Technical University of Crete

School of Electrical & Computer Engineering

Pro-active Automatic scaling support for Apache Flink in Kubernetes in the Cloud

by

Alexandros Nikolaos Zafeirakopoulos

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DIPLOMA OF ELECTRICAL AND COMPUTER ENGINEERING

Chania, November 2022

“*Sic Parvis Magna - Greatness from Small Beginnings*”
Sir Francis Drake, English Explorer

Abstract

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. It executes arbitrary dataflow programs in a data-parallel and pipelined manner in event-driven applications such as, fraud detection (i.e. detection of suspicious transactions), anomaly detection (i.e. detection of rare or suspicions events), rule-based alerting (i.e. identification of data which satisfy one or more rules) and many more. Despite its versatility, Apache Flink cannot automatically and optimally adjust the utilization of its underlying computing resources when streaming sources produce data at varying speeds. In order to address this issue, we describe an autonomous agent to support dynamic autoscaling for Apache Flink on Kubernetes. This agent monitors, models and adjusts Flink's behaviour by optimally modifying its allocated resources in order to match the incoming workload while achieving minimum cost. The decision making process is based on operator idleness and changes to the input's record lag. We prove that our model not only successfully maintains the performance of the application while minimizing infrastructure costs, but can provide a better performance-to-cost ratio compared to already existing work on Flink autoscaling. The effectiveness of our model is supported by an exhaustive set of synthetic and real life workloads aimed to simulate a plethora of possible scenarios.

Contents

List of Figures

1 Introduction

1.1 Problem Definition

In the ever growing world of cloud-computing, dynamic resource management is imperative in order to achieve a healthy balance between cost and performance. The microservice architecture model [30] consists of several fine-grained services that are independently scalable and deployable. A scalable service should be able to handle dynamically varying workloads without noticeable degradation in system performance while simultaneously making sure that no significant resource under-utilization occurs.

Stream processing enables a variety of brand-new applications characterized by increased data generation and low latency response. Apache Flink is an open-source, distributed stream processing engine for stateful computations over unbounded and bounded data streams. Streaming jobs are typically long running and thus workload variations are expected to occur over an application's lifetime. Static provisioning of cloud resources, by setting the job's parallelism at launch-time, can prove inefficient and costly. If too few resources are allocated (under-provisioning), the application will not keep up with the increasing workload. If too many resources are allocated, so as to match the maximum workload, the system will run over-provisioned for a significant portion of the time and incur unnecessary infrastructure costs.

The parallelism of a Flink application cannot be modified during runtime [25]. It can only be changed by manually stopping the job, taking a savepoint of the current state of the stream and then restarting the job with a different parallelism from this snapshot.

When restarting a job, the state must be written either on a persistent storage or on memory (depending on its size) [41]. This action can be done asynchronously so as to not disturb the flow of the pipeline, therefore any time delay can be considered negligible. However, restoring from this savepoint can take a significant amount of time in which no incoming records are being processed until the pipeline is restored to its previous state. This means that when the application is successfully restarted it must try to catch up to any accumulated records that were not being processed during this downtime.

Decision making models which take into consideration the above limitation while simultaneously adapting to the incoming workload allow for efficient and dynamic scaling of Apache Flink applications with the goal of minimizing the expected downtime.

1.2 Scope of Thesis

In this thesis we present HYAS, a hybrid, threshold-based decision making agent which attempts to model the behaviour of a Flink cluster when under load and successfully react to various changes in the workload by provisioning or un-provisioning the required infrastructure resources efficiently.

There are two main types of autoscaling methods to consider when implementing elasticity in a cloud environment. Reactive scaling and proactive scaling. With reactive scaling, resources are scaled based on changes in the workload which are monitored in real-time. Alternatively in proactive scaling, machine learning and artificial intelligence techniques are used to analyze workloads and predict when more or fewer resources will be needed by training a behaviour model. HYAS can be considered a hybrid approach of these two methods since it relies on metrics from a real-time monitoring system in order to calculate the rate of change of unprocessed records of its input and predict whether a scale up or scale down decision must be made without the need of training a machine learning model.

We compare HYAS with Smilax's exploration mode [19], a reactive autoscaling agent for Apache Flink as well as with Kubernetes' own Horizontal Pod Autoscaler (HPA) [21], a general purpose scaling controller for container-based applications. From these comparisons, we can claim that HYAS provides better resource utilisation (cost reduction) and performance when deployed on a cloud environment from both Smilax and HPA. This can be attributed to HYAS's unique decision making process which monitors the rate of change of records not yet processed by Flink (decreasing or increasing) and allows for a more accurate and cost-efficient scaling action to be made.

Lastly, as stated on paragraph 1.1 Flink requires a minimum amount of downtime when rescaling to a new parallelism as the job must be first be stopped and then restarted. In this thesis, we have not found an adequate solution in order to tackle this issue and thus we have implemented a minimum amount of downtime (90 seconds) between scaling actions. That is, that no scaling decisions will be made for this period of time. The Flink community is aware of this limitation but no plans of presenting a solution are underway since this would require changing Flink from a stateful stream processing engine to a stateless one [36].

1.3 Thesis Structure

The remainder thesis structure is as follows:

- Chapter 2 provides some necessary background on scaling a microservice architecture as well as the related work on Flink Autoscaling.
- Chapter 3 has a more in depth look at the inner workings and configurations of Apache Flink. It also presents the Kubernetes environment, all the features of Kubernetes that we use in this work along with all the tools and services used for our infrastructure.
- Chapter 4 presents the architecture of our cluster, which was deployed in the Google Cloud Platform. This chapter also describes the implementation of our work in the form of the autoscaling agent and its behaviour.
- Chapter 5 presents the experiments implemented to test the efficiency of the proposed autoscaling model as well as all the results and comparisons between other autoscaling methods.
- Chapter 6 and 7 generalize the conclusions of this work and provide input on possible future work respectively.

2 Related Work

2.1 Background

Microservices

Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs [44]. Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.

A well defined aspect of microservices is autonomy and specialization. Each component service in a microservices architecture can be developed, deployed, operated, and scaled without affecting the functioning of other services while simultaneously, each service is designed for a set of capabilities and focuses on solving a specific problem.

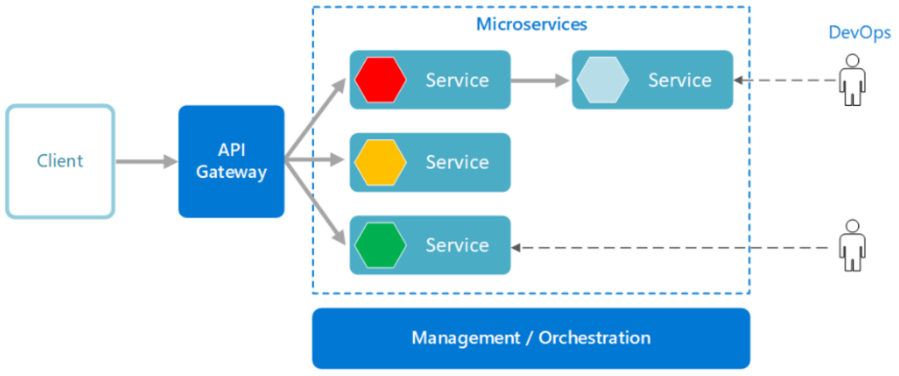


Figure 1: Microservice Architecture

One of the key benefits of adopting this architectural approach is scalability. Microservices allow each service to be independently scaled to meet demand for the application feature it supports. In cloud computing there are two main forms in which a service can be scaled, horizontal and vertical scaling.

Horizontal scaling, commonly referred to as scale-out, is the capability to automatically add services/instances in a distributed manner in order to handle an increase in load [22]. Conversely, with vertical scaling (scaling up or down), you can increase or decrease the capacity of existing services/instances by upgrading the memory (RAM), storage, or processing power (CPU) [42]. One common example of both of these kinds of scalability involves a hardware server. Suppose that network demand means a server has to handle significantly more data transfers. IT managers could add processing power or memory to the single server to increase its capability, or they could link it to other servers. The former approach illustrates vertical scaling while the latter illustrates horizontal scaling.

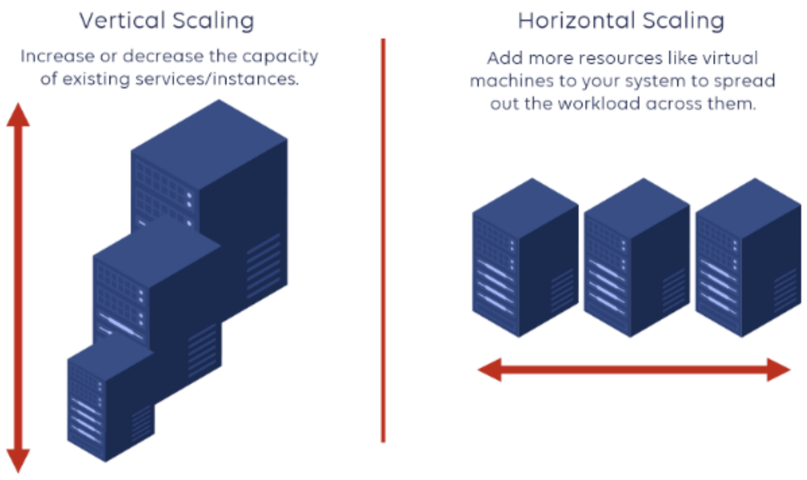

Vertical Scaling
Increase or decrease the capacity of existing services/instances.
Horizontal Scaling
Add more resources like virtual machines to your system to spread out the workload across them.

Figure 2: Vertical vs Horizontal scaling

When designing an autoscaler mechanism in a cloud environment the are two main techniques one must adopt in order to best suit its needs. These auto-scaling techniques are classified into two major groups [34]:

- Reactive techniques, where the scaling action is in reaction to a change in the system, and therefore does not anticipate such a change.
- Predictive or proactive techniques, which attempt to anticipate future changes in the system by performing the necessary scaling actions before such changes occur.

When defining a scaling action to be made by a controller, we essentially need to describe what form of resources must be scaled (e.g. processing power, memory, storage, read/write speeds), when should the action be taken, how many resources must adjusted (added or removed) and which scaling method must be used (horizontal or vertical). This scaling action will be made based on one or more inputs (SLA violations, workload changes, predictive models) and result in the system changing on a reactive or proactive manner.

Containers

Each microservice runs in a container [17], a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another.

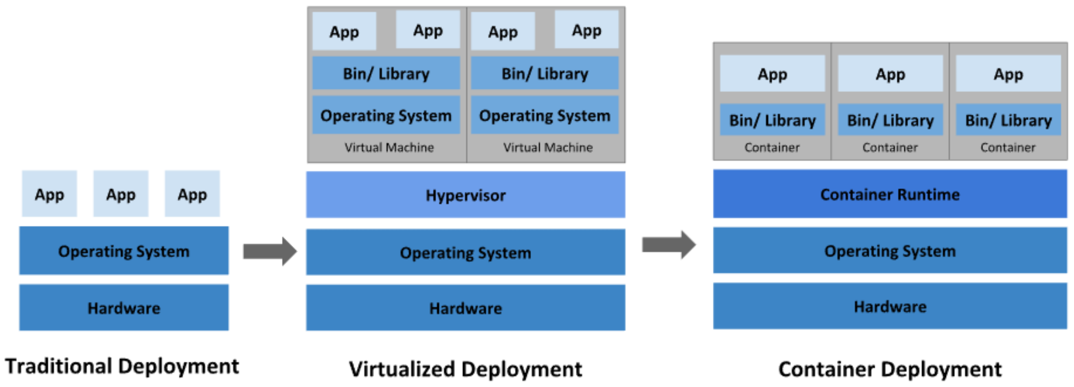


Figure 3: Traditional vs Virtualized vs Container deployment

In a traditional deployment, applications would run on physical servers. There was no way to allocate specific resources for each application in a physical server and thus one application would take up most of the resources, and as a result, the other applications would underperform.

To counteract this problem, virtualization was introduced. It allows multiple Virtual Machines (VMs) to run on a single physical server's CPU. Virtualization allows applications to be isolated between VMs and provides a level of security as the information of one application cannot be freely accessed by another application. Each VM is a full machine running all the components, including its own operating system, on top of the virtualized hardware.

A natural extension of this era are containers, which are similar to virtual machines, with the ability to share the Operating System (OS) between the applications. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

A few key benefits of packaging an application in a container are:

- Lightweight application deployment and creation.
- Environmental consistency across development, testing, and production. Runs the same on a laptop as it does in the cloud.
- Cloud and OS distribution portability
- Resource isolation
- Loosely coupled, distributed and elastic microservices. Applications are broken into smaller, independent pieces and can be deployed and managed dynamically

2.2 Related Work

The main idea of our hybrid decision making agent has been described initially in [41]. Varga et al. present an architecture that automatically scales the parallelism of Flink jobs by utilizing the Horizontal Pod Autoscaler (HPA) framework provided by Kubernetes and adjusting the available number of TaskManagers in the cluster thus, scaling the parallelism of the whole job accordingly. The scaling policy receives metrics from Kafka and Flink through Prometheus (record lag, throughput, idle time of operators) and makes a decision whether a scale up or down action must be taken.

Additionally, they analyze the downtime caused by the scaling operation and how it is affected by the size of the application state Flink has to store before restarting its job. They were able to establish a linear relationship between state size and the duration of savepointing to a persistent storage.

Contrary to our work, Varga et al. have not compared their autoscaling agent against other autoscaling systems. Additionally, our autoscaling agent is not implemented using HPA's framework but is running as separate python script independent of the Kubernetes cluster. While we have based our own scaling policy from the equations of [41], our goal was to establish if there was room to implement a more cost efficient and effective scaling agent specifically for Flink compared to Kubernetes' HPA and Smilax under a dynamic workload.

Smilax [19] is a statistical machine learning autoscaler agent for applications running on Apache Flink. Smilax agent acts proactively by predicting the forthcoming workload in order to adjust the allocation of workers to the actual needs of an application ahead of time.

During an online training phase (exploration mode), Smilax builds a model which maps the performance of the application to the minimum number of servers. During the work (optimal) phase, Smilax maintains the performance of the application within acceptable limits (i.e. defined in the form of SLAs) while minimizing the utilization of resources. Apache Flink and Smilax are deployed on Docker Swarm, a low-footprint virtualization platform based on Docker containerization.

In our work, we will be using Smilax as a baseline benchmark for our decision making agent in terms of both performance and cost comparisons.

The DS2 controller [25] is an automatic scaling controller for dynamic resource allocation of streaming analytic applications at the operator level. While DS2 was designed to be stream engine agnostic it was tested on Apache Flink, Apache Heron [10] and Timely Dataflow [39] on native machines running Debian GNU/Linux. It is able to calculate, in real-time, the optimal parallelism for each operator by detecting possible bottlenecks in the dataflow (i.e. which operator slows down the whole application) and thus it operates online and in a reactive manner.

In contrast to our work, which monitors and scales applications at job level (i.e. multiple operators or tasks may execute in a job), DS2 is designed to adjust the parallelism of each operator separately in order to maintain high throughput. This is a desirable and more fine-grained approach to autoscaling streaming applications which can be applied to HYAS if our scaling policy is changed to calculate the true (i.e. maximum) and observed processing rates of each operator as described on [25].

Ververica Autopilot [43] is a proprietary solution for autoscaling Apache Flink resources developed by Ververica and introduced in Ververica Platform 2.2. The autoscaler performs dynamic resource adjustments horizontally, in a reactive manner on Apache Flink applications by scaling the whole pipeline (i.e. changing the parallelism of the whole executed job).

Autopilot supports applications with multiple sources, while in our work, the supported applications contain only one source.

Lastly, Wybe J. C. Koper presents in his Masters thesis [26] a comparison between different autoscaling techniques, specifically for Apache Flink, based on average number of resources used, average latency and the number of scaling actions. Koper implements modified versions of DS2 [25], Varga et al. [41], Dhalion [1] as well as Kubernetes' Horizontal Pod Autoscaler (HPA). The evaluation of each algorithm is made using three different queries from the Nexmark benchmark suite [40] with a sinusoidal load pattern. Results from each comparison highlight the significance of a cooldown period between scaling actions, the importance of message queue specific metrics when scaling and that general purpose autoscaling controllers based on coarse metrics such as CPU utilization and memory are ill-equipped for stream applications.

In contrast to our work, we compare our autoscaling agent with only HPA and Smilax using a single application (query) running on Flink (instead of 3) but with 3 different workload variations (instead of 1). Additionally, we examine the cost discrepancies of each algorithm when deployed on Google's Kubernetes Engine (GKE).

The following autoscaling solutions where not designed specifically for Apache Flink or any stream processing engine but for general purpose web-applications.

Arabnejad et al. [13] compare two different autoscaling types of Reinforcement Learning (RL), which is SARSA and Q-learning. The autoscaler dynamically resizes Web applications in order to meet the quality of service requirements.

Bibal Benifa and D. Dejey [24] propose the RLPAS algorithm, which applies RL using a neural network in order to reduce the time for convergence to an optimal policy.

Rossi, Nardelli and Cardellini [37] propose RL solutions for controlling the horizontal and vertical elasticity of container-based applications in order to cope with varying workloads.

3 Infrastructure and Tools

3.1 Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams [6]. ‘Stateful’ means that applications can maintain an aggregation or summary of data that has been processed over time as a state.

3.1.1 Stream and Batch Processing

Any kind of data is produced as a stream of events. Credit card transactions, sensor measurements, machine logs, or user interactions on a website or mobile application, all of these data are generated as a stream [38]. Data can be categorized in unbounded and bounded streams.

As depicted on figure 4, bounded streams have a defined start and a finite end and thus each portion of the stream can first be consumed in its entirety before being processed. This is called **batch processing**, where the entire dataset is ingested to a batch processor (e.g. Flink) before producing any results. Conversely, unbounded streams have a defined start but no finite end as they are constantly producing data that must be continuously processed. This is called **stream processing**, where the input may never end, and so the data has to be processed at the moment it arrives.

In our work we will be handling inputs in the form of unbounded data streams which are generated constantly and must be promptly handled after they have been ingested.

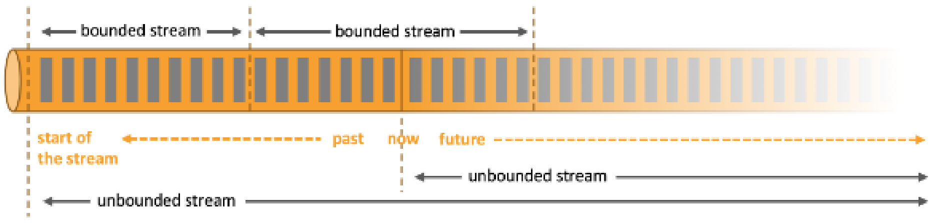


Figure 4: Bounded vs Unbounded data streams

3.1.2 Use Cases

Event-driven Applications

An event-driven application is a stateful application that is characterized by the need to trigger an action based on incoming events [9]. An event is a change in state, or an update, like an item being placed in a shopping cart on an e-shop website.

Figure 5 depicts the differences between the traditional application architecture and an event-driven architecture. In the traditional application design, applications read and store data to a remote transactional database which is independent of the application. The main difference between the traditional application design and an event-driven application, which is based on stateful stream processing, is that data and computation are co-located within the application (as depicted on figure 5) which in turn achieves local (in-memory or disk speed) data access. A persistent remote storage is only used in fault-tolerance which is achieved by periodically writing asynchronous checkpoints to said storage.

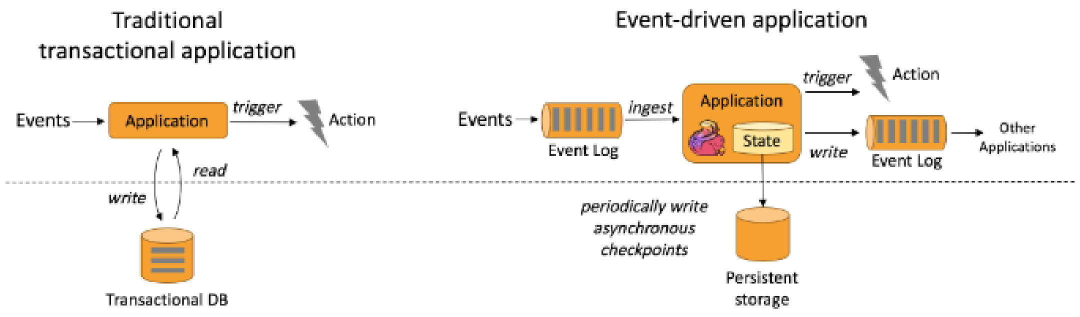


Figure 5: traditional vs event-driven application architecture

Typical real life examples of an event-driven applications are [9]:

- Fraud detection
- Anomaly detection
- Rule-based alerting
- Web application (social network)

Data Analytics Applications

Analytical jobs ingest raw data with the purpose of obtaining information, patterns and insights [9]. Figure 6 is helpful in visualizing some key differences between streaming and batch analytical applications. Traditionally, batch analytics are processed as a collection of periodic queries or applications on bounded data sets. Whenever new data changes need to be incorporated to the result of the analysis, new data must first be added to the existing dataset and the query needs to be resubmitted or any applications rerun. This is in contrast with streaming analytics which have the ability to ingest real-time event streams in the form of continuous queries as unbounded data sets. This allows for the continuous production of up-to-date analytical results as new events are consumed in a streaming manner.

The results can then be stored to a persistent storage (for both batch and stream analytics), produced in a user friendly format as reports (for batch analytics) or retained as an internal state to be queried by dashboard applications in a real-time fashion (for stream analytics).

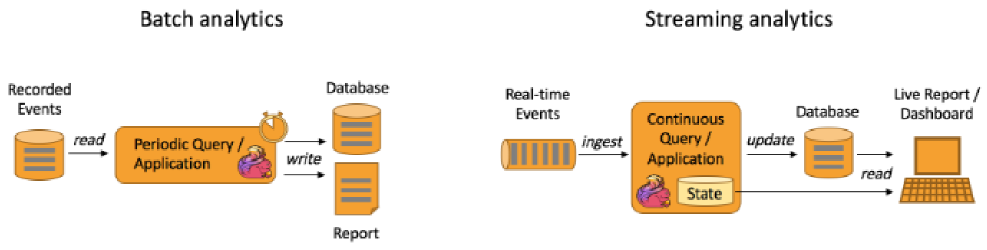


Figure 6: batch vs streaming analytics

Typical real life examples of data analytics applications are [9]:

- Quality monitoring of Telcommunication networks
- Analysis of product updates and experiment evaluation in mobile applications
- Ad-hoc analysis of live data in consumer technology
- Large-scale graph analysis

3.1.3 Flink Jobs

In Apache Flink, applications (or Jobs) are composed of streaming dataflows that can be transformed by user-defined operators which contain custom functions. These dataflows form directed graphs (i.e. data pipelines) that start with one or more sources, and end in one or more sinks [3].

Figure 7 presents an example of a streaming dataflow in the form of a data pipeline. In this particular example, records are entering the dataflow through a source operator (e.g. a Kafka consumer), they are transformed [18] consecutively by two operators with one or more functions and lastly they exit the dataflow through a sink.

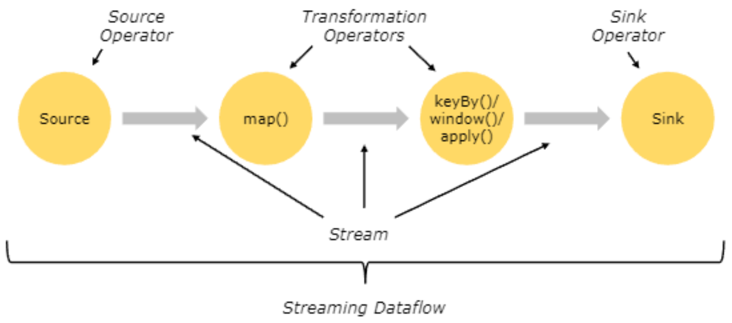


Figure 7: An example of a streaming dataflow

Figure 8 depicts an abstract form of the several connectors that can act as source and sink operators for Flink. An application can ingest real-time data from streaming sources such as message queues or distributed logs, like Apache Kafka [11] or Amazon Kinesis [2] in the form of source operators. Flink also supports the ability to consume bounded data-sets and historic data from transactional databases or any other file repository [3]. Similarly, the streams of results being produced by a Flink application can be sent to a wide variety of systems that can be connected as sinks (i.e other applications, databases, streams).

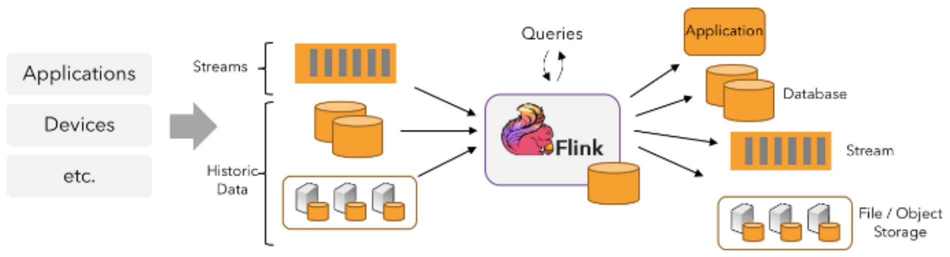


Figure 8: An abstract depiction showing the variety of systems that can connect to Flink either as a source or sink operators

Applications running in Flink are inherently parallel and distributed [3]. Figure 9 presents a parallel view of a dataflow with all operators having a parallelism of 2 except for the sink operator which has 1.

A stream has one or more stream partitions (depicted as edges on fig. 9) and each operator has one or more operator subtasks (depicted as nodes on fig. 9). The operator subtasks are independent and execute in different threads or even different machines. The number of operator subtasks is the parallelism of the whole operator. Different operators of the same program can have different levels of parallelism (as seen with with the sink operator on fig. 9).

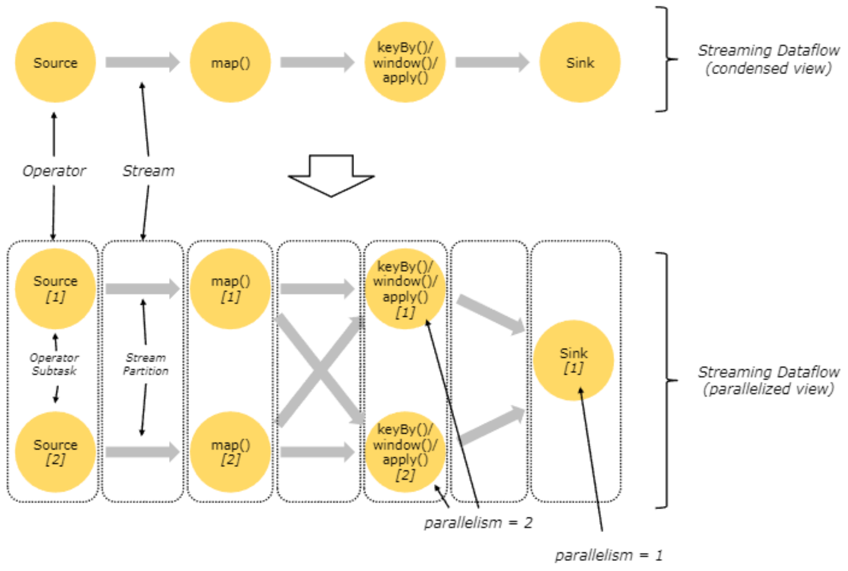

Source
map()
keyBy()/ window()/ apply()
Sink
Streaming Dataflow (condensed view)
Operator
Stream
Source [1]
Operator Subtask
Source [2]
Stream Partition
map() [1]
map() [2]
keyBy()/ window()/ apply() [1]
keyBy()/ window()/ apply() [2]
Sink [1]
Streaming Dataflow (parallelized view)
parallelism = 2
parallelism = 1

Figure 9: How a streaming dataflow can be depicted in parallel view

3.1.4 Stateful Stream Processing

Operations in Apache Flink can be stateful. In simple terms this can mean that past events can influence the way a current event is being processed.

Some examples of stateful operations: [8]

- When an application searches for certain event patterns, the state will store the sequence of events encountered so far.
- When aggregating events per minute, the state holds the aggregates.
- When training a machine learning model over a stream of data points, the state holds the current version of the model parameters.
- When historic data needs to be managed, the state allows efficient access to events that occurred in the past.

In Flink, the state is considered a snapshot of an individual operator at any given time which holds information about the application's execution till a particular point in time. Multiple snapshots of all the operators in the dataflow are able to capture the entire state of the pipeline and store it in memory or at a persistent storage. This way Flink can keep track of how many and which records have been processed up to a certain point.

There are two main types of snapshots:

- Checkpoint [5]: a snapshot taken **automatically** by Flink for the purpose of being able to recover from faults.
- Savepoint [7]: a snapshot triggered **manually** by a user (or an API call).

When a failure occurs or a manual restart of the job graph is triggered (i.e. when a rescaling action is made) the state is accessed either remotely or locally and is then redistributed across all parallel instances allowing processing to resume from the point in time which the snapshots have been taken.

3.1.5 Flink Cluster Architecture

The Flink runtime consists of two types of processes: a JobManager and one or more TaskManagers [4]. Figure 10 depicts a typical Flink Cluster including the client.

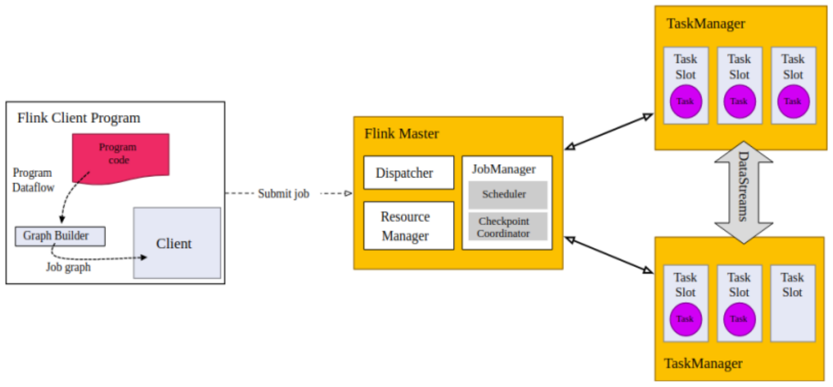


Figure 10: A typical Flink Cluster with one JobManager and two TaskManagers

The Client is not part of the runtime and program execution, but is used to prepare and send a dataflow to the JobManager.

JobManager

The JobManager has a number of responsibilities related to coordinating the distributed execution of Flink Applications. It decides when to schedule the next task (or set of tasks), reacts to finished tasks or execution failures, coordinates checkpoints and coordinates recovery on failures [4].

The JobManager consists of three different components:

- The ResourceManager, which is responsible for resource de-/allocation and provisioning in a Flink cluster (task slots).
- The Dispatcher, which provides a REST interface to submit Flink applications for execution and starts a new JobMaster for each submitted job.
- A JobMaster, which is responsible for managing the execution of a single JobGraph.

TaskManager [4]

The TaskManagers execute the tasks of a dataflow and exchange the data streams. The smallest unit of resource scheduling in a TaskManager is a task slot. The number of task slots in a TaskManager indicates the number of concurrent processing tasks (operators). Flink requires that at least one TaskManager must be present at all times in a cluster.

Each worker (TaskManager) is a JVM (Java Virtual Machine) process, and may execute one or more subtasks in separate threads. Each task slot represents a fixed subset of resources of the TaskManager. A TaskManager with three slots, for example, will dedicate 1/3 of its managed memory and CPU to each slot.

In our work, each TaskManager will be configured with only one task slot containing a part of the executed parallel dataflow as depicted on figure 11. This will mean that each time we add or remove a TaskManager the whole pipeline will be scaled accordingly as more operator subtasks are created or destroyed.

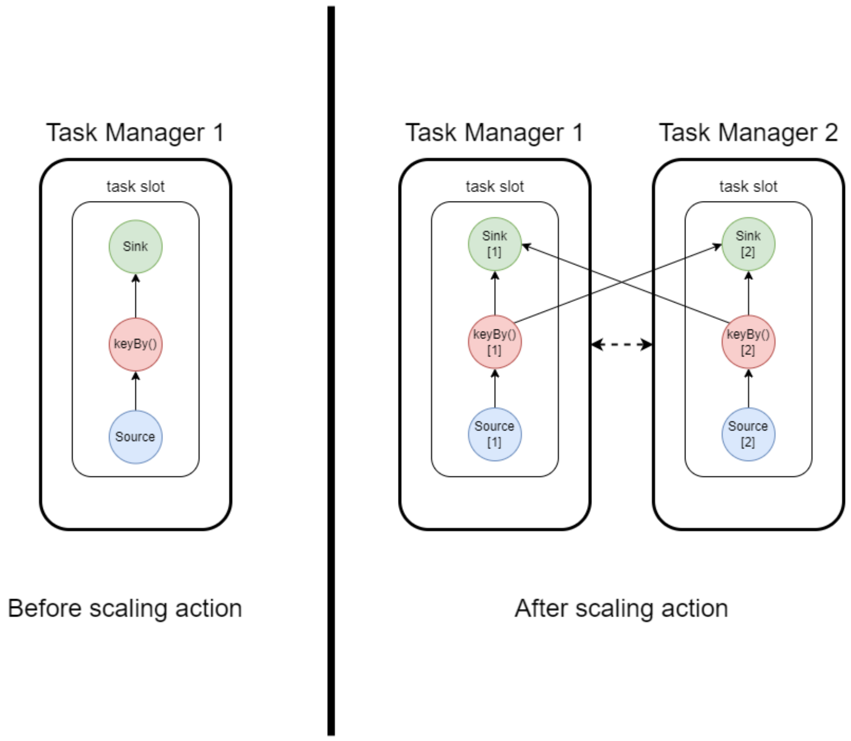


Figure 11: Abstract depiction of scaling the whole pipeline from 1 to 2 TaskManagers which will result in a parallelism of 2 for each operator

3.1.6 Cluster Modes

The jobs of a Flink Application can either be submitted to a long-running Flink Session Cluster, a dedicated Flink Job Cluster, or a Flink Application Cluster [4]. The difference between these options is mainly related to the cluster's life cycle and to resource isolation guarantees.

We will analyze only the differences between a Flink Session Cluster and a Flink Application Cluster since a Flink Job Cluster is deprecated.

Flink Application Cluster

- Cluster Lifecycle: a Flink Application Cluster is a dedicated Flink cluster that only executes jobs from one Flink Application and where the main() method runs on the cluster rather than the client. There is no need to start a Flink cluster first and then submit a job to the existing cluster session instead, the application logic and dependencies are packaged into an executable job JAR and the cluster entry-point is responsible for calling the main() method to extract the JobGraph. This allows a Flink Application to be deployed like any other application on Kubernetes, for example. The lifetime of a Flink Application Cluster is therefore bound to the lifetime of the Flink Application.

- Resource Isolation: in a Flink Application Cluster, the ResourceManager and Dispatcher are scoped to a single Flink Application, which provides a better separation of resources than the Flink Session Cluster.

Flink Session Cluster

- Cluster Lifecycle: in a Flink Session Cluster, the client connects to a pre-existing, long-running cluster that can accept multiple job submissions. Even after all jobs are finished, the cluster (and the JobManager) will keep running until the session is manually stopped. The lifetime of a Flink Session Cluster is therefore not bound to the lifetime of any Flink Job.

- Resource Isolation: TaskManager slots are allocated by the ResourceManager on job submission and released once the job is finished. Because all jobs are sharing the same cluster, there is some competition for cluster resources — like network bandwidth in the submit-job phase. One limitation of this shared setup is that if one TaskManager crashes, then all jobs that have tasks running on this TaskManager will fail.

In most cases, users will deploy Flink as an Application Cluster as it provides by far the easiest configuration needed since all the application logic is packaged into a executable job JAR. However, if multiple jobs need to run on a single cluster a Flink Session Cluster is much better suited due to the independence provided between the cluster lifecycle and any Flink job.

In our work, we have chosen to deploy Flink as an Application Cluster primarily because we wanted to utilize Flink's Reactive Mode [35] (further analyzed on section 5.1.1) which is available only on this cluster mode for the time being.

3.2 Kubernetes

3.2.1 Overview

Kubernetes, is an open source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications. It provides a framework with which to run distributed systems efficiently.

Kubernetes' main benefits include [28]:

- Service discovery and load balancing: Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- Storage orchestration: Kubernetes allows for automatic mounting of various storage systems such as local storages, public cloud providers, and more.
- Automated rollouts and rollbacks: By describing the desired state of a deployed container using Kubernetes, this state can be maintained at a controlled rate. For example, Kubernetes can automate the creation of new containers for a deployment, remove existing containers and adopt all their resources to the new container.
- Automatic bin packing: A user-defined cluster of nodes can be defined to run containerized tasks with a set amount of CPU and memory for each container. Kubernetes can then fit containers onto those nodes accordingly in order to make the best use of available resources automatically.
- Self-healing: Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to user-defined health checks and doesn't advertise them to clients until they are ready to serve.
- Secret and configuration management: Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. Secrets and application configurations can be deployed and updated without the need to rebuild container images.

3.2.2 Infrastructure and Components

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications in the form of Pods. Every cluster has at least one worker node. The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster.

Figure 12 presents the architecture and the components of a Kubernetes cluster.

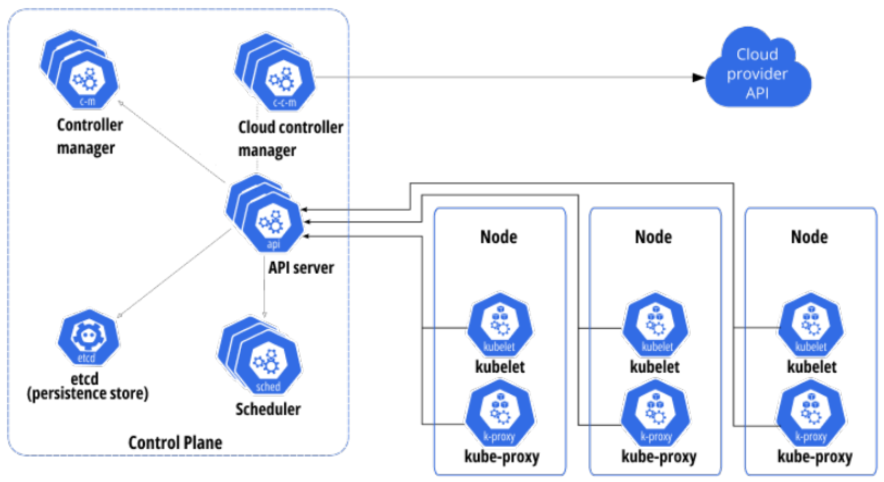


Figure 12: An anatomy of a Kubernetes cluster and its components.

The control plane's components make global decisions about the cluster [27] (i.e. scheduling as well as detecting and responding to cluster events).

- The API server is a component that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.
- Etcd is a consistent key-value storage used by Kubernetes to store all cluster data.
- The scheduler is a component that watches for newly created Pods with no assigned node, and selects a node for them to run on.
- The kube-controller-manager runs all controller processes, and the cloud-controller-manager is responsible for linking the cluster to the cloud provider's API and

exposes the components that interact with the cloud platform if the cluster is not running locally.

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment [27].

- kubelet is an agent that runs on each node in the cluster that makes sure that containers are running in a Pod and manages the containers that Kubernetes creates.
- The kube-proxy component is responsible for maintaining network rules on the Node, which allow the Pods to communicate with other services inside and outside of the cluster.

3.2.3 Services and Network Traffic

Kubernetes uses IP addresses to enable communication between services and components. A Pod is assigned an IP address upon creation, but this IP address is temporary and changes every time the Pod restarts (due to a crash or update). For this reason, Kubernetes introduced a resource called Kubernetes Services [29].

Kubernetes Services are abstractions that allow the Pod to use the network to communicate (either for internal cluster communication or external network communication). Each Pod bounds with its respective Kubernetes Service, and the Service is responsible for forwarding any traffic to the Pod. The Service discovers the Pod's IP address upon creation or change and exposes a permanent address (user-defined in the Kubernetes Service YAML configuration) and a port so that other services can communicate. A Kubernetes Service can be a ClusterIP, NodePort, LoadBalancer or External Name type.

The LoadBalancer type exposes the Service to the external network, using the cloud provider's load balancer routes. The NodePort type exposes the Service on each Node's IP at a predefined port. Allowing this port through the cluster's firewall makes the Service accessible through the external network. The ClusterIP service is the most used one and exposes the Service on a cluster-internal IP, so it's accessible to the rest of the cluster's services.

3.3 Apache Kafka

Apache Kafka is an open-source distributed event streaming platform used by thousands of companies for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications [11].

Event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time.

Some key use case examples for event streaming are [11]:

- To process payments and financial transactions in real-time, such as in stock exchanges, banks, and insurances.
- To track and monitor cars, trucks, fleets, and shipments in real-time, such as in logistics and the automotive industry.
- To continuously capture and analyze sensor data from IoT devices or other equipment, such as in factories and wind parks.
- To serve as the foundation for event-driven architectures and microservices.

Kafka combines three key capabilities for event streaming [11]:

1. To publish (write) and subscribe to (read) streams of events, including continuous import/export of your data from other systems.
2. To store streams of events durably and reliably for as long as you want.
3. To process streams of events as they occur or retrospectively.

Kafka is a distributed system consisting of servers and clients that communicate via a high-performance TCP network protocol. Kafka is run as a cluster of one or more servers that can span multiple datacenters or cloud regions. Some of these servers form the storage layer, called the brokers. Clients are considered distributed applications and microservices that read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner even in the case of network problems or machine failures.

Events

Kafka reads and writes data in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers.

- Event key: ”Alice”
- Event value: ”Made a payment of $200 to Bob”
- Event timestamp: ”Jun. 25, 2020 at 2:06 p.m.”

Producers are those client applications that publish (write) events to Kafka, and consumers are those that subscribe to (read and process) these events. In Kafka, producers and consumers are fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability. For example, producers never need to wait for consumers.

Events are organized and durably stored in topics. A topic is similar to a folder in a filesystem, and the events are the files in that folder. Topics in Kafka are always multi-producer and multi-subscriber (i.e a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events). Events in a topic can be read as often as needed, and unlike traditional messaging systems, events are not deleted after consumption. Instead, if event deletion is necessary, a custom duration for how long Kafka should retain said events can be defined through a per-topic configuration setting.

Partitions

Topics are partitioned, meaning a topic is spread over a number of ”buckets” located on different Kafka brokers. This distributed placement of data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.

Figure 14 depicts a topic which has four partitions P1 to P4. Two different producer clients are publishing, independently from each other, new events to the topic by writing events over the network to the topic's partitions. Events with the same key (denoted by their color in the figure) are written to the same partition. Note that both producers can write to the same partition if appropriate.

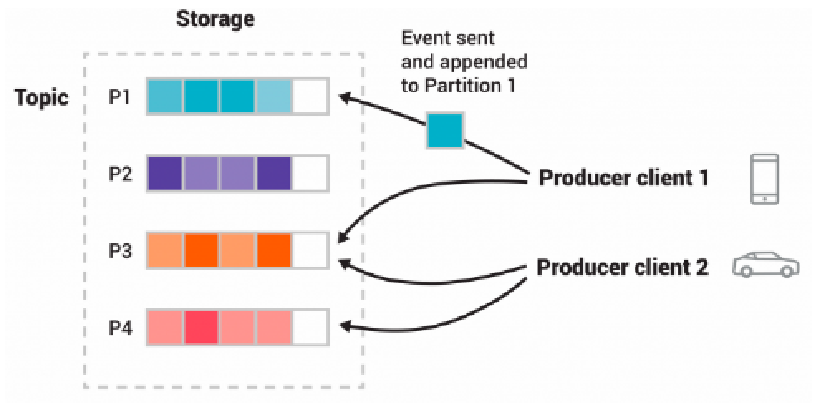


Figure 13: Anatomy of a Kakfa topic.

3.3.1 Apache Zookeeper

ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization and configuration maintenance [12].

ZooKeeper allows distributed processes to coordinate with each other through a shared hierarchical namespace which is organized similarly to a standard file system. The namespace consists of data registers, called znodes, and these are similar to files and directories. Unlike a typical file system, which is designed for storage, ZooKeeper data is kept in-memory, which means ZooKeeper can achieve high throughput and low latency numbers.

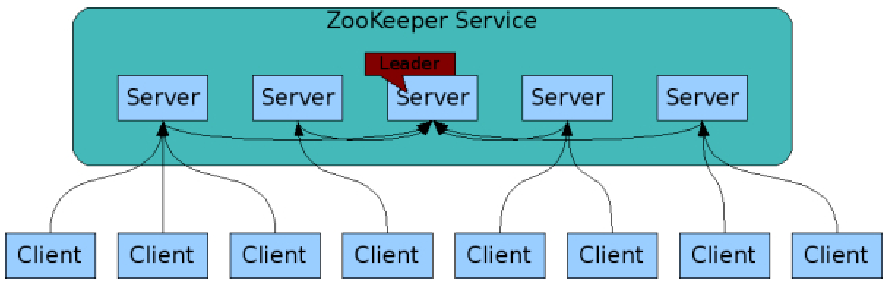


Figure 14: Zookeeper's ensemble of servers and its clients connections.

Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a set of hosts called an ensemble. The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.

Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

3.4 Prometheus

Prometheus is an open-source systems monitoring and alerting toolkit. Prometheus collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels [33].

Metrics, in their most simple term, are numeric measurements. Time series means that changes are recorded over time. What users want to measure differs from application to application. For a web server it might be request times, for a database it might be number of active connections or number of active queries etc.

Prometheus fundamentally stores all data as time series, meaning streams of timestamped values belonging to the same metric and the same set of labeled dimensions. Besides stored time series, Prometheus may generate temporary derived time series as the result of queries [31].

Every time series is uniquely identified by its metric name and optional key-value pairs called labels. The metric name specifies the general feature of a system that is measured (e.g. http_requests_total - the total number of HTTP requests received). Labels enable Prometheus' dimensional data model. Any given combination of labels for the same metric name identifies a particular dimensional instantiation of that metric (for example: all HTTP requests that used the method POST to the /api/tracks handler). The query language (PromQL) allows filtering and aggregation based on these dimensions. Changing any label value, including adding or removing a label, will create a new time series.

The Prometheus client libraries offer four core metric types [32].

- Counter: A counter is a cumulative metric that represents a single monotonically increasing counter whose value can only increase or be reset to zero on restart. For example, a counter can be used to represent the number of requests served, tasks completed, or errors.
- Gauge: A gauge is a metric that represents a single numerical value that can arbitrarily go up and down. Gauges are typically used for measured values like temperatures or current memory usage, but also "counts" that can go up and down, like the number of concurrent requests.
- Histogram: A histogram samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.
- Summary: Similar to a histogram while also providing a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

4 Autoscaler Model for Apache Flink

4.1 System Architecture in Kubernetes

In this thesis, we have deployed our Apache Flink cluster in Kubernetes both locally and in a cloud environment. Figure 15 depicts an abstract architecture of our infrastructure with one Kubernetes cluster and two worker nodes running in the cloud.

It should be noted that each Pod runs a containerized application each with its own respective Kubernetes service (as described on section 3.2.3) responsible for all internal node communications as well as forwarding any external requests (i.e. any metrics scraped by Prometheus will be sent to our autoscaler) outside of the cluster as required.

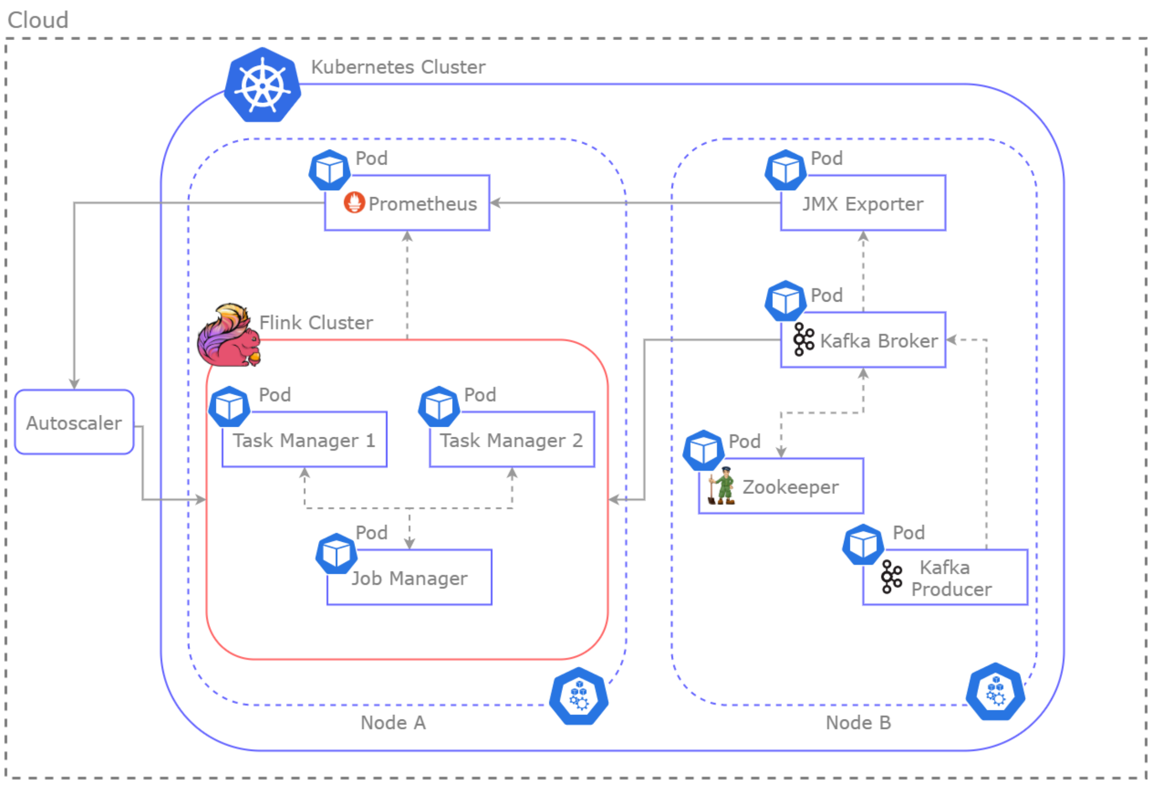


Figure 15: Abstract system architecture of our infrastructure on Kubernets in the cloud depicting a Flink cluster with 2 TaskManagers.

In node A we have deployed our Apache Flink Cluster with one JobManager and at least one TaskManager (up to a maximum of 6) as containerized applications each running in one Pod. The number of allocated TaskManagers varies over time and is adjusted by our Autoscaler running outside the Kubernetes cluster (or even outside of the cloud environment) as a separate python script. The Autoscaler, receives metrics from both Kafka and Flink from Prometheus (also present on node A) which in turn are used as inputs for the decision making process of our autoscaling agent. Any scaling actions are then forwarded in Kubernetes' command line tool (kubectl) which communicates with the cluster's control plane and modifies the number of TaskManagers (pods) available for Flink accordingly.

Node B contains all the necessary deployments for our Kafka cluster. The Kafka producer houses a custom load testing application (described on section 5.1.1) necessary for stressing our Flink job at various intervals. This application does not necessarily need to be present inside the Kubernetes cluster and can also receive input from outside sources such as news streams, twitter etc. Records are produced at varying quantities (from the Kafka producer) which are then in turn written to a specific Kafka topic in our broker. The Kafka broker is used as publish/subscribe system and acts as Flink's input stream. Each TaskManager depicts a Kafka consumer which has subscribed to a specific topic in the Kafka broker and can read from one or more partitions present there. Lastly, Zookeeper is a dependency of Apache Kafka since it has the responsibility of managing the Kafka cluster. It has a list of all the Kafka brokers and thus is able to notify Kafka if any broker or partition goes down and if any new brokers or partitions are up.

In order to monitor Kafka, a JMX agent is used. The direct connection between Kafka and Prometheus was not possible, so we use a JMX agent to retrieve any necessary metrics from the Kafka broker such as the volume of records written to each topic which when aggregated depicts the workload created by our Kafka producer. Java Management Extensions (JMX) is a technology which enables managing and monitoring applications, system objects, devices, and service-oriented networks [23]. The agent retrieves metrics from Kafka and makes them available by exposing a HTTP server (housed inside the JMX exporter). Prometheus retrieves the Kafka metrics by querying the HTTP endpoint of JMX exporter. In this way, we are able to monitor the workload of Kafka topics and thus the workload of the running job.

Figure 16 presents our system architecture after a scaling action with 4 TaskManagers (Pods) all present within a single node. Notice how only the pods containing TaskManagers have been scaled (compared to figure 15) and no other resources in both Nodes A and B.

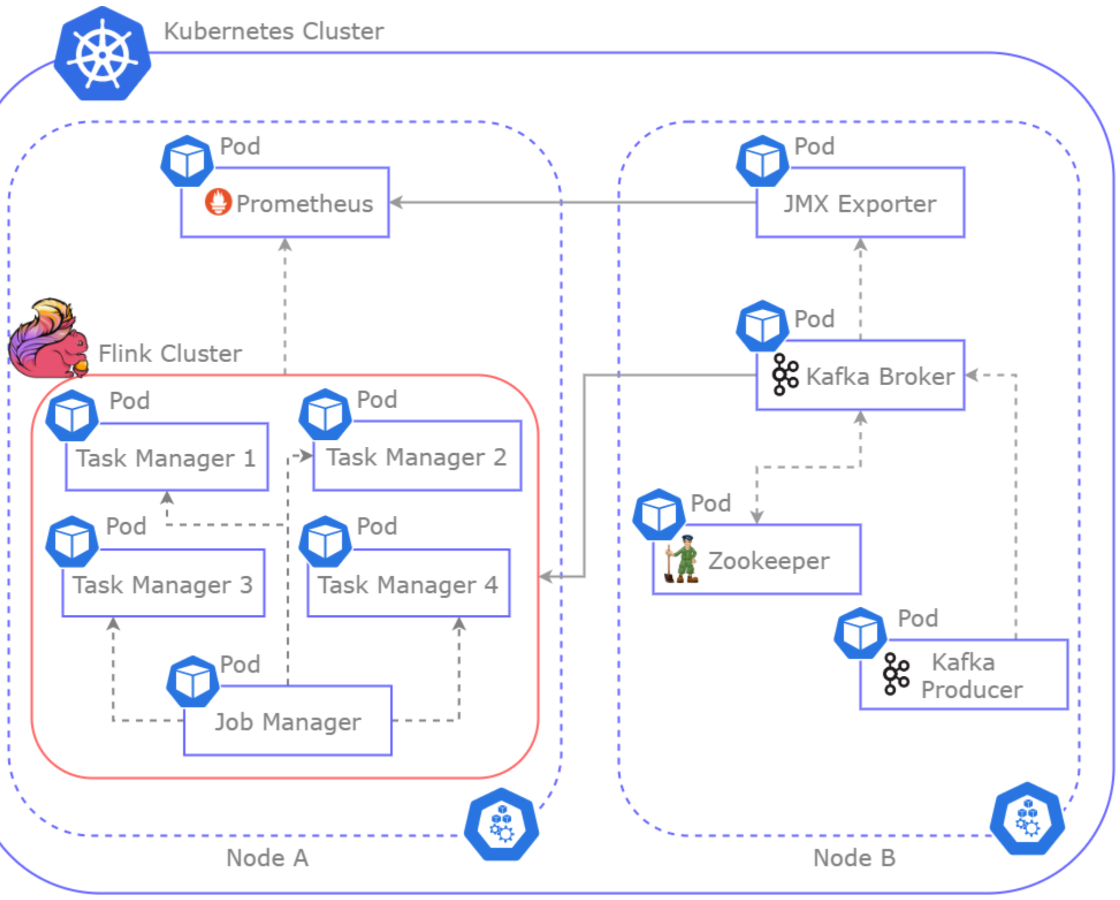


Figure 16: System architecture of our infrastructure with 4 TaskManagers present on the Flink cluster in Node A. Note that the autoscaler is not depicted

This is the default behaviour when adjusting the number of pods on a specific deployment (i.e. TaskManager) in a Kubernetes environment. Each pod created or destroyed is part of a single node unless the node capacity is exceeded (by default up to 110 pods can exist in a single node) in which case a new node is created if possible.

At this point it should be noted that the Google Cloud Platform (GCP), the cloud provider which we have deployed our cluster, includes charges per virtual machine (node) in a Kubernetes cluster. Our autoscaling algorithm (explained in detail on section 4.2) adjusts the allocated resources on a per-pod basis in a node as depicted in figure 16. This would mean that we would see no cost difference between our implementation when compared to HPA and Smilax since the number of nodes would remain the same through the whole scaling operation.

Normally, this is not an issue since a better pod allocation method on the same amount of nodes would still entail fewer infrastructure costs implicitly. In figure 16 we have made the assumption that in Node A only Apache Flink would be present and no other resource intensive application. By loosening this assumption and populating the node with a multitude of applications suddenly we have no way of knowing if the node will be stressed either by Flink or any of the other applications, which would result in the creation of additional nodes and thus incurring further infrastructure costs. It is thus made clear that even in the scenario of having Flink's entirety of TaskManagers in one node, efficient resource allocation is still paramount in order to achieve unnecessary costs.

The ideal scenario for an autoscaling agent which adjusts the number of pods in a cluster would be to stress a kubernetes node with only Apache Flink. However, this is not possible with our current processing capabilities since a node has a maximum capacity of 110 pods, far above the maximum number of TaskManagers (6) we have chosen for our evaluations.

Therefore, in order to better simulate the cost discrepancies between our algorithm and it's competitors, we have elected to make the assumption that each TaskManager of Apache Flink would populate its own node. This would mean that on the creation of a TaskManager a node would also be created and subseqeuntly at the deletion of a TaskManager its corresponding node would also be deleted. This scenario has been tested with all our experiments and has no practical limitations by both Kubernetes and Apache Flink.

This can be better visualized on figure 17 in which a scaling action has resulted in the creation of 3 TaskManagers (pods) and the subsequent creation of a node for each one of them. Notice that Flink did not require the scaling of the pod containing the JobManager which can still communicate with each TaskManager now present on different nodes. Additionally neither Prometheus needed to be scaled and can populate the original node containing the JobManager.

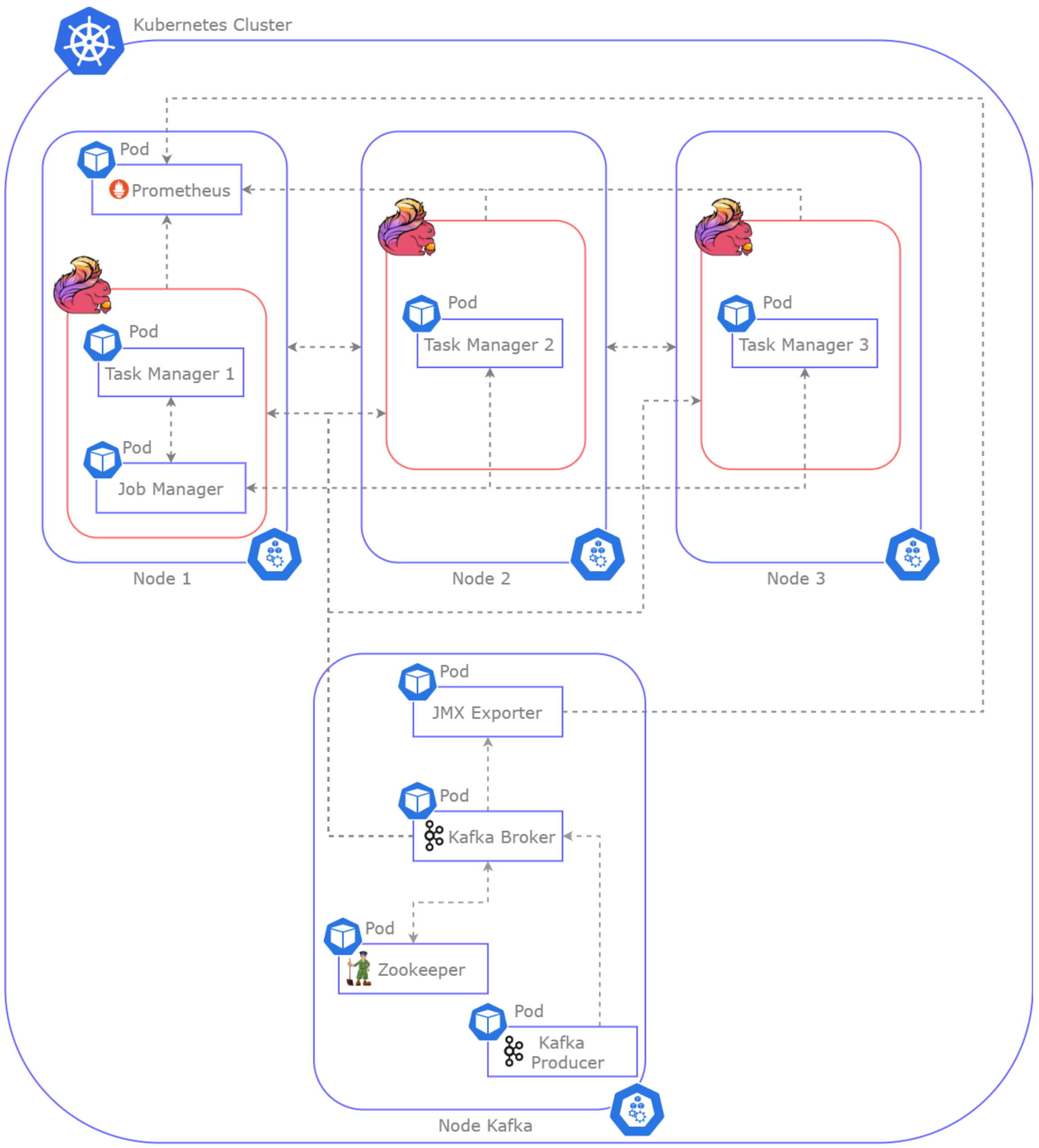


Figure 17: System architecture of our infrastructure with 3 TaskManagers present each populating its own node

4.2 HYAS: Hybrid Autoscaling Agent for Apache Flink

The implemented scaling policy used by our agent is based on [41]. The algorithm assumes that the streaming job of Flink reads input data from a Kafka topic, performs some sort of calculations over the records (i.e. identifying fraud events) and then either discards the results or feeds them again on another Kafka topic. The records produced in the topic are evenly distributed among partitions resulting in each consumer processing an equal workload.

In our work, a Kafka consumer exists in the form of a TaskManager. Each consumer (TaskManager) reads data from a Kafka topic which can have one or more partitions contained in the Kafka broker. The Kafka producer is responsible for creating and writing said data on the broker.

In Kafka, if the rate of production of records exceeds the rate at which they are being consumed, consumers will exhibit lag. This lag can be described as the difference between the number of records read by a consumer (committed offset) and the total number of records produced (latest offset) at a certain point in time.

The committed offset can only be calculated by the consumers (i.e. Flink), however accurately calculating this difference is not possible therefore Flink provides an upper bound for the total lag using the *records_lag_max* metric which indicates how far a consumer is behind the head of the message queue. In other words it represents the maximum lag in terms of number of records for any partition read by a consumer.

To better visualize this metric lets assume, for example, a Kafka topic containing 100,000 records evenly distributed between 10 partitions (i.e. each partition having 10,000 records) and 5 Kafka consumers (TaskManagers). For each consumer 2 partitions would be assigned each with 10,000 records. As the job is started, messages will begin to be consumed from each partition (faster than they are produced) decreasing the number of records on each partition (i.e. the lag) and thus the metric *records_lag_max* would give the largest value between the 2 partitions for each consumer.

By summing this value for each source task of Flink (i.e. for each consumer) we model the total lag of our application using equation (1).

$$totalLag = \sum_{i \in TaskManager} record\_lag\_max_i \tag{1}$$

A constant value of (1) indicates that our current amount of task managers are able to keep up with the workload. If this value is increasing, we need to scale up since we are underprovisioned. If this value is decreasing it means that we might be overprovisioned for the current workload.

Since we are looking whether the value from equation (1) is increasing or decreasing and not its numerical value a better representation of this would be to calculate the derivative of (1) as the rate of change of the total lag (in records/second) over a 1 minute period. Additionally, we calculate the total rate at which our job consumes records (throughput) by summing the *records_consumed_rate* metric from Flink for each consumer. Equation (2) summarizes this.

$$throughput = \sum_{i \in TaskManager} records\_consumed\_rate_i \tag{2}$$

The ratio between the derivative of (1) and (2) produces a dimensionless value which represents the rate at which the total lag of our Kafka queue is increasing (when having a positive value) or decreasing (when having a negative value) relative to the current throughput of our Flink cluster. This relation can be summarized in equation (3).

$$relativeLagChangeRate = \frac{deriv(totalLag)}{throughput} \tag{3}$$

With equation (3) we have answered the question *when* should we scale our resources which represents a prediction that the application will soon become overprovisioned or underprovisioned without the aid of a machine learning algorithm. A natural continuation of this thought would be *how much* should we scale our infrastructure. We begin by presenting the relation between the *throughput* of our system and the number of deployed TaskManagers (deployments or replicas). In an ideal scenario we would say that this is a one to one relationship. Meaning that if we double the number of our TaskManagers the throughput would also double (and vice versa an increase in throughput by a factor of two would mean that we have added double the number of machines).

To answer the question of how many task managers we need to add in order to match the incoming workload we basically need to answer the question of how much throughput we need in order for (3) to be equal to zero. Since equation (3) is a ratio it's value can only be zero only if the numerator is also zero.

Consider the following example with which we base our reasoning. Let the rate of change of the total lag be 4000 records/s and the throughput of our system 4000 records/s as well. The ratio of equation (3) would give us the dimensionless value of 1 as its result. In order for as to reduce the value of the numerator to zero we need to increase our throughput to a value of 8000 records/s so as to match the rate at which records are produced. By doubling the throughput of our system we have also doubled the number of machines (task managers) available for our cluster since we have established a one to one relationship between the throughput and the number of deployments. This means that equation (3) holds the necessary information for us to calculate the required amount of task managers needed in order to much the incoming workload. The above reasoning can be summarised in equation (4).

$$requiredDeployments = \lceil currentDeployments \times (relativeLagChange + 1) \rceil \quad (4)$$

By adding the identity element of the multiplication function (the number 1) in *relativeLagChangeRate* the resulting number is the multiplier necessary for the number of consumer replicas to keep up with the current workload. As stated in the above example, we have reasoned that we require double the number of existing machines in order to keep up with the load. To test the validity of equation (4) we take the value of *relativeLagChangeRate*, we add 1 and then multiply it by the number of current deployments present in our cluster (for example lets assume 2 task managers). The resulting number from this operation is 4 meaning that we must at least double our existing number of task managers in order to keep up with the load.

Most of the time the value of *relativeLagChangeRate* will not be an integer and thus we must overestimate the number of task managers needed for the current workload in order to achieve a smooth operation of our application. This is the reason why we have chosen to calculate the mathematical ceiling of (4).

Lastly, in order to prevent any outliers (such as spikes in the workload) from impacting our scale up decision and resulting in a false-positive scaling action we also use the *backPressureMsPerSecond* (with values from 0-1000 indicating how much backpressured a task is) metric from Flink. Backpressure [14] refers to the situation where a system is receiving data at a higher rate than it can process. Lets examine an example between server communications. Backpressure can commonly occur in this scenario when one server is sending requests to another faster than it can process them. If server A sends 100 rps (requests per second) to server B, but server B can only process 75 rps, we have a 25 rps deficit. Server B might be falling behind because it needs to do processing or it might also need to communicate with other servers downstream. The ideal reaction is to "backpressure" the whole pipeline from sink to the source, and throttle the source in order to adjust the speed to that of the slowest part of the pipeline, arriving at a steady state (this is how most stream processing engines react to backpressure). Equation (5) calculates this metric by finding the maximum value among all source tasks.

$$backPressure = max_{sourceTasks}(backPressureMsPerSecond) \tag{5}$$

Equation 5 basically acts as a safety net alongside *relativeLagChangeRate* to make sure that a scale up action is taken only when our consumers are backpressured and their throughput must me adjusted to that of the slowest part in the pipeline. However, in our experiments we noticed no difference in scaling actions when using this metric alongside *relativeLagChangeRate* most likely due to the fact that when the value of *relativeLagChangeRate* increases (i.e. when lag occurs in the Kafka queue) our consumers are already backpressured. Nevertheless, we will be presenting this metric in our algorithm for completeness purposes.

relativeLagChangeRate is only meaningful for scale up decisions, since a decreasing lag might still warrant the current number of replicas, until the application catches up to the latest records. In simple terms, there are instances when the value of *relativeLagChangeRate* is negative or even zero (i.e. our lag is decreasing or stabilizing) that we have no way of knowing if we are overprovisioned and our current number of consumers could decrease and still be able to keep up with the workload. We require a metric that can provides us information about the capacity (or load) that each consumer is operating at.

For this reason we can use the *idleTimeMsPerSecond* metric from Flink. This metric represents (in milliseconds) the amount of time in which a task has no records to process. The value of this metric can be between 0 and 1000 meaning that when it has its maximum value the task is idle and when it has its minimum value the task is processing records at maximum capacity. We use this metric for each consumer of our application and calculate the minimum between all of them. Equation (6) depicts this relation.

$$idleTime = min_{nonSourceTasks}(idleTimeMsPerSecond) \tag{6}$$

It is worth noting that for our scale up decision, the resulting number of task managers could change incrementally as well as non-incrementally due to calculating the value of *requiredDeployments* as described above. For our scale down decision we have chosen only to adjust the number of deployments in our cluster incrementally (i.e decrease the current number of deployments by 1). Since a scaling action which removes resources from a cluster is much more disruptive (especially in Flink) we wanted to avoid any unnecessary fluctuations in the number of TaskManagers in order to smooth out the resulting process.

By combining the above equations we can model Flink's behaviour for various workloads quite accurately and arrive at a decision making algorithm implemented on a control loop.

Algorithm 1 Scaling policy of hybrid controller

```
 1: while True do

 2:     totalLag = $\sum_{i \in TaskManager} record\_lag\_max_i$
 3:     throughput = $\sum_{i \in TaskManager} records\_consumed\_rate_i$
 4:     relativeLagChangeRate = $\frac{deriv(totalLag)}{throughput}$

 5:     requiredDeployments = $\lceil currentDeployments \times (relativeLagChange + 1) \rceil$
 6:     idleTime = $min_{nonSourceTasks}(idleTimeMsPerSecond)$
 7:     backPressure = $max_{sourceTasks}(backPressureMsPerSecond)$

 8:     if $relativeLagChangeRate > 0$ and $backPressure \geq threshold$ then
 9:         scaleUp(requiredDeployments)
10:     else if $relativeLagChangeRate \leq 0$ and $idleTime \geq threshold$ then
11:         scaleDown(currentDeployments - 1)
12:     end if

13: end while
```

Additionally, a cooldown period of 90 seconds is implemented between actions (i.e. no scaling decisions will be made for this period of time) in order to accommodate for Flink's required downtime when restarting with the new parallelism. It is also worth noting that the statements presented in lines 8 and 10 of algorithm 1 must be true over a period of 1 minute in order for a scaling action to be made. This is implemented in order to prevent outliers and small changes in *relativeLagChangeRate* to influence our scaling policy.

Line 8 contains a threshold value indicating when a scale up decision will be made if the backpressure metric exceeds this value. In our work we have chosen to set this threshold to the maximum value of *backPressureMsPerSecond* (i.e. 1000) since we want a scale up decision to be made when at least one source task is fully backpressured by the pipeline.

The threshold value presented on line 9 of algorithm 1 represents when a scale down decision will be made. The value of *idleTime* allows as to effectively control how fast our algorithm will release TaskManagers. For example, setting the value of the threshold closer to 0 (e.g. 100) will mean that TaskManagers will be removed when they are processing records almost at full capacity and the load cannot be shared on fewer machines if some are removed. Alternatively, setting the threshold at a value closer to 1000 (e.g. 900) will keep TaskManagers longer than they are needed since they are processing records at low capacity and the load could be shared on fewer machines. Optimally calculating this threshold is important in order to prevent situations when a TaskManager is removed but immediately after lag is exhibited and a scale up decision must be made. A good practice when not having sufficient information about this threshold is to set it closer to 1000 rather than 0 to prevent a situation as described above from happening, especially with Apache Flink.

In our work, we have chosen the value of the threshold at 500 since it represents a good balance between having idle TaskManagers not processing any records (thus increasing the cost of our infrastructure) and releasing TaskManagers too early when there are still records to be processed.

5 Evaluation

In this section, we discuss the purpose of the experiments concluded on section 5.2, we briefly present the inner workings of both algorithms that will be used as a benchmarks compared to HYAS (i.e. HPA and Smilax) and lastly present the application (along with its use case) that is running as our job on Apache Flink.

By conducting comparisons between a purely reactive approach on autoscaling Flink (Smilax's exploration mode) and a general purpose scaling controller (HPA) we aim to present scenarios where our hybrid approach (HYAS) can result in:

- More effective TaskManager allocation for a given workload (i.e. fewer machines used when not needed).
- Infrastructure cost reductions when deployed on a cloud environment using Kubernetes.
- Better performance of Apache Flink applications by reducing the maximum overall lag exhibited in the Kafka queue (i.e. better response times).

5.0.1 Smilax

Smilax [19] is an autonomous agent which monitors and maintains the performance of Apache Flink within acceptable limits. During a training phase (exploration mode), a reactive scaler collects workload and performance information and adjusts (scales-up or down) the number of TaskManagers whenever the performance limit (i.e. the SLA) is violated. During the optimal phase, the agent explores the performance and builds a statistical machine learning model which registers the optimal mapping between workload, performance and number of TaskManagers. For our experiments we will be comparing results from Smilax's reactive controller (exploration mode) as due to time constraints the optimal phase could not be implemented on Kubernetes.

The autoscaler is based on a reactive mechanism which acts as soon as the SLA is violated. The SLA is defined as the ratio between the number of records that have not yet been processed and remain in the Kafka queue (i.e. the size of the queue) and the rate of the workload. The simple reactive policy for a scale up decision on Smilax can be depicted as:

$$\frac{queue\_length}{workload} \geq SLA \tag{7}$$

In this work, the SLA threshold is 90% (i.e. less than 10% of the number of records can remain in the queue or more than 90% of the records are processed instantly).

The theoretical capacity for each TaskManager (i.e. the maximum records/sec it can handle) is calculated every time a scale up action is performed and is subsequently used for a scale down decision to be made only when more than 90% of TaskManagers are running than those needed.

It should be noted that Smilax implements a 4 minute cooldown between scaling actions (up or down) in order to allow for the Flink application to catch up to any accumulated records during its required downtime.

5.0.2 Horizontal Pod Autoscaler (HPA)

In Kubernetes the Horizontal Pod Autoscaler (HPA) [21] automatically scales the number of pods (either up or down) in a deployment, based on that resource's CPU or RAM utilization incrementally.

In simple terms, the scaling policy of the Horiontal Pod Autoscaler can be described as a ratio between a desired metric value and the current metric value of a resource. Equation 8 describes this relationship.

$$desiredReplicas = \lceil (currentReplicas \times (currentMetricValue/desiredMetricValue) \rceil \tag{8}$$

For example, if the current metric value is 200, and the desired value is 100, the number of replicas will be doubled, since 200/100 = 2.0. If the current value is instead 50, the number of replicas is halved since 50/100 = 0.5.

It is also worth noting that HPA's default behaviour includes a stabilization window (i.e. a period when no scaling action will be made) in order to reduce the fluctuation of the number of replicas in a deployment and prevent scaling up and down decisions to be made one right after the other. The value of this stabilization windows is 300 seconds (5 minutes) only when scaling down.

In our work, we have configured HPA with a target utilisation on CPU load of 90% (i.e. the desiredMetricValue of equation 8) meaning that the controller will adjust the number of TaskManagers only when the target utilisation is above (or below) this metric on average for each pod.

5.0.3 Flink application: Clicks Fraud Detection

In order to evaluate the performance of our decision making agent, a load testing application [16] was required to run as a job for Apache Flink. This application is based on click fraud [15] a type of fraud that occurs on the Internet in pay-per-click (PPC) online advertising. The owners of websites that post the ads are paid an amount of money determined by how many visitors to their sites click on the ads. Fraud occurs when a person, automated script, or computer program imitates a legitimate user of a web browser, by clicking on such an ad without having an actual interest in the target of the ad's link.

The application receives records from a Kafka topic which has the following JSON format:

```
{
    ”ip”: ”205.0.44.187”,
    ”userID”: ”e61b8f7a-5029-433a-9e44-79d5f514d309”,
    ”timestamp”: 1595431611,
    ”eventType”: ”display/click”
}
```

The application attempts to capture these kinds of frauds by searching for three patterns:

- Count the User IDs per unique IP address in a window of 60 seconds.
- Count the IP addresses per unique User ID in a window of 60 seconds.
- Calculate Click-Through Rate (CTR) per UID. In other words, is measures how many times an ad is clicked on, as compared to the number of times the ad is shown.

Consequently, the production rate of these records represents the workload of our application. The results for each pattern can then be fed through to a new Kafka topic, stored in persistent files or emitted as reports on dashboard application outside of the cluster for live analytics but in our case they are discarded since our application is mainly used as a load testing tool to stress the whole infrastructure.

5.1 Testbed

5.1.1 Configurations

We configured the Kubernetes cluster described on section 4.1 in the Google Cloud Platform (GCP). Our cluster was deployed in the europe-west1-d region with Kubernetes version (1.23.12). The type and amount of virtual machines running on the cluster are configured as a node pool, a template which contains all the specifications necessary to create a cluster node.

For our machines we decided to go with the e2-standard general purpose type with 8 CPUs, 32GB of RAM and an SSD of 40GB for persistent storage. The virtual machines are located in the same zone as our cluster running a Linux-based container-optimized OS that runs flawlessly with containers.

A summary of our configurations in GCP for our Google Kubernetes Engine is presented on table 1.

Attributes	**Configuration**
Location Type	Zonal
Zone	europe-west1-d
Cluster Version	1.23.12-gke-600
Machine Type	e2-standard-8
vCPU	8
RAM	32GB
Image	Container-Optimized OS with containerd
Boot Disk	SSD/40GB

Table 1: Cluster and Node configuration

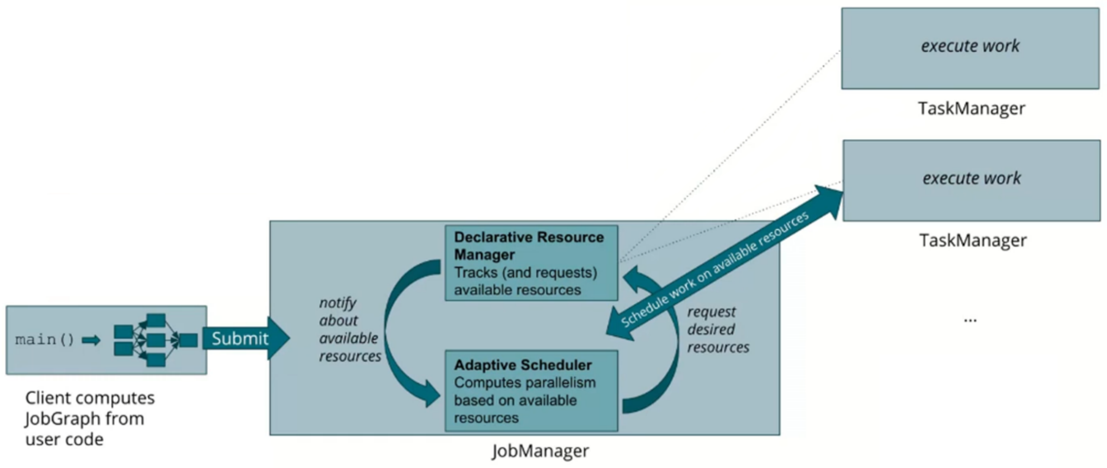
Flink

Flink is deployed as a Flink Application Cluster. In this way, the lifecycle of the cluster is dependent from the lifetime of the running job. For our evaluations we have chosen to submit a single running job.

The scaling operation on a Kubernetes cluster is presented on figure 17 in which on the creation of a TaskManager (Pod) a Kubernetes node would also be created and subsequently at the deletion of a TaskManager it's corresponding node would also be deleted. Each TaskManager has exactly one task slot. This means that the number of TaskMangers reflects the parallelism of our job. By scaling the amount of TaskManagers present we can scale the whole pipeline running on Flink as depicted in figure 11. With pipeline scaling, we are able to achieve an evenly-balanced load across tasks and operators and configure the parallelism of the running job by simply changing the number of available TaskManagers.

We have also configured Flink to use its new Reactive Mode feature [35]. As stated on section 1.1 Flink's parallelism can only be changed manually by stopping the job and restarting from the savepoint created during shutdown with a different parallelism. With previous versions of Apache Flink, each step of the aforementioned process needed to be initiated with an HTTP request to Flink's REST API containing all the necessary information for the scaling action. The Flink scheduler had no way of detecting when a new TaskManager was available in the cluster and so the process needed to be done manually.

Since Flink 1.13 an adaptive scheduler has been added as part of Flink's Reactive mode [35]. Reactive Mode configures a job so that it always uses all resources available in the Flink cluster (i.e. TaskManagers) automatically without needing to set up an external HTTP server to communicate with Flink's API in order to signal that a TaskManager has been added (or removed). With this mode, we only need to change the replica factor of a Kubernetes deployment and Flink's scheduler will handle the process of stopping and restarting our job with the new parallelism. The above description of Flink's adaptive scheduler is depicted in figure 18.



execute work
TaskManager
execute work
TaskManager
...
main() ➡
Submit
Client computes JobGraph from user code
notify about available resources
Declarative Resource Manager
Tracks (and requests) available resources
Schedule work on available resources
request desired resources
Adaptive Scheduler
Computes parallelism based on available resources
JobManager

Figure 18: Reactive Mode of Apache Flink

Kafka Broker

The Flink application consumes records from a specific Kafka topic by assigning each TaskManager (Kafka consumer) a certain number of partitions. There are two distinct cases which we need to keep in mind when scaling an application with Kafka as its source.

- When there are more TaskManagers (consumers) than Kafka partitions, some of the consumers will just remain idle, not reading any data. This will result in an unbalanced distribution of the workload compared to the number of availabe TaskManagers.
- When there are more Kafka partitions than TaskManagers, consumers will be assigned to multiple partitions at the same time. If the total number of partitions can not be evenly divided with the total number of TaskManagers then some consumers will be assigned less partitions than some others unbalancing the distribution of the workload.

Kafka does not support the dynamic increase or decrease of the partition number for an existing topic. For this reason, we set a static number of partitions during the creation of the topic. The number of partitions must be at least equal to the maximum number of TaskManagers in our infrastructure in order to avoid having idle consumers.

An adequate solution to counter the workload skew that is created when all consumers are not assigned the same number of partitions could not be found and is beyond the scope of this thesis. Thus, to circumvent this issue we have chosen to create 60 partitions in a topic and have no more than 6 TaskManagers running in our cluster. This aleviates the problem since the number 60 can be divided with a remainder of zero for all the numbers from 1 through 6.

Kafka Producer

The Kafka producer for this particular application is responsible for generating records at various rates to be written at a specific Kafka topic. This is a achieved by using a python script which simulates a 'click' in the form of a JSON file with a randomly generated IP, UUID and timestamp representing a single record. These records are then written to a Kafka topic at various intervals simulating a fluctuating real time workload. It should be noted, that in order to achieve a more evenly distributed balance for our workload, the records are written in each partition of the topic in a round-robin fashion.

The rate at which these records are written needs to be such that our application will be stressed and thus would require additional machines in order to keep up with the increasing workload. This fluctuating workload, allows us to simulate various points in the application's lifetime where our resources are overprovisioned or underprovisioned and thus would require a corresponding scaling action to be taken. In this way, we can efficiently test the validity, stability and performance of our autoscaling agent.

5.1.2 Infrastructure Cost Calculating Function

In order to create a cost calculating function for our infrastructure, we must first understand how the Google Cloud Platform (GCP) charges for it's Kubernetes cluster. According to the GCP pricing documentation [20], the main factors that GCP charges upon are the CPU and RAM allocation per machine with prices varying per region. Ingress network traffic is not charged, while egress traffic is charged based on the amount of data exchanged between Nodes. In our work, egress traffic is considered (as per figure 17) any data exchanged between Node Kafka and each node containing a TaskManager (i.e. records sent to TaskManagers for consumption, metrics from JMX to Prometheus etc.) as well as any necessary data or metrics exchanged between nodes with each TaskManager (Nodes 1 to 3). Allocated storage space is also charged by GCP, but for comparison purposes it's cost is constant for all different algorithms since it is not adjusted.

Table 2 summarizes our infrastructure's costs per machine.

Resources	**Cost (US$)**
Predefined vCPU	\$0.023993/vCPU/hour
Predefined RAM	\$0.003216/GB/hour
Ingress Traffic	\$0 (not charged)
Egress Traffic	\$0.01/GB

Table 2: GCP pricing

From our experiments, we have observed that the total egress traffic in our Kubernetes cluster is below 100GB therefore the total cost for each experiment is less than \$1. Thus we would be omitting it from our cost calculating function since the main cost difference lies in the number of nodes (i.e. TaskManagers) each algorithm chooses to create at any given time. Equation (9) summarizes this function.

$$totalCost = totalCost_{CPU} + totalCost_{RAM} \tag{9}$$

where

$$totalCost_{CPU} = nCPU \times vCPU_{cost} \times nodes \tag{10}$$

and

$$totalCost_{RAM} = totalRAM \times RAM_{cost} \times nodes \tag{11}$$

5.2 Experiments

In this section, we present all our experimental results regarding the performance and cost of our autoscaling decision making agent for Apache Flink. We also make comparisons between Smilax's [19] exploration mode and Kubernetes' default Horizontal Pod Autoscaler [21] (HPA) configured at 90% CPU utilization (presented briefly on sections 5.0.1 and 5.0.2 respectively).

Our benchmarks inlcude 3 set of workload tests aimed to reproduce as many possible realistic scenarios for our application when under load.

- Synthetic workloads in the form of a Gaussian distribution and a volume spike of records.
- Real life workloads: a dataset containing an aggregation of the requests made to the 1998 FIFA World Cup Web site.

All our experiments run the clicks fraud detection application (described on section 5.0.3) as a Flink job.

5.2.1 Gaussian

A Gaussian bell curve workload aims to reproduce a gradually increasing curve, following with a peak period and then smoothly returning to its initial value. This type of workload allows us to assess the behaviour of our autoscaling agent for a gradual increase and decrease in the volume of records produced over a relatively long period of time.

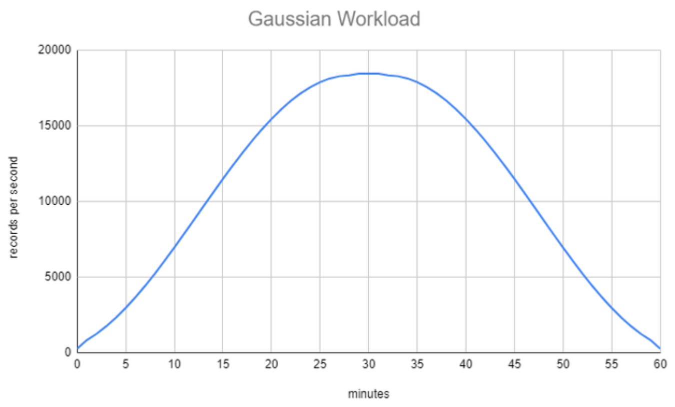


Figure 19: Gaussian Distribution Workload. Note that our workload curve starts from zero and hitting a peak of ∼18000 records/s over a period of 1 hour.

Figure 20 depicts the time shift between our 3 algorithms (i.e. when each one of them decided that a scale up or scale down decision must be taken for a certain point in time). Note that the points in time our application was stressed and a scale up action was required was on the 15 minute mark where all algorithms scaled from 1 to 2 TaskMangers and at the 27-28 minute mark where HPA and HYAS scaled from 2 to 3 and Smilax from 4 to 5 TaskManagers.

We observe that all 3 algorithms scaled up from 1 task manager to 2 at relatively same time signatures with Smilax continuing to add TaskManagers up to a maximum of 6. We can also observe from figure 20 that each algorithm scaled up incrementally (i.e. added only one TaskManager at a time). This can be attributed to the fact that a Gaussian workload is a gradual and predictable increase (or decrease) with no abrupt changes in record production alleviating the need for making scaling up decisions which add more than one TaskManager at a time.

Tables 3 and 4 depict various metrics taken at the moment Smilax and HYAS decided that a scale up decision must be made respectively. We notice that Smilax scales up in situations when the Kafka queue lag is decreasing, enabling only a small number of records to be processed by Flink due to the downtime necessary in order for the parallelism to be adjusted. This behaviour can be explained due to the fact that Smilax's scaling policy (as explained on section 5.0.1) is a static number depicting the ammount of records that remain in the Kafka queue relative to the workload and not the rate of change (i.e increase or decrease) of the queue. In contrast with Smilax, HYAS decides that a scaling action will be taken only when the rate of change of the Kafka queue is increasing thus resulting in fewer TaskManagers needed in total compared to Smilax. This allows the Apache Flink application running, to process records with the least amount of disruptions (downtime) possible.

Task Managers	**Lag change rate**	**Slow records (SLA = 0.1 or 10%)**	**Workload**
$1 \rightarrow 2$	> 0 (lag is increasing)	0.14	9037
$2 \rightarrow 3$	< 0 (lag is decreasing)	0.10	13395
$3 \rightarrow 4$	< 0 (lag is decreasing)	0.82	16519
$4 \rightarrow 5$	< 0 (lag is decreasing)	1.59	18186
$5 \rightarrow 6$	< 0 (lag is decreasing)	1.41	14231

Table 3: Smilax metrics for each scale up action taken (6 maximum TaskManagers)

Task Managers	**Lag change rate**	**Slow records (SLA = 0.1 or 10%)**	**Workload**
$1 \rightarrow 2$	> 0 (lag is increasing)	0.15	8974
$2 \rightarrow 3$	> 0 (lag is increasing)	0.07	18270

Table 4: HYAS metrics for each scale up action taken (3 maximum TaskManagers)

Comparing HYAS with Kubernetes' HPA we can observe that while both utilize the same maximum number of Task Managers their main difference lies in the time they require to scale back down to 1 task manager when the workload subsides. We notice a 5 minute delay between the two implementations when reducing the number of TaskManagers from 3 to 2 and a 10 minute delay when returning from 2 TaskManagers to 1. The main reason for this slow reaction from HPA to release TaskManagers can be attributed to the 5 minute cooldown between scaling decisions that exists to it's default configuration (as explained in section 5.0.2) compared to HYAS's 90 seconds.

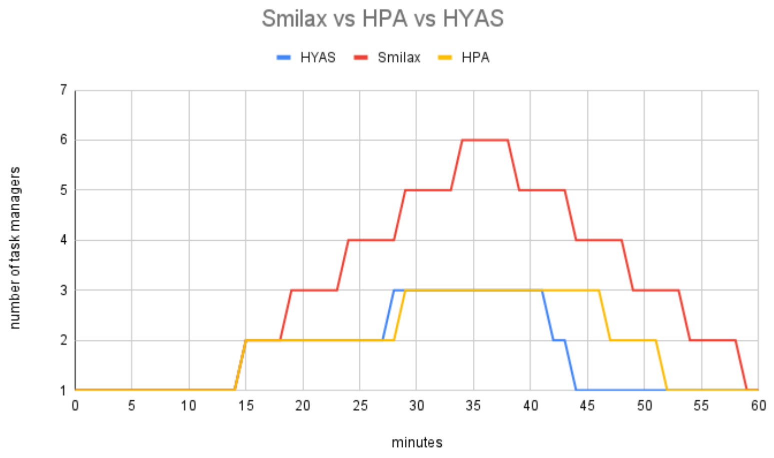


Figure 20: Time Shift for each Algorithm

Consequently, we can infer from figure 21 that our implementation utilizes, on average, a smaller amount of task managers for the duration of the test thus resulting in fewer infrastructure costs as depicted in figures 22 and 23 compared to both HPA and Smilax. For the resulting differences in GCP pricing we have utilized equation (9) described on sub-section 5.1.2

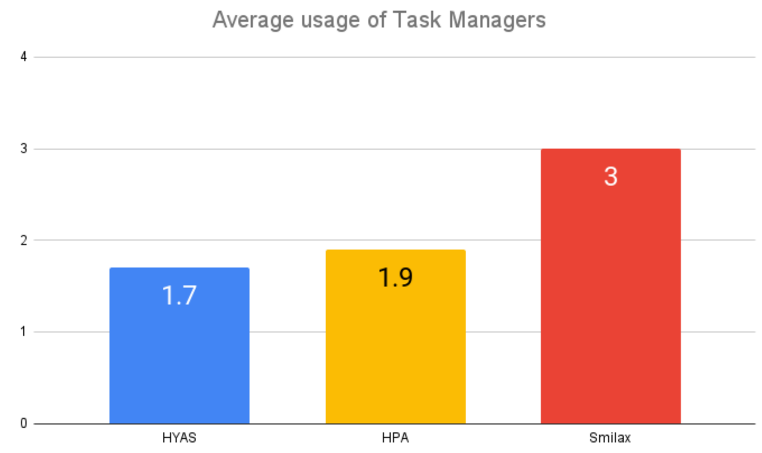


Figure 21: Average Usage of Task Managers

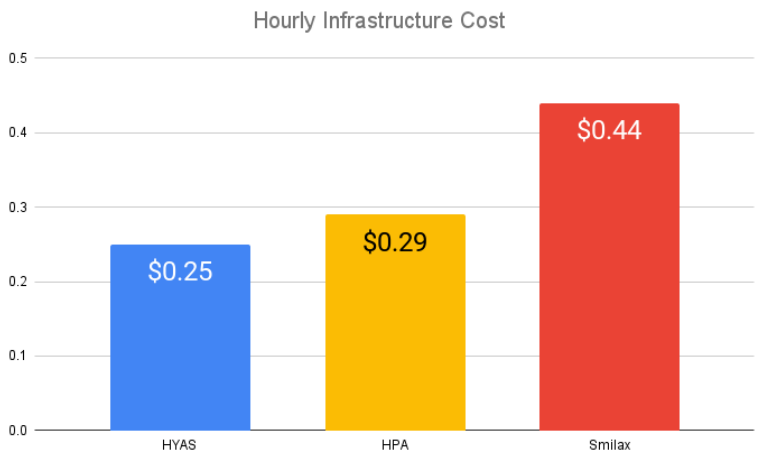


Figure 22: Average Hourly Cost (Gaussian)

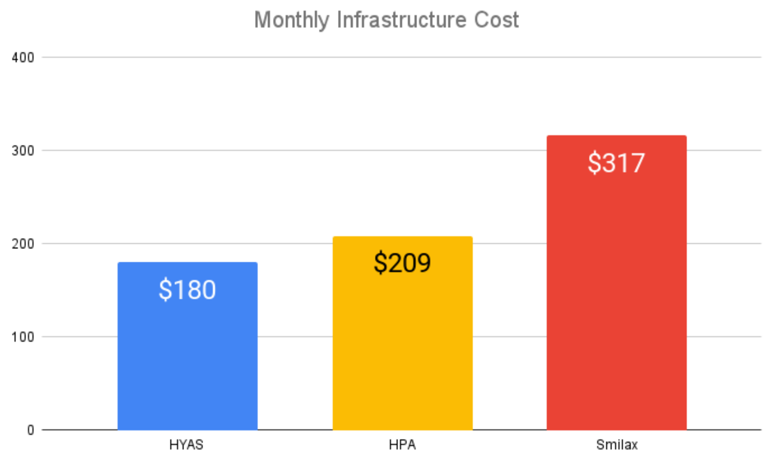


Figure 23: Average Monthly Cost (Gaussian)

In order to better evaluate the efficiency of our autoscaling method, infrastructure costs must be presented alongside performance comparisons for a certain system. When measuring the performance of a Flink cluster which consumes records from a message queue (such as Kafka) as its input stream, the only way of depicting if an application is keeping up to a given workload is to determine whether records are processed in a timely manner and no lag is exhibited (i.e. if consumers are keeping up with the producers). This can be measured by observing the maximum record lag for each system (i.e. each algorithm) when stressed in a particular point in time.

We present the performance of Apache Flink clusters by measuring the time each algorithm requires to empty the Kafka queue when under stress for its corresponding number of task managers. This response time, is directly associated with the maximum number of record lag observed in the Kafka queue of each system when under stress meaning that whichever algorithm produces a smaller maximum queue lag when the application is stressed will have a faster response time.

Figures 24 and 25 depict this relationship between the response time (in minutes) of each algorithm and its maximum observed queue lag when under stress. Note that the response time does not include the downtime for Flink to restart its job. Additionally, all algorithms are stressed the most at around the same point in time (at around 18000 records/sec or the 27-30 minute mark on figure 19) despite having a different amount of TaskManagers available at the time (throughput). This can be attributed to the fact that Smilax has not allowed Flink sufficient time in order to process the volume of records present on the queue since it triggers a scale up decision every 4 minutes thus introducing unnecessary downtime where no records are being processed and the lag is increased.

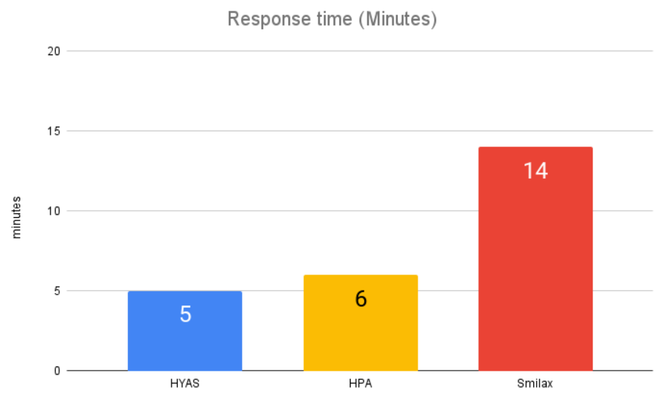


Figure 24: Time for each Algorithm to Empty the Kafka Queue

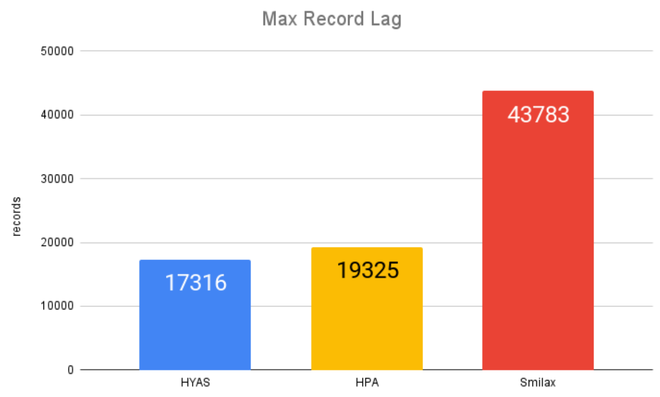


Figure 25: Maximum Observed Queue Lag Under Stress

From figures 24 and 23 we can observe that at best our algorithm performs better compared to both HPA and Smilax and at worst outperforms Smilax but its performance is on par with HPA while always incurring fewer infrastructure costs for this particular workload distribution.

5.2.2 Spike

The purpose of creating a distribution that produces records at high volume abruptly is to test the efficiency of our algorithm when confronted with these types of spikes in the workload. With this test we can also assess the behaviour of our decision making agent for taking non-incremental scale up actions. A feature which both HPA and Smilax could not reproduce.

Our spike workload reaches ∼20000 records/s almost instantaneously and remains there for a period of 5 minutes where it then drops back down to 0.

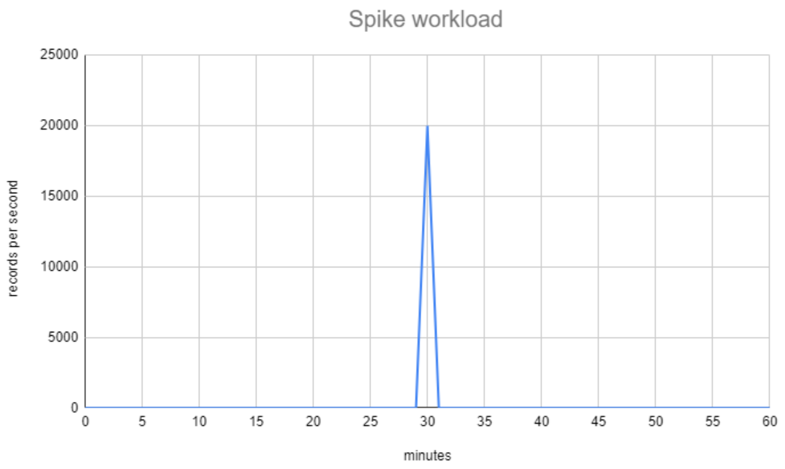


Figure 26: Abstract form of a Spike Workload

Figure 27 depicts the observed time shift between our 3 algorithms. We notice that our implementation reaches its maximum required TaskManager capacity without delay in a non-incremental fashion while both HPA and Smilax do so incrementally. Notice that Smilax again reaches a maximum TaskManager number of 6 compared to only 3 which HPA and HYAS require. This is for the same reasons presented on section 5.2.1 when compared to HYAS.

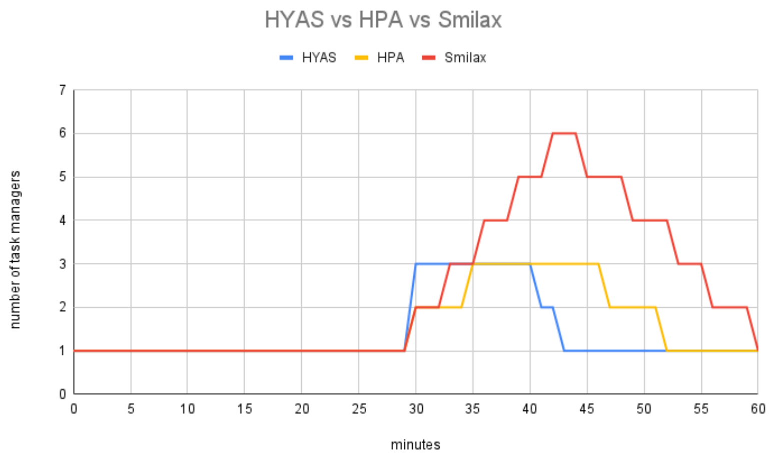


Figure 27: Time Shift for each Algorithm

This non-incremental fashion which HYAS chooses to adjust resources (i.e. add as many TaskManagers as required) results in a significant decrease in the maximum observed Kafka queue lag for our system and thus a much faster response time to empty this queue as observed in figures 28 and 29.

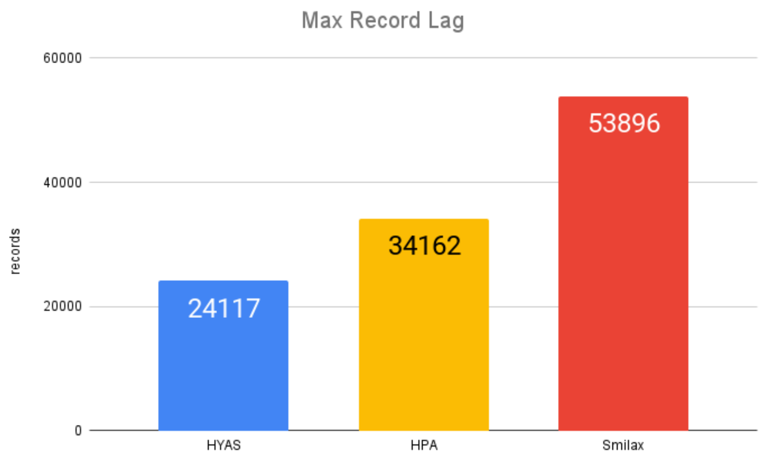


Figure 28: Maximum Observed Queue Lag Under Stress

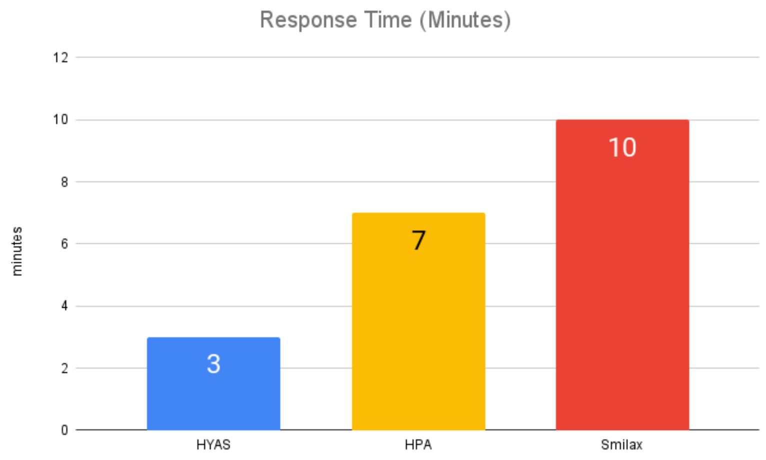


Figure 29: Time for each Algorithm to Empty the Kafka Queue

The performance differences presented on figures 28 and 29 are mainly attributed to the required downtime that Flink needs in order to change the parallelism of a job. When a scaling action is taken Flink must take a savepoint, restart the job with the new parallelism and catch up to any accumulated records that were written in the Kafka queue while it was restarting. In this particular test, our algorithm takes only 1 scaling action (for scaling up) in order to match the incoming workload, while HPA takes 2 and Smilax 5, reducing the time during which Flink is not processing any records and resulting in less overall lag present.

Despite this non-incremental scaling action which our algorithm chooses to implement (when scaling up) the resulting infrastructure costs depicted on figures 30 and 31 are still less than HPA and Smilax. Since our scaling method reacted to the spike in the workload with only 1 scaling action it was able to empty the Kafka queue faster which resulted in releasing any unnecessary task managers quicker (albeit non-incrementally) and thus incur less costs on average compared to both HPA and Smilax.

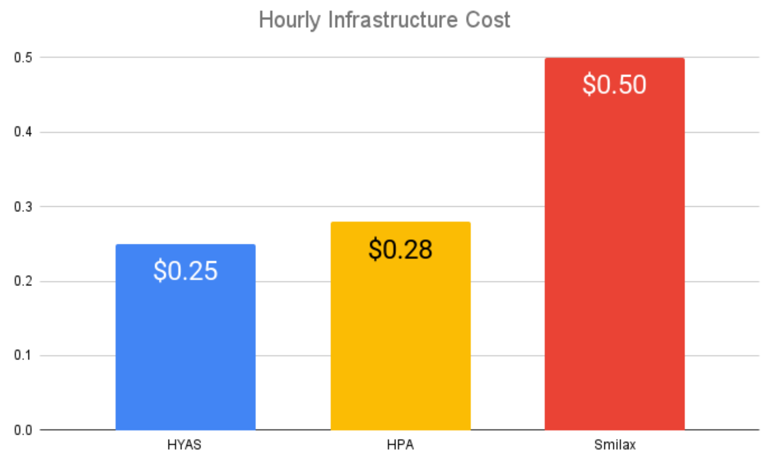


Figure 30: Average Hourly Cost (Spike)

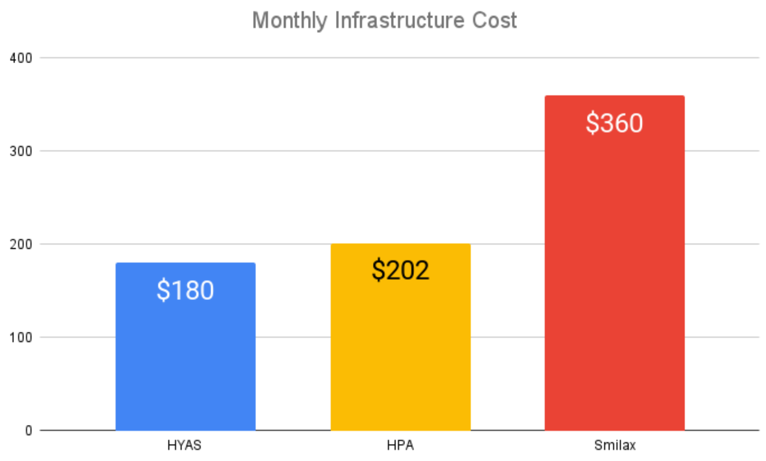


Figure 31: Average Monthly Cost (Spike)

5.2.3 FIFA 98 World Cup Dataset

The FIFA 98 Dataset consists of all the requests made to the 1998 World Cup Web site between April 30, 1998 and July 26, 1998. During this period of time the site received 1,352,804,107 requests.

The dataset exists in a format that could not be used as an input by our Kafka producer. In order to address this issue, we converted and scaled the whole dataset to be more in line with our synthetic experiments (i.e. having proportional peaks and valleys in the workload) and chose a single day as our subset from the whole dataset.

The resulting subset is a 24-hour period of aggregated requests per minute for which we choose to conduct our test on a 9.5 hour slice. Since our previous evaluation workloads where synthetic a real life distribution of requests per second provides a more realistic insight on the behaviour of our autoscaling algorithm.

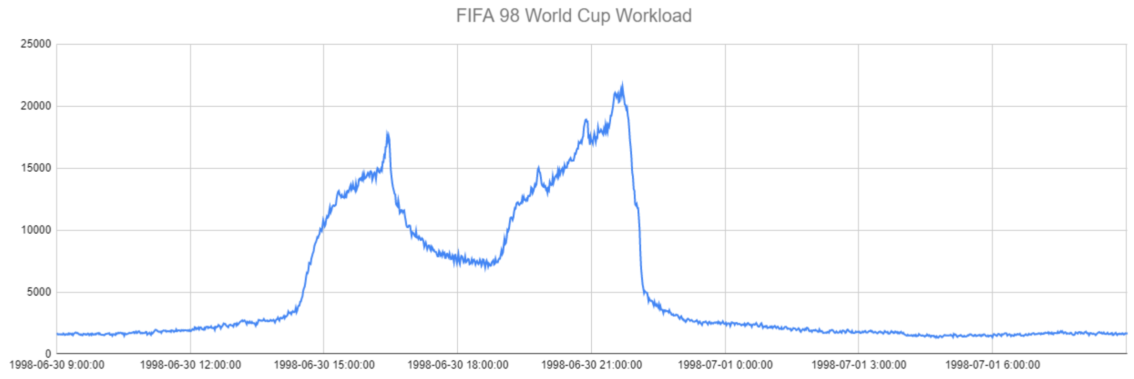


Figure 32: FIFA 98 World Cup Workload

The 9 and a half hour slice which we chose for our evaluation is between 1998-06-30 14:00 and 1998-06-30 22:30 (from figure 32) since it is the period with the most activity in terms of requests per minute for the world cup site.

Figure 33 and 34 represents the observed time shift between all algorithms when compared to HYAS. Note that the points in time our application was stressed and a scale up action was required was on the 15:00 and the 19:00 minute mark (where all algorithms scaled from 1 to 2 TaskMangers) and at the 21:30 minute mark where each algorithm chose to adjust the number of TaskManagers differently.

The most important key points that we can present when comparing HYAS with HPA from figure 33 are as follows:

- When compared to HPA, we can observe that our algorithm chooses to adjust the maximum available number of TaskManagers to 3 while HPA chooses to remain with 2 when the application is stressed the most at around the 21:30 minute mark on figure 32.
- HPA decides to remain with 2 TaskManagers at around the 16:30 minute mark when the workload subsides while HYAS reduces its available resources to 1 TaskManager.
- We observe fluctuations in the number of TaskManagers from HPA (at around the 18:30 minute mark) indicating that a stable state cannot be reached and the algorithm continuously adjusts the available resources. In contrast HYAS makes a scaling action only when necessary, avoiding disruptions in Flink's processing.

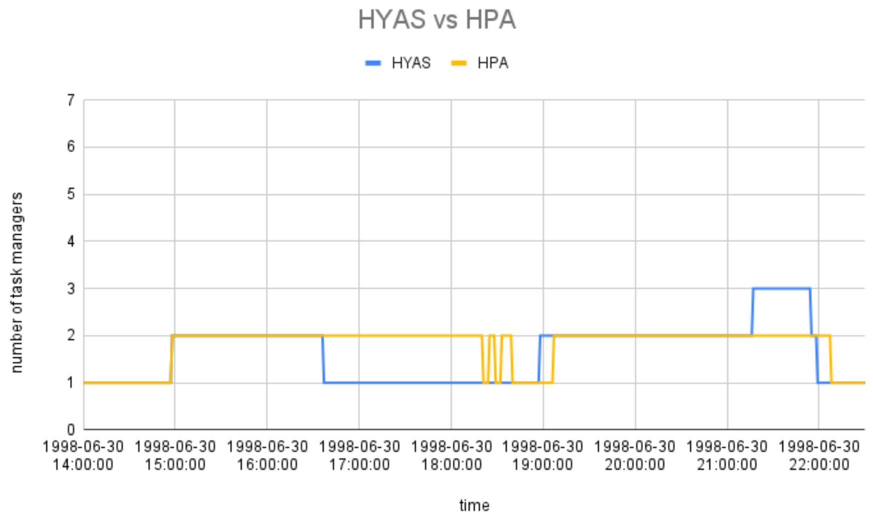


Figure 33: Time Shift for HYAS and HPA

Additionally, when comparing HYAS with Smilax, the most important key points that we can present from figure 34 are as follows:

- We observe that Smilax performs more in line with HYAS when presented with a real life workload spanning around 9 hours. It seems that a slower, more gradual increase in the production of records allows for Smilax sufficient time to remove any records remaining in the queue and not trigger additional scaling actions when the application is first stressed at around the 15:00 and 19:00 minute mark.

- However, the same cannot be said when the workload reaches its maximum output at around the 21:30 minute mark. Here we again observe Smilax triggering a chain reaction of scaling actions reaching a maximum number of 6 TaskManagers when the application is stressed the most. We can conclude that an abrupt change in a small period of time when the application is stressed the most (i.e. when we have reached the maximum capacity of the available TaskManagers) is the situations where Smilax makes more scaling actions than necessary.

- Similar with HPA, Smilax chooses to remain to 2 TaskMangers at around the 16:30 minute mark (where HYAS scaled down to 1) finally reducing that number to 1 at around the 17:45 minute mark.

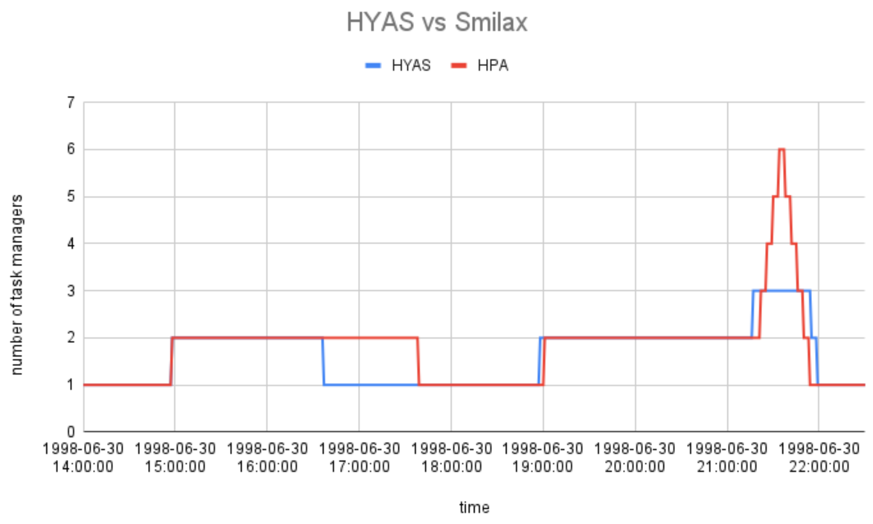


Figure 34: Time Shift for HYAS and Smilax

Figure 35 presents the average usage of TaskManagers that each algorithm chose to use for the whole 9.5 hour duration of the test. Note that Smilax and HPA have the same usage of TaskManagers (on average) despite choosing a different number of maximum machines. This can be attributed to the fact that on long running real life workloads, variations in the number of TaskManagers have little impact on cost when used on a small period of time (as we observe from Smilax's and HYAS's scale up actions on the 21:30 minute mark that lasted for only about 30 minutes).

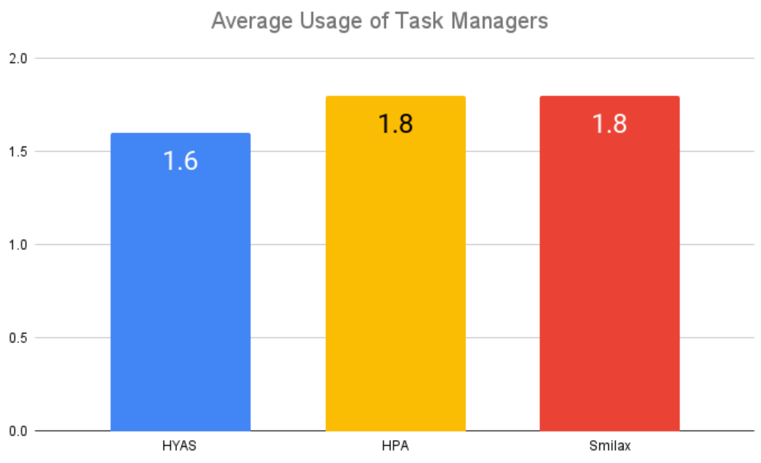


Figure 35: Average Usage of Task Managers

Consequently, we observe from figure 36 that HYAS results in fewer overall infrastructure costs compared to HPA and Smilax. This is mainly credited on HYAS's faster scale down action taken around the 16:30 minute mark on figures 33 and 34 which resulted on fewer TaskManager utilization (on average) and thus less costs in contrast with HPA and Smilax.

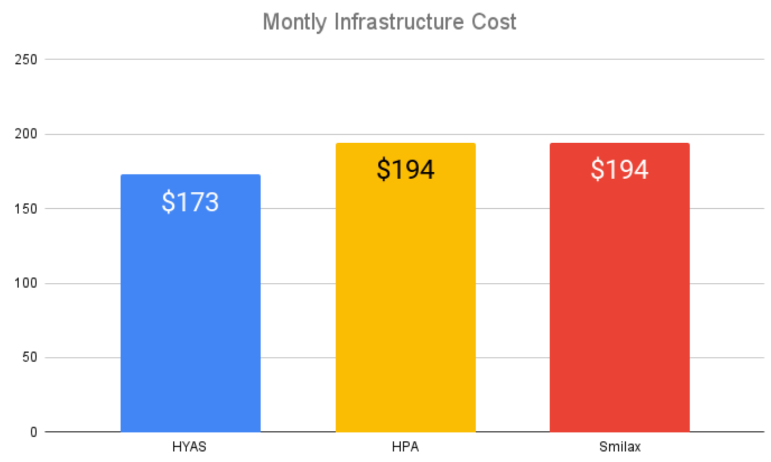


Figure 36: Average Monthly Cost (FIFA 98)

Lastly, figures 37 and 38 present the performance (i.e. response time) and maximum record lag for each algorithm measured at the point in time in which our application was stressed the most (i.e. the 21:30 minute mark). When comparing performance metrics for each algorithm we can observe that HPA had the worst performance followed by Smilax and then HYAS.

HPA's reduced performance in this particular workload distribution is attributed to the fact that it failed to make a scale up action when the application was stressed the most at around the 21:30 minute mark, resulting in unprocessed records to be piling up on the Kafka queue as the current number of TaskManagers (throughput) could not keep up. This further illustrates the fact that general purpose autoscaling controllers which base their scaling policy on simple metrics (such as CPU utilization) are ill suited for adjusting the available resources of Apache Flink.

Smilax's performance in this real life workload can be explained from the fact that it reacted successfully when the application was stressed the most and thus triggering a scale up decision (compared to HPA). However, this scaling action resulted in more unnecessary scale up decisions to be made which hindered Flink's ability to process records effectively due to the downtime incurred by these decisions.

Conversely, HYAS achieves the best performance among the three algorithms due to the fact that it successfully identified the need for a scaling decision to be made when it was needed (compared to HPA) without overshooting the required number of TaskManagers for this particular workload distribution (compared to Smilax).

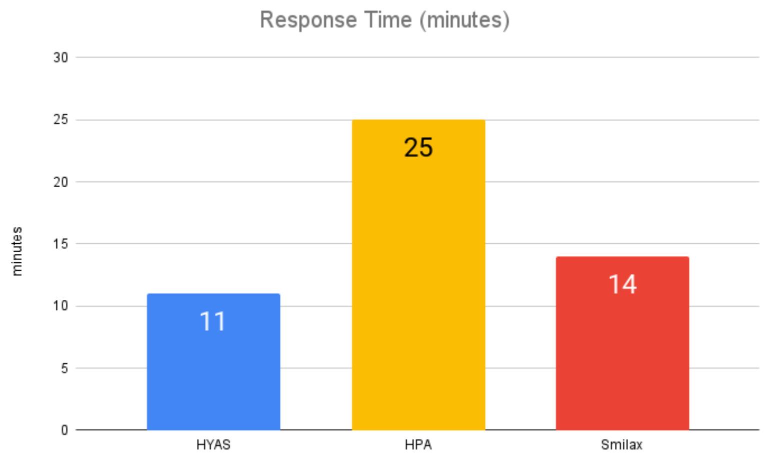


Figure 37: Time for each Algorithm to Empty the Kafka Queue

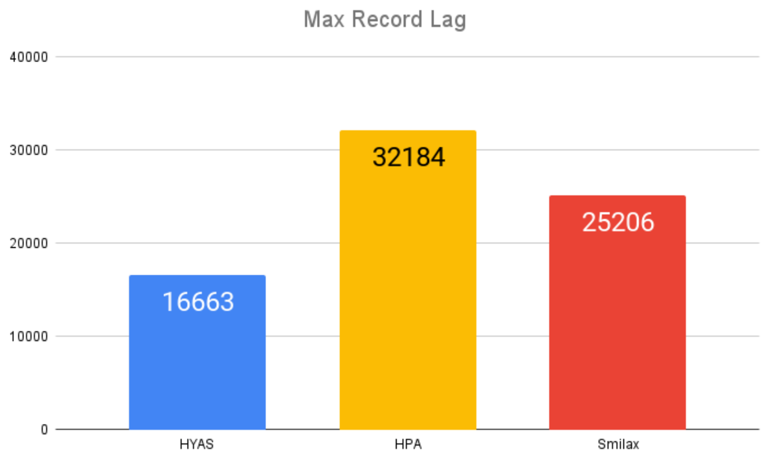


Figure 38: Maximum Observed Kafka Lag Under Stress

6 Conclusions

In this thesis, we have worked to create and implement an effective decision making agent for autoscaling Apache Flink with the goal of minimizing infrastructure costs in a cloud environment while still maintaining acceptable performance metrics. This work, has been deployed in the cloud using Google's Kubernetes Engine in order to utilize its effective elasticity mechanisms for creating and scaling container based applications.

In order to support Flink's high throughput and in-memory speed requirements we have paired it with Apache Kafka, a distributed event streaming platform, as its streaming source in the form of a publish/subscribe system. We expand the capabilities of a static Flink cluster by providing dynamic resource allocation to its task managers in order to accommodate the scaling need that a wide series of fluctuating workloads creates.

A known limitation of Apache Flink that must be taken into account when designing an autoscaling mechanism is that the parallelism of a Flink application cannot be modified at runtime. Restarting the job with a new parallelism from a savepoint can take a significant amount of time since the state must be written on a persistent storage (or on memory), new resources must be allocated (pods) and then the job must be restarted with the new parallelism. In this time no incoming records are being processed until the pipeline is restored to its previous state. This means that when the application is successfully restarted it must try to catch up to any accumulated records that were not being processed during this downtime.

Our proposed solution to this limitation is implemented in the form of an autoscaling agent which monitors Flink's cluster and models its behaviour using changes to the input's record lag and operator idleness. Through a decision making algorithm, optimal modification of Flink's underlying resources is achieved.

We evaluate the performance and cost requirements of our decision making model by comparing it with Kubernetes' Horizontal Pod Autoscaler (HPA) and Smilax's exploration mode through a variety of synthetic and real life workloads aimed to stress our application in various points in time. The results of these benchmarks have shown that our implementation provides a better combination of performance-to-cost ratio than both HPA and Smilax in all tested scenarios. The benefits mentioned above prove that our decision making agent can be used as a proof of concept and basis for designing and improving optimal autoscaling mechanisms specifically for Apache Flink.

7 Future Work

The work presented on this thesis while effective on its original goal of creating an autoscaler for Apache Flink leaves room for further improvements to be implemented.

- Currently, our autoscaler takes rescaling actions that effect the parallelism of the whole pipeline in the Flink job since each Task Manager is equipped with one task slot. Allowing Task Managers to be configured with more than one task slot and rescaling each operator independently from the pipeline as proposed on [25] would improve the overall resource utilization of the job and thus reduce infrastructure costs.

- A (linear or non-linear) correlation could be established between the state size that Flink needs to store when taking a savepoint and the total amount of downtime required for restarting a job. This would greatly improve the additional time (downtime) that the application requires in order to catch up to any accumulated records when a new parallelism is implemented and a job is restarted.

- Supporting multiple Flink Jobs with multiple Kafka topics. Currently, our implementation utilizes only one job reading from a single Kafka topic as its input stream.

  In order for our infrastructure to support multiple topics on multiple jobs, Flink's cluster mode will have to change to a session cluster (3.1.6) being independent of all running jobs.

  Additionally, we must determine whether more topics would require more replicas of our Kafka broker in order to support the additional workload (a monitoring system targeting Kafka's load and capacity would be required).

  Lastly, our autoscaling algorithm would not have to change since it monitors exchibited lag on all Kafka partitions regardless of the number of topics or jobs running.

- While our current decision making agent is threshold-based, a more advanced reinforcement learning (RL) method could be implemented.

  Threshold-based rules are simple to implement and deploy in a cloud environment but their performance is depended on the quality of those thresholds which most of the time are unknown and differ from system to system.

The important feature of RL approaches is learning without prior knowledge of the target scenario in an online fashion with actual observations. However, the main drawback of this method is that it requires a training phase, and the solution is tailored to the characteristics of the workload. This means that a solution (if any) is not stable for floating workloads and that training has to recommence each time the workload changes. In turn, the stability of the existing solution has to be tested periodically in order to identify when training has to be rerun.

References

[1] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. ≪Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned≫. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 970–973. DOI: `10.1109/CLOUD.2018.00148`.

[2] *Amazon Kinesis*. URL: `https://aws.amazon.com/kinesis/`.

[3] *Apache Flink Applications*. URL: `https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/learn-flink/overview/`.

[4] *Apache Flink Architecture*. URL: `https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/concepts/flink-architecture/`.

[5] *Apache Flink Checkpoints*. URL: `https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/ops/state/checkpoints/`.

[6] *Apache Flink Description*. URL: `https://flink.apache.org/flink-architecture.html`.

[7] *Apache Flink Savepoints*. URL: `https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/ops/state/savepoints/`.

[8] *Apache Flink Stateful Operations*. URL: `https://nightlies.apache.org/flink/flink-docs-release-1.16/docs/concepts/stateful-stream-processing/`.

[9] *Apache Flink Use Cases*. URL: `https://flink.apache.org/usecases.html`.

[10] *Apache Heron*. URL: `https://incubator.apache.org/clutch/heron.html`.

[11] *Apache Kafka*. URL: `https://kafka.apache.org/`.

[12] *Apache Zookeeper*. URL: `https://zookeeper.apache.org/doc/r3.8.0/zookeeperOver.html`.

[13] Hamid Arabnejad, Claus Pahl, Pooyan Jamshidi, and Giovani Estrada. ≪A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling≫. In: May 2017. DOI: `10.1109/CCGRID.2017.15`.

[14] *Backpressure*. URL: `https://medium.com/@jayphelps/backpressure-explained-the-flow-of-data-through-software-2350b3e77ce7`.

[15] *Click Fraud*. URL: `https://en.wikipedia.org/wiki/Click_fraud`.

[16] *Click Fraud Application*. URL: `https://github.com/Nada-S/Clicks_fraud_detection_with_Kafka_and_Flink`.

[17] *Containers.* URL: https://www.docker.com/resources/what-container/.

[18] *Flink Stream Transformations.* URL: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/operators/overview/.

[19] Panagiotis Giannakopoulos and Euripides Petrakis. «Smilax: Statistical Machine Learning Autoscaler Agent for Apache FLINK». In: Apr. 2021, pp. 433–444. DOI: 10.1007/978-3-030-75075-6_35.

[20] *Google Cloud Platform Pricing.* URL: https://cloud.google.com/compute/all-pricing.

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

[22] *Horizontal scaling.* URL: https://aws.amazon.com/blogs/architecture/architecting-for-reliable-scalability/.

[23] *Java Management Extensions (JMX).* URL: https://www.oracle.com/technical-resources/articles/javase/jmx.html.

[24] Bibal Benifa Jv and Dejey Dharma. «RLPAS: Reinforcement Learning-based Proactive Auto-Scaler for Resource Provisioning in Cloud Environment». In: *Mobile Networks and Applications* 24 (Aug. 2019), pp. 1–16. DOI: 10.1007/s11036-018-0996-0.

[25] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. «Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows». In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 783–798. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/kalavri.

[26] Wybe J. C. Koper. «A Comparison of Auto-scaling Techniques for Distributed Stream Processing». 2022. URL: http://resolver.tudelft.nl/uuid:0ae72b82-b3af-4afb-a8d8-8040a226f045.

[27] *Kubernetes Components.* URL: https://kubernetes.io/docs/concepts/overview/components/.

[28] *Kubernetes Overview.* URL: https://kubernetes.io/docs/concepts/overview/.

[29] *Kubernetes Service.* URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[30] *Microservice Architecture.* URL: https://www.ionos.com/digitalguide/websites/web-development/microservice-architecture/.

[31] *Prometheus Data Model.* URL: https://prometheus.io/docs/concepts/data_model/.

[32] *Prometheus Metric Types.* URL: https://prometheus.io/docs/concepts/metric_types/.

[33] *Prometheus Overview.* URL: https://prometheus.io/docs/introduction/overview/.

[34] Víctor Rampérez, Javier Soriano, David Lizcano, and Juan A. Lara. "FLAS: A combination of proactive and reactive auto-scaling architecture for distributed services". In: *Future Generation Computer Systems* 118 (2021), pp. 56–72. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2020.12.025. URL: https://www.sciencedirect.com/science/article/pii/S0167739X20330879.

[35] *Reactive Mode.* URL: https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/elastic_scaling/.

[36] *Rescalable State in Apache Flink.* URL: https://flink.apache.org/features/2017/07/04/flink-rescalable-state.html.

[37] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. «Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning». In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD).* 2019, pp. 329–338. DOI: 10.1109/CLOUD.2019.00061.

[38] *Stream Processing.* URL: https://flink.apache.org/flink-architecture.html.

[39] *Timely Dataflow.* URL: https://docs.rs/timely/latest/timely/.

[40] Peter A. Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. «NEXMark – A Benchmark for Queries over Data Streams DRAFT». In: 2002.

[41] Balázs Varga, Márton Balassi, and Attila Kiss. «Towards autoscaling of Apache Flink jobs». In: *Acta Universitatis Sapientiae, Informatica* 13.1 (2021), pp. 39–59. DOI: doi:10.2478/ausi-2021-0003. URL: https://doi.org/10.2478/ausi-2021-0003.

[42] *Vertical Scaling.* URL: https://www.stormit.cloud/blog/scalability-in-cloud-computing-horizontal-vs-vertical-scaling/.

[43] *Ververica Autopilot.* URL: https://docs.ververica.com/user_guide/application_operations/autopilot.html#.

[44] *What are Microservices?* URL: https://aws.amazon.com/microservices/.