A Comparison of Auto-scaling Techniques for Distributed Stream Processing

Wybe J. C. Koper

TU Delft

A Comparison of Auto-scaling Techniques for Distributed Stream Processing

by

Wybe J. C. Koper

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday May 31, 2022 at 11:00 AM.

Student number:	4533909
Project duration:	August 30, 2021 – May 31, 2022
Thesis committee:	Prof. dr. ir. G.J.P.M Houben, TU Delft
	Prof. dr. D. Spinellis, TU Delft
	Dr. A. Katsifodimos, TU Delft, supervisor

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

TU Delft

Abstract

As the world continues to embrace cloud computing, more applications are being scaled elastically. Elastic scaling allows applications to add or remove computing resources based on the load experienced by the application. When the load is high more resources are provisioned enabling the application to keep up with the load. When the load is low resources are removed ensuring that no resources are sitting idle. When implemented correctly elastic scaling allows applications to use fewer resources while maintaining application performance. One type of application that can benefit greatly from elastic scaling is a distributed stream processing application. Distributed stream processing applications are suited well for elastic scaling because the data coming through the data stream can be dynamic. This dynamic data stream makes it difficult to provision the right amount of computing power. One way to solve this problem is by using an auto-scaler that elastically scales the stream processing application. In this thesis, we compare different auto-scaling techniques for the distributed stream processing application Apache Flink. We implement a modern version of DS2 [20] using metrics native to Apache Flink. An auto-scaler designed specifically for Apache Flink by Varga et al. [32]. A modified version of the Dhalion [8], and a simple CPU based Kubernetes Horizontal Pod Auto-scaler (HPA). We compare the auto-scalers on the average number of resources used, the average latency, and the number of rescale operations. Our results show the importance of a cooldown period between scaling events. The benefits of incorporating metrics from the message queue into the scaling decision, and that throughput based evaluation methods work well for determining by how much to scale.

Preface

After six years at TU Delft, this thesis marks the end of my studies. Before picking a thesis topic I had set two main goals for my thesis. The first goal was to get hands on experience with cutting edge technologies such as Kubernetes. The second goal was to do my thesis at a company. I wanted to get hands on experience with cutting edge technologies because throughout my masters I had mostly been doing theoretical work and felt my practical skills could be improved. I wanted to do my thesis at a company because throughout my master's the COVID-19 pandemic had been raging on, and I was quite sure that if I had to spend another nine months working alone in my room I would not be setting myself up for a successful thesis. Doing my thesis at a company would give me more support during my thesis and allow me to get out of my room and meet new people. I am happy to say that I have achieved both the goals I had set for my thesis. These goals could not have been achieved without the help of quite a few people.

Firstly, I would like to thank everyone at Info Support and especially my company supervisor Rinse van Hees for helping me throughout the entire thesis process. Brainstorming interesting topics together before starting my thesis, and helping to write the proposals really set me up to have a successful and enjoyable thesis. The weekly discussions also really helped me keep progress going while giving me a fresh perspective when I got stuck.

Second, I would like to thank my supervisor at TU Delft Asterios Katsifodimos. Without your support, I would not have been able to get the TU Delft to allow me to do a thesis at a company. The flexibility you gave me when determining a thesis topic allowed me to pick a topic I was genuinely interested in. Furthermore, the feedback I received from our meetings was instrumental in ensuring the quality of research done in this thesis.

Third, I would like to thank my friends. Spending time together throughout the myriad of lock-downs endured in the last 3 years has really kept me going and kept life reasonably enjoyable given the circumstances.

Finally, I would like to thank my family and my girlfriend for their love and support. I hope that I can continue to make you all proud as I move to a new phase in my life.

Wybe J. C. Koper
Delft, May 2022

Contents

1 Introduction

Cloud computing has revolutionized how businesses build and operate their applications. Before the advent of cloud computing, companies would have to procure, configure and upgrade their own hardware. With cloud computing, companies can rent the hardware they need at the click of a button. This ease of provisioning hardware has allowed developers to create applications that scale out (add servers) or scale in (remove servers) based on the load on the application. This has historically not been possible due to the long procurement time of hardware. Companies had to choose between over-provisioning or under-provisioning their hardware. Over-provisioning would waste resources but the application would be able to handle more traffic while under-provisioning would maximally utilize resources but could lead to service level agreement (SLA) violations. Applications that have the ability to scale in and out are called elastic applications. In this relatively new world of elastic applications, questions have arisen such as: When should an application scale in or out? By how much should an application scale in or out? What metrics are good indicators that an application needs to be re-scaled? A system that answers all these questions is known as an auto-scaler. An auto-scaler monitors the elastic application and determines when to scale the application in or out and by how much.

Auto-scalers can be divided into two groups, auto-scalers that look at the current state of the system and base their scaling decisions on this state, these are known as reactive auto-scalers. Or auto-scalers which forecast the future state of the system and make scaling decisions based on the forecasted future state, these are known as proactive auto-scalers. The downside of reactive auto-scalers is that they always lag behind the true needs of the system, as the re-scale operation is triggered only when some SLA is violated. The proactive auto-scalers can mitigate this problem as long as they can reasonably forecast the future state of the system [26].

Another emerging trend in the information revolution is stream processing. Traditionally companies would use batch processing for their data processing and analytic needs. With batch processing data is collected into batches and analyzed at a specific time period, resulting in a delay between the time that the data was collected and when it is processed. With stream processing data is processed and analyzed in real-time. Stream processing also plays an important role in the deployment of machine learning models. For example, credit card fraud can be detected right after a credit card swipe using a machine learning model deployed on a stream processing framework. As companies continue to push for real-time data analytics the adoption of stream processing frameworks will only increase.

Stream processing is suited particularly well for the elastic environment of the cloud as data streams are inherently unbounded. This unbounded nature of the incoming data is not the case for batch processing where the size of the input data is already known. Provisioning resources for batch processing is relatively straightforward as the system needs to process batches of a certain known size. In regard to stream processing applications, it would be ideal if the provisioned resources are dependent on the amount of data coming in. If the number of resources is dependent on the amount of data coming in, then a balance can be found between the number of provisioned resources and the streaming applications SLAs.

The idea of utilizing cloud resources in an environmentally friendly way by using them efficiently while meeting SLA requirements fits into the green computing paradigm [21]. More efficient usage of computing resources leads to less energy consumption which in turn leads to less greenhouse gas

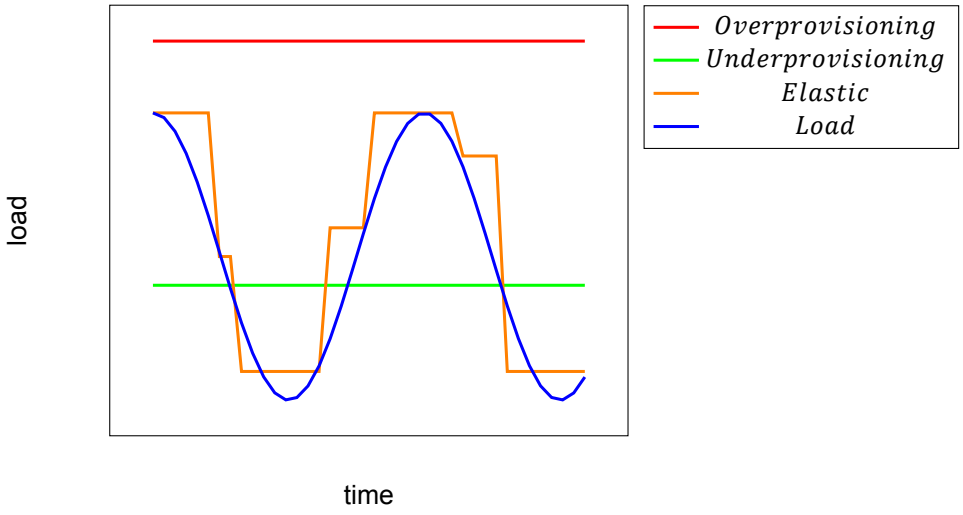

Figure 1.1: Different resource allocation scenarios for a hypothetical sinusoidal system load.

emissions for as long as most energy is obtained from hydrocarbons. Utilizing fewer resources is not just beneficial for the environment but also for companies' bottom lines as cloud resources incur costs based on the pay-as-you-go model. Meeting SLA requirements is also important as users demand fast response time. No customer wants to wait seconds before a payment is processed or a movie recommendation is made. The relative importance of resource utilization versus SLA requirements is dependent on the business context of the application.

In this work we compare six different auto-scalers implemented for Apache Flink running on Kubernetes. The auto-scalers are tested in an experimental setup that creates a sinusoidal load pattern augmented with some noise. The six implemented auto-scalers are the state of the art stream processing auto-scaler DS2 [20] adapted to use metrics native to Apache Flink, a reactive auto-scaler based on the Dhalion auto-scaling framework [16], the Kubernetes Horizontal Pod Auto-scaler (HPA) [5], an auto-scaler designed specifically for Apache Flink by Varga et al. [32], the last two auto-scaler are modified variants of our DS2 and Varga auto-scaler implementations made after observing the experimental results of their unmodified counterparts.

1.1. Problem Statement

Many papers have been written proposing different types of auto-scalers for distributed stream processing such as DRS [17], DS2 [20], and Dhalion [16], but there are few that compare these different auto-scalers. Papers such as DRS [17] and Varga et al.[32] show only how their own auto-scaler performs without any comparison to the state of the art. This lack of comparisons is detrimental to the field of auto-scaling distributed stream processing applications as there is no clear consensus on which auto-scalers perform best under which scenarios.

The experiments that were performed to test the performance of the proposed auto-scalers were quite simple. The load on the stream processing application was often static or modified slightly at a certain time point which was the case for Kalavri et al. [20] and Fu et al. [17]. In a real world stream processing application the load could be much more dynamic. Without testing auto-scalers for distributed stream processing under a more realistic system load the true benefit of the auto-scalers can not be determined.

Furthermore, most experiments do not mimic a real world setup of stream processing applications. In the paper by Kalavri et al. [20] the source operators of the streaming dataflow also generate the data. In a real world system, the source of the streaming dataflow would read from a messaging queue such as Apache Kafka. When such a message queue is in place, metrics from this message queue can be used to make scaling decision creating more complex auto-scaling solutions which combine application metrics from both the stream processing application and message queue.

1.2. Research Questions

Given the problems described we will explore the following research questions in this thesis:

(**RQ0):** Which auto-scalers from the literature can be used for scaling distributed stream processing applications?

(**RQ1):** How can we compare these auto-scalers?

(**RQ2):** How can these auto-scalers be improved?

1.3. Organization

To answer RQ0, we explore different auto-scaling implementations from the scientific literature in chapter 2. In chapter 3 we describe the implementations of the different auto-scalers for the stream processing framework Apache Flink. In chapter 4 we describe the experimental setup for comparing these different auto-scalers (RQ1). In chapter 5 we describe the results of our experiments and use these results to answer RQ2. Finally, in chapter 6 we reflect on the limitations of the experiments and propose some ideas for future work.

2

Background & Related Work

In this section, we explain different auto-scaling techniques. We explain some of the basics of stream processing. Furthermore, we look at the modern infrastructure stack for distributed applications and what performance metrics can be used by an auto-scaling system. We also describe how the messaging queues present in front of most stream processing applications work. Finally we discuss some of the promising auto-scalers for distributed stream processing found in the literature.

2.1. Auto-scaling

A system that automatically adjusts the resources needed by an application is called an auto-scaling system. All auto-scaling systems have to deal with the following problems.

- **Over-provisioning:** the application has more resources than needed, resulting in resources sitting idle or operating below their capacity. However, slight over-provisioning of resources is desirable in practice to cope with minute workload fluctuations [22].
- **Under-provisioning:** the application does not have enough resources to comply with the SLA for the application.
- **Oscillation:** oscillations occur when scaling operations are carried out in quick succession before the effect of the scaling operation can be properly measured. Using a cooldown period is a common solution to this problem [10].

Two forms of scaling exist namely vertical scaling (definition 2.1.1) and horizontal scaling (definition 2.1.2). In this thesis, we focus on horizontal scaling because vertical scaling is bounded by hardware limitations while horizontal scaling is unbounded [31]. Furthermore horizontal scaling of an application also contributes to fault tolerance and high availability which is not the case for vertical scaling.

Definition 2.1.1 (Vertical scaling)**.** Vertical scaling entails adding more computing power by upgrading the existing machine in the resource pool of the application.

Definition 2.1.2 (horizontal scaling)**.** Horizontal scaling entails adding more computing power by adding more machines to the resource pool of the application.

The type of application being re-scaled also matters to the auto-scaling system, in particular, whether the application has state (definition 2.1.3). Scaling a stateful application (definition 2.1.5) incurs more overhead than scaling a stateless application (definition 2.1.5). In order to scale stateful applications, the state has to be saved periodically [13]. When a scaling event is triggered the application experiences some downtime while resources are added or removed. Once this process has finished the previously saved state can be reloaded onto the available resources and the application can start processing again.

Definition 2.1.3 (State)**.** A set of conditions at a moment in time.

Definition 2.1.4 (Stateful application)**.** A stateful application is an application that remembers one or more preceding events in a given sequence of interactions.

Definition 2.1.5 (Stateless application)**.** A stateless application is an application that has no record of previous interactions, each interaction is handled based entirely on the information that comes with it.

An auto-scaling process satisfies the MAPE loop of autonomous systems [24]. MAPE stands for monitor, analyze, plan and execute. A monitoring system collects and stores performance metrics, these metrics are then analyzed to see if they violate an SLA or will violate an SLA in the future. Then in the planning phase, the auto-scaler determines when to scale and by how much. In the execution step the application is actually scaled in our out. When designing an auto-scaler only the analysis and planning steps have to be considered, the monitoring and executing steps are usually taken care of by other systems.

- **Monitor:** to meet the SLA requirements of an application the auto-scaling system needs to obtain metrics about application performance. Cloud providers typically provide these types of metrics through an API. The quality and availability of these metrics are important for the auto-scaler as it uses these metrics to make decisions. The metrics looked at by an auto-scaling system can range anywhere from hardware metrics such as CPU utilization to application-level metrics such as the queue length of a messaging queue. The cadence at which these metrics are collected is also important as forecasting methods work better when more data is available.
- **Analyze:** the analysis step of an auto-scaler can be split into two types, checking if the system is currently violating any SLA (reactive) or forecasting the observed metrics to see if an SLA violation will likely occur in the future (proactive). Reactive auto-scalers always lag behind the true needs of the system as scaling an application is not instantaneous, therefore from a theoretical standpoint, proactive auto-scalers appear superior given some ability to correctly predict the future needs of the system.
- **Plan:** in the planning phase of the auto-scaling system, the auto-scaler must decide how many resources to add or remove to find the correct balance between resource utilization and SLA compliance. The system must also decide when would be the best time to scale to optimally use resources and balance potential SLA violations.
- **Execute:** the execution step of scaling the application can be done by using the API of the cloud provider. The cloud provider will then add or remove resources. If the infrastructure that the application runs in is configured correctly then all the networking will also be changed automatically. The execution step can also be handled by container orchestration solutions more on those in section 2.4.

2.1.1. Auto-scaling Techniques

Many different types of auto-scalers have been designed over the years. In a review article by Lorido-Botran et al. [22] auto-scalers were classified into five groups: threshold-based, reinforcement learning, queuing theory, control theory, and time series analysis.

- **Threshold-based rules:** threshold-based auto-scalers are the simplest. If a certain metric is above or below a threshold for a set amount of time the auto scaler will scale in or out. This type of reactive auto-scaler is widely used and available from most cloud providers.
- **Reinforcement learning (RL):** reinforcement learning based auto-scalers model the auto-scaler as an agent in an environment. The agent has three possible moves, do nothing, scale in or scale out. Based on the decisions of the agent, it receives a reward, the reward is based on some function that combines both resource utilization and SLA compliance. Using RL the auto-scaler can learn through trial and error a scaling policy. In theory, this type of auto-scaler sounds great as no tuning of the auto-scaler will be required it just ”learns”, but in practice, RL is plagued by long learning times resulting in low industry adoption [22].
- **Queuing Theory (QT):** queue theory based auto-scalers model the system as a queue. A queue has items going in, waiting, and going out. From an application perspective, this could be seen

as requests coming in, being processed, and leaving. When a system can be modeled as queue useful properties can be derived such as queue length. These properties can then be used to determine if the system is appropriately scaled given the input and output rates. Queue theory based models work best when an application is in steady state meaning the input and output rates of the queue are not changing. However, in practice, steady state is rarely the case so the queueing model needs to be recomputed often [22].

- **Control Theory (CT):** control theory based auto-scalers also rely on a model of the application. Control theory based auto-scalers work by trying to maintain the value of a controlled variable (e.g. CPU utilization) close to a certain target value, by adjusting the manipulated variable (e.g. number of servers). The effectiveness of control theory based auto-scalers is model-dependent similar to queuing based auto-scalers. An example of a feedback control system can be seen in Figure 2.1, within the controller there is a model that determines what the value of the manipulated variable should be based on the control error.

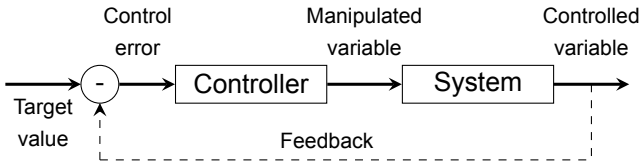

Figure 2.1: Block diagram of a feedback control system.

- **Time series analysis (TS):** time series based auto-scalers use time series forecasting methods to predict performance metrics of the system. The forecasted performance metrics allow the auto-scaler to proactively make scaling decisions. The effectiveness of time series based auto-scalers is highly dependent on getting an adequate fit of the time series model on metric data. If a poor fit is obtained then the forecasts will not be representative of the actual observed values.

2.2. Stream Processing

Stream processing is a term that encompasses a variety of systems. A stream processing system (SPS) typically contains modules that compute in parallel [28]. These modules can have different functions. Sources that pass data into the system; operators that perform some computations and sinks that pass data from the system. SPS use streams to communicate between different modules. In Figure 2.2 an architecture of a system that uses a SPS is displayed. Various devices stream data to a message queue, the message queue is then consumed by the stream processor which performs some form of computation and publishes the results to various destinations.

Definition 2.2.1 (stream)**.** A stream is an infinite list of elements a_0, a_1, a_2, ... taken from some data set of interest A, and is usually formalized mathematically as a function T → A, wherein T= $\mathbb{N}$ = {0,1,2, ...} represents discrete time [28].

Since a data stream is an infinite list of elements (definition 2.2.1) that can vary, there is an inherent need for scaling. This is not the case for batch processing systems as with batch processing systems the size of the number of records to be processed is already known before the batch job is started. In this work, we compare different auto scalers for the stateful stream processing framework Apache Flink [12]. Apache Flink was released in 2011 and has seen widespread industry adoption.

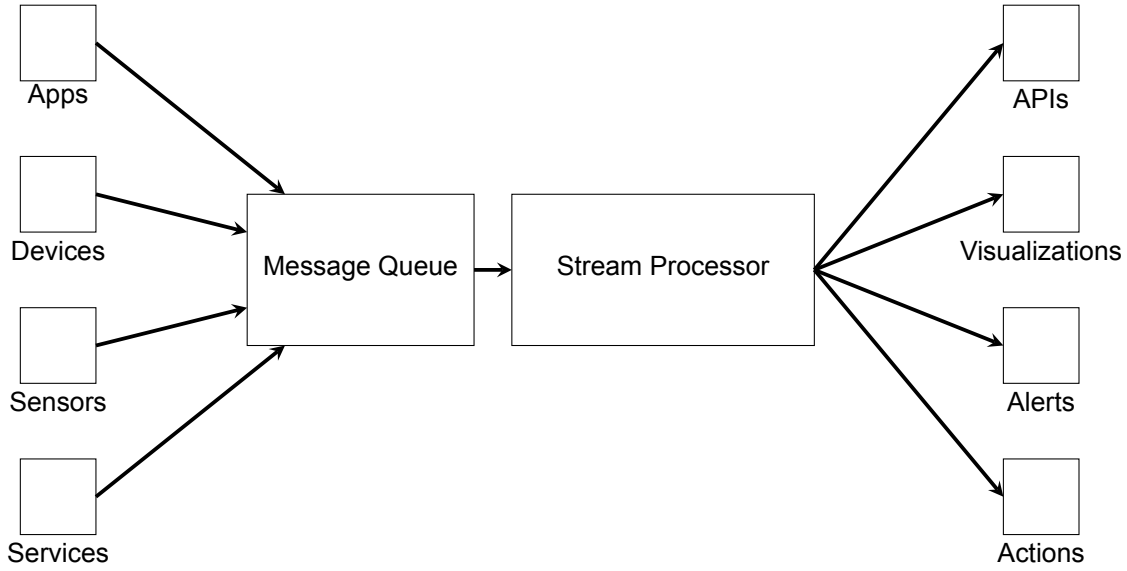

Figure 2.2: Architecture of an application that obtains data from various sources, stores data in a message queue which is consumed by a streaming framework.

2.2.1. Datastream Operations

Data going through a datastream or stored in batches can be analyzed using queries. A query is simply a request to the application managing the data, to retrieve and/or modify said data. A common language to express queries on datasets is called Structured Query Language(SQL). Distributed stream processing frameworks like Apache Flink do not directly support SQL for processing data streams. Developers have to write code that extends common data processing operations available in Apache Flink. Some common operations are:

- **Map:** A map function takes one element and maps it to another element.
- **Filter:** A filter function evaluates a boolean expression for each element and keeps only those for which the expression is true.
- **Keyby:** A keyby function partitions the datastream into disjoint datastreams. The keyby function is vital for distributed stream processing as it ensures that records with the same keys go to the same nodes for processing.
- **Window:** A window function groups data according to some characteristics. For example, all data that arrived in the last 5 seconds.
- **Union:** Combines one or more datastreams into a single datastream.
- **Window Join:** Joins two datastream into one based on a certain key and common window.

In Listing 2.2.1 an example query is shown on a dataset containing bids for an auction. The query selects the ids and amounts of all bids that are higher than 1000. In Figure 2.3 this query is shown as a dataflow graph. The data containing the bids is read from a source, passed through a filter operator where only bids higher than 1000 are selected, and then passed on to a sink where the resulting data is stored. This dataflow graph can be parallelized and therefore distributed over multiple computing nodes.

```
1 SELECT bids.id, bids.USD_amount, FROM bids WHERE bids.USD_amount > 1000;
```

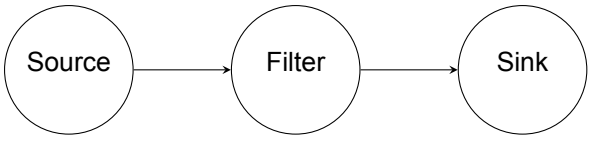

Figure 2.3: Dataflow graph with parallelism of 1 of SQL query found in Listing. 2.2.1

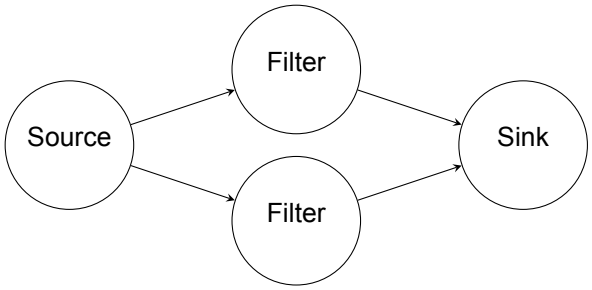

Figure 2.4: Dataflow graph where filter operator has parallelism of 2.

2.2.2. Auto-scaling Stream Processing

Auto-scaling a stream processing application requires two main steps. The first step is to determine whether the application is over or under-provisioned, this can be done by observing and analyzing application metrics which will be discussed in section 2.4.1. Once it is determined that the application is not provisioned correctly it needs to be re-scaled. The dataflow graph of the query being executed can be re-scaled by adding or removing nodes and adjusting the parallelism of the operators in the dataflow graph. As an example, given the dataflow graph in Figure 2.3, its operators can be parallelized to increase throughput as can be seen in Figure 2.3. However, finding the optimal parallelism is non-trivial as operators have inter-dependencies and operators have different processing rates. Some operators like maps typically process records quite fast while joins are much slower. Apache Flink [12] has recently released Reactive Mode [2] which abstracts away the scaling of the Apache Flink dataflow graph. If resources are added or removed from the Flink cluster then Apache Flink will scale each operator to the maximum possible parallelism on the provided resources [2]. This abstraction simplifies developing auto-scalers for Apache Flink, however, the amount of resources to add or remove still needs to be determined by the auto-scaler.

2.3. Messaging Services

Message queuing systems have become commonplace in distributed systems. Applications need to communicate to exchange data with each other which can become complex quite quickly [23]. Without a messaging service layer between applications, every application needs custom integrations to exchange data as can be seen in the left part of Figure 2.5. This integration problem can be solved by putting a message queueing system in-between applications which produce and consume data. A message queue provides a common interface for exchanging data which greatly simplifies the application architecture as can be seen in the right of Figure 2.5.

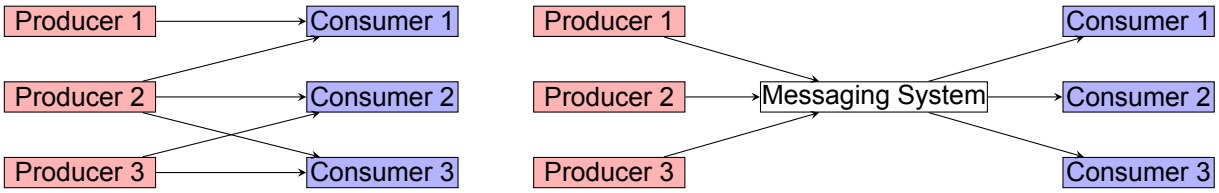

Figure 2.5: On the left, a distributed system without a messaging queue. On the right, a distributed system with a messaging queue.

2.3.1. Apache Kafka

Apache Kafka [1] was initially open-sourced as a message queuing platform in 2011 by LinkedIn. Since then its functionality has evolved considerably and now also supports stream processing. In this thesis, we will only be utilizing Kafka as a message queue and use Apache Flink for stream processing. A diagram of a Kafka cluster can be seen in Figure 2.6. A Kafka cluster contains one or more brokers, a broker is simply a server that is running Kafka. Producers which produce data and consumers which consume data can connect to the Kafka cluster. Data is produced to and consumed from a topic. A topic can be further split into one or more partitions. Finally, the replication factor of the topic has to be set which defines the number of copies of the topic. There is always one partition which is the leader.

Data from producers is written to the leading partition. The Kafka cluster automatically copies data from the leading partition to the replicas. Kafka is designed for high availability, if a broker goes down Kafka will elect a new leader partition. Producers and consumers will also not notice if a broker goes down as failover happens seamlessly as long as there are enough brokers and the replication factor is appropriately set.

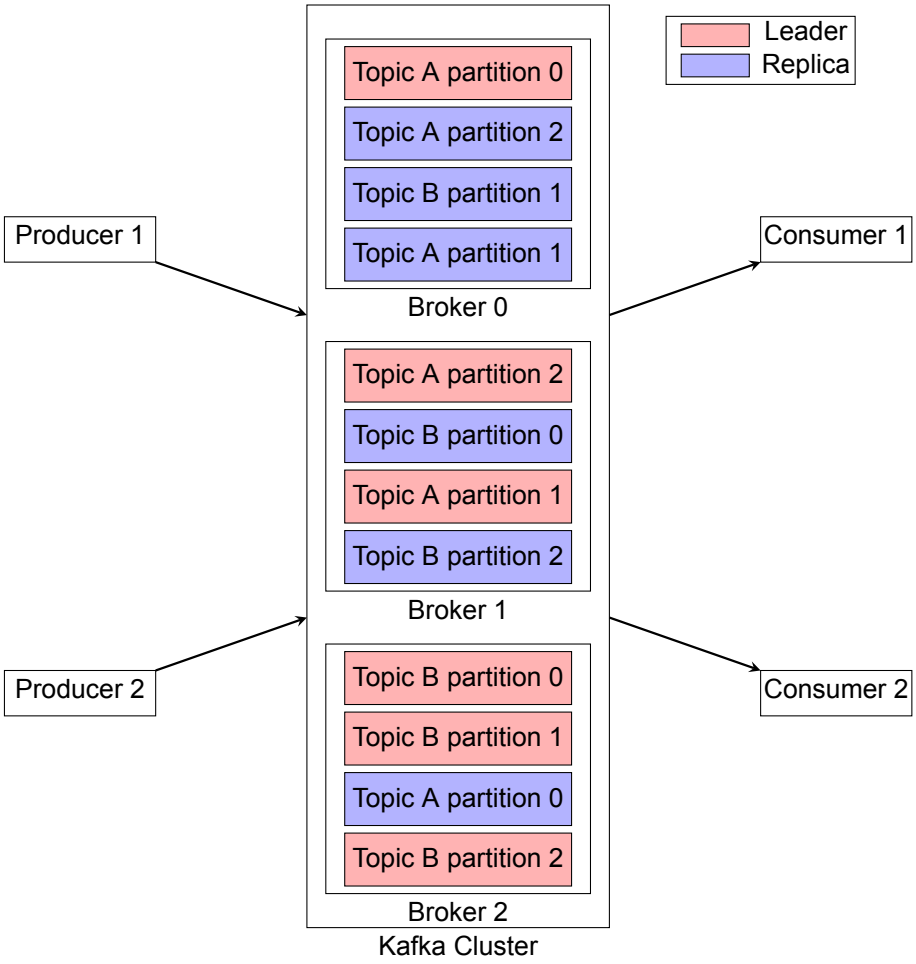

Figure 2.6: The architecture of a Kafka cluster with three brokers, two topics, three partitions per topic, and a replication factor of two.

2.4. Modern Infrastructure

The runtime environments of applications are constantly evolving. A couple of years ago virtual machines (VMs) were the go-to solution for running applications, but the industry has now shifted to containers. Containers can be thought of as lightweight VMs, as they are orders of magnitude smaller than VMs [25]. This smaller size allows more applications to be put on a single compute node as can be seen in Figure 2.7. This smaller size is a result of using kernel-level namespaces (a feature of Linux kernels) which isolate the container from the host. A container is therefore simply an isolated process [25]. Now imagine an organization that has to manage hundreds or even thousands of containers. How will containers be restarted when they go down? How will secrets be managed? This is where container orchestration technologies, such as Kubernetes [5], come into play. Kubernetes is one of the most used container orchestration solutions currently on the market. Kubernetes also comes with its own auto-scaling solution which we will explain further in section 3.2.

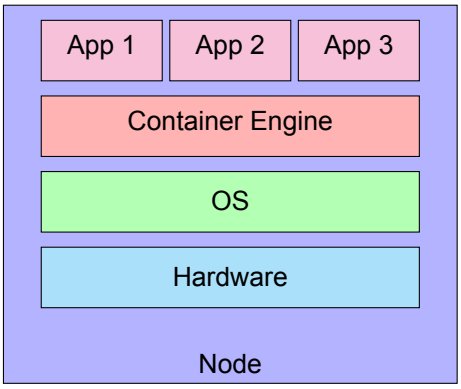

Figure 2.7: Modern solution stack for running containerized applications.

2.4.1. System Metrics

The runtime environment generates metrics of all levels of abstraction of the computing node, from hardware metrics all the way up to application metrics. An auto-scaling system needs these metrics to measure and analyze application performance and check if SLA violations are occurring or are likely to occur in the future. Different auto-scaling systems look at different metrics, the simplest auto-scalers look at low-level metrics such as CPU utilization. This type of auto-scaler is then usable for every application as every application is generating CPU utilization metrics. For more complex applications, metrics such as message queue size could be looked at. However, an auto-scaler that uses queue metrics would then only be appropriate for applications that actually utilize a message queue or can be modeled as one. Examples of common metrics can be found below.

- Hardware metrics
  - CPU utilization: the amount of time the CPU is used divided by the elapsed time
  - Memory usage: the amount of memory being used
- OS metrics
  - CPU time per process: the amount of time the CPU is used for a specific process divided by elapsed time
  - Load: number of processes waiting for CPU time
- Stream processing metrics
  - Latency: the time it takes for a record to be processed
  - Throughput: number of records processed in a given time period
  - Consumer latency: the amount of time a record spends in a queue before being consumed by a streaming service.
  - Consumer lag: the amount of unconsumed records in the message queue
  - Operator idle time: the amount of time an operator is idle
  - Operator busy time: the amount of time an operator is busy
  - Operator throughput: the amount of records per second processed by an operator
  - Backpressure: this metric will be explained in section 2.4.2
- Message queue metrics
  - Queue length: length of the queue
  - Arrival rate: number of records arriving in a given period of time

2.4.2. Backpressure

Backpressure occurs in a stream processing system when some operator(s) cannot process records at the rate at which they are being received. This causes the input buffers of that operator to build up, then the upstream operators' output buffers will also fill up [3]. When the output buffer of an operator is full its processing rate is reduced. Backpressure propagates all the way through the system until it reaches the source operator. In Figure 2.8 the buffer queues of the map operators are shown, if the source operator sends records to the map operators at a rate faster than they can be processed, the buffer queues will fill up. Once the buffer queues of the map operators are full, the output buffer of the source operator will fill up. When the source operators' output buffer is full, its processing rate will be reduced.

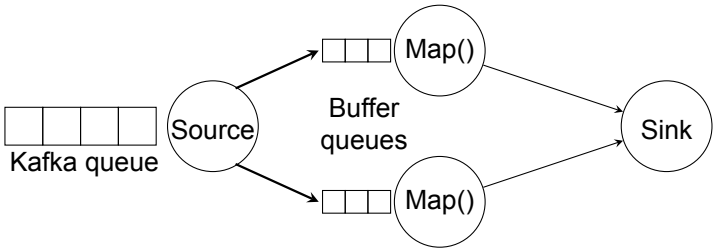

Figure 2.8: Dataflow graph showing buffer queues of operators and Kafka queue.

2.4.3. Prometheus

All the metrics mentioned in the previous section need to be exposed and collected. This can be done with a monitoring application called Prometheus [6]. Prometheus scrapes metrics from application endpoints and stores them in a time series database. These metrics can then be queried using the Prometheus Query Langauge (PromQL). Apache Flink, Apache Kafka, and Kubernetes all support settings for publishing their metrics to a specific endpoint. Prometheus is then configured to scrape these endpoints, allowing for monitoring of all applications in the cluster.

2.4.4. Apache Flink on Kubernetes

Apache Flink can be run on Kubernetes. To run Apache Flink on Kubernetes we need a pod containing a Jobmanager container and a pod containing a Taskmanager container. A pod is the smallest deployable unit of compute that can be created and managed in Kubernetes[1]. A pod can contain one or more containers. The Jobmanager pod contains the Flink query to be executed, manages checkpoints, and coordinates the Taskmanagers. The pod containing the Taskmanager will do the actual computations instructed by the Jobmanager. Kubernetes can be instructed to remove or add Taskmanger pods. When the amount of Taskmanagers changes and Reactive Mode (see section 2.2.2) is enabled the Jobmanager will automatically rescale the query on the available Taskmanagers. We will use different auto-scalers to instruct Kubernetes to add or remove Taskmanagers based on metrics monitored by Prometheus. An overview of a simple setup of Apache Flink on Kubernetes can be seen in Figure 2.9.

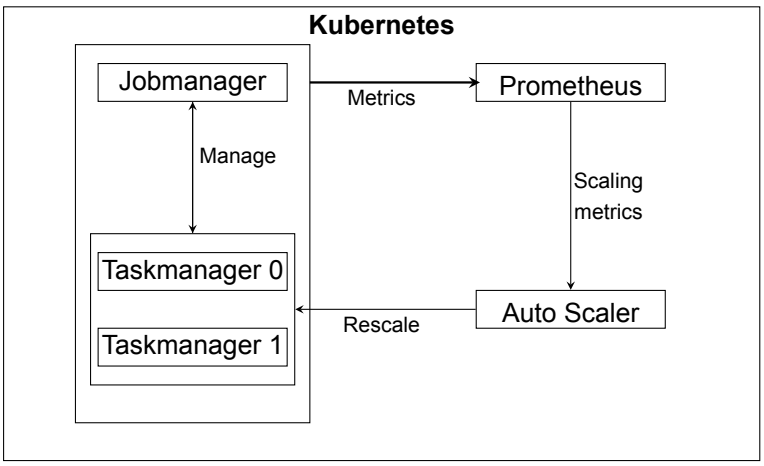

Figure 2.9: An overview of the pods required to run Apache Flink as well as Prometheus for monitoring metrics and an auto-scaler to add or remove Taskmanagers.

2.5. Related Work

Fu et al. published a paper where they introduced DRS a queue theory based auto-scaler for real-time stream analytics systems [17]. DRS uses queueing networks to model the dataflow graph. By using metrics from the streaming system such as input rate, processing rate, number of processors per operator, and inter-arrival times of inputs to an operator in combination with a queueing model, DRS is

[1] https://kubernetes.io/docs/concepts/workloads/pods/

able to estimate the latency per operator given a certain number of processors assigned to that operator. By being able to estimate the latency of every operator the entire latency of the streaming dataflow can be determined by summing up the latencies of each operator. DRS is then able to determine the amount of processors required per operator to satisfy a latency SLA. Fu et al. [17] then go on to conduct some experiments with DRS under a dynamic load. DRS is shown to be able to successfully adjust a streaming dataflow given a variable input rate. A downside of DRS is that it is not able to deal with dataflow graphs with multiple sources or sinks, as the latency calculation of the entire system is then no longer the sum of the latency of all individual operators.

In 2017 Floratou et al. published a paper about self-regulating stream processing in Heron [16]. Self-regulation includes, auto-scaling the streaming job, self-healing after a crash, and detecting and removing slow instances. Floratou et al. then go on to introduce a system that allows a stream processing engine to become self-regulating called Dhalion. The system has three parts, a symptom detection phase, a diagnosis generation phase, and a resolution phase. In the symptom detection phase, specific metrics are measured such as backpressure and buffer queue lengths. These symptoms are then used to diagnose different problems such as over or under-provisioning, or slow computing nodes. In the final phase, Dhalion has policies to resolve the diagnosed problems. The authors then go on to implement an auto-scaler using the Dhalion system. The auto-scaler detects backpressure and pending packets. Where pending packets are the amount of records in the input queues of the operators. If there is no backpressure and the number of pending packets is close to zero then the auto-scaler will scale down the operator by some configurable factor. If there is backpressure and the number of pending packets for the same operators are equal then there is under-provisioning. The factor that the operator that created the backpressure is scaled up by is determined by the amount of time that the operator is consuming data normally divided by the amount of time that data consumption for that operator is suspended. If 80% of the time data is consumed normally by the operator and 20% of the time data consumption is suspended then the operator is not able to handle $\frac{1}{4}$ of the data and thus needs to be scaled up 25%. The authors then go on to test the auto-scaler in some simple experiments where the desired throughput must be reached given a certain parallelism. The auto-scaler is able to reach the desired throughput after some time but takes many re-scale operations to get to the required parallelism to meet the desired throughput.

In 2018 one year after the Dhalion paper was published Kalavri et al. published a paper proposing a new method for auto-scaling distributed streaming dataflows named DS2 [20]. DS2 combines a general performance model of the of the streaming dataflow with some system metrics to estimate the true processing and output rates of individual operators [20]. This performance model and the true processing rates could then be used to determine the optimal parallelism of the streaming dataflow. The details of how DS2 works will be discussed in section 3.5. Kalavri et al. showed that DS2 was able to converge to an optimal parallelism in fewer steps than Dhalion, on a range of queries. DS2 was shown to be stable and have short settling times. Where stability entails that the parallelism of the dataflow is not oscillating, the short settling time refers to the fact that within a few configuration changes a parallelism is found to satisfy a target throughput rate. For DS2 to find the optimal parallelism the input rate to the streaming dataflow should be constant. However, in a real-world system, this is often not the case. The experiments carried out by Kalavri et al. do not contain substantial experiments about how DS2 performs in a dynamic environment.

More recently in 2021, Varga et al. [32] published a paper where they described a scaling architecture for Apache Flink deployed on Kubernetes. The scaling architecture utilized the Kubernetes HPA framework described later in section 3.2. Varga et al. incorporated metrics from the Kafka queue in front of the stream processing system. The auto-scaler built by Varga et al. [32] was not compared against other auto-scaling systems. The authors did investigate the effect of state size on scaling downtime. Larger state times result in a longer scaling downtime. There was some variance in the scaling downtime caused by variability in the time to delete and initialize pods. Further experiments were conducted where the Kubernetes variability was removed by directly measuring the restore time of a single operator. The authors found that there was a linear relationship between state size and load time. Varga et al. suggested that future auto-scalers should take into account the state size and scaling downtime when making scaling decisions.

Almost no previous research has been done comparing multiple different auto-scalers for distributed stream processing under a dynamic load. The comparison made between Dhalion [16] and DS2 [20] by Kalavri et al. was geared towards finding the optimal parallelism given a certain throughput target.

The target throughput was changed once during one of the experiments by Kalavri et al. but this can hardly be considered a dynamic load. However previous research has been conducted on comparing auto-scalers for general-purpose scaling of stateless applications like web servers. One such paper by Bauer et al. proposed a new auto-scaler called Chameleon [11]. Bauer et al. then went on to compare Chameleon with five popular auto-scalers from the literature: Adapt [9], Hist [30], ConPaaS [15], Reg [19], and Reactive [14]. Unfortunately, we cannot use these auto-scalers for distributed stream processing as they do not take into account the inter-dependencies between resources. These auto-scalers are developed mostly for stateless web servers. Bauer et al. generate workloads based on 5 real-world traces. Experiments were run in 3 different environments, a private CloudStack-based environment, a public AWS EC2 environment, and an OpenNebula based cloud environment. The evaluation metrics used by Bauer et al. come from a paper by Herbst et al. [18]. Herbst et al. define metrics called over and underprovisioning accuracy. Which determines how much an auto-scaler was over or under-provisioned throughout an experiment. Herbst et al. also define the over and underprovisioning timeshare which determines the amount of time the auto-scaler is either over or under-provisioned throughout the experiment duration. For exact details on how to calculate these metrics, we refer to the original paper by Herbst et al. [18]. Bauer et al. ended up showing that the Chameleon auto-scaler performed significantly better than the other auto-scalers.

2.6. Conclusion

In this chapter, we explored the theory behind auto-scaling systems and how auto-scalers satisfy the MAPE loop of autonomous systems. We looked at different auto-scaling techniques; rule based, time series, reinforcement learning, queue theory, and control theory. We also explored some of the basics of stream processing and how to scale stream processing frameworks. The architecture of Apache Kafka was explained in detail as well as what metrics are important for monitoring stream processing applications. Finally, we looked at related work on auto-scaling distributed stream processing applications. From the discussed auto-scalers we will be implementing DS2 [20] as it is highly regarded based on the number of citations. We will be implementing a modified version of Dhalion as the original version of Dhalion performed worse than DS2 in the experiments conducted by Kalavri et al. [20], however, some of the principles explained in the Dhalion paper can be used to create a new auto-scaler. We will also be implementing the auto-scaler proposed by Varga et al. [32] as it has been published recently and is easy to implement with the Kubernetes HPA. We will not be implementing DRS [17] as the underlying queuing model is too rigid. The underlying queueing model does not allow for the modeling of dataflow graphs with multiple sources which is quite common for stream processing queries. Finally, we will also be implementing a Kubernetes CPU HPA explained further in section 3.2. The Kubernetes CPU HPA will serve as a baseline as it is an out of the box general purpose auto-scaler.

3

Auto-scaler Implementations

In this section, we describe the metrics, evaluation criteria, and implementations of six different auto-scalers. The first auto-scaler is the out of the box horizontal pod auto-scaler (HPA) [4] from Kubernetes configured to use CPU utilization. The second is the reactive auto-scaler proposed by Varga et al. [32] which we will refer to as Vargav1. The third is an adaptation of the Varga et al. auto scaler named Vargav2. The fourth is an adaptation of the Dhalion auto-scaler [16], the fifth is a modern implementation of DS2 [20] called DS2-modern, and the final auto-scaler is a modified DS2 auto-scaler that has some extra logic to deal with the presence of a queue in front of the streaming application called DS2-modern-adapted.

3.1. Auto-scaling Architecture

All auto-scalers for stream processing systems have to solve three key problems. The first problem that must be solved is how to detect if there is over or under-provisioning. This detection is achieved by monitoring certain metrics which vary across auto-scaler implementations. After detecting an undesired state the auto-scaler has to determine how many resources should be added or removed. The amount of resources to add or remove can be determined using a performance model or could simply be a user configurable parameter. Finally, the scaling operation has to be carried out. For all auto-scalers, we will use Flinks' Reactive Mode to re-scale the dataflow topology. Flinks Reactive Mode sets the parallelism of all operators to the maximum value possible given the provided resources.

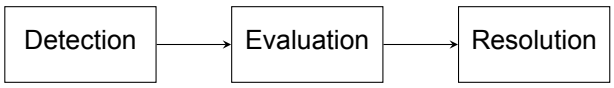

Figure 3.1: The three main steps an auto-scaler for stream processing has to implement.

3.2. Kubernetes HPA

The first implemented auto-scaler is the horizontal pod auto-scaler which is build into Kubernetes. The horizontal pod auto-scaler automatically adds or removes pods based on a certain metric. A pod is a group of one or more containers. The HPA will add or remove pods containing an Apache Flink Taskmanger application. The Apache Flink Jobmanager will then rescale the dataflow topology on the remaining Taskmanager pods.

3.2.1. Metrics

The target metric configured for the Kubernetes HPA is CPU utilization. CPU utilization is a commonly used metric to scale applications, as CPU utilization can give an indication of how much load an application is under. When there are multiple Taskmanagers, the CPU utilization of each task manager pod is averaged.

3.2.2. Evaluation

Once the Kubernetes HPA has obtained the CPU utilization value it is evaluated against a target CPU utilization to determine the ideal number of Taskmanagers. The HPA determines the ideal number of replicas using equation 3.1. As an example let's say the $desiredMetricValue$ is set to 60%, the $currentMetricValue$ average across all pods is 90% and $currentReplicas$ is 10. Then the $desiredReplicas = 10 * \frac{3}{2} = 15$.

$$desiredReplicas = ceil\left[currentReplicas * \frac{currentMetricValue}{desiredMetricValue}\right] \tag{3.1}$$

3.2.3. Implementation

The YAML file for the Kubernetes HPA can be found in Figure 3.2.3. The performance metric chosen can be seen on line 10 of Figure 3.2.3. The $minReplicas$ and $maxReplicas$ determine the minimum and maximum amount of Taskmanagers. For scaling down there is a default stabilization window of 5 minutes. When the HPA is evaluating the state of the system it looks at all desired states within the past 5 minutes and selects the state with the maximum desired number of Taskmanagers. This stabilization window tries to prevent the auto-scaler from oscillating.

```
1  apiVersion: autoscaling/v2beta2
2  kind: HorizontalPodAutoscaler
3  metadata:
4      name: flink-taskmanager
5      namespace: default
6  spec:
7      maxReplicas: 16
8      minReplicas: 1
9      metrics:
10     - type: Resource
11       resource:
12         name: cpu
13         target:
14             type: Utilization
15             averageUtilization: 70
16     scaleTargetRef:
17       apiVersion: apps/v1
18       kind: Deployment
19       name: flink-Taskmanager
```

Figure 3.2: Kubernetes YAML configuration for CPU based HPA.

3.3. Vargav1

The second implemented auto-scaler was proposed in a paper by Varga et al. [32]. The proposed auto-scaler leverages the Kubernetes HPA framework in combination with two custom metrics designed specifically for Apache Flink.

3.3.1. Metrics

Relative lag change The first metric used by Varga et al. is called Relative Lag Change. To obtain the change of the lag we must first obtain the lag. The lag refers to the number of records stored in the Kafka queue that have not yet been consumed by Apache Flink. Since the data is split up into partitions on the Kafka cluster, the total lag is the sum of the lag for each partition as can be seen in equation 3.2. Flink does not keep track of the lag for every partition only the maximum lag across all partitions, so the maximum lag value is used.

$$totalLag = \sum_{i \in partitions} records\_lag\_max_i * assigned\_partitions_i \tag{3.2}$$

To obtain the change in the $totalLag$ (in records/second) the derivative of the $totalLag$ is taken using the $deriv$ function available in PromQL. Varga et al. [32] then divide the derivative of the $totalLag$ by the rate at which the job is processing records. The rate at which the job is processing records is obtained by summing up the $records\_consumed\_rate$ for the source operators. Dividing the rate of change of the $totalLag$ by the $totalRate$ at which messages are consumed and adding 1 gives a metric that determines the required amount of replicas. The calculation for the $relativeLagChangeRate$ can be found in equation 3.3. The target value for the $relativeLagChangeRate$ is 1 as in that case the system can keep up with the load.

$$relativeLagChangeRate = 1 + \frac{deriv(totalLag)}{totalRate} \tag{3.3}$$

Utilization The second metric used by Varga et al. [32] is called Utilization. The Utilization metric uses the the $idleTimeMsPerSecond$ metric exposed by Apache Flink. This metric represents the time in milliseconds that an operator is idle. A task can be idle due to two reasons, there is no data to be processed, or data being processed by the system is bottle-necked at some downstream operator. The Utilization metric is calculated using equation 3.4 The target utilization is set to a number less than 1.

$$utilization = 1 - \frac{avg(idleTimeMsPerSecond)}{1000} \tag{3.4}$$

In the original implementation by Varga et al., the source operators were excluded from the utilization calculation due to the observation that the $idleTimeMsPerSecond$ was either 0 ms when the job was able to keep up with records and there was no lag or 1000 ms when the job was at maximum capacity. In Flink 1.14.3 the Kafka consumer was replaced with a Kafka Connector for which the $idleTimeMsPerSecond$ metric behaves normally so the source operators were not removed in our implementation.

3.3.2. Evaluation

Since the Vargav1 auto-scaler is implemented using the Kubernetes HPA framework the evaluation formula is the same as described previously in equation 3.1. However, for Vargav1 there are two metrics evaluated. When the Kubernetes HPA evaluates two metrics the higher desired replica count is always chosen.

Given the fact the higher of the two desired replica counts is chosen. A problem with the $relativeLagChangeRate$ arises. When the auto-scaler can keep up with the number of records and there is no lag it might be appropriate to scale down. However the $relativeLagChangeRate$ in this case is 1, so no down-scaling can occur as the $dersiredMetricValue$ equals the $currentMetricValue$ so the number of replicas remains constant. This problem can be resolved by adding another term to the equation for the Relative Lag Change metric which can be seen in equation 3.5 When the $totalLag$ is below the user set $threshold$ the $currentMetricValue$ becomes negative. The HPA will then only use the Utilization metric to make scaling down decisions as the utilization metric will yield the larger desired number of replicas.

$$\frac{totalLag - threshold}{abs(totalLag - threshold)} * relativeLagChangeRate \tag{3.5}$$

3.3.3. Implementation

Vargav1 was implemented using Kubernetes HPA YAML which can be seen in Figure 3.3.3. To use the Kubernetes HPA with custom metrics a Prometheus adapter [7] was installed on the cluster. The Prometheus adapter obtains and computes the metrics from Prometheus and makes them available to the Kubernetes HPA.

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: Taskmanager-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: flink-Taskmanager
  minReplicas: 1
  maxReplicas: 16
  metrics:
  - type: Pods
    pods:
      metric:
        name: utilization
      target:
        type: AverageValue
        averageValue: 0.7
  - type: Pods
    pods:
      metric:
        name: lag_rate
      target:
        type: AverageValue
        averageValue: 1
  behavior:
    scaleDown:
      stabilizationWindowSeconds: 480
```

Figure 3.3: Kubernetes YAML configuration for Vargav1.

3.3.4. Vargav2

After running experiments described in chapter 4 and observing the performance of Vargav1 described in chapter 5 we decided to try to improve Vargav1 by implementing a cooldown period. A cooldown period is beneficial for scaling stream processing systems because when the system re-scales, lag builds up. This lag needs to be processed when the stream processing system is back online causing a temporary spike in metric values when working away the lag. The Kubernetes HPA framework does not allow for the easy configuration of a cooldown period after scaling. We implemented a type of cooldown by multiplying both the Utilization and Relative Lag Change metric by the term in equation 3.6. The $cooldownPeriod$ is a user configurable parameter. When there has been a scaling operation during the $cooldownPeriod$ the derivative will be non-zero. The expression will then evaluate to zero. During the $cooldownPeriod$ the Utilization and Relative Lag Rate will both be zero. To prevent the HPA from scaling down when both metrics evaluate to zero the downscale stabilization window can be increased.

$$deriv(numRegisteredTaskmanagers[cooldownPeriod]) == 0 \tag{3.6}$$

3.4. Dhalion-adapted

The original implementation of the Dhalion [16] was much more than just an auto-scaler, it could also detect slow instances and data skew. We have decided to create a new auto-scaler inspired by some of the Dhalion features. In the DS2 paper by Kalavri et al. [20] experiments showed that DS2 outperformed Dhalion in every scenario. This is why we have decided to not implement the original version of Dhalion

but to create a new auto-scaler with some similarities to the original Dhalion implementation.

3.4.1. Metrics

The original implementation of Dhalion contained a backpressure detector [16], to detect backpressure. This is one of the features we keep from the original version as backpressure is a good indicator for an under-provisioned streaming dataflow.

In the original implementation, there was a pending packets detector. The pending packets detector determined the input buffer queue lengths of the operators. If the buffer queues are close to zero then the streaming dataflow can handle the input rate. If the buffer queues are non zero then the streaming dataflow could likely not handle the input rate. For our implementation, we will not be looking at buffer queues directly but at the Kafka queue in-front of the streaming application. If the Kafka queue length is close to zero then the streaming dataflow can handle the input rate. If the Kafka queue length is non zero then the streaming dataflow can likely not keep up. However, instead of looking at the length of the Kafka queue, we look at the consumer latency. The consumer latency represents the average amount of time a record spends in the Kafka queue. The consumer latency is more interpretable when compared to the Kafka queue size.

We also use the Average CPU utilization, Kafka input rate, the derivative of Kafka queue lag, and throughput, the reasoning for these metrics will be explained in the evaluation section. A list of all metrics used for our Dhalion implementation can be seen below:

- **Maximum backpressure:** this metric returns the maximum of the amount of time that each operator is backpressured in ms.
- **Average CPU utilization:** the average CPU utilization of the Taskmanagers.
- **Average event time lag:** the amount of time that a record spends in the Kafka queue averaged by partition.
- **Kafka input rate:** the number of records per second written into Kafka.
- **Throughput:** the number of input records to the Sink.
- **Derivative of Kafka lag:** the number of records the Kafka lag is increasing/decreasing per second.

3.4.2. Evaluation

The stream processing system is under-provisioned if the average event time lag is above a user specified threshold and the derivative of the Kafka lag is positive. If the Kafka lag is increasing then the streaming processing system cannot keep up with the input rate. The event time lag threshold ensures that a scale up is not triggered due to noise.

When the stream processing system can keep up with the input rate there is no event time lag, if there is also no backpressure and the CPU utilization is low then the stream processing system is likely over-provisioned. The evaluation criteria for over-provisioning is therefore: the maximum backpressure is below a user specified threshold, the event time lag is below a user specified threshold and the CPU utilization is below a user specified threshold.

In the original implementation of Dhalion [16] the scale up factor was determined for each operator individually. The scale up factor was computed by dividing the amount of time data consumption was suspended by the amount of time the operator was processing records normally. For example, if 20% of the time data consumption was suspended and 80% data flow was normal then the scale up factor is $\frac{20}{80} = 25\%$. Instead of calculating the parallelism for each operator, we will take a global approach to determine the scale up factor. We look at the throughput of the entire system and the input rate to the Kafka cluster. The scale up factor is then $\frac{KafkaInputRate}{Throughput}$, as an example say the input rate to the Kafka queue is 200 records per second and the throughput is 100 records per second then the scale up factor is $\frac{200}{100} = 2$. The scale up factor only works for scaling up because when there is lag the throughput will be at a maximum value. When the cluster is over-provisioned the throughput will be equal to the input rate of the Kafka cluster. To scale down we will use a user configurable percentage, which is the same method as in the original Dhalion implementation. We also implemented an over-provisioning factor and cooldown period.

3.4.3. Implementation

Dhalion-adapted is deployed to the Kubernetes cluster as a python script running in a container. Metrics are queried from the Prometheus REST API. The user configurable parameters are:

- **COOLDOWN** The minimum amount of time between scaling operations.
- **LATENCY_THRESHOLD** The event time lag threshold.
- **OVERPROVISIONING_FACTOR** The percentage to over-provision the Taskmanagers by.
- **BACKPRESSURE_THESHOLD** The minimum backpressure threshold value.
- **CPU_THESHOLD** The minimum CPU utilization threshold value.
- **MIN_REPLICAS** The minimum amount of replicas.
- **MAX_REPLICAS** The maximum amount of replicas.
- **SCALING_FACTOR_PERCENTAGE** The percentage to scale the Taskmanager down by.

3.5. DS2-modern

The fifth implemented auto-scaler is a modern version of DS2 created by Kalavri et al. [20]. DS2 works by using knowledge of the topology of the dataflow graph in combination with measured processing rates to determine the true processing rate of operators. Once the true processing rates of the operators are known the optimal parallelisms can be computed.

3.5.1. Metrics

When Kalavri et al. [20] implemented DS2 in Apache Flink version 1.4.1 in 2018, few metrics were exposed natively by Apache Flink. Kalavri et al. edited the source code of Apache Flink to measure the input rate per operator, output rate per operator, and the useful time per operator. In 2022 the patch applied to create those metrics no longer works on Flink 1.14.3. However, Flink now provides similar metrics to those implemented by Kalavri et al. therefore we decided to use the metrics provided by Apache Flink to implement DS2. The metrics we used to implement DS2 are:

- $numRecordsInPerSecond$ - The number of records the operator receives per second.
- $numRecordsOuPerSecond$ - The number of records the operator outputs per second.
- $busyTimeMsPerSecond$ - The time (in milliseconds) the operator is busy (neither idle nor back pressured) per second.
- $kafka\_server\_brokertopicmetrics\_messagesin\_total$ - The number of records the Kafka cluster has received.

3.5.2. Evaluation

To understand how DS2 evaluates the state of the streaming dataflow we first have to establish some definitions. Kalavri et al. define the following terms:

Definition 3.5.1 (Useful time)**.** The time spent by an operator instance in deserialization, processing, and serialization activities.

When the useful time is measured, it can be used to calculate the true processing rate.

Definition 3.5.2 (True processing rate)**.** The number of records an operator instance can process per unit of useful time.

In our implementation, we use the $busyTimeMsPerSecond$ metric described in the previous section as the useful time equivalent. We chose the $busyTimeMsPerSecond$ metric because it measures the duration an operator is actually processing and not idle or backpressured. The processing time includes the serialization, deserialization, and processing times which were measured in the original implementation of the useful time metric by Kalavri et al. [20]. The true processing rate is calculated

by dividing the input rate per operator by the useful time of that operator. Using the true processing rates and knowledge of the dataflow topology the optimal parallelism can be computed using equation 3.7 There is one problem though, the source operators do not have an upstream operator. So for the Kafka connector operators we use the Kafka input rate as the aggregate true output rate of the upstream operator when determining the optimal parallelism.

$$\text{Optimal parallelism for operator } o_i = \frac{\text{aggregated true output rate of upstream operators}}{\text{average true processing rate of } o_i} \tag{3.7}$$

To calculate the optimal parallelism of the first operator in the dataflow topology DS2 used a statically set input rate as the numerator in equation 3.7. We however want DS2 to adjust to a dynamic input rate. So for the first operator in the dataflow topology, we set the numerator for the in equation 3.7 equal to the rate of the number of messages being produced to the Kafka cluster. This rate can be obtained by using the PromQL rate function on the $kafka\_server\_brokertopicmetrics\_messagesin\_total$ metric.

3.5.3. Implementation

In the original version of DS2 the parallelism was adjusted per operator. We are only interested in the maximum parallelism outputted by DS2 as this is what we re-scale the number of Taskmanager to, Flinks Reactive Mode will then scale each operator to the maximum parallelism. Scaling operators individually results in more scaling overhead when running on Kubernetes because the Jobmanager executing the query has to be stopped and a new one has to be started. We implement the DS2 evaluation script in a container by using the original Rust code[1] which was open sourced by Kalavri et al. [20]. Metrics are collected from Prometheus to compute the processing rates. The true processing rates and dataflow topology are then used as inputs to the DS2 script, which outputs the optimal parallelism for the dataflow graph. The maximum parallelism is then used as the required number of Taskmanagers. We also implemented a cooldown period after scaling events and an overprovisioning factor. The user configurable parameters are:

- **COOLDOWN** The minimum amount of time between scaling operations.
- **OVERPROVISIONING_FACTOR** The percentage to overprovision the Taskmanagers by.
- **MIN_REPLICAS** The minimum amount of replicas.
- **MAX_REPLICAS** The maximum amount of replicas.

3.5.4. DS2-modern-adapted

The original version of DS2 does not take into account the lag built up in the Kafka queue. Which led to some poor results explained in detail in section 5.12. The DS2-modern implementation is only concerned with the input rate into the Kafka queue and whether it can handle that rate. To make DS2 aware of the lag in the Kafka queue we add the event time lag metric. If the event time lag is above a certain threshold then scaling up is allowed, if the Event time lag is below a certain threshold then scaling down is allowed. These parameters are meant to stabilize DS2-modern as for example DS2-modern-adapted cannot scale down if there is still lag. The user configurable parameters are:

- **COOLDOWN** The minimum amount of time between scaling operations.
- **OVERPROVISIONING_FACTOR** The percentage to overprovision the Taskmanagers by.
- **MIN_REPLICAS** The minimum amount of replicas.
- **MAX_REPLICAS** The maximum amount of replicas.
- **SCALE_UP_LAG_THRESHOLD** The event time lag threshold before scaling up is allowed.
- **SCALE_DOWN_LAG_THRESHOLD** The event time lag threshold before scaling down is allowed.

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

3.6. Conclusion

In this section we have described the metrics, evaluation methods, and implementation of six different auto-scalers. Starting from the simplest auto-scaler a Kubernetes HPA which scales based on CPU utilization. A more complicated Kubernetes HPA utilizing custom metrics proposed by Varga et al. [32] designed for stream processing. A second version of the Varga auto-scaler with a cooldown period. An adapted implementation of Dhalion using a variety of metrics along with an evaluation strategy based on Kafka input rates and throughput. Finally, we implemented DS2 using metrics provided by Apache Flink. We also adapt DS2 to take into account the Kafka queue as the original implementation was not designed with a queue application in front of the stream processing system. In the next chapter we will describe the experimental setup used to compare these different auto-scalers.

4

Experimental design

In this chapter, we describe the experimental setup used to compare the different auto-scalers. We explain the dataflow topologies of different queries from the Nexmark dataset [29], the setup of our Kubernetes cluster, the different auto-scaler configurations, the varied parameters across experiments, and the load pattern used.

4.1. Nexmark Queries

To evaluate the different auto-scalers we use three different queries from the Nexmark benchmark suite [29] for stream processing applications. The Nexmark dataset contains queries over three entity models representing an online auction application [29]. The three entity models are:

- Person: a person submitting an item for auction or making a bid.
- Auction: represents an item under auction.
- Bid: represents a bid for an item under auction.

The three chosen queries represent some basic operations possible in Apache Flink but by no means represent all possible operations. The reason for testing the auto-scaling applications with different queries is that different queries have different dataflow topologies. In a longer dataflow graph backpressure will take longer to propagate than in a shorter dataflow graph. The different queries also have different operators which can produce different metric behaviors. The behavior of the metrics monitored by each auto-scaler will thus differ per query. To account for this variable behavior per query we test the auto-scalers on the three queries specified in the next section.

4.1.1. Query 1

The first query we use is query 1 in the Nexmark benchmark. The query answers the question ”what are the values of the bids in euros?” The query illustrates a simple mapping operation, where the bid values are converted from USD into euros. In the original implementation of the Nexmark benchmark, a custom Flink source was constructed which generated the bids. In our setup, we create a Kafka producer which produces bids to a Kafka topic. The bids are then read by the Flink Kafka connector source. Finally, the records are disposed of with a Sink operator. The dataflow graph of Query 1 can be seen in Figure 4.1

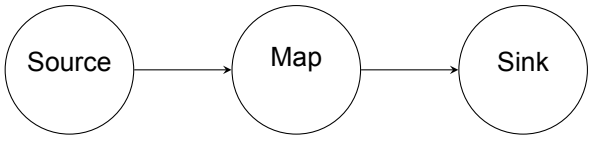

Figure 4.1: Dataflow graph of Nexmark query 1.

4.1.2. Query 3

The second query chosen for our experiments is query 3 from the Nexmark benchmark. Query 3 is more complicated than query 1 as it contains multiple sources and joins them together. Query 3 answers the question: ”What people are selling in the states Oregon, Idaho, or California, and in what auctions?” For those familiar with SQL the query can be seen in Listing 4.1.2. The dataflow graph of this query can be seen in Figure 4.2.

```
1 SELECT Istream(P.name, P.city, P.state, A.id)
2 FROM Auction A [ROWS UNBOUNDED], Person P [ROWS UNBOUNDED]
3 WHERE A.seller = P.id AND (P.state = 'OR' OR P.state = 'ID' OR P.state = 'CA');
```

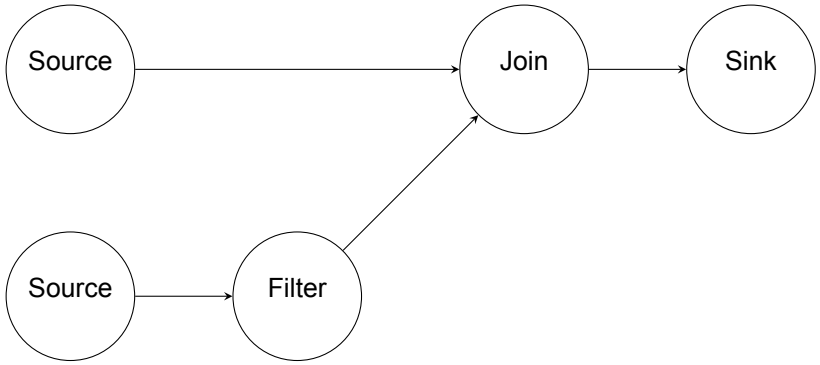

Figure 4.2: Dataflow graph of Nexmark query 3, the bottom source represents a Kafka connector consuming a datastream of Persons from Kafka. The person stream is filtered to only contain persons from Oregon, Idaho, and California. The top source is a Kafka connector consuming a datastream of Auctions from Kafka. The two data streams are joined in the join operator and discarded by the Sink operator.

4.1.3. Query 11

The third query answers the question: ”How many bids did a user make in each session they were active?” This question can be answered using session windows. In this query, a session window collects the bids for a specific user until there is a gap larger than a certain amount of seconds in between bidding events. As an example let’s say we have four bids all from the same user. The bids were generated at minutes 0, 1, 3, and 7 the session window size is set at 3 minutes. Then bids 0,1,3 would be grouped into one session and the bid with event time 7 would be in its own session. Within these windows, the aggregation would be performed. Query 11 is different from the other two queries because it is stateful. The session windows collect and hold previous records, if the system were to crash these session windows would have to be reloaded or the computation would be incorrect. The SQL equivalent of query 11 can be seen in Listing 4.1.3. The dataflow topology of query 11 can be seen in Figure 4.3.

```
1 SELECT  bid.bidder, COUNT(*) FROM bid TIMESTAMP BY bid.datetime
2 GROUP BY bid.bidder, SessionWindow(seconds, 10)
```

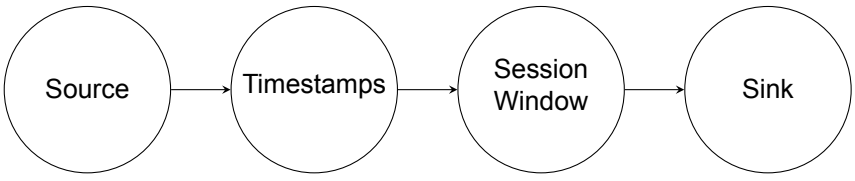

Figure 4.3: Dataflow graph of Nexmark query 11, bids are read from Kafka using the Kafka connector source. The timestamp of the bids is then made available to Apache Flink using the Timestamps operator. The bids are then collected and aggregated in the Session Window before the aggregated data is sent to the Sink.

4.2. Load Profile

The load experienced by a stream processing system can vary greatly per use case. Consumer-facing applications typically have some form of periodic behavior while other systems might have static or

more random load patterns. To test both the up-scaling and down-scaling capabilities of the auto-scalers have chosen to generate a sinusoidal load pattern for each query mentioned above. The rate at which records are produced is updated every minute i using equation 4.1. We also add some noise to the sinusoidal load pattern to make sure that the tested auto-scalers can handle a noisy environment. To add the noise to the signal we add $randomValue$ to the signal which is a positive or negative number in a certain range.

$$RecordsPerSecond_i = randomValue + verticalShift + Amplitude * cos(i * \frac{2 * \pi}{period}) \quad (4.1)$$

4.3. Evaluation Metrics

We will compare the different auto-scalers on three different metrics, the average latency of the records in the Kafka queue (also know as the event time lag) for the duration of the experiment. The average number of Taskmanagers for the duration of the experiment and the number of re-scaling operations. The average amount of Taskmanagers will allow us to compare the amount of resources used by the different auto-scalers. Stream processing applications often have latency SLAs so we use the average latency as one of our evaluation metrics. We expect to see a trend that with more resource consumption the average latency will go down. The number of rescaling operations is chosen as an evaluation metric as in our setup the rescale operations do not incur large overhead due to the small experiment scale. In real-world use cases re-scaling can incur a much larger overhead so auto-scalers that scale less frequently are preferred.

4.4. Experimental Setup

To test the different auto-scalers we set up a Kubernetes cluster using the Google Cloud Kubernetes engine. The cluster contains three Kafka brokers. A load generator that produces records to a Kafka topic. Apache Zookeeper which coordinates the Kafka brokers. Flink Taskmanagers which execute the operations of the streaming dataflow. A Flink Jobmanager which manages the streaming dataflow coordinating the Taskmanagers and taking checkpoints and savepoints. Checkpoints and savepoints are stored by mounting a locally running NFS server as an NFS volume on the Flink Jobmanagers and Taskmanagers. The NFS server uses a GCP persistent disk with 10GB of storage to save files. The cluster also contains a Prometheus pod which scrapes and stores all metrics from applications within the cluster. An overview of the different applications present in the cluster can be seen in Figure 4.4 The versions of the various technologies used are: Google cloud GKE version: 1.21.6-gke.1503, Node type: e2-standard-4 (4 vcPU, 16 GB memory), Number of nodes: 4, Apache Kafka version: 2.7.0, Apache Flink version: 1.14.3, Zookeeper version: 3.4.10, Prometheus version: 2.34.0, Prometheus adapter version: 0.9.1.

4.5. Experiment Monitoring

Given the experimental setup described in the previous section, a large number of metrics will be collected by Prometheus. To observe the health of the streaming dataflow we only need to look at a subset of these metrics. In Figure 4.5 the important metrics used to monitor the health of the streaming dataflow are shown. The Kafka input rate represents the number of records per second written to the Kafka brokers. Taskmanagers represents the number of registered Taskmanagers available to the Flink Jobmanager. The lag represents number of records stored in the Kafka queue not yet consumed by Flink. The latency represents the average number of seconds that a record spends waiting in the Kafka queue. The CPU utilization represents the average CPU utilization of the Taskmanagers. The busy time represents the average busy time per operator. The idle time represents the average idle time per operator. Finally, the backpressure represents the maximum backpressure of all operators. These metrics will be used in the next chapter to explain experimental results.

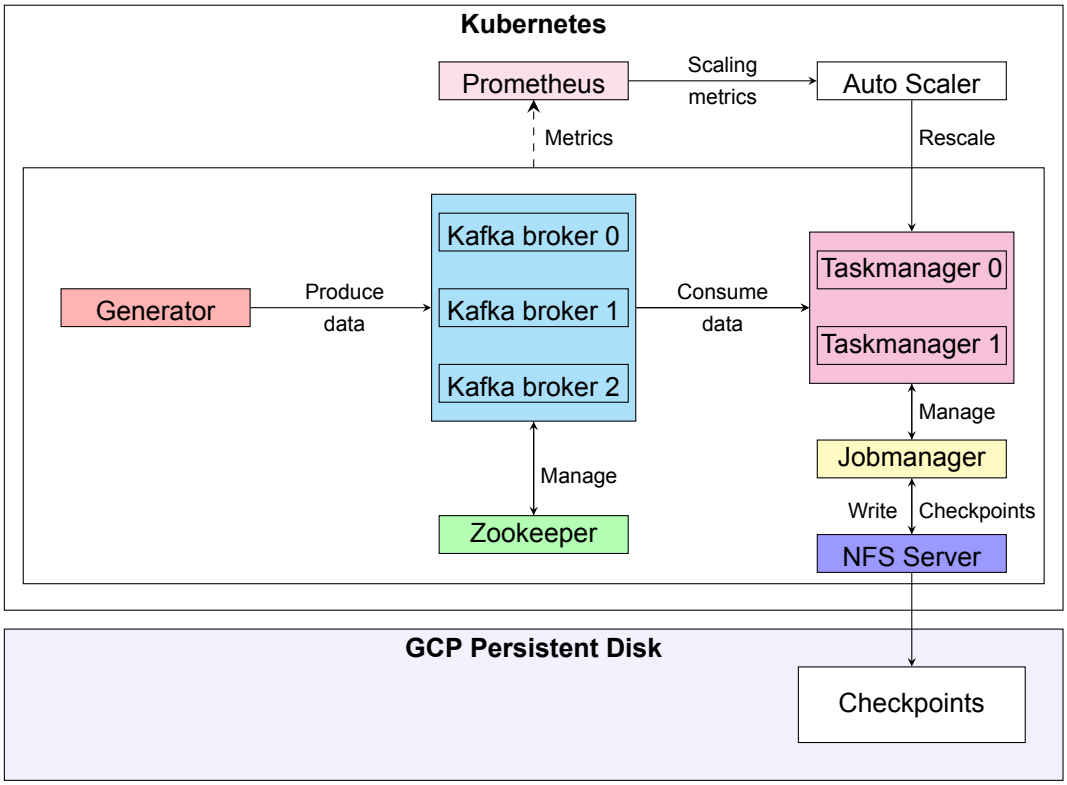

Figure 4.4: An overview of the pods contained in the Kubernetes cluster used to run experiments.

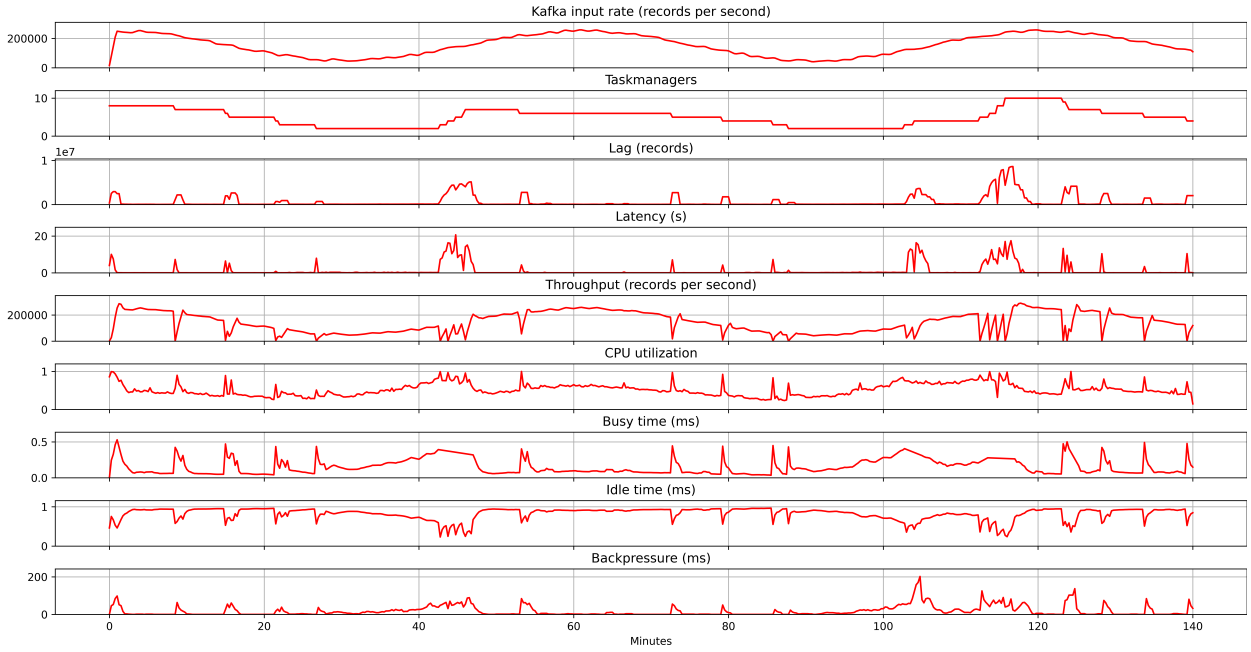

Figure 4.5: Telemetry metrics collected per experiment.

4.6. Experiment Parameters

For each of the three queries, we use a slightly modified load pattern. The load pattern is different per query because the computational complexity is different per query. Therefore the three experiment queries have different values for the load generation equation 4.1. The values used for the load generation function can be seen in Table 4.1. For query 3 there are two source functions the rate of each source function is half the value obtained from equation 4.1.

Query	Experiment Time (minutes)	period (minutes)	yshift	amplitude	randomValue (range)
query-1	140	60	150,000	100,000	(-10000, 10000)
query-3	140	60	50,000	25,000	(-10000, 10000)
query-11	140	60	100,000	50,000	(-10000, 10000)

Table 4.1: Parameters for load generation function described in equation 4.1 for each of the three queries.

For each query, we run the auto-scalers with three different values of a specific metric. For the CPU HPA, we change the target CPU utilization as this is the only metric the CPU HPA uses to scale. For Vargav1 and Vargav2 we change the target $utilization$ because the utilization has a wider range of values depending on the state of the streaming dataflow and thus has a larger effect on scaling decisions than the $relativeLagChangerate$ which according to Varga et al. [32] should have a target of 1 or slightly below 1 to add some over-provisioning. For Dhalion-adapted there are quite a few configurable parameters. We chose to adjust the $LATENCY\_THRESHOLD$ to see if scaling earlier or later would have an effect. We did not alter other parameters as we had limited cloud computing budget. For DS2-modern and DS2-modern-adapted we only change the $OVERPROVISIONING\_FACTOR$ as DS2-modern does not have any other parameters except $COOLDOWN$ which we decided to keep constant across experiments to limit experiment count. The values for all auto-scalers were simply best guesses, no hyper-parameter tuning has taken place due to the excessive effort that would be required. All auto-scalers have the same minimum amount of 1 Taskmanager and maximum amount of 16 Taskmanagers. The configurable parameters for the autos-scalers are as follows:

CPU HPA
- **varied parameter: CPU utilization**, values 50, 70, 90

Vargav1
- Stabilization window: 480 seconds
- Threshold: 50000 records
- **varied parameter: utilization**, values 0.3, 0.5, 0.7

Vargav2
- Stabilization window: 480 seconds
- Cooldown period: 2 minutes
- Threshold: 50000 records
- **varied parameter: utilization**, values 0.3, 0.5, 0.7

Dhalion-adapted
- COOLDOWN: 120 seconds
- OVERPROVISIONING_FACTOR: 20%
- BACKPRESSURE_THESHOLD: 100 ms
- CPU_THESHOLD: 60%
- SCALING_FACTOR_PERCENTAGE: 20%
- **varied parameter: LATENCY_THRESHOLD**: 1, 5, 10 seconds

DS2-modern
- COOLDOWN: 240 seconds
- **varied parameter: OVERPROVISIONING_FACTOR**, values 0%, 33%, 66%

DS2-modern-adapted
- COOLDOWN: 240 seconds
- SCALE_UP_LAG_THRESHOLD: 5 seconds

- SCALE_DOWN_LAG_THRESHOLD: 1 second
- **varied parameter: OVERPROVISIONING_FACTOR**, values 0%, 33%, 66%

4.7. Conclusion

In this chapter we described the experimental setup. We explained the topologies of Nexmark queries 1,3 and 11. The sinusoidal load profile used during the experiments. The three evaluation metrics used to compare the different autoscalers: average number of Taskmanagers, average latency, and number of rescaling operations. We also discussed the applications running in our Kubernetes environment. Finally, we described the auto-scaler parameters used throughout the experiments. In the next chapter we will discuss the results of these experiments.

5 Results

In this chapter, we discuss the results of our experiments. We show scatter plots of the different auto-scalers with different parameters for each query type. We also show the actual numerical results of the experiments including the number of rescaling operations per auto-scaler in section 5.7. Finally, we investigate individual experiments to explain the causes behind some unexpected results.

5.1. Query 1

In Figure 5.1 we can see the results for query 1. Most auto-scalers are towards the left of the Figure at around an average of 3 to 4 Taskmanagers. This indicates that the auto-scalers are using resources efficiently. Within this 3 to 4 auto-scaler range, there is a large difference in the average latency between the different auto-scalers. The pattern observed by the points plotted on the scatter plot is not unexpected as there is a trade-off between the number of resources used and average latency. If a lot of resources are used then the average latency is low, if few resources are used then the average latency is expected to be higher. Vargav1 appears to be an outlier in regards to this trend, the reason for this will be discussed later in section 5.9.

5.2. Query 3

In Figure 5.3 we can see the same pattern as for query 1. Most auto-scalers are clustered towards the left of the scatter plot, which is again not unexpected as auto-scalers try to use resources efficiently. The average number of Taskmanagers required does appear to be higher for most auto-scalers. This is due to the increased computational complexity of query 3. The average latency is much higher for most auto-scalers for query 3 when compared to query 1, this can again be attributed to the increased complexity of the dataflow topology.

5.3. Query 11

In Figure 5.5 again we see auto-scalers clustered towards the left between 4 and 6 Taskmanagers on average. However, there are quite a few outliers that were not observed in query 1 and query 3. These outliers which have both high average latency and high average number of Taskmanagers include Dhalion-adapted and the CPU HPA with a utilization target of 70%. This is unexpected as we expect auto-scalers with a high number of average Taskmanagers to have a low latency as observed in queries 1 and 3. The cause for these outliers has to do with backpressure buildup explained in detail in section 5.8.

5.4. Results Query 1

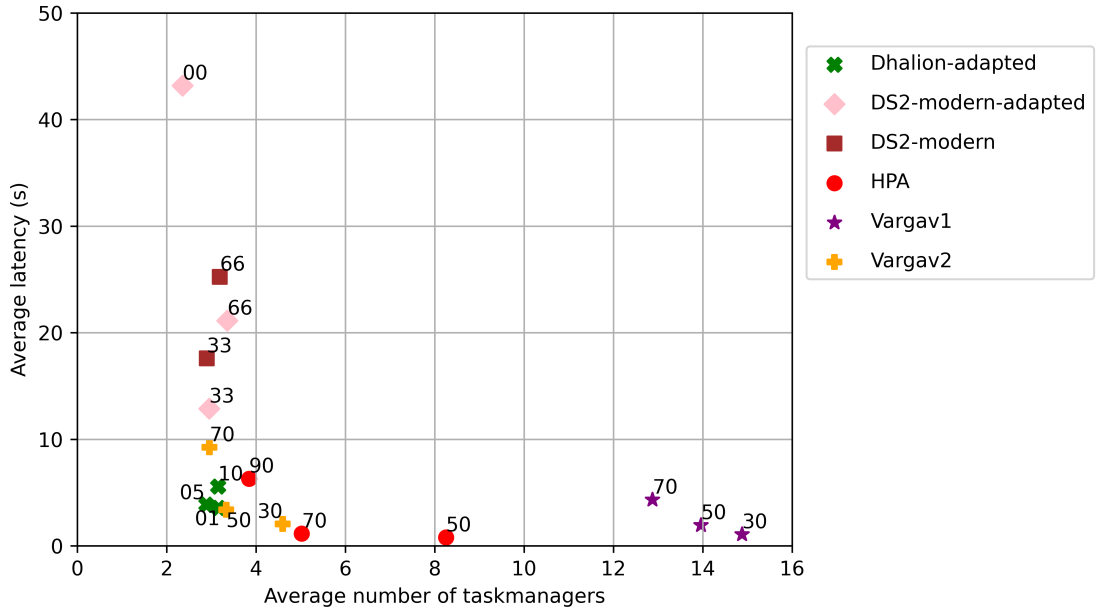

Figure 5.1: Scatter plot showing the average number of Taskmanagers used and average latency for the different auto-scalers for query 1.

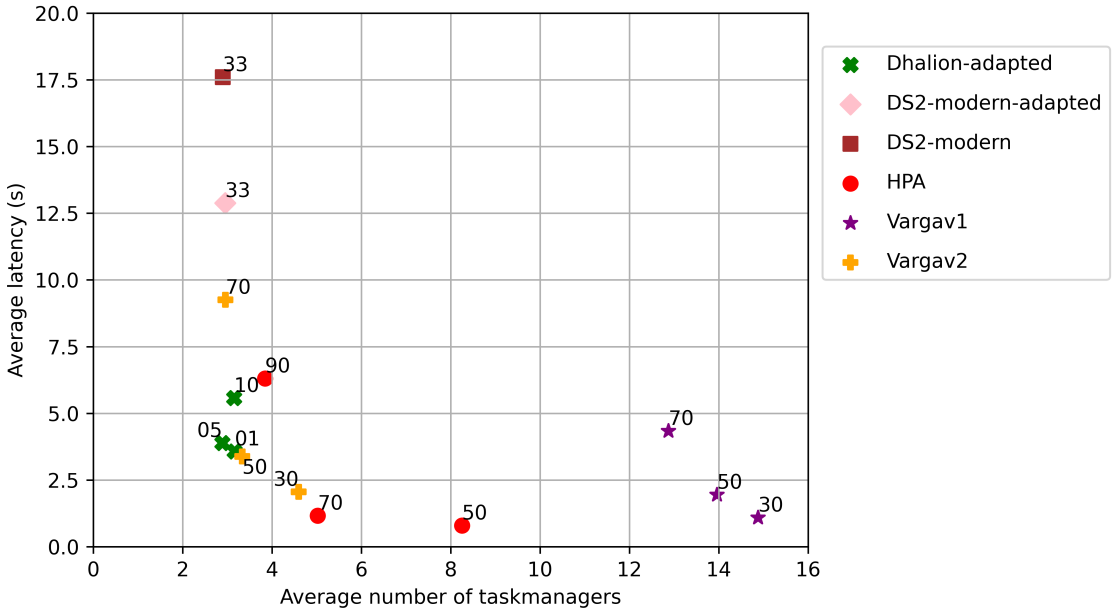

Figure 5.2: Zoomed in version of scatter plot showing the average number of Taskmanagers used and average latency for the different auto-scalers for query 1.

5.5. Results Query 3

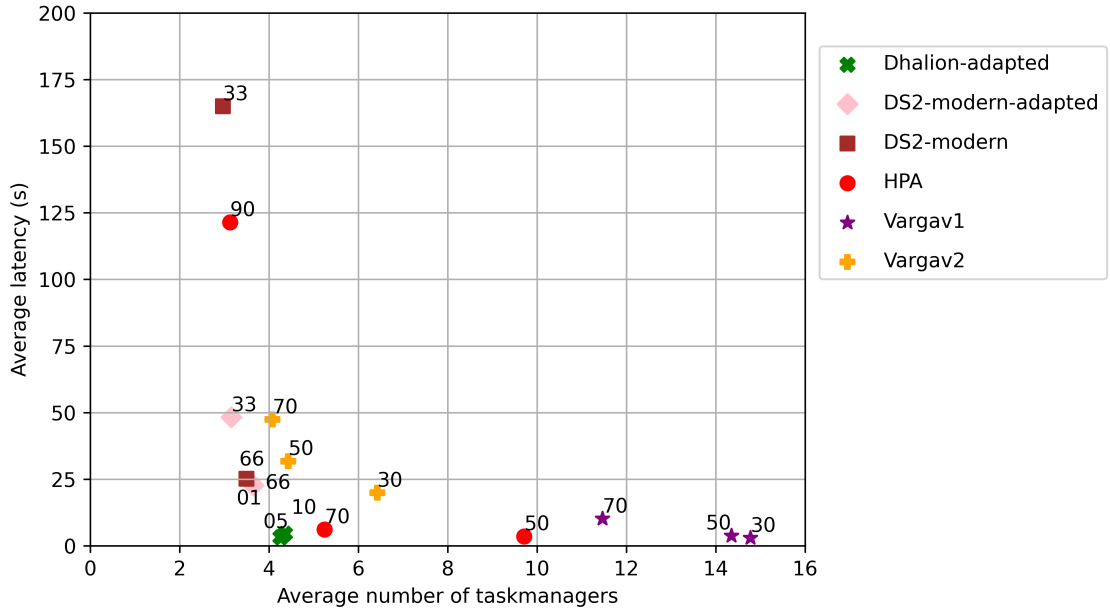

Figure 5.3: Scatter plot showing the average number of Taskmanagers used and average latency for the different auto-scalers for query 3.

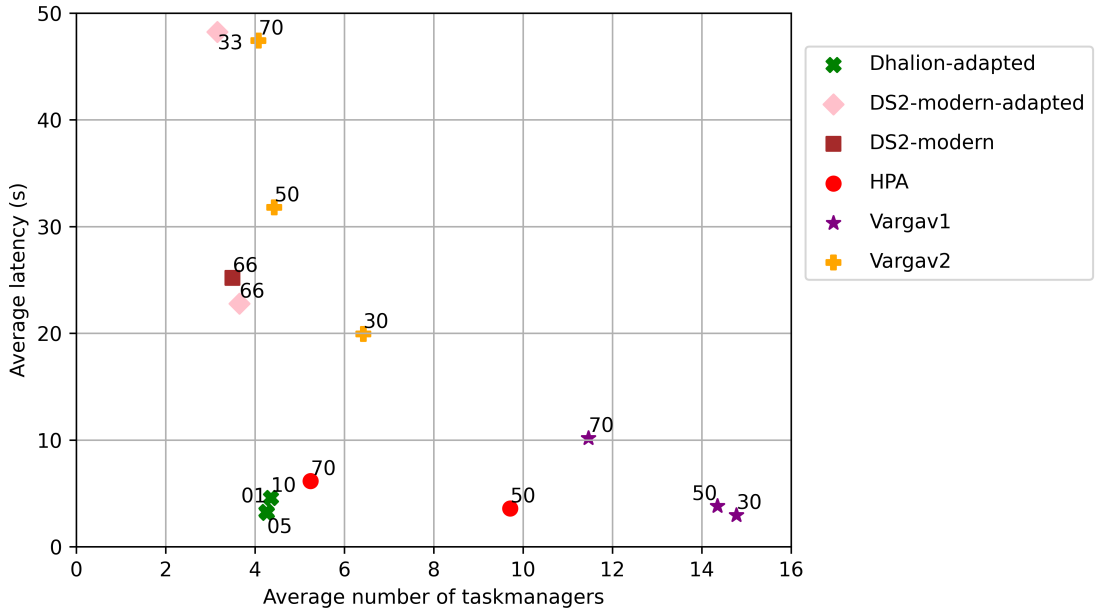

Figure 5.4: Zoomed in version of scatter plot showing the average number of Taskmanagers used and average latency for the different auto-scalers for query 3.

5.6. Results Query 11

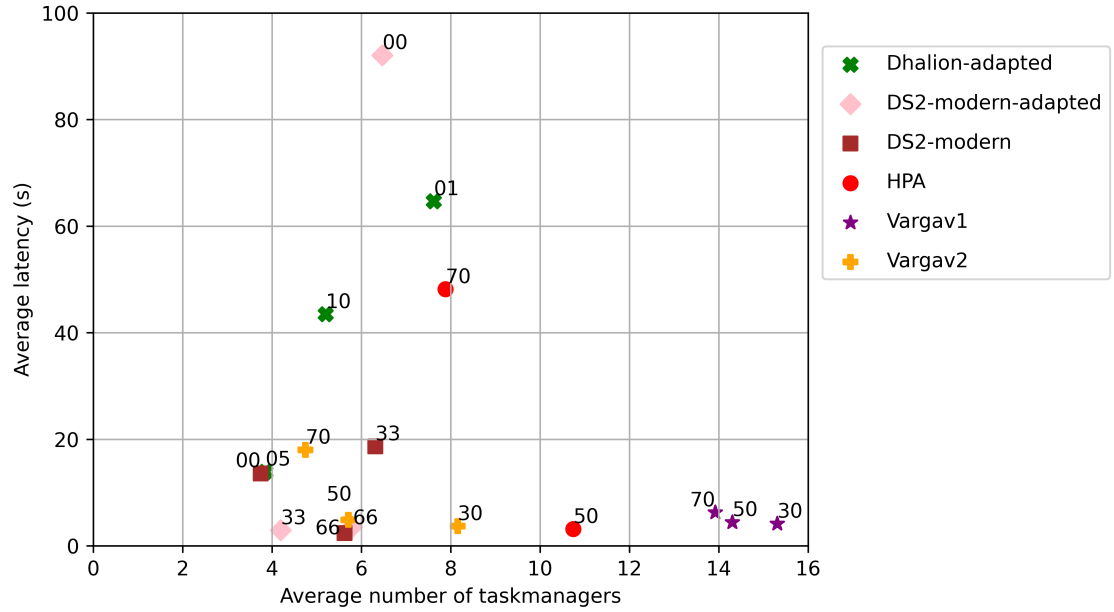

Figure 5.5: Scatter plot showing the average number of Taskmanagers used and average latency for the different auto-scalers for query 11.

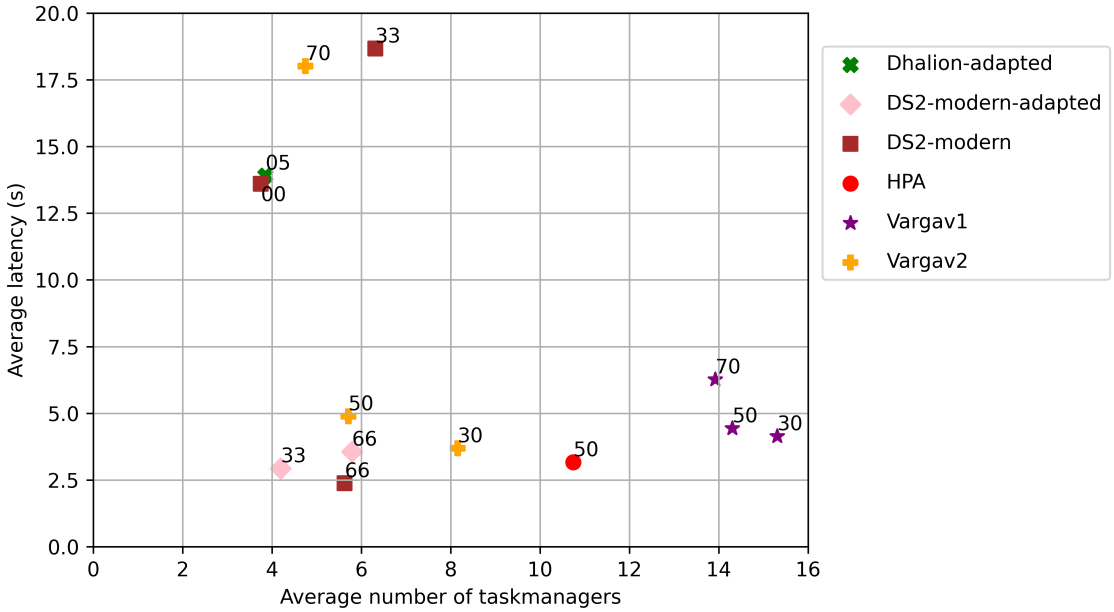

Figure 5.6: Zoomed in version of scatter plot showing the average number of Taskmanagers used and average latency for the different auto-scalers for query 11.

5.7. Summarized Results

Auto-scaler	Metric value	Latency (s)	Taskmanagers	Latency + Taskmanagers	Scaling operations
HPA	70	1.16	5.02	6.18	28
Vargav2	30	2.05	4.60	6.65	26
Vargav2	50	3.39	3.33	6.72	17
Dhalion-adapted	01	3.58	3.16	6.74	22
Dhalion-adapted	05	3.88	2.89	6.77	20
Dhalion-adapted	10	5.58	3.15	8.72	23
HPA	50	0.79	8.25	9.04	40
HPA	90	6.31	3.84	10.15	16
Vargav2	70	9.25	2.96	12.21	13
DS2-modern-adapted	33	12.88	2.95	15.83	16
Vargav1	50	1.95	13.96	15.91	39
Vargav1	30	1.09	14.88	15.97	40
Vargav1	70	4.34	12.87	17.21	49
DS2-modern	33	17.59	2.90	20.49	16
DS2-modern-adapted	66	21.13	3.36	24.49	21
DS2-modern	66	25.24	3.19	28.43	24
DS2-modern-adapted	00	43.19	2.35	45.54	11
DS2-modern	00	121.59	2.19	123.78	13

Table 5.1: Query 1 experiment evaluation metrics per auto-scaler.

Auto-scaler	Metric value	Latency (s)	Taskmanagers	Latency + Taskmanagers	Scaling operations
Dhalion-adapted	05	3.18	4.27	7.45	23
Dhalion-adapted	01	3.27	4.25	7.53	24
Dhalion-adapted	10	4.58	4.36	8.93	30
HPA	70	6.16	5.24	11.41	27
HPA	50	3.58	9.71	13.29	42
Vargav1	30	2.97	14.78	17.74	37
Vargav1	50	3.82	14.35	18.17	43
Vargav1	70	10.18	11.46	21.65	47
Vargav2	30	19.94	6.42	26.36	44
DS2-modern-adapted	66	22.78	3.65	26.42	16
DS2-modern	66	25.18	3.50	28.68	20
Vargav2	50	31.79	4.43	36.22	32
DS2-modern-adapted	33	48.23	3.16	51.39	11
Vargav2	70	47.43	4.07	51.50	33
HPA	90	121.42	3.13	124.55	8
DS2-modern	33	164.96	2.97	167.93	22
DS2-modern-adapted	00	247.74	3.11	250.85	9
DS2-modern	00	971.28	2.52	973.80	31

Table 5.2: Query 3 experiment evaluation metrics per auto-scaler.

Auto-scaler	Metric value	Latency (s)	Taskmanagers	Latency + Taskmanagers	Scaling operations
DS2-modern-adapted	33	2.94	4.20	7.13	14
DS2-modern	66	2.38	5.63	8.01	28
DS2-modern-adapted	66	3.56	5.79	9.35	15
Vargav2	50	4.89	5.71	10.60	27
Vargav2	30	3.69	8.15	11.84	29
HPA	50	3.16	10.74	13.91	44
DS2-modern	00	13.60	3.75	17.35	24
Dhalion-adapted	05	13.91	3.85	17.76	22
Vargav1	50	4.44	14.30	18.74	40
Vargav1	30	4.14	15.31	19.45	36
Vargav1	70	6.27	13.92	20.19	38
Vargav2	70	18.01	4.75	22.76	29
DS2-modern	33	18.67	6.31	24.98	25
Dhalion-adapted	10	43.46	5.20	48.66	27
HPA	70	48.16	7.88	56.04	26
Dhalion-adapted	01	64.66	7.62	72.28	29
DS2-modern-adapted	00	92.06	6.47	98.53	18
HPA	90	428.35	2.73	431.08	7

Table 5.3: Query 11 experiment evaluation metrics per auto-scaler.

5.8. HPA

The HPA performs well for queries 1 and 3 but performs poorly for query 11.
For query 1 the CPU HPA performs well for all three CPU utilization values as can be seen in Figure 5.2. When the target CPU utilization value is increased, fewer resources are used. However, this comes at the cost of some average latency. For query 3 the CPU HPA still performs quite well with a target value of 70 % as can be seen in Figure 5.4. For query 3 the performance of the HPA with a target CPU utilization of 90% is poor because backpressure builds up due to not scaling. When backpressure builds up the average CPU utilization stays at around 90% which is the target so no scaling command is issued. The HPA with a target CPU utilization of 50% has low average latency but used almost twice as many resources as the HPA with a target CPU of 70% as can be seen in Table 5.2. For query 11 all the HPA perform quite poorly compared to the other auto-scalers as can be seen in Figure 5.5.

The HPAs poor performance for query 11 is a result of a scale-up loop which could be prevented with a cooldown period.
In Figure 5.7 we see some key experiment metrics for the HPA with a target CPU utilization of 70%. At around 50 minutes the HPA makes a decision to scale up by one Taskmanager. Due to the scaling operation, some latency builds up. Flink starts processing records again and the throughput is higher than before the scaling operation. Instead of working away the lag, the HPA decides to scale up again, resulting in more lag build-up. When Flink is reconfigured it starts to process the records stored in the Kafka queue spiking the CPU utilization again triggering a scale-up operation. The HPA gets stuck in this loop and scales up to the maximum number of allowed Taskmanagers. This can be prevented by having a cooldown period after scaling but the Kubernetes HPA does not provide a configurable cooldown period out of the box. After the maximum number of Taksmanagers is reached we expect the throughput to go up very high as all the lagged records are processed away. However, this does not happen as can be seen between minutes 70 and 90. The throughput is ”stuck” at around 100,000 records per second. The throughput is limited because the job is backpressured as can be seen in the Backpressure plot in Figure 5.7. The backpressure is high due to the computationally and memory-intensive session windows unique to query 11.

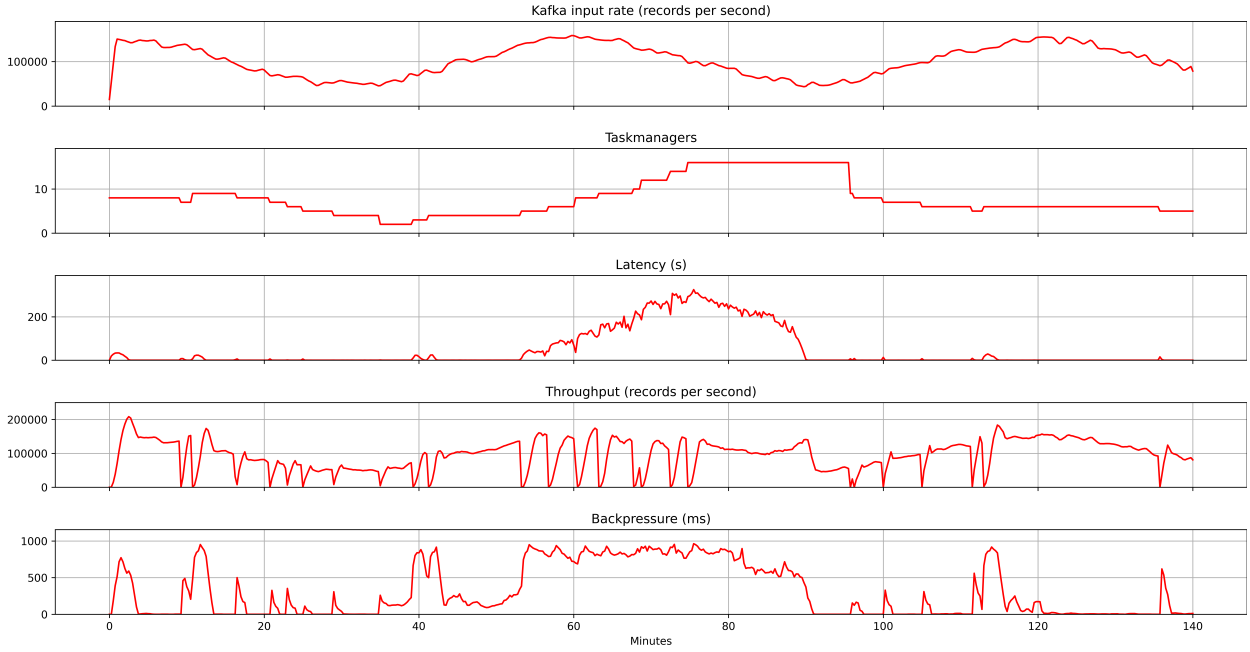

Figure 5.7: Telemetry data of HPA with target CPU utilization of 70% for query 11.

5.9. Vargav1

Vargav1 has low average latency but high average number of Taskmanagers due to constantly being overprovisioned.

Vargav1 performed poorly in all experiments. From Figure 5.2, Figure 5.4 and Figure 5.6 we can see that the Vargav1 auto-scaler has a high average number of Taskmanagers. Constantly having a high number of Taskmanagers results in a low average latency as there are always enough resources available to process the incoming load. When varying the utilization parameter between 30, 50, and 70% we see that with a higher utilization fewer resources are used and the average latency is slightly higher. This is is the same trend observed with the CPU HPA. This is not unexpected as the utilization metric used by Vargav1 is a subset of the CPU utilization metric.

Vargav1 is almost constantly overprovisioned due to its difficulty scaling down caused by lack of cooldown period.

The constant overprovisioning is caused by difficulty scaling down as can be seen in Figure 5.8. After scaling down some lag builds up, when the dataflow is back online there is a large amount of lag to be processed. These records are consumed at the maximum possible rate by Flink spiking the utilization triggering a scale-up event. So almost every time Vargav1 attempts to scale down it scales right back up again. This can be prevented by implementing a cooldown which we have done for Vargav2.

5.10. Vargav2

Adding a cooldown period to Vargav1 yielded a great reduction in the number of resources used while slightly increasing the average latency.

The performance of Vargav2 is much better than Vargav1 as can be seen in Figures 5.2, 5.4 and 5.6. The improved performance is not surprising as Vargav2 can scale down normally due to the implemented cool-down mechanic described in section 3.3.4. The normal scale-down behavior can be seen in Figure 5.9. Vargav2 does have a higher average latency but this is a result of using fewer resources than Vargav1. The trend identified for Vargav1 with regards to varying the utilization parameter is the same for Vargav2, a higher utilization uses fewer resources but at a cost of higher average latency.

5.11. Dhalion-adapted

Dhalion-adapted performed well for queries 1 and 3 due to its latency threshold metric.

Dhalion-adapted performed well compared to the other auto-scalers for queries 1 and 3, but not for query 11 as can be seen in Figures 5.1, 5.3 and 5.5 respectively. In Figure 5.10 we can see the telemetry data for query 1 Dhalion-adapted with a latency threshold of 5 seconds. Every time some latency builds up above the set threshold and the auto-scaler is not in cooldown more resources are added. Resources are only added when Flink cannot keep up with the input rate resulting in low resource utilization. At the same time when Flink is underprovisioned the latency in the Kafka queue increases and more resources are added until the latency is below the threshold value keeping the latency low throughout the experiments. The latency threshold for scaling up allows Dhalion-adapted to efficiently balance resource consumption and latency, resulting in a good performance for queries 1 and 3. For queries 1 and 3 varying the latency threshold value had little effect as can be seen in Figures 5.2, 5.4.

Dhalion-adapted had mixed results for query 11 due to a low cooldown period causing instability.

In Figure 5.11 we can see the telemetry data for query 11 Dhalion-adapted with a latency threshold of 1 second. At approximately 50 minutes Dhalion-adapted starts to scale up in rapid consecutive steps. Dhalion-adapted determines the scale-up factor by looking at the input rate and throughput value as described in chapter 3. However the throughput has not stabilized yet, Dhalion-adapted is using a lower value for throughput making the scale-up factor greater than 1, triggering a scaling event. This problem could likely be resolved by increasing the cooldown period letting the throughput stabilize before computing the scale-up factor. The variability in the results of Dhalion-adapted with different latency thresholds is due to the fact that for a threshold of 1 and 10 seconds the scale-up problem described above was observed. For the experiment with the latency threshold of 5 seconds it appears we got ”lucky” and the scale-up loop did not take place. With an increased cooldown we would expect the Dhalion-adapted auto-scalers with 1 and 10 seconds latency thresholds to be close to the one with a 5 second latency threshold as we observed in query 1 and query 3.

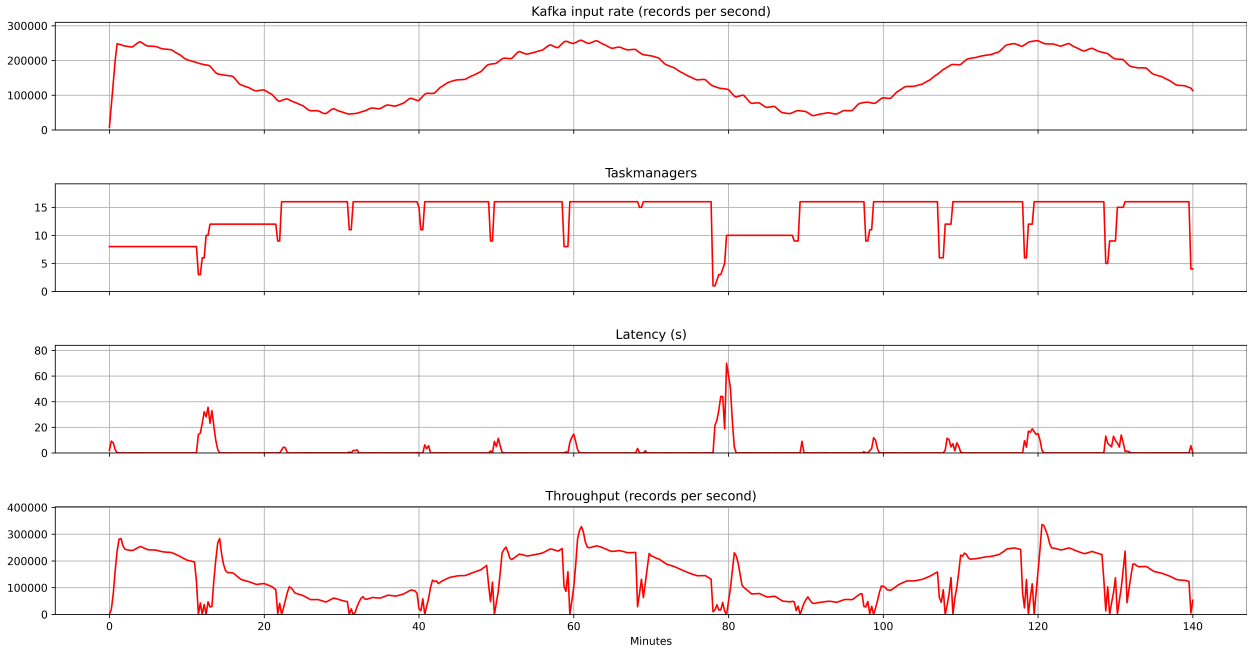

Figure 5.8: Telemetry data of Vargav1 with target utilization of 50% for query 1.

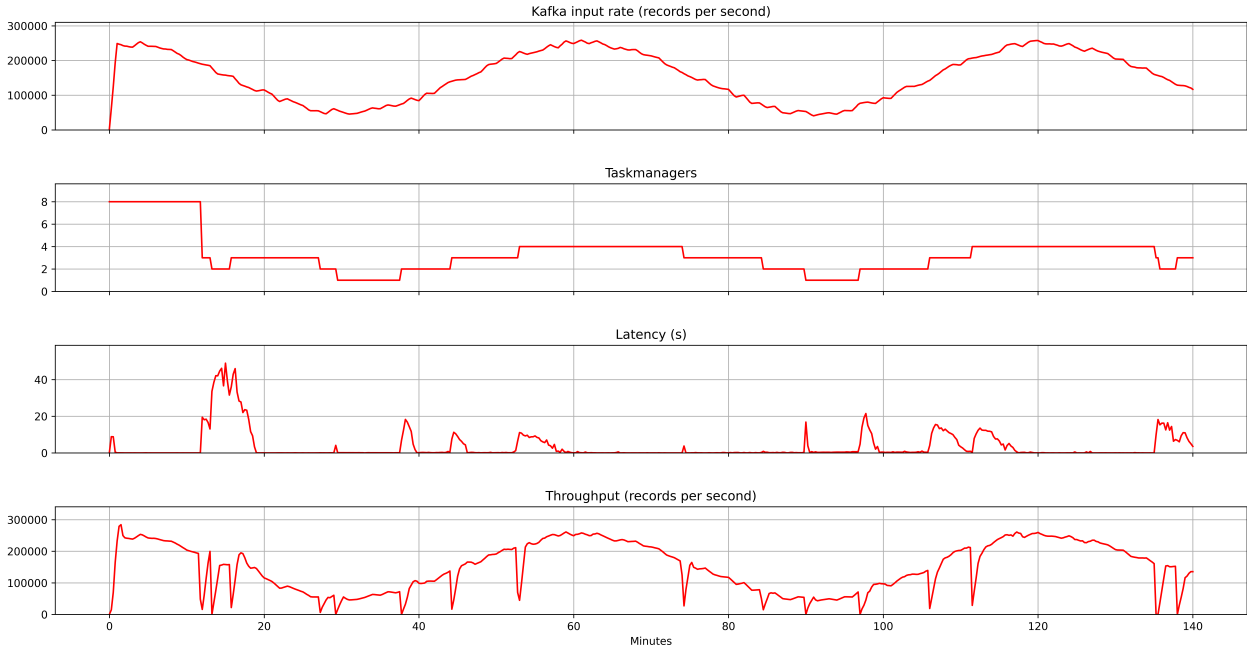

Figure 5.9: Telemetry data of Vargav2 with target utilization of 50% for query 1.

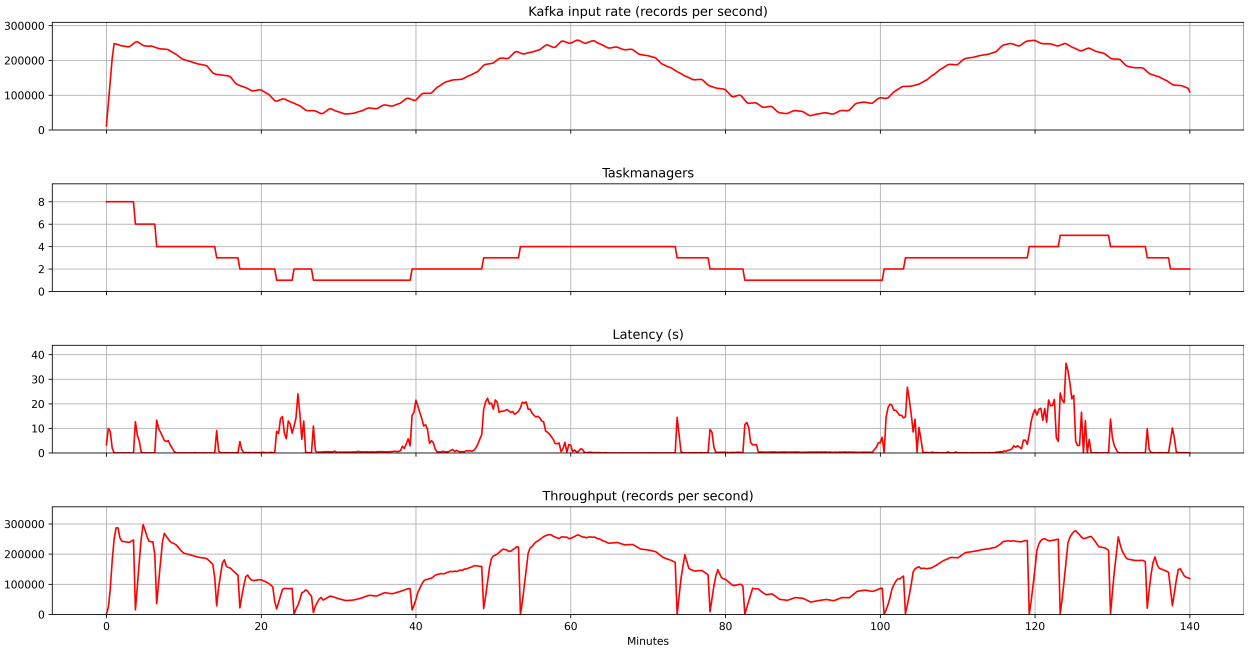

Figure 5.10: Telemetry data of Dhalion-adapted with a latency threshold of 5 seconds for query 1.

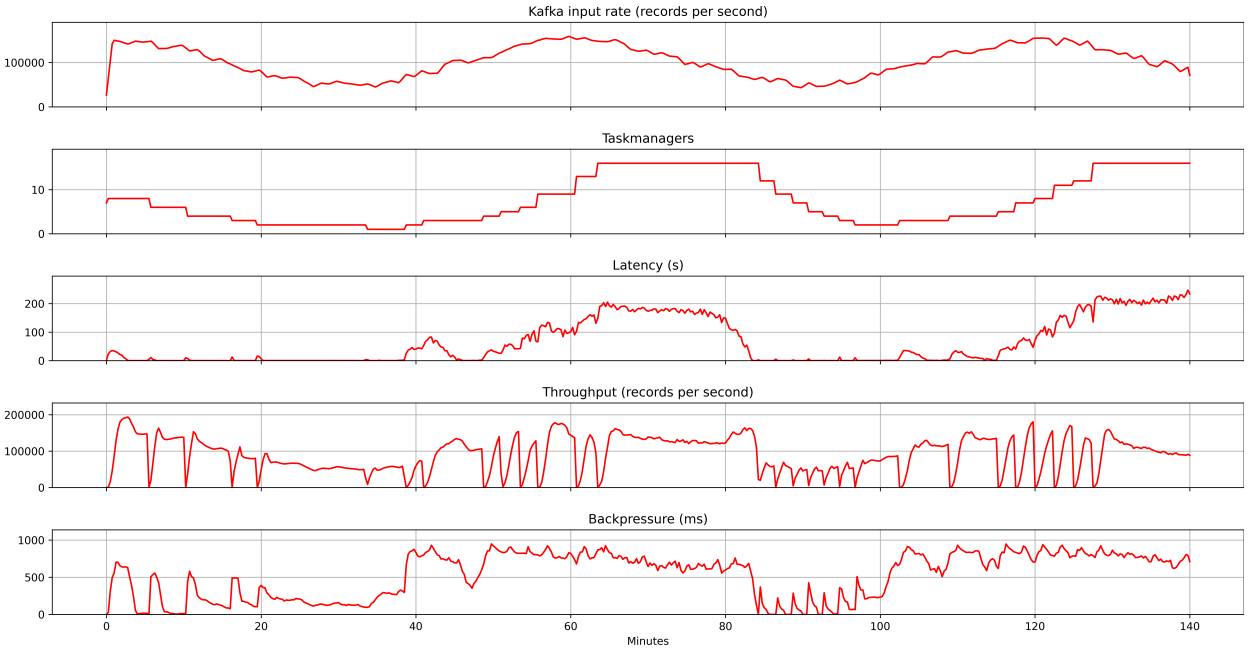

Figure 5.11: Telemetry data of Dhalion-adapted with a latency threshold of 1 second for query 11.

5.12. DS2-modern

DS2-modern performed poorly for queries 1 and 3 due to overestimating the true processing rates of operators.
For DS2-modern the results are mixed across the experiments. In Table 5.1 we see the results of all DS2-modern experiments for query 1. The DS2-modern with 0% over-provisioning performed so poorly that the average latency was over 100 seconds. The reason DS2-modern performed so poorly was because it was constantly under-provisioned. The under-provisioning was due to the fact that the true processing rates of the operators was overestimated. The true processing rates of the operators was estimated using the output rate of the upstream operators divided by the busy time per operator. The busy time per operator was often lower than expected for query 1. At maximum throughput, the busytime per operator would often be around 800 ms while it would be expected to be at 1000 ms. This low busy time results in a higher than expected true processing rate per operator and thus a lower maximum parallelism. This under-provisioning can be mitigated with the over-provisioning factor. We see that when over-provisioning by 66% and 33% the results of DS2-modern are much better. For query 3 we observe the same as can be seen in Table 5.2, DS2-modern under-provisions causing high average latency. For query 11 the model does not appear to under-provision as much this is caused by the higher computational load of query 11 increasing the $busyTimePerSecond$ metric resulting in a lower estimated true processing rate and thus a higher maximum parallelism.

DS2-moderns' performance model does not yield the appropriate parallelism required by Flink due to the input rate used to determine the optimal parallelism differing from the actual input rate into Flink.
The DS2 performance model tries to find the optimal parallelism based on the input rate into the Kafka queue (producer rate). However, the input rate (consumer rate) into the stream processing system is not equal to the input rate into the Kafka queue. If lag builds up then the stream processing system should scale up to deal with this lag, however, the DS2 model is partially oblivious to the lag as it bases the optimal parallelism on the input rate to the Kafka queue. DS2 does measure the processing rates of the operators which would be higher than expected when lagged records are being processed. This difference between the rate at which DS2 thinks it should be processing records and the rate at which it is actually processing records causes the model the oscillate occasionally worsening performance. The input rate of DS2 cannot be set based on the aggregated output rate of the Flink Kafka connector because optimal parallelism of the Kafka connectors also needs to be determined. Furthermore, DS2-modern occasionally makes decisions that are not ideal such as scaling down when there is lag present, this phenomenon can be observed in Figure 5.12 where at around 50 minutes a scale down occurs while there is a large amount of lag present. This can be mitigated by only scaling down if the lag/latency is below a certain threshold as described in section 3.5.4.

5.13. DS2-modern-adapted

DS2-modern-adapted had fewer scaling operations compared to DS2-modern resulting in increased performance for most experiments.
DS2-modern-adapted implemented some safeguards into DS2-modern stopping it from scaling down if the latency is above a certain threshold and triggering model evaluation if the latency is above a certain threshold. In Table 5.1 we can see that the DS2-modern-adapted outperformed its DS2-modern counterpart with the same over-provisioning factor. This same fact holds for query 3 as can be seen in Table 5.2. For query 11 the results are a bit mixed between DS2-modern and DS2-modern-adapted, DS2-modern with an over-provisioning factor of 66% appears to outperform DS2-modern with an over-provisioning factor of 66% as can be seen in table 5.3. The reason for this is that DS2-moderns performance model worked exceptionally well for query 11 versus query 1 and query 3. So DS2-modern made few decisions that led to excess latency. However, the difference in the number of scaling operations between DS2-modern with an over-provisioning factor of 66% is large. In a setup where there is more overhead for rescaling DS2-modern-adapted would likely come out on top as it is more conservative with its scale-up operations.

For query 11 DS2-modern-adapted with 0% over-provisioning unperformed by a large amount when compared to DS2-modern with 0% over-provisioning due to a scale-up loop.
For query 11 DS2-modern-adapted with a scale-up factor of 0% performed very poorly as can be seen in Table 5.3, the root cause was a too small scale-up step in combination with a short cooldown period. The small scale up resulted in lag not being worked away, the DS2-modern-adapted model then again wanted to scale up creating more lag. This could have been prevented with a longer cooldown time just like for Dhalion-adapted or by increasing the over-provisioning factor. The other DS2-modern-adapted auto-scalers did not suffer from this problem because the over-provisioning factor caused them to make larger scale up steps.

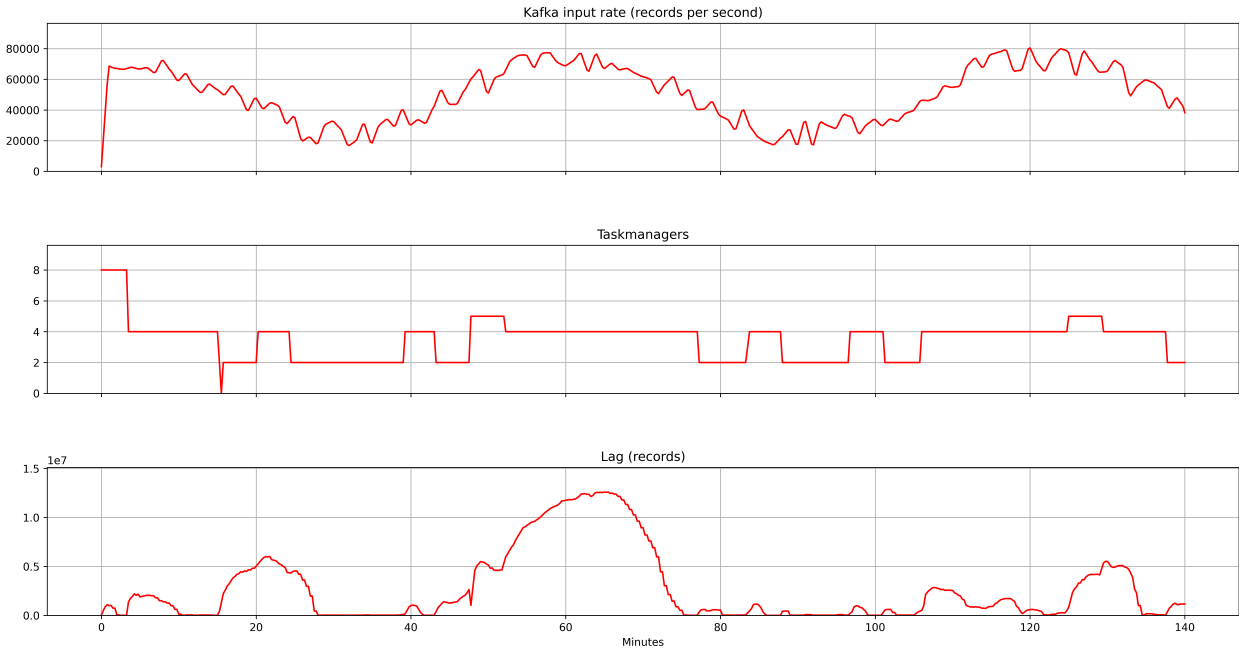

Figure 5.12: Telemetry data of DS2-modern with over-provisioning factor of 66% for query 1

5.14. Conclusion

In this chapter we discussed the results of our experiments. We showed visualizations of the average latency and the average number of Taskmanagers for each auto-scaler. Dove into the root causes of the results by looking at telemetry plots of individual auto-scalers. We found that adding a cooldown period to Vargav1 makes it perform better. The Dhalion-adapted auto-scaler performed well for query 1 and query 3 put poorly for query 11 due to a too short cooldown period. DS2-modern did not perform very well when compared to other auto-scalers due to the performance model not taking lag into account causing instability and under-provisioning which can be mitigated somewhat using an overprovisioning factor. DS2-modern-adapted fixed some of the problems of DS2 performing better across queries 1, 3, and 11.

6

Discussion & Future Work

6.1. Discussion

In this thesis, we implemented different auto-scalers for distributed stream processing. We set up experiments to compare these auto-scalers for three different queries. After running the experiments and analyzing the results we are now ready to reflect on the research questions posed in section 1.2.

(**RQ0**): **Which auto-scalers from the literature can be used for scaling distributed stream processing applications?**

From the literature, we identified four different auto-scalers for distributed stream processing DS2 [20], DRS [17], Vargav1 [32], and Dhalion [16]. DS2 [20] was the most highly regarded based on the number of citations and its improvements over Dhalion [16]. Prior to DS2, Dhalion was the state of the art based on its citations. Vargav1 [32] had only been published recently but is simple to implement due to its utilization of the Kubernetes HPA framework. We did not end up implementing DRS due to the underlying queueing model which is too rigid. The queueing model does not allow for complex dataflow topologies such as that of query 3 of the Nexmark benchmark [29]. No previous research has been done on using a Kubernetes CPU HPA to scale stream processing applications, however, we decided to include it in our experiments as the Kubernetes CPU HPA served as a simple baseline that requires little extra effort to implement. Throughout the auto-scaling literature, there are many proposed and implemented auto-scalers, however, they are rarely applicable to stream processing. Simple stateless applications with no scaling downtime can rely on coarse metrics such as CPU utilization and memory as there is no complicated behavior such as backpressure. Auto-scalers designed for simple stateless applications like web servers also have an easy time determining how many resources to add as the relationship between number of requests processed and servers is one to one. For stream processing applications there is overhead due to the interdependence between operators on different servers making the relationship between number of requests processed and the number of servers non-trivial.

(**RQ1**): **How can we compare these auto-scalers?**

To compare the different auto-scalers we needed to set up a test environment. The test environment consisted of a load-generating application that wrote records to a Kafka queue. The records on the Kafka queue would then be consumed by the distributed stream processing application Apache Flink. The infrastructure was monitored using Prometheus, the metrics provided by Prometheus could then be used by the auto-scalers to make scaling decisions. Throughout the experiments, the average number of resources used by Apache Flink was monitored, the average latency of the records in the Kafka queue, and the number of re-scaling operations. These metrics gave us insights into how well the auto-scaler performed. The auto-scalers all had different configurable parameters. To determine the effect that these parameters had on the auto-scalers, we altered one of the configurable parameters per auto-scaler. Furthermore, stream processing applications can run many different types of queries which all have different computational

complexities and stress resources in a different manner. To account for this difference in computational complexities we tested all auto-scaler on three different queries. Unfortunately, due to cost constraints, we were only able to run experiments once per auto-scaler. Given the non-deterministic characteristics of distributed systems, we cannot say with any statistical significance that one auto-scaler is better than another.

(**RQ2): How can these auto-scalers be improved?**

For Vargav1 the improvement was quite simple, implementing a cooldown period after a scaling operation. However, the Kubernetes HPA framework makes implementing this cooldown quite difficult so through a rather convoluted approach a cooldown can be implemented by setting the metrics to zero if a scaling event had occurred. Improving DS2-modern was also quite straightforward, incorporating knowledge of the amount of lag present. Dhalion-adapted, Vargav1, Vargav2, and DS2-modern-adapted all use metrics from the Kafka queue to make scaling decisions. The Kafka queue gives a clear indication of whether the streaming job can keep up with the load. If the lag in the Kafka queue is increasing the job is under-provisioned, if there is no lag then the job could be over-provisioned. Dhalion-adapted and DS2 both look at throughput rates to determine how much to scale. The throughput rate gives a much better indication of how much to scale, in the case of Dhalion-adapted by dividing the input rate by the throughput to obtain the scaling factor and in the case of DS2 by creating a performance model based on the throughput rates. Using throughput-based evaluation methods yielded better results than non throughput based methods like that of the CPU-based HPA. A problem we did not solve for DS2 was how to set the input rate of the performance model. The original implementation of DS2 [20] generated records from within the Flink source, there was no queue in front of the stream processing application. If Flink was not able to handle the input rate backpressure would occur and the source rate would be throttled. In a real-world system, a producer would never be throttled lag would just build up in the queue. When lag has built up in the queue the real input rate into the stream processing system no longer matches the input rate used by the performance model causing the model to yield sub-optimal results.

6.2. Limitations

One aspect we did not take into account in our experiments was the effect of scaling overhead. Varga et al. [32] showed that depending on the state size the re-scale operation can vary a lot. Query 1 and 3 had very little state stored (only the consumer offsets of the Kafka connectors), query 11 was the only query with state where the maximum state was about 600 MB. The re-scale operation for all three queries was therefore quite low in the seconds range. In production setups, the state stored by a query can be in the gigabyte range and scaling operations can take minutes. If there is more overhead then auto-scalers that have fewer scaling operations perform better. However, the auto-scalers were not tested under these conditions. In Tables 5.1, 5.2 and 5.3 we do show the number of re-scaling operations per Taskmanager which gives an indication of which auto-scalers would perform better in contexts with high scaling overhead.

We only explored one configurable parameter per auto-scaler while there are many more that could be configured differently. The state-space of the different auto-scalers, in particular, Dhalion-adapted with six configurable parameters is too large to test as computing power is not free. Furthermore, the auto-scaler parameters were not tuned in any way, they were simply best guesses and held constant across experiments. This led to some poor results for Dhalion-adapted for query 11 which would have performed better if the cooldown period was longer. We tested the different auto-scaler for three different queries but there are many more possible queries. The extended Nexmark benchmark[1] contains 22 queries. Each query is computationally different and produces different metric behaviors.

Given the non-deterministic nature of distributed systems running in a cloud environment, our results are not statistically significant as we only ran experiments once per auto-scaler. If experiments were run twice there is no doubt that the outcomes would not be identical. We have however not quantified how much this difference is due to only running the experiments once. However, for the Dhalion-adapted implementations with a different latency threshold, we see that for queries 1 and 3 in Figures 5.2 and 5.4 that they are clustered quite closely together, giving some indication of the vari-

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

ance between experiments. We were limited in how many experiments could be run due to the cost of running experiments in the cloud.

6.3. Next Generation Auto-scaler

Given the experimental results, the next generation auto-scaler should not be implemented using the Kubernetes HPA framework due to its lack of easy to configure cooldown and set evaluation method. Furthermore, the Kubernetes HPA uses the same metrics for detecting the need for a re-scaling operation and determining by how much to scale. We imagine an auto-scaler that would take into account the state of the queue in front of the stream processing application for detecting the need for re-scaling. If there is no lag then the auto-scaler could potentially scale down, if there is lag and it's increasing the auto-scaler should scale up. To determine how much to scale we imagine an auto-scaler that keeps track of its configuration and the maximum achieved throughput rate for that configuration. Instead of using a performance model like DS2 the application measures and stores the true processing rate of the application. Once the maximum throughput is known for a certain configuration the throughput per Taskmanager can be determined. Once the throughput per Taskmanager is determined scaling decisions can be made based on the input rate and an over-provisioning factor. The throughput should only be measured at its maximum which occurs when there is lag in the Kafka queue.

6.4. Future Work

When re-scaling the streaming job, the streaming job had to be stopped. Stopping the streaming job caused lag to build up which had to be processed after the streaming job was back online. Recent research such as CLONOS [27] has shown that it is possible to have an operator failure in a streaming job that can be recovered without stopping the streaming job by using standby operators. Extending this technology to allow streaming jobs to scale up by adding an operator while the job is still running would remove the scaling overhead for scaling up. This would reduce the latency as the streaming job can keep processing records when scaling up.

There is currently no benchmark or standard for testing auto-scaling applications in general and specifically for stream processing. This lack of benchmarks makes it difficult for researchers to compare and test the performance of the auto-scalers that they have implemented. If there was a benchmark for auto-scaling techniques researchers could spend their time iterating on their ideas instead of implementing their own testing environment which requires a substantial amount of engineering. The Nexmark[2] benchmark on Github already supports reading data from a Kafka queue. Perhaps with some slight modifications, it would be possible to run the benchmark with different load patterns on a Kubernetes cluster with monitoring already set up. This would allow researchers to use a common test environment to test their auto-scalers.

Throughout this thesis, we have utilized Flinks' Reactive Mode which scales each operator to the maximum possible parallelism. This differs from the approach taken by the original implementation of DS2 [20] where each operator was scaled independently. It would be interesting to see the difference in performance between these two approaches for different stream processing systems. If the performance difference is minor then an equivalent version of Flinks' Reactive Mode could be implemented for other stream processing applications. This simplifies the evaluation step greatly as now the per operator parallelism no longer has to be determined just the parallelism of the system as a whole.

[2]https://github.com/nexmark/nexmark

Bibliography

[1] Apache Kafka, . URL https://kafka.apache.org/.

[2] Apache Flink Reactive Mode. . URL https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/elastic_scaling/.

[3] Backpressure. URL https://flink.apache.org/2021/07/07/backpressure.html.

[4] Horizontal Pod Autoscaler. URL https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[5] Kubernetes. URL https://kubernetes.io/.

[6] Prometheus, . URL https://prometheus.io/.

[7] prometheus-adapter, . URL https://github.com/kubernetes-sigs/prometheus-adapter.

[8] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned. *IEEE International Conference on Cloud Computing, CLOUD*, 2018-July:970–973, 2018. ISSN 21596190. doi: 10.1109/CLOUD.2018.00148.

[9] Ahmed Ali-Eldin, Johan Tordsson, and Erik Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. *Proceedings of the 2012 IEEE Network Operations and Management Symposium, NOMS 2012*, (978):204–212, 2012. doi: 10.1109/NOMS.2012.6211900.

[10] Mohammad S. Aslanpour, Adel N. Toosi, Javid Taheri, and Raj Gaire. AutoScaleSim: A simulation toolkit for auto-scaling Web applications in clouds. *Simulation Modelling Practice and Theory*, 108(June 2020):102245, 2021. ISSN 1569190X. doi: 10.1016/j.simpat.2020.102245. URL https://doi.org/10.1016/j.simpat.2020.102245.

[11] Andre Bauer, Nikolas Herbst, Simon Spinner, Ahmed Ali-Eldin, and Samuel Kounev. Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):800–813, 2019. ISSN 15582183. doi: 10.1109/TPDS.2018.2870389.

[12] Paris Carbone, Asterios Katsifodimos, † Kth, Sics Sweden, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. 36, 2015.

[13] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. Elastic stateful stream processing in storm. *2016 International Conference on High Performance Computing and Simulation, HPCS 2016*, pages 583–590, 2016. doi: 10.1109/HPCSim.2016.7568388.

[14] Trieu C. Chieu, Ajay Mohindra, Alexei A. Karve, and Alla Segal. Dynamic scaling of web applications in a virtualized cloud computing environment. *Proceedings - IEEE International Conference on e-Business Engineering, ICEBE 2009; IEEE Int. Workshops - AiR 2009; SOAIC 2009; SOKMBI 2009; ASOC 2009*, pages 281–286, 2009. doi: 10.1109/ICEBE.2009.45.

[15] Hector Fernandez, Guillaume Pierre, and Thilo Kielmann. Autoscaling web applications in heterogeneous cloud infrastructures. *Proceedings - 2014 IEEE International Conference on Cloud Engineering, IC2E 2014*, (March 2014):195–204, 2014. doi: 10.1109/IC2E.2014.25.

[16] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: Self-regulating stream processing in Heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017. ISSN 21508097. doi: 10.14778/3137765.3137786.

[17] Tom Z J Fu, Jianbing Ding, Richard T B Ma, Senior Member, Marianne Winslett, Associate Member, Yin Yang, and Zhenjie Zhang. DRS : Auto-Scaling for Real-Time Stream Analytics. 25(6): 3338–3352, 2017.

[18] Nikolas Roman Herbst, Samuel Kounev, Andreas Weber, and Henning Groenda. BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*, pages 46–56, 2015. doi: 10.1109/SEAMS.2015.23.

[19] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, 2011. ISSN 0167739X. doi: 10.1016/j.future.2010.10.016. URL http://dx.doi.org/10.1016/j.future.2010.10.016.

[20] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, E T H Zurich, Matthew Forshaw, Timothy Roscoe, E T H Zurich, Implementation Osdi, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need : fast , accurate , automatic scaling decisions for distributed streaming dataflows This paper is included in the Proceedings of the. *Osdi*, 2018.

[21] Patrick Kurp. Green Computing. *Communications of the ACM*, 51(10):11–13, 2008. ISSN 15577317. doi: 10.1145/1400181.1400186.

[22] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing*, 12(4):559–592, 2014. ISSN 15707873. doi: 10.1007/s10723-014-9314-7.

[23] L. Magnoni. Modern messaging for distributed sytems. *Journal of Physics: Conference Series*, 608(1), 2015. ISSN 17426596. doi: 10.1088/1742-6596/608/1/012038.

[24] Michael Maurer, Ivan Breskovic, Vincent C. Emeakaroha, and Ivona Brandic. Revealing the MAPE loop for the autonomic management of cloud infrastructures. *Proceedings - IEEE Symposium on Computers and Communications*, pages 147–152, 2011. ISSN 15301346. doi: 10.1109/ISCC.2011.5984008.

[25] Dirk Merkel. Docker : Lightweight Linux Containers for Consistent Development and Deployment Docker : a Little Background Under the Hood. *Linux Journal*, 2014(239):2–7, 2014. URL http://delivery.acm.org.ezproxy.library.wisc.edu/10.1145/2610000/2600241/11600.html?ip=128.104.46.196&id=2600241&acc=ACTIVESERVICE&key=066E7B0AFE2DCD37.066E7B0AFE2DCD37.4D4702B0C3E38B35.4D4702B0C3E38B35&__acm__=1557803890_216b4a0168a6b29b8f2e7a74.

[26] Laura R. Moore, Kathryn Bean, and Tariq Ellahi. Transforming reactive auto-scaling into proactive auto-scaling. *Proceedings of the 3rd International Workshop on Cloud Data and Platforms, CloudDP 2013 - Co-located with ACM 13th EuroSys*, pages 7–12, 2013. doi: 10.1145/2460756.2460758.

[27] Pedro F. Silvestre, Marios Fragkoulis, Diomidis Spinellis, and Asterios Katsifodimos. Clonos: Consistent Causal Recovery for Highly-Available Streaming Dataflows. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1637–1650, 2021. ISSN 07308078. doi: 10.1145/3448016.3457320.

[28] Robert Stephens. Acta Informatica, Volume 34, Number 7 - SpringerLink. 541:491–541, 1997. URL http://www.springerlink.com/index/EYJMC2PKM2KC4T3U.pdf%5Cnpapers2://publication/uuid/72DBA7EC-764A-4509-8559-0794EC57A5E6.

[29] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. NEXMark - A Benchmark for Queries over Data Streams. Technical Draft. (October), 2003.

[30] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier Internet applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1), 2008. ISSN 15564665. doi: 10.1145/1342171.1342172.

[31] Giselle Van Dongen and Dirk Van Den Poel. Influencing Factors in the Scalability of Distributed Stream Processing Jobs. *IEEE Access*, 9:109413–109431, 2021. ISSN 21693536. doi: 10.1109/ACCESS.2021.3102645.

[32] Balázs Varga, Márton Balassi, and Attila Kiss. Towards autoscaling of Apache Flink jobs. *Acta Universitatis Sapientiae, Informatica*, 13(1):39–59, 2021. doi: 10.2478/ausi-2021-0003.