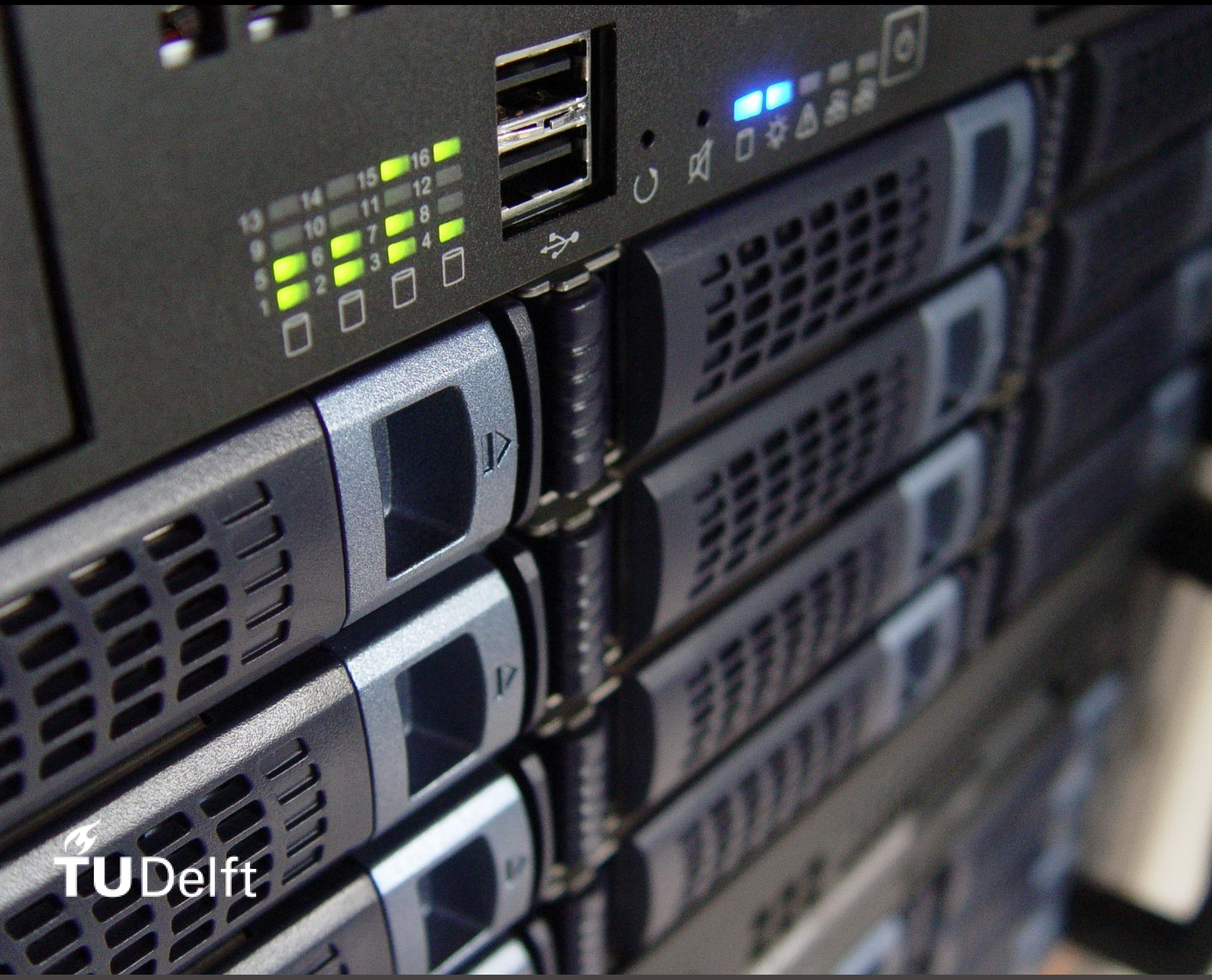
An Experimental Evaluation of Auto-scaling Techniques for Distributed Stream Processing Systems

J.B. Kanis



An Experimental Evaluation of Auto-scaling Techniques for Distributed Stream Processing Systems

by

J. B. Kanis

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on March 28, 2023 at 15:30 PM.

Student number:	4719808	
Project duration:	April 20, 2022 – March 28, 2023	
Thesis committee:	Dr. A. Katsifodimos,	TU Delft, supervisor
	Dr. J.E.A.P. Decouchant,	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Abstract

The introduction of cloud hosting has made it possible to elastically provision distributed stream processing systems (SPEs). By dynamically scaling the different operators of the system, resource consumption can be minimised while meeting the system service-level objectives. In the literature, many different auto-scaling techniques are proposed that make scaling decisions based on the current state of the system. However, these techniques are poorly evaluated and are rarely compared with each other. This makes it difficult to determine the state-of-the-art for auto-scaling techniques targeting SPEs, which slows down its development. In this paper, we design and implement a modular framework to evaluate the performance of state-of-the-art auto-scalers targeting SPEs. We implement state-of-the-art auto-scalers Dhalion [18], DS2 [36], and Varga et al. [66], using Kubernetes horizontal pod auto-scaler [38] as baseline. We perform an end-to-end experimental evaluation of the auto-scalers and investigate their performance when run on different queries and workload patterns. Furthermore, we investigate the convergence time of the auto-scalers and evaluate their scaling accuracy. The results emphasise the difficulty of capturing the complex relationships of different operators and the struggle to balance resource efficiency and the performance of the system. Moreover, it shows the inherent weakness of reactive auto-scalers to react slowly to changing workloads and reveals the importance of considering the current health of the system when issuing scaling actions.

Preface

Before you lies my thesis as a conclusion to my study at Delft University of Technology. Starting my thesis with a thrilling search for a topic surrounding online similarity joins, I eventually ended up with an experimental evaluation of auto-scalers for distributed stream processing systems. Although reading papers appeared to be not my favourite occupation, I really enjoyed the technical part of implementing the experimental framework and running the experiments. With this thesis I can finally see the work come to fruition, but not without the support of certain people. First, I want to thank my co-daily supervisor Georgios Siachamis for successfully guiding me to the end of this thesis. Our weekly meetings provided structure and motivation to continue working. Furthermore, his flexibility and fast responses to my daily spam has helped me through the project. Secondly, I want to thank my daily supervisor and thesis advisor Asterios Katsifodimos for being there whenever I needed my stack of documents signed and providing the necessary guidance to ensure I can successfully finish the thesis. I really appreciate his enthusiastic responses when discussing my process and will never forget the uncountable 'good job' jokes. Finally, I want to thank my family, friends, and of course my girlfriend for being there for me, providing the safe environment in which I was able to finish my thesis.

J. B. Kanis
Delft, March 2023

Contents

Abstract iii

1 Introduction 1

2 Background 5
2.1 Cloud Computing and elasticity 5
2.2 Auto-scalers 5
2.2.1 MAPE-loop 6
2.2.2 Areas of deployment 7
2.3 Auto-scaling Techniques 8
2.3.1 Threshold-based rules 8
2.3.2 Reinforcement learning 8
2.3.3 Queuing theory 9
2.3.4 Control theory 9
2.3.5 Time series analysis 9
2.4 Stream Processing 10
2.4.1 The basics of stream processing 10
2.4.2 Stream processing at run-time 11
2.4.3 Elasticity and stream processing applications 12
2.5 Experimental evaluation of auto-scalers for distributed stream processing 12
2.5.1 Paper selection 13
2.5.2 Experiment types 13
2.5.3 Performance analysis metrics 14
2.5.4 Conclusion 16

3 Related work 17

4 Auto-scalers for Stream Processing 19
4.1 Paper selection 19
4.2 Dhalion 19
4.3 DS2 20
4.4 HPA 21
4.5 HPA-Varga 21
4.6 Conclusion 22

5 Experimental design 23
5.1 Experiment requirements 23
5.2 Experimental types 24
5.3 Performance metrics 25
5.4 End-to-end Auto-scaler evaluation 26
5.4.1 Bench-marking framework 26
5.4.2 Queries 26
5.4.3 Workloads 28
5.5 Stream Processing Engine 29
5.5.1 Architecture 29
5.5.2 operators 30
5.5.3 Dynamic scaling 31
5.5.4 Backpressure mechanism 31
5.5.5 Available Metrics 31
5.6 Container deployment 31
5.6.1 Containers 31

5.6.2 Kubernetes . . . 32
5.7 Messaging System . . . 32
5.7.1 The need for a messaging system . . . 32
5.7.2 Apache Kafka . . . 33
5.8 Monitoring System . . . 33
5.8.1 Prometheus . . . 33
5.8.2 Conclusion . . . 34

6 Implementation **35**
6.1 Kubernetes . . . 35
6.2 Data Generator . . . 35
6.2.1 Architecture . . . 36
6.2.2 Implementation . . . 37
6.3 Apache Kafka . . . 37
6.3.1 Implementation . . . 37
6.3.2 Metrics . . . 37
6.4 Apache Flink . . . 37
6.4.1 Scaling . . . 38
6.4.2 Metrics . . . 38
6.5 Prometheus . . . 39
6.6 Auto-scaler Architecture . . . 39
6.6.1 Auto-scaler Logic . . . 39
6.6.2 Configurations . . . 39
6.6.3 Application Manager . . . 41
6.6.4 Scale-Manager . . . 41
6.6.5 Implementation . . . 42
6.7 Auto-scaler Logic . . . 42
6.7.1 Dhalion Implementation . . . 42
6.7.2 DS2 Implementation . . . 43
6.7.3 HPA-CPU Implementation . . . 44
6.7.4 HPA-Varga Implementation . . . 44
6.8 Conclusion . . . 45

7 Results **47**
7.1 Parameter Optimisation . . . 47
7.1.1 Experimental Setup . . . 48
7.1.2 Dhalion . . . 48
7.1.3 DS2 . . . 50
7.1.4 HPA-Varga . . . 52
7.1.5 HPA-CPU . . . 54
7.1.6 Conclusion . . . 57
7.2 Workload Comparison . . . 58
7.2.1 Cosine pattern . . . 58
7.2.2 Random . . . 60
7.2.3 Increasing workload . . . 63
7.2.4 Decreasing . . . 65
7.3 Query Comparison . . . 68
7.3.1 Queries 1 and 2 . . . 68
7.3.2 Query 3 . . . 69
7.3.3 Queries 5 and 11 . . . 69
7.3.4 Conclusion . . . 70
7.4 Convergence Time . . . 71
7.5 Overview of the results . . . 73
7.5.1 Experimental evaluation . . . 73
7.5.2 Auto-scaler performance . . . 74

8 Discussion and future work **77**
8.1 Discussion . . . 77

8.2 Limitations 78
8.3 Future work 78

1 Introduction

With the increase in available data on the web, more and more applications require data to be processed in an online fashion. As data streams are infinite, have non-static arrival rates, and require on-demand processing, additional challenges are brought to the field of data processing. While the first stream-processing engines were designed to run only on individual servers, the increase in data availability combined with the need for fast and real-time results has made distributed processing essential. As a result, stream processing has shifted towards distributed stream processing. Popular Stream Processing Engines (SPEs), such as Apache Flink [15], Apache Storm [62], or Apache Heron [30] do not only support parallel stream computations but also support deploying stream processing operators in elastic cloud environments.

This brings additional challenges in terms of providing adequate resources for SPEs. Traditionally, static provisioning required the user to reserve resources before the application's deployment, forcing the users to rent the number of resources required for peak traffic to meet the system's service-level objectives (SLOs). This results in the system being over-provisioned most of the time, inducing significant overhead costs for users. With the emergence of cloud-computing technology, the process of acquiring and releasing resources has been simplified, allowing for elastic provisioning of resources and renting resources on-demand [45]. By reducing the number of resources used by the application in low traffic and increasing the number of resources in peak traffic, users can use the resources more efficiently, reducing resource costs and energy consumption while meeting the system's SLOs.

However, acquiring and releasing resources on demand is a complex task that requires constant monitoring of the application. Appropriate scaling decisions should be made that take the health of the system, the current and expected workload, and the complex relationships between different parts of the system into consideration. Moreover, as scaling actions can be costly and take time to execute, inaccurate decisions can induce significant overhead, leaving the system in an undesirable state for a long time. Manual execution of this task is time-consuming, expensive, and requires extensive knowledge of the application. Because of this, the process is often performed by automated systems that are put in charge of making scaling decisions. These systems are referred to as auto-scalers.

In the literature, many different auto-scalers are proposed for the dynamic scaling of SPEs. As SPEs deal with high workload fluctuations, complex relationships between their operators and high scaling overhead, elastic provisioning of SPEs is a difficult challenge. Auto-scalers attempt to solve this problem by making use of various techniques like control-theory [36], queue theory [20], or reinforcement learning [44, 12]. While these auto-scalers are well designed and depend on well-established theoretical frameworks, they lack proper evaluation of their performance. Additionally, evaluation experiments often require only a few scaling actions and are run over simple workload patterns for a short amount of time. An overview of the evaluation benchmarks and their comparison with different auto-scalers is presented in table 1.1. As shown, experiments are performed over many different data sets and are rarely compared with other auto-scalers. This makes it difficult to determine the current state-of-the-art for auto-scaler for SPEs and slows down the development of these solutions.

Paper	Year	Dataset	Autoscaler Comparison
DRS [19, 20]	2017	Pattern detection over video and micro-blogs	No
Hidalgo et al. [31]	2017	Processing of a Twitter- and news stream	No
Liu et al. [41]	2017	Sentiment analysis, 2-stage word count topology	Stela [67], Heinze et al. [27]
Lombardi et al. [43]	2017	Twitter dataset real-world and syntactic	No
Dhalion [18]	2017	2-stage word count topology	No
Elastic-PPQ [48]	2018	DEBS 2013 non-burstiness & bursty workloads	No
Cardellini et al. [8]	2018	Query from DEBS 2015 grand challenge	No
DS2 [36]	2018	2-stage word count topology	Dhalion
Doan et al. [12]	2020	Clarcknet Traces [63]	Arabnejad et al. [3]
Varga et al. [66]	2021	A keyed processing function over generated data	No

Table 1.1: Summary of evaluation methods of recent proposals for auto-scalers for SPEs. The table shows the use of diverse datasets for the evaluation of auto-scalers and a general lack of proper performance comparisons between auto-scalers.

In this work, we design and implement a framework for evaluating the performance of auto-scalers for SPEs. We implement the framework for Apache Flink [15] and deploy the framework in Kubernetes [38]. Using this framework, we implement and evaluate state-of-the-art auto-scalers Dhalion [18], DS2 [36], and Varga et al. [66]. As a baseline, we use Kubernetes' built-in general-purpose Horizontal Pod Auto-scaler (HPA) [38]. In this evaluation, we compare the performance of the auto-scalers when run on top of different operator topologies and different workloads. We investigate the time the auto-scalers need to find the optimal parallelism configuration and analyse the performance of the auto-scalers when run with different configurations.

In this work we make the following contributions:

- We provide an overview of state-of-the-art auto-scaling solutions for SPEs and provide an extensive analysis of their evaluation methods.
- We distinguish five categories of experiments used for the evaluation of auto-scaling solutions for SPEs and investigate common metrics used for the comparison of their performance.
- We tackle the shortcomings of state-of-the-art evaluation methods for auto-scalers targeting SPEs and design a modular framework for the end-to-end evaluation of their performance. The framework is designed to be easily extensible with additional operator topologies, workloads, and auto-scalers.
- We extend auto-scalers HPA [38] and Varga et al. [66] to support operator-based scaling and implement them together with state-of-the-art auto-scalers Dhalion [18] and DS2 [36] in the framework.
- We investigate the sensitivity of the parameters of the auto-scalers and determine the optimal parameters for the auto-scalers to be used in the experimental evaluation.
- We compare the performance of state-of-the-art auto-scalers when run with different auto-scaler configurations under a diverse set of workloads and operator topologies.
- We make several suggestions for future research in the area of auto-scalers for SPEs and their evaluation methods.

We start with an investigation into the state-of-the-art of auto-scalers for SPEs. In chapter 2, we introduce the topic of auto-scaling, cloud computing, and stream processing. Furthermore, it includes a literature survey regarding evaluation methods for auto-scalers for SPEs and provides an overview of common metrics used to investigate their performance. Chapter 3 contains a literature overview regarding the development of auto-scalers for SPEs, followed by a more detailed description of the state-of-the-art in chapter 4. This is followed by chapter 5 where we discuss the design for the evaluation framework. Here we discuss the different technologies we will use, introduce different workload

patterns and discuss the different queries the experiments will run on. Chapter 6 discusses the implementation of this framework, together with the different design choices made. This section can be especially useful for the reader who wishes to extend the framework with additional workloads, queries, or auto-scalers. Finally, we discuss the results of the experiments in chapter 7. Finally, chapter 8 discusses the results obtained in chapter 7 and provides several suggestions for future research.

2
Background

2.1. Cloud Computing and elasticity

Cloud computing and elasticity Cloud computing is a broad paradigm that is focused on the on-demand provisioning of infrastructures, services and applications [47]. Over the past decade, cloud systems like Amazon AWS[1], Microsoft Azure [2], and Google Cloud [3] have been employed to work as management middle-ware to provide processing power, storage space, and bandwidth to clients [45]. Cloud users can rent these resources on demand and run their applications on them. Doing so, these infrastructures allow for elastic scaling of resources while paying as you go. Key characteristics of cloud computing include the extensive use of virtualisation (using virtual machines and containerisation), unlimited scalability of resources and extensive service level objectives (SLOs) that both the cloud's providers and users have to meet [7].

Provisioning A question brought forward by this development is how to make efficient use of these services. On one hand, cloud users want to minimise costs and rent as few resources as possible, while they are, at the same time, required to meet the SLOs of their users. Traditionally, this would require cloud users to rent enough resources to be able to handle peak traffic without SLO violations. On low traffic this would result in *over-provisioning*, inducing additional costs for running the application. As an alternative, cloud users can choose to rent fewer resources, reducing costs, but resulting in *under-provisioning* in peak traffic, leading to SLO violations.

Elastic provisioning With the introduction of elastic scaling in cloud computing, users are now able to use *elastic-provisioning*, where the number of resources used corresponds to the current workload of the system. Optimally, the number of resources used at any point in time should be the minimal amount of resources required to process the incoming workload without any SLO violations. In this work, we refer to this as *optimal-provisioning*. The challenge here is, however, to decide *when* to scale the system and *how*. This is a difficult task that requires constant monitoring of the system. Scaling decisions should take the health of the system, the current and expected workload, and the complex relationships between different parts of the system into consideration. As performing these tasks manually is difficult and expensive, scaling decisions are often made by automated systems called auto-scalers.

2.2. Auto-scalers

Auto-scalers are automatic systems in charge of the scaling activities of an application, without requiring human intervention [45]. Auto-scalers determine *when* to scale the application and *how*. The goal of an auto-scaler is to automatically scale the application to minimise the number of resources used,

[1] `https://aws.amazon.com/`
[2] `https://azure.microsoft.com/en-us`
[3] `https://cloud.google.com/`

while at the same time minimising SLO violations. In the literature, different types of auto-scalers have been suggested, addressing different area-specific challenges. In this section, we further discuss auto-scalers and how they follow the MAPE-loop for autonomous systems. We then discuss different areas of deployment and different techniques used for making scaling decisions and estimating the future workload of the system.

2.2.1. MAPE-loop

Auto-scalers follow the MAPE-loop for autonomous systems [33, 47]. The MAPE-loop (as shown in figure 2.1) consists of a loop of four phases: monitoring, analysis, planning, and execution. Of these four steps, auto-scalers are concerned with the analysis and planning phase, leaving the other two phases to external systems. In this section, we discuss each of the four phases individually and comment on the role auto-scalers have in these phases.

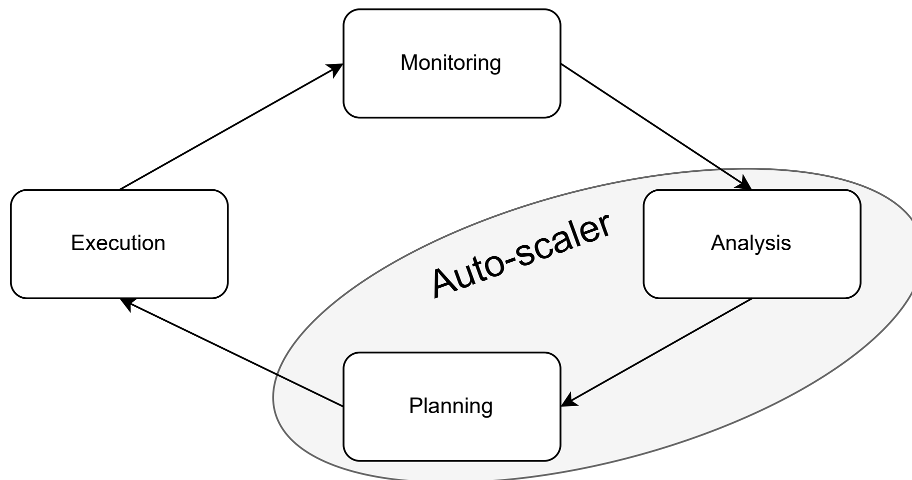


Figure 2.1: Visualisation of the MAPE-loop for autonomous systems [33]. Auto-scalers are only concerned with the analysis and planning phase, leaving the monitoring and execution phase to external systems.

Monitoring phase In the monitoring phase, a monitoring tool monitors the employed system and collects the necessary information the auto-scaler needs to make scaling decisions. This is not done by the auto-scaler itself, but by an external monitoring tool. The monitoring tool collects performance metrics, whose quality highly influences the quality of the auto-scaler [45]. Based on the source of the metrics, Ghanbari et al. [24] categorise performance metrics into seven categories: hardware, general OS processes, load balancer, web server, application server, database server, or message queue. Overall, the monitoring tool needs to minimise additional overhead when collecting metrics, as this can have a significant impact on the performance of the processing system. Many auto-scaler proposals like DRS [20] or DS2 [36] spend additional efforts in their auto-scaler evaluation to ensure the monitoring system does not induce significant costs on the overall system.

Analysis phase The performance metrics collected by the external monitoring tool are passed to the auto-scaler for analysis. In the analysis phase, the collected raw metrics are combined into more complex metrics, like the true processing rate used by Kalavri et al. [36], or the relative lag change, proposed by Varga et al. [66]. Other approaches use the data to predict future workloads [31, 43]. This allows the auto-scalers to perform scaling actions before the actual workload change occurs, which is especially useful in systems where adding resources may take a long time. Systems that base their scaling decisions on predicted future workloads are called *pro-active* auto-scalers. Auto-scalers that make scaling decisions on the current workload are called *reactive* auto-scalers. While reactive auto-scalers may respond too late to workload changes, proactive auto-scalers can be inaccurate in their workload estimations, leading to additional costs.

Planning phase After the data analysis, and the current or future state of the application is known, the auto-scaler decides on the scaling actions to undertake. With these actions, the auto-scaler attempts to both minimise the number of resources used and the amount of SLO violations.

Execution phase After the auto-scaler suggests a scaling action, the action is forwarded to an external system that is responsible for its execution. This may be done by calling a cloud provider's API, or with a specialised module provided by the application itself. The auto-scaler is not aware of the complexities of the underlying system and does not deal with it [45]. Scaling an application takes time and may temporarily reduce the performance of the application. To prevent unnecessary costs, the auto-scaler needs to be aware of this and propose scaling actions appropriately.

2.2.2. Areas of deployment

Many different applications are hosted on cloud infrastructures and support elastic scaling. As these different applications have different complexities and different needs for resource consumption, auto-scalers are designed for specific applications, like distributed data stores, distributed stream processing engines, and high-performance computing. While some auto-scalers focus on one of these specific applications, other auto-scalers attempt to generalise over different applications to propose general-purpose auto-scalers. In this section, we discuss the different applications auto-scalers are designed for and comment on their underlying complexities.

General purpose General-purpose auto-scalers are designed to be used for any type of application. This requires the auto-scaler to generalise over different types of applications and make simplifying assumptions. The workload of the system is estimated using general metrics such as the average CPU load and workload intensity. While these metrics are shown to correlate well with quality-of-service metrics [56], they only provide a simplified insight into the performance of the application and do not work well for all scenarios. While some general purpose operators (like [44]) do allow defining custom metrics, the methods are not able to capture the sophisticated underlying relationship between different parts of the application, resulting in decreased performance. The evaluation of general-purpose auto-scalers is generally done on the business-logic tier of web applications [22, 2]. These systems generally consist of self-contained units that do not require the auto-scaler to consider their underlying relationships. Some auto-scalers consider multi-tier applications, where different parts of the system can be scaled individually. Nisar et al. [1], for example, split a web application into both a business-tier and a database-tier aspect and suggest scaling actions for both of them individually.

Distributed data stores Distributed data stores are systems that manage large amounts of data and ensure the accessibility of the data for its users. For this, the system is not only concerned with managing fluctuating workloads where the system must process incoming data requests, but it should also consider the smart placement of specific data on servers and use replication for both accessibility and fault-tolerance of the system. In addition, databases tend to have specific requirements such as ACID compliance, making scaling itself already a difficult challenge. To solve this problem, Barker et al. [5] propose ShuttleDB that introduces a migration and replication method, combining VM-level and database level scaling. They propose an auto-scaler that uses the latency of the system to issue scaling actions accordingly. Casalicchio et al. [9] design an auto-scaler for the already elastic data store Cassandra. The auto-scaler attempts to minimise the energy consumption of the system while ensuring data availability and scaling according to the workload.

Distributed stream processing systems Distributed stream processing systems consist of different operators that continuously retrieve, process, and emit data. These operators are linked to each other in a topology, having data flow from operator to the operator until it eventually leaves the system. This makes the operators strongly dependent on each other. Given that every operator has different processing speeds and differs in the amount of data they output, deciding which operators to scale and when to do so, is a difficult challenge. Scaling a single operator may influence the upstream operators in the topology, also requiring scaling operators in those systems. As addressed in Floratou et al. [18], and Kalavri et al. [36], determining the optimal operator-specific parallelism can be a difficult challenge. Allowing the operators to share the resources between operators, simplifies this process significantly.

Varga et al. [66] propose an auto-scaler that only scales the available workers of the system, having the SPE dividing the resources between operators dynamically using Flink Reactive [49].

High performance computing High-performance computing (HPC) is concerned with the execution of complex calculations on high-performance infrastructures. To support elastic scaling, these systems often rely on specific generalisations, requiring the calculations to be implemented as batch jobs or by enforcing specific design patterns [34]. A more challenging field of HPC includes the processing of irregularly structured problems. These processes include tightly coupled parallel processes that are not always able to scale, making auto-scaling these applications a challenging task [57, 26].

2.3. Auto-scaling Techniques

Lorido-Botran et al. distinguish five categories for auto-scalers, based on the theory it is based on: using threshold-based rules, reinforcement learning, queuing theory, control theory, and time series analysis. We will discuss the five categories individually.

2.3.1. Threshold-based rules

Auto-scaling policies using threshold-based rules are one of the most simplistic scaling policies out there. They consist of several rules concerning one or more metrics. The rules can be formalised as a number of conditions that have to hold, followed by an action that should be performed. For example, if the latency of the system is above 10s for more than 1 minute, we scale the system up with 20% of its capacity. Its simplistic nature makes the policy easy to understand and intuitive for cloud users. Though, they are unable to cover the complex nature of the system and require extensive tuning of its parameters to achieve high performance [45]. Heinze et al. [28] address this issue by including a feedback system that optimises its thresholds to achieve higher performance. Simmons et al. [59] use a hierarchical strategy tree that switches between different policies depending on its performance over time.

The threshold-based approach only concerns the planning phase of the MAPE loop and is by default reactive in its nature. Though, by combining the approach with other techniques, it can be extended to a more sophisticated auto-scaler with better performance. The conditions of the rules can, for example, be based on a prediction module using queuing theory [53] or time series analysis [11], making the auto-scaler proactive.

2.3.2. Reinforcement learning

Another technique is to use reinforcement learning [60] for determining the scaling decisions of the auto-scaler [12, 8, 27, 61, 39]. A reinforcement learning model acts as an agent interacting with its environment. The agent selects and executes actions provided as an action set, changing the state of the environment. Based on the 'goodness' of the action, the agent receives a reward. Using trial and error, the agent learns to optimise the reward it receives. In the literature, multiple ways of 'learning' have been suggested, like using Q-learning and SARSA [60].

When using reinforcement learning for auto-scaling, the action set, state space, and reward function have to be defined. Both the action set and state space depend on the application, the auto-scaler is deployed in. The reward function generally takes into account the cost of acquiring resources (horizontally or vertically) and the cost of violating SLOs.

The main advantage of auto-scalers using reinforcement learning is that they require minimal configuration. While other techniques, like threshold-based rules, require tedious tweaking of parameters, a reinforcement model does not require prior knowledge about the workload distribution but figures this out by itself. Still, as Doan et al. [12] explain, designing the reward function of the system is no easy task and also requires tweaking of the parameters for optimal performance. In addition, the algorithm requires significant time to learn how to scale the application accurately. To minimise the training time, authors propose using a hybrid approach where an auto-scaler using threshold-based rules is run until the accuracy of the reinforcement model is good enough to be deployed [61, 39].

2.3.3. Queuing theory

Queuing theory [25] is the mathematical study of queues and has been extensively used for modelling internet applications. It is generally used to analyse systems of stationary nature, having constant arrival and processing rates. Given a queuing model and some input parameters, the model can predict performance metrics. In auto-scalers, this can be employed in the analysis phase when estimating the performance of the system in terms of queuing length or for estimating the expected latency. These prediction modules can be used to estimate future system performance [69, 65], or to estimate the expected performance of the system when given a specific amount of resources [50].

The standard description of a queuing model is Kendall's notation [58]. Here a queue is denoted as A/S/c/K/N/D, with A being the inter-arrival time distribution, S the service time distribution, c the number of servers, K the maximum queue length, N the population size (finite or infinite), and D the queue discipline (e.g. FIFO, LIFO). When K, N, or D are not present, $K = \infty$, $N = \infty$, and D = FIFO is assumed. This model of a queue can be used to model complex applications with different types of operators, creating a queuing network. Solving the model using periodically calculated parameters corresponding to the current status of the system, provides insights into the performance of the system that can be used as input for the planning phase of the MAPE loop. Some auto-scalers using control theory for their auto-scalers are [69, 65, 20, 50, 22].

2.3.4. Control theory

Control theory [40] is generally used for the automatic management of information processing systems. The idea is to deploy a model or an algorithm that steers the application towards a desired state. To do so, the controller has access to a controlled variable that it attempts to hold close to a desired value by adjusting the manipulated variable. For auto-scalers, the controlled variable can be, for example, the system's average CPU load, or its processing latency. The manipulated variable can, for example, be the parallelism for the system, or the number of resources assigned to the system. Some control-theory based auto-scalers are [68, 23, 48, 56, 36]. In general, three different types of controllers are used for auto-scaling [45]: fixed gain controllers, adaptive controllers, and model predictive controllers.

Fixed gain controllers Fixed gain controllers are the most simple ones and consist of a fixed model with pre-set parameters. The model represents the relationship between the control variable and the manipulated variable, allowing the controller to determine how to adjust the manipulated variable to move the control variable to the desired state. While this makes for a simple model with predictable behaviour, it requires a lot of manual fine-tuning of the parameters and is not able to adjust itself to a changing environment. This makes the controller undesirable as an auto-scaler when run in a dynamic environment.

Adaptive controller The adaptive controller is similar to the fixed gain controller, but adjusts the parameters of the model depending on the conditions of the environment. While this makes the model more complex and more difficult to understand, it can adapt to changing environments, making it more suitable to be used for auto-scalers that run for a long time in changing environments.

Model predictive controllers The model predictive controller predicts the future behaviour of the system, allowing it to respond to changes in the control variable before they occur. The prediction module can be created using different performance models using moving average [52], time series analysis [37], or linear regression [21].

2.3.5. Time series analysis

Time series analysis [11] is a widely used method for analysing data as a sequence of data points collected over a period of time. Given a list of periodically collected data points, future values can be predicted by identifying the underlying pattern of the data and extrapolating it to future values. As an alternative approach, future values can also be directly predicted using techniques such as moving average, exponential smoothing, and auto-regression [45].

Time series analysis is used in the analysis phase of auto-scalers to predict future workloads [2, 42, 43, 44, 31, 5]. Taking a proactive approach, future workloads or resource requirements can be estimated

and the application can be scaled beforehand. This allows the auto-scaler to deal with the delay of adding resources to the system and has therefore been recognised as an effective auto-scaling technique. Though, the accuracy of the time series analysis depends on the predictability of the pattern of the workload and may therefore not always predict sudden increases or burstiness. In addition, the accuracy of the model might reduce significantly as workload characteristics change over time.

2.4. Stream Processing

In the literature, many auto-scalers are designed for distributed stream processing systems. These systems perform operations on unbound data streams that continuously arrive at the system and require on-demand processing. They generally consist of several individual operators that pipe the output of their operator into the input of another operator. This makes the performance of the operators strongly dependent on the performance of the other operators, making operator-based scaling a complex task. In this section, we introduce the topic of stream processing, provide insight into its run-time behaviour, and discuss efficient resource provisioning.

2.4.1. The basics of stream processing

With the increase in data availability on the web, there has been a shift in focus from batch processing to online processing of data. Batch processing is the processing of bounded streams, and online processing is the processing of bounded data streams.

Data streams Data streams can be conceptualised as a list of data records that arrive at specific points in time at the system. There are two types of data streams: bounded streams and unbounded streams. A bounded stream is a finite set of data records that has a defined start and end. Due to its finite nature, the data can first be ingested by the system before performing operations on it. This allows the system to collect characteristics of the data that allow for efficient optimisations when performing operations on the data. Unbounded streams, on the other hand, are infinite sets of data that have no defined end. The data arrives as it is generated and must be continuously processed, as its infinite nature makes it impossible to wait for all input data to arrive. As we do not know what data will arrive in the future, it is difficult to perform optimise the performance operations on the data.

Processing online data streams As the data characteristics of unbound streams are generally unknown before processing, the operations are pre-defined as queries. The SPE splits these queries into several operators that all perform simple operations on the incoming data. These operators are then chained together into a topology by directing the output of an operator into the input of another operator. We visualise an example of such a topology in figure 2.2. In this figure, we have the data of a single data stream arriving at the source of the system. The source operator ingests the data and sends it to the filter operator. This operator filters the data and and sends it to the sink operator. The sink operator then outputs the data to be processed by another external system.

Figure 2.2: Example operator topology based on query 2 of the Apache Beam Nexmark Benchmark [6].

Different operators Overall, a stream processing topology may consist of many different operators. We list some of the most commonly used operators here.

- **Source**: A source is the entry of the streaming application. It retrieves data from an external application and inserts it into the stream application.
- **Sink**: A sink is the last operator in the streaming application. It receives the results of the application and outputs it to an external system.
- **Map**: A map operator is a stateless computation that executes function f on a specific operator. A map function retrieves record r, and returns f(r), in a one-to-one fashion.

- **Filter**: A filter operation is a stateless computation that, given a record r, and condition c checks whether condition c(r) holds. If it holds, it passes the record to the next operator, else it discards it.
- **KeyBy**: A key-by operation is an operation that partitions a data stream into disjoint data streams, distinguished by key k. This allows for aggregating or joining records with for which key k is the same.
- **Join**: A join operation is an operation that joins records from two data streams into a single record based on a join condition. This is a very expensive operation, as it, depending on the join type, requires storing the records of one or both streams in memory.
- **Window**: A window operation groups together different operators based on a specific window condition. The window condition is mostly based on the arrival time of an operator and defines, for example, how far apart records might have occurred to consider them for the same aggregation.
- **Aggregation**: An aggregation operation merges records into a single value.

Resource Assignment The stream processing engine consists of several different operators that all have different resource requirements. While some operators perform simple tasks, other operators perform complex tasks that require the maintenance of a large amount of state. Some operators would therefore require more resources than other operators. Operators consist of several workers that perform the operations defined by the operators. In our work, we assume workers have the same amount of resources assigned to them. The more workers assigned to an operator, the more resources it has, and the higher its throughput. The parallelism of an operator is defined as the number of workers assigned to the operator. We refer to the defined parallelism of every operator in the topology as the systems *parallelism configuration*.

2.4.2. Stream processing at run-time

We describe the run-time behaviour of an SPE from two points of view: from a global view, and a local view. The global view explains how the SPE interacts with its surrounding environment, and the local view explains how the individual operators of the stream processing engine behave and interact with its neighbour operators. While describing the run-time behaviour of SPEs, we introduce several metrics that allow us to investigate the SPE's performance.

Global view The processing of unbound data streams starts with data records arriving at the source operators of the stream application system. The amount of records that arrive per second is defined as the *input_rate* of the system. The source operator will attempt to process as much data as possible. When more data is arriving than the system can process, the data is discarded or temporarily stored in a queue of the SPE or of an external messaging system. For our work, we assume the data is stored in an external messaging system from which the source operators fetch data. When storing the data in a queue, the stream application will attempt to process both the records in the queue and the newly arriving records. The time records spent in the queue are defined as the *latency* of the system. The amount of records in the queue is defined as the system's *lag*. The amount of records that are being processed by the system, is defined as the systems *throughput*. Here, we distinguish two types of throughput, the *input-throughput* and the *output-throughput*. The input throughput is the number of records the system can input per second. The output throughput is the number of records the system outputs to its sink. In the perfect system, the system's input throughput should be equal to the input rate, with the lag being 0. The output-throughput, however, highly depends on the operators of the system and may differ from the input_throughput.

Local view Every operator in the system retrieves records from its upstream operators, executes their operation, and then outputs its results to the downstream operator. When the operator is not able to process all incoming records, the records are added to a local queue and processed later. When the queue runs out of space, the upstream operator is blocked from outputting any more records to the operator, forcing it to slow down its processing speed so that the downstream operator can keep up. This will, again, cause the upstream operator to not keep up with the incoming rate of their upstream operator, eventually slowing down the entire streaming application. The initial operator blocking its upstream

operator is called the *bottleneck operator*. The bottleneck operators experiencing *back-pressure* are said to be *back-pressured* by the bottleneck operator. The amount of time per second the operator is not able to process its incoming data due to it being back-pressured is the operator *back-pressure-time*. Besides not being able to process data due to back pressure, the operator can also be idle due to fewer records arriving at the operator than it can process. We define the operator's idle time as the time the operator spends waiting instead of processing the incoming data stream. The *idle-time* of the operator is its back-pressure-time together with the time per second the operator is waiting for input. The *busy-time* of the operator is the opposite of the idle-time, being defined as the time the operator is processing data per second (1 - idle-time).

2.4.3. Elasticity and stream processing applications

It is the job of the auto-scaler to determine the optimal parallelism of every individual operator in the systems. Here, the parallelism configuration must correspond to the system's input rate. We define four different scenarios describing the state of resources provisioning of the SPE: under-provisioning, over-provisioning, perfect-provisioning and oscillation,

Under-provisioning To be able to process all incoming data, a stream processing system should have sufficient resources available for all its operators. When not enough resources are available to the stream operator, we call the application to be *under-provisioned*. When the system is under-provisioned, the queue and the latency of the system go up and the throughput of the system will drop below the system's input rate. In addition, some of the operators will likely be back-pressured. When this happens, resources should be added to the application.

Over-provisioning When the system has more resources available, than it needs, we call the application to be over-provisioned. When the system is *over-provisioned*, the lag of the system is zero, the input throughput of the system is equal to the input rate of the system, and both the busy-time and CPU-load of the operators are low. When this happens, resources should be removed from the application to reduce costs.

Perfect-provisioning While the term perfect-provisioning is not generally used in the literature, we define perfect-provisioning as the scenario where the system is both not under-provisioning and over-provisioning. When perfect-provisioning, the system is using minimal resources, while being able to keep up with the incoming workload.

Oscillation Changing the parallelism configuration of an SPE, generally induces additional scaling overhead. This can cause the system to *oscillate* between different parallelism configurations. When, for example, the auto-scaler decides to scale down from an over-provisioning state, the resulting scaling overhead of the scale-down action might trigger a scale-up action, returning to the initial configuration. This can cause the system to oscillate between these two different parallelism configurations, inducing additional overhead and reducing the performance of the system. Introducing a *cool-down-period*, after a scaling action where scaling is not allowed, allows the system to stabilise before deciding on a scaling action again, preventing the system from oscillating.

2.5. Experimental evaluation of auto-scalers for distributed stream processing

We investigate the state-of-the-art for the evaluation of auto-scalers targeting SPEs. To do so, we survey the literature and identify five types of experiments that are commonly used when analysing the auto-scalers performance. When doing so, we find that latency is the most commonly used performance metric that, together with the resource utilisation of the system, provides a good overall insight into the effectiveness of the auto-scaler. In this section, we start by discussing the method of paper selection we use for the survey, followed by an overview of the different experimental evaluation methods used. Finally, we provide an overview of commonly used metrics when analysing the performance of auto-scalers targeting SPEs.

2.5.1. Paper selection

The papers used in the literature survey are selected in two steps. First, we select papers from the following query on Google Scholar. [4]:

`(”autoscaling” OR ”auto-scaling” OR ”auto-scaler”) AND (”stream processing” OR” ”streaming systems” OR ”streaming system” OR ”data stream” OR ”data streaming”) AND ”cloud” AND ”elastic”`

This results in a total of 42 accessible and non-duplicate papers. From these papers, we filter out unrelated papers that do not address the auto-scaling of elastic applications, resulting in a total of 27 papers. We select the papers addressing the problem of auto-scaling of SPEs and go through their related work sections in search of additional papers. This results in a total of 18 papers regarding auto-scaling for SPEs, ranging from 2014 to 2021. We survey and investigate the evaluation methods of these papers from the year 2017 and up.

2.5.2. Experiment types

From the papers, we distinguish five types of experiments that are generally performed when evaluating auto-scalers for SPEs. In this section, we discuss each type of experiment individually and summarise the evaluation methods used in the selected papers in table 2.1.

Paper	Year	Performance	Accuracy	Convergence	Prediction	Overhead
DRS [19, 20]	2017	W	Yes	No	No	Yes
Hidalgo et al. [31]	2017	W, C	Yes	No	No	No
Liu et al. [41]	2017	W, A	No	No	No	No
Lombardi et al. [43]	2017	W, C	Yes	No	Yes	Yes
Dhalion [18]	2017	W	No	Yes	No	No
Elastic-PPQ [48]	2018	W, C	No	No	No	No
Cardellini et al. [8]	2018	C	No	No	No	No
DS2 [36]	2018	W, A	Yes	Yes	No	Yes
Doan et al. [12]	2020	A	No	No	No	No
Varga et al. [66]	2021	No	No	No	No	Yes

Table 2.1: Evaluation methods used in state-of-the-art auto-scalers for Distributed Stream Processing. Under performance, W refers to performance evaluations under different workloads, C to performance evaluations under different configurations, and A to evaluations comparing different auto-scalers.

Performance analysis The performance analysis is the most commonly performed evaluation of auto-scalers. The goal of the analysis is to investigate the performance of the auto-scaler and compare its behaviour when run under different types of workload [36, 43, 48, 31, 19, 18], or with different auto-scaler configurations [43, 48, 8, 41, 31].

The general approach is for the authors to select several performance metrics, that measure the effectiveness of the auto-scaler and use these to compare its performance in different experimental runs. As a baseline, authors may choose to use a simple auto-scaler using threshold-based rules [8] or run the application without any auto-scaler at all [19]. Common metrics used for this are the latency of the system [19, 20, 41, 8], the number of resources used [43, 36, 8], and the system's throughput [31, 41, 48]. Other metrics may include the average CPU usage [31, 48], the systems downtime percentage [8], or the system's buffer usage [12].

Sometimes, the authors follow the behaviour of the auto-scaler throughout the execution of the experiment and comment on the effectiveness of its behaviour. Floratou et al. [18], for example, show through the analysis of the auto-scalers run-time behaviour that when running the auto-scaler no back-pressure can persist in the system. Fu et al. [20] analyse the performance of their auto-scaler by comparing the performance of the stream processing system with and without running the auto-scaler.

Parallelism accuracy A second way authors evaluate auto-scalers is by analysing the accuracy of the suggested parallelism of the system. Optimally, the suggested parallelism of the auto-scaler should

[4] https://scholar.google.com/

be as low as possible without under-provisioning or over-provisioning. In the literature, both Fu et al. [20] and Kalavri et al. [36] have the auto-scaler analyse the system with a constant-workload. After proposing a scaling action, the authors run the system with different parallelism configurations to inspect the behaviour of the system when run on specific configurations. This allows them to identify the optimal parallelism configuration and comment on the accuracy of the proposed parallelism configuration. When choosing the initial parallelism configuration of the system, Hidalgo et al. [31] have the auto-scaler predict the system when run with minimal parallelism, half the optimal parallelism, and the optimal parallelism, investigating the influence of the systems parallelism configuration on the auto-scalers performance. Extending this experiment to investigate the auto-scalers' prediction accuracy when over-provisioning, would provide additional insights into the parallelism accuracy when deciding to scale down. The inspected papers, however, did not consider this scenario.

Convergence time Another way to evaluate an auto-scaler is by analysing the time the auto-scaler requires to converge to a specific parallelism. We say that an auto-scaler converges when the auto-scaler stops suggesting scaling actions on a constant input rate. Both [18] and [36] perform this experiment. Floratou et al. [18] start the system at minimal parallelism and a constant input rate. Measuring the number of scaling actions the application needs to converge, the authors show that the auto-scaler eventually assigns enough resources to the system to be able to handle the incoming workload. Kalavri et al. [36] extend this experiment by also reducing the input rate after convergence to test both the convergence time of the auto-scaler when under-provisioning and when over-provisioning.

Prediction accuracy Pro-active auto-scalers predict the future behaviour of the workload to be able to react to workload changes before they occur. These systems generally make use of a prediction module that allows them to predict the future workload or future CPU usage. Lombardi et al. [43] evaluate their prediction model by comparing their predicted CPU usage with the actual CPU usage the system had at a certain point in time. This provides an indication of the accuracy of the prediction module.

Overhead estimation When deploying an auto-scaler to minimise the resources costs in a cloud environment, it is important to know the costs of the auto-scaler deployment. Costs can be induced by the auto-scaler deployment itself, or by the monitoring system that retrieves the necessary information about the auto-scaler. Fu et al. [20] investigate additional overhead when running their auto-scaler on top of an SPE. Kalavri et al. [36] investigate the overhead incurred by metric measurement. Lombardi et al. [43] analyse additional overhead caused by tweaking of the auto-scalers configurations, and Varga et al. [66] analyse the overhead caused by the scaling actions.

Other Furthermore, evaluations may be performed for the auto-scaler-specific functionalities. Lombardi et al.[43], for example, perform an additional evaluation of the auto-scaler when multiple stream processing applications are run on the same cluster.

2.5.3. Performance analysis metrics

We further investigate the different performance metrics used by the auto-scalers to evaluate their performance. These metrics indicate the auto-scalers effectiveness and can be used to compare different auto-scalers, different configurations, or the auto-scalers' performance under different workloads. We now briefly discuss the use of the different metrics. Our findings of the literature search are summarised in table 2.2.

Latency Latency (also referred to as total latency [41] or delay [48]) is one of the most used metrics and can be defined as the time it takes for a record to be processed by the system. It is calculated as the difference in the arrival time of a tuple, and the time it is processed. High latency is an indication of the under-provisioning of the system. The latency of the system cannot be used to distinguish the difference between efficient provisioning and over-provisioning, as the latency will be minimal in both cases. For this reason, latency is generally used in combination with the number of resources used or the total CPU usage, to distinguish between these two cases. Generally, papers focus on the average latency of the system [43, 41, 36]. Though some papers also consider the variance [20] or percentiles

Paper	Year	Metrics used			
		Latency	Resource efficiency	Throughput	Resource usage
DRS [19, 20]	2017	Yes	Yes	No	No
Hidalgo et al. [31]	2017	No	Yes	Yes	Yes
Liu et al. [41]	2017	Yes	Yes	Yes	No
Lombardi et al. [43]	2017	Yes	Yes	No	No
Dhalion [18]	2017	Yes	Yes	No	No
Elastic-PPQ [48]	2018	Yes	Yes	Yes	Yes
Cardellini et al. [8]	2018	Yes	Yes	No	No
DS2 [36]	2018	Yes	Yes	No	No
Doan et al. [12]	2020	No	No	No	No
Varga et al. [66]	2021	No	No	No	No

Table 2.2: Metrics used for evaluation of auto-scaler performance of state-of-the-art auto-scalers for distributed stream processing.

of the latency [8] to get an indication of the stability of the system when running the auto-scaler for longer periods. Cardellini et al. [8] define an SLO for the maximum latency and show how often the SLO is violated when running the system.

Resource efficiency As the aim of the auto-scalers is to maintain high performance with minimal resources, It is important to keep track of the number of resources used by the system when running the auto-scaler. While papers generally consider the accuracy of the number of resources suggested, the amount of resources used is often not aggregated and compared between different experiment runs. A reason for this can be that most experiments in the literature are run for a short amount of time and only require a few scaling actions. Some examples where the number of resources are aggregated and compared between runs are [8] and [43], which compare the average amount of replicas used. The amount of resources used is generally defined as the total amount of replicas in the system [8, 18, 43]. When running a topology with multiple operators this would be the sum of the parallelisms of the operators. Kalavri et al. [36] only considers the number of replicas of the 'main operator' of the system.

Throughput Throughput can also be used as an indication of the performance of the auto-scaler. Throughput is generally used as the input throughput, defined as the number of records the system accepts into its system per second. While the latency is also influenced by the lag of the system, the throughput allows for investigating the accuracy of the proposed parallelism by comparing it with the input rate. In addition, Mencagli et al. [48] use the throughput to investigate the stability of the system under specific configurations. Floratou et al.[18] define a throughput SLO and show that the suggested scaling actions by the auto-scaler eventually meet the SLO.

Resource usage While the latency and throughput can show whether the system is over-provisioning, they are unable to distinguish over-provisioning and perfect-provisioning from each other. Further investigating the resource usage allows us to make this distinguishment. When over-provisioning, resources will be idle for a significant amount of time. To measure the current resource usage, Mencagli et al. [48] use idle time as a percentage of the time the system is waiting for incoming records. Hidalgo et al. [31] use both the CPU usage and memory usage. Other papers do use the resource usage in their system when deciding whether to scale down [43, 66], but do not consider it in the evaluation of the system.

Other Besides these three metrics, papers do use other metrics to measure the performance of the system. Kalavri et al. [36] and Floratou et al. [18], for example, measure the number of steps it takes for their auto-scaler to converge to the optimal parallelism. Doan et al. [12] use the total amount of buffer usage to determine whether the auto-scaler is over-provisioning or not. Cardellini et al. [8] also consider the system's downtime caused by scaling actions proposed by the auto-scaler. Also, more auto-scaler-specific metrics are used. Doan et al. [12], for example, use the rewards generated by

the reward function of their reinforcement model to get an estimation of both the performance of the auto-scaler and the training time required to achieve high accuracy.

2.5.4. Conclusion

The recent developments around cloud computing have opened up the possibility for the elastic provisioning of distributed applications. Auto-scalers are systems that, given metrics from an external monitoring system, automatically decide whether to scale the application and by how much. They are designed for many types of applications and are based on five different techniques, which are threshold-based rules, reinforcement learning, queuing theory, control theory, and time series analysis [55]. Over the past several years, many auto-scalers are developed for distributed stream processing systems (SPEs). SPEs process complex queries by dividing them into smaller operators that are piped into each other creating an operator topology. When scaling the system, auto-scalers have to consider the relationships between these operators, making effective scaling of the application a difficult job. Furthermore, the lack of proper evaluation of the performance of these auto-scalers has made it difficult to determine the state-of-the-art of auto-scalers targeting SPEs and has slowed down its development. In this section, the evaluation methods of state-of-the-art auto-scalers targeting SPEs are evaluated and the results show that auto-scalers are often poorly evaluated with simple experiments based on unrealistic scenarios. Therefore, five types of experiments are distinguished that are used to analyse the performance of auto-scalers for SPEs showing that performance analysis is the most common one. When analysing the performance of these auto-scalers, latency and throughput are often used as performance metrics.

3 Related work

Throughout the literature, many proposals for auto-scalers targeting distributed stream processing applications have been made. One of the first auto-scalers specifically designed for distributed stream processing engines (SPEs) that support state-full operations was proposed by Gedik et al. [23]. Their solution makes use of a simple control algorithm deployed on local hosts that execute scaling actions when backpressure is observed. Heinze et al. [27] attempt to optimise the utilisation of the system while maintaining low latency. They show that an auto-scaler using threshold-based rules can result in good performance when run on individual hosts. Still, when running on the entire system, it results in low performance due to its simplistic assumptions. Because of the dependency of the threshold-based approach on its initial configuration, the authors show that reinforcement learning can result in high performance while minimising the initial configuration costs. This problem is further addressed by Heinze et al. [28], where the authors propose an online parameter optimisation technique that detects changes in the workload pattern and adapts the scaling policy accordingly.

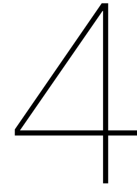
Lohrmann et al. [42] propose a predictive latency model that is based on queuing theory. The module allows the system to investigate the expected latency given a specific parallelism for each operator. Using this model, the system can estimate the optimal parallelism for the topology by comparing different configurations. Xu et al. [67] introduce Stela as a stream processing engine that supports elastic scaling with minimal scaling overhead, while optimising the post-scaling throughput by smartly assigning the workers to specific operators. While they support scaling operators, they do not implement any scaling policy. Hochreiner et al. [32] proposes a distributed stream processing engine that supports elastic scaling, and that allows for scaling the system at run-time. Using this implementation, they introduce a simple auto-scaler that uses threshold-based rules to test the scaling capabilities of the system.

Floratou et al. [18] propose a framework to create self-regulating streaming systems that are self-tuning, self-stabilizing, and self-healing. The authors propose two scaling policies based on the back-pressure status of the system. Hidalgo et al. [31] propose a self-adaptive processing graph that divides the workload of overloaded operators over multiple replicas. Using a control algorithm they can scale the topology both reactive and proactive, improving the performance and resource efficiency of the system. Liu et al. [41] develop a profiling model as a feedback-control loop that learns the relationship between the provisioned resources and the application's performance and scales the application accordingly. Lombardi et al. [43] propose ELYSIUM as an auto-scaler that optimises resource consumption by both considering horizontal scaling and vertical scaling. They investigate the trade-off between both scaling actions and combine them into an auto-scaler. Fu et al. propose DRS [19, 20] as a queuing-theory-based auto-scaler that models the relationship between the provisioned resources and the application performance. Just as [42], they compare the expected performance of different resource allocations and choose the most optimal one. Only, instead of modelling each operator as an individual queuing system, they model the entire topology as a generalised Jackson network, allowing for more accurate performance estimations.

Cardellini et al. [8] propose an auto-scaler for stream processing in a decentralised environment. They propose a two-layered hierarchical structure that allows for autonomous control over the individual operators in the system. Their solution contains two reinforcement-based learning approaches that are evaluated against a simple threshold-based approach.

Kalavri et al. [36] propose DS2 as a control-based auto-scaler for distributed stream processing. Introducing the notion of true processing and true output rate the authors propose an auto-scaler that can estimate the optimal parallelism for every operator in a single iteration. The authors show that this approach outperforms state-of-the-art system Dhalion [18] requires multiple scaling actions to reach the optimal configuration, while DS2 only requires 1. Mencagli et al. [48] propose Elastic-ppq as a system that can process spatial preference queries over dynamic data streams. Combining Elastic-ppq with an auto-scaler, the authors can scale their system to the current workload, minimising resource usage.

Lombardi et al. [44] propose PASCAL as a general-purpose auto-scaler architecture that makes use of reinforcement learning. It combines a proactive approach to forecasting incoming workloads with a profiling system for estimating the optimal provisioning. The authors implement PASCAL for an SPE (Apache Storm) and a distributed datastore (Apache Cassandra) and show its effectiveness for both use cases. Doan et al. [12] propose using fuzzy deep reinforcement learning for auto-scaling streaming architectures. Here, they estimate the initial Q-values of the reinforcement system by use of a deep neural network. This significantly shortens the training time of the model and increases its performance. Still, the authors note that it is a non-trivial task to tune the parameters of the reward function of reinforcement learning systems. Varga et al. [66] propose two custom metrics that can be used in combination with Kubernetes' out-of-the-box auto-scaler HPA [38] for the scaling of SPEs. In addition, they perform an experimental evaluation of the role of state size on the scaling overhead of Apache Flink. Arkian et al. [4] propose Gesscale as an auto-scaling for distributed stream processing in geo-distributed environments. Using a performance model, the system decides when to scale the application and on which geo-distributed servers to add resources to optimise the maximal sustainable throughput of the system.



Auto-scalers for Stream Processing

In this work, a benchmarking framework that allows for the performance analysis of auto-scalers for SPEs has been designed and implemented. Using this framework, the performance of several state-of-the-art auto-scalers are evaluated and compared. In the section below, the process of selecting these systems and the underlying theory of the auto-scalers will be discussed in more detail.

4.1. Paper selection

In section 2.5, an analysis of the literature was performed and several auto-scalers targeting SPEs were distinguished. From these auto-scalers, we excluded auto-scalers that are designed for application-specific scenarios, such as decentralised environment [8], and geo-distributed servers [4]. Furthermore, we excluded PASCAL [44], and the auto-scaler of Doan et al. [12], as they require extensive training of the auto-scaler before they can be deployed. This leaves us with DRS [20], Dhalion [18], DS2 [36], and Varga et al. [66]. DRS relies on a queue-theory-based prediction module that is difficult to implement for different queries. For this reason, we do not consider DRS in the performance evaluation of this paper but do suggest implementing the auto-scaler in future works. Furthermore, we use general-purpose auto-scaler HPA as a baseline in the experiment.

4.2. Dhalion

Dhalion [18] is a general-purpose framework for self-regulating stream processing applications. It is designed as a general-purpose system that provides self-regulatory capabilities to underlying stream processing systems. While implementing some of these capabilities themselves, it is up to the user to extend the logic of Dhalion to fit their specific use case. In this section, we discuss the overall architecture of Dhalion, followed by the authors' auto-scaler design.

The framework Dhalion's framework consists of three layers: a symptom detection layer, a diagnosis generation layer, and a resolution layer (figure 4.1). The detection layer is responsible for monitoring the application and identifying symptoms that may indicate the system to be in an unhealthy state (like back-pressure detection, or data skew). The layer receives metrics, analyses them to identify symptoms, and passes the found symptoms to the diagnosis generation layer. Some examples of symptoms are backpressure, data skew, or lag. After receiving the symptoms, the diagnosis generation layer investigates them and determines what could be the cause of these symptoms (like under-provisioning, over-provisioning, data-skew, and slow operators). The diagnoses are then passed to the resolution layer which takes appropriate action given the diagnosis (e.g. scaling, re-partitioning, restarting a node).

Auto-scaler In the paper, the authors design and implement a reactive auto-scaler that is based on threshold-based rules as an example application of their framework. We consider their proposed auto-scaler in our experimental evaluation. Every five minutes, the system analyses the state of the application, collecting metrics which are passed to the symptom generation layer. The symptom gen-

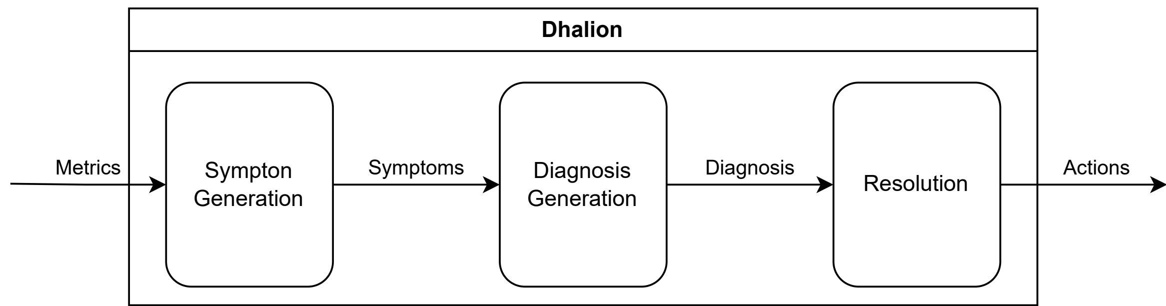


Figure 4.1: The three-layered framework of auto-scaler Dhalion [18].

eration layer analyses the metrics and determines whether there is back-pressure or lag. Given the symptoms, the diagnosis generation layer investigates whether the symptoms can be explained by over-provisioning or under-provisioning. If it is back-pressure, the bottleneck operator of the system is said to be under-provisioned. If there is no back-pressure, every operator without lag is said to be over-provisioned. This information is then passed to the resolution layer, which issues an appropriate scaling action to the under-and over-provisioning operators accordingly.

If an operator is under-provisioned, the auto-scaler determines a scale-up factor to scale the operator up with. This scale-up-factor is determined in the following way:

$$scale_up_factor = 1 + \frac{amount_of_records_suspended}{amount_of_records_processed} \tag{4.1}$$

The scale-up factor denotes the amount of data the system was not able to handle. For example, if the system suspends 20% of the incoming records and can process 80% of the incoming records, then the scale_factor would be $1 + 20/80 = 1.25$. Given that the operator has a current parallelism of 8, the new suggested parallelism would be $8 * 1.25 = 10$.

If an operator is said to be over-provisioned, Dhalion scales the operator down with a pre-defined scale-down factor. This factor is set manually by the user. The authors use a scale-down factor of 0.8 in their experiments. An over-provisioning operator with parallelism 20, would in that case be scaled to a parallelism of $20 * 0.8 = 16$.

4.3. DS2

DS2 [36] is a reactive *control theory* based auto-scaler for distributed stream processing operators. The goal of the auto-scaler is to scale the target application to satisfy the SASO properties: stability (no oscillation), accuracy (accurately finding optimal parallelism), short-settling times, and not overshooting. It addresses the problem that state-of-the-art auto-scalers for stream processors, like Dhalion, require multiple scaling actions before they converge to the optimal parallelism. This introduces additional scaling overhead and takes significant time before the system can handle the changed workload, violating SLOs or introducing extra costs due to over-provisioning. To combat this issue, DS2 contains a comprehensive performance model that can estimate the parallelism for all operators within a single scaling decision. This is done with the use of two custom metrics: useful time, and true rate.

Useful time Useful time is defined as the time spent by an operator instance in deserialization, processing, and serialization activities. This is the time the operator is processing the stream and excludes time spent awaiting input or output.

True rate The true processing rate is defined as the number of records in which operators process records per unit of useful time. The operator uses two different types of true rates: true input rates and true output rates. True input rate is calculated as the input throughput of the operator, divided by the useful time. The true output rate is calculated as the output throughput of the operator, divided by the useful time. Both metrics represent the input/output of the operator when it is run at a full 100% capacity.

Scaling factor Using these metrics, a per-operator scaling factor is calculated in the following way. Given an operator, we sum the true output rates of upstream operators that directly feed into the operator and call this the aggregated output rate of the upstream operators. The scale factor of the operator is now defined as:

$$scale_factor = \frac{aggregated_output_rate_{upstream_operators}}{true_processing_rate_{operator}} \quad (4.2)$$

For the source operators, the *input-rate* of the system is used as the true output rate. Using the formula, the scale factor for every operator in the system is calculated in a single traversal of the topology. The result is the optimal parallelism configuration for the entire topology.

4.4. HPA

Horizontal Pod Autoscaler (HPA) [38] is the built-in general-purpose autoscaler of Kubernetes. It runs as a simple loop that monitors a control variable (e.g. CPU- or memory usage) and attempts to keep the control variable close to a specified target variable by scaling the application. Scaling is done horizontally (hence the name) by increasing or decreasing the number of replicas in the cluster.

Given a resource metric to monitor and a target value for the resource metric, HPA periodically fetches the current value of the resource metric. Using this value, it determines the desired parallelism of the system with the following formula:

$$desired_replicas = \lceil current_parallelism * \frac{current_metric_value}{target_metric_value} \rceil \quad (4.3)$$

The resulting desired-replicas value is saved in a list containing all previous results. HPA then takes the highest desired-replicas value predicted in the previous n minutes (5 minutes by default) and scales the application to this amount of replicas. This so-called, stabilization window prevents the system from oscillating between desired configurations as a result of scaling costs.

HPA allows the monitoring of multiple control variables. When doing so, it calculates the desired amount of replicas for both variables individually and chooses the highest desired amount of replicas. Because HPA is only able to change the number of replicas in the system, it is not able to change the individual parallelism of an operator in the topology. To apply HPA to stream processing, we can have HPA scale the number of workers present in the system and have the stream processing engine detect changes in the parallelism of workers and spread the workload equally over these workers. An example of a system that allows doing so is Flink Reactive [49]. For the experimental evaluation, we let HPA keep track of the average CPU usage of the workers in the topology. From this point on, we refer to HPA targeting the workers' CPU usage as HPA-CPU.

4.5. HPA-Varga

Varga et al. [66] propose an architecture extending the HPA autoscaler for auto-scaling of distributed stream processing [66]. The authors propose two different metrics to be used as resource metrics for HPA: relative-lag-change and utilisation. Because HPA can only scale the number of available workers for the entire system, the metrics are measured over the entire system.

Utilisation Utilisation is a metric providing additional insight into the system's performance. It allows the auto-scaler to distinguish over-provisioning from perfect-provisioning, by analysing how much of the available resources are used for processing. Utilisation makes use of the *idle-time-per-second* metric. The utilisation of the system is calculated using the following formula:

$$utilisation = 1 - avg(idle_time_per_second) \quad (4.4)$$

Utilisation is a value between 0 and 1. When setting the target value close to one, the auto-scaler suggests faster scale-down actions to get higher utilisation. This results in lower costs but might result in under-provisioning. When lowering the target value, the auto-scaler will suggest scale-down actions less often, possibly resulting in over-provisioning.

Relative lag change rate Relative lag change is a metric that provides an estimation of the size of the workload the system is not able to handle. It makes use of the derivative of the system's lag and the application's input throughput. The relative lag change rate is calculated using the following formula:

$$relative_lag_change_rate = 1 + \frac{deriv(total_lag)}{input_throughput} \tag{4.5}$$

The relative-lag-change rate denotes the rate at which the workload is increasing (> 1) or decreasing (< 1). When equal to 1, the lag is not changing and the system can handle the workload. For this reason, the target value of the relative lag change rate is set at 1.0. Because HPA only considers the metric requesting the highest parallelism, the authors propose ignoring the relative lag change rate when the lag is below a specified threshold. This allows the auto-scaler to consider the suggested parallelism of the utilisation metric, supporting scale-down actions when over-provisioning.

4.6. Conclusion

For the experimental evaluation of auto-scalers targeting SPEs, we select auto-scalers DS2 [36], Dhalion [18], and Varga et al. [66] to be evaluated in the experimental evaluation using Kubernetes built-in general-purpose horizontal pod auto-scaler (HPA) [38] as baseline. Dhalion makes use of threshold-based rules and issues scaling actions based on the back-pressure status of the system. DS2 is a control theory based auto-scaler that estimates the optimal parallelism configuration of the optimal system in a single iteration by using the true input and output rate of the operators. Varga et al. [66] propose two application-specific metrics named utilisation and relative-lag-change to be used as control-variables in HPA. Furthermore, as baseline HPA with CPU-utilisation as target value was used.

5 Experimental design

The aim of this work is to investigate how state-of-the-art auto-scalers for SPEs perform in different real-world scenarios and to evaluate their performance. In this section, we discuss the design of the experiments and comment on the different design choices we make. We start by discussing the different experiments we integrate into the framework and the performance metrics we use evaluating the performance of the auto-scalers. Next, we comment on the different technologies we use for the implementation of the framework.

5.1. Experiment requirements

For our experimental evaluation, we create a framework that supports end-to-end evaluation of auto-scalers with different workloads on different operator topologies. The framework is easy to extend with new scenarios and new auto-scalers can easily be added. For the design of this framework, we take inspiration from the current state-of-the-art for the evaluation of auto-scalers targeting SPEs, as described in section 2.5.2. We differ in two ways from current state-of-the-art methods for auto-scaler comparisons: the auto-scalers are evaluated end-to-end and the framework supports the comparison of different auto-scalers.

End-to-end evaluation State-of-the-art auto-scalers targeting SPEs are generally evaluated with shallow experiments that are run for a small amount of time with unrealistic changes in input rate that only require one or two scaling actions. Kalavri et al. [36], for example, evaluate their operators' performance on constant input rate with different initial parallelism configurations and Floratou et al. [18] investigate a single topology with two input-rate changes. While this provides a controlled environment for the authors to reason about the accuracy of the auto-scaler, it does not show the performance of the auto-scaler in any production-like environment, where the auto-scaler is run for a long amount of time and has to deal with different complex input-rate patterns. For our experimental evaluation, we focus on these more production-like environments and investigate the auto-scalers when run for a significantly longer period. This way we can see how the auto-scaler deals with scaling overhead, reason over the timing of certain scaling actions, and how the auto-scaler reacts to different workload patterns when run with different topologies.

Auto-scaler comparison Another limitation of state-of-the-art evaluation of auto-scalers for SPEs is the general absence of a comparison between auto-scalers. One reason for this is the absence of a general testing framework in which auto-scalers can be implemented and compared. Currently, authors are required to build both the testing environment and other state-of-the-art auto-scalers from scratch, requiring much additional work. Kalavri et al. [36] design an auto-scaler for several SPEs, allowing it to reuse the testing framework of Dhalion [18] and compare its performance with it. This evaluation framework, however, is very limited. For our experimental evaluation, we design a modular framework that can easily be extended with more auto-scalers and that can compare their performance. This

brings about two challenges: how to compare different auto-scalers effectively, and how to create a framework that can be extended with more auto-scalers.

5.2. Experimental types

As discussed in the previous section, we design a system that supports end-to-end evaluation of auto-scaler and allows for the comparison of the performance of different auto-scalers. As shown in section 2.5.2, we distinguish five different experiments for evaluating auto-scalers for SPEs: performance analysis, parallelism accuracy, convergence time, prediction accuracy, and overhead analysis. As the main goal is to create a framework that allows for the comparison of different auto-scalers we primarily focus on the performance analysis experiment. We later extend this to also include the possibility to reason over the parallelism accuracy of the system and to compare the convergence time of different auto-scalers. We do not consider the prediction accuracy experiment, because of its auto-scaler-dependent nature, and consider the overhead analysis to be out-of-scope for this work. Furthermore, as we discuss in section 6, the auto-scalers make use of the same monitoring system and only differ in their auto-scaler logic. As the auto-scalers are deployed as individual containers, we consider the difference in overhead negligible.

Performance analysis The goal of the evaluation framework is to gather insights into the overall performance of the auto-scaler under different workloads when run on different operator topologies. The performance of the auto-scaler is then compared to the performance of other auto-scalers or to itself when run with different configurations. As discussed in section 2.5.2, the performance analysis experiment is done by selecting several metrics that are collected during run-time and used to evaluate the performance of the system. For these metrics, we focus on the two most common performance metrics used in the performance analysis for auto-scalers targeting SPEs: latency and throughput. In addition, we investigate the efficiency of the resource usage of the auto-scaler and investigate the scaling overhead induced by the auto-scalers. We discuss the performance metrics in more detail in section 5.3. The auto-scalers are run for a configurable amount of time on different operator topologies with a diverse number of workloads. These workloads include a sinus pattern, an increasing pattern, a decreasing pattern and a randomly generated pattern (more information in section 5.4.3).

Convergence time While the main focus of the experiment is the performance analysis of auto-scalers, we add an additional experiment similar to the one of [18] and [36] investigating the convergence time of the auto-scalers. For this, we use the convergence workload, as described in section 5.4.3. By starting at minimal parallelism, we let the auto-scalers converge to a constant input rate. After the auto-scaler converges, the input rate is decreased, requiring the auto-scaler to scale down. This combination allows us to investigate the convergence time of the auto-scalers when scaling up and scaling down. Adapting fast to a change in the system's workload reduces the time the system spends in a sub-optimal state, minimising latency. Furthermore, requiring more scaling actions to adapt to a new workload results in additional scaling overhead.

Parallelism Accuracy Ideally, an auto-scaler should suggest a parallelism configuration that can handle the incoming workload while minimising the resources. To investigate the accuracy of the suggested parallelisms of auto-scalers, both Kalavri et al. [36] and Fu et al. [20] run the systems on different parallelism settings and determine which parallelism is the most optimal one. Then, by comparing the suggested parallelism of the auto-scaler and the actual optimal parallelism, they can determine the accuracy of the auto-scaler. Though, when running the auto-scaler in a production-like environment, the suggested parallelism should also consider scaling overhead and workload variations. This makes determining the optimal parallelism a bit more complex. While, for example, a scaling decision that results in over-provisioning can be considered inaccurate, when after scaling the workload of the system suddenly goes up, this can again be considered to be an accurate decision. For this reason, when running the auto-scalers in a production-like environment, it is difficult, or even impossible to determine the optimal parallelism at a certain time. For this reason, we do not perform a parallelism accuracy analysis as DS2 [36] and Dhalion [20]. However, we can get an estimation of the overall performance and resource usage of the auto-scalers on the entire run. By comparing the performance and the resource usage of different auto-scalers with each other, we can get an indication of the efficiency, and

with that the accuracy of the system's predictions.

5.3. Performance metrics

For the comparison of different auto-scalers, we select several metrics that provide insight into the overall performance of the auto-scaler. As discussed before, for this we focus on the two most commonly used metrics for the evaluation of auto-scalers targeting SPEs: latency and throughput. In addition, to get an estimation of the efficiency of resource usage of the system, we investigate the number of resources used by the system and the number of scaling actions it triggered. Finally, to investigate the convergence time of the system, we investigate the system's convergence time while running the system on the convergence workload (section 5.4.3.

Latency As shown in section 5.3, latency is the most used metric used for evaluating auto-scalers targeting SPEs. For this, authors generally use the overall latency of the system [41, 20]. The overall latency is the difference in time between the record arriving at the SPe and when the SPE is done processing the record. This is, however, difficult to measure and is affected by the topology operators of the system. Instead, we define latency as the difference in time between the records arriving at the SPE and the time it is received by the SPE. When using a messaging system as middleware, this would be the time the records spend inside the messaging system before being emitted by the SPE. We follow Cardellini et al. [8] in their approach by aggregating the latency and investigating the average latency, the 50th percentile latency and the 95th percentile latency. In addition, we also look at the maximum latency of the system.

Throughput For throughput, both the input-throughput and output-throughput can be used. The input throughput is the number of records that enter the SPE and the output throughput is the number of records per second that leave the SPE. As the output rate depends on the selectivity of the system's operators, we choose the input rate as the metric we look at for the throughput. By comparing the input throughput with the input rate, we can see whether the system can keep up with the input rate.

Resources efficiency We also want to get an estimation of the efficiency of the auto-scalers in terms of resource consumption. For this, we consider both the average and maximum amount of workers deployed at any point in time. This is equal to the sum of the parallelism of the operators.

Convergence time Finally, as we also investigate the convergence time of the system, we want to get an estimation for this. For this, we consider the total amount of time it takes for the auto-scaler to converge when using the convergence workload (section 5.4.3). In addition, we also measure the total number of scaling actions required before converging to a parallelism configuration.

Resource usage As throughput cannot distinguish over-provisioning from perfect-provisioning, we also consider the resource usage. The higher the resource usage, the more efficient the resources are used. This allows us to distinguish over-provisioning from both under-provisioning and perfect-provisioning. When the throughput is equal to the input-rate and the resource usage is high, the system is perfect-provisioning. For the experiments, we use the average CPU-usage as metric for the resource usage.

Number of scaling actions Another metric, not generally considered in the performance analysis of auto-scalers for SPEs, is the number of scaling actions issued by the auto-scaler. Depending on the topology of the system, scaling the application can induce a significant overhead. This is especially the case with stateful operators, who need to store their state before being able to scale the application. Because of this, the number of scaling actions performed by the application can have a significant impact on the performance of the system. For this reason, we also consider the total number of scaling actions performed during the experiment.

5.4. End-to-end Auto-scaler evaluation

For evaluating the auto-scalers, we create a modular framework that can test the auto-scalers end-to-end in a production-like environment. As a basis for this framework, we select the Apache Beam Nexmark Benchmarking framework [64, 6]. This is a benchmarking framework specifically designed for benchmarking SPEs and is used in the evaluation of state-of-the-art auto-scaler DS2 [36]. The framework consists of a large number of stream-processing queries, which are implemented together with a data generator. We select several queries from the benchmarking framework and extend the data generator to generate data in different syntactic patterns. In this section, we start by providing some background information for the Nexmark benchmarking framework. Next, we discuss the different queries and the different workloads we use for the experimental evaluation.

5.4.1. Bench-marking framework

For the experimental design, we consider the Nexmark Benchmark suite for queries over continuous data streams [64]. This framework is later extended with more complex queries by the Apache Beam as the Apache Beam Nexmark suite [6]. The Apache Beam Nexmark suite is state-of-the-art in terms of benchmarking queries over data streams and is used for the evaluation of state-of-the-art auto-scaler DS2 [36]. The framwork is implemented[1] for state-of-the-art SPE Apache Flink [15].

The Nexmark benchmark suite concerns an EBay-like online auction store, containing person-, auction-, and bid objects that arrive at the system as three separate streams. Persons are newly registered users with an ID, name, email address, credit card number, city, and state field. Auctions are auctions hosted by players containing an id, name, description, initial bid, sell price, timestamp, expiration date, the sellers-ID, and a category. A bid holds the auction ID it corresponds to, a Person-ID of the bidder, a price, and a timestamp. The benchmark contains a wide variety of queries having different processing weights and different state sizes. We select a number of these queries for our experiments.

5.4.2. Queries

From the Apache Beam Nexmark suite, we select queries 1, 2, 3, 5, 8, and 11 based on their processing requirements and different state sizes. These queries are already used in the evaluation of state-of-the-art auto-scaler DS2 [36] and contain a wide variety of stream processing complexities.

Query 1 Query 1 is a simple query that contains a simple 1-to-1 mapping function. The query accepts the Bids stream and converts the bid value from dollars to euros. This query tests the processing speed of the system and acts as a reference point for the other queries. The corresponding topology of the query is visualised in figure 5.1.

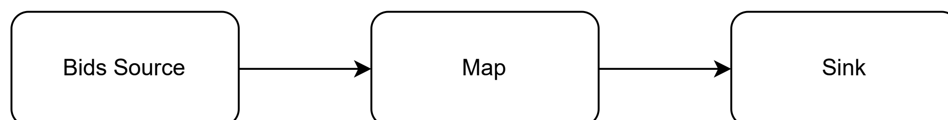


Figure 5.1: Operator topology of query 1 of the Apache Beam Nexmark Benchmark [6].

Query 2 Query 2 is also a simple query that selects the bids that correspond to specific auction IDs. The query filters out the bids with different auction IDs and prints out the item ID and the corresponding bid. The topology of the query is visualised in figure 5.2.

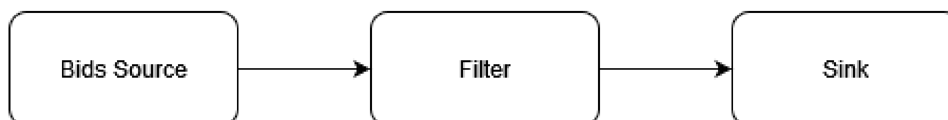


Figure 5.2: Operator topology of query 2 of the Apache Beam Nexmark Benchmark [6].

[1] `https://github.com/nexmark/nexmark`

Query 3 Query 3 is a more complex query, that tests the join functionality of the system. It filters out persons from a specific state and joins them with the auctions that they are hosting. As join operators require saving the data that has previously been seen by the system for future join operations, the state of the query grows large. This results in high processing times and large scaling overhead. The topology of the query is visualised in figure 5.3.

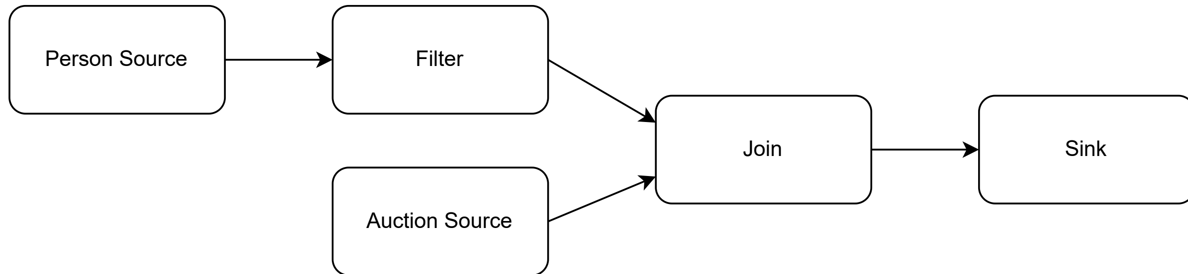


Figure 5.3: Operator topology of query 3 of the Apache Beam Nexmark Benchmark [6].

Query 5 Query 5 is a sliding window aggregation that selects the auction with the highest number of bids placed in the previous period. The sliding window size of the query is set to 60 minutes with a 1-minute interval. As the query keeps track of a counter for every unique auction ID it finds, the state of the query can grow significantly. However, when using hash-partitioning without data-skew, the processing time of the query is, in opposition to query 3, linear to the input rate. The topology of the query is visualised in query 5.4.

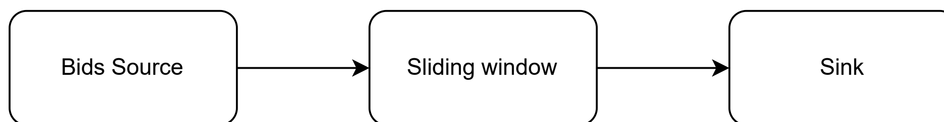


Figure 5.4: Operator topology of query 5 of the Apache Beam Nexmark Benchmark [6].

Query 8 Query 8 concerns a window join of the person and the auction stream. The goal of query 8 is to select the persons that have joined and created auctions in the last period of 10 seconds. Due to the small window size, the state of the operation will not grow too big as in query 3. The topology of query 8 is shown in figure 5.5.

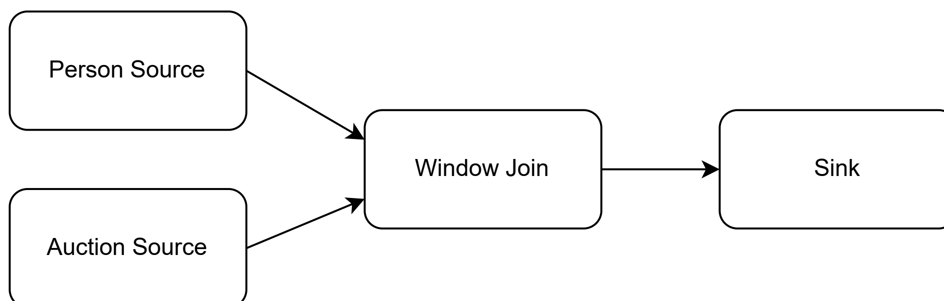


Figure 5.5: Operator topology of query 8 of the Apache Beam Nexmark Benchmark [6].

Query 11 Query 11 is the only query of our selected six that is introduced by the Apache Beam Nexmark benchmark suite. The query concerns a session window that counts the number of bids users make during the session. While previous queries had windows of constant size, the session window may take a variable amount of time, introducing additional challenges for the SPE. The topology of query 11 is shown in figure 5.6.

Figure 5.6: Operator topology of query 11 of the Apache Beam Nexmark Benchmark [6].

5.4.3. Workloads

For our end-to-end evaluation of state-of-the-art auto-scalers, we investigate their performance under different circumstances. For this, we adapt the Apache Beam Nexmark workbench to generate syntactic workloads. We use the following workloads for the experimental evaluation: cosine, random, increasing, decreasing, and convergence workload. All but the convergence workload are run over a total of 140 minutes. The convergence workload is run for a total of 90 minutes. In this section, we describe the different workload patterns and discuss their relevance for the experimental evaluation.

Cosine workload The cosine workload follows a cosine pattern, as shown in figure 5.7. The cosine workload allows for testing both the scaling up and scaling down capabilities of the auto-scalers. To mimic a more real-world scenario, we introduce a bit of random noise in the input rate of every minute.

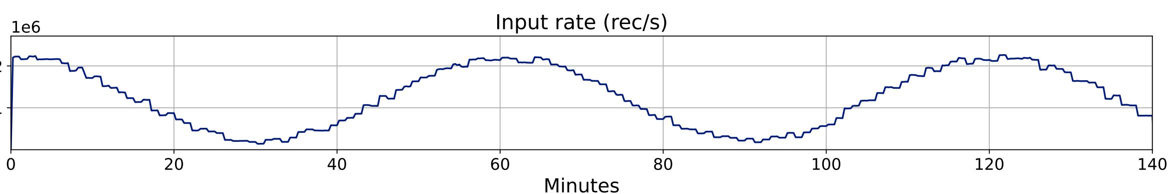


Figure 5.7: Workload following the cosine pattern with a minimum input rate of 250.000 records per second, a maximum input rate of 2.200.000 records per second, a maximum of 100.000 random noise and a cosine period of 60 minutes.

Random workload To mimic more real-world scenarios, we introduce the random workload. The random workload starts at a specific input rate and adds or subtracts a random number to the input rate of the system. By not following a predefined pattern, we make it more difficult for the auto-scalers to anticipate changes in the input rate, potentially uncovering unwanted behaviour programmed into the auto-scalers.

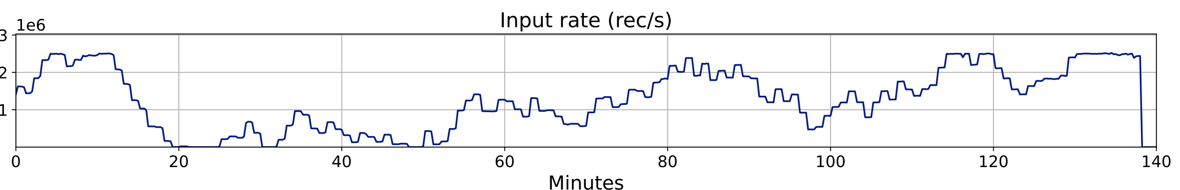


Figure 5.8: Workload following the random pattern with a maximum change in input-rate of 500.000 records per second. The maximum input-rate is set at 2.5 million records per second.

Increasing workload The increasing workload includes a simple workload that starts at a specific value and then randomly increases over time. In the experiments, we start at zero input rate and set the initial parallelism of the system to its minimum. This removes the need for the auto-scalers to converge to the proper parallelism to get up to track with the initial input rate.

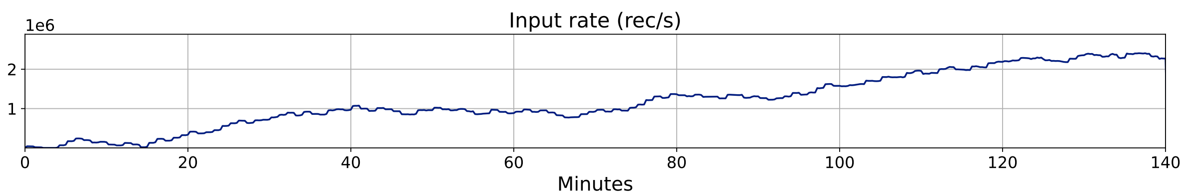


Figure 5.9: Workload following the increasing pattern starting at zero records per second, randomly increasing over 140 minutes until it reaches a maximum of 2.5 million records per second.

Decreasing workload The decreasing workload is the opposite of the random workload. It starts at a specific workload and then randomly decreases towards zero. Because of the high initial workload, we try to guess the optimal parallelism of the workload to reduce the time it takes for the auto-scaler to converge to the initial perfect parallelism.

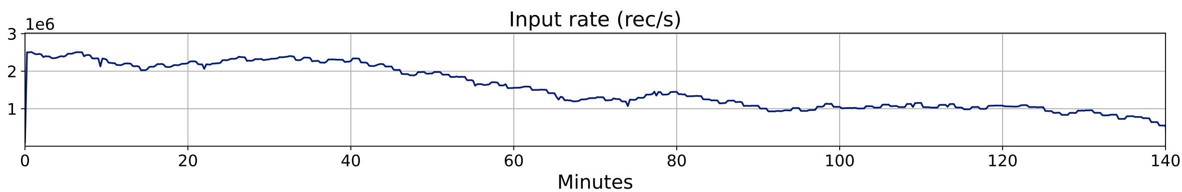


Figure 5.10: Workload following the decreasing pattern starting at 2.5 million records per second, randomly decreasing over 140 minutes until it reaches close to zero records per second.

Convergence workload To investigate the time it takes for auto-scalers to converge to the optimal parallelism configuration, we introduce a convergence workload inspired by Kalavri et al. [36]. The convergence workload starts at an input rate of 0 records per second with the operators having minimal parallelism. Then, the input rate is increased to a higher input rate and kept constant until the auto-scaler converges. After this, the input rate is lowered to a lower input rate which requires the system to scale down again. This allows us to investigate the time the auto-scalers need for scaling up and down.

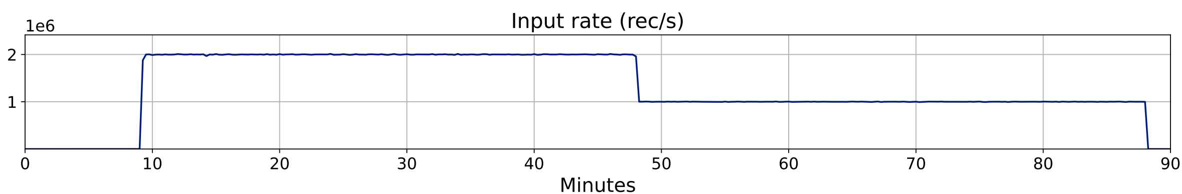


Figure 5.11: Workload following the convergence pattern, starting at 0 records per second, increasing to 2 million records per second, decreasing to 1 million records per second.

5.5. Stream Processing Engine

For the experimental evaluation, we use Apache Flink [15] as SPE. Apache Flink is an open-source stream-processing and batch-processing framework that can be deployed in distributed environments. It is used in the experimental analysis of both DS2 [36] and Varga et al. [66], and is widely considered to be state-of-the-art. In this section, we discuss Apache Flink and its underlying technologies. We start by discussing its architecture, followed by its supported operators, scaling functionality, and its backpressure mechanism. Finally, we discuss the metrics that are made available through its built-in monitoring system.

5.5.1. Architecture

A simple Apache Flink deployment [14] (referred to as a job) consists of three parts: the Job-manager and Task-managers. The job-manager coordinates the job and assigns tasks to the task-managers.

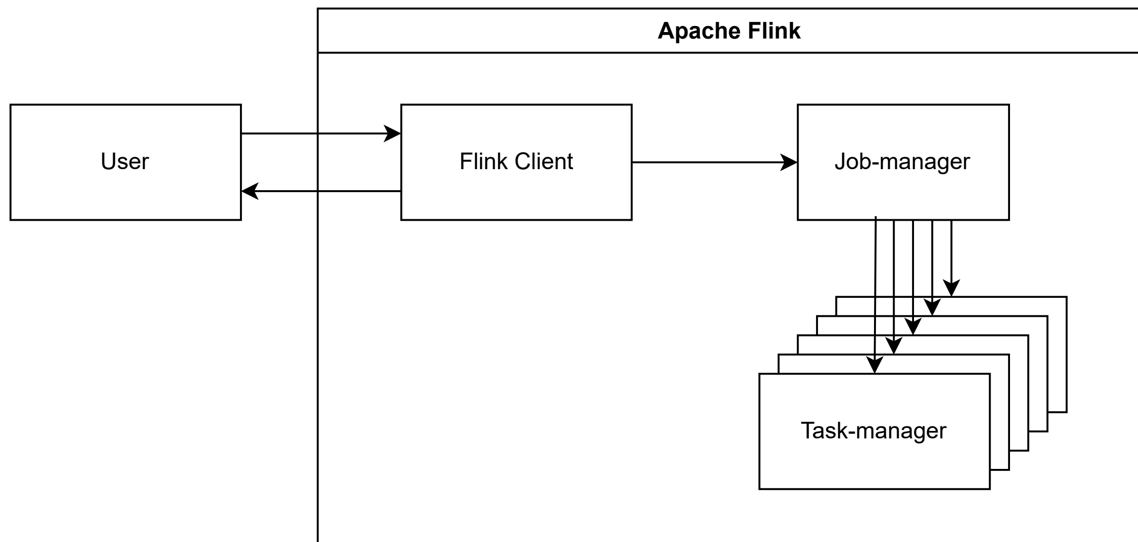


Figure 5.12: Visualisation of the architecture of Apache Flink [17].

The task-managers are the workers of the system and execute tasks assigned to them by the job manager. The user interacts with the job-manager through the Flink client. The Flink client compiles the job submitted by the user and sends it to the job-manager to be deployed. Flink's architecture is visualised in figure 5.12.

Jobmanager The job-manager is the coordinator of the job. It receives a job description, which includes the dataflow graph (operator topology), parallelism and other configurations from the user and creates an execution graph for the task-managers. In this execution graph, every task-manager in the cluster is assigned a task, retrieving data from other task-managers, processing it and forwarding it to the next task-managers. The task of the job-manager is to supervise the execution of the job, manage failure and issue checkpoints.

Taskmanagers A task-manager is a worker in Apache Flink, each having a set amount of resources assigned to it in terms of CPU and memory. A task-manager gets assigned tasks from the job-manager, telling it what data to accept, how to process it and where to forward it to. A task-manager can perform one or more tasks at the same time, sharing resources between the different tasks. When performing the tasks, the task-manager will occasionally be instructed to make a checkpoint, backing up all the data it is currently processing. In case of a failure, the system rolls back to the last checkpoint, ensuring exactly-once state consistency and fault tolerance.

5.5.2. operators

A job includes a dataflow graph describing how data should flow through the different operators. Apache Flink supports a large number of operators.

- Map: Takes one element and produces one element.
- Filter: Evaluates a boolean function for each element and returns the element if the boolean function is True
- Flatmap: Takes one element and produces zero, one or more elements as a result.
- KeyBy: Partition a stream into disjoint partitions, based on a key. Apache Flink implements this function by use of a hash partition that sends similar keys to similar partitions.
- Window: Given a stream, group the data according to some characteristic defined by the window.
- Join: Given two streams, combine them into a single stream.

Some of these operations require information from previously processed records. These operations are called stateful. Stateful operations can be costly and are more expensive to scale. Defining a window constraint to the stateful operations allows the system to remove the information of records that fall outside the window. This lowers the costs of the operation and makes the system run more efficiently.

5.5.3. Dynamic scaling

Apache Flink supports elastic scaling in two ways: using Flink Reactive, or by redeploying the job.

Reactive mode The reactive mode of Apache Flink is a feature that allows for automatic scaling of the streaming application. Apache Flink monitors the number of task-managers deployed in the cluster. When a task-manager gets added or removed, Flink will automatically include or remove the task-manager from the current job and divide the workload equally between all available task-managers. Apache Flink will share the resources for all operators between the available resources and ensure optimal performance given the resources. This removes scaling complexity and only requires the user to determine the appropriate amount of resources for Flink Reactive. While this makes for more easy deployment of Apache Flink, it does take away a lot of control from the user, removing the ability for the user to assign resources to individual operators.

Redeployment To maintain control over the resource assignment, the system can also be scaled by redeploying the job. Apache Flink does not have an automatic procedure for this, so the user has to manually do so. This is done by first instructing the job-manager to create a checkpoint and stop the job. After stopping the job, the user can deploy a new job with a different configuration. As this requires stopping the job for a considerable amount of time, restarting the job will induce significant overhead. Though, it does give control over the resource assignment to the user.

5.5.4. Backpressure mechanism

Taskmanagers have two buffers in Apache Flink: an input-buffer, and an output-buffer. The in-buffer contains all incoming records that the operator has received, but has not yet processed and the output-buffer contains all records that the operator has processed but are not yet been accepted into the input-buffer of the downstream operator. The operator only processes records if there is enough space in the output-buffer. When an operator is not able to handle the incoming workload, its input-buffer gets filled and no longer accepts records from its upstream operator. The output-buffer of this operator then starts to fill as the records cannot be transferred to the downstream operator. This lets the upstream operator slow down its processing speed, triggering the same process for its upstream operator. This continues, eventually slowing down the speed at which the source operators accept incoming records. The entire system is now back-pressured.

5.5.5. Available Metrics

Apache Flink contains a built-in monitoring and metric system that allow users to effectively monitor the status of Flink jobs [16]. Apache Flink contains several pre-defined metrics that users can extend with custom-built metrics that fit their needs. The metric system provides information about the CPU usage, memory usage, and thread usage of the task-managers, together with their network usage and the status of their buffers. For each job, it reports the operators' backpressure time, throughput, and latency. When using Apache Kafka as a messaging system, the monitoring system also provides information about Kafka's input rate, lag, and the input-throughput of the system.

5.6. Container deployment

For the deployment of the different parts of the system, we package the individual parts in separate containers. The containers are then deployed on a Kubernetes server. These two technologies are now discussed individually.

5.6.1. Containers

A container is a standard unit of software, packaging the code and its dependencies required for running it in any computing environment, making it portable to be run across cloud and OS distributions

[13]. Traditionally, applications were run on physical servers, providing no support for resource allocation of the application. When virtualisation was introduced, it became possible to run multiple virtual machines in isolated operating systems on top of a single server. This enabled the users to divide the resources of the server flexibly between the individual machines, allowing for better scalability of the application. Containers work relatively the same as virtual machines, only with relaxed isolation constraints. This allows the containers to share the operating system with other applications while keeping their own file system and their assigned resources. This results in a much lightweight application that is decoupled from the underlying infrastructure, making it portable across cloud and OS distributions. For the containerisation of the different components of the framework, we use Docker[2].

5.6.2. Kubernetes

Kubernetes [38] is an open-source system for automatic deployment, scaling, and management of containerised applications. The bundling of an application into containers allows Kubernetes to run the application as a distributed system. Kubernetes provides deployment patterns, automatically handles container failures, provides manual and automatic resource allocations, and supports both horizontal and vertical scaling of these containers.

5.7. Messaging System

For the experimental setup, we use Apache Kafka as a messaging system functioning as middleware between the data generator and the SPE. We first introduce the need for a messaging system and then provide some background information about Apache Kafka.

5.7.1. The need for a messaging system

Large distributed systems consist of a large number of components that require continuous communication for sharing data and information. Maintaining this communication while dealing with network unreliability, strong coupling, and heterogeneity of components can be challenging [46]. One solution for this is connection-oriented communication with protocols such as TCP, as characterised in figure 5.13. This allows for fast and inexpensive communication between applications, but requires the components of the system to be constantly available, requires the distributed system to be transparent, and requires standardisation of the data structures shared between components [46].

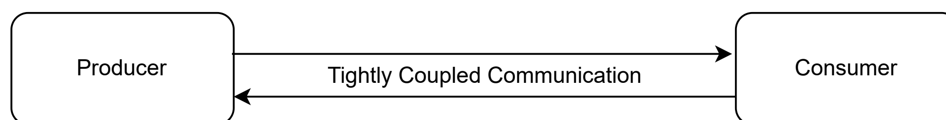


Figure 5.13: Traditional messaging using tightly coupled communication. Figure is inspired by [46].

An alternative approach is the introduction of a messaging system as middle-ware to manage the communication between different components. Data is sent from the producer to the messaging system, which stores the data in a queue. The consumer can then retrieve the data from the messaging system (figure 5.14). This allows for loosely coupled communication between the producer and the messaging system and does not require the consumer to be constantly available. Furthermore, the components do not need to be aware of each other's location, and no standard data structure is required but can be negotiated with the messaging system.

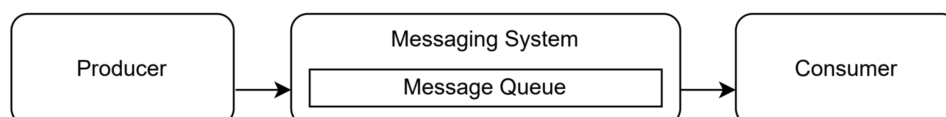


Figure 5.14: Messaging system supporting loosely coupled communication. Figure is inspired by [46].

[2]https://www.docker.com/

5.7.2. Apache Kafka

Apache Kafka [35] is a state-of-the-art messaging system that was initially designed at Linked-In, open-sourced in 2011, and further developed from a messaging queue to a full-fetched stream processing system [10]. The key notion of Kafka is to be a stateless broker, requiring consumers to maintain their state regarding data that is processed [46]. This requires Kafka to persist a single message copy independently from the number of consumers, resulting in high throughput and low latency. Kafka is designed as a Zookeeper-based distributed system [70], which allows for efficient scaling. Apache Kafka is a good solution for data movement and is often used to pipe data streams to different SPEs [46].

In Apache Kafka, data is stored in partitions, which are replicated over the cluster given a replica factor. Partitions are stored on individual Kafka servers, called Kafka brokers. Producers determine in which partition a message should be stored and send messages directly to the responsible broker. This is the broker containing the lead partition. The responsible broker stores the message in the partition and copies it to its replicas. Consumers can subscribe to one or more partitions, reading the data from the partitions. Kafka allows the categorisation of data into different topics. These topics indicate the type of data that is being stored and requires the data to be stored in separate partitions. Figure 5.15 illustrates an example of a Kafka cluster setup with three Kafka brokers, managing two topics with a replication factor of two.

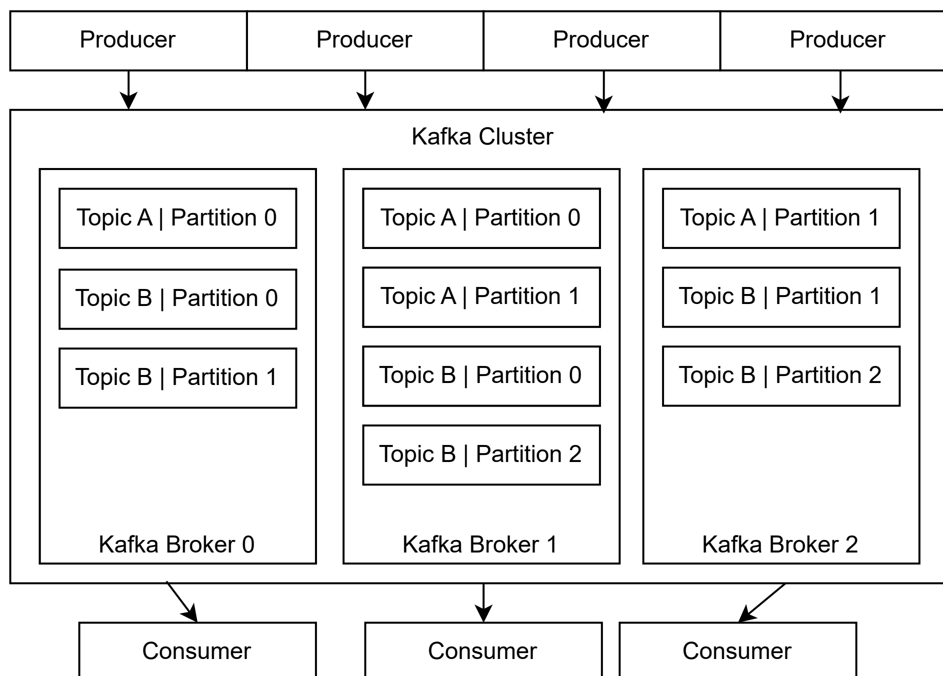


Figure 5.15: Example Kafka Cluster architecture with two topics: one having two partitions, and one having three partitions. The cluster is configured to have a replication factor of two.

5.8. Monitoring System

As discussed in section 2.2.1, auto-scalers depend on external monitoring systems to monitor the system and provide the necessary metrics. As a monitoring system for the framework, we use Prometheus [54]. Prometheus is an open-source monitoring tool that can monitor arbitrary applications, store the data accordingly and allow for retrieving, querying, and visualising the data accordingly.

5.8.1. Prometheus

Prometheus is originally built and open-sourced by Soundcloud in 2012 as a monitoring and alerting toolkit [54]. It collects and stores data as a time series, allowing for querying and over-time analysis of the data. Data is collected by scraping instrumented jobs that expose their current metric status through a web server. It also contains several built-in configurations allowing for automatic service dis-

covery. Running Prometheus with this setting enabled in a Kubernetes cluster allows Prometheus to automatically discover containers run in the cluster and collect their provided metrics. In addition, the Prometheus server can be queried by external services, providing the requested information. Snapshots can be created from the Prometheus deployment, storing the collected data as a backup. Using this, Prometheus can be snapshot and removed after running the experiment and later redeployed for analysis.

5.8.2. Conclusion

The evaluation framework extends the state-of-the-art for the evaluation of auto-scalers targeting SPEs by introducing proper end-to-end evaluation of the auto-scalers when run on different workloads and with different operator topologies. The framework is designed to compare the performance of state-of-the-art auto-scalers and can easily be extended with more workloads and different operator topologies. We have selected queries 1, 2, 3, 5, 8, and 11 from the Apache Beam Nexmark benchmark [6] as operator topologies, and proposed a cosine, random, increasing, decreasing, and a special convergence workload pattern. To compare the performance of the different auto-scalers, we have selected performance metrics latency, throughput, resources efficiency, resource usage, and the total number of scaling actions. As it is impossible to determine the optimal parallelism in a changing workload, the input rate with the system's input throughput was compared to get an estimation of the auto-scalers' parallelism accuracy. Furthermore, the system ran on a special convergence workload to investigate the convergence time of the auto-scalers. For the experimental evaluation, we use open-source state-of-the-art SPE Apache Flink and Apache Kafka as a messaging system. We deploy the framework in Kubernetes.

6

Implementation

As discussed in section 5, we design an evaluation framework for a performance analysis of state-of-the-art auto-scalers targeting SPEs. This framework consists of five parts: A data generator, an SPE (Apache Flink), a messaging system connecting the data generator and the SPE (Apache Kafka), a monitoring system (SPE), and auto-scalers. Figure 6.1 provides a global overview of the interaction between the different parts. We start by discussing the overall framework and the interaction between its different parts. Next, we discuss the implementation of the selected state-of-the-art auto-scalers and their integration within the Framework.

6.1. Kubernetes

Deployment The framework consists of five different systems: The data generator, Apache Kafka, Apache Flink, Prometheus, and the auto-scaler. These different systems are containerised and run as separate deployments in a Kubernetes cluster. The data generator, Prometheus, and the auto-scaler are deployed as individual containers. Apache Kafka runs its brokers in separate containers and requires Zookeeper to be deployed to coordinate its deployment. Apache Flink separates the job-manager deployment from the task-manager deployment. The job-manager and each individual task-manager are run as separate containers. Furthermore, an NFS server is run for persistent storage used by the check-pointing system of Apache Flink.

System interactions Interactions between the different systems are visualised in figure 6.1. The data generator generates data records that are passed to messaging service Apache Kafka. These records are then consumed by Apache Flink's task-managers, who are coordinated by the job-manager. When scaling, the task-managers create a checkpoint and save it to the NFS server. When starting a new job, this checkpoint is used as starting point for the system. Prometheus retrieves metrics from Apache Kafka and the job-manager and makes these available to the auto-scaler through its API. In addition, the auto-scaler directly communicates with the job-manager to retrieve additional information about the job. The Scale-Manager, which is run as part of the auto-scaler deployment, interacts with the Kubernetes API for scaling the number of available task-managers and redeploying the job. It also instructs the job-manager through its API to stop the current job and create a checkpoint.

6.2. Data Generator

The data generator is responsible for generating the records that require processing by the SPE. As described in section 5.4.3, the data generator needs to be able to generate different workloads. In this section, we discuss the overall architecture of the data generator and its modules. We explain how the data generator can easily be extended to generate additional workloads and discuss its deployment inside the Kubernetes cluster.

Kubernetes
Kubernetes API
Scale task-managers, deploy job-manager
Auto-scaler
Data Generator
Records
Metrics
Prometheus
Metrics
Apache Kafka
Topology information
Records
Metrics
Apache Flink
Job-manager
Task-managers
Save and retrieve checkpoints
NFS server

Figure 6.1: System overview of the Kubernetes cluster, displaying its container deployments and their interactions.

6.2.1. Architecture

The data generator consists of two parts: the data generator and a modular pattern generator. The architecture of the data generator is visualised in figure 6.2.

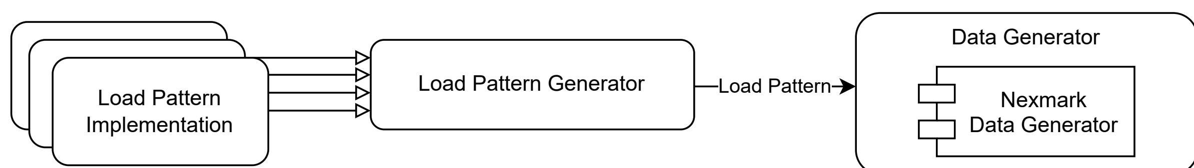


Figure 6.2: Architectural overview of the data generator.

Data generator We adapt the existing Nexmark benchmark implementation [1] to be able to produce records following an arbitrary input workload. The workload is fed to the system through an array of per-minute input rates. Every minute, the next input rate from the array is used as the number of records

[1] `https://github.com/nexmark/nexmark`

the data generator generates per second. During the experiments, we disable the hot-item functionality built into the original Nexmark Benchmark system. This lets all persons host three actions and place 10 bids over time, preventing data skew. When a stream is not being used in the current query, no data is generated. We extend the data generator to support parallel execution, allowing for higher input rates.

Pattern generator We extend the system with a modular load-pattern generator module, that generates the array representing the syntactic workload to be used by the data generator. After the necessary configurations are set, the data generator fetches the load pattern from this module before starting the data generator. We implement a large number of load patterns, including the cosine, random, decreasing, increasing, and convergence load patterns. In addition, we add the possibility to add random spikes to different workloads. We are, however, unable to use this functionality for our experimental evaluation due to time constraints.

6.2.2. Implementation

We implement the system in Java, with Apache Maven 3.0 as a project management system and compression tool. During deployment, the workbench uses 18 CPUs and 18 GB of memory. The data generator is run with a parallelism of 36, each one of them acting as a separate Kafka producer. This allows the data generator to produce a total of 2.5 million records per second. Data generated by the data generator is forwarded to the Kafka cluster, who makes the data available to Apache Flink.

6.3. Apache Kafka

To add fault tolerance and prevent data loss, Apache Kafka is used as messaging service. It receives data from the data generator and passes it through to Apache Flink. When Apache Flink is unable to process the incoming data due to under-provisioning or unavailability, Apache Kafka stores it in its local FIFO queue. Apache Kafka provides performance metrics used by some of the auto-scalers and our performance analysis. We discuss the available metrics of Apache Kafka and then discuss its implementation in our cluster.

6.3.1. Implementation

So, as discussed before, the data generator connects to Apache Kafka by multiple Kafka producers. In total, we have three Kafka brokers handling the incoming requests, each requesting 2 CPUs and 4GB of memory. For each topic, the replication factor is set to one and we set the number of partitions to the maximum amount of consumers available in the setup, which is similar to the maximum amount of available task-managers. To prevent massive amounts of data to be stored on our server, we set the retention time (time for the data to be removed from the partition) to five minutes, with the retention check interval set to five seconds. As we have limited storage on the server, we assume data will spend no longer than 5 minutes in the Kafka queue before being processed by Apache Flink. 5 minutes is, because of this, an upper bound for the latency of the system at any point in time.

6.3.2. Metrics

While Kafka provides several metrics that can be useful for the implementation of different auto-scalers, we only use the total amount of messages received by the Kafka brokers to determine the system's input rate. Varga et al. [66] also use the maximum queue size of any of the Kafka partitions for their original HPA-Varga implementation to get an estimation of the total amount of records in the queue. Apache Flink, however, has since added a metric that shows the total amount of pending records in the Kafka queue, providing a better estimation of the total lag.

We use the following metrics of Apache Kafka:

- *Input_rate*: the sum of the total amount of messages received by the Kafka brokers per second.

6.4. Apache Flink

As SPE we use Apache Flink. As a basis for the Nexmark queries, we use the implementation used for DS2's [36] evaluation [2]. As the original queries require manual scaling, we extend the queries to

[2] `https://github.com/strymon-system/ds2`

support our custom scaling protocol. The topology of the queries follows the topologies discussed in section 5.4.1. The sources of the topology fetch the data from Apache Kafka and feed them into the system. Apache Flink attempts to retrieve as many records as it can handle, fetching records from the queue in FIFO order. If Apache Flink is under-provisioned, its backpressure mechanism automatically slows down the rate at which records are fed into the system, increasing the size of the Kafka queue.

6.4.1. Scaling

For the experimental evaluation, we use operator-based scaling. We do this by disabling slot sharing in the task-managers, only allowing them to work on a single operator. Because operator-based scaling can is not natively supported by Apache Flink, we write a scaling module that does this automatically. When receiving a scaling request, it lets the task-managers create a checkpoint that is saved at the persistent volume and stops the job. After the checkpoint is created and the job is stopped, Kubernetes is instructed to change the number of available task-managers to the required number of task-managers. If the number amount of task-managers is exceeded, the available task-managers are divided relatively to the number of requested resources over the operators. After the new task-managers are available, a new job-manager is deployed with the requested parallelism. After this, scaling is done. During scaling, Apache Flink is not able to process the incoming records, increasing the size of the Kafka queue.

6.4.2. Metrics

During the experiment, many metrics provided by Apache Flink are used for the decision-making of the auto-scalers and the performance evaluation. These metrics are retrieved by Apache Flink's built-in monitoring system and are made available through Prometheus. We list the metrics retrieved from this system below. While most of the metrics are available in Prometheus, some information is fetched by the auto-scalers directly from the job-manager. This includes the current topology, its parallelism configuration, a mapping between task-managers and the operators they are assigned to, and the CPU load of the task-managers. Also, the scaling module fetches information about the current status of the stopping job and the checkpoint that is being made.

We use the following metrics of Apache Flink:

- Latency: The average amount of time records spend in the Kafka queue before being emitted by Apache Flink.
- Input throughput: The number of records per second being emitted by Apache Flink. For DS2, we also fetch this metric on a per-operator basis.
- Output throughput: The number of records per second processed by Apache Flink. For DS2, we also fetch this metric on a per-operator basis.
- Lag: We fetch the lag as the total number of records Flink knows are pending in the Kafka cluster.
- CPU-load: The average utilisation of the task-managers CPU.
- Backpressure-status: A binary value indicating whether an operator is considered to be back-pressured.
- Backpressure-time: The amount of time per second task-managers spend back-pressured.
- Buffer-in usage: A per-operator metric indicating how much the operators' buffers are filled.
- Idle-time: The amount of time per second task-managers spend waiting for input.
- Amount of task-managers: The total amount of task-managers available to the Job-manager at any point in time.
- Instance-input-rate: the input rate of an individual operator instance. The sum of the instance input rates is equal to the input throughput.
- Instance-output-rate: the output rate of an individual operator instance. The sum of the output rates of all sink instances is equal to the output throughput.
- Instance-busy-time: the amount of time per second an individual operator instance spends processing records.

6.5. Prometheus

For monitoring the system, we use Prometheus. Prometheus is configured to automatically discover services in the Kubernetes cluster and scrape them accordingly. Auto-scalers fetch data from Prometheus by calling its API with a query, requesting the necessary information. In practice, we found that metrics are not always correctly retrieved, making Prometheus return empty results when querying. For this reason, we take the average result over three times the scrape interval, to prevent empty results.

Implementation Prometheus is deployed using Helm [29] as a separate container in the cluster. It is configured to automatically discover and scrape services deployed in the Kubernetes cluster, scraping metrics from Apache Flink, Apache Kafka, the Kubernetes cluster, and itself. The scrape interval of the system is set to five seconds.

Fetching performance metrics After running an experiment, performance metrics are retrieved from Kubernetes by calling its API. In addition, a snapshot of the Prometheus instance is saved, together with the timestamps of the time the experiment took place in. This allows for further analysis and querying of Prometheus after the experiment is finished and the containers are removed. The data fetched from Prometheus is saved in a standardised format, allowing for automatic reading and generation of plots using Python's Pandas [3] and MathplotLib [4] libraries.

6.6. Auto-scaler Architecture

The auto-scaler is run from a main function that accepts the provided parameters and instantiates the requested auto-scaler with the provided configuration. Every auto-scaler contains a run() method that starts the functionality of the auto-scaler. An auto-scaler contains three helper classes: a *Configurations* class containing all configurations provided by the user, an *ApplicationManager* that acts as an adapter between the auto-scaler and the interactions with external applications, and a *ScaleManager* that contains the functionality required by the application to scale the application. The interaction between the different parts of the system is visualised in figure 6.3. We discuss the different modules of the auto-scalers in this section. In section 6.7, we discuss the implementation of the individual auto-scalers.

6.6.1. Auto-scaler Logic

The main module of the framework is the module containing the auto-scalers logic. In figure 6.3 this is visualised as the module named *DS2*. We implement four different auto-scalers: Dhalion, DS2, HPA-CPU, and HPA-Varga. To minimise code duplication, we design the auto-scalers as shown in figure 6.4. DS2, Dhalion, and HPA extend an auto-scaler class that can be instantiated and run from the Main class. The HPA auto-scaler module is defined as an abstract class, containing the basic functionality of HPA. Both HPA-CPU and HPA-Varga extend this functionality with their own metrics.

6.6.2. Configurations

The *Configurations* class contains all configurations of the auto-scaler that it requires during run-time. As can be seen in figure 6.3, we have three types of classes surrounding the *Configurations* class: the *ExperimentData* class, a global Configurations class and the auto-scaler specific configurations class (shown in the example as *DS2-Configurations*).

Configurations The *Configurations* class contains all configurations that are required for all auto-scalers and includes configurations for the scaling module, configurations for the Prometheus module, and some general cluster configurations.

ExperimentData The *ExperimentData* class is a separate class that includes data specific to the experiments that are run. This includes a mapping between source operator names and their corresponding parameters to set their parallelism, a method that maps topic names to source operators, and some name conversions to allow compatibility with the output of Prometheus and the job-manager.

[3] `https://pandas.pydata.org/`
[4] `https://matplotlib.org/`

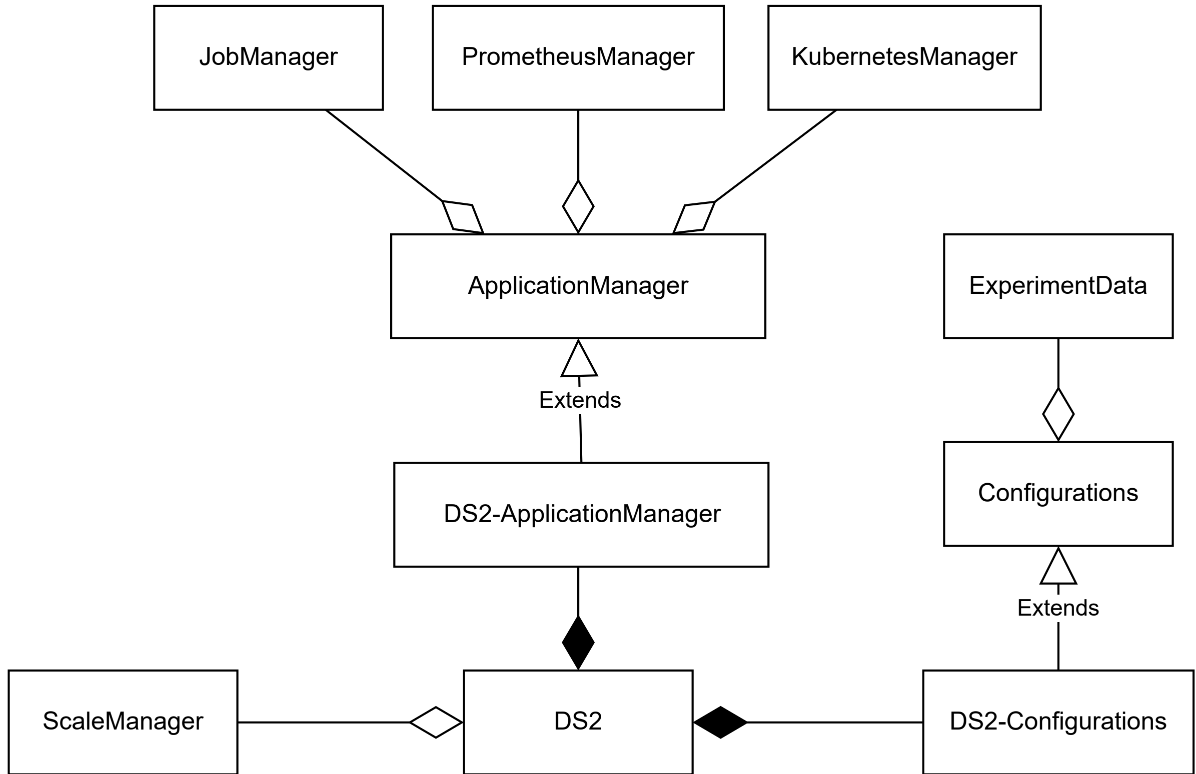


Figure 6.3: Example architecture of the implementation of the DS2 auto-scaler. It shows its different modules and the loosely coupled interaction of the auto-scaler with the ScaleManager and other external applications.

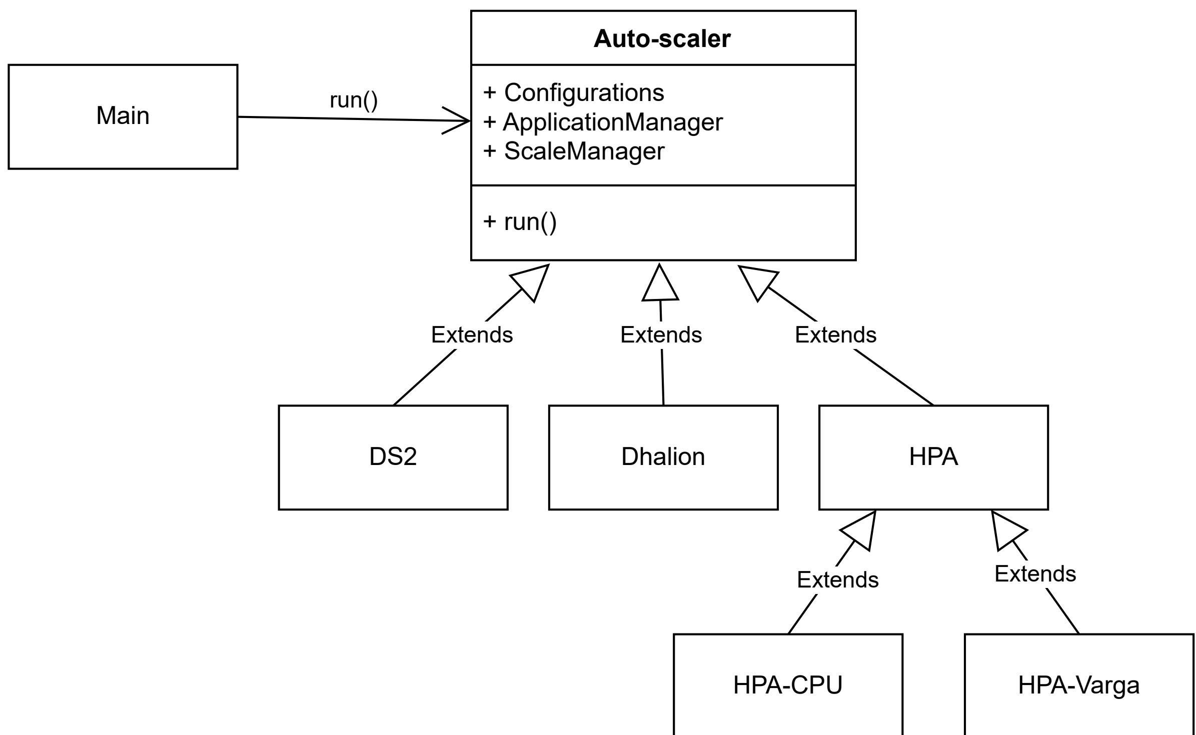


Figure 6.4: Diagram visualising the implementation and inheritance of shared functionalities of the different auto-scalers.

When introducing additional experiments, this class can be updated with additional experiment details. Extending this feature to use a more structured approach, like a JSON input file would allow for even better configurability of the application.

Auto-scaler specific configurations When auto-scalers require additional configurations, an auto-scaler-specific configuration can be created to extend the configurations class. By enforcing this configuration class type in the constructor of the auto-scaler, the main function creating and running the auto-scaler is forced to provide the configurations required for running the auto-scaler. We discuss the auto-scaler dependent configurations in section 6.7.

6.6.3. Application Manager

The Application Manager functions as an adapter between the auto-scaler and the external services. It consists of three parts: the global *ApplicationManager*, service-specific managers, and an auto-scaler-specific application manager. Their interactions can be seen in figure 6.3 with the *JobManager*, *PrometheusManager*, and *KubernetesManager* as service-specific managers, and the *DS2-ApplicationManager* as the auto-scaler specific manager. We now discuss the three individual parts of the Application Manager.

Service specific application Manager During deployment, the auto-scaler interacts with multiple services in the cluster. As these interactions may change over time, we create for every external service a separate manager that is responsible for all interactions with that specific service. For our experiment, this is a Kubernetes Manager responsible for the interaction with the clusters' API, a Job-manager Manager, responsible for the interaction with the job-manager of Apache Flink, and a Prometheus manager, responsible for the interaction with Prometheus.

ApplicationManager The applications manager acts as an adapter between the auto-scaler and the external service managers. It provides all functionality that is required by the auto-scalers and performs this by interacting with the service-specific application managers. When services change or a new service is required for running the auto-scalers, interactions with external services have only to be changed in the application manager, without the need for changing the logic of the auto-scalers. The ApplicationManager contains the shared functionality required by the different auto-scalers.

Autoscaler specific application manager When an auto-scaler requires additional interactions with external services, the ApplicationManager can be extended with an application-specific manager. In this manager, auto-scaler-specific functionality can be placed, decoupling the auto-scalers' functionality from its interactions with external services.

6.6.4. Scale-Manager

As the execution of the scaling actions is not part of the auto-scaler, we create an additional module that scales the system. The Scale-Manager provides two types of scaling: by using Flink Reactive or by performing operator-based scaling. When using Flink reactive, the Scale-Manager changes the parallelism of the current task-manager deployment, after which Flink Reactive divides the resources between the operators automatically. When performing operator-based scaling, the auto-scaler does so by stopping the current job and creating a checkpoint. The task-managers are scaled, after which a new job-manager configuration file is generated with the new parallelism configurations. When the task-managers are ready, the new job-manager is deployed with the previously created checkpoint. After scaling, depending on the configuration of the system, the Scale-Manager triggers a cool-down period that disables scaling for a period, allowing the system to recover from its scaling action.

When higher parallelism is requested than supported by the cluster, the Scale-Manager can be configured to cancel the scaling action or to divide the available resources relatively to the requested resources. When, for example, in a three-operator topology a parallelism of 10, 30, and 20 is requested, but a maximum of 30 task-managers are available, the operators are scaled to a parallelism of 5, 15, and 10 respectively. For our cluster, we set the maximum amount of task-managers to 80 task-managers. Due to technical constraints, we also enforce a maximal increase in a parallelism of

15 per operator per scaling action, as larger scaling actions tend to crash the cluster. Finally, an over-provisioning factor can be configured for the auto-scalers. This over-provisioning factor increases or decreases the per-operator parallelism accordingly.

6.6.5. Implementation

The auto-scalers are implemented in Python and are containerised and run as a Docker container [13]. The interaction with the Prometheus pod and the job-manager pod is done by accessing their API using HTTP requests.

6.7. Auto-scaler Logic

We design a modular system for the implementation of the auto-scalers, separating the auto-scalers' logic from its interactions with the cluster. This allows the auto-scalers to share functionalities, allowing for fast implementation of additional auto-scalers. First, we discuss the global framework and the design choices we made. Next, we discuss the implementation of the individual auto-scalers, whose theory we discuss in section 4. As these auto-scalers are designed for different SPEs and are based on metrics not directly available in Apache Flink, we have to make some adaptations to make the auto-scaler work in our current context. We first discuss Dhalion and DS2, followed by HPA-CPU, and HPA-Varga. Metrics mentioned in this section are discussed in more detail in section 5.3.

Auto-scaler run-time blueprint In the literature, auto-scalers generally consist out of a module that, when called, determines whether to scale the system at that specific point in time. For this, it is up to the user to determine when to call this module and what to do with the result. To create a fair comparison between the different auto-scalers, we use the same 'decision moments' for each auto-scaler. This is done by running the auto-scaler in a loop, waiting for some time and then calling the decision module of the auto-scaler. If the proposed parallelism configuration is different from the current parallelism, the Scale-Manager is called, and the system is scaled accordingly. After the scaling, and the cool-down period followed by it, the loop repeats. For our experiments, we set the iteration period to 15 seconds, equal to the time we average the Prometheus' values over.

6.7.1. Dhalion Implementation

Implementation We implement Dhalion as described in the original paper. We first check whether back-pressure exists in the system, by checking the *backpressure-status* of the individual operators in the topology. As the source operators do not have upstream operators, we check whether the lag of the system is increasing at a larger speed than the *kafka-lag-rate-to-be-backpressured* threshold. If backpressure exists, we determine the cause of the backpressure by following the back-pressured operator downstream, until we find the first operator that is not back-pressured. This operator is recorded as the system's bottleneck and is scaled up accordingly. If no back-pressure exists in the system, we assume the system is in a healthy state and explore the possibility to scale down some of the operators that are potentially over-provisioned. As described in the original paper, we select all operators that do not have records in their buffer. We say an operator has no lag if its *buffer-in-usage* is below the *buffer-usage-close-to-zero* threshold. As the source operators of the topology only fetch records from the Kafka queue when it can process them, we look at the size of the Kafka queue instead of the buffer-in usage of the source operators. For this, we say a source-operator has no lag if the lag of the system is smaller than the *kafka-lag-close-to-zero* threshold. If an operator has no lag, it is scaled down accordingly.

Scaling up Dhalion determines the scale-up factor of the operator by comparing its queue size (the workload it was not able to handle) with its input throughput (the workload it was able to handle). Because of the back-pressure mechanism of Apache Flink, we can only use the size of the Kafka queue to estimate the load the system was unable to handle. As this is an indication of the performance of the entire system and not of the operator itself, we look at the *backpressure time* of its upstream operators instead. An operator's *backpressure time* is the time per second it spends backpressured. When scaling up a bottleneck operator, we take the largest backpressure time of its upstream operators and

calculate the scale-up factor in the following way:

$$operator_scale_up_factor = \frac{backpressure_time}{1 - backpressure_time} \tag{6.1}$$

As the source operators do not have an upstream operator, we calculate the scale-up factor using the input rate and the change in the system's lag. We calculate the scale-up factor for the source operators using the following equation

$$source_operator_scale_up_factor = \frac{rate(lag)}{input - throughput} \tag{6.2}$$

Scaling down For scaling down the operator, Dhalion allows the user to provide the scale-down factor as the initial configuration of the system. In the experimental evaluation of Dhalion, the authors set this factor to 1.2, scaling the operator's parallelism down with 20%. For our experimental evaluation, we also use this configuration.

Configurations As described above, Dhalion has four configurations that have to be provided by the user. The configurations and their default values are shown in table 6.1.

Dhalion Configurations	Default Value
Scale-down factor	1.2
Kafka-lag-rate-to-be-backpressured threshold	1000 records/s
Buffer-usage-close-to-zero threshold	0.2
Kafka-close-to-zero threshold	10,000 records

Table 6.1: Default configuration settings of auto-scaler Dhalion.

6.7.2. DS2 Implementation

Impementation For DS2, we use the decision-making module of the original DS2 implementation of Kalavri et al. [36], available on GitHub [5]. To connect the DS2 module with our application, we fetch the necessary metrics required for the decision-making, and write them to file. The file locations are then passed as parameters to the DS2 module when running it using Cargo [6]. When the DS2 module finishes processing, it returns the recommended parallelisms. These are then passed to the Scale-Manager, scaling Apache Flink accordingly.

Metrics As metrics, DS2 requires the topology of the system, its parallelism configuration, per-topic input rates, and for each operator instance its *instance-input-rate*, *instance-output-rate*, *instance-true-processing-rate*, and *instance-true-output-rate*. The *instance-input-rate* and *instance-output-rate* are directly provided by Apache Flink, as discussed in section 5.3. The *instance-true-processing-rate* is calculated by dividing the *instance-input-rate* by its *instance-busy-time*. The *instance-true-output-rate* is calculated by dividing the *instance-output-rate* with the *instance-busy-time*.

Over-provisioning-factor The original implementation of DS2 assumes records to be discarded when the SPE is unable to process them, having a constant lag of 0. Because of this, DS2 proposes parallelism configurations that can process the incoming workload but are not able to process the build-up lag in the system. This results in the Kafka queue being filled up constantly, having high latency and bad overall performance. To counter this effect, we make use of an over-provisioning factor, that increases the recommended parallelism of DS2 with itself. As default, we use an over-provisioning factor of 1.2, resulting in a 20% higher parallelism than DS2 requests.

Configurations Table 6.2 provides an overview of the auto-scaler configurations of DS2 and their default values.

[5] https://github.com/strymon-system/ds2
[6] https://doc.rust-lang.org/cargo/

DS2 Configurations	Default Value
Over-provisioning factor	1.2

Table 6.2: Default configuration settings of DS2.

6.7.3. HPA-CPU Implementation

Operator-based scaling HPA is a general-purpose auto-scaler that is unable to capture the complex relationships between different operators. This means that HPA does not support operator-based scaling and is only able to scale the number of available task-managers in the cluster. Because of this, we implement HPA ourselves and adapt it to support operator-based scaling. We implement two auto-scalers for HPA. Our first auto-scaler targets the CPU utilisation of the task-managers and is used as a baseline in the experimental evaluation. We call this auto-scaler HPA-CPU. The second auto-scaler uses the metrics suggested by Varga et al. [66]. We refer to this auto-scaler as HPA-Varga. As HPA-CPU and HPA-Varga use the same basic HPA functionality, we implement HPA in a separate module that can be extended for HPA-CPU and HPA-Varga.

HPA implementation In every iteration, HPA determines the optimal parallelism for every operator in the topology. This is added to a list, where the suggested parallelism is saved together with a timestamp. The highest suggested parallelism of the previous *scale-down-window-period* is selected as the desired parallelism of the operator. The combination of the desired parallelisms of the operators is then passed to the Scale-Manager to execute the desired scaling action.

HPA-CPU HPA-CPU attempts to keep the average CPU utilisation of every operator close to the CPU-utilisation-target-value. For this, the operator has to retrieve the CPU load of the different task-managers, which is directly available in Prometheus. The provided information, does, however not include the operator the task-manager belongs to. This information is fetched from the Job-Manager API, providing a mapping between the task-manager-IDs and the operators. Using this information, the CPU load of the task-managers belonging to the operators is averaged and used as controlled value for HPA-CPU. HPA will now attempt to have the average CPU load match the CPU-load-target-value, which we set at 0.7, or 70%.

Configurations Table 6.3 provides an overview of the different configurations used by HPA-CPU and their default values.

HPA-CPU Configurations	Default Value
Scale-down-window-period	300s
CPU-utilisation-target-value	0.7

Table 6.3: Default configuration settings of HPA-CPU.

6.7.4. HPA-Varga Implementation

Operator-based scaling HPA-Varga is originally designed to be used in combination with Flink Reactive. This creates several problems when implementing it for operator-based scaling. We reuse the HPA module used for HPA-CPU and configure it to use the utilisation and relative-lag-change as controlled variables. We set the default target value of the utilisation to 0.7 and the default target value of the relative-lag-change to 1.0.

Relative-lag-change value When extending HPA-Varga to operator-based scaling, we run into a problem when considering the relative-lag-change. For calculating the relative-lag-change, the original implementation uses the size of the Kafka queue (lag). As this is an indication of the health of the entire system and not of an individual operator, we cannot directly use it for operator-based scaling. To solve this, we check whether the lag of the system is increasing. If the lag of the system is increasing slower than the *minimum-kafka-lag-rate-when-backpressured* threshold, we set the relative-lag-change of all

operators to -1. This makes the auto-scaler suggest minimal parallelism for the relative-lag-change value. As HPA always picks the highest parallelism when dealing with multiple controlled variables, HPA will now always propose the suggested parallelism determined by the utilisation factor. If the lag is larger than the *minimum-kafka-lag-rate-when-backpressured* threshold, we assume there is backpressure in the system. Using the backpressure_status metric, we search for the bottleneck operator causing the lag. If none of the operators is experiencing backpressure, we assume that the source operators are the cause of the backpressure. After we identified the cause of the backpressure, we calculate the operators' relative-lag-change using the lag of the Kafka queue and set the relative-lag-change of the other operators to -1. We do this because the bottleneck operator can be seen as the cause of the increase in the lag, resulting in the high relative-lag-change.

Relative-lag-change metrics In the original implementation of Varga et al., the maximum lag in one of the consumers is used to estimate the total Kafka queue size. The authors calculate the Kafka queue size by multiplying the maximum consumer lag by the number of consumers in the system. While this gives a rough estimation of the health of the system, it only provides an upper bound for the Kafka queue size. In a recent update of Apache Flink, we are now able to get the exact size of the Kafka queue, using the pending-records metric. For our implementation, we use this metric instead of Kafka's maximum consumer lag to get the current lag of the system. We calculate the derivative of the rate in change of the lag using Prometheus' built-in derived function. This function estimates the metrics' derivative by considering the values of the previous derivative-period-length, which we set, just as in the original paper, to 60 seconds.

Utilisation For the utilisation value of every individual operator, we average the idle time of the operators' instances. We then subtract the idle time from 1, resulting in the per-operator utilisation time. This value is then used to determine the optimal parallelism.

Configurations Table 6.4 provides an overview of the different configurations used by HPA-Varga and their default values. Just as with HPA-CPU, we set the scale-down-window size to HPA's default value of 300 seconds. We consider the system to be back-pressured when the Kafka lag is increasing faster than 1000 records per second.

HPA-Varga Configurations	Default Value
Scale-down-window-size	300s
Utilization-target-value	0.7
Relative-lag-change-target-value	1.0
Kafka-lag-rate-to-be-backpressured threshold	1000 records
Derivative-period length	60s

Table 6.4: Default configuration settings of HPA-Varga.

6.8. Conclusion

The different systems required for the evaluation framework are containerised and ran in a Kubernetes cluster. The data generator and the auto-scalers are designed to minimise work when adding new workload patterns and auto-scalers. This is done by placing shared functionality in common modules and by separating the data-pattern generation logic from the data generator itself. Furthermore, interactions between the auto-scalers and external systems are performed through an adapter, making it possible to change the auto-scalers' interaction with external systems without having to change its logic. The auto-scaler logic of HPA-Varga and HPA-CPU is adapted to support operator-based scaling. Furthermore, a custom version of HPA is implemented that supports operator-based scaling.

7 Results

In this chapter, the results of the experimental evaluation are presented and used to compare the performance of the auto-scalers. We start with a parameter optimisation of the most significant configurations of the auto-scalers to ensure good performance in the remainder of the experiments. Next, we investigate the performance of the auto-scalers when run on different workloads. We investigate the ability of the auto-scalers to react to fast increases in the workload and their ability to save resources when the workload goes down. Next, we investigate the performance of the auto-scalers when run on different types of queries. The queries have different structures, different sources, and different operator types, resulting in the queries having different resource requirements. We investigate how well the auto-scalers can estimate the number of resources needed by the auto-scalers and how they respond to this in terms of run-time behaviour. After that, we investigate the convergence time of the auto-scalers and investigate the accuracy of the proposed parallelism configurations. We finish the chapter by providing an overview of the experiments we perform and their results. Furthermore, we discuss the performance of the individual auto-scalers and comment on their strengths and weaknesses.

Some of the experiments result in a crash in Apache Flink. This tends to happen during scaling where task-managers are responsible for a lot of state and fail to create a checkpoint. This causes the job-manager to lose connection with the task-managers causing the job to fail. During experimentation, this issue mostly happens with auto-scaler HPA-Varga. HPA-Varga often issues large scale-down operations when the workload of the system is decreasing. When the input rate goes up again and the auto-scaler issues scale-up requests, the low number of task-managers are all responsible for a large amount of state and fail to create a checkpoint. As we are unable to fix this problem within our time constraints, we leave the results of HPA-Varga out when this happens and leave it for future work.

7.1. Parameter Optimisation

When deploying an auto-scaler, the user is often required to configure the auto-scaler for optimal performance. Choosing the right configurations is essential for the auto-scaler to function properly in a production environment. This can, however, be a difficult task that requires extensive knowledge of the system and of the auto-scaler itself. The auto-scalers we use for the experimental evaluation contain several parameters that have to be set before deployment. In this section, we investigate the impact the different parameters have, reason over their influence on the run-time behaviour of the auto-scalers, and select the most appropriate parameters for the remainder of the experimental evaluation. While doing so, we comment on the basic mechanics the auto-scalers use for determining the right scaling actions and show how this is found in the results of the experiments. We start by discussing the experimental setup, together with the parameters chosen per auto-scaler to investigate. Next, we present the results of the experiments, discuss the influence of the parameters on the behaviour of the auto-scalers and select the parameters to use during the experimental evaluation.

7.1.1. Experimental Setup

Ideally, we would optimise the parameters of the auto-scalers on every experiment we perform in the experimental evaluation. As this would require us to run every experiment multiple times on every auto-scaler, we are unable to do so due to time constraints. Instead, we choose to evaluate the auto-scalers on the cosine workload pattern. While the pattern of the cosine workload is predictable, it shows the ability of the auto-scalers to both scale up and down the parallelism of the system, providing insight into these two activities. Furthermore, we run the auto-scaler on both queries 1 and 11. With query 1 being a state-less query, it is also representative of query 2. Query 11 is a state-full session-window operation that requires significantly more resources than query 1. We choose this query as a representative for queries 3, 5, and 8. We set the maximum input rate of the cosine pattern to 2.200.000 records/second and the minimum to 200.000 records/second, with a random noise that deviates a maximum of 100,000 records per second from the actual input rate. We use a cosine period of 60 minutes and run the experiment for a total of 140 minutes. For query 1, every operator starts with an initial parallelism of 5 and for query 11 with an initial parallelism of 10. For every auto-scaler, we select the most significant configuration that can have the most impact on the performance of the auto-scaler. For Dhalion, this is the *scale-down-factor*, for DS2, this is the *over-provisioning factor*, for HPA-CPU this is the *CPU-utilisation-target-value* and for HPA-Varga this is the *Utilisation-target-value*. The rest of the auto-scaler configurations are set to their default value as discussed in section 6.7.

7.1.2. Dhalion

For Dhalion we investigate the impact of the scale-down factor on the performance of the auto-scaler. As discussed in section 6.7, the scale-down factor is a pre-set configuration that determines by how much Dhalion scales down an operator in the system when it decides it should scale down. In the original paper, this value is set to 1.2. For our experiment, we investigate the performance of the auto-scaler when set to 1.1, 1.15, 1.2, 1.25, and 1.3. We summarise the results of Dhalion in table 7.1 and 7.2 and graphs 7.1, 7.2.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion 1.1	16.8	31	14.8	0.121	65.1	27
Dhalion 1.15	15.4	32	20.1	3.6	80.1	27
Dhalion 1.2	13.2	30	68.3	23.7	233.3	27
Dhalion 1.25	14.2	38	104.1	36.1	339.5	27
Dhalion 1.3	15.1	41	106.9	41.8	348.7	26

Table 7.1: Dhalion Configurations for query 1.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion 1.1	34.5	66	84.7	7.8	366	26
Dhalion 1.15	32.1	66	107.1	24.7	334.2	27
Dhalion 1.2	22.8	51	130.7	60	353.3	27
Dhalion 1.25	35.7	73	121.5	52.2	358.2	26
Dhalion 1.3	30.9	73	145.3	90.2	388.6	27

Table 7.2: Dhalion Configurations for query 11

The scale-down-factor In general, the higher the scale-down factor, the more aggressively the auto-scaler reduces resources when it finds the system to be in a healthy state with the buffers of the operator empty. Scaling down the operator parallelism by too much causes backpressure in the system, increasing the lag. This again triggers a scale-up action, converging the auto-scaler to the optimal parallelism. Though, as we scale down all operators with an empty buffer, while only scaling up the single bottleneck of the system, it may take multiple scale-up actions before we correct for the overly-aggressive scale-down action. Setting the scale-down factor too high would therefore induce significant costs and the configuration should therefore be set carefully. Setting the scale-down factor too low would, however, cause the system to respond to a workload reduction slowly, resulting in constant over-provisioning.

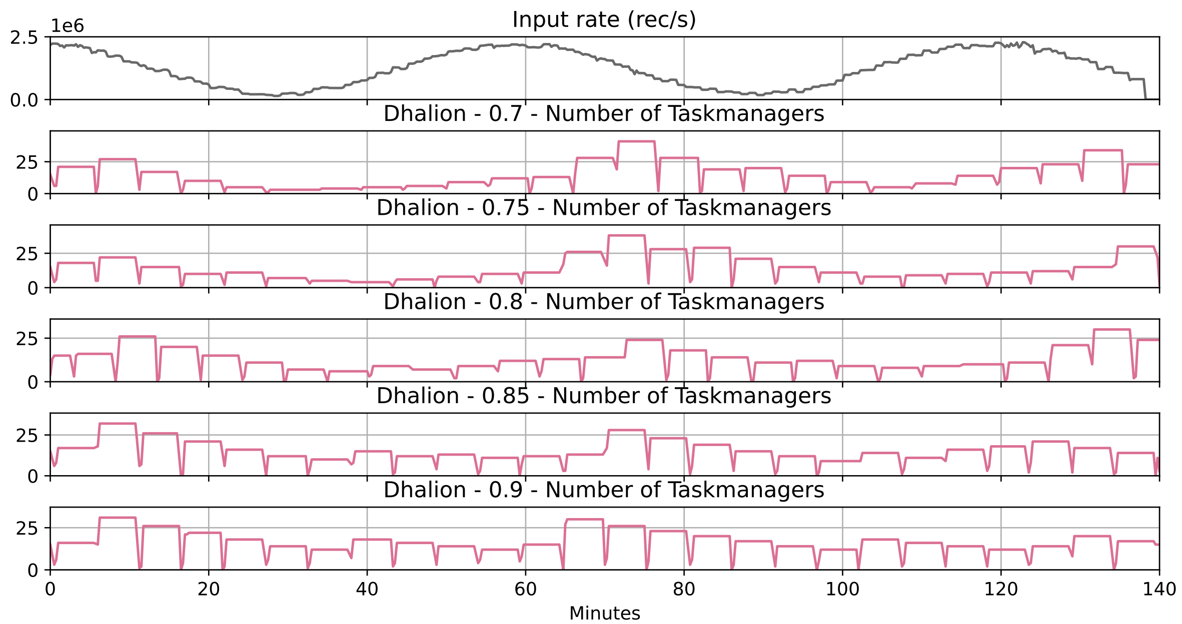


Figure 7.1: Task-managers deployed by Dhalion with different scale-down factors when run on query 1 with the cosine workload pattern.

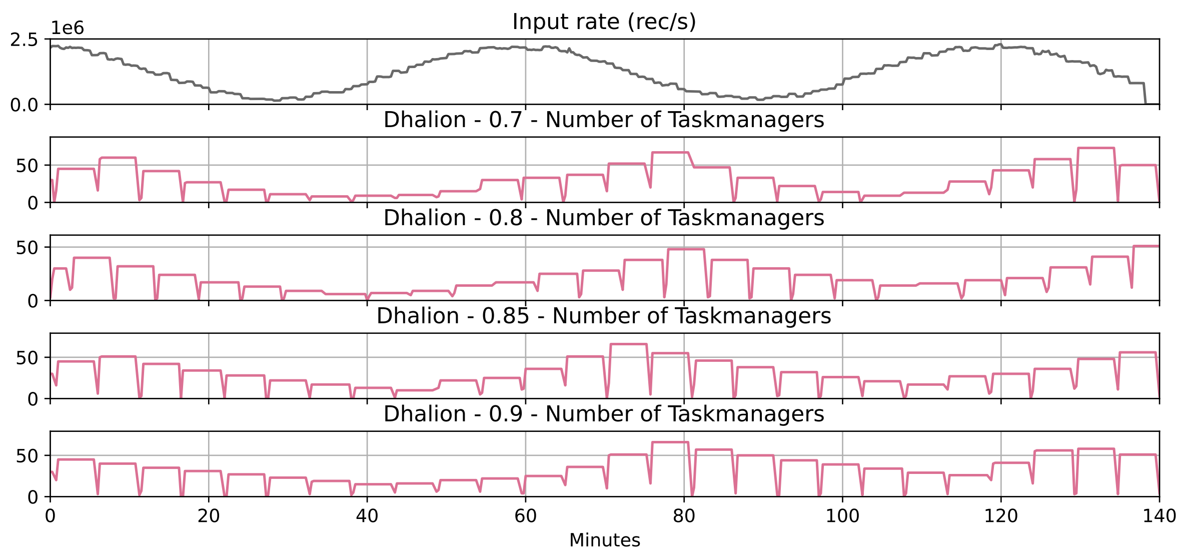


Figure 7.2: Task-managers deployed by Dhalion with different scale-down factors when run on query 11 with the cosine workload pattern.

While this ensures low latency, it does result in inefficient use of the task-managers.

Performance analysis Looking at the results in table 7.1 and 7.2, we see the average amount of task-managers deployed in the experiment increasing the further away the configuration is set from 1.2 for both query 1 and 11. One would expect the latency to decrease when the number of deployed task-managers increases. While this is the case for the scale-down-factor parameters lower than 1.2, the latency only increases with a higher scale-down-factor. The increase in latency can be explained by two things. First, scaling down too aggressively causes backpressure and may require an additional scale-up action to correct for its inaccuracy. In the run of Dhalion on Query 1 with a scale-down factor of 1.25, this can be seen to even trigger a scale-up action while the input rate is decreasing. The second problem that arises can be observed in query one, where the runs with the scale-down factor of 1.25

and 1.3 both result in a parallelism of 1 for every operator on the lowest point of the cosines pattern. While this is enough to process 200.000 records per second, it should be scaled up again when the input rate starts rising. Scaling up again is, however, performed slowly. This is because Dhalion only scales up a single bottleneck operator at a time and because a significant increase in parallelism from parallelism 1 requires a high scale factor. One scale-up operation for an operator might therefore not be enough to handle the incoming workload, requiring multiple scaling operations per operator. In query 11, only the run with scale-down-factor 1.1 can keep up with the steep workload increase. It manages to do so by barely scaling down from the initial parallelism configuration when the workload was high. While this results in low latency, it comes at the cost of the auto-scaler over-provisioning the system extensively when the workload is low.

The optimal scale-down-factor As the scale-down factor should be set to let the auto-scaler respond correctly to a decreasing workload and not to correct an inherent flaw of the auto-scaler itself when scaling up, we prefer using the scale-down factor of 1.2 for the experimental evaluation. While the scale-down factor of 1.2 induces more latency than a lower factor, it reduces the average number of task-managers significantly. Furthermore, 1.2 was also the scale-down factor used by Floratou et al. [18] when evaluating Dhalion. For this reason, we use the scale-down factor of 1.2 for the experimental evaluation.

Conclusion

- Dhalion is slow in its response to increasing workloads. By scaling down slowly at the decreasing part of the cosine pattern, Dhalion can handle the increasing part better.
- The higher the scale-down factor, the more aggressively the system is scaled down. When scaling down too aggressively, an additional scale-up action is required to converge to the optimal parallelism.
- Setting a too-low scale-down factor results in over-provisioning of the system.
- The experimental evaluation has pointed out a scale-down factor of 1.2 to be an appropriate trade-off between preventing over-provisioning and reducing latency.

7.1.3. DS2

For DS2, we investigate the impact of the over-provisioning factor on the performance of the auto-scaler. The over-provisioning factor is introduced for DS2 to account for the lag that builds up when the auto-scaler is unable to process the incoming workload when scaling. DS2 is designed to suggest a parallelism configuration that allows the system to process the same amount of tuples per second as the input rate. This means that it is not designed to provide enough resources to also decrease the lag of the system, resulting in high latency. The over-provisioning factor allows for deploying resources on top of the resources suggested by DS2 to process additional records and reduce the lag of the system. To illustrate the need for the over-provisioning factor and show the danger of setting the over-provisioning factor too high, we run DS2 with an over-provisioning factor of 1.0, 1.2 and 1.4. The results are summarised in table 7.3 and 7.4, and graphs 7.3 and 7.4.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
DS2 1.0	6.5	15	311.8	317.5	554.6	23
DS2 1.2	9.4	15	39.4	29.3	110.5	23
DS2 1.4	10.3	15	11.7	3.6	41.7	22

Table 7.3: DS2 Configurations for query 1.

Performance analysis As seen in the tables, the average latency of DS2 without any over-provisioning is higher than 5 minutes for both queries 1 and 11. This is caused by the lag building up during scaling operations and never being reduced. When setting an over-provisioning factor of 1.2 or 1.4, more resources are used, resulting in significantly lower latency. The maximum number of task-managers for

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
DS2 1.0	9.4	30	388.5	388.9	588.9	26
DS2 1.2	13.2	30	284.1	316.3	499.2	26
DS2 1.4	15.8	33	160	137.2	405.5	26

Table 7.4: DS2 Configurations for query 11.

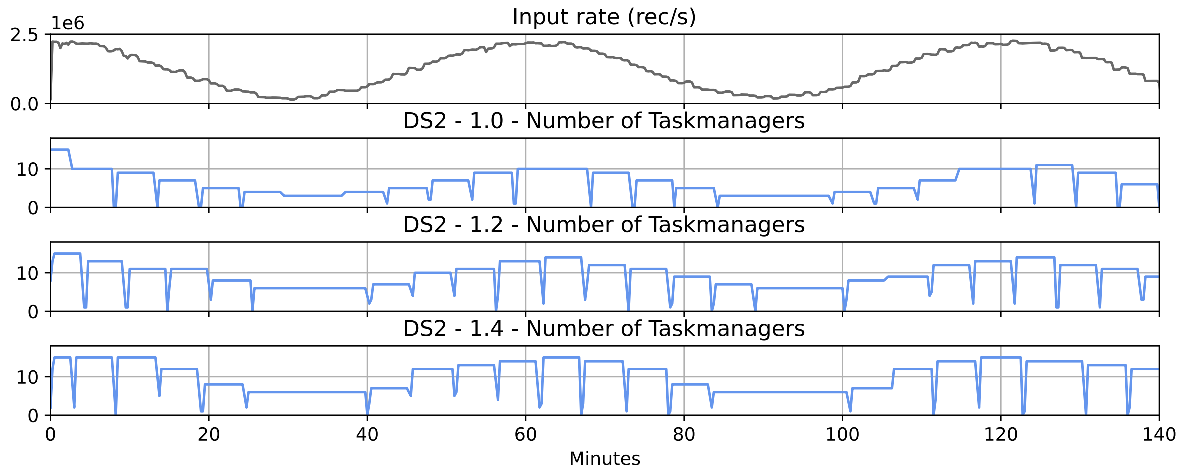


Figure 7.3: Task-managers deployed by DS2 with different scale-down factors when run on query 1 with the cosine workload pattern.

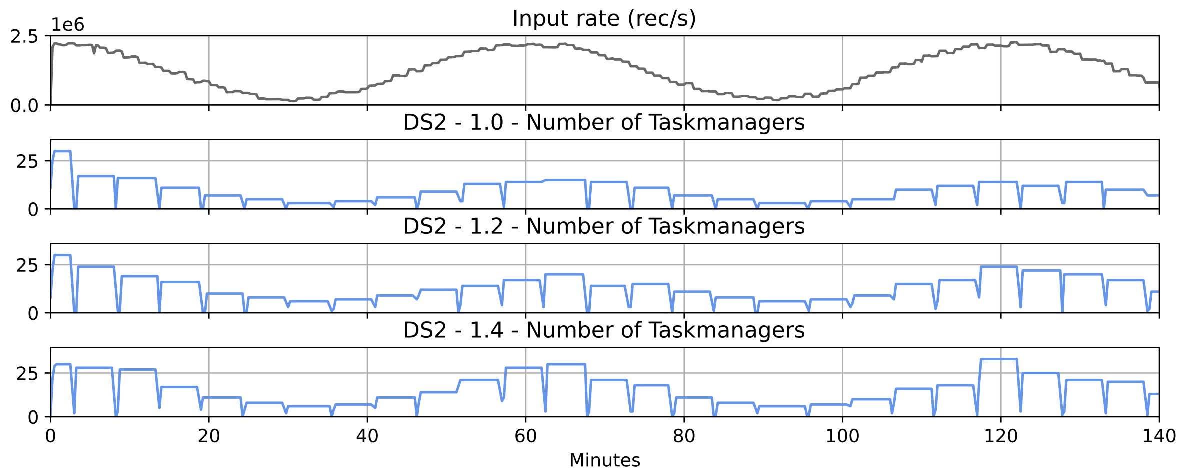


Figure 7.4: Task-managers deployed by DS2 with different scale-down factors when run on query 11 with the cosine workload pattern.

query 1 is 15 for all three configurations and in query 11 30 for both factors 1.0 and 1.2. The similarity in this metric is caused by the initial parallelism of the system from which the system is scaled down to never reach its initial parallelism again. When not considering the initial parallelism, the maximum parallelisms of the three configurations grow relative to the over-provisioning factor with configurations 1.0, 1.2, and 1.4 having in query 1 a maximum number of task-managers of 10, 13, and 15 respectively. Furthermore, as can be seen in figure 7.3, the scaling operations of all three configurations take place at the same time and only differ in the parallelisms the auto-scalers scale to. Furthermore, DS2 has trouble adjusting to the change in workload in the increasing parts of the cosine pattern. This explains the high average latency in query 11 for all three configurations. While DS2 with over-provisioning factor 1.4 is only using one or two additional task-managers on average in comparison with factor 1.2,

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
HPA-Varga 0.6	9.6	17	12	3.2	42.6	26
HPA-Varga 0.7	10.2	26	34.8	24.3	113.3	26
HPA-Varga 0.8	9.4	23	85.1	55.9	285.8	26

Table 7.5: HPA-Varga Configurations for query 1.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
HPA-Varga 0.6	-	-	-	-	-	-
HPA-Varga 0.7	26.5	69	219.3	244.1	379.8	26
HPA-Varga 0.8	-	-	-	-	-	-

Table 7.6: HPA-Varga Configurations for query 11.

it does have significantly lower latency than the other two factors. The reduction in latency in query 11 when using factor 1.4 is also much higher than in query 1. This can be explained by the scaling costs of statefull query 11 being more costly than in query 1.

The optimal over-provisioning factor In general, DS2 seems to function better with an over-provisioning factor of 1.4. This over-provisioning factor does, however, not only correct for the scaling overhead, but also corrects for the inaccuracy of DS2's suggested parallelism configurations. The parallelism configurations suggested by DS2 generally lead to under-provisioning, inducing high latency. As we do not want to correct for inherent inaccuracies of the auto-scaler logic itself, we prefer the over-provisioning factor of 1.2. Furthermore, an increase of 40% of resources is too much and not realistically accepted by SLOs. For these reasons, we use an over-provisioning factor of 1.2 for this experimental evaluation.

Conclusion :

- DS2 requires an over-provisioning factor to allow the system to process lag caused by scaling actions.
- When using an over-provisioning factor of 1.0, the system is unable to reach an input throughput equal to or larger than the input-rate of the system for both queries 1 and 11.
- An over-provisioning factor of 1.4 only requires a few more task managers on average while significantly reducing the latency of the system.
- We select 1.2 as the default configuration for DS2's over-provisioning factor as we do not want to correct for inaccuracies of the auto-scalers logic itself and because increasing the suggested resources by more than 20% is too much and not realistically acceptable by SLOs.

7.1.4. HPA-Varga

HPA-Varga has two metrics that it tries to keep at target value: relative-lag-change and utilisation. As discussed in section 4, when the relative lag change is larger than one, the system is unable to keep up with the input rate. When the relative lag change is smaller than one, the system can process both the input rate and the lag, eventually returning to a relative lag change of one when the lag is gone. For this reason, we set the relative-lag-change-target-value to one and do not consider other target values for the metric. Setting an appropriate value for the utilisation target is, however, a bit more complex. The utilisation metric corresponds to the time per second the task-managers spend processing records. The higher the utilisation-target-value, the faster the auto-scaler scales down the auto-scalers and the slower the auto-scaler scales up the auto-scalers. We investigate the performance of HPA-Varga with a utilisation-target-value of 0.6, 0.7, and 0.8. The results are summarised in table 7.5, 7.6, and graphs 7.5 and 7.6. Due to technical problems, we are unable to run HPA-Varga with parameters 0.6 and 0.8 on query 11. This leaves us with only a run with a utilisation target value of 0.7 for query 11, making it impossible to compare the behaviour of the three configurations on query 11. We leave this for future work.

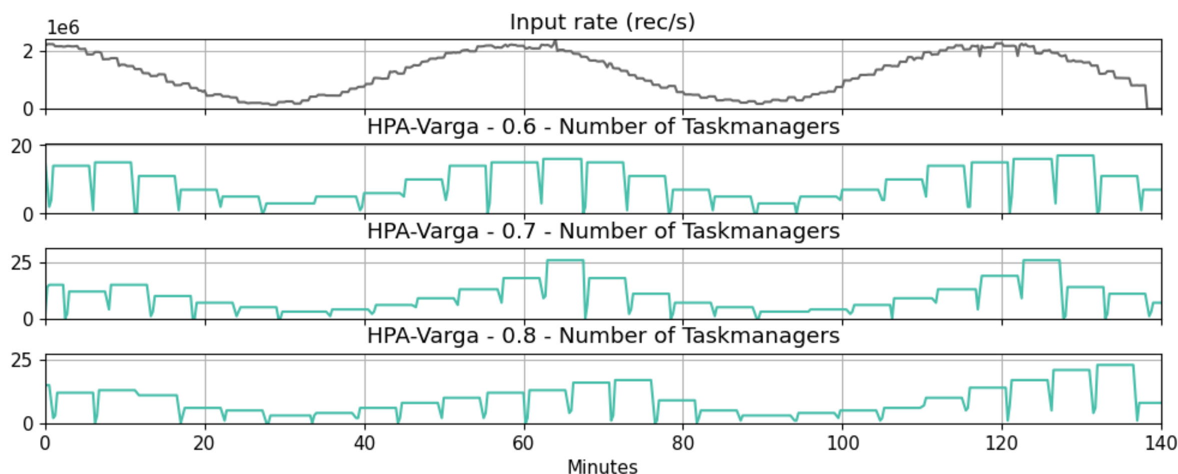


Figure 7.5: Task-managers deployed by HPA-Varga with different utilisation target values when run on query 1 with the cosine workload pattern.

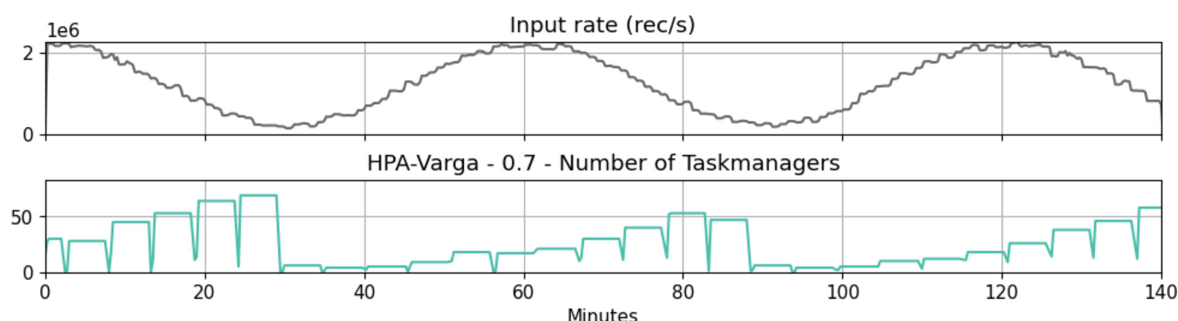


Figure 7.6: Task-managers deployed by HPA-Varga with 0.7 utilisation target values when run on query 11 with the cosine workload pattern.

Performance analysis The higher the utilisation target, the more the auto-scaler attempts to use the full capacity of the task-managers, scaling down faster and scaling up later. In the results of query 1, we see, however, that HPA-Varga with a utilisation target of 0.6 has the lowest latency and uses fewer resources than the runner-up of HPA-Varga with a utilisation target value of 0.7. After the input rate drops down in the first cosine period, the auto-scalers with a higher utilisation target value scale down faster than the auto-scaler with a lower utilisation target. As HPA-Varga with a utilisation target of 0.6 scales down slower than the other two auto-scalers, it handles the workload better when it starts rising again, scaling up quickly without letting the lag increase too much. The other two configurations do, however, scale up slower, increasing the lag. From these two configurations, HPA-Varga with a utilisation target of 0.7 can recover more quickly from this increase than the auto-scaler with a utilisation target of 0.8. Furthermore, while both auto-scalers save resources by scaling down faster than HPA-Varga with a utilisation target of 0.6 when the input rate is decreasing, HPA-Varga with a utilisation target of 0.7 eventually uses more resources when catching up with the input-rate when it is increasing. HPA-Varga with a utilisation target of 0.8 does save a bit more resources than HPA-Varga with a utilisation target of 0.6 but has significantly higher latency. In query 11, we only manage to get results for HPA-Varga with a utilisation target of 0.7. Here, we see the auto-scaler taking a long time to catch up when the workload of the system is increasing, after which the auto-scaler issues a large scale-down action that drops the auto-scaler to minimal parallelism. While this saves resources, it also results in the auto-scaler having trouble handling the input rate when it is increasing again, resulting in high lag. HPA-Varga with a utilisation target of 0.6 would scale down slower, using more resources when the input rate starts increasing again. This would allow the auto-scaler to handle the steep increase in workload better than HPA-Varga with a utilisation target of 0.7.

Optimal utilisation target value In the experimental evaluation, we find that the tendency of utilisation target 0.6 to maintain an over-provisioned state for a longer period of time enabled it to handle sudden increases in the workload more efficiently. This allows it to be both resource efficient and achieve low latency when run on the cosine pattern. HPA-Varga with utilisation target 0.7 shows to be more aggressive when scaling down the auto-scaler from an over-provisioned state. This reduces the resource consumption of the auto-scaler, but makes it more difficult for the auto-scaler to handle the input-rate when it starts increasing again. Still, if the workload remains low for a longer period of time or would increase slower than it does in the cosine pattern, the auto-scaler would be more resource efficient than the auto-scaler with utilisation target 0.6. Furthermore, a utilisation target of 0.6 sets the target of the auto-scaler to use only 60% of the resources available by the task-managers. While the utilisation target works well for the cosine pattern, the auto-scaler would when run on other workload-patterns tend to over-provision. As a utilisation target of 0.7 seems more reasonable for an auto-scaler to target, we choose to use the utilisation target of 0.7 for the experimental evaluation.

Conclusion

- The higher the utilisation target value, the faster HPA-Varga scales down and the slower it scales up.
- The utilisation target of 0.6 achieves low latency and is resource effective when run on the cosine pattern. By leaving the system in an over-provisioned state when the input rate is low, the system can handle the increasing parts of the cosine pattern better, resulting in low latency and more efficient resource usage.
- A utilisation target of 0.6 prefers the system to be in an over-provisioned state. While this results in a good performance on the cosine pattern, it is likely to over-provision when run on other workload patterns. For this reason, we prefer the more resource-efficient utilisation target of 0.7 for the experimental evaluation.

7.1.5. HPA-CPU

For HPA-CPU, we investigate the impact of setting a different CPU-utilisation-target-value. The higher the target value, the more efficient HPA-CPU will try to use the task-managers, issuing scaling-down actions faster and postponing scale-up actions. We investigate the performance of HPA-CPU with a *CPU-utilisation-target-value* of 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0. The results are summarised in table 7.7 and 7.8, and graphs 7.7 and 7.8.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
HPA-CPU 0.5	17	32	4	0.013	30.2	26
HPA-CPU 0.6	13.4	24	3.9	0.012	28.2	26
HPA-CPU 0.7	11.4	21	6.4	0.018	35.8	26
HPA-CPU 0.8	9.8	22	36	14.2	136.1	26
HPA-CPU 0.9	8.7	16	102	43.7	302.6	26
HPA-CPU 1.0	4.3	15	327.7	377.9	60.6	3

Table 7.7: HPA-CPU Configurations for query 1.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
HPA-CPU 0.5	23.9	49	30.2	0.029	141.8	26
HPA-CPU 0.6	19.5	53	107.8	52.8	318.9	26
HPA-CPU 0.7	14.2	34	178	146.9	399.9	26
HPA-CPU 0.8	11	30	254.9	294.3	488.1	25
HPA-CPU 0.9	8.8	30	296	328.9	539.9	20
HPA-CPU 1.0	7.4	30	339.1	385.1	582	7

Table 7.8: HPA-CPU Configurations for query 11.

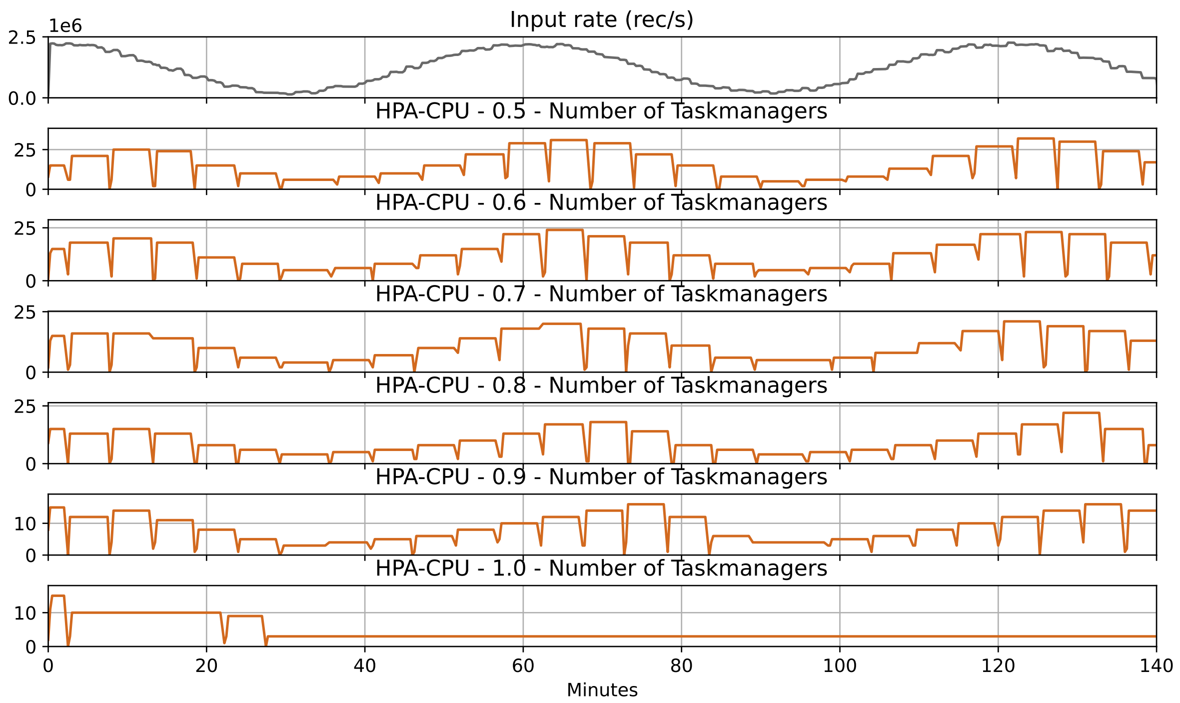


Figure 7.7: Task-managers deployed by HPA-CPU with different CPU-utilisation target values when run on query 1 with the cosine workload pattern.

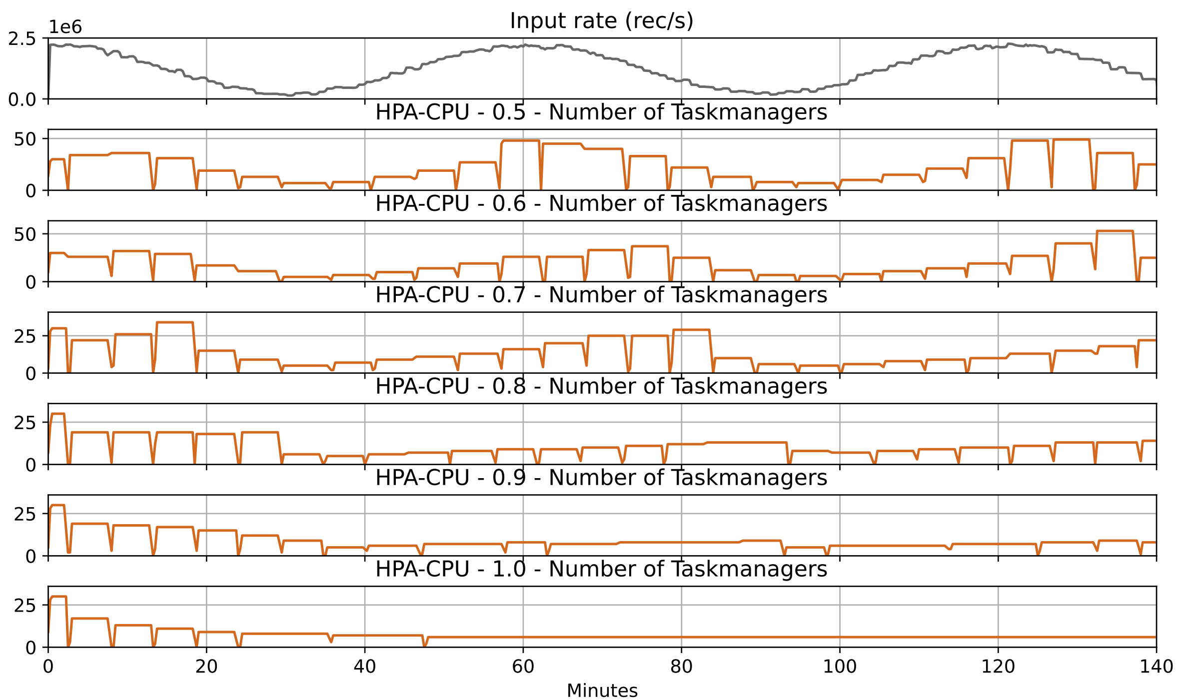


Figure 7.8: Task-managers deployed by HPA-CPU with different CPU-utilisation target values when run on query 11 with the cosine workload pattern.

CPU-utilisation target HPA-CPU scales the application to keep the CPU-utilisation metric close to the CPU-utilisation target. To achieve this, it scales more conservatively when the CPU-utilisation target is high and more aggressively when the CPU-utilisation target is low. This results in the general trend

of HPA-CPU with a high target value using few task-managers but having high latency. HPA-CPU with a low target value uses more task-managers but also achieves low latency. To find the optimal target value for both queries, we have to find the best trade-off between these two extremes.

Setting CPU-utilisation target value too high The first thing one might notice is that the target value of 1.0 has little scaling actions for queries 1 and 11. The auto-scaler scales down from its initial parallelism configuration towards a parallelism of 1 per operator and then stays there for the remainder of the experiment. This is also the case for the target value of 0.9 for query 11 where it uses few resources, and sometimes scales up or down an operator with a single task-manager. A reason for this is that the CPU load seems to max out around 90% utilisation. Every task-manager is assigned a single CPU, which it uses to maintain communication with the job-manager, run its application logic, run its garbage collection module, and process incoming records. As the CPU-utilisation metric only shows the percentage of the CPU that is being used for processing records, a full 100% utilisation can never be reached and the task-manager maxes out the maximum number of records it can process per second before this happens. The auto-scaler assumes, though, that the auto-scaler can process more workload and that the system should scale down the 100% CPU usage. For this reason, a target value smaller than 0.9 is desirable.

Setting the CPU-utilisation target value too low When comparing the remaining target values, we notice that the target value of 0.8 leads to under-provisioning, having significantly higher latency than the other configurations. For query 1, target values 0.5, 0.6, and 0.7 all have low latency with target-value 0.7 using the least amount of task-managers. Both 0.5 and 0.6 use more task-managers while reaching similar latency as the 0.7 target value, with the 0.5 target value even over-provisioning with more than 50%. For query 11, only the 0.5 configuration has low latency. When the input rate is going up, HPA-CPU fails to scale the auto-scalers sufficiently, resulting in under-provisioning and increasing the lag. A reason for this is that the scale factor is calculated by dividing the current value by the target value of the system. The lower the target value, the higher the scale factor can grow when the system is unable to keep up with the input rate. For this reason, only the target value of 0.5 can scale enough to keep up with the fast-increasing workload.

The optimal CPU-utilisation target value While the CPU-utilisation-target-value of 0.5 achieves as only target-value low latency in both query 1 and 11, it also leads to over-provisioning for query 1. Its low target value allows the auto-scaler to increase fast enough to manage the increasing workload of the cosine pattern. When the workload in the system is changing less rapidly, the low target-value results in over-provisioning, making the target value unsuitable as the default configuration in the experimental evaluation. Target-value 0.7 achieves a more appropriate trade-off between its latency and its resource efficiency. For query 1 it uses significantly fewer resources than the 0.5 target value while maintaining similar latency. It appears, however, unable to scale up fast enough to keep up with the fast-increasing workload in query 11. Still, the auto-scaler can recover from this sudden increase in workload, eventually reducing the lag in the system to zero. For this reason, we select the CPU-utilisation-target-value of 0.7 as the best all-round configuration to be used for the experimental evaluation.

Conclusion

- The higher the CPU-utilisation target, the fewer task-managers are deployed and the higher the latency.
- As the CPU-utilisation metric of Apache Flink does not include the processing time the task-managers require for its management and garbage collection, the value of the metric can never grow bigger than 90%. This makes the CPU-utilisation-target-value of 0.9 and above unsuitable as CPU-utilisation target values.
- Low CPU-utilisation-target-values can produce higher scaling factors than high CPU-utilisation-target-values, allowing it to scale faster.
- We select 0.7 as the default CPU-utilisation-target-value for HPA-CPU for the experimental evaluation.

7.1.6. Conclusion

To optimise the configurations of the auto-scalers for this experimental evaluation, the most significant configurations of each auto-scaler was compared. The auto-scalers ran on query 1 and 11 with the workload following the cosine pattern. For Dhalion, 0.8 is selected as optimal scale-down-factor. For DS2 1.2 is selected as over-provisioning factor. HPA-Varga 0.7 is selected as optimal utilisation-target-value and for HPA-CPU 0.7 is selected as optimal CPU-utilisation-target-value.

7.2. Workload Comparison

In this section, we investigate the performance of the auto-scalers when run on different workloads. The auto-scalers are run on the cosine-, random-, decreasing-, and increasing workload, when run on a stateless operator topology (query 1) and a statefull operator topology.

7.2.1. Cosine pattern

The Cosine workload is designed to test the scale-up behaviour and the scale-down behaviour of the auto-scalers. During the experiment, we run the auto-scalers on the cosine pattern as defined in 5.4.3. The cosine pattern starts at 2.2 million records per second, following the cosine pattern down to 200,000 records per second at its lowest point. The cosine period is set to 60 minutes and the experiment is run for 140 minutes. During the discussion of the results, we first comment on the performance of the different auto-scalers, after which we dive into more detail regarding the different bottlenecks of the auto-scalers to effectively scale the system. The results of the experiment are summarised in table 7.9 and 7.10. Furthermore, we visualise the task-managers and latency of the different auto-scalers when run on query 11 in graphs 7.9 and 7.10.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion	13.2	30	68.3	23.7	233.3	27
DS2	9.4	15	39.4	29.3	110.5	23
HPA-CPU	11.4	21	8.6	0.018	35.8	26
HPA-Varga	10.2	26	17	24.3	113.3	26

Table 7.9: Auto-scaler run on query 1 with cosine pattern.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion	22.8	51	130.7	60	353.3	27
DS2	13.2	30	284.1	316.3	499.2	26
HPA-CPU	14.2	34	178	146.9	399.9	26
HPA-Varga	26.5	69	219.3	244.1	379.8	26

Table 7.10: Auto-scaler run on query 11 with cosine pattern.

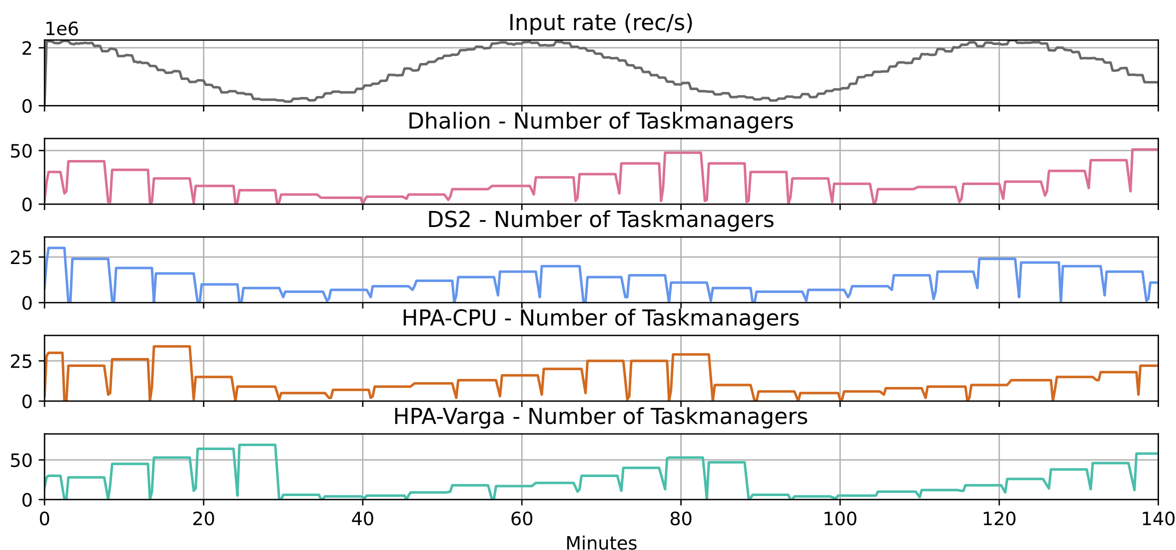


Figure 7.9: Task-managers deployed by the auto-scalers when run on query 11 with the cosine workload pattern.

Performance analysis There is a large difference in the performance of the auto-scalers in both queries 1 and 11. With HPA-CPU having low latency in query 1 without using a lot of additional task-

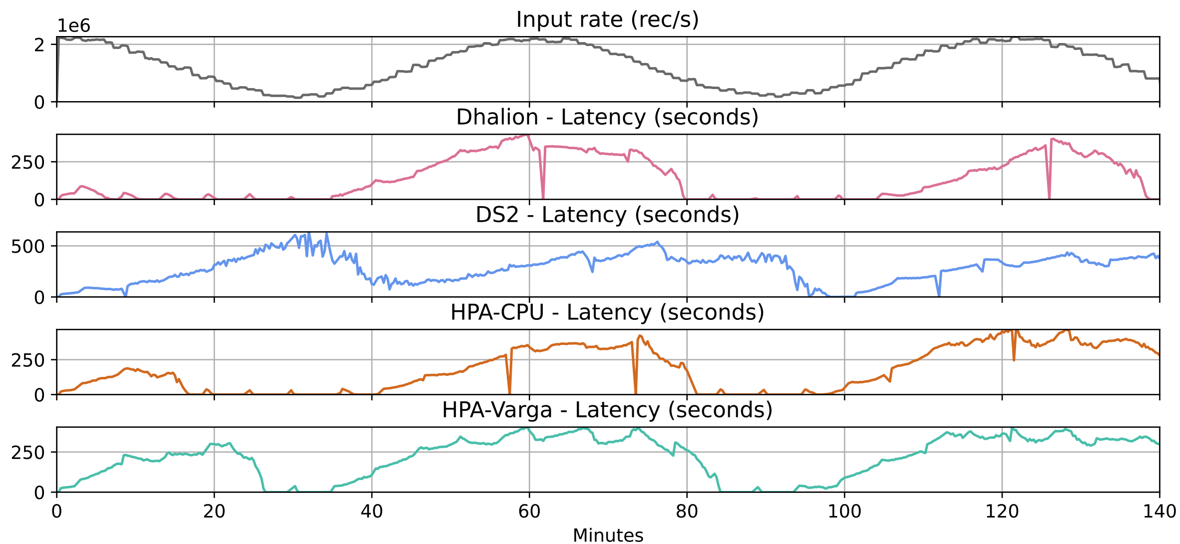


Figure 7.10: Average latency of the different auto-scalers when run on query 11 with the cosine workload pattern. As can be seen in the graph, all four auto-scalers have trouble keeping up with the workload when the input-rate is increasing, causing significant latency. Dhalion is the only auto-scaler able to maintain low latency when scaling up from the under-provisioned state at the beginning of the experiment.

managers, it is unable to keep up with the increasing workload in query 11. In query 1, Dhalion uses the most resources and achieves the highest latency. In query 11, it, however, achieves the lowest latency of all four auto-scalers in query 11. DS2 is the most resource-efficient auto-scaler for both queries 1 and 11, never issuing scaling operations that surpass the initial configuration of the system. In both queries it, however, is unable to keep up with the workload, resulting in high latency. HPA-Varga works well for query 1 but is unable to effectively scale the system to handle the workload in query 11, using the most amount of resources while achieving high latency.

Increasing workload The main bottleneck of the auto-scalers is the increasing part of the cosine pattern. While most of the auto-scalers can handle the increase in input rate for query 1, all four auto-scalers fail to scale up fast enough to manage the steep increase of the more intense workload of query 11. When this happens, Dhalion, HPA-CPU, and HPA-Varga successfully scale up the system until the Kafka queue is empty and the system is back in a healthy state. DS2, on the other hand, does not concern itself with the system's health and only scales the system based on the change in input rate. This causes the auto-scaler to scale down the system before it can reduce the latency, keeping the latency high.

Initial parallelism configuration In query 11, the experiment starts in an under-provisioned state. This causes Dhalion, HPA-CPU, and HPA-Varga to issue scale-up actions until the system is back in a healthy state. Considering the backpressure status of the system, Dhalion is the only auto-scaler that successfully identifies the operator that is being under-provisioned and scales it accordingly resulting in an immediate decrease in the system lag. HPA-CPU and HPA-Varga, on the other hand, require multiple scaling actions before they can find a parallelism configuration that can handle the systems workload and work away the lag.

Conclusion

- The main bottleneck for the auto-scalers is the steep workload increase of the cosine pattern. While none of the auto-scalers can keep up with the steep workload increase, HPA-CPU, HPA-Varga, and Dhalion keep scaling up the system until they work away the lag of the system.
- DS2 only bases its scaling operations on the input rate and does not consider the health of the system. Because of this, the auto-scaler fails to reduce the lag of the system, resulting in high latency.

- Considering the backpressure status of the system, Dhalion can identify and scale up the operator being under-provisioned in the system more effectively than any of the four operators.

7.2.2. Random

Next, we run the different auto-scalers on the random workload, as defined in section 5.4.3. This workload is designed to mimic a more real-world scenario, where the workload is difficult to predict. For this reason, we pay extra attention to the behaviour of the individual auto-scalers and discuss the correctness of their behaviour in certain situations. The pattern is generated from an initial input rate of 1,250,000 records per second. Every minute, the input rate is increased or decreased with a random number between zero and 500,000 records per second. As an additional constraint, we do not allow the input rate to surpass the maximum input rate of 2.5 million records per second. The experiments on query 1 start with an initial parallelism of 5 per operator and the experiments on query 11 start with an initial parallelism of 10 per operator. The results are summarised in table 7.11 and 7.11 and graphs 7.13, 7.12, 7.15, and 7.15. Due to technical constraints, we are unable to run HPA-Varga on the random workload for query 11 without crashing the system. For this reason, its results are left out.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion	14.9	33	76.1	29.7	337.9	27
DS2	9.4	16	71.8	41.9	223.2	26
HPA-CPU	11.1	24	20.1	0.125	155.6	26
HPA-Varga	10.4	31	51.4	24.5	215.3	26

Table 7.11: Auto-scaler run on query 1 with random pattern.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion	35.3	76	84.7	35.5	310.1	26
DS2	14.8	32	295.8	341.3	611.8	26
HPA-CPU	14.2	30	273.8	313.1	499.7	27
HPA-Varga	-	-	-	-	-	-

Table 7.12: Auto-scaler run on query 11 with random pattern.

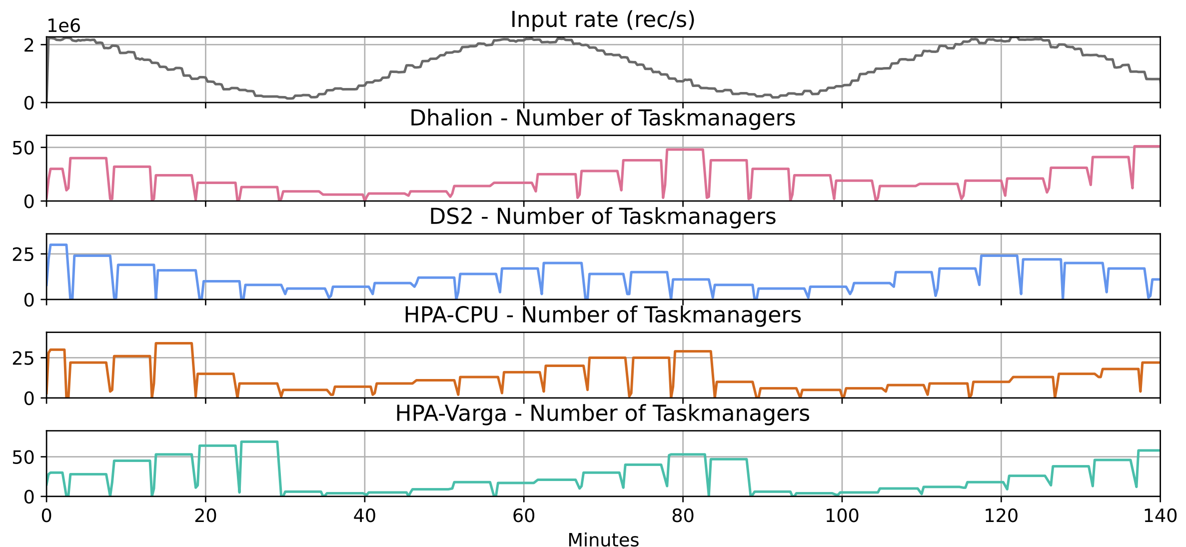


Figure 7.11: Task-managers deployed by the auto-scalers when run on query 11 with the cosine workload pattern.

Performance analysis For query 1, HPA-CPU achieves low latency without using many additional resources in comparison with DS2 and HPA-Varga. DS2 uses around 2 task-managers less than HPA-

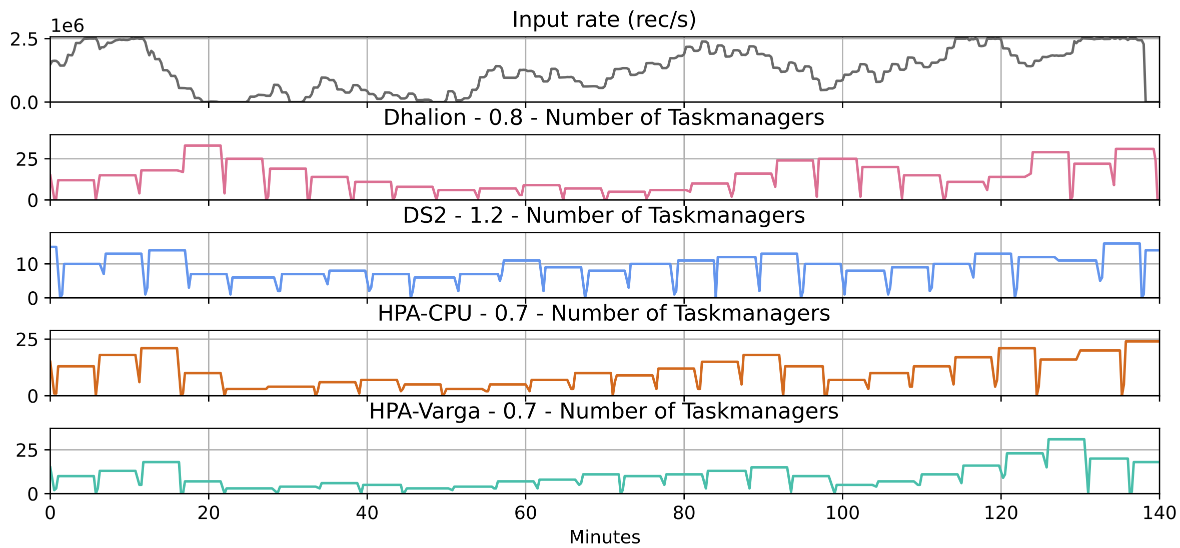


Figure 7.12: Average latency of the different auto-scalers when run on query 1 with the random workload pattern.

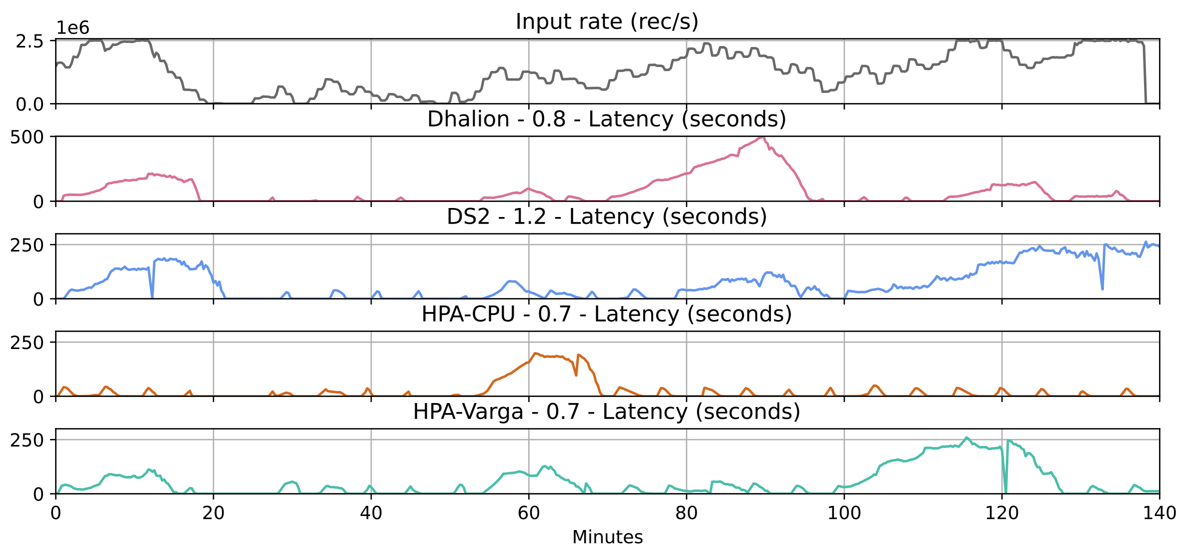


Figure 7.13: Task-managers deployed by the auto-scalers when run on query 1 with the random workload pattern.

CPU, achieving a high average latency of 71 seconds. HPA-Varga sits between DS2 and HPA-CPU in terms of both the average amount of task-managers and average latency. It does, however, use a maximum of 31 task-managers during the experiment, which is almost double the maximum number of task-managers used by DS2. Dhalion performs worse than the other three auto-scalers, using almost 50% more resources than the others while achieving high latency. For query 11, Dhalion also uses significantly more task-managers than DS2 and HPA-CPU. It does, however, manage to achieve as only auto-scaler a relatively low average latency. Both DS2 and HPA-CPU have an average latency of around five minutes, using less than half the task-managers Dhalion uses.

Under-provisioning detection The random load pattern starts in a slightly under-provisioned state, quickly increasing to the maximum input rate of 2.5 million records per second. This requires the auto-scalers to respond quickly with a scale-up action to prevent an increase in the lag of the system. HPA-CPU is the only auto-scaler that correctly responds to this, with the other auto-scalers issuing scale-down actions instead. Using the CPU utilisation of the task-managers, HPA-CPU can determine that the system is currently being under-provisioned and scales the operators accordingly. Because

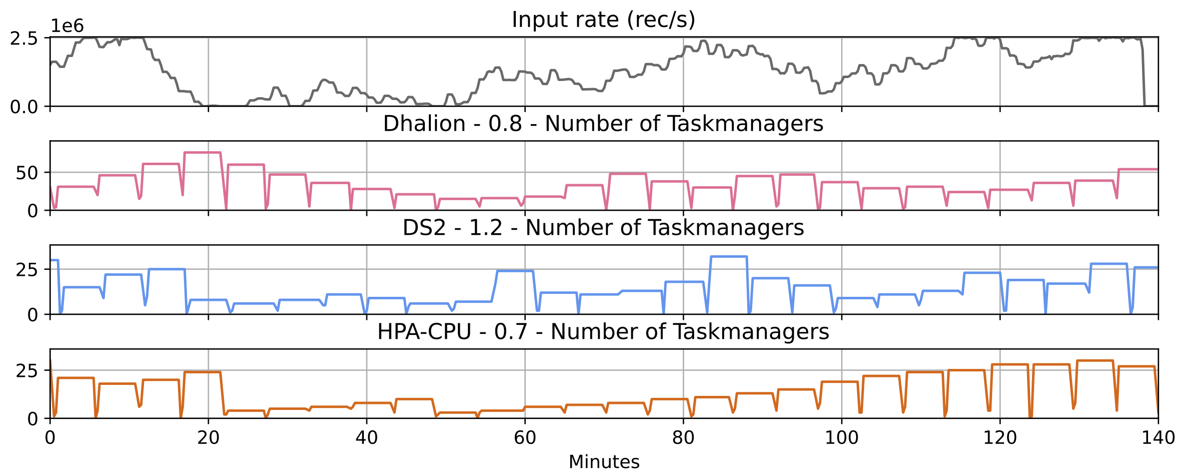


Figure 7.14: Task-managers deployed by the auto-scalers when run on query 11 with the random workload pattern.

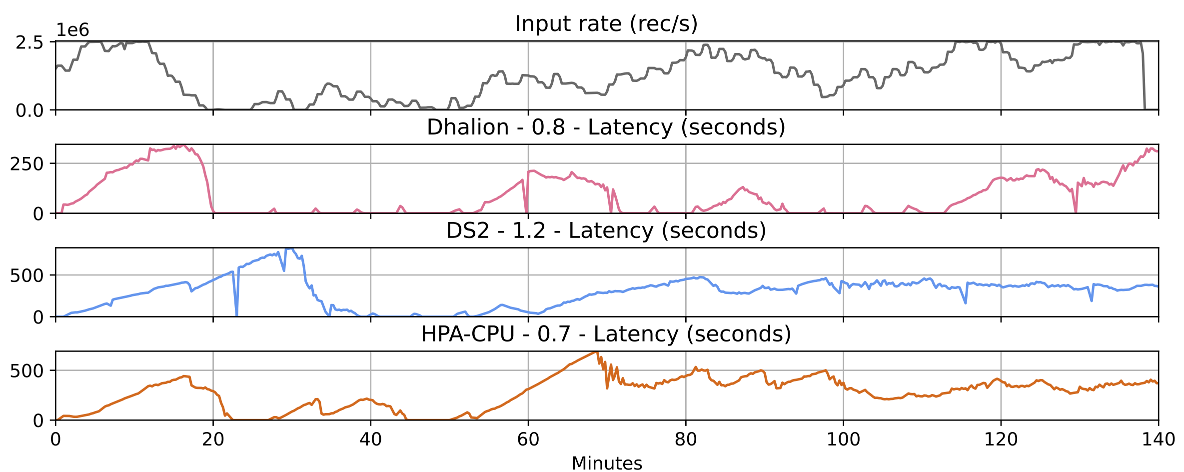


Figure 7.15: Average latency of the different auto-scalers when run on query 11 with the random workload pattern.

the experiment just launched, the Kafka queue is still almost empty and no backpressure is measured by the system. This causes HPA-Varga and Dhalion to not register the system as under-provisioned, issuing scale-down operations. This is different in query 11, where it takes more resources to process the same number of records. This results in back pressure being registered faster, allowing Dhalion to respond more quickly. HPA-CPU, Dhalion, and DS2 issue scale-up actions, with only Dhalion scaling up with enough additional resources to work away the lag of the system. Because of the way HPA-CPU calculates its scale factor, it is unable to produce a scale factor high enough that allows it to scale to the parallelism it requires, taking more time before it can work away the lag.

HPA-CPU behaviour HPA-CPU performs well on query 1 but fails to scale the system fast enough to keep up when the workload is increasing fast in query 11. As we discuss in the experimental evaluation of the Cosine pattern, HPA-CPU is unable to significantly increase the parallelism of an operator due to its way of calculating the scale factor. This results in HPA-CPU scaling up the operators too slowly, inducing significant latency in query 11. In query 1, the scale factors it uses are enough to scale operators, resulting in both low latency and low resource usage.

Dhalion behaviour In general, Dhalion reacts too slowly to changes in the input rate, taking nearly 20 minutes to reach the required parallelism configuration of the workload. While this is a common problem with reactive auto-scalers reacting only when the system is already unable to keep up with the workload, Dhalion has two additional issues. First, Dhalion uses the backpressure status of the

operator topology to determine the health of the system. As it takes time for backpressure to present itself in the system when the system is under-provisioned, Dhalion recognises under-provisioning late. Secondly, Dhalion only scales a single operator per scaling action, requiring multiple scaling actions. As scaling actions take a long time to execute, this causes Dhalion to respond late to workload changes. Still, Dhalion is designed to keep scaling up until the system is back in a healthy state. This ensures that given enough time, Dhalion will eventually catch up with the lag and recover from an unhealthy state. This persistence, in combination with its tendency to use rather high scale-up factors when scaling up, allows it as the only auto-scaler to recover from the second increase of workload in query 11.

DS2 behaviour Comparing the auto-scalers, DS2 seems to be able to adjust faster and more flexibly to the change in input rate in both queries 1 and 11. Still, it is almost constantly under-provisioned only catching up with the lag when the input rate is close to zero. When the input rate goes down, DS2 tends to almost immediately issue a scale-down action, no matter the current state of the system. This results in the system being almost constantly under-provisioned and having high latency.

HPA-Varga behaviour While only being able to run HPA-Varga on query 1, we can see that the auto-scaler is able to keep the system in a relatively healthy state, while not using too many task-managers. The scaling actions of HPA-Varga appear quite similar to the scaling-actions of HPA-CPU, scaling down to low parallelism in only 2 scaling actions when the workload drops down and also having trouble scaling-up fast enough to keep up with fast increasing workload. HPA-Varga tends to scale the application more conservative than HPA-CPU, resulting in it using less resources. This also leaves the system in an under-provisioned state for a longer amount of time, causing backpressure and high latency.

Conclusion

- For query 1, HPA-CPU can achieve low latency while using similar resources as HPA-CPU and HPA-Varga and fewer resources than Dhalion.
- Dhalion is the only auto-scaler that can keep up with the steep increase in workload in query 11. When doing so, it uses more than twice the resources as the other auto-scalers
- HPA-CPU correctly identifies the under-provisioned state at the start of query 1, keeping latency low. It is, however, unable to scale up fast enough in query 11 to keep up with the steep increase in workload at the beginning.
- Using the backpressure status, Dhalion fails to identify under-provisioning at the beginning of query 1. Still, its large scale-up actions allow it to recover quickly from this mistake.
- While performing well on query 1, HPA-CPU increases the parallelism in query 11 by too little, resulting in high latency.
- Dhalion is less resource efficient than the other auto-scalers. This enables the auto-scaler to keep up with steep workload increases when other auto-scalers are unable to.
- DS2 can respond fast and flexibly to changes in the input rate but fails to consider the current state of the system, scaling down before it can process the lag.
- The similarity of the CPU-utilisation metric and the utilisation metric results in similar behaviour between HPA-Varga and HPA-CPU. They both scale down fast in response to a decrease in workload but fail to scale up fast enough in response to fast-increasing workloads. From the two, HPA-Varga seems to scale up more conservatively, using fewer resources, but having higher latency.

7.2.3. Increasing workload

The increasing workload pattern is designed to investigate the scale-up functionality of the auto-scalers. The pattern is predictable and requires the auto-scalers to carefully time their scaling actions. The pattern starts at zero input rate and goes up over a period of 140 minutes until it reaches 2,500,000 records per second. As there is initially no data to process, the system starts at an initial parallelism of

1 per operator. We run the experiment on queries 1 and 11. The performance metrics are summarised in table 7.13 and 7.2 and graphs 7.16 and 7.17. Due to technical constraints, we were unable to successfully run HPA-Varga on the increasing workload for query 11. For this reason, it is left out of the results.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion	15.3	44	44.8	6.5	169.4	27
DS2	9.1	15	30	0.033	158	13
HPA-CPU	10.7	21	4.8	0.017	32.2	24
HPA-Varga	8.4	16	10.9	2.3	38.7	20

Table 7.13: Auto-scaler run on query 1 with increase pattern.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion	28.8	67	98.4	44.5	329.2	27
DS2	12.8	30	246.9	313.5	385.2	26
HPA-CPU	17.7	44	125.5	117.3	324.5	26
HPA-Varga	-	-	-	-	-	-

Table 7.14: Auto-scaler run on query 11 with increase pattern.

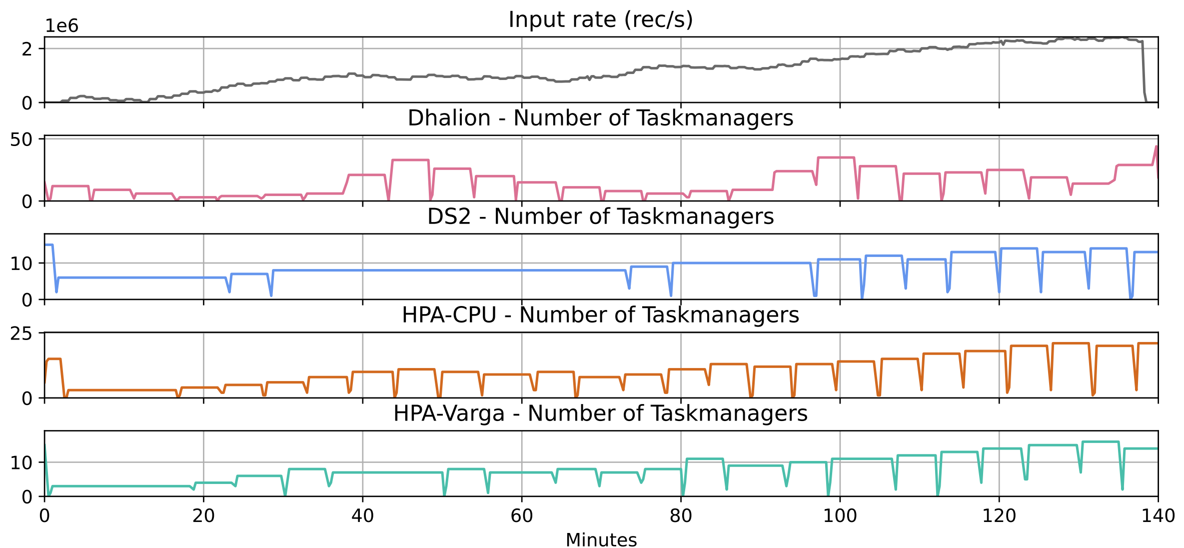


Figure 7.16: Task-managers deployed by the auto-scalers when run on query 1 with the increasing workload pattern.

Performance analysis - Query 1 With an average latency of 4.8 seconds, HPA-CPU achieves the lowest average latency for query 1. For this, it uses slightly more task-managers than HPA-Varga which has, as runner-up, an average latency of 10.9 seconds. HPA-Varga uses the least amount of task-managers for query 1, using a maximum of 16 task-managers. While the 50th percentile of the latency of DS2 is low, its average latency is high. This is caused by the auto-scaler starting to oscillate when the input rate reaches 2 million records per second, inducing scaling overhead and causing the latency to go up. DS2 has significantly fewer scaling actions than the other auto-scalers, only scaling four times in the first 90 minutes. Dhalion has a high average latency and uses 50% more task-managers than HPA-CPU. It notices the under-provisioning of the system too late, scaling up when the latency is already high. Furthermore, after scaling up and working away the lag, it spends the next 50 minutes scaling down from the over-provision state, using more resources than it needs.

Performance analysis - Query 11 Running on query 11, Dhalion achieves the lowest average latency of 98.4 seconds while using almost twice as many task-managers as the other auto-scalers. HPA-CPU

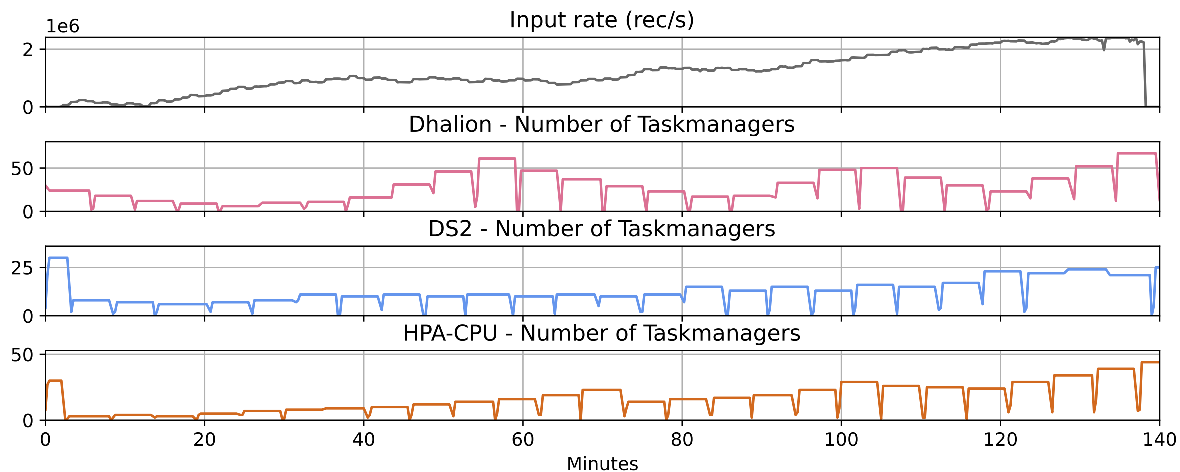


Figure 7.17: Task-managers deployed by the auto-scalers when run on query 11 with the increasing workload pattern.

comes in second with an average latency of 125.5 seconds, using on average 17.7 task-managers. DS2 uses the least amount of task-managers, scaling down from the initial configuration of parallelism 30 and never reaching it again. DS2 is, however, almost constantly under-provisioning, resulting in an average latency of above four minutes. While DS2 only required 13 scale actions for query 1, it here uses just as many as Dhalion and HPA-CPU. All three auto-scalers respond too late to the increasing workload, causing high latency in all three of them. From this, DS2 is never able to recover, maintaining high latency throughout the experiment. By deploying more task-managers Dhalion can recover faster from this than HPA-CPU. During the experiment, both auto-scalers repeat the cycle of under-provisioning, scaling up to work away the lag, and then slowly scaling down until it is under-provisioning again.

Conclusion

- HPA-CPU achieves low latency in query 1 while using slightly more task-managers than HPA-Varga.
- Until the 100-minute mark, DS2 only issues four scaling actions while keeping up with the input rate. After this point, DS2 starts oscillating, increasing the latency of the system.
- Dhalion, DS2 and HPA-CPU have high latency and do not scale the system in time in query 11, causing high latency.
- Dhalion uses more than twice the task-managers on average, achieving significantly lower average latency.

7.2.4. Decreasing

The decreasing workload pattern is designed to investigate the scale-down functionality of the auto-scalers. The pattern is predictable and requires the auto-scalers to carefully time their scaling actions. The pattern starts at 2,500,000 records per second and decreases over a period of 140 minutes until it reaches an input rate close to zero. We start the experiment with an initial parallelism of 5 when run on query 1 and 10 when run on query 11. The performance metrics are summarised in table 7.15 and 7.16 and graphs 7.18 and 7.19. Due to technical constraints, we were unable to successfully run both Dhalion and HPA-Varga on the decreasing workload for query 11. For this reason, it is left out of the results.

Performance analysis - query 1 When run on query 1, HPA-CPU has the lowest latency of the auto-scalers using 3 more task-managers than the runner-up HPA-Varga. HPA-Varga uses the least amount of task-managers while achieving an average latency of 24.2 seconds. Dhalion uses more task-managers than any of the auto-scalers, achieving an average of 44.4 seconds. DS2 uses a similar

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion	16.7	38	44.4	20	182.7	27
DS2	10.1	16	127.5	148.6	260.3	24
HPA-CPU	13.7	23	6.4	0.017	36.8	21
HPA-Varga	10	18	24.2	11.9	92.8	22

Table 7.15: Auto-scaler run on query 1 with decrease pattern.

Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
Dhalion	-	-	-	-	-	-
DS2	14.4	30	343.8	354	429.1	26
HPA-CPU	23.8	49	107.9	40.4	323.6	23
HPA-Varga	-	-	-	-	-	-

Table 7.16: Auto-scaler run on query 11 with decrease pattern.

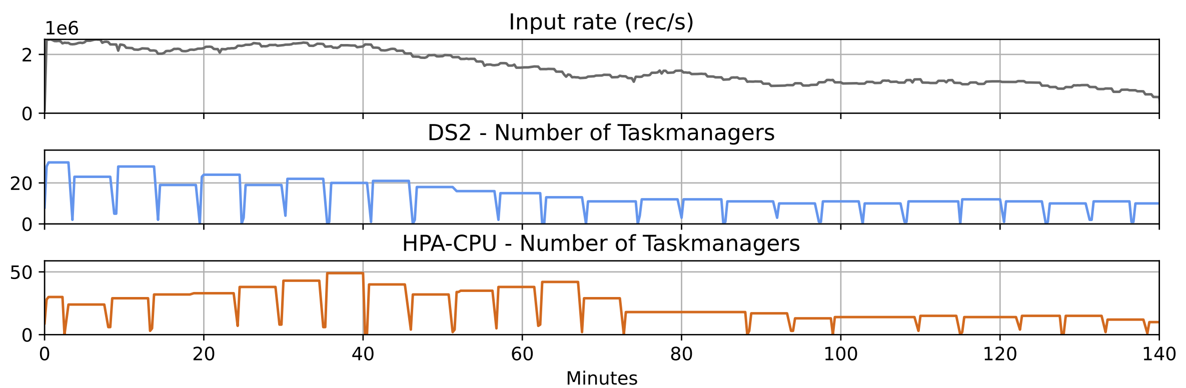


Figure 7.18: Task-managers deployed by the auto-scalers when run on query 1 with the decreasing workload pattern.

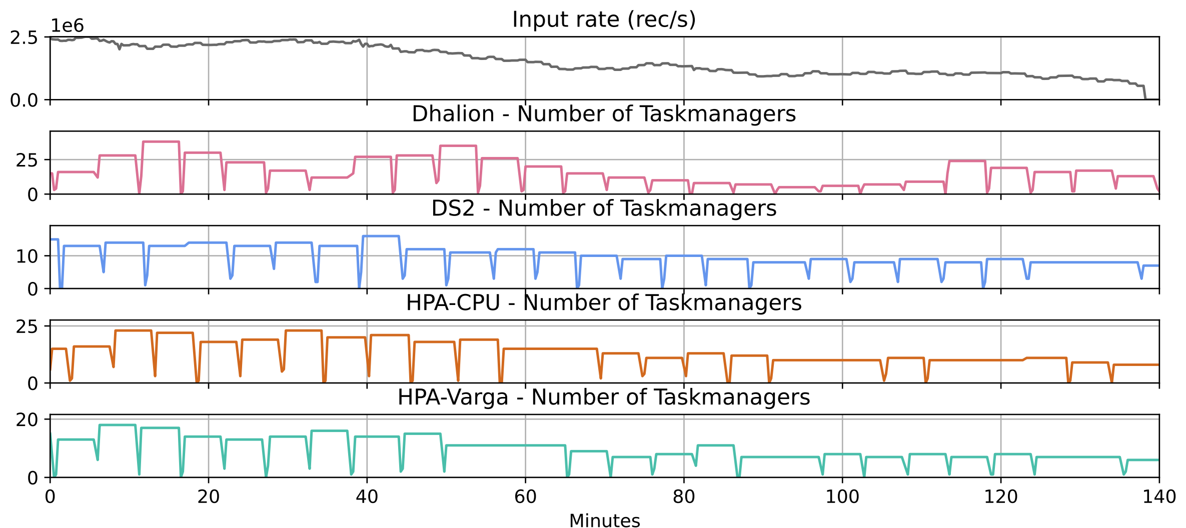


Figure 7.19: Task-managers deployed by the auto-scalers when run on query 11 with the decreasing workload pattern.

amount of task-managers as HPA-Varga but has an average latency of over two minutes. The decrease pattern starts in an under-provisioned state, requiring the auto-scalers to scale up to prevent extensive increase in the latency. Both HPA-CPU and HPA-Varga do this effectively, reducing the latency to

zero in 10 minutes. Dhalion is late to respond to the under-provisioned state, allowing the latency to grow bigger and requiring more time and resources to work away the lag. DS2 fails to recognise the system to be in an under-provisioned state, only scaling down from the initial parallelism. This causes high latency for the first 100 minutes of the system, eventually catching up with the lag when the input rate has dropped down to around 1 million records per second. In the first half of the experiment, the input-rate remains quite stable around 2.5 million records per second. Here, HPA-CPU and HPA-Varga auto-scalers seem to be oscillating, failing to find the optimal parallelism configuration. During the experiment, Dhalion fails to distinguish optimal-provisioning from over-provisioning, issuing scale-down operations every time it finds the system to be in a healthy state, eventually causing under-provisioning.

Performance analysis - query 11 In query 11, DS2 uses significantly fewer task-managers than HPA-CPU but also achieves a lot higher average latency. HPA-CPU uses 10 more task-managers than DS2 on average, even increasing the total number of task-managers to 49. This results, however, in an average latency of 107.8 seconds. The 50th percentile of the application, however, has a latency of 40.4 seconds, which is still quite acceptable for the system. Just as with query 1, DS2 fails to recognise the system to be in an under-provisioned state at the beginning of the experiment, never working away the lag and keeping high latency during the experiment. HPA-CPU does recognise the system to be in an under-provisioned state but fails to scale up the application enough to handle the input rate. Only when the input rate drops down below 1 million records per second, HPA-CPU can work away the lag and maintain low latency.

Conclusion

- Both HPA-CPU and HPA-Varga achieve low latency while using a few resources in query 1. HPA-CPU has low latency while using more resources than HPA-Varga. HPA-Varga is more resource-efficient, having a higher average latency.
- DS2 fails to recognise the system to be in an under-provisioned state at the beginning of the experiments. It is constantly under-provisioning, never working away the initial lag and causing high latency throughout the entire experiment.
- Dhalion is unable to determine the difference between over-provisioning and perfect-provisioning. It keeps scaling down resources until it reaches an under-provisioned state, triggering scale-up actions again leading the application to an over-provisioned state where it works away the lag. From here, it repeats the process and starts scaling down again until it reaches an under-provisioned state.
- For query 11, HPA-CPU fails to scale up the application enough to handle the input rate, only catching up with the lag halfway through the experiment, resulting in high latency.

7.3. Query Comparison

In this section, we investigate the performance of the auto-scalers when run on different queries. The queries consist of different operator topologies containing different operators that all have different resource requirements. We investigate how well the auto-scaler estimates the resources needed to process the input rate and how they respond to this in terms of run-time behaviour. As discussed in section 5.4.1, we implement and run the experiments on queries 1, 2, 3, 5, 8, and 11 from the Apache Beam Nexmark Suite [6]. Queries 1 and 2 are simple stateless queries that perform a mapping and a filter operation on the incoming records. Queries 3 and 8 are both join operations that require saving incoming records for joining with incoming records. In query 3 we configure the system to save the records for a maximum of five minutes after which the records are deleted. Query 8 uses a tumbling window that defines the period in which records must arrive to be allowed to join. Queries 5 and 11 both aggregate over the bids stream by counting the occurrences of unique auction and person IDs. Query 5 does so over a sliding window, and query 11 does so over a session window. The experiments are run on the cosine workload pattern. As some of the queries require significantly more resources than others when processing records, we use different parameters for the cosine pattern. The parameters used for the cosine pattern per query are shown in table 7.17. The results of the experimental runs are summarised in table 7.18.

Query	Min. input rate	Mix. input rate	Max. random noise
1	200,000	2,200,000	100,000
2	200,000	2,200,000	100,000
3	3,000	33,000	1,500
5	10,000	110,000	7,500
11	200,000	2,200,000	100,000

Table 7.17: Input rates used when run on different queries with the workload following the cosine workload pattern.

Technical problems When running the experiments on query 8, we found that the job-manager kept crashing during scaling. When scaling the system, the job is stopped and task-managers are instructed to save their state by creating checkpoints. These checkpoints are then used by the new job to retrieve the state of the previous job and to continue processing. In query 8, scaling operations issued by the auto-scalers often result in task-managers failing to create a checkpoint. This causes the job to fail and prevents the SPE from processing the incoming records. While the auto-scalers often succeed in restarting the job later in the experiment, the failing job has a significant impact on the results of the experiments. For this reason, we have decided to keep the query out of the experimental evaluation of the system and leave it for future work.

7.3.1. Queries 1 and 2

Both queries 1 and 2 are stateless queries consisting of three operators that require little processing power to process their incoming records. By setting the input-rate high, we can still stress the system and require active scaling to keep the system both resource efficient and have low latency. As can be seen in the table, the results of both queries are rather similar. For both queries 1 and 2, HPA-CPU and HPA-Varga achieve low latency while using not too many task managers. From these two auto-scalers, HPA-Varga is more resource efficient but achieves higher latency than HPA-CPU. Dhalion uses the most resources in both queries 1 and 2 and at the same time achieves the highest latency. As can be seen in graph 7.1, Dhalion scales up slowly when the input rate is rising in the cosine pattern. This increases the latency, causing Dhalion to scale up even further to work away the lag. When scaling down after working away the lag, Dhalion is again slow in scaling down, keeping the system in an over-provisioned state for a long time. DS2 is the most resource-efficient in queries 1 and 2. It issues fewer scaling actions for both queries but keeps the system in an under-provisioned state when the input rate is high. As DS2 tends to be quite conservative when scaling, the proposed configurations lead in this case to slight under-provisioning, resulting in high latency. With its initial parallelism being the maximum number of task managers deployed in both queries 1 and 2, DS2 proposes less high parallelism configurations than the other auto-scalers.

Query	Auto-scaler	$avg(TM)$	$max(TM)$	$avg(lat)$	$Pctl_{50}(lat)$	$Pctl_{95}(lat)$	$Scale_{no}$
1	Dhalion	13.2	30	68.3	23.7	233.3	27
1	DS2	9.4	15	39.4	29.3	110.5	23
1	HPA-CPU	11.4	21	8.6	0.018	35.8	26
1	HPA-Varga	10.2	26	17	24.3	113.3	26
2	Dhalion	13.2	29	87.4	15.4	314.7	27
2	DS2	8.9	15	40.5	36.3	122.2	21
2	HPA-CPU	10.7	19	8.6	0.021	38.9	26
2	HPA-Varga	8.9	19	17	11.2	56.9	26
3	Dhalion	16.1	50	1.7	0.015	14.3	27
3	DS2	-	-	-	-	-	-
3	HPA-CPU	7.8	50	73.7	41.2	275.5	23
3	HPA-Varga	11.1	50	72.1	44.5	205.4	24
5	Dhalion	8.9	30	11	0.023	56.2	26
5	DS2	23.2	68	81	53.9	245.9	26
5	HPA-CPU	5.3	30	18.6	0.95	67.8	23
5	HPA-Varga	5.8	30	60.9	39	198.9	24
11	Dhalion	22.8	51	130.7	60	353.3	27
11	DS2	13.2	30	284.1	316.3	499.2	26
11	HPA-CPU	14.2	34	178	146.9	399.9	26
11	HPA-Varga	26.5	69	219.3	244.1	379.8	26

Table 7.18: Results of auto-scaler runs on operator topologies of Apache Beam Nexmark Queries 1, 2, 3, 5, and 11.

7.3.2. Query 3

Query 3 is a statefull join operation that requires the system to save all incoming records in state for at least five minutes. As this data has to be transferred to the new topology when scaling, this process may be quite expensive. As this requires considerable more resources to process the incoming records, we lower the input-rate for query 3 significantly to make it possible to run the query on our designated server. This also reduces the workload of the system, lowering the overall resource requirements.

7.3.3. Queries 5 and 11

Query 5 and 11 are both a three operator based topologies that count unique IDs provided by the Bid stream. Query 5 uses a five minute sliding window that fires a new window every minute and query 11 uses a session window that ends a session after ten seconds of inactivity. In practice, this results in significantly larger windows for query five, requiring more resources from the system. To allow query 5 to be run on our designated server, we use a lower input-rates for query 5. We use the same input-rates as query 1 and 2 for query 11.

Query 5 For query 5, both Dhalion and HPA-CPU achieve low latency. While the average latency of Dhalion is slightly lower than that of HPA-CPU, HPA-CPU uses almost half the task-managers as Dhalion. HPA-Varga uses a similar number of resources as HPA-CPU but has significantly higher latency. HPA-CPU assigns on average 1 or 2 task-managers to each operator. In comparison to HPA-Varga, it scales up faster when the input rate is increasing, allowing the system to process the increasing input rate. HPA-Varga scales up too late, increasing the latency. Dhalion also scales up in time to prevent the latency to increase when the workload is increasing. The scaling actions issued by Dhalion are larger than those of HPA-CPU, causing over-provisioning and resulting in higher resource usage. While DS2 scales more conservatively in queries 1, 2 and 11, resulting in low resource usage in these queries, it uses significantly more resources for query 5. Investigations into the scaling actions issued by DS2 reveal the auto-scaler issues large parallelisms for the sink operator of the topology. The parallelism of the source operators and the count operator are throughout the entire experiment set between 1 and 5, while the parallelism of the sink is set much higher, even once having a parallelism of 62. Because the sink operators require few resources, this is extreme over-provisioning. The high

parallelisms of the sink operator are caused by the busy time of the operator becoming zero, resulting in a division by zero in the DS2 module, returning infinite as desired parallelism. Adding an additional check to ensure no zero is passed to the DS2 module would fix this problem. Furthermore, DS2 is often under-provisioning the other two operators, causing the auto-scaler to have high latency.

Query 11 The high input rate used in query 11 requires the auto-scalers to deploy more resources than the other queries. To keep up with the workload and prevent over-provisioning, the auto-scalers must issue larger scale-up actions when the input rate is increasing and larger scale-down actions when the input rate is decreasing. While HPA-CPU and HPA-Varga manage to scale down fast to prevent over-provisioning, they fail to scale up fast enough when the input rate is increasing resulting in high latency. Dhalion scales down too slowly the third time the workload is decreasing, resulting in over-provisioning. This over-provisioning state does, again, help the auto-scaler to scale up fast enough when the input rate is increasing, resulting in significantly lower latency. While DS2 can scale rather flexibly and issues large scaling operations, it is almost constantly under-provisioning resulting in high latency. The long cool-down period may be cause all four auto-scalers to have trouble keeping up with the increasing workload. To prevent oscillation and to let the system recover from the increase in lag after the scaling action, we disable scaling for a total of five minutes. As the parallelism configuration is set using the information before the scaling action, the auto-scalers are cannot respond to changes in the workload during this period. When the input rate is increasing as fast as in query 11, the parallelism configuration issued when scaling may be under-provisioning during the cool-down period. To prevent this, auto-scalers may issue larger scaling actions to prevent this scenario from happening. While especially HPA-CPU manages to do this in query 1, it is still not enough in query 11, leading to under-provisioning.

7.3.4. Conclusion

In this section, the performance of the auto-scalers when run on different queries has been compared. The results show that HPA-CPU and HPA-Varga are resource efficient and achieved low latency when run on queries 1, 2, 3, and 5. Here, HPA-Varga uses less resources but also has a higher latency than HPA-CPU. However, both auto-scalers were unable to scale fast enough to keep up with the steep workload increase in query 11. This causes the auto-scalers to be under-provisioned for a long time, resulting in high latency. DS2 is often the most resource efficient in comparison with the other auto-scalers. It responds quick to changes in the workload, barely over-provisioning when the workload is decreasing. However, DS2 does often issue scale down operations while the latency of the system is still high. This causes the auto-scaler in having the highest latency in query 3, 5, and 11, with only Dhalion having higher latency in query 1 and 2. In query 5, a busy time of 0 of the sink operator causes DS2 to suggest high parallelism for the sink operator. This makes the auto-scaler almost three times as resource inefficient as the other auto-scalers. In all five queries, Dhalion uses significantly more resources than the other auto-scalers. This is caused by Dhalion scaling down slowly, resulting in over-provisioning. By over-provisioning on the decreasing part of the cosine pattern, Dhalion is able to better handle the steep increase in workload when the input-rate is going up again. Hence, it allows Dhalion to have the lower latency in query 11 than the other auto-scalers.

Autoscaler	Query 1				Query 11			
	Stage 1		*Stage 2*		*Stage 1*		*Stage 2*	
	$Time_{min}$	$Scale_{no}$	$Time_{min}$	$Scale_{no}$	$Time_{min}$	$Scale_{no}$	$Time_{min}$	$Scale_{no}$
Dhalion	40+	-	40+	-	-	-	-	-
DS2	40+	-	11	3	40+	-	40+	-
HPA-CPU	40+	-	9	2	40+	-	40+	-
HPA-Varga	40+	-	40+	-	-	-	-	-

Table 7.19: Converge times of auto-scalers on convergence workload for query 1 and 11. Stage one represents the part of the experiment where the input-rate is set to 2 million records per second. Stage two represents the part of the experiment where the input-rate is set to 1 million records per second.

7.4. Convergence Time

For the convergence time evaluation, we investigate the time it takes for the auto-scalers to converge to a specific parallelism configuration when the input rate is constant. For this, we use the convergence workload pattern that consists of three stages. The experiment starts at stage zero with an input rate of zero minutes, allowing the auto-scalers to start up. Then, the input rate is increased to 2 million records per second and stays there for a total of 40 minutes (stage 1). After this, we reduce the input rate to 1 million records per second and keep it there for the remaining 40 minutes of the experiment (stage 2). As defined in section 2.5.2, we say an auto-scale converges when it stops suggesting scaling actions on a constant input rate. For both input rates, we measure the time it takes before the auto-scaler converges and the number of scaling operations it requires. By investigating the throughput of the system and the average CPU utilisation under the chosen parallelism configuration, we comment on the accuracy of the proposed configuration. The results are summarised in table 7.19 and graphs 7.20 and 7.21. Due to technical constraints, we are unable to run both Dhalion and HPA-Varga for query 11. We, therefore, leave it out of the results.

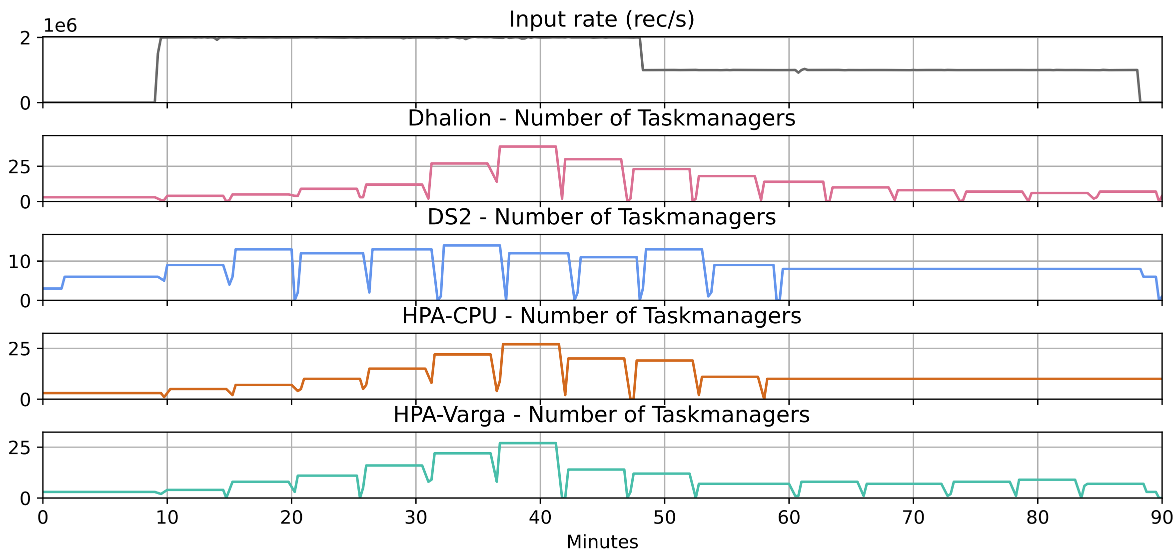


Figure 7.20: Online task-managers during convergence experiment on query 1.

Convergence time In the table, we see that only DS2 and HPA-CPU can converge in stage 2 of query 1. DS2 takes 11 minutes and 3 scaling actions to converge and HPA-CPU takes 9 minutes and 2 scaling actions to converge. This suggests that HPA-CPU can converge faster to an appropriate parallelism configuration than DS2. Both HPA-Varga and HPA-CPU are unable to converge in any of the experiments. One reason for this, which is also suggested in the experimental evaluation of Kalavri et al. [36], is that both auto-scalers scale a single operator at a time, requiring multiple steps to scale all operators accordingly. Furthermore, as the operators are linked together, changing the paral-

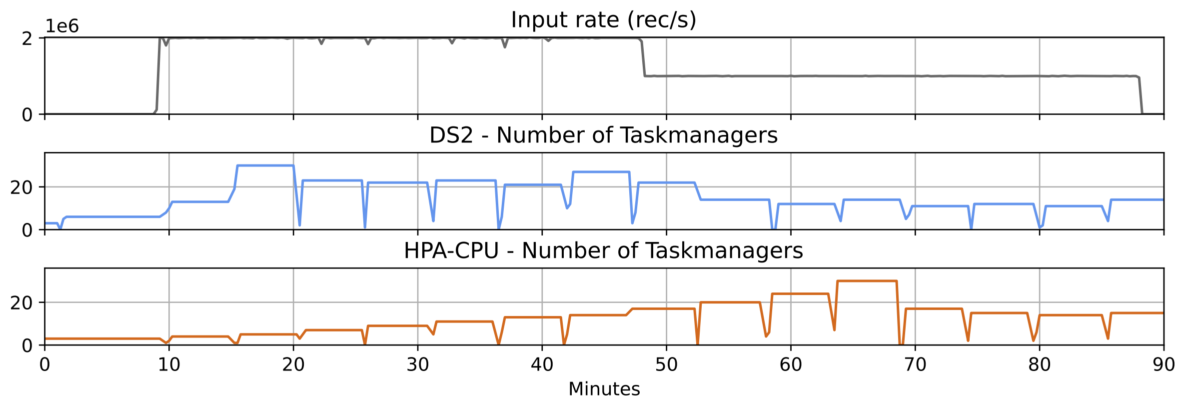


Figure 7.21: Online task-managers during convergence experiment on query 11.

lelism of a single operator may cause other auto-scalers to become under-provisioned, requiring even move-scaling actions. DS2 and HPA-CPU, on the other hand, determine for each operator individually whether to scale them. This allows them to converge to a specific configuration faster than HPA-Varga and Dhalion.

Prediction accuracy The parallelism configuration suggested by DS2 requires a total of 9 task-managers, while HPA-CPU requires 10 task-managers. After working away the lag, both HPA-CPU and DS2 have a throughput of 1 million records per second, showing that their final configuration is indeed enough to manage the input rate. The fact that DS2 achieves the same throughput as HPA-CPU while using fewer resources, shows that HPA-CPU is over-provisioning. When converged, HPA-CPU has an average CPU-utilisation value of 56%. DS2 has an average CPU-utilisation value of 65%. While the averages also include the CPU utilisation of the sink-operator which is generally low and cannot get scaled down further, it indicates that both auto-scalers are likely still over-provisioned. Future investigation into the optimal parallelism under different input rates will provide further insights into the accuracy of the auto-scalers.

Query 1 - Converging in stage 1 None of the auto-scalers succeeds in converging to the input rate of 2,000,000 records per second in stage one. Dhalion, HPA-CPU and HPA-Varga first try to reduce the lag in the system, after which they scale down to converge to the input rate. Due to the initial slow start of the auto-scalers and because the auto-scalers scale up rather slowly, they take a long time to scale up enough so they can handle both the input rate and reduce the lag at the same time. Only after the lag is gone, the auto-scalers scale down converging to the input rate. As this process takes longer than 40 minutes, none of these three auto-scalers manages to converge in stage 1. DS2, on the other hand, can scale more flexibly and scales up to appropriate parallelism in only a few scale actions. This parallelism, however, is not enough for the auto-scaler to reduce the lag in the system. This results in the busy time metric continuously shifting value, causing changes in the true input- and output rates measured by the auto-scaler. Together with the additional lag caused by scaling, this causes the auto-scaler to oscillate, never converging.

Query 1 - Converging in stage 2 In stage two, the auto-scalers start in an over-provisioned state, scaling down to the optimal parallelism. From this state, DS2 and HPA-CPU manage to converge within 2 and 3 scaling actions. Dhalion and HPA-Varga, on the other hand, fail to do so. Dhalion is designed to scale down by 20% when the system is in a healthy state. Starting at a parallelism of 25 task-managers, it takes many scaling actions for Dhalion to reach a parallelism configuration for the new workload. HPA-Varga can scale down rather fast and finds a parallelism configuration where it stays for roughly 10 minutes. After this, it starts oscillating, repeatedly scaling up and down some of its operators. One reason for this might be the utilisation metric of the source-operators of the system changes frequently triggering scaling actions.

Query 11 Both HPA-CPU and DS2 fail to converge when run on query 11. As the workload of query 11 is higher than that of query 1, it requires the auto-scalers to scale up the operators more. While DS2 can do this successfully, HPA-CPU scales up too slowly, increasing the lag of the system. The auto-scaler keeps scaling up until it has processed the lag of the system, only starting to scale down when it is already in stage two of the experiment for 20 minutes. The remaining 20 minutes are not enough for HPA-CPU to converge. While DS2 can scale up faster than HPA-CPU, it also fails to converge in both stage 1 and stage 2. Its parallelism configuration is unable to reduce the lag of the system, causing the auto-scaler to oscillate.

Conclusion

- In query 1, only HPA-CPU and DS2 manage to converge in stage two of the experiment. When converged, both auto-scalers have a throughput similar to the input rate. To achieve this, DS2 uses slightly fewer resources than HPA-CPU. Their average CPU-utilisation metrics suggest both DS2 and HPA-CPU are over-provisioning.
- All four auto-scalers are unable to converge in stage one of the experiment within 40 minutes. Dhalion, HPA-CPU, and HPA-Varga require a longer period to converge as they try to first work away the lag before converging to a lower parallelism configuration. DS2 manages to scale up faster than the other auto-scalers but fails to work away the lag. This causes instability in the busy-time metric of the operator, causing the auto-scaler to oscillate.
- None of the auto-scalers succeed in converging in query 11. HPA-CPU scales up too slowly, taking an hour to recover from the high input rate of stage 1. DS2 manages to scale up faster but oscillates in both stage 1 and stage 2, never converging.

7.5. Overview of the results

In this section, an overview of the experimental evaluation of Dhalion, DS2, HPA-Varga, and HPA-CPU is provided. First, the different experiments and their results in terms of latency, resource efficiency, convergence time and accuracy are evaluated. Thereafter, the run-time behaviour of the individual auto-scalers and their strengths and weaknesses are discussed.

7.5.1. Experimental evaluation

For the experimental evaluation, the experiments were divided into four parts, which are parameter optimisation, a performance analysis of the auto-scalers when run on different workload patterns, a performance analysis of the auto-scalers when run on different queries, and a convergence time analysis. The different experiments are discussed in the paragraphs below.

Parameter optimisation As the auto-scalers have several parameters that have to be set appropriately for optimal performance, we have run and compared the auto-scalers with different parameters set for their most significant configurations. As we do not have the resources to optimise the parameters for every experimental run, the auto-scalers were run on a cosine workload on two queries that are presentable for the other queries and compared with their resource efficiency and latency. Using the results, the best parameters for the auto-scalers were selected and used in the remaining experiments. By comparing the performance of the different parameters settings, we choose a scale-down-factor of 1.2 for Dhalion, an over-provisioning factor of 1.2 for DS2, a utilisation target of 0.7 for HPA-Varga, and for HPA-CPU a CPU utilisation target of 0.7.

Workload comparison Using these parameters, the performance of the auto-scalers was compared to the random, cosine, increasing, and decreasing workloads that were defined in section 5.4.3. The results show that the auto-scalers struggle with rapid increases in the workload which are found in the increasing parts of the cosine and random workloads. We hypothesise that this is caused by the long cool-down period we issue after scaling, preventing auto-scalers to scale even when the system is being under-provisioned. For all four of the workloads, HPA-CPU and HPA-Varga have the best performance in terms of latency and resource usage when run on query 1. HPA-Varga is more resource efficient than HPA-CPU, while HPA-CPU achieves lower latency. However, the auto-scalers struggle to scale up fast

enough when run on query 11, resulting in high latency. This is especially a problem when running on the cosine and random workload patterns, as both of them include steep workload increases. Dhalion on the other hand achieves both high latency and high resource usage on all four of the workload patterns. Nevertheless, by over-provisioning on the periods with low workloads, Dhalion can handle sudden increases in workload on the cosine and random pattern better compared to other auto-scalers. On both the increasing and decreasing workload, Dhalion struggles to find the correct scaling factor, spending most of its time recovering from inaccurate scaling decisions. Furthermore, DS2 is more resource efficient than the other auto-scalers and has a fast response to changes in the input rate. When doing so, DS2 does not consider the size of the Kafka queue, which leads to it scaling down even when the system is back-pressured. This results in high latency. Additionally, DS2 performs well on the decreasing workload, issuing only a few scaling actions and steadily following the trend of the workload pattern. However, the suggested parallelism configurations are under-provisioning, resulting in high latency. On the decreasing workload pattern, DS2 struggles to correct for the initial under-provisioned state at the beginning of the experiment, causing the auto-scaler to oscillate.

Query comparison The impact of the different query configurations on the performance of the auto-scalers has been investigated. As the different queries have different operator topologies that have different resource requirements, we have investigated how well the auto-scalers estimate the resources needed to process the input rate and how they respond to this in terms of run-time behaviour. The results show that when run on queries 1, 2, and 5, HPA-CPU and HPA-Varga perform well with HPA-Varga being more resource-efficient and HPA-CPU having lower latency. However, both auto-scalers do struggle with the more resource-intense queries 3 and 11, having high latency in both of them. Except for query 5, DS2 is more resource efficient compared to other auto-scalers. When run on query 5, a division by zero error causes the auto-scaler to issue large scaling operations on the sink operator. DS2 is generally under-provisioning and often fails to work away the lag of the system resulting in high latency. Dhalion uses significantly more resources compared to the other auto-scalers on all five queries. While the slow scaling of Dhalion often results in high latency, Dhalion can handle the fast increase in the workload of the more resource-intense queries 3 and 11 better. This results in low latency in both queries.

Convergence time The convergence time of the auto-scalers has been investigated by running them on the convergence workload pattern on both queries 1 and 11. The results indicate that only DS2 and HPA-CPU manage to converge on the second part of the experiment on query 1. Here, DS2 uses fewer resources than HPA-CPU while both achieving a throughput equal to the input rate. Moreover, the CPU utilisation of both auto-scalers is low, indicating that both auto-scalers are over-provisioning. Further analysis of the performance of different parallelism configurations on the constant input rates would provide better insights into the accuracy of the auto-scalers. In the other experiments, all four auto-scalers are unable to converge to a parallelism configuration within the 40-minute period. This is because the auto-scalers have to work away the lag before they can converge to a parallelism configuration. As this takes a long time, the auto-scalers are unable to converge in time.

7.5.2. Auto-scaler performance

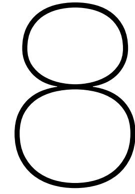
Four state-of-the-art auto-scalers targeting SPEs were evaluated and compared during different experiments. The results of these experiments provide insights into the run-time behaviour of the auto-scalers. In this section, the auto-scalers are individually discussed and evaluated based on their strengths and weaknesses.

Dhalion The scaling mechanism of Dhalion causes the auto-scaler to be slow to scale up and down. This causes under-provisioning when the input rate goes up and over-provisioning when the input rate goes down. While this generally leads to bad performance, the over-provisioning state does help Dhalion to handle sudden increases in workload. Dhalion uses back-pressure to estimate the health of the system. As back-pressure is measured when the system is already under-provisioning, scaling actions performed by Dhalion are already too late. However, the back-pressure status does allow Dhalion to accurately determine the bottleneck of the system, which issues well-targeted scaling actions. Hence, during most of the experiments, Dhalion appears to be resource inefficient and has high latency.

DS2 DS2 scales more flexibly, having its scaling actions directly reflect the change in input rate. However, when issuing scaling actions, DS2 does fail to consider the health of the system when making scaling decisions, often scaling down before it can reduce the latency. Furthermore, the parallelism configurations proposed by DS2 are often under-provisioning, which further increases the latency. In general, DS2 is resource efficient when scaling but often has high latency. It also issues fewer scaling actions, settling on a parallelism configuration faster than the other auto-scalers.

HPA-Varga HPA-Varga can issue large scale-down actions resulting in a fast reaction to decreasing workloads. However, it has a slow response to increasing workloads, eventually scaling up to an over-provisioned state to work away the lag. In general, the auto-scaler is resource efficient and can correctly estimate the appropriate parallelism configuration when the workload is not increasing too fast. When it is increasing fast, the auto-scaler is often too late in issuing scale-up actions, resulting in high latency.

HPA-CPU HPA-CPU is resource efficient and has low latency when the input rate is not increasing too much. When the workload is increasing fast, it has the same problems as HPA-Varga and fails to scale up fast enough to prevent the latency from increasing. In general, its performance is comparable with HPA-Varga, with HPA-Varga being a bit more resource-efficient and HPA-CPU having lower latency. By using the CPU utilisation, HPA-CPU can recognise under-provisioning before back-pressure is detected, allowing it to scale up faster than Dhalion. While the auto-scaler is not aware of the underlying structure of the operator topology, it can detect operators that are over- or under-provisioning and scale them accordingly.



Discussion and future work

8.1. Discussion

In this thesis, an extensive study of state-of-the-art auto-scalers targeting SPEs has been performed, which shows that auto-scalers targeting SPEs are poorly evaluated using simple experiments with unrealistic workload patterns. As a result, no comparison of the auto-scalers' performance can be made, making it difficult to determine the state-of-the-art. This slows down the development of auto-scaling solutions. In this thesis, we develop and implement a modular framework that can evaluate state-of-the-art auto-scalers targeting SPEs. Using this framework we evaluate and compare the performance of auto-scalers Dhalion [18], DS2 [36], HPA-Varga [66], using HPA-CPU [38] as a baseline.

The framework Five types of experiments are identified to evaluate the performance of auto-scalers targeting SPEs, which are performance analysis, parallelism accuracy, convergence time, prediction accuracy, and overhead estimation. Additionally, latency, resource efficiency, throughput, and resource usage have been identified as the four most common metrics used for evaluating the performance of the auto-scalers. Hence, the framework is designed to provide an end-to-end evaluation of the auto-scalers that can analyse and compare the auto-scalers' performance, converge time and parallelism accuracy. Latency and resource efficiency are used to measure the performance of the auto-scalers, and throughput and CPU usage are used to measure the parallelism accuracy of the auto-scalers. Furthermore, a convergence workload pattern was implemented to investigate the auto-scalers' convergence time. The framework is designed to be modular and easily extendable with additional auto-scalers, workload patterns, and queries.

The auto-scalers State-of-the-art auto-scalers Dhalion [18], DS2 [36], and HPA-Varga [66] have been implemented and compared using Kubernetes build-in horizontal pod auto-scaler [38] targeting CPU-utilisation as a baseline. As HPA-Varga and HPA-CPU cannot scale operators individually, their logic was adapted to support operator-based scaling. Furthermore, the metrics used by the auto-scalers are mapped to the metrics available in Apache Kafka and Apache Flink, making them compatible with the framework. The auto-scalers were implemented in a modular fashion, dividing the specialised auto-scaler logic from the rest of the framework and maintaining common functionalities in shared modules. This allows for fast implementation of additional auto-scalers and makes it easy to use the auto-scalers in different environments.

Experimental evaluation Using the framework, four different experiments, starting with a parameter optimisation to optimise the parameters of the auto-scalers, were performed to evaluate the performance of the selected auto-scalers. The auto-scalers were run on different workload patterns following a cosine, a randomly generated, an increasing and a decreasing workload pattern and on Apache Beam Nexmark queries 1, 2, 3, 5 and 11. The results show good performance of HPA-CPU and HPA-Varga on runs with slow workload increases, with HPA-Varga being more resource-efficient and HPA-CPU having lower latency. When the workload increases rapidly, the queries scale up slowly, causing high

latency. DS2, on the other hand, is resource efficient and has its scaling actions directly reflect the changes in input rate. However, it fails to consider the health of the system by scaling down before it can reduce the lag. Furthermore, its parallelism configurations are often under-provisioning, increasing the latency. Finally, Dhalion reacts slowly to workload changes, causing over-provisioning when the input rate decreases and under-provisioning when the input rate increases. Using the back-pressure status of the system, the auto-scaler can identify and scale bottleneck operators consistently, ensuring the system eventually returns to a healthy state. In the convergence time experiments, only DS2 and HPA-CPU succeed to converge in a single experiment. In the other experiments, the auto-scalers were unable to converge within the 40-minute period. The parallelism configurations of DS2 and HPA-CPU both allow the system to achieve the same throughput as the input rate. However, both of them have low CPU utilisation, indicating that both auto-scalers are over-provisioning.

8.2. Limitations

Paper selection When investigating auto-scalers targeting SPEs, mainly papers from 2017 and 2018 were considered, only using two papers from the period after 2019. While this can be explained as a decrease in attention to this field of research, it can also be caused by a bias in the paper selection method. The paper selection was based on an initial Google Scholar search, after which papers were selected from their related work sections. From these papers, auto-scalers were selected on the received attention and their adaptability of the proposed solution to different queries and workloads. This resulted in a bias for older papers. More extensive research into the developments of auto-scalers targeting SPEs over the past several years may provide better insights into the evaluation methods of these auto-scalers and can help better tailor the framework to the needs of the more recent works.

Parameter optimisation For the experimental evaluation, the auto-scalers' parameters were optimised by comparing their performance on queries 1 and 11 with the workload following the cosine pattern. Using the results, the most appropriate parameters were selected and used for all experiments of the experimental evaluation. While we argue that these two experiments are representative of the rest, there might be different configurations than the selected ones that can achieve better performance for specific experimental settings. However, the selected configurations have an overall good and representative performance for each auto-scaler. We recognise the need for exhaustive parameter optimisation to improve the confidence of our results. Although the long run-times of the experiments and the volume of the required runs make this infeasible in the scope of this project, we urge future research to include such optimisation experiments.

Scaling limitations To prevent SPEs from crashing during scaling, the change in parallelism is limited to 15 per operator. While DS2 and HPA-CPU do not scale by this large amount, Dhalion and HPA-Varga occasionally proposed larger scaling actions. As this affects the results of these experiments, performing the experiments on a cluster that does not need these restrictions might improve the accuracy of the results.

Statistical significance As the experiments are influenced by many factors and are only run once due to time constraints, we cannot claim that the results of the experiments are statistically significant. Running the experiments again may provide different results and lead to different conclusions. Overall, we do recognise specific patterns of the auto-scalers allowing us to comment on their specific behaviour. Still, without running the experiments multiple times, we are unable to quantify the statistical significance and cannot comment on the possible variance between multiple similar experimental runs.

8.3. Future work

Different input rates To run the experiments on our designated cluster, we used lower input rates for the more resource-intensive queries during the performance analysis. As this makes it more difficult to estimate the actual resource intensity of the queries, the effect of more resource-intense queries on the performance of the queries is difficult to compare. Running the experiments with the same input rates allows for better comparison between the performance of the auto-scalers and may provide additional insights into the effectiveness of the auto-scalers on different operator topologies.

Convergence experiments In the convergence experiments, only auto-scalers DS2 and HPA-CPU managed to converge within the 40-minute period. As most of the auto-scalers require more time to converge, increasing the period length would allow more auto-scalers to converge, providing better insight into the convergence time of the auto-scalers. The parallelism configurations the auto-scalers converged to can also be used to investigate the parallelism accuracy of the auto-scalers using systems throughput and CPU usage. By running the system on different parallelism configurations, the optimal parallelism configuration under specific input rates can be determined. This information can then be used to determine the parallelism accuracy of the parallelism configurations more accurately.

Proactive auto-scalers In the experimental evaluation, we find that many of the auto-scalers react too slowly to workload changes, causing resource inefficiency and high latency. This is an inherent flaw of reactive auto-scalers, only responding to workload changes when the system is already failing to process the workload. Over the past several years, the focus of research has shifted more towards developing proactive auto-scalers (like PASCAL [44], or Doan et al. [12]). These auto-scalers base their scaling decisions on predicted future workloads. This allows them to respond earlier to workload changes, already having the resources ready when the workload starts increasing. While we were unable to include proactive auto-scalers in the experimental evaluation because of their long training time, doing so in future work will provide better insights into their behaviour and may help in investigating the trade-off of these two approaches.

Production environments The goal of the framework is to provide the necessary tools for an end-to-end evaluation of the auto-scalers under different workloads and queries. While this is a step up from the current state-of-the-art for auto-scaler evaluation, it still does not fully capture the auto-scalers behaviour in production-like environments. These systems run for a much longer time, have much larger states, and deal with less predictable workloads that may include data skew. Working towards an evaluation framework that can capture the challenges of real-time production environments, would provide more valuable insights into the performance of the auto-scalers and would assist the development of auto-scalers for these settings.

DS2 improvements DS2 does not consider the systems lag and suggests parallelism configurations that are unable to process both the input rate and the lag. In this work, we corrected this by using an over-provisioning factor. However, the over-provisioning factor may not always accurately capture the influence of the scaling overhead throughout the experiment, which influences the results of the performance analysis. Furthermore, extending the auto-scalers logic to consider the scaling-overhead when proposing scaling actions may result in better results and a more accurate representation of the auto-scaler. Beside this, DS2's suggested parallelism configurations generally lead to under-provisioning. This is caused by DS2 incorrectly assuming that when assigning a task-manager enough workload, the busy time of the task-managers can increase to 100%. In practice, the back-pressure mechanism slows down the record intake before that can happen, leaving DS2 to constantly overestimate the true-processing rate of the operators. Calculating the true-processing rate by considering a better estimation of the maximum busy time will result in more accurate parallelism configurations, better overall performance, and a more fair representation of the DS2 auto-scaler.

HPA-CPU and HPA-Varga improvements Both the CPU-usage used by HPA-CPU and the utilisation of HPA-Varga are limited in terms of the maximum value they can reach. When the system is under-provisioning, the utilisation metric and CPU-usage increase to its maximum value. As the scale factor of HPA is calculated under the assumption that the control value can go up or down indefinitely, no matter how much further the workload increases, both metrics will result in the same scale actions. This causes both HPA-CPU and HPA-Varga to scale up too slowly when the workload is increasing fast. Incorporating the limited ranges of the metrics into the scale-factor calculation may remove this limitation and result in better performance of the auto-scalers.

Dhalion improvements The general problem of Dhalion is that it scales too slowly. This causes Dhalion to have poor performance when the input rate is increasing and decreasing. Extending Dhalions' logic to scale up more than one operator at a time and allowing it to estimate how much to scale down by using a performance metric like the CPU usage can result in a better performing and more resource-efficient auto-scaler.

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. 2005. The Design of the Borealis Stream Processing Engine. (2005).

[2] Leonardo Aniello, Silvia Bonomi, Federico Lombardi, Alessandro Zelli, and Roberto Baldoni. 2014. An Architecture for Automatic Scaling of Replicated Services. In *Networked Systems (Lecture Notes in Computer Science)*, Guevara Noubir and Michel Raynal (Eds.). Springer International Publishing, Cham, 122–137. https://doi.org/10.1007/978-3-319-09581-3_9

[3] Hamid Arabnejad, Pooyan Jamshidi, Giovani Estrada, Nabil El Ioini, and Claus Pahl. 2016. An Auto-Scaling Cloud Controller Using Fuzzy Q-Learning - Implementation in OpenStack. In *Service-Oriented and Cloud Computing (Lecture Notes in Computer Science)*, Marco Aiello, Einar Broch Johnsen, Schahram Dustdar, and Ilche Georgievski (Eds.). Springer International Publishing, Cham, 152–167. https://doi.org/10.1007/978-3-319-44482-6_10

[4] HamidReza Arkian, Guillaume Pierre, Johan Tordsson, and Erik Elmroth. 2021. Model-based Stream Processing Auto-scaling in Geo-Distributed Environments. In *2021 International Conference on Computer Communications and Networks (ICCCN)*. 1–10. https://doi.org/10.1109/ICCCN52240.2021.9522236 ISSN: 2637-9430.

[5] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. 2014. ShuttleDB: Database-Aware Elasticity in the Cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*. 33–43.

[6] Apache Beam. 2023. Apache Beam Nexmark benchmark suite. https://beam.apache.org/documentation/sdks/java/testing/nexmark/

[7] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 25, 6 (June 2009), 599–616. https://doi.org/10.1016/j.future.2008.12.001

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

[9] Emiliano Casalicchio, Lars Lundberg, and Sogand Shirinbab. 2017. Energy-aware auto-scaling algorithms for Cassandra virtual data centers. *Cluster Computing* 20, 3 (Sept. 2017), 2065–2082. https://doi.org/10.1007/s10586-017-0912-6

[10] Confluent. 2023. What is Kafka? https://www.confluent.io/what-is-apache-kafka/

[11] Jonathan D Cryer. 1986. *Time series analysis*. Vol. 286. Duxbury Press Boston.

[12] Dong Nguyen Doan, Daniela Zaharie, and Dana Petcu. 2020. Auto-scaling for a streaming architecture with fuzzy deep reinforcement learning. In *European Conference on Parallel Processing*. Springer, 476–488.

[13] Docker. 2023. Use containers to Build, Share and Run your applications. https://www.docker.com/resources/what-container/

[14] Apache Flink. 2022. Apache Flink Deployment Overview. https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/deployment/overview/

[15] Apache Flink. 2022. Apache Flink — Stateful Computations over Data Streams. https://flink.apache.org/

[16] Apache Flink. 2022. Metrics. https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/ops/metrics/

[17] Apache Flink. 2022. What is Apache Flink? — Architecture. https://flink.apache.org/flink-architecture

[18] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836. Publisher: VLDB Endowment.

[19] Tom ZJ Fu, Jianbing Ding, Richard TB Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: dynamic resource scheduling for real-time analytics over fast streams. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. IEEE, 411–420.

[20] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2017. DRS: Auto-Scaling for Real-Time Stream Analytics. *IEEE/ACM Transactions on Networking* 25, 6 (Dec. 2017), 3338–3352. https://doi.org/10.1109/TNET.2017.2741969 Conference Name: IEEE/ACM Transactions on Networking.

[21] Alessio Gambi and Giovanni Toffetti. 2012. Modeling Cloud performance with Kriging. In *2012 34th International Conference on Software Engineering (ICSE)*. 1439–1440. https://doi.org/10.1109/ICSE.2012.6227075 ISSN: 1558-1225.

[22] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. 2014. Adaptive, Model-driven Autoscaling for Cloud Applications. (2014).

[23] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1447–1463. https://doi.org/10.1109/TPDS.2013.295 Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[24] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, and Gabriel Iszlai. 2011. Exploring Alternative Approaches to Implement an Elasticity Policy. In *2011 IEEE 4th International Conference on Cloud Computing*. 716–723. https://doi.org/10.1109/CLOUD.2011.101 ISSN: 2159-6190.

[25] Manish K. Govil and Michael C. Fu. 1999. Queueing theory in manufacturing: A survey. *Journal of Manufacturing Systems* 18, 3 (Jan. 1999), 214–240. https://doi.org/10.1016/S0278-6125(99)80033-8

[26] Jens Haussmann, Wolfgang Blochinger, and Wolfgang Kuechlin. 2019. Cost-efficient parallel processing of irregularly structured problems in cloud computing environments. *Cluster Computing* 22, 3 (Sept. 2019), 887–909. https://doi.org/10.1007/s10586-018-2879-3

[27] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In *2014 IEEE 30th International Conference on Data Engineering Workshops*. 296–302. https://doi.org/10.1109/ICDEW.2014.6818344

[28] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*. 276–287.

[29] Helm. 2023. Helm: the package manager for Kubernetes. https://helm.sh/

[30] Apache Heron. 2022. Apache Heron: A realtime, distributed fault-tolerant stream processing engine. https://heron.apache.org/

[31] Nicolas Hidalgo, Daniel Wladdimiro, and Erika Rosas. 2017. Self-adaptive processing graph with operator fission for elastic stream processing. *Journal of Systems and Software* 127 (2017), 205–216. Publisher: Elsevier.

[32] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. 2016. Elastic stream processing for the internet of things. In *2016 IEEE 9th international conference on cloud computing (CLOUD)*. IEEE, 100–107.

[33] Bart Jacob, Richard Lanyon-Hogg, Devaprasad K Nadgir, and Amr F Yassin. 2004. A practical guide to the IBM autonomic computing toolkit. *IBM Redbooks* 4, 10 (2004), 1–268.

[34] Rachana Jannapureddy, Quoc-Tuan Vien, Purav Shah, and Ramona Trestian. 2019. An auto-scaling framework for analyzing big data in the cloud environment. *Applied Sciences* 9, 7 (2019), 1417. Publisher: MDPI.

[35] Apache Kafka. 2023. Apache Kafka. https://kafka.apache.org/

[36] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 783–798.

[37] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. 2009. Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters. In *Proceedings of the 6th international conference on Autonomic computing (ICAC '09)*. Association for Computing Machinery, New York, NY, USA, 117–126. https://doi.org/10.1145/1555228.1555261

[38] Kubernetes. 2023. Production-Grade Container Orchestration. https://kubernetes.io/

[39] Changsik Lee, Taeheum Na, Seungwoo Hong, and Taeyeon Kim. 2022. Application Aware Auto-Scaling in the Cloud: A Machine Learning Approach. https://doi.org/10.2139/ssrn.4250242

[40] James R. Leigh. 2004. *Control Theory*. IET. Google-Books-ID: 3P1zTw1HmyIC.

[41] Xunyun Liu, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu, and Rajkumar Buyya. 2017. A stepwise auto-profiling method for performance optimization of streaming applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 12, 4 (2017), 1–33. Publisher: ACM New York, NY, USA.

[42] Björn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 399–410. https://doi.org/10.1109/ICDCS.2015.48 ISSN: 1063-6927.

[43] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. 2017. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 3 (2017), 572–585. Publisher: IEEE.

[44] Federico Lombardi, Andrea Muti, Leonardo Aniello, Roberto Baldoni, Silvia Bonomi, and Leonardo Querzoni. 2019. Pascal: An architecture for proactive auto-scaling of distributed services. *Future Generation Computer Systems* 98 (2019), 342–361. Publisher: Elsevier.

[45] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. 2014. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing* 12, 4 (Dec. 2014), 559–592. https://doi.org/10.1007/s10723-014-9314-7

[46] L Magnoni. 2015. Modern Messaging for Distributed Sytems. *Journal of Physics: Conference Series* 608 (May 2015), 012038. https://doi.org/10.1088/1742-6596/608/1/012038

[47] Michael Maurer, Ivan Breskovic, Vincent C. Emeakaroha, and Ivona Brandic. 2011. Revealing the MAPE loop for the autonomic management of Cloud infrastructures. In *2011 IEEE Symposium on Computers and Communications (ISCC)*. 147–152. https://doi.org/10.1109/ISCC.2011.5984008 ISSN: 1530-1346.

[48] Gabriele Mencagli, Massimo Torquati, and Marco Danelutto. 2018. Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Generation Computer Systems* 79 (2018), 862–877. Publisher: Elsevier.

[49] Robbert Metzger. 2021. Scaling Flink automatically with Reactive Mode. `https://flink.apache.org/2021/05/06/reactive-mode.html`

[50] Rafael Moreno-Vozmediano, Rubén S. Montero, Eduardo Huedo, and Ignacio M. Llorente. 2019. Efficient resource provisioning for elastic cloud services based on machine learning techniques. *Journal of Cloud Computing* 8, 1 (2019), 1–18. Publisher: Springer.

[51] Abid Nisar, Waheed Iqbal, Fawaz Bokhari, Faisal Bukhari, and Khaled Almustafa. 2019. Dynamic Horizontal and Vertical Scaling for Multi-tier Web Applications. *Intelligent Automation and Soft Computing* (2019), –1–1. `https://doi.org/10.31209/2019.100000159`

[52] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys '09)*. Association for Computing Machinery, New York, NY, USA, 13–26. `https://doi.org/10.1145/1519065.1519068`

[53] John Pemberton. 2011. Time Series Analysis with Applications in R, Second edition. *Journal of Applied Statistics* 38, 6 (June 2011), 1311–1312. `https://doi.org/10.1080/02664760903075663`

[54] Prometheus. 2023. From metrics to insight. `https://prometheus.io/`

[55] E. G. Radhika and G. Sudha Sadasivam. 2021. A review on prediction based autoscaling techniques for heterogeneous applications in cloud environment. *Materials Today: Proceedings* 45 (2021), 2793–2800. Publisher: Elsevier.

[56] Navaneeth Rameshan, Ying Liu, Leandro Navarro, and Vladimir Vlassov. 2016. Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-tenancy. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 233–244. `https://doi.org/10.1109/CCGrid.2016.71`

[57] Rodrigo da Rosa Righi, Vinicius Facco Rodrigues, Cristiano André da Costa, Guilherme Galante, Luis Carlos Erpen de Bona, and Tiago Ferreto. 2016. AutoElastic: Automatic Resource Elasticity for High Performance Applications in the Cloud. *IEEE Transactions on Cloud Computing* 4, 1 (Jan. 2016), 6–19. `https://doi.org/10.1109/TCC.2015.2424876` Conference Name: IEEE Transactions on Cloud Computing.

[58] John F. Shortle, James M. Thompson, Donald Gross, and Carl M. Harris. 2018. *Fundamentals of Queueing Theory*. John Wiley & Sons. Google-Books-ID: P4ZZDwAAQBAJ.

[59] Bradley Simmons, Hamoun Ghanbari, Marin Litoiu, and Gabriel Iszlai. 2011. Managing a SaaS application in the cloud using PaaS policy sets and a strategy-tree. In *2011 7th International Conference on Network and Service Management*. 1–5. ISSN: 2165-963X.

[60] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[61] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. 2006. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *2006 IEEE International Conference on Autonomic Computing*. 65–73. `https://doi.org/10.1109/ICAC.2006.1662383`

[62] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. ACM, Snowbird Utah USA, 147–156. `https://doi.org/10.1145/2588555.2595641`

[63] ClarkNet HTTP Trace. 2023. ClarkNet HTTP Trace (From the Internet Traffic Archive). ftp://ita.ee.lbl.gov/traces/

[64] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. 2008. *Nexmark–a benchmark for queries over data streams (draft)*. Technical Report. Technical Report. Technical report, OGI School of Science & Engineering at ….

[65] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. 2008. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 3, 1 (2008), 1–39. Publisher: ACM New York, NY, USA.

[66] Balázs Varga, Márton Balassi, and Attila Kiss. 2021. Towards autoscaling of Apache Flink jobs. *Acta Universitatis Sapientiae, Informatica* 13, 1 (2021), 39–59.

[67] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 22–31. https://doi.org/10.1109/IC2E.2016.38

[68] Yanlong Zhai and Wu Xu. 2017. Efficient Bottleneck Detection in Stream Process System Using Fuzzy Logic Model. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 438–445. https://doi.org/10.1109/PDP.2017.71 ISSN: 2377-5750.

[69] Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. 2007. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *Fourth International Conference on Autonomic Computing (ICAC'07)*. IEEE, 27–27.

[70] Apache ZooKeeper. 2020. Welcome to Apache Zookeeper. https://zookeeper.apache.org/