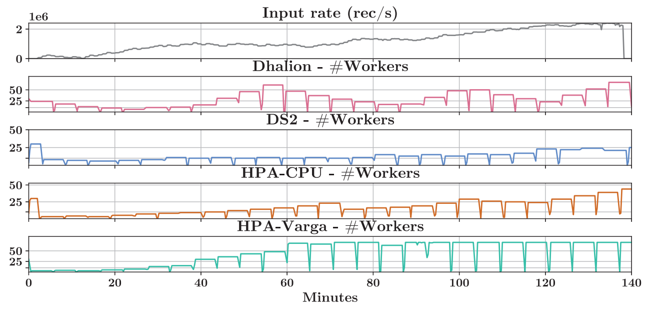
6.2 Workload Comparison

Our first set of experiments involves four workloads from Section 5.3: increasing, decreasing, cosine, and random pattern workloads. The selected workloads represent heavily dynamic workload patterns whose input rate changes constantly. Due to space constraints, we limit our evaluation of the autoscalers across different workloads to two queries, the *Map* and the *Session Window Aggregate*. We choose the *Map* query as a reference query because of its simplicity and lack of state, allowing us to investigate the performance of the autoscalers on a query with low computational complexity. For such a query, performance is mostly influenced by the ability of the system to ingest and circulate the input to its operators rather than the actual computation. Alternatively, the *Session Window Aggregate* query represents a common operation in real-time analytics. Calculating time intervals for session windows makes the *Session Window Aggregate* one of the most computationally complex queries available. In contrast to the *Map* query,

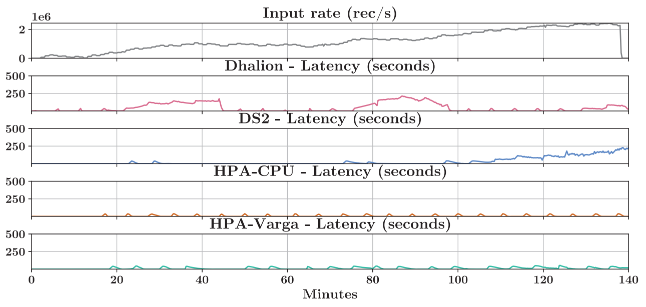
[1]https://prometheus.io/
[2]https://kafka.apache.org/



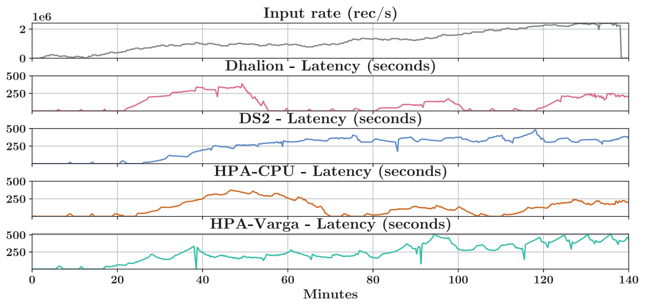
(a) Workers deployed for *Map*.



(b) Workers deployed for *SWA*.



(c) Latency of *Map*.



(d) Latency of the *SWA*.

Figure 3: Increasing pattern

the *Session Window Aggregate* query employs a heavy computational task that dominates the impact on the performance of the underlying system and, thus, the employed autoscaler.

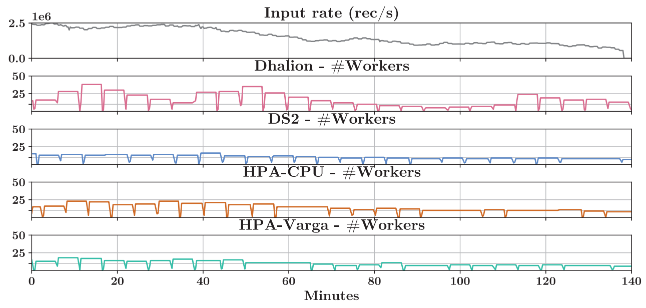
Increasing pattern. As discussed in Section 5.3, the increasing workload pattern allows us to study the scaling-up behavior of the autoscalers in isolation. Figure 3 showcases the performance of the autoscalers on this workload in terms of the number of deployed workers and the resulting latency throughout the execution of the two deployed queries.

– *Performance on Map.* Figure 3a present the number of deployed workers per minute during the *Map* query execution. We observe that the number of workers Dhalion recommends does not follow the input pattern. Although the input rate constantly increases, Dhalion has large intervals of decreasing resources. At first, Dhalion reacts slowly to the increasing rate; the backpressure keeps increasing, translating to increasing latency (Figure 3c), which triggers an aggressive scale-up. Evidently, Dhalion fails to suggest the right resources leading to overprovisioning and a large scale-down interval that does not match the expected behavior. DS2 avoids unnecessary rescaling actions while providing the minimum required resources for low latency when the input remains relatively low. When the input increases significantly, every scaling action generates progressively more lag, and DS2 rescales more frequently. This causes a constant increase in latency due to DS2 failing to accommodate the generated lag. The HPA-based solutions assign workers to follow the input pattern smoothly. Both autoscalers keep latency low and perform more frequent rescaling actions than DS2. However, HPA-Varga constantly suggests a lower number of workers without compromising performance. Overall, HPA-based autoscalers match the input pattern and perform better in latency. However, they assign a slightly higher number of resources than DS2.

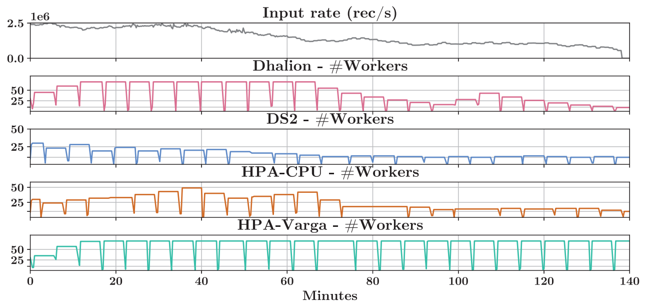
– *Performance on* SWA. We observe in Figure 3b that the methods follow a similar trend. Dhalion shows the same behavior as in the *Map* query; it fails to react on time to the input increase, then aggressively overprovisions resources, leading to large scale-down intervals. Contrary to the *Map* query, DS2 performs frequent rescaling actions even while the input remains low. The latency (fig. 3d) starts to increase early on, and DS2 never manages to recover. However, DS2 assigns the lowest number of workers throughout the experiment. HPA-Varga follows the input pattern but aggressively issues scale-up actions, reaching the maximum available resources early. Despite the many resources, it fails to ensure low latency. HPA-CPU matches the input pattern, steadily providing more resources. However, it still fails to retain low latency.

Decreasing pattern. In contrast to the increasing pattern, the decreasing workload employs a constantly decreasing input rate. This experiment demonstrates the scale-down performance of the autoscalers in isolation. Figure 4 illustrates the behavior of the autoscalers in terms of workers deployed and latency.

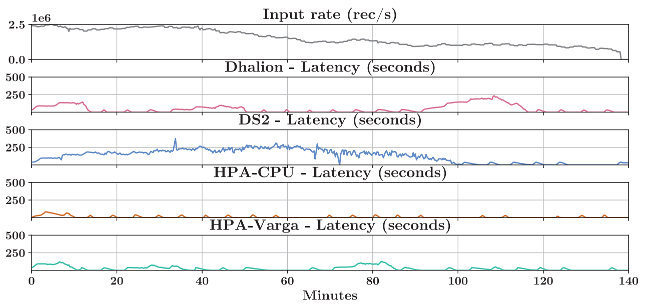
– *Performance on Map.* Similar to the increasing workload, Dhalion does not react to input on time, resulting in fluctuating behavior. DS2 again keeps the resources low but fails to reduce the latency before the input decreases sufficiently. HPA-CPU follows the decreasing pattern and retains low latency throughout the execution while issuing slightly fewer rescaling actions than the rest. HPA-Varga also keeps latency relatively low while recommending fewer resources than HPA-CPU.



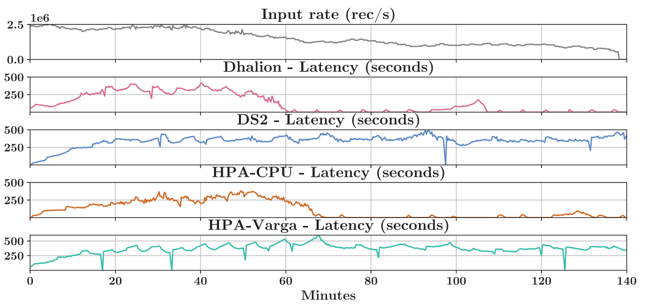
(a) Workers deployed for *Map*.



(b) Workers deployed for *SWA*.



(c) Latency of *Map*.



(d) Latency of *SWA*.

Figure 4: Decreasing pattern

– *Performance on* SWA. The *SWA* query presents a challenge to all methods in the decreasing pattern. Dhalion immediately considers the initial parallelism inadequate and quickly scales the system to the maximum available resources. Despite utilizing all the available workers, Dhalion fails to handle the high latency early on. It only manages to reduce the latency after the input rate has been decreased significantly, although it issues an unexpected upscale. HPA-CPU achieves the same performance in terms of latency. However, it assigns significantly fewer resources throughout the execution. DS2 and HPA-Varga cannot retain low latency throughout the entire run. DS2's provisioning adheres to the input pattern, while HPA-Varga quickly maximizes the resources, remaining at the highest parallelism for the entire run.

Cosine pattern. The cosine pattern combines increasing and decreasing input behavior and composes a representative real-world workload that requires varied scaling actions. It is an interesting experiment that allows us to evaluate the autoscalers on an explainable, highly dynamic workload. We illustrate the performance of the autoscalers in Figure 5.

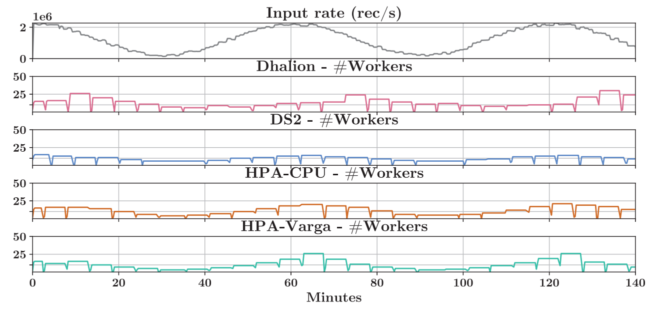
– *Performance on* Map. Contrary to its behavior on the increasing and decreasing patterns, Dhalion's resource allocation follows the input with a small delay. This delay causes a latency increase during periods of high input rate. Dhalion manages to recover during periods of lower input rate. Similarly, DS2 suffers from a small latency increase only during periods of high throughput. DS2 keeps providing fewer resources throughout the experiment while following the input pattern. This is a consequence of failing to handle the generated lag arising from rescaling. The same latency behavior is observed for HPA-Varga, resulting from a slow scale-up of workers. In contrast, HPA-CPU maintains low latency while adjusting resources on time to keep up with the input.

– *Performance on* SWA. Similar to *Map* query, Dhalion does not react in time to the input changes, resulting in a latency increase. DS2 follows the input accurately. However, it suffers from high latency since it doesn't recover even during low input periods. Both HPA-based autoscalers have high latency for high input periods as they react late to the input changes. However, HPA-CPU achieves the same performance with fewer deployed workers.

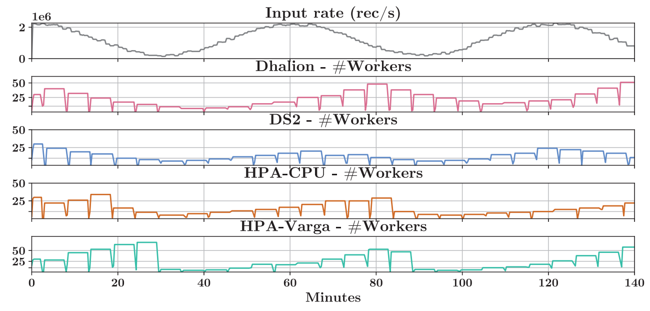
Random pattern. The random workload pattern is the most complex and another representative real-world challenging pattern. It resembles real-world traffic with sudden spikes and irregular input changes, making the scaling actions varying and less obvious.

– *Performance on Map.* Unsurprisingly, Dhalion fails to match resources with the input pattern. It reacts slowly to the random pattern's sudden input changes, leading to large periods of high latency. Despite the randomness of the pattern, DS2 manages to assign resources according to the input pattern. However, in terms of latency, it fails to adapt during prolonged periods of high input rates. HPA-Varga and HPA-CPU adapt to the input pattern for the majority of the time. However, HPA-Varga suffers from sudden increases in input rate leading to temporary higher latency. HPA-CPU maintains low latency for most of the experiment, except for a high latency period at a sudden increase in input rate.

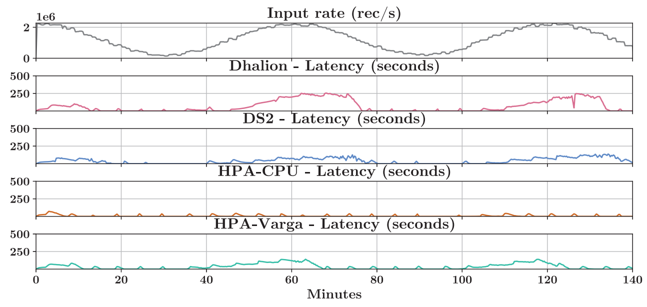
– *Performance on SWA.* Only DS2 manages to follow the progression of the input pattern for the *SWA* query. However, latency remains high for all periods of medium to high input rates. Both HPA-based



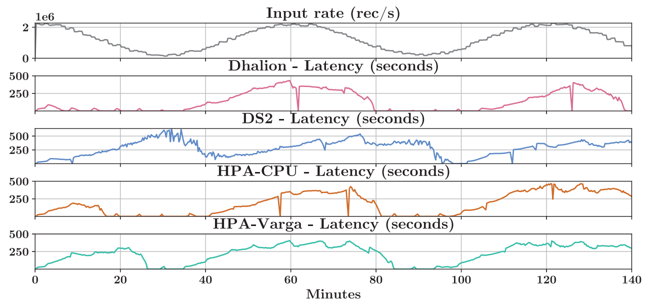
(a) Workers deployed for *Map*.



(b) Workers deployed for *SWA*.



(c) Latency of *Map*.



(d) Latency of *SWA*.

Figure 5: Cosine pattern

autoscalers fail to match the input at any moment, and after a while, they constantly scale up, trying to deal with the lag accumulated in the input queue. Although Dhalion does not keep up with the changes in the input rate, it has the best overall performance in terms of latency and the longest periods of low latency.

6.3 Query Comparison

Although the *Map* and the *Session Window Aggregate* queries are representative processing tasks, we evaluate the autoscalers on additional queries for completeness. Due to space limitations, we only present the performance of additional queries deployed on the cosine workload. Figure 7 illustrates the performance of the autoscalers on the additional queries.

Performance on Filter. The *Filter* query belongs to the same class of stateless low-complexity computation tasks as the *Map* query. As so, we expect similar behavior from the autoscalers. Indeed, DS2 and HPA-CPU show the same behavior as in *Map*. During peak periods of activity, Dhalion follows the input with a slight delay, resulting in high latency; similar to Dhalion's behavior in *Map*. HPA-Varga has a similar resource provisioning as in *Map* but performs slightly better in terms of latency because of better resource allocation.

Performance on Incremental Join. *Incremental Join* is the most complex query employed. This gives us an opportunity to evaluate the autoscalers on a system under stress, even during low input rates. Dhalion is the only autoscaler that matches the input rate and achieves low latency for the whole experiment duration. However, it employs twice as many resources as DS2 or HPA-CPU. DS2 and HPA-CPU minimize the scaling actions and the deployed resources but also see latency increases. HPA-Varga reacts slowly to the input increases, reflecting high latency during periods of high input rate.

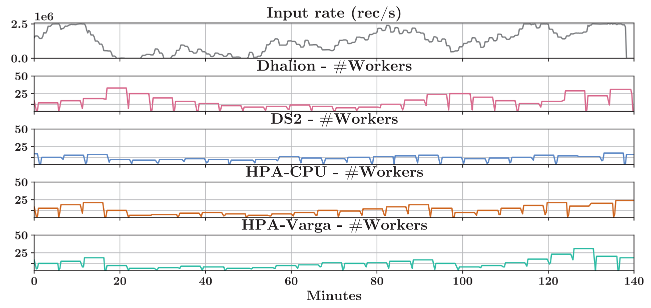
Performance on Sliding Windowed Aggregate. The *Sliding Windowed Aggregate* (SA) differentiates from *SWA* in the type of window employed. While a session window produces a constant flow of records throughout the system, a sliding window produces output only when a time interval ends.

For *SA*, HPA-CPU performs the best as it follows the input pattern and keeps latency low while provisioning a minimal number of workers for the entire run. DS2 cannot align with the input pattern as it perceives dead periods as underprovisioning. Thus, it decides that the system underperforms and falsely raises the resources. Dhalion has competitive performance with delayed scale-up decisions that lead to small latency spikes when the input rate increases. At the same time, on average, it deploys more workers than HPA-CPU. HPA-Varga provisions resources similarly to HPA-CPU but fails to allocate them correctly, impacting the latency.

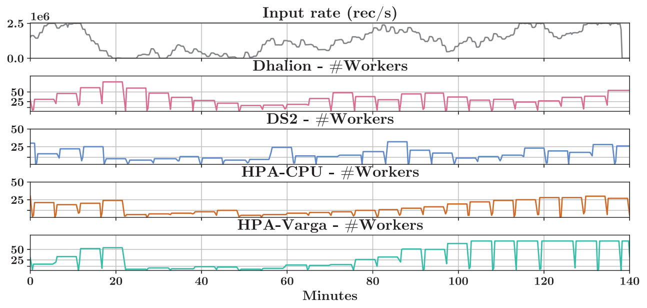
6.4 Convergence comparison

We perform a convergence experiment to evaluate the ability of the autoscalers to converge to an optimal configuration, given a specific input rate. We use the steps workload discussed in Section 5.3 and the *Map* and *SWA* queries. The experiment assesses the time and scaling actions required to converge to an optimal configuration.

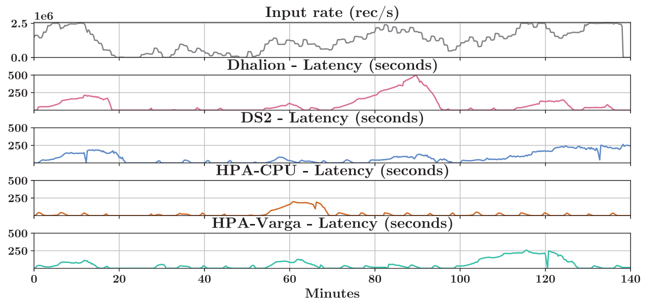
Figures 10 and 11 show the deployment of workers over time. Dhalion shows a slow reaction to input changes. As a result, it fails to converge within the provided time frame to any of the two input rates. HPA-Varga reacts slowly to the increased input rate and does not converge regardless of the input rate and the query. Surprisingly HPA-CPU is also slow to react to the high input change



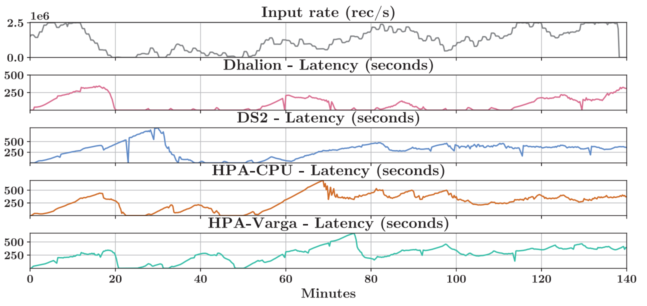
(a) Workers deployed for *Map*.



(b) Workers deployed for *SWA*.



(c) Latency of *Map*.



(d) Latency of *SWA*.

Figure 6: Random pattern

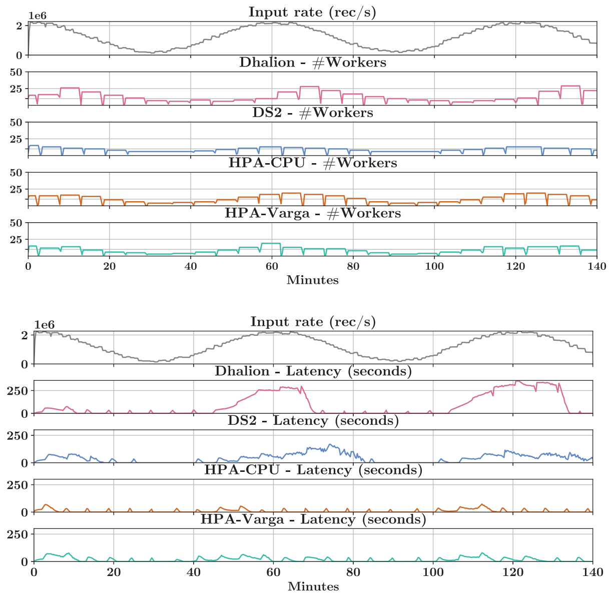


Figure 7: *Filter.*

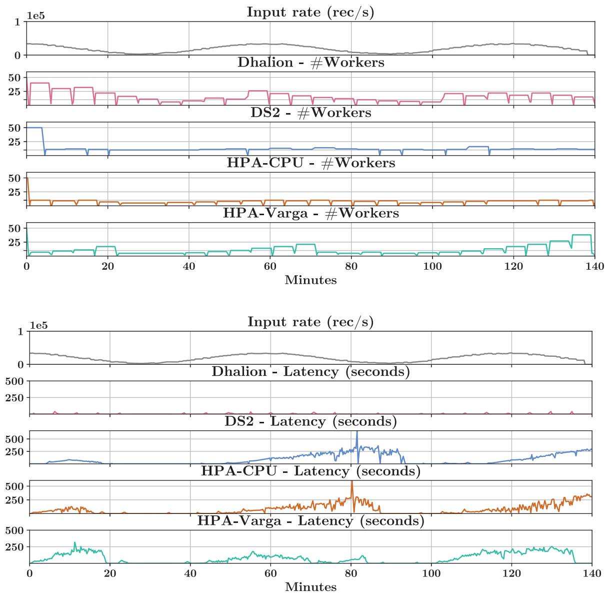


Figure 8: *Incremental Join.*

for both queries. However, it converges for the medium input rate within two scaling actions for the *Map* query. We deploy DS2 with and without overprovisioning. Both versions react quickly to input changes but only manage to converge for the medium input rate of the *Map* query within two and three scaling actions, respectively. Although DS2, without overprovisioning, temporarily decides on a stable configuration, it continues oscillating after a while.

6.5 Summary of findings

In our experiments with different workloads, we observed varying behavior among autoscalers. DS2 consistently follows the input pattern but may encounter occasional high latency. Dhalion struggles to adapt to less complex patterns, reacts slowly to more complex patterns, and allows for high latency. HPA-Varga generally aligns

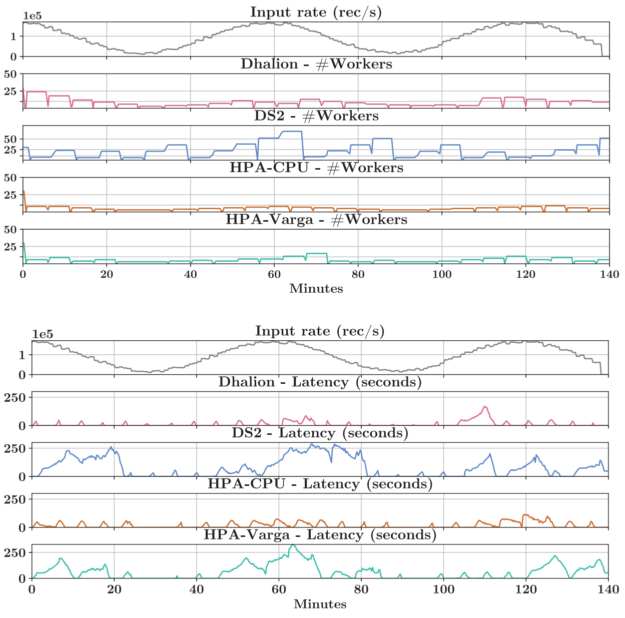


Figure 9: *Sliding Windowed Aggregate.*

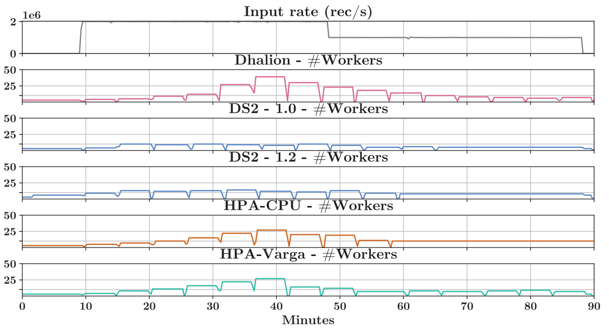


Figure 10: Convergence of *Map.*

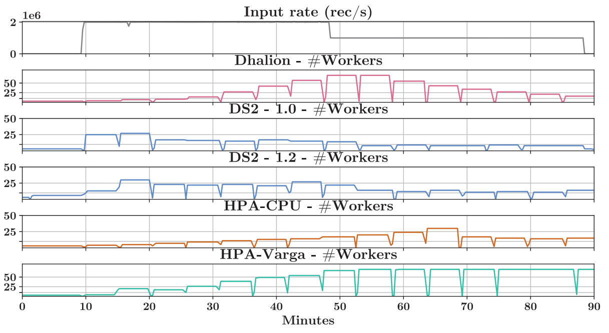


Figure 11: Convergence of *SWA.*

with the input patterns but may react slowly to input rate increase. HPA-CPU outperforms all autoscalers for less complex queries in terms of latency, adjustment, and resource utilization. However, HPA-CPU fails to sustain low latency when facing high-input periods for more demanding queries.

Experiments with additional queries support our earlier observations. HPA-based autoscalers perform well for stateless queries. HPA-CPU matches the input pattern while maintaining low latency. HPA-CPU also performs adequately for the *sliding windowed aggregate* query but fails to provide enough resources in the case of the *Incremental Join* query. HPA-Varga performs worse on complex queries, both in terms of latency and deployed workers. Dhalion reacts slowly to input changes and allocates resources inefficiently, leading to high latency. DS2 allocates fewer resources and avoids unnecessary scaling actions. However, it struggles to maintain low latency in high throughput periods, especially for complex queries. Finally, our convergence experiments show that none of the evaluated autoscalers can converge within the time limits of our experiments for complex queries. Only HPA-CPU and DS2 converge when scaling down from a higher load to a medium input rate.

The design choices of each autoscaler reflect on its performance. DS2 adjusts to the input rate accurately and fast due to effective metrics and efficient scaling of multiple operators at once, by propagating changes to downstream operators. However, DS2 does not consider the lag generated when it issues scaling actions, leading to high latency. Dhalion relies on backpressure and input buffer usage to decide on scaling actions. However, when backpressure can be detected, the system has already entered an unhealthy state. Additionally, Dhalion only scales a single operator in each scaling action, reacting slowly to changes. Dhalion fails to distribute efficiently resources to the operators resulting to unstable performance and slow convergence. HPA-CPU solely depends on CPU load, which may not accurately reflect the performance impact of complex stateful queries that involve accessing large datasets from memory or disk. Despite our best efforts, we could not overcome that HPA-Varga is designed to work on a deployment level rather than on an operator level. Its utilization metric can be directly measured per operator, while the relative lag can be calculated only indirectly.

As seen in our experiments, the evaluated autoscalers are affected by the lag generated during the rescaling actions, and none can currently consider it when deciding on the optimal configuration. This generated lag is partially a side effect of current systems' inability to migrate their state without a stop-and-restart process. Although the problem of state migration is orthogonal to autoscaling, it plays a crucial role in the performance. Despite prior work introducing proposals for on-the-fly state migration techniques with low overhead, stream processing systems have yet to adopt it.

Previous evaluations. The original evaluation of Dhalion shows a necessity for numerous rescaling actions and considerable time to achieve desired throughput convergence. Our experiments validate this observation, as the time frame are insufficient for Dhalion to reach convergence. In our work, we observe that Dhalion deploys more resources than the other autoscalers. The findings from the DS2 evaluation show lower resource deployments and faster convergence with fewer actions than Dhalion. Unlike the original evaluation, DS2 does not always converge within the time frame set in our experiments. HPA-Varga and HPA-CPU do not conduct an evaluation using the same metrics or offer similar insights.

Limitations. In this work, we propose a principled evaluation framework for evaluating control-based autoscalers. Evaluating additional autoscalers under this framework can provide rich insights, as we showcase with our experiments, and might lead to different conclusions. Furthermore, we evaluate the performance of the autoscalers on top of Apache Flink. Our evaluation framework assumes durable input and output queues and a stream processing engine that allows for per-operator scaling. The evaluated autoscalers are agnostic to the specifics of the underlying rescaling mechanisms of an SPE and only require specific metrics provided by the engine as well as a rate control mechanism in the case of Dhalion. However, extending the current evaluation to other stream processing engines, such as Storm[3] and Heron [21], can provide valuable insights regarding the autoscalers' applicability and the configurations' performance based on the rescaling mechanisms.

7 CONCLUSION

In this work, we highlighted the lack of significant comparison between existing autoscaling solutions in stream processing. We provided a principled experimental framework to evaluate performance and identify unsolved challenges. We extensively evaluated four control-based autoscalers on dynamic workloads and queries. Surprisingly, a method utilizing CPU usage outperforms state-of-the-art solutions for minimal queries in all workloads. We showcased that none of the evaluated autoscalers can perform well for complex queries over highly dynamic workloads. We discuss the impact of the autoscalers' design choices on their performance, and we argue that the poor performance of the evaluated autoscalers is a result of their inability to account for the lag generated during an autoscaling action or due to slow reactions to the input changes. Finally, we urge stream processing engines to adopt online state migration techniques as it would significantly improve the performance of autoscaling.

ACKNOWLEDGMENT

This publication is part of project number 19708 of the Vidi research program, partly financed by the Dutch Research Council (NWO). It is also partially funded by ICAI AI for Fintech Research Lab. We would like to thank Job Kanis for his contribution to the early stages of development.

REFERENCES

[1] Extended nexmark benchmark from apache beam project. https://beam.apache.org/documentation/sdks/java/testing/nexmark/. Accessed: 2024-2-20.

[2] Kubernetes horizontal pod autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed: 2024-2-20.

[3] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 480–491.

[4] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth. Model-based stream processing auto-scaling in geo-distributed environments. In *30th International Conference on Computer Communications and Networks, ICCCN 2021, Athens, Greece, July 19-22, 2021*, pages 1–10. IEEE, 2021.

[5] M. V. Bordin, D. Griebler, G. Mencagli, C. F. R. Geyer, and L. G. L. Fernandes. Dspbench: A suite of benchmark applications for distributed data stream processing systems. *IEEE Access*, 8:222900–222917, 2020.

[6] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo. Decentralized self-adaptation for elastic data stream processing. *Future Gener. Comput. Syst.*, 87:171–185, 2018.

[7] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. Peng, and P. Poulosky. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2016, Chicago, IL, USA, May 23-27, 2016*, pages 1789–1792, 2016.

[8] D. N. Doan, D. Zaharie, and D. Petcu. Auto-scaling for a streaming architecture with fuzzy deep reinforcement learning. In *Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops*, volume 11997, pages 476–488, 2019.

[9] A. Floratou, A. Agrawal, B. Graham, S. Rao, and K. Ramasamy. Dhalion: Self-regulating stream processing in heron. *PVLDB*, 2017.

[10] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 411–420, 2015.

[11] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. DRS: auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):3338–3352, 2017.

[12] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang. Drs: Auto-scaling for real-time stream analytics. *IEEE/ACM Trans. Netw.*, 25(6):3338–3352, 2017.

[13] B. Gedik, S. Schneider, M. Hirzel, and K. Wu. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distributed Syst.*, 25(6):1447–1463, 2014.

[14] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *Workshops Proceedings of the 30th International Conference on Data Engineering Workshops, ICDE 2014, Chicago, IL, USA, March 31 - April 4, 2014*, pages 296–302. IEEE Computer Society, 2014.

[15] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer. Auto-scaling techniques for elastic data stream processing. In *The 8th ACM International Conference on Distributed Event-Based Systems, DEBS '14, Mumbai, India, May 26-29, 2014*, pages 318–321. ACM, 2014.

[16] G. Hesse, C. Matthies, M. Perscheid, M. Uflacker, and H. Plattner. Espbench: The enterprise stream processing benchmark. In *ICPE '21: ACM/SPEC International Conference on Performance Engineering, Virtual Event, France, April 19-21, 2021*, pages 201–212.

[17] N. Hidalgo, D. Wladdimiro, and E. Rosas. Self-adaptive processing graph with operator fission for elastic stream processing. *J. Syst. Softw.*, 127:205–216, 2017.

[18] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar. Elastic stream processing for the internet of things. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 100–107, 2016.

[19] 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 *Usenix OSDI*, 2018.

[20] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *2018 IEEE 34th International Conference on Data Engineering*, pages 1507–1518, 2018.

[21] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, page 239–250, New York, NY, USA, 2015.

[22] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, pages 53:1–53:8, 2015.

[23] X. Liu, A. V. Dastjerdi, R. N. Calheiros, C. Qu, and R. Buyya. A stepwise auto-profiling method for performance optimization of streaming applications. *ACM Trans. Auton. Adapt. Syst.*, 12(4):24:1–24:33, 2018.

[24] B. Lohrmann, P. Janacik, and O. Kao. Elastic stream processing with latency guarantees. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS, Columbus, USA, 2015*, pages 399–410.

[25] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Trans. Parallel Distributed Syst.*, 29(3):572–585, 2018.

[26] F. Lombardi, A. Muti, L. Aniello, R. Baldoni, S. Bonomi, and L. Querzoni. PASCAL: an architecture for proactive auto-scaling of distributed services. *Future Gener. Comput. Syst.*, 98:342–361, 2019.

[27] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 2014.

[28] R. Lu, G. Wu, B. Xie, and J. Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2014, London, United Kingdom, December 8-11, 2014*, pages 69–78, 2014.

[29] G. Mencagli, M. Torquati, and M. Danelutto. Elastic-ppq: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Gener. Comput. Syst.*, 79:862–877, 2018.

[30] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.*, 52(2), 2019.

[31] P. Tucker, K. Tufte, V. Papadimos, and D. Maier. Nexmark–a benchmark for queries over data streams (draft). Technical report, 2008.

[32] G. van Dongen and D. V. den Poel. Evaluation of stream processing frameworks. *IEEE Trans. Parallel Distributed Syst.*, 31(8):1845–1858, 2020.

[3] https://storm.apache.org

[33] B. Varga, M. Balassi, and A. Kiss. Towards autoscaling of apache flink jobs. *Acta Universitatis Sapientiae, Informatica*, 13:1–21, 2021.

[34] B. Varga, M. Balassi, and A. Kiss. Towards autoscaling of apache flink jobs. *Acta Universitatis Sapientiae, Informatica*, 13(1):39–59, 2021.