Technical University of Crete
School of Electrical & Computer Engineering

Optimizing Stream Processing Efficiency and Cost
TALOS
Task Level Autoscaler
for Apache Flink Platform

by

Ntouni Ourania

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

THESIS COMMITTEE
Professor Euripides G. M. Petrakis, Thesis Supervisor
Professor Vasileios Samoladas
Professor Nikolaos Giatrakos

Chania, March 2024

Abstract

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Big Data platforms, due to their dynamic nature, often face fluctuations in data streams workloads, leading to potential over-provisioning or under-provisioning because of static resource allocation. Most existing solutions solve the resource adaptation problem by scaling the entire Job. These solutions are sub-optimal since not all Tasks are equally stressed and need not be scaled. Therefore, we decided to develop an agent that manages the resources of a Flink job during runtime, at a Task level. This thesis presents TALOS, an innovative task autoscaler specifically designed for Apache Flink. TALOS dynamically changes the parallelism of tasks within Flink jobs in response to real-time workload fluctuations. TALOS is a threshold-based agent that utilizes a combination of metrics, such as Kafka consumer lag, throughput, backpressure, buffer metrics, and idleness to make scaling decisions. This algorithm targets not only in optimizing the performance of the Flink Job, but also in minimizing the cost in cloud environments, especially for long running applications. Notable is, that TALOS monitors each task separately, without taking into consideration output and input rate of upstream or downstream tasks, respectively. In this thesis, 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. Our system is compared against Flink Kubernetes Operator autoscaler, a threshold based algorithm, and both of the agents are tested in pretentious workloads.

Περίληψη

Το Apache Flink είναι μια πλατφόρμα και μηχανή κατανεμημένης επεξεργασίας για stateful υπολογισμούς σε αόριστες και περιορισμένες ροές δεδομένων. Οι πλατφόρμες Big Data , λόγω της δυναμικής φύσης τους, συνήθως αντιμετωπίζουν διακυμάνσεις στο φορτίο ροών δεδομένων, προκαλώντας over-provisiong ή under-provisioning πόρων λόγω της στατικής δέσμευσης πόρων. Οι κυρίως υπάρχουσες λύσεις, είναι κλιμάκωση σε επίπεδο job. Αυτού του είδους λύσεις, δεν αποτελούν την βέλτιστη προσέγγιση, σε περιπτώσεις όπου δεν ¨πιέζονται' όλα τα task το ίδιο. Στο πλαίσιο αυτό, αποφάσισαμε να αναπτύξουμε έναν πράκτορα που διαχειρίζεται τους πόρους ενός Flink job κατά τη διάρκεια εκτέλεσης, σε επίπεδο διεργασίας. Η παρούσα διατριβή παρουσιάζει τον TALOS, έναν πρωτότυπο Autoscaler σε επίπεδο διεργασίας που σχεδιάστηκε ειδικά για το Apache Flink. Το TALOS αλλάζει δυναμικά τον παραλληλισμό των tasks σε Flink jobs ανάλογα με τις διακυμάνσεις του φορτίου δεδομένων κατά τον χρόνο εκτέλεσης. Το TALOS είναι ένας πράκτορας που βασίζεται σε τιμές κατώφλιου (threshold based) και χρησιμοποιεί έναν συνδυασμό μετρικών, όπως το Kafka consumer lag,throughput, backpressure, buffer metrics, και idleness για να λάβει αποφάσεις για τον καινούριο παραλληλισμό των διεργασιών. Αυτός ο αλγόριθμος στοχεύει στην βελτιστοποίηση της απόδοσης του Flink Job, αλλά και στην ελαχιστοποίηση του κόστους σε περιβάλλοντα cloud, ειδικά για εφαρμογές μεγάλης διάρκειας. Αξίζει να σημειωθεί ότι το TALOS παρακολουθεί κάθε task ξεχωριστά, χωρίς να λάβει υπόψη τη ταχύτητα εξόδου και είσοδου των upstream ή downstream tasks , αντίστοιχα. Σε αυτήν τη διατριβή, δείχνουμε ότι το μοντέλο μας επιτυγχάνει να διατηρήσει επιτυχώς την απόδοση της εφαρμογής, ενώ ελαχιστοποιεί το κόστος, παρέχοντας μία βέλτιστη αναλογία απόδοσης-κόστους σε σχέση με την υπάρχουσα εργασία στην κλιμάκωση του Flink job. Η σύγκριση υλοποιείται με το Flink Kubernetes Operator autoscaler, έναν threshold-based αλγόριθμο, και οι δύο πράκτορες ελέγχονται σε απαιτητικά φορτία ροής δεδομένων.

Acknowledgements

The past five years at this University have been both demanding and immensely fulfilling. At the outset, I must extend my deepest appreciation to my thesis advisor, Euripides Petrakis, whose unwavering support, insightful guidance, and invaluable ideas have been crucial to the completion of this thesis. I am also grateful to my friends Nikos P., Evangelia T., Dimitris Mp., Nikos T. and Constantinos M., for their pivotal role in my life, in all aspects. Furthermore, I owe a tremendous debt of gratitude to my family, for the unlimited support all these years.

Contents

1 Introduction

In today's data-driven landscape, big data platforms are the impulse behind the operations of many enterprises. These platforms empower organizations to manage the massive volumes of data generated daily for absorbing information, performing analytics, and making crucial decisions. Within those sectors exist numerous big data platforms, such as Apache Spark and Apache Flink. Among them, Apache Flink has attracted our interest for the reason that it is based on a data streaming model and it is optimized in real time processing.

1.1 Problem Definition

As the considerable volume, speed, and diversity of data, continue to increase, there is need for dynamically scalable and effective processing solutions. Big data platforms like Apache Flink are driven to manage the complexities of real-time data streams. Nevertheless, it also plays a pivotal role to dynamically adapt to fluctuations of the workload and also to demand of the existing resource. It is very important to manage resource effectively and ensure optimal performance and cost-efficiency, especially in the presence of oscillating workloads characterized by rapid fluctuations in data send rate and processing requirements.

Given the prolonged duration of streaming jobs, fluctuations in workload are anticipated throughout an application's lifespan. Nevertheless, statically provisioning cloud resources by setting the job's parallelism at launch-time can result in inefficiencies and unnecessary expenses.More precisely, the fixed parallelism decision,and fluctuating workloads, can lead to either under-provisioning or over-provisioning. *Under-provisioning* leads to performance degradation when workload increases, while *over-provisioning* incurs unnecessary infrastructure costs during periods of lower demand.

The main problem is to find the optimal parallelism for a Flink job that maximizes the performance, the reliability, and the resource efficiency, while minimizing the costs, the latency, and the downtime. In our case "update", refers to the new parallelism. The problem is challenging because of the dynamic and heterogeneous nature of the data and the resources, the trade-offs and constraints involved in scaling, and the complexity and diversity of the Flink applications. The problem is important because scaling affects the quality of service, the user satisfaction, and the operational and financial outcomes of the Flink applications.

In Apache Flink deployments, the additional challenge lies in reconciling the competing demands of task scaling and job scaling. Task scaling offers the advantage of fine-grained resource management, enabling optimization at the level of individual computational tasks to enhance performance. Lately, a few authors have shown interest in creating task scaling agents, such as DS2 [1], QAAS [2], which will be discussed in detail, in *Related Work* chapter. On the other hand, job scaling simplifies resource allocation by optimizing resources at the job level, ensuring efficient utilization across multiple tasks within a job. Below, we will present a proposed solution to address this challenge. HYAS [3] is a worth mentioned job autoscaler, which will be mentioned below, as well. However, scaling actions results

downtime. Downtime refers to the period during which a Flink application is not processing data because it is being updated. In our case "update", refers to redeploying the job with different parallelism. This downtime causes records to accumulate in queues, underscoring the inefficiency in the performance. Nevertheless, TALOS is not focalized in minimizing downtime, but combines cost and performance optimization. Optimizing cost means ensuring that these resources are utilized as efficiently as possible, reducing waste and avoiding unnecessary expenses without compromising performance. Below, we will present a proposed solution to address this challenge.

1.2 Proposed Solution

In this thesis we propose we present TALOS, an agent that suggests flexibility in task scaling, emphasizing the algorithm's ability to dynamically adjust resources allocated to individual tasks based on workload variations. The developed algorithm is thresehold based and prioritizes detecting changes in workload, by giving emphasis in queued records. TALOS is an agent, which optimizes the performance of the job, without maximazing cost.

There are two primary approaches to autoscaling in cloud environments: reactive scaling and proactive scaling. Reactive scaling adjusts resources based on real-time workload changes, while proactive scaling uses machine learning to predict resource needs. TALOS adopts a reactive approach, leveraging real-time monitoring metrics to predict scale-up or scale-down decisions without requiring machine learning model training and taking into consideration the change rate of the workload.

In TALOS, each task autonomously determines how to scale based on its own data processing needs, without being influenced by the data processing outcomes of preceding tasks in the data flow, known as upstream operators. This independance between tasks,allowing TALOS to effectively handle cases of split data, i.e when a task send different portions of data between following tasks, known as "downstream" operators. Moreover, ensures that TALOS can react promptly and accurately to workload changes, regardless of how data is partitioned or distributed across different operators within the Apache Flink application. This feature enhances TALOS's ability to adapt dynamically to varying workload conditions and optimize resource allocation accordingly.

Our inspiration for developing our approach at the task level stemmed from the innovative concepts introduced by HYAS. By adapting and refining these ideas, we aimed to create a task scaler that could effectively manage resource allocation within Apache Flink applications. Furthermore, the pioneering efforts of Kalavri's work [1] provided valuable insights and inspiration, serving as a catalyst for the development of the initial task scaler. Drawing upon these influences, we sought to enhance scalability, efficiency, and adaptability in task-level resource management within Apache Flink deployments.

1.3 Thesis Structure

The remainder thesis structure is as follows:

- Chapter 3 provides some necessary background technologies.
- Chapter 4 analyzes the development of our scaling model
- Chapter 5 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. Also, represents the 2 experiments, that we exploit for the algorithms comparison.
- Chapter 6 and 7 generalize the conclusions of this work and provide input on possible future work respectively.

2 Related Work

In this section, we review some of the existing works on autoscaling algorithms for stream processing applications, and compare them with our work. Autoscaling is the ability to adjust the resources and the parallelism of a stream processing application according to the workload and the performance requirements. However, autoscaling poses significant challenges in terms of state management, latency, accuracy, and cost.

- **HYAS** [3] is a hybrid job autoscaling method for stateful distributed stream processing systems, such as Apache Flink. It monitors and models the responsiveness of the Flink operators based on their idleness, backpressure, and input record lag. It uses rule and threshold policies to adjust the parallelism of the operators according to the workload and the performance requirements. HYAS aims to minimize resources and achieve faster response time. HYAS is based on the research paper by Varga et al [4]. In contrast to our work, we developed a task scaler based on HYAS idea.

- **DS2** [1] is an automatic scaling controller for distributed streaming dataflows, such as those implemented by Apache Flink. DS2 uses a general performance model of streaming dataflows and lightweight instrumentation to estimate the true processing time and output rates of individual operators. True processing time refers to the actual time it takes for a system to process a given set of data inputs from the moment they are received until the final output is produced. Based on these estimates, DS2 decides when and how to scale the tasks of the pipeline to achieve optimal throughput.DS2 optimizes dataflow scaling within a maximum of three steps, preventing excessive fluctuations and reducing the time needed to stabilize. The algorithm has been integrated into the Apache Flink Kubernetes operator, and in the experiments chapter, we will conduct a comparative analysis against our own algorithm.

- **QAAS** [2] is a quick accurate task auto-scaling method for streaming processing jobs in Apache Flink. QAAS uses a nonlinear model to represent the performance of operators, which are the basic units of processing logic in Flink. QAAS divides the operator performance into three stages: non-competition, non-full competition, and full competition. Based on these stages, QAAS can accurately predict the CPU utilization and throughput of operators given their parallelism. QAAS also collects metrics from running Flink jobs and estimates the optimal parallelism for each operator to achieve the target utilization and avoid backpressure. QAAS is a proactive auto-scaler, meaning that it anticipates the load changes and adjusts the resources accordingly, rather than reacting to the load changes after they occur.

- **Meces** [5] is a method for rescaling stateful distributed stream processing systems, such as Apache Flink, with low latency and high efficiency. It prioritizes the state migration of the most critical operators, which are the ones that have the largest backlog and the highest processing rate. It uses a graph partitioning algorithm to minimize

the communication cost and the state migration time. State migration refers to the process of transferring the operational state (data and configuration) of these critical tasks to new or different resources during scaling operations Meces aims to improve the scalability, the reliability, and the resource utilization of stream processing applications. In comparison with our work, Meces focalizes in state migration, decreasing latency peaks after scaling and it is ideal for stateful applications.

- **Smilax** [6] is a statistical machine learning agent for Apache Flink applications. It predicts the future workload and adjusts the number of workers to match the needs of the application. It uses online training to build a model that maps the performance of the application to the minimum number of servers. It aims to optimize the resource utilization, the latency, and the cost of Flink applications.
  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.

- **Ververica Platform Autopilot** [7] is a feature that monitors and models the responsiveness of the Flink operators based on their idleness, backpressure, and input record lag. It uses rule and threshold policies to adjust the parallelism of the operators according to the workload and the performance requirements. Ververica Platform Autopilot aims to optimize the resource utilization, the latency, and the cost of Flink applications. Unlike our implementation, Ververica's Autopilot scales the entire pipeline and employs a distinct algorithm.

Our work differs from these works in several aspects, such as the scaling objectives, the scaling granularity, the scaling frequency, and the scaling evaluation. Furthermore, the main difference is the target, since TALOS tries to optimize both performance and cost (minimizing resources). Yet, we will concentrate in the DS2 method and we discuss the differences in the following sections.

Autoscaler	Task Scaling	Job Scaling	Re-active	Pro-active
TALOS	✓		✓	
HYAS		✓	✓	✓
Smilax		✓		✓
DS2	✓		✓	
Meces	✓		✓	
QAAS	✓			✓
Ververica		✓	✓	

Table 1: Summarization of all methods

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.[8] 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 [9] 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 [10] propose RL solutions for controlling the horizontal and vertical elasticity of container-based applications in order to cope with varying workloads.

3 Background

This chapter will give a comprehensive introduction to the technologies that are used in this project. These technologies are essential for the development of this thesis.

3.1 Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and at any scale.[11]

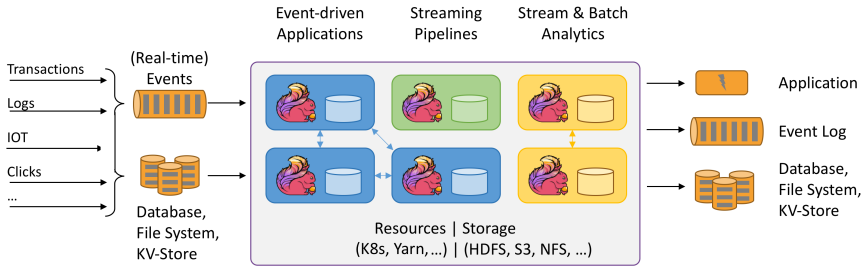

Figure 1: Apache Flink

Stream Processing

The DataStream API in Apache Flink supports multiple runtime execution modes, providing flexibility based on the characteristics and requirements of the use case. Any kind of data is produced as a stream of events. Data can be processed as *unbounded* or *bounded* streams.

Unbounded streams have a start but no defined end. They do not terminate and provide data as it is generated. Unbounded streams must be continuously processed, i.e., events must be promptly handled after they have been ingested. It is not possible to wait for all input data to arrive because the input is unbounded and will not be complete at any point in time. Processing unbounded data often requires that events are ingested in a specific order, such as the order in which events occurred, to be able to reason about result completeness.

Bounded streams have a defined start and end. Bounded streams can be processed by ingesting all data before performing any computations. Ordered ingestion is not required to process bounded streams because a bounded data set can always be sorted. Processing of bounded streams is also known as batch processing.

The discrimination of bounded and unbounded data is depicted in Figure 2.

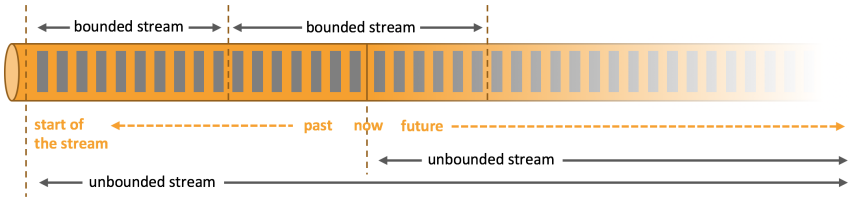

Figure 2: Bounded and Unbounded Stream

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. **Stateful processing** allows Flink applications to maintain and leverage state information across multiple events. This capability proves essential in applications where understanding the context of events and tracking their evolution over time is necessary. Thus, Flink needs to be aware of the state in order to make it fault tolerant using *checkpoints* and *savepoints*.

Checkpoints

Checkpoint is an automatic, asynchronous snapshot of the state of an application and the position in a source stream. In the case of a failure, a Flink program with checkpointing enabled will, upon recovery, resume processing from the last completed checkpoint, ensuring that Flink maintains exactly-once state semantics within an application.

A core element in Flink's distributed snapshotting are the stream **barriers**. These **barriers** are injected into the data stream and flow with the records as part of the data stream. Barriers never overtake records, they flow strictly in line. A barrier separates the records in the data stream into the set of records that goes into the current snapshot, and the records that go into the next snapshot. Each barrier carries the ID of the snapshot whose records it pushed in front of it. Barriers do not interrupt the flow of the stream and are hence very lightweight. Multiple barriers from different snapshots can be in the stream at the same time, which means that various snapshots may happen concurrently.

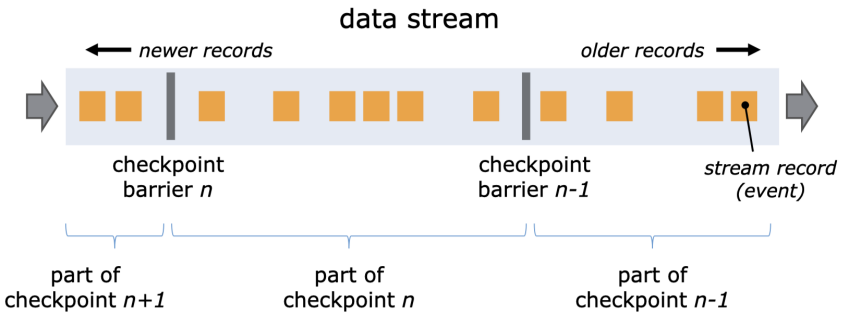

Figure 3: Checkpointing

Savepoints

All programs that use checkpointing can resume execution from a savepoint. **Savepoints** allow both updating your programs and your Flink cluster without losing any state. **Savepoints** are manually triggered checkpoints, which take a snapshot of the program and write it out to a state backend. They rely on the regular checkpointing mechanism for this.

Savepoints are similar to checkpoints except that they are triggered by the user and don't automatically expire when newer checkpoints are completed.

Timely Stream Processing

Timely stream processing is an extension of stateful stream processing in which time plays some role in the computation. Among other things, this is the case when you do time series analysis, when doing aggregations based on certain time periods (typically called windows), or when you do event processing where the time when an event occurred is important.

When referring to time in a streaming program (for example to define windows), one can refer to different notions of time:

Processing time refers to the machine time of the machine that is executing the respective operation. In processing time mode within a streaming program, time-based operations like time windows rely on the system clock of the machines executing the code of the operators. For instance, an hourly processing time window collects data between the times when the system clock indicates the full hour. So, if the application starts at 9:15 am, the first hourly window covers data from 9:15 am to 10:15 am, and subsequent windows follow suit. Processing time offers high performance and low latency but lacks determinism in distributed and asynchronous environments, due to factors like varying record arrival speeds and system outages.

Event time is the time that each individual event occurred on its producing device, and is associated with each record before the latter enters Flink. When using event time, time progression depends on the events' timestamps. When events arrive in order or all events have arrived, this timing mechanism provides consistent and deterministic results. However, when out-of-order events arrive, latency is induced. As it is only possible to wait for a finite period of time, this places a limit on how deterministic event time applications can be.

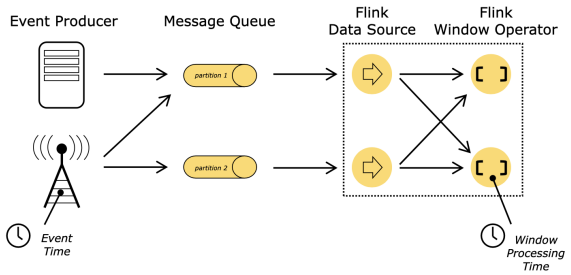

Figure 4: Event Time in Flink Job

Flink Architecture

Flink is a distributed system and requires effective allocation and management of compute resources in order to execute streaming applications. It integrates with all common cluster resource managers such as Hadoop YARN and Kubernetes, but can also be set up to run as a standalone cluster or even as a library.

The Flink runtime consists of two types of processes: a JobManager and one or more TaskManagers.

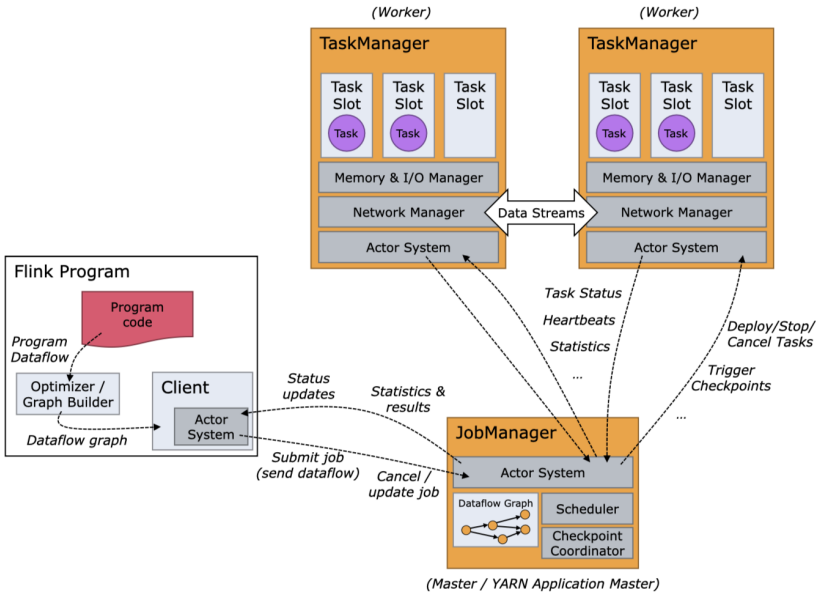

Figure 5: Flink Architecture

The following information provides more insights into the components of the architecture.

Job Manager

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, among others.

JobMaster is responsible for managing the execution of a single JobGraph. Multiple jobs can run simultaneously in a Flink cluster, each having its own JobMaster.

Dispatcher provides a REST interface to submit Flink applications for execution and starts a new JobMaster for each submitted job. It also runs the Flink WebUI to provide information about job executions.

ResourceManager is responsible for resource de-/allocation and provisioning in a Flink cluster — it manages task slots, which are the unit of resource scheduling in a Flink cluster (see TaskManagers). Flink implements multiple ResourceManagers for different environments and resource providers such as *YARN*, *Kubernetes* and *standalone deployments*. In a standalone setup, the ResourceManager can only distribute the slots of available TaskManagers and cannot start new TaskManagers on its own.

Figure 6: Job Manager

Task Managers

The TaskManagers (workers) execute the tasks of a dataflow, and buffer and exchange the data streams. Each TaskManager is a JVM process, and may execute one or more subtasks in separate threads. There must always be at least one TaskManager to run a job. Furthermore, each TaskManager contains at least one task slot which is the smallest unit of execution in a cluster.

Tasks and Operator chains

Tasks are components of a job in execution. Every task is carried out by a specific number of threads, equivalent to its corresponding level of parallelism. Each task includes at least one operator. If there are multiple operators within the same task, they are considered chained. The practice of chaining operators into tasks is a valuable optimization, as it minimizes the overhead associated with thread-to-thread handover and buffering. This, in turn, enhances overall throughput while concurrently reducing latency. Flink has built in mechanism, if chaining is not disabled, to optimize operator chaining. Important to refer is that a chain can be created only if no network action (i.e. keyBy(), shuffle(), rebalance(),redistributing data across different task instances) has occurred between the chained operators. Last but not least, each task is the place where each parallel instance of an operator is executed. The sample dataflow in the figure 7 below is executed with five subtasks, and hence with five parallel threads.

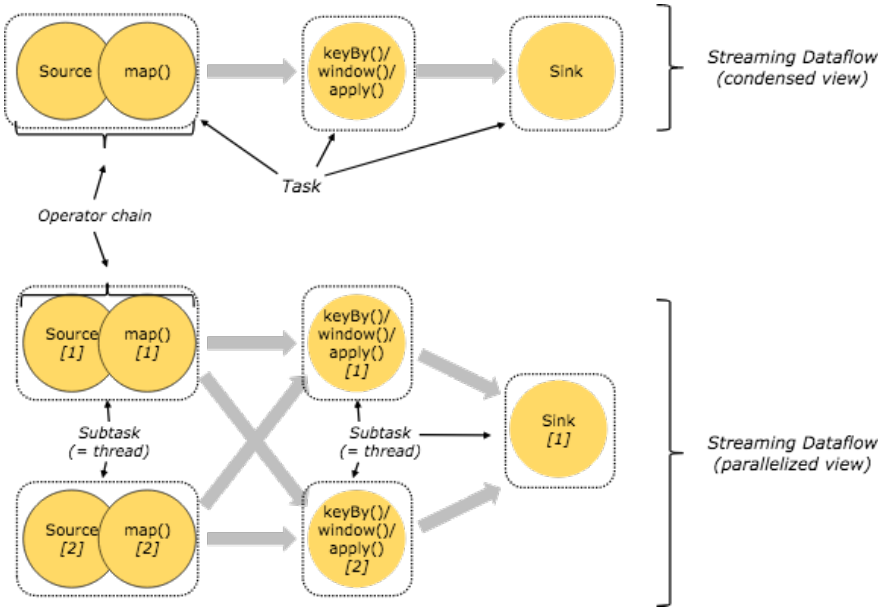

Figure 7: Flink dataflow example

Task Slots and Resources

Each TaskManager is a JVM process and may execute one or more subtasks in separate threads. To control how many tasks a TaskManager accepts, it has so called task slots (at least one). In other words, the number of task slots can define the maximum parallelism, that a TaskManager supports. By default, Flink allows subtasks to share slots even if they are subtasks of different tasks, so long as they are from the same job. The result is that one slot may hold an entire pipeline of the job. Slot sharing has a main benefit, better resource utilization. Without slot sharing, the non-intensive source/map() subtasks would block as many resources as the resource intensive window subtasks.
Each task slot represents a fixed subset of resources of the TaskManager. Slotting the resources means that a subtask will not compete with subtasks from other jobs for managed memory, but instead has a certain amount of reserved managed memory. Note that no CPU isolation happens here; currently slots only separate the managed memory of tasks.
By adjusting the number of task slots, users can define how subtasks are isolated from each other. Having one slot per TaskManager means that each task group runs in a separate JVM (which can be started in a separate container, for example). Having multiple slots means more subtasks share the same JVM. Tasks in the same JVM share TCP connections (via multiplexing) and heartbeat messages. They may also share data sets and data structures, thus reducing the per-task overhead.

In the following example, there are two TaskManagers, each equipped with three slots. A TaskManager with three slots, will dedicate 1/3 of its managed memory to each slot. However, as mentioned earlier, threads are contending for CPU. Furthermore, it is notable that only parallel instances of different tasks are located in the same slot.

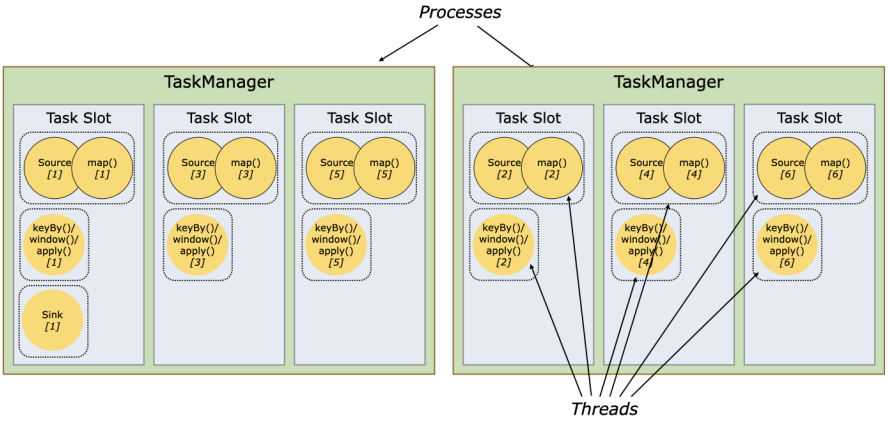

Figure 8: Example

Deployment

Apache Flink presents diverse deployment options tailored to various infrastructure and scalability requirements. **Kubernetes (K8s)** stands out as a popular choice, allowing users to efficiently orchestrate and manage Flink clusters within containerized environments. The Kubernetes deployment can take a native approach, integrating directly with Flink, or leverage the Flink Kubernetes Operator, a topic explored in detail in the subsequent subsection.

Another deployment avenue involves **Apache Hadoop YARN**, providing compatibility with Hadoop-based environments. Integration with YARN facilitates resource management and multi-tenancy, promoting seamless coexistence with other components in the Hadoop ecosystem.

Additionally, Flink supports **standalone deployments**, offering manual configuration and cluster management for straightforward setups or testing scenarios, affording users full control over Flink cluster configurations. In every deployment scenario, Flink ensures accessibility through a RESTful API and a Web UI, enabling direct interaction with Flink applications.

3.2 Flink Kubernetes Operator

The Flink Kubernetes Operator is a tool that extends the Kubernetes API with the ability to manage and operate Flink deployments on Kubernetes. It allows users to deploy and monitor Flink applications and sessions, upgrade, suspend and delete them, and integrate them with logging and metrics. Also, Apache Flink K8s Operator has an autoscaling mechanism. [12]

The installation of the operator is facilitated through Helm directly onto the Kubernetes cluster. Following installation, users generate YAML files to define Flink deployments. Within these files, configurations not only allow users to control the specifications of TaskManagers and JobManagers but also provide the flexibility to designate nodes based on specific requirements, i.e allocatic specific portion of resources for Task Managers and Job manager respectively. Additionally, users specify essential details such as the desired JAR file containing the Flink application, as well as preferences for parallelism and checkpointing behavior.

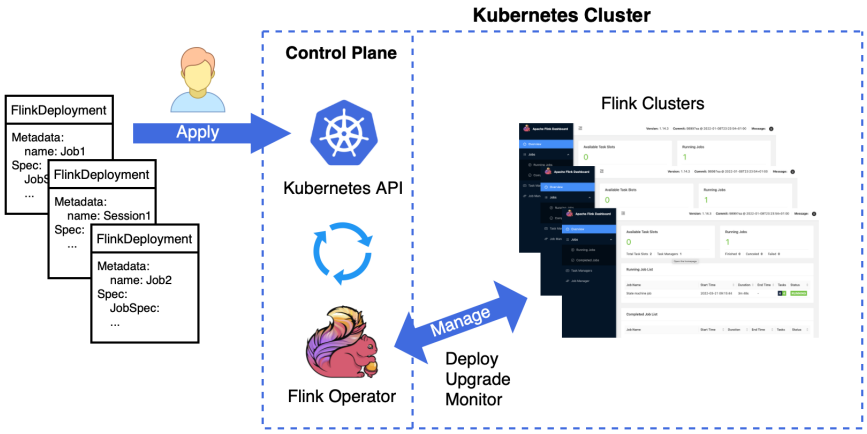

Figure 9: Flink K8s Operator and the Flink cluster

Autoscaling Mechanism

The built-in autoscaling mechanism in Apache Flink Kubernetes Operator is a feature that allows you to automatically adjust the parallelism of your Flink job vertices based on the metrics collected from the running Flink tasks. The autoscaling mechanism aims to eliminate backpressure and achieve the target utilization, i.e the busy percentage of an operator that is set by the user, and catch-up duration that you set for your Flink job. The autoscaling mechanism works by querying the Flink metric system for the backlog, incoming data rate, processing rate, and busy time of each task. It then computes the required processing capacity for each job vertex and applies the parallelism overrides through the Flink REST API. The autoscaling mechanism is based on the research paper by Kalavri [1]

3.3 Apache Kafka

Kafka is used to build real-time streaming data pipelines and real-time streaming applications. A data pipeline reliably processes and moves data from one system to another, and a streaming application is an application that consumes streams of data. [13]

Kafka follows the publish-subscribe model, where producers publish messages to topics, and consumers can subscribe to these topics in order to receive the sent messages. It also is a pull-based system, meaning that consumers can choose when they want to receive messages by sending a request to the broker. Push-based architectures would deliver the messages to consumers as soon as they are available, and then notify them for this action. Its distributed architecture allows for fault-tolerant, high-throughput and low latency messaging abilities. This design enables Kafka to be incorporated to applications that require highly scalable messaging services. Kafka is consisted of servers (Brokers) and clients (Producers and Consumers). In order for all parts of the system to communicate, a binary protocol over TCP is used.

Brokers are key components of the storage and distribution of data. A broker can be thought of as a single instance of a Kafka server, responsible for managing one or more Kafka topics and their associated partitions. A Kafka cluster can be deployed with either one or more brokers, with the latter providing fault tolerance and better scalability.

Records are the basic units of data exchange in Kafka's environment. They are also referred to as events or messages. Each record contains a key, value, timestamp and optional metadata headers.

Topics, much like folders in a file system, are logical channels to which messages are published by producers, and from which they are consumed by consumers. Topics serve as a mean for categorizing data in a Kafka broker. Each topic is identified by its name. Contrast to many other messaging systems, Kafka does not delete messages upon consumption. Instead, time or size based retention policies can be configured for each topic independently. For instance, a topic whose messages are consumed slowly, can have a much greater retention period than one that has its messages consumed with a higher rate. Messages that have not been deleted can be re-consumed as many times as necessary. Topics can be replicated, even across different regions, so that there are always multiple brokers that have a copy of the data in case of an outage or broker maintenance etc.

Partitions are a way of breaking down a topic into smaller, independently manageable segments. Each partition holds part of the total data published on the topic it belongs. Every message that is published to a topic is placed on a single partition either based on the key of the record, or a partitioning policy (e.g. round-robin manner). Messages within different partitions can be processed independently by Kafka consumers and can be spread across multiple Kafka broker nodes. This parallelism enhances the overall throughput, scalability and fault tolerance of Kafka. The messages residing in a partition are guaranteed to be in the order they arrived from their respective producers.

Consumers are client applications that consume (read) records from a Kafka broker. They are fully decoupled and agnostic from producers, which is a crucial design element in order to achieve the scalability that Kafka is known for. Consumers can be grouped into consumer groups and leverage the topic partitioning, by assigning different partitions to different consumers within the group. Each consumer within the group reads from its assigned partitions independently, allowing for concurrent message processing.

Producers are client applications that publish (write) records to a Kafka broker. They are not required to await the consumption of messages by consumers before dispatching subsequent ones. Furthermore, producers offer fine-grained control over message delivery reliability through acknowledgment settings (acks) and retry configurations. By proper configuration, producers can choose the level of acknowledgment they require, ranging from no acknowledgment, to acknowledgment from all replicas. This mechanism allows for adaptation for different scenarios, balancing performance and reliability.

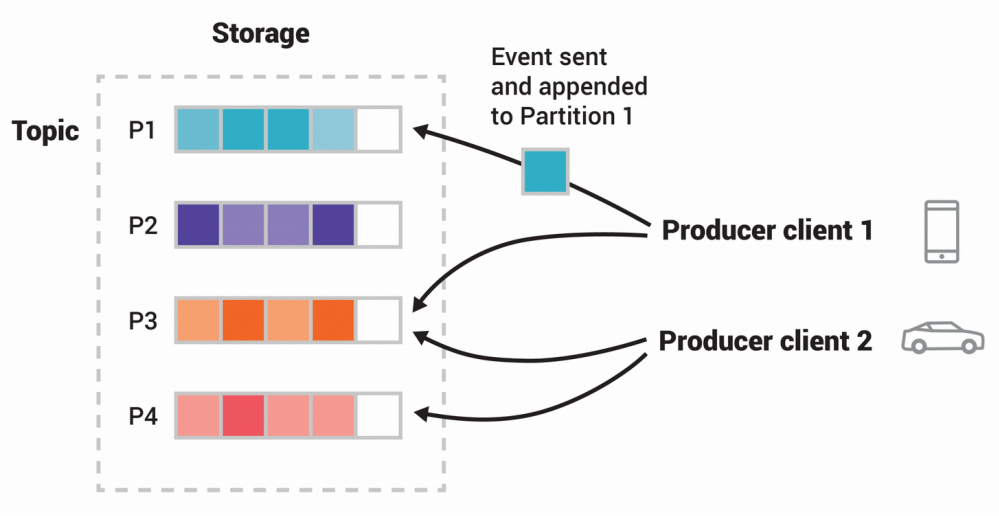

Figure 10: Distribution of messages from different producers

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 [14]

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 ap- plication 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 di- mensions. Besides stored time series, Prometheus may generate temporary derived time series as the result of queries. 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.

- **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 obser- vations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

3.5 Microservices

Microservices are an architectural and organizational approach to software develop- ment where software is composed of small independent services that communicate over well-defined APIs. Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.[15]

A well defined aspect of microservices is autonomy and specialization. Each com- ponent 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.

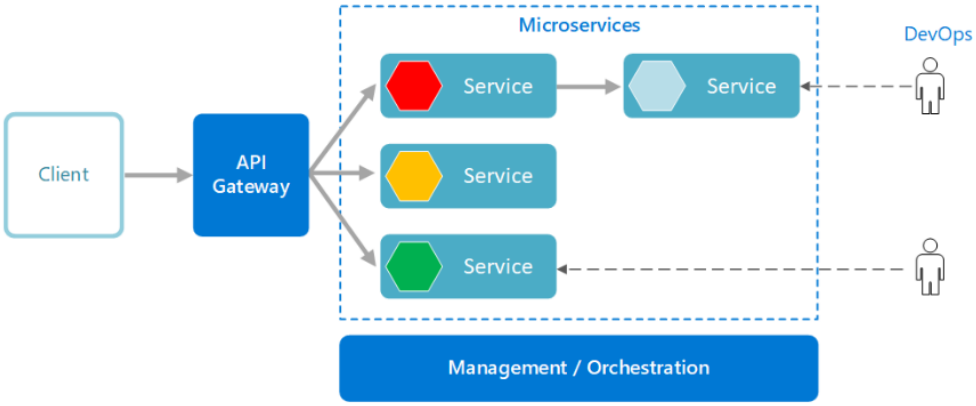

Figure 11: Microservice Architecture

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. 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) . 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.

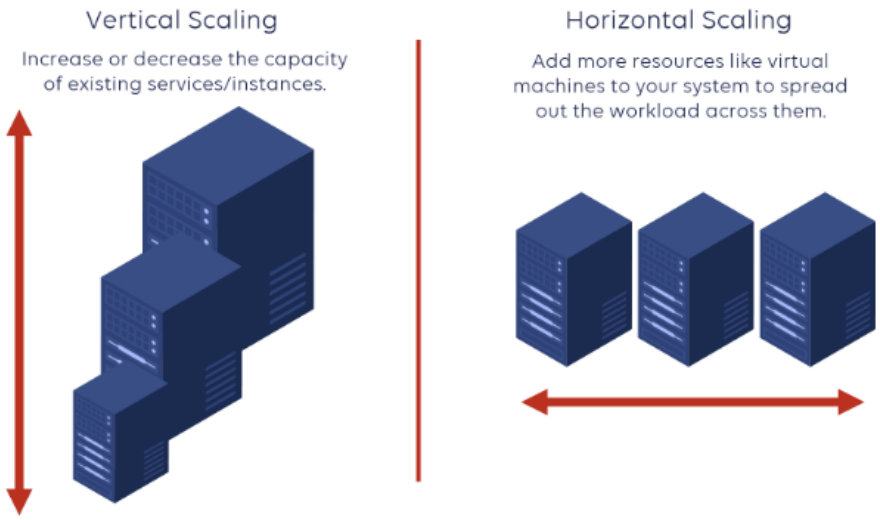

Figure 12: Horizontal and Vertical Scaling

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:

- 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, 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.

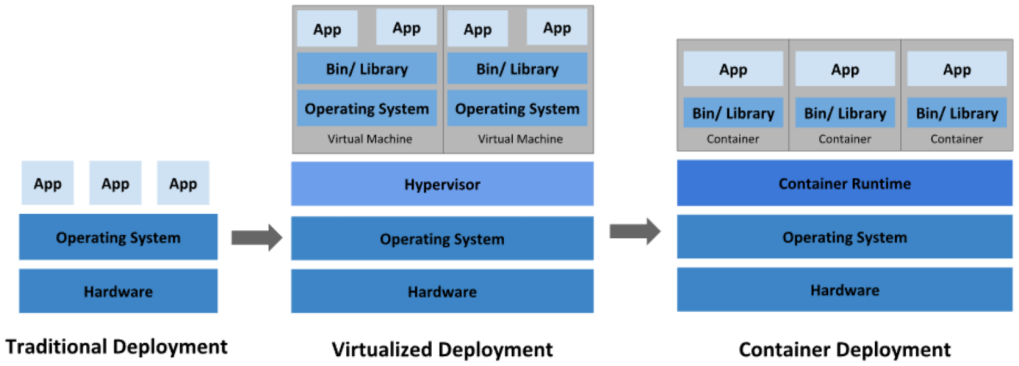

Figure 13: Traditional vs Virtualized vs Container deployment

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 ma- chines, 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

3.6 Kubernetes

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.[16]

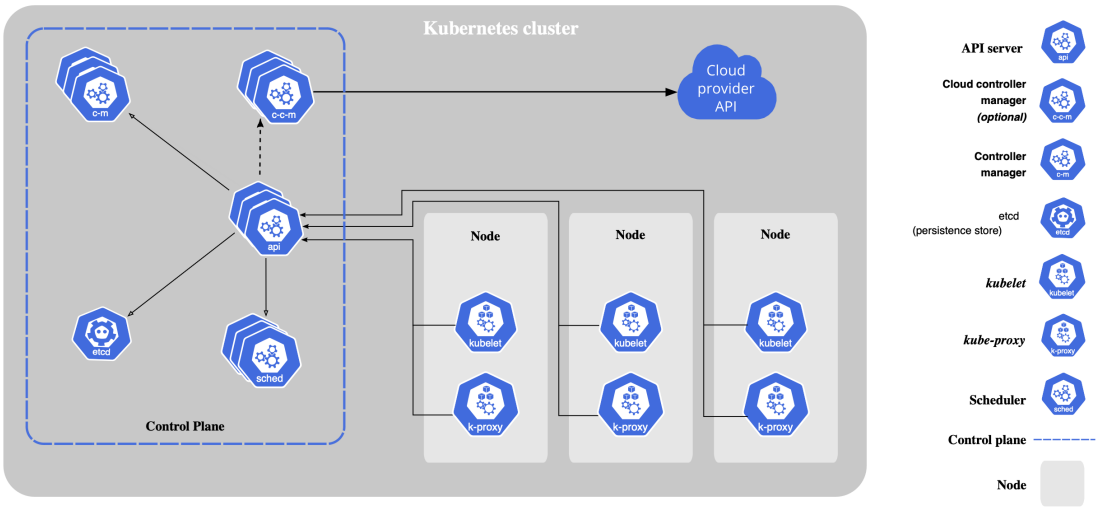

Figure 14: Kubernetes Architecture

Kubernetes architecture is built on a master-worker model, featuring essential components like the API server, etcd for configuration storage, and controllers for maintaining the cluster's desired state. Nodes, or worker nodes, execute the workload, managed by components such as Kubelet, Container Runtime, and Kube Proxy. Pods, the smallest deployable units, encapsulate one or more containers, fostering encapsulation and resource sharing. Kubernetes' controllers and services contribute to automated scaling, self-healing, and load balancing, making it the go-to solution for deploying resilient and scalable cloud-native applications. Each of the components will be examined in detail in the subsequent analysis.

Components

Node

Kubernetes runs your workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run *Pods*. Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have only one node. The components on a node include the *kubelet*, a container runtime, and the *kube-proxy*.

Kubelet is the primary "node agent" that runs on each node. It can register the node with the apiserver using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider.

Kube-proxy runs on each node. This reflects services as defined in the Kubernetes API on each node and can do simple TCP, UDP, and SCTP stream forwarding or round robin TCP, UDP, and SCTP forwarding across a set of backends. Service cluster IPs and ports are currently found through Docker-links-compatible environment variables specifying ports opened by the service proxy. There is an optional addon that provides cluster DNS for these cluster IPs. The user must create a service with the apiserver API to configure the proxy.

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A Pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

Control plane

Control plane components manage the worker nodes and the Pods in the cluster. Furthermore, they are responsible for responding to cluster events. For example, a cluster event may be the loss of a pod for a specific deployment. In this case, the replicas field number is not satisfied and action is taken by K8s control plane. In production environ- ments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

3.7 Google Cloud Platform

The Google Cloud Platform (GCP) stands out as an extensive cloud computing solution provided by Google, significantly influencing the landscape of cloud technology and services. Offering a diverse array of cloud-based solutions, GCP encompasses computing, storage, databases, machine learning, and data analytics, among other services. Renowned for its globally distributed large-scale data centers, GCP delivers high-performance solutions. Notable services within GCP include Google Compute Engine, providing virtual machines, Google Kubernetes Engine for efficient container orchestration, BigQuery specializing in data analytics, and TensorFlow, a powerful tool for machine learning applications

4 Task Autoscaler for Apache Flink

TALOS, or TaskFlexScaler, represents a novel model designed to efficiently scale the tasks within a Flink job. Operating on a threshold-based approach, this algorithm harnesses multiple metrics to make informed decisions regarding the parallelism of tasks. By incorporating a diverse range of metrics, TALOS ensures a comprehensive evaluation process, thereby enhancing its scalability and adaptability within Flink environments.

We have chosen a combination of metrics meticulously selected to capture the nuances of the pipeline's requirements. By "needs," we refer to the dynamic conditions of the pipeline, determining whether increased parallelism is necessary during periods of strain or decreased parallelism is warranted when the pipeline is critically idle. Additionally, tailoring different parallelism levels for each task holds potential for improved outcomes, as certain tasks may not require extensive resource allocation. This deliberate approach ensures a reflective response to the varying demands of the pipeline, optimizing resource utilization and enhancing overall performance.

4.1 Metric Analysis

This subsection provides an overview of the metrics analysis conducted as part of our Task Auto-scaler for Apache Flink (TALOS) algorithm. Metrics are crucial components guiding TALOS's decision-making process, facilitating dynamic adjustments to task parallelism within Flink jobs. By examining various metrics, we gain insights into real-time performance and resource needs, illuminating the relationship between workload dynamics and parallelism adjustments.

Kafka Consumer Lag

Kafka consumer lag, which measures the delay between a Kafka producer and consumer, is a key Kafka performance indicator.More specifically, Kafka consumer lag is the difference between the last offset stored by the broker and the last committed (i.e the most recent position in a Kafka partition where a consumer group has acknowledged processing messages up to) offset for that partition.

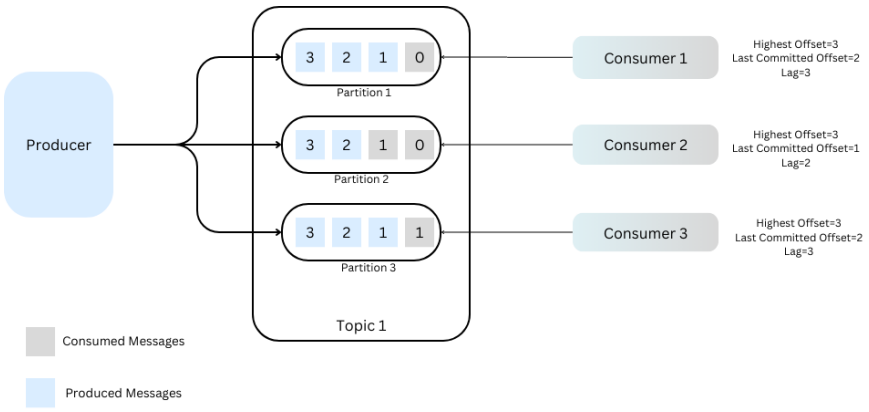

Figure 15: Representation of Kafka Lag

In the context of our agent's scaling policy, Kafka lag refers to the discrepancy between the rate at which records are produced and consumed within a Kafka topic. This lag is crucial to monitor as it reflects the difference between the number of records consumed by a consumer (known as the committed offset), in our case the consumer is the Source of the flink job, and the total number of records produced (the latest offset)[17]. If the production rate surpasses consumption, consumers will exhibit lag, which can impact the real-time processing of data. To manage this, Flink provides an upper bound for the total lag using the records lag max metric, indicating how far behind a consumer is from the head of the message *queue*. This metric offers insights into the maximum lag in terms of the maximum number of unprocessed records across all partitions being consumed, helping to ensure efficient processing and timely data consumption. The performance of the Kafka Source,i.e the consumer, is assessed by monitoring this metric, which provides insights into the lag between record production and consumption within Kafka.

Throughput

Throughput stands as a pivotal metric in gauging the effectiveness of any Flink job or its individual tasks, revealing the rate at which records can be processed within a given time-frame. TALOS is interested in monitoring the throughput of individual tasks. To calculate the throughput of a specific **intermediate** task (i.e all tasks of the pipeline except for the source, because it communicates with external systems, Kafka in our case), a combination of two metrics, namely **recordsInPerSec** and **busyTimeMs**, is employed. **recordsInPerSec** metric represents the number of records this operator/task receives per second [11] and **busyTimeMs** metric quantifies the duration, in milliseconds, a component spends on active computation tasks [11]. Introducing a very useful metric, that DS2 [1] used it too, that signifies throughput per unit of busy time, $\frac{recordsInPerSec}{BusyTimems}$, provides a comprehensive perspective on the task's efficiency and speed. This metric essentially denotes the average number of records a task can handle within a single millisecond of active processing time, encapsulating both the pace and effectiveness of the Flink job. A heightened value of this

metric suggests that the task efficiently processes a substantial volume of data within minimal busy time, indicative of superior performance. Conversely, a diminished metric value implies that the task handles a smaller data volume over an extended busy time, indicating suboptimal performance. Furthermore, it's notable that the throughput of **source** tasks is directly correlated with the rate of consumption from Kafka, reflecting the efficiency of data ingestion and processing from the Kafka source.

The theory suggests that the speed at which data moves through a pipeline is largely determined by the efficiency of the individual tasks. A satisfying instance is a relay race: if the runner at the end is slow, it doesn't matter how fast the earlier runners are—they all have to wait. Similarly, in data processing pipelines, if the tasks that come later can't keep up, the entire process slows down. This means that the maximum speed at which data can flow through the pipeline is limited by how quickly these downstream tasks can process it. If they're fast and efficient, the pipeline can handle more data per unit of time. But if they're slow or inefficient, they become a bottleneck, limiting the overall throughput of the pipeline. So, optimizing these downstream tasks—making them faster and more efficient—is crucial for maximizing the overall speed and efficiency of the data processing pipeline. It's like fine-tuning the engine of a car to get the best performance out of it.

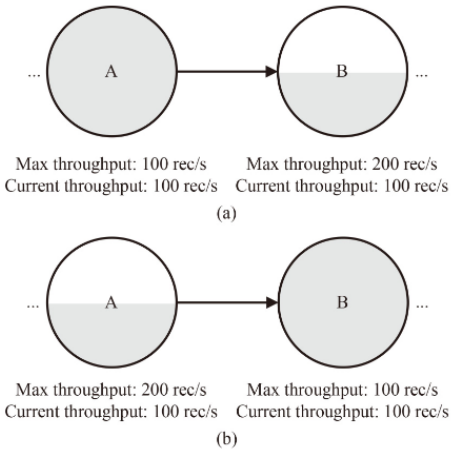

Figure 16: The impact of upstream operators on throughput [2]

Backpressure

Flink uses backpressure to adapt the processing speed of individual operators. The operator can struggle to keep up processing the message volume it receives for many reasons. The operation may require more CPU resources than the operator has available, The operator may wait for I/O operations to complete. If an operator cannot process events fast enough, it build backpressure in the upstream operators feeding into the slow operator. This causes the upstream operators to slow down, which can further propagate the backpressure to the source and cause the source to adapt to the overall throughput of the application by slowing down as well [18].As illustrated, the third task in the sequence triggers a backpressure condition for the second task, which in turn extends the backpressure to the first task in the

pipeline.

Figure 17: Backpressure example in Flink Job [19]

Buffer Metrics

In this part, we focus on input and output buffer pools for intermediate tasks. This approach is aimed at tracking the records queued in each task throughout their execution. In order to maintain consistent high throughput, Flink uses network buffer queues (also known as in-flight data) on the input and output side of the transmission process. Each subtask has an input queue waiting to consume data and an output queue waiting to send data to the next subtask.

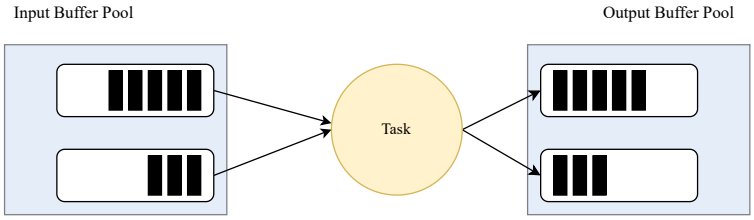

Figure 18: Input and Output buffer pool in a task with two buffers in each pool

More specifically, a buffer pool in Flink is a collection of buffers that are used to hold data temporarily during the network transmission between tasks in a Flink job. Buffer pools live in the input and in the output of a flink task. It is shown below, the process of propagating data.

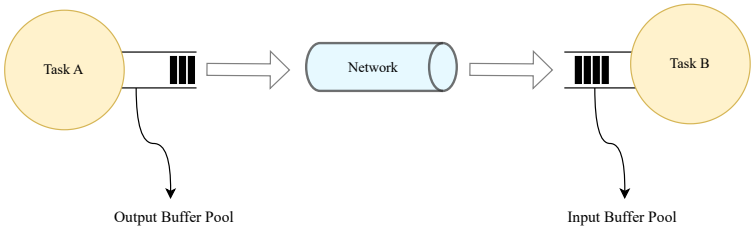

Figure 19: Processed data propagation, one input and output buffer respectively

An important role has the input buffer pool in calculating the number of queued records.Therefore, it is assumed that the lag experienced by an intermediate task corresponds to the records

queued. As mentioned above, when an operator is backpressured or has low throughput, records accumulate in the input buffers, awaiting processing. Thus, to compute the queued records of tasks (except for source), buffer pool metrics are used, as are shown in the formula below.

- **InPoolUsage**: An estimate of the input buffers usage, it represented as percentage
- **bufferSize**: current Input buffer size, calculated as the product of the number of buffers in the pool and the size of each buffer in bytes.

$$\text{Queued Records}_n = \frac{\text{bufferSize} \times \text{InPoolUsage}}{\text{avg(bytesPerRecord)}}$$

Other key metrics that are used, are inPoolUsage and outPoolUsage. Those two metrics are interpreted as the buffers' usage, as percentage. By monitoring those metrics, we can deduce to task's "health", i.e if the task is backpressured, or comprises the bottleneck of the pipeline. It is also important to note the levels of *in* and *out* buffer pool usage and in the table below, all the possible scenarios are quoted.

- **OK**: ratio ≤ 0.1
- **LOW**: $0.10 <$ ratio ≤ 0.5
- **HIGH**: $0.5 <$ ratio ≤ 1

	OutPoolUsage **High**	OutPoolUsage **Low**
InPoolUsage **High**	The current vertex is backpressured by downstream operators	If the upstream task is backpressured then the current vertex is the bottleneck
InPoolUsage **Low**	Starts being backpressured, although the upstream task has not yet been affected	✓

Table 2: InPoolUsage and OutPoolUsage Scenarios

Scenarios Analysis

- **1.1**: Both input and output buffers are full. Output buffers are full, due to the fact that the downstream task is already saturated, preventing the processing of additional records. Likewise, the input buffers are congested as a result of the output buffers being full, thereby impeding further processing. As result, the current task is backpressured by downstream tasks

- **1.2**: Input buffers are full and the output buffers' usage is low. When the upstream task experiences backpressure, it signifies that the current task is the bottleneck. This occurs because the upstream task sends data at a high and the current task cannot keep up, due to lower throughput
- **2.1**: Input buffers' usage is low and output buffers are full. The current task starts being backpressured by downstream tasks, without upstream tasks being affected, since input buffers' usage is low
- **2.2**: Both input and output buffers' usage is low. In this case, operator can cope with this sending rate and the downstream task is in a healthy condition as well.

Idleness

Idleness of a task in Apache Flink is a state that indicates that the task has no input data to process at the moment. This can happen when the task's source or upstream tasks are temporarily inactive or slow, and do not emit any records or watermarks.Moreover, Idleness is complement to busyness of a task. This metric is very important for taking scaling down decisions.

4.2 Algorithm Analysis

In our algorithm, the primary target is optimizing the performance of the pipeline, dynamically. The focus is on identifying tasks that are either under-provisioned and struggling with the workload or over-provisioned and underutilized, so that making effective scaling decisions accordingly. The procedure involves collecting metrics, evaluating them against predetermined thresholds, and then adjusting task parallelism for each task separately. It should be noted, however, that source tasks are handled differently from sinks and intermediates, which will be discussed in the following paragraphs, starting with source tasks. Source cannot be monitored as the other tasks, because their input is Kafka, which is an external service. TALOS was based on HYAS [3] scaling policy, although we differentiate TALOS, in order to convert it in a task scaler.

Source task is the first step in a pipeline and is responsible for reading data from an external service, for example Kafka[**?**, **?**, kafka-docs] or producing data. Thus, sources are responsible for the data flow in the pipeline. It is difficult to recognize whether the source task is the performance bottleneck of the pipeline via the other tasks, so we have to examine the external sources. In this algorithm, we assume that the external source is Kafka, on the ground that flink provides metrics for Kafka sources. A possible way to discern, if the source task has low read speed, is kafka lag. TALOS observes the rate of change in Kafka lag, for the last one minute, in order to ascertain whether lag grows or drops. Nevertheless, this metric is not enough for the decision of the new parallelism. We have to take into consideration the change rate of lag relative to throughput, where source task's throughput is assumed as the consume rate. The reason for taking this relationship (Kafka lag relative to throughput) into consideration is that it shows us if task is going to be over-provisioned or under-provisioned. All those metrics are exported from Prometheus [14], where Flink [11] and Kafka [13] have

exposed their metrics. The relationship is given below:

$$totalLag = \sum_{i \in instances} \text{record\_lag\_max}_i \quad (records) \tag{1}$$

$$throughput = \sum_{i \in instances} \text{records\_consumed\_rate}_i \quad (records/sec) \tag{2}$$

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

The above formula indicates, whether a scaling up or scaling down action will be executed. A positive *relativeLagChangeRate* indicates that there is need of more instances, however, if it is negative, then a scaling down action may be necessary. For scaling down decision, there is a complementary factor that is needed, idleness. When idleness of the source task is over a threshold value, and the above indicator is negative, then source instances will be decreased. Last but not least, an additional condition is backpressure. In case source is backpressured, no scaling action will happen. The explanation is that in this case source is not the bottleneck and a downstream task caused the backpressure. Whenever source task starts being backpressured, Kafka lag is increasing due to the inability to forward records to downstream operators. Therefore, monitoring backpressure on the source task is crucial not to mislead us to incorrect scaling decisions. Below is a part of the algorithm for taking scaling decisions for source tasks, in order to understand scaling conditions, *scale up* and *scale down* will be discussed in following. Following, we represent a high level approach, to understand the conditions of scaling actions for Source operators.

Algorithm 1 Source Scaling Policy

```
while True do
    for each Operator in Operators do
        if Operator is Source then
            if isbackpressured then
                Do nothing
            end if

            if relativeLagChangRate>0 and busy≥ threshold then
                scale up
            end if

            if relativeLagChangRate<0 and idle≥ threshold  then
                scale down
            end if
        else
            ⋮                                   ▷ Logic for downstream operators
        end if
    end for
end while
```

Continuing with **intermediate** and **sink** tasks, we apply the same principles as with source tasks, although with different metrics to derive the aforementioned formulas. The reason for using different metrics is the independent monitoring between tasks and the lag for intermediate operators is to exposed by Flink, so we use buffer metrics to compute it. For those tasks we compute lag, i.e the queued records of a task, using the formula in 4.1 and the throughput as we mentioned in the previous subsection additionally.

$$totalLag = \sum_{i \epsilon instances} \frac{\text{bufferSize} \times \text{InPoolUsage}}{\text{avg(bytesPerRecord)}} \quad (records) \tag{4}$$

$$throughput = \frac{avg(recordsInPerSec)}{avg(BusyTimems)} \quad (records/sec) \tag{5}$$

$$\boxed{relativeLagChangeRate = \frac{\text{deriv}(totalLag)}{throughput}} \tag{6}$$

At this point, the main difference between source tasks and downstream tasks are the conditions on executing scaling actions. For the purpose of detecting the bottleneck, we utilize inPool and outPool usage as it is aforementioned. More specifically, the case where inPoolUsage is high, the outPoolUsage is low and the upstream operator starts being backpressured, concerns us because the vertex is the expected bottleneck of the pipeline. The logic of scaling up or down is below:

Algorithm 2 Intermediate tasks Scaling Policy

```
while True do
    for each Operator in Operators do
        if Operator is Source then
            ⋮                                          ▷ Logic for source tasks
        else
            if relativeLagChangRate>0 and busy≥ threshold and isBottleneck then
                scale up
            end if
            if relativeLagChangRate<0 and idle≥ threshold  then
                scale down
            end if
        end if
    end for
end while
```

Note: *isBottleneck* is when 0.5 < inPoolUsage ≤ 1 and 0.10 < outPoolUsage ≤ 0.5, and the upstream operator starts being backpressured (backpressure > 500). The backpressure limitation is important, because low outPoolUsage and high inPoolUsage, is not enough, to address a tasks as the bottleneck. In case of filter operation for instance, the task may have a large input throughput, but the output may be empty, due to filtering.

Now, determining the necessary instances of a task to accommodate incoming workloads depends on achieving a throughput that reduces the *relativeLagChangeRate* to zero, necessitating a zero numerator, which represents lag. Ideally, we assume a direct correlation: doubling the number of task instances should double the throughput they can handle, indicating a linear scalability. This rationale,i.e the correlation between tasks's parallelism and throughput ,is applied iff there is not skewed distribution in data, i.e an unbalanced distribution of data to parallel instances in a task. Therefore, this is the formula for new parallelism π for a specific task, in **scale up** action:

$$\pi = \lceil currentParallelism \times (relativeLagChange + 1) \rceil \tag{7}$$

Nevertheless, in a scale down action, parallelism is decreased by 1. The rationale for such as preservative decision, is to prevent under-provisioning, in case of workload spikes or sudden workload raise. Thus, we decide to deplete the parallelism conservatively, in order to keep up with the workload in unpredictable fluctuations, since our target is the good performance of the pipeline. Following, the TALOS algorithm is presented, that contains both aforementioned algorithm parts.

Algorithm 3 TALOS Scaling Algorithm

while True **do**
  for each Operator in Operators **do**
    if Operator is *Source* **then**

      if isbackpressured **then**
        Do nothing
      end if
      totalLag = $\sum_{i \in instances}$ record_lag_max$_i$
      throughput = $\sum_{i \in instances}$ records_consumed_rate$_i$

    else
      totalLag = $\sum_{i \in instances} \frac{\text{bufferSize} \times \text{InPoolUsage}}{\text{avg(bytesPerRecord)}}$

      throughput = $\frac{avg(recordsInPerSec)}{avg(BusyTimems)}$
    end if

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

    ▷ *If task is source isBottleneck flag is always true*
    if relativeLagChangRate>0 **and** busy≥ threshold **and** isBottleneck **then**
      $\pi = \lceil currentParallelism \times (relativeLagChange + 1) \rceil$
    end if

    if relativeLagChangRate<0 **and** idle≥ threshold **then**
      π = (currentParallelism - 1)
    end if
  end for
end while

Below we created an identifying example, for plain explanation of scaling up operators in a pipeline. In this example, there are isolated two tasks of a pipeline, all metrics have numerical values, in order to understand better the process of identifying if there is need of a scale up action

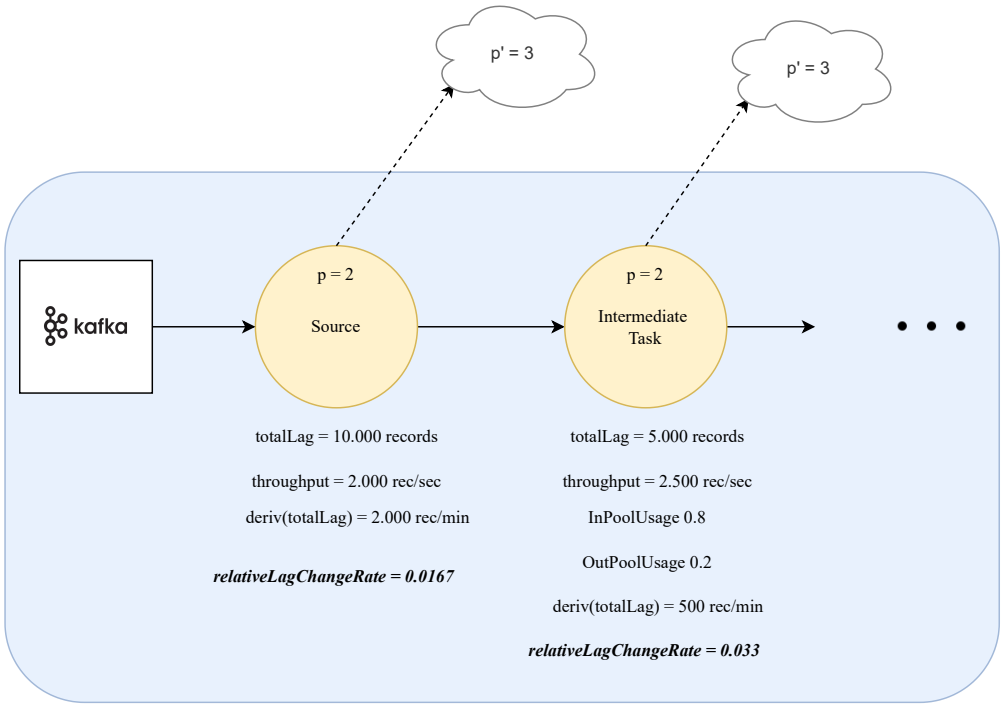

Figure 20: Example for scale up state

5 Performance Evaluation

In order to assess the effectiveness and practical applicability of the proposed scaling algorithm, a comprehensive performance evaluation is conducted. This section aims to provide a detailed analysis of the algorithm's efficiency, scalability, and overall impact on the execution of Apache Flink-based workflows. Furthermore, we will conduct a comparative analysis between our scaling algorithm and the Apache Flink Kubernetes Operator Autoscaler. This comparison aims to assess the respective strengths and weaknesses of each scaling approach.

5.1 Infrastructure

In this subsection, we describe the infrastructure we used to conduct our experiments and evaluate our system. We discuss the platform, the cluster, the nodes, and the tools we used, and how they contributed to our experimental setup.

Flink Kubernetes Operator

The use of the Flink Kubernetes Operator for deploying and later utilizing it as a scaler in our experimental setup for comparison and it was deployed using helm. This operator simplifies the deployment and management of Apache Flink applications on Kubernetes clusters, providing a declarative way to configure and scale Flink jobs dynamically. The operator seamlessly integrates with Kubernetes resources, optimizing the allocation of Flink TaskManagers and JobManagers for efficient resource utilization. Job manager and Task managers are set to allocate 1 CPU, 1 GB memory respectively.

Flink Job

For our experiments we used a Flink streaming job designed for click fraud detection. Click fraud poses a significant threat to the integrity of online advertising platforms, where malicious actors generate fake clicks to exploit pay-per-click advertising models.

The job employs windowing techniques, watermarking, and sliding time windows to identify patterns within specified time intervals. Key functionalities of the application include counting the number of clicks per IP address and user ID, calculating click-through rates (CTR) per user, and identifying instances where CTR surpasses a predefined threshold. Watermarking is utilized for handling event time, and sliding windows are implemented to account for lateness in event arrivals. Below is the pipeline of the Flink job.

The Flink job consumes JSON-formatted events from a Kafka topic named "topic" using a Kafka source. The results, highlighting patterns indicative of potential fraudulent activities, are then sent back to Kafka, specifically to the "topic-out" Kafka topic. Notably, this process incorporates a strategic split in the initial task.

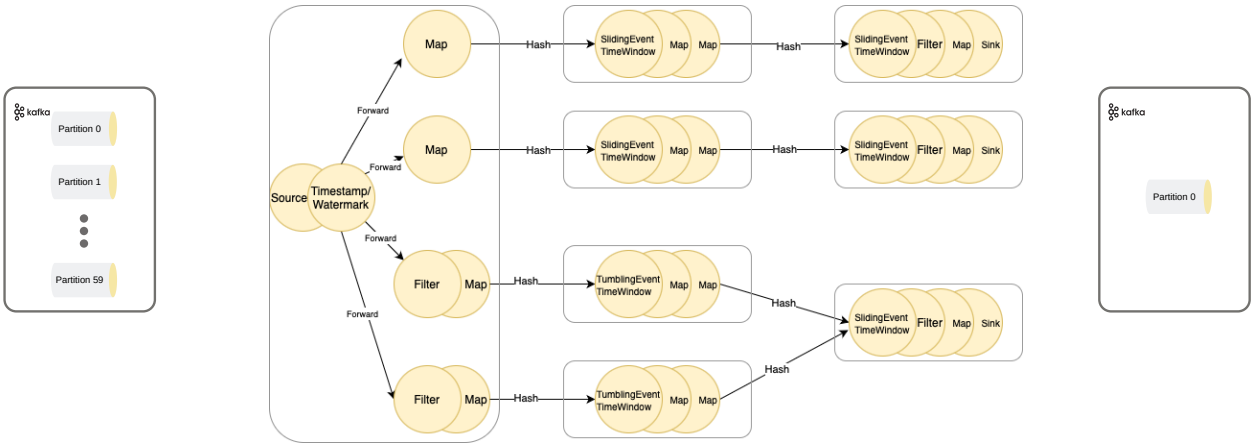

Figure 21: Click Fraud Detection Pipeline

Apache Kafka

In our experimental setup, Apache Kafka was deployed using Helm. The Kafka configuration, featuring a replica count of 1 and a 100GB persistence size, ensured data resilience and provided ample storage capacity. The input topic 'topic' was deliberately configured with 60 partitions,a deliberate choice aimed at introducing ample pressure in the Flink job. The main reason for this choice was reflecting scenarios of high-throughput and distributed data processing challenges. Furthermore, metrics monitoring through JMX, with *metrics.jmx.enabled: true*, provided essential insights.

Prometheus

Prometheus plays a pivotal role in our algorithm by providing essential real-time metrics for our Apache Flink-based system. The metrics are listed above:

- inPoolUsage
- outPoolUsage
- backpressure
- idle Time
- Kafka lag
- Buffers size
- Throughput

Moreover, the use of Prometheus helped us in tracking and analyzing experiment outcomes.

Google Cloud Platform

In conducting our experiments, we utilized the capabilities of the Google Cloud Platform (GCP) to create a robust and scalable computing environment. Specifically, our infrastructure was hosted on the Google Kubernetes Engine (GKE). Within this GKE-hosted environment, we deployed a cluster comprising three compute nodes, strategically placed in the Europe West 1c zone. Each node was configured as a C2-standard 8 instance, equipped with 8 virtual CPUs (4 cores) and 32 GB of memory. This configuration was chosen to ensure ample computational power and memory resources, aligning with the demands of our Apache Flink job scaling experiments.

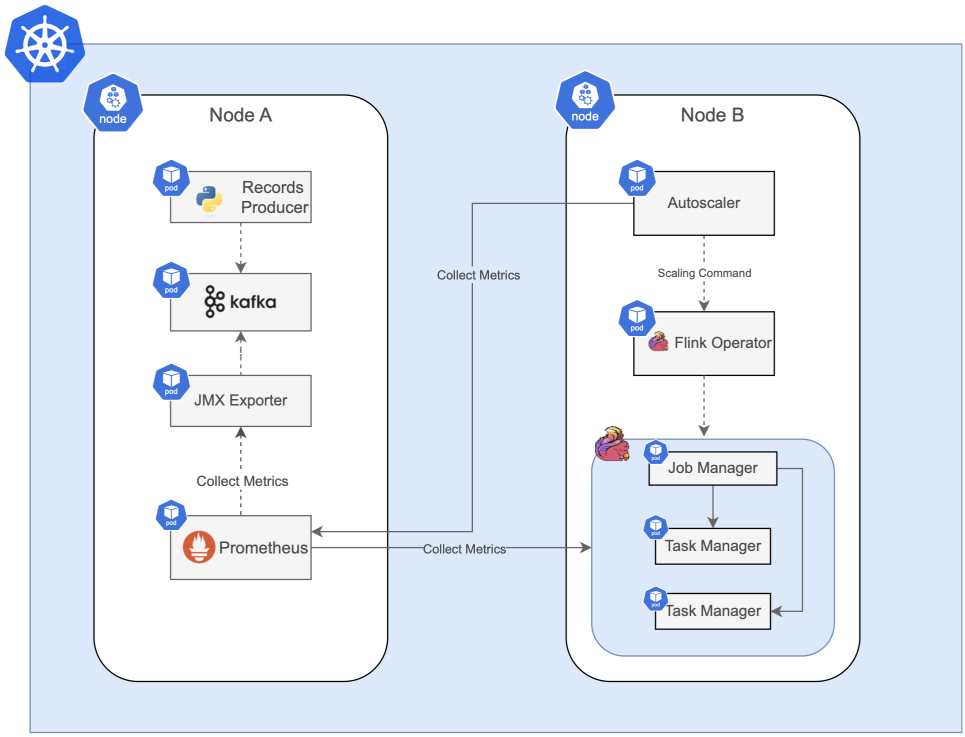

Figure 22: Caption for the image

5.2 Experiment 1: Short-Duration Scalability Test

In this experiment, we will conduct a one-hour test with an oscillating workload. The workload is illustrated below. For both algorithms we are going to test the autoscalers with different time intervals between scaling actions, in order to choose the best for the 10 hour experiment 2. The intervals are 3 minutes and 5 minutes. In the below discussion, we are going to refer to Flink Kubernetes Operator autoscaler, as FKO.

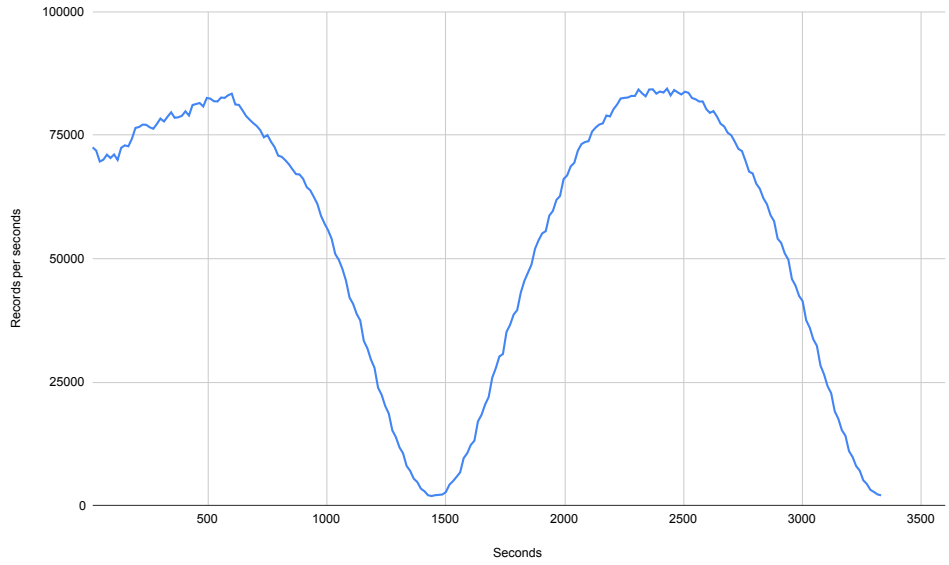

Figure 23: Workload for experiment 1

5.2.1 Scaling Actions

TALOS Scaling Actions - 3min Interval

Initially, let's examine the outcomes of the TALOS Algorithm. The graph portrays the one-minute rate of incoming produced messages. The red and yellow dots in the graph give us a quick visual on when scaling actions occurred. Red dots denote occasions of scaling up, indicating heightened parallelism, whereas yellow dots indicate instances of scaling down, signifying a reduction in parallelism.

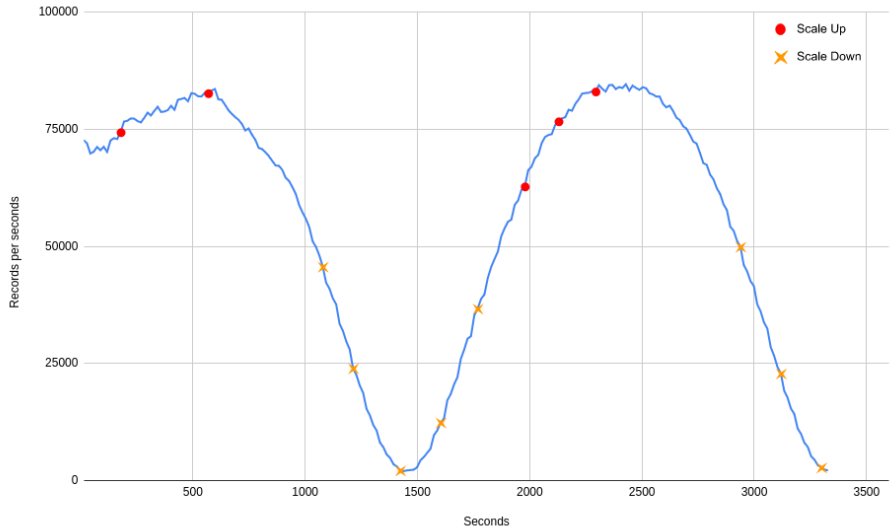

Figure 24: Workload distribution and scaling actions' moments

Flink Kubernetes Operator Scaling Actions - 3min Interval

Continuing with taking a closer look at Flink Kubernetes Operator's algorithm in our experimental setup. Hereafter, below is the figure for this algorithm.

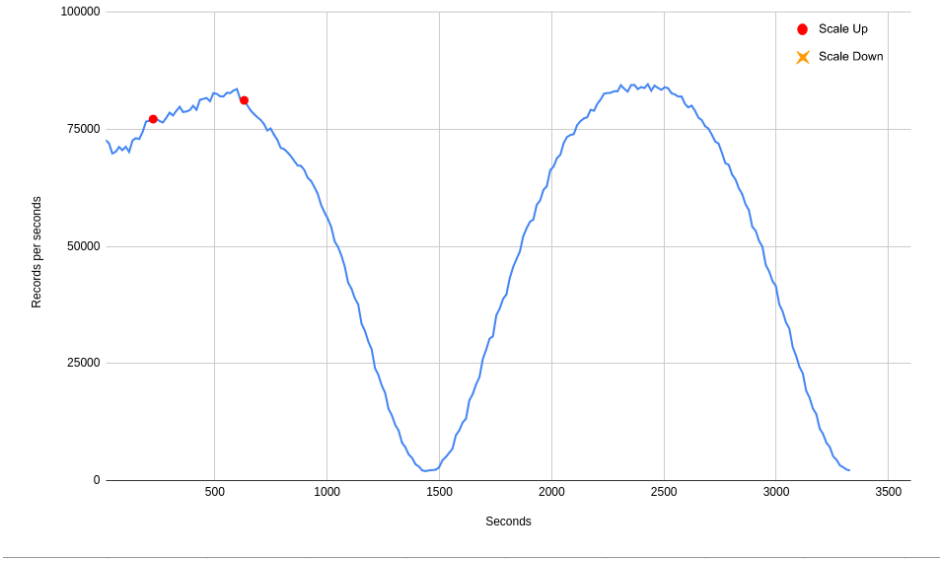

Figure 25: Workload distribution and scaling actions' moments

Since, the scaling actions are apposed above, let us now inspect the orchestration of task managers, as depicted in the graph below and the tasks' parallelism separately.

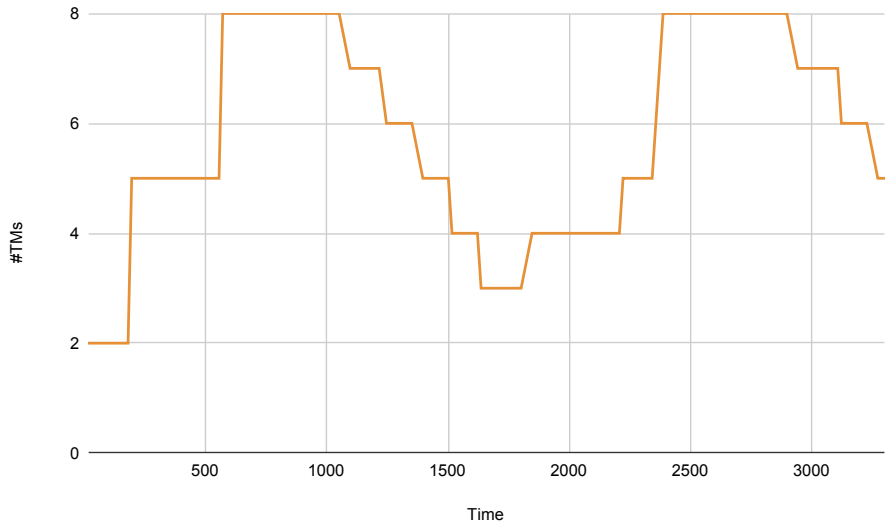

Figure 26: Task Managers in TALOS

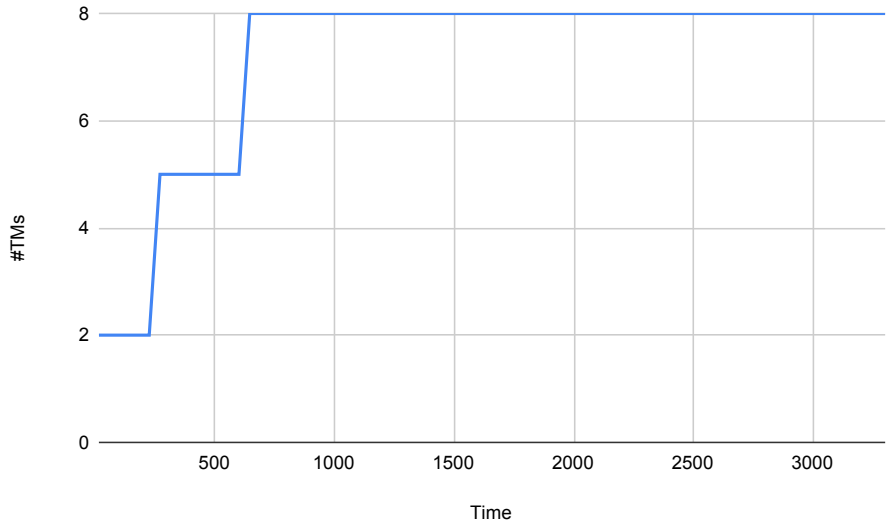

Figure 27: Task Managers in FKO

From the two aforementioned figures, we observe that the number of Task Managers in the TALOS algorithm varies from 2 to 8, alternating according to the workload. In contrast, examining the results of the Flink Kubernetes Operator, we note that the task managers initially increase as the load grows, without any subsequent reduction during the experiment. However, conclusive insights cannot be drawn at this point, and further analysis is required, which will be conducted below.

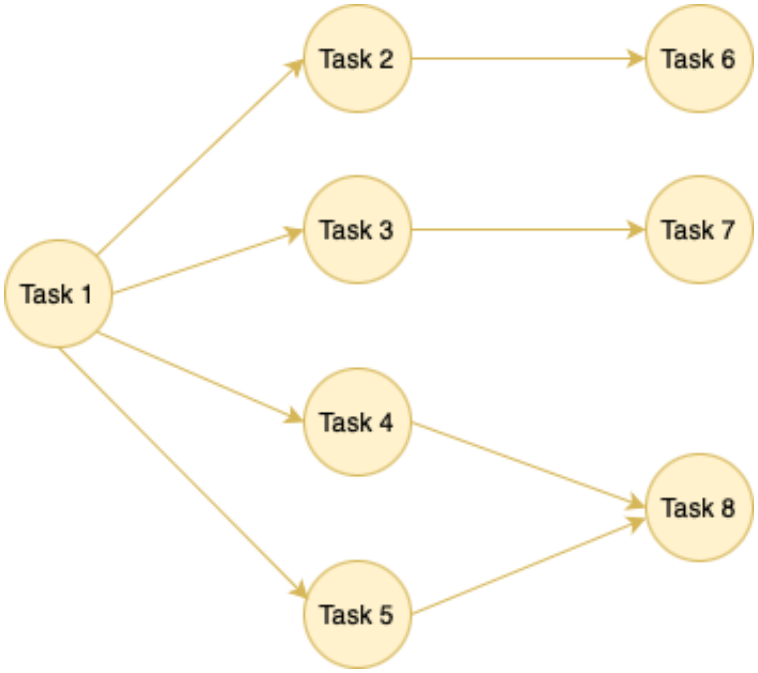

Figure 28: High level depiction of tasks

	Scaling Actions
Task 1	2 → 5 → 8 → 7 → 6 → 5 → 4 → 3 → 4 → 5 → 8 → 7 → 6 → 5
Task 2	2 → 2 → 2 → 2 → 1 → 1 → 1 → 1 → 1 → 2 → 2 → 2→ 1 → 1
Task 3	2 → 1 → 2 → 2 → 1 → 1 → 1 → 1 → 1 → 2 → 2 → 2→ 1 → 1
Task 4	2 → 1 → 2 → 1 → 1 → 1 → 1 → 1 → 1 → 1 → 2 → 1→ 1 → 1
Task 5	2 → 1
Task 6	2 → 1
Task 7	2 → 1
Task 8	2 → 1

Table 3: TALOS Task Scaling Actions

	Scaling Actions
Task 1	2 → 5 → 8
Task 2	2 → 1 → 2
Task 3	2 → 1 → 2
Task 4	2 → 1
Task 5	2 → 1
Task 6	2 → 1
Task 7	2 → 1
Task 8	2 → 1

Table 4: Flink Kubernetes Operator Scaling Actions

The analysis reveals a notable influence of tasks 1, 2, 3, and 4 on the overall pipeline dynamics, particularly during their fluctuations. Task 1, representing the source operation, stands out with the most pronounced impact. This can be attributed to the source component managing data ingestion from 60 Kafka partitions. The challenge arises due to the producer's high send rate to Kafka, leading to increased pressure on the source task. Below, are shown the results of the scaling actions, where the interval is 5 minutes between scaling actions, and the metrics monitoring interval is 3 minutes, i.e the system will wait for 5 minutes after scaling actions, otherwise it monitors metrics metrics after 3 minutes.

TALOS Scaling Actions - 5min Interval

Following, the scaling actions for each algorithm are illustrated.

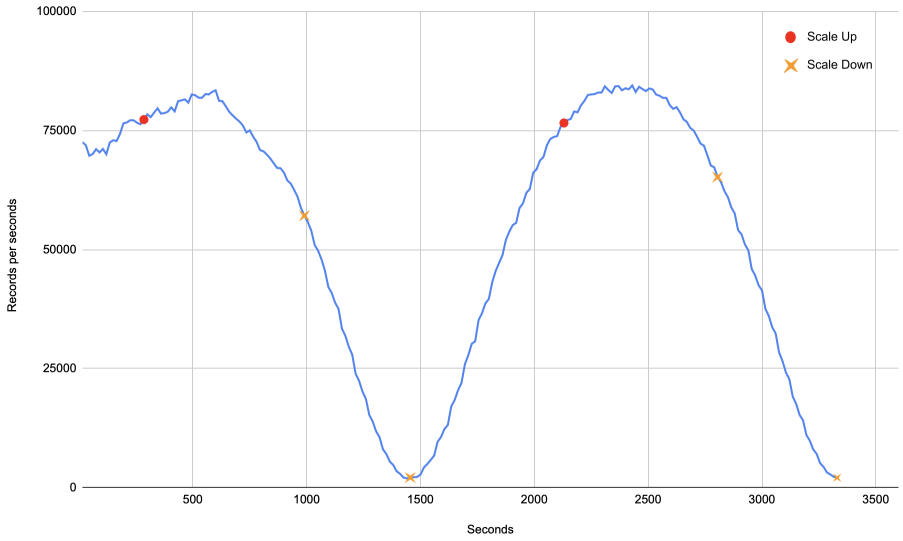

Figure 29: Scaling Actions For TALOS

Flink Kubernetes Operator Scaling Actions - 5min Interval

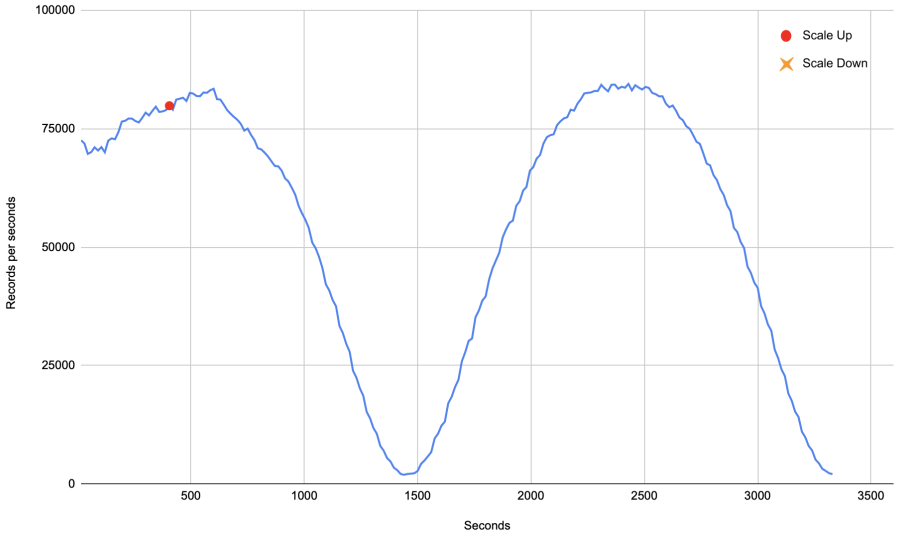

Figure 30: Scaling Actions For FKO

Since, the scaling actions are apposed above, let us now inspect the orchestration of task managers, as depicted in the graph below and the tasks' parallelism separately.

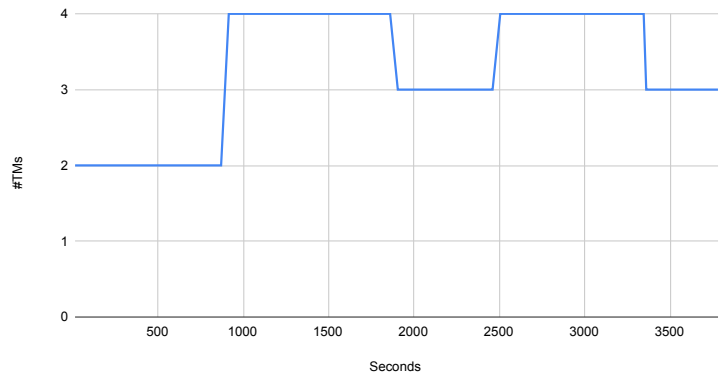

Figure 31: Task Managers For TALOS

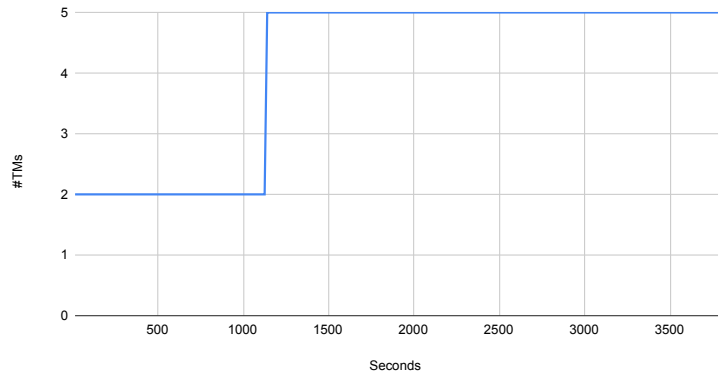

Figure 32: Task Managers For FKO

It is noticeable that the system's behaviour is better with a 5 minutes interval, since the system has more time to stabilize. Using larger time interval, both algorithms choose lower parallelism in the peaks of the workload. This is a rationale behaviour, since in 3 minutes the job is not fully stabilized, and as a result both scaling algorithms set bigger parallelism to catch the backlog, that happened after the first scaling action. Let us now take a closer look at tasks parallelism. In the following tables are in detail the scaling actions of each tasks. Tasks are in respect to figure 28.

	Scaling Actions
Task 1	$2 \rightarrow 4 \rightarrow 4 \rightarrow 3 \rightarrow 4 \rightarrow 4 \rightarrow 3$
Task 2	$2 \rightarrow 2 \rightarrow 1 \rightarrow 1 \rightarrow 2 \rightarrow 1 \rightarrow 1$
Task 3	$2 \rightarrow 2 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 1 \rightarrow 1$
Task 4	$2 \rightarrow 1$
Task 5	$2 \rightarrow 1$
Task 6	$2 \rightarrow 1$
Task 7	$2 \rightarrow 1$

Table 5: TALOS Task Scaling Actions

	Scaling Actions
Task 1	2 → 5
Task 2	2 → 2
Task 3	2 → 1
Task 4	2 → 1
Task 5	2 → 1
Task 6	2 → 1
Task 7	2 → 1

Table 6: FKO Task Scaling Actions

As it is shown, TALOS decided for the source task parallelism 4,as the max parallelism, and as the minimum parallelism 3, during the experiment. For task 2 and 1, the parallelism is increased, only when the workload, starts to descend. As we aforementioned, task 1 is the source task, that reads Kafka and tries to reach the produce rate. In the next subsection, we will discuss the results of choosing larger interval between scaling actions, and the affection of deciding different parallelism.

5.2.2 Algorithms Comparison

Observing the **TALOS** graphs, it's evident that the count of task managers varies in response to the workload.The dynamic adjustment of task managers plays a pivotal role in achieving optimal resource utilization based on the current workload.The increment in the number of pods, with each representing a Task Manager in our cluster, may lead to a corresponding increase in virtual machines (VMs), if the Kubernetes cluster autoscaler is enabled. This escalation in VMs, while enhancing system capacity, is accompanied by an associated rise in costs. Furthermore, this dynamic orchestration significantly improves overall system performance by allocating resources precisely where and when they are needed. Nevertheless, there are instances of swift scaling down actions, attributed to temporary increases in idleness.

FKO algorithm is designed to optimize the throughput of the Flink job. Notably, the algorithm tends to be more conservative in making decisions to scale down. However, this cautious approach has its drawbacks, as it may result in the allocation of extra resources even in situations where the workload does not justify it. As mentioned earlier, upon activation of the Kubernetes cluster autoscaler with an initial 2 VMs, an extra 3 nodes are necessary as the parallelism reaches 8 and 5 respectively for intervals 3 minutes and 5 minutes. Consequently, the ongoing addition of nodes, even when unnecessary, the results are increasing charges.

Results for 3 minutes Interval between scaling Actions

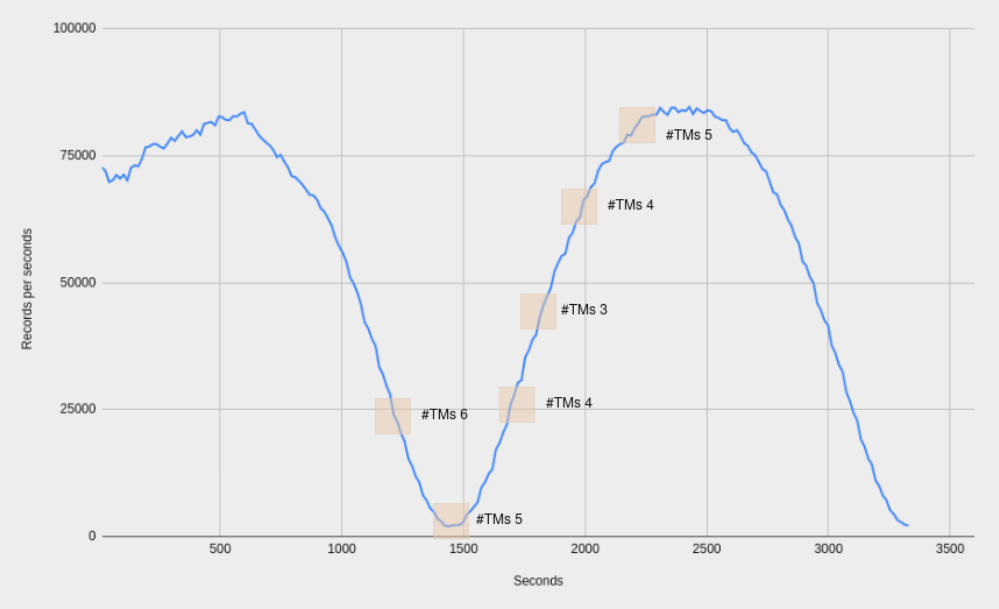

Figure 33: Intervals of latency measurements

In the figure 33 the markers correspond to the intervals during which we decided to measure latency and lag. We opted for these intervals because the two algorithms chose different parallelization strategies. Notable it is, that at first two scaling actions, reaction is the same. Let us proceed to comparing the algorithms, where the workload decreases and the TALOS decides scaling down actions.

Latency	TALOS #TMs 6	FKO algorithm #TMs 8
Minimum	10.4 ms	9.13 ms
Average	36 ms	29 ms
Maximum	77 ms	66 ms

Latency	TALOS #TMs 5	FKO algorithm #TMs 8
Minimum	10 ms	9 ms
Average	15 ms	9 ms
Maximum	23 ms	9 ms

Latency	TALOS #TMs 4	FKO algorithm #TMs 8
Minimum	48 ms	23 ms
Average	69 ms	42 ms
Maximum	83 ms	75 ms

Latency	TALOS #TMs 3	FKO algorithm #TMs 8
Minimum	89 ms	55 ms
Average	144 ms	73 ms
Maximum	234 ms	121 ms

Latency	TALOS #TMs 4	FKO algorithm #TMs 8
Minimum	75 ms	53 ms
Average	123 ms	110 ms
Maximum	285 ms	274 ms

Latency	TALOS #TMs 5	FKO algorithm #TMs 8
Minimum	77 ms	65 ms
Average	97 ms	95 ms
Maximum	129 ms	114 ms

Table 7: Compare latency with respect to parallelism

As it shown in the above tables, the values of average latency is ≅ 80% congruent in both algorithms' result. TALOS, which decided for a lower parallelism level, achieves latency metrics close to those of the Flink Kubernetes Operator, which selected a higher parallelism of 8. Below, we will discuss the consume rate in each of the above cases.

Workload rate (rec/sec)	#TMs TALOS	Consume rate TALOS (rec/sec)	#TMs FKO	Consume rate FKO (rec/sec)
22.419	6	24.435	8	22.595
1455	5	2.323	8	1703
20.444	4	23.325	8	21.595
45.474	3	48.902	8	47.471
66.123	4	70.132	8	69.865
76.567	5	82.896	8	79.405

Table 8: Consume rate in each case

The above table presents the consume rate of the pipeline. The consume rate is larger or almost equal with the workload rate. This higher consume rate, especially when it exceeds the workload rate, can be attributed to the accumulation of a backlog, a direct consequence of scaling actions within the system. Such a backlog arises as the pipeline adjusts to the scaling operation, temporarily allowing data to queue up before it can be processed at the new scale. TALOS' strategic decision to scale down does not detrimentally impact the pipeline's ability to manage the workload, since the pipeline's consume rate is larger than the workload. However, frequent scaling actions cause lag, i.e backlog. Thus, following we will show that not often scaling actions will cause less lag and the pipeline converges to workload rate.

Results for 5 minutes Interval between scaling Actions

As we notice in the above section, 5 minutes interval between the scaling actions, helps the system not taking arbitrary decisions. Such arbitrary decisions arise when the system lacks sufficient time to stabilize, leading to confusion for the algorithms due to the backlog created by prior scaling activities. Consequently, the interval of 3 minutes is inadequate for the system to make well-considered decisions. Let us prove that 5 minutes interval has better results, in the light of maximizing performance and minimizing parallelism, i.e cost.

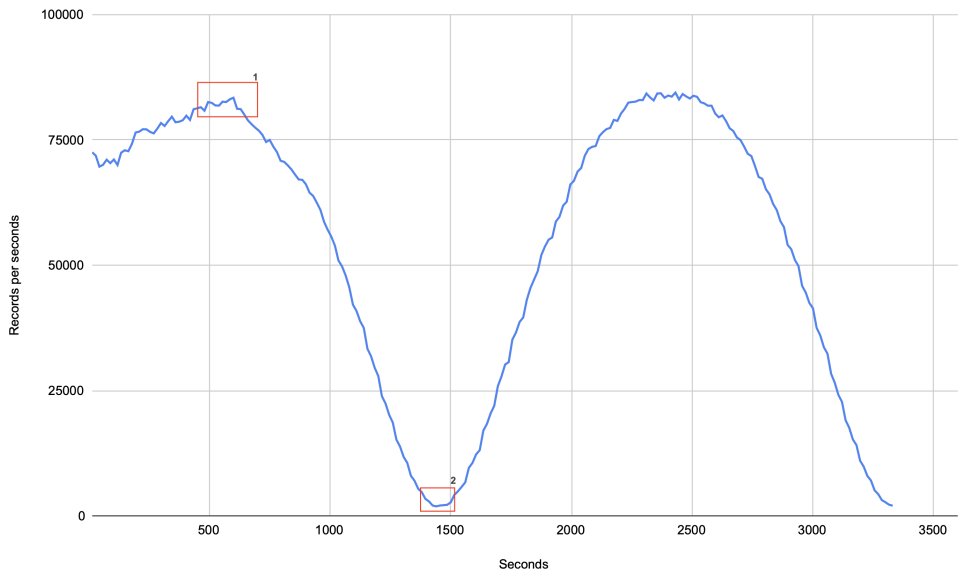

Figure 34: Intervals of latency measurements

We examine latency and consume rate in those two phases. In phase 1, TALOS chose 4,2,2 for tasks 1,2,3 respectively. Flink kubernetes operator decided 5,2,1 for tasks 1,2,3 respectively. In phase 2, TALOS and FKO decided 3,1,1 and 5,2,1 respectively. In the following tables, latency and consume rates are represented.

Phase	Avg Latency TALOS	Avg Latency FKO
Phase 1	114 ms	110 ms
Phase 2	22 ms	21 ms

Table 9: Latency for each phase

Phase	Workload rate (rec/sec)	Consume rate TALOS (rec/sec)	Consume rate FKO (rec/sec)
Phase 1	83.117	86.483	86.394
Phase 2	2.222	2.403	2.345

Table 10: Consume rate for each phase

Both algorithms achieved a notable decrease in latency, with TALOS at 22 ms and FKO at 21 ms, illustrating that efficiency in data processing was maintained across both setups, even with reduced parallelism levels, in case of TALOS decisions. Concerning consume rates, both TALOS and FKO managed to process data at rates slightly above the incoming workload

rate in the first stage, demonstrating an adeptness at handling and surpassing the rate of incoming data. Specifically, TALOS showed a consume rate of 86.483 rec/sec, and FKO had 86.394 rec/sec, against a workload rate of 83.117 rec/sec. In phase 2, with a reduced workload rate of 2.222 rec/sec, both systems effectively adjusted, with TALOS at a consume rate of 2.403 rec/sec and FKO at 2.345 rec/sec. The consume rate is slightly over the workload rate, and this happens because the system achieved to stabilize, in case of TALOS.
With the deliberate 5-minute intervals between scaling actions, suggests the importance of giving the system enough time to adjust post-scaling. This timing seems to prevent premature scaling decisions, facilitating more precise adjustments by both TALOS and FKO in response to real-time demands. This strategic approach helps in maintaining operational efficiency. Therefore, in the following experiment, we will choose the 5 minute interval between scaling actions, and 3 minutes to monitor the desired metrics, in case no scaling action occured.

5.3 Experiment 2: Long-Duration Scalability Test

In this experiment, we will conduct a 10-hours test with an oscillating workload. The workload is illustrated below.

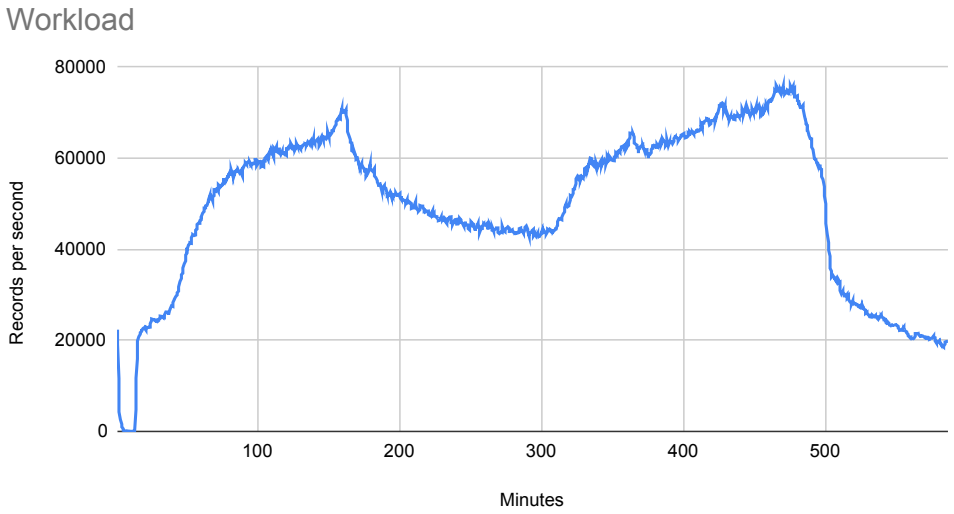

Figure 35: Workload for experiment 2

5.3.1 Scaling Actions

As it is shown, this kind of workload has a lot of fluctuations and it is ideal for evaluating both algorithms. Beneath we are going to examine the performance of both algorithms. In this experiment we follow a different line in the results representation, due to the large duration (10 hours).

TALOS Algorithm

Following is apposed the scaling actions in a Task level. As it is shown in figure 36, we included only tasks 1,2 and 3, since are the only tasks that scaling actions occur. The other tasks scale down at the beginning of the experiment, where the input rate is low and the parallelism stays on 1 for the whole duration of the experiment.

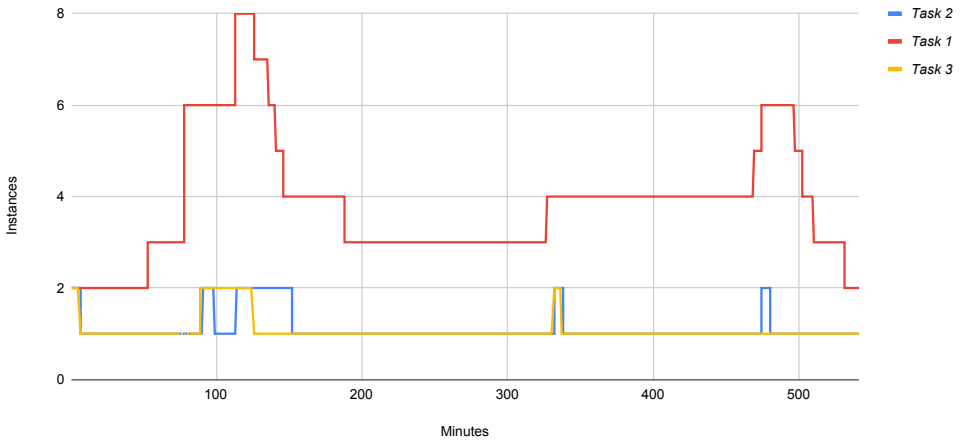

Figure 36: Instances of Tasks 1,2 and 3

Now, we have to inspect more thoroughly the scaling actions and map them with the workload.

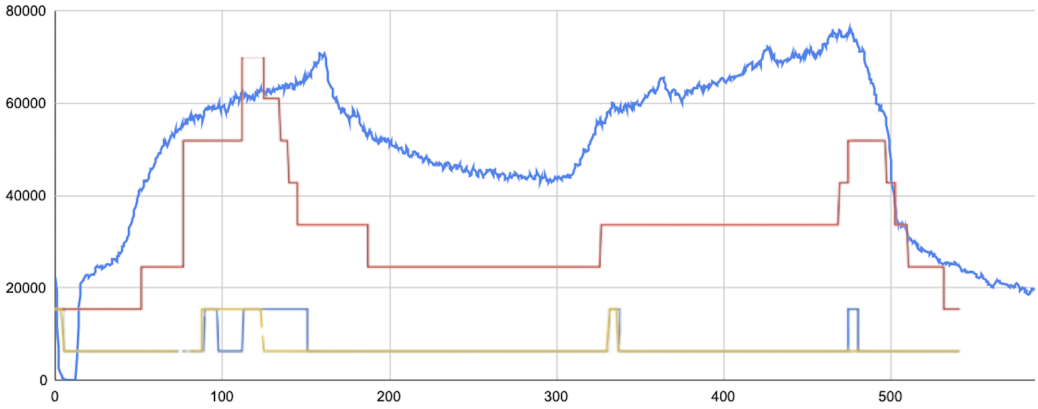

Figure 37: Workload and Tasks' parallelism

As it is illustrated above, we can observe that the parallelism changes as the workload changes as well. Significant points are when source's parallelism increases from 6 to 8, and then starts decreasing again, when the peak of the workload takes place. This happens due to the initial rapid escalation, possibly caused by a large number of records being queued. Thus, the algorithm increased parallelism to handle the records raise and after addressing

this situation, and transitioning to a period of balance, started to decrease parallelism since there was no need and could deal with the workload. After that, parallelism stays consistent at 3,1,1 for tasks 1,2 and 3 respectively. In the next phase of workload escalation, increment is softer, therefore TALOS avoided a steep ascent in parallelism. The decision of increasing the parallelism occured, before the peak arises, and the maximum parallelism was 6,2,1 for tasks 1,2 and 3 respectively. Thereafter, as sending rate decreases, the parallelism begins to decline too.

Flink Kubernetes Operator

In this part, we examine flink kubernetes operator behaviour for the above workload. As is depicted in the figure, the operator maintains parallelism steady at 5 and begins to reduce it at the end of the experiment. Lastly, the other tasks' parallelism changes only at the start of the experiment, from 2 to 1.

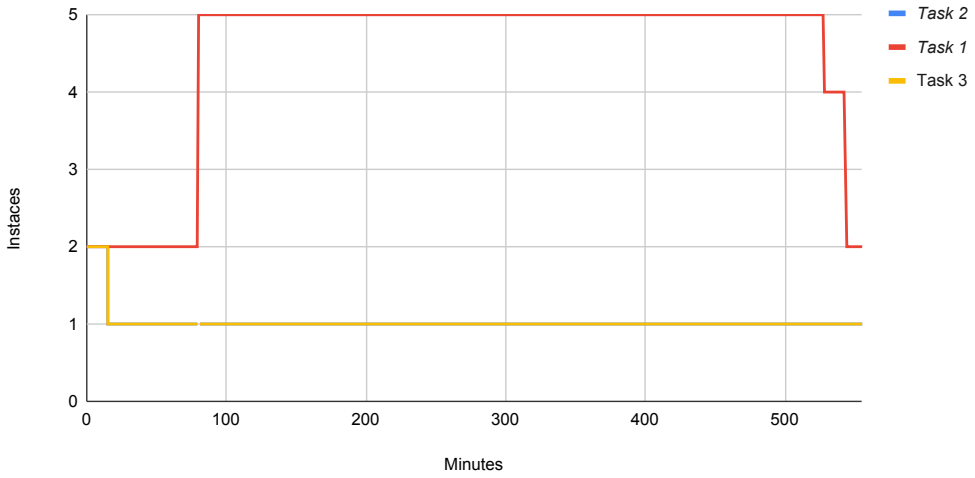

Figure 38: Instances of Tasks 1,2 and 3

Now, we have to inspect more thoroughly the scaling actions and map them with the workload.

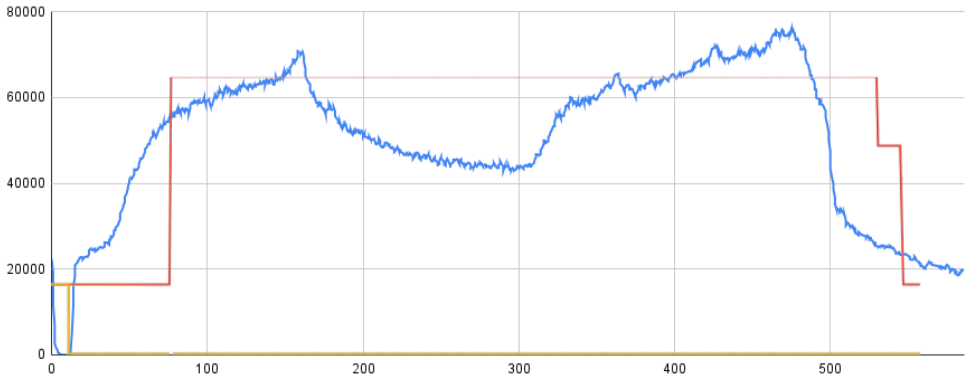

Figure 39: Workload and Tasks' parallelism

As demonstrated in this graph parallelism is 5 for almost for the whole duration of the experiment. As it is noted in experiment 1, flink kubernetes operator keeps parallelism at a high value for proactively managing potential workload fluctuations. Moreover, the parallelism decreased when the send rate takes a long-lasting downward turn. However, this behaviour has drawbacks, that will be discussed in the following sections.

5.3.2 Algorithms Comparison

In this subsection, we quote the results of the second experiment. We refer to latency, lag and throughput in different phases of the workload. These metrics are key indicators of performance. Metrics are only recorded if the system stabilizes, especially after scaling actions, where we ensure a waiting period of at least 1.5 minutes for the system to settle. Also, in this experiment, due to the duration we focalize at the "big picture" and not in specific parts of the workload.

	Average Latency
TALOS	173 ms
FKO Algorithm	151ms

Table 11: Average Latency for 10 hours experiment

As it is shown above, in table 11, here is a noticeable difference of 22 ms in average latency between the two measurements. Whether this difference is significant depends on the context of the application and its requirements. Nevertheless, this difference is rational since Flink kubernetes operator opts for a higher parallelism value for a larger portion of the time, in comparison with TALOS.

	Average Lag
TALOS	771 records
FKO Algorithm	849 records

Table 12: Average Lag for 10 hours experiment

Considering this metric, TALOS has a better performance, since lag has a smaller average value, than FKO algorithm. A possible explanation is that TALOS, decided larger parallelism, in the two peaks of the workload. By allocating more resources to handle the increasing demand, TALOS effectively minimizes lag. This approach highlights the importance of dynamic resource management in optimizing system performance, particularly in scenarios with fluctuating workloads.

Let us now take a closer look at throughput. We examine throughput in the two peaks of the workload and in the almost stable send-rate phase, as it is depicted below in 4.1. Throughput is measured by the number of records the pipeline processes per minute, and then calculated per second.

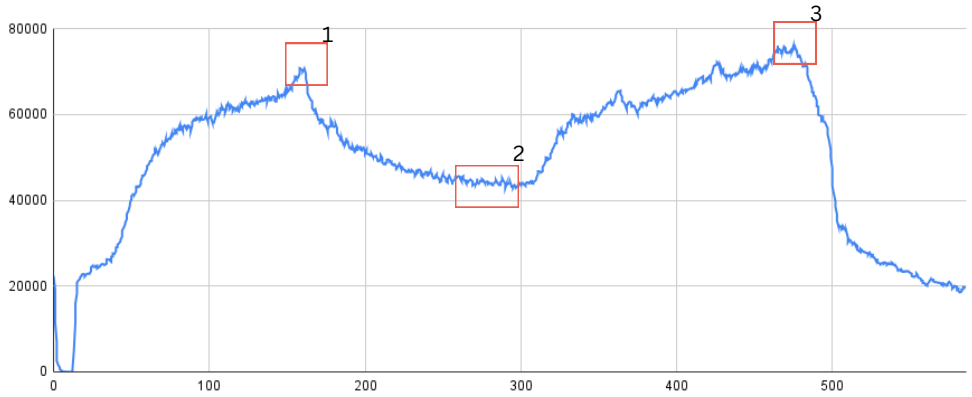

Figure 40: Moments for average throughput calculation

Phase	TALOS Throughput	FKO Algorithm Throughput
1	73.898 rec/sec	71.991 rec/sec
2	44.691 rec/sec	44.945 rec/sec
3	81.359 rec/sec	79.306 rec/sec

Table 13: Throughput table

Continuing with analyzing each phase:

- **Phase 1:** In this phase send rate is ≈70.000 rec/sec. TALOS decided parallelism 4,1,1 (tasks 1,2,3 respectively) and throughput is 73.898 rec/sec, bigger than the send rate. However, this is rationale behaviour due to possible lag that has to process,

due to previous scaling action. Although the low parallelism, pipeline can handle the send rate, and can process more than the send rate. Flink Kubernetes operator chose parallelism to be 5 and throughput is related to send rate.

- **Phase 2:** In this phase send rate is ≈44.000 rec/sec. Both pipelines are related to send rate, although TALOS has chosen a lower parallelism, i.e 3,1,1, and Flink kubernetes operator chose 5.
- **Phase 3:** In this phase send rate is ≈76.000 rec/sec. TALOS has chosen parallelism 6,1,1, in contrast to Flink Kubernetes operator, that decided parallelism 5,1,1. Both pipelines can keep up with the send rate, due to high parallelism and can handle the lag as well. Potentially, the higher parallelism is not needed for this send rate, yet TALOS chose this value the parallelism, in order to handle the arisen lag.

Continuing with calculating the cost, we have to take a closer look to the number of task managers during the experiment. As it is aforementioned in the experiment infrastructure, each TaskManager is a pod. We start the experiment with two VMs, for the whole setup and parallelism 2. However, when parallelism is over 4, a third VM is needed as we mentioned in the experiment 1.

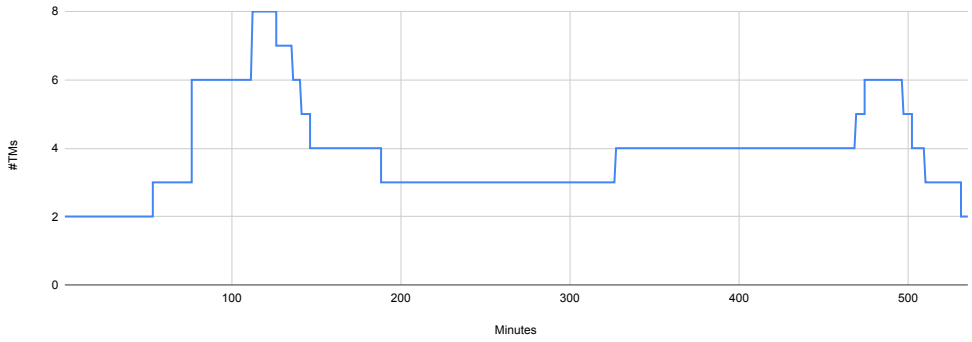

Figure 41: Task Managers in TALOS

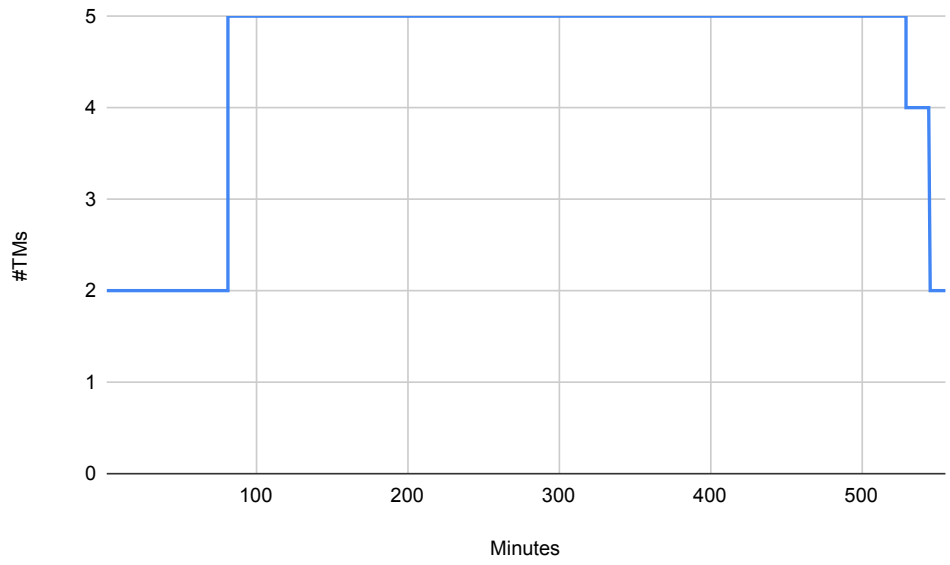

Figure 42: Task Managers in FKO

The histogram above displays the total cost for the 10-hour experiment alongside an estimated monthly cost, assuming the workload pattern repeats daily.

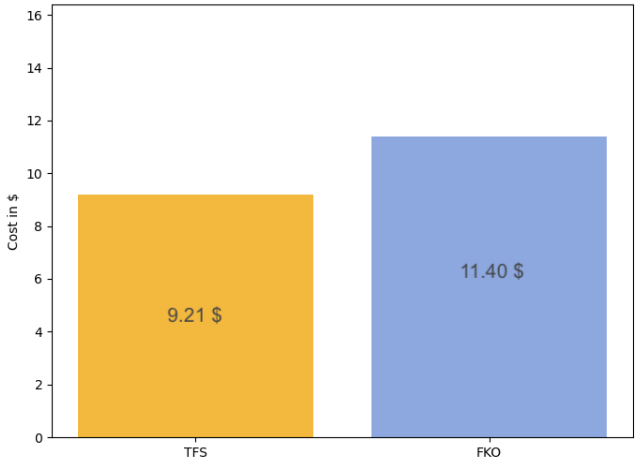

Figure 43: Cost for 10 hour experiment

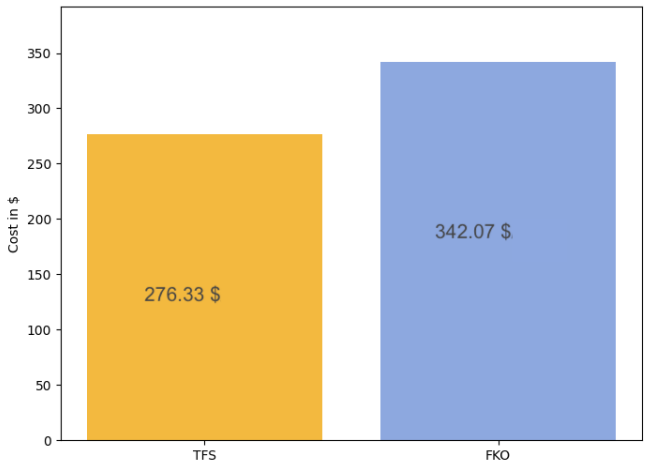

Figure 44: Monthly Cost

The monthly cost for this experiment using TALOS is 276.33$. In contrast, FKO gave the cost of 342.07 $ per month. As it is illustrated above, it is obvious that the decisions of Flink kubernetes operator have a larger cost, since its target is optimizing throughput and not optimzing cost. On the other hand, TALOS tries to achieve great performance and better cost. TALOS escalates the parallelism and doesn't allocates redundant resources. Nevertheless, in the following section, weaknesses and strengths, will be discussed.

6 Conclusion

In this thesis we have developed TALOS, a Task Auto-scaler for Apache Flink. TALOS' goal is optimizing Flink jobs' performance while minimizing infrastructure costs in a cloud environment. This work has been deployed in the cloud using Google's Kubernetes Engine to utilize its effective elasticity mechanisms for creating and scaling container-based applications.

TALOS is a re-active task scaler, determining the optimal parallelism for each task independently within the pipeline. Its innovative approach lies in the autonomy of monitoring metrics for each task, which are not influenced by either the output rate of upstream operators or the input rate of downstream operators. Consequently, the parallelism chosen for each task is exclusively based on metrics specific to that task. The fundamental monitoring metrics in TALOS are the number of queued records and the throughput of each task individually. 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 tested TALOS under various fluctuating workloads with high send rate in Kafka, demonstrating its effectiveness in dynamically adjusting task parallelism to maintain optimal performance across diverse and challenging workload scenarios. Essential observations from the performance evaluation indicate that TALOS dynamically adjusts the parallelism of tasks within the data processing pipeline, significantly enhancing both the throughput and resource utilization. TALOS scaling approach allows for a more tailored response to workload changes, leading to better overall system performance and effective response in over-provisioning scenarios, while conserving the system against sudden peaks. In the experiments, we compared TALOS with the Apache Flink Kubernetes Operator's Autoscaler that implements DS2 task scaling algorithm. DS2' primary objective is throughput optimization, without taking into consideration the cost. Furthermore, a limitation of DS2, one of the monitoring metrics, includes relying on the output rate of upstream operators. This dependency can lead to suboptimal decisions for a task in scenarios where data is split. Observing the experiments' results of Flink Kubernetes operator, scaling down operations are limited, even when necessary, as the algorithm's objective is enhancing throughput and that leads to overprovisioning. Consequently, this approach incurs higher operational costs over time due to the maintenance of high parallelism over time.

In conclusion, this study represents the effectiveness of TALOS in enhancing the scalability, performance, and cost-efficiency of Apache Flink-based applications. By strategically adjusting to workload variations, TALOS not only ensures high performance and reliability but also leads to a more sustainable utilization of computational resources. Nevertheless, we have to mention a downside of related to downtime, as adjustments in task parallelism can lead to brief periods of system unavailability, since the decisions follow the workload's fluctuations. Therefore, TALOS is not ideal for application that need 24/7 availability. However, long term applications that can tolerate periods of downtime and prioritize scalability, performance, and cost-efficiency in environments where workload fluctuations are common, TALOS is optimum.

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, TALOS has a conservative approach for scaling down actions, since decreases parallelism by one. A possible optimisation, is developing a scaling down formula, insomuch scaling down by one, in case of high current parallelism, can cause multiple scaling actions. Such an approach often leads to numerous scaling actions, which, in turn, can result in downtime. This aspect of the algorithm presents a challenge that needs addressing

- Developing techniques for efficient state management that can convene with the scaling algorithm. This includes exploring distributed data stores, in-memory computing, and checkpointing and savepointing mechanisms to ensure state consistency and quick recovery in the event of failures

- Available slots exploitation for scaling actions. Flink is in progress new mechanism, that is an extension of Adaptive scheduler. This extension greedily assigns all the resources that are all available in the cluster job and automatically rescales the job without the need of redeployment. This could be a good extension to TALOS, in order to avoid downtime in scaling actions, in case there are available slots, to handle the desired parallelism.

- Using Machine Learning techniques, statically choose the ideal parallelism. This agent can be trained in various workloads and choose the ideal parallelism for each task, before the application starts. Such an approach, avoids downtime during runtime, since there are dynamically scaling actions

- Using stand-by tasks. Using stand-by tasks, can erase downtime, in case of scaling up actions. Those tasks can be deployed as a separate job, with savepoints, in order to continue from the moment, that larger parallelism is needed. However, this can increase cost, since deploying tasks as separate jobs, results more task managers, i.e pods in a cloud environment

- While our current decision making agent is threshold-based, a more advanced reinforcement learning (RL) method could be implemented. While the initial stages of job execution rely on a threshold-based algorithm, such as TALOS, for determining optimized parallelism, the RL agent can be trained by these actions, after enough training, the agent will make the decisions. This can be called as adaptive decision-making mechanisms. This hybrid approach leverages the strengths of both strategies.

References

[1] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 783–798, USENIX Association, Oct. 2018.

[2] S. Liu, Y. Li, H. Yang, M. Dun, C. Chen, H. Zhang, and W. Li, "Qaas: quick accurate auto-scaling for streaming processing," *Frontiers of Computer Science*, vol. 18, Aug. 2023.

[3] A. N. Zafeirakopoulos and E. G. M. Petrakis, "Hyas: Hybrid autoscaler agent for apache flink," in *Web Engineering* (I. Garrigós, J. M. Murillo Rodríguez, and M. Wimmer, eds.), (Cham), pp. 34–48, Springer Nature Switzerland, 2023.

[4] B. Varga, M. Balassi, and A. Kiss, "Towards autoscaling of apache flink jobs," *Acta Universitatis Sapientiae, Informatica*, vol. 13, p. 39–59, June 2021.

[5] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang, "Meces: Latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, (Carlsbad, CA), pp. 539–556, USENIX Association, July 2022.

[6] P. E. Giannakopoulos, *Smilax: Statistical Machine Learning Autoscaler Agent for Apache FLINK*, p. 433–444. Springer International Publishing, 2021.

[7] Ververica Autopilot.

[8] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, "A comparison of reinforcement learning techniques for fuzzy cloud auto-scaling," 05 2017.

[9] B. B. Jv and D. Dharma, "Rlpas: Reinforcement learning-based proactive auto-scaler for resource provisioning in cloud environment," *Mobile Networks and Applications*, vol. 24, pp. 1–16, 08 2019.

[10] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and vertical scaling of container-based applications using reinforcement learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338, 2019.

[11] Apache Flink Documentation.

[12] Apache Flink Kubernetes Operator Documentation.

[13] Apache Kafka Documentation.

[14] Prometheus Documentation.

[15] Microservices Documentation.

[16] Kubernetes Documentation.

[17] Kafka Consumer Lag.

[18] Backpressure and Troubleshooting.

[19] Backpressure Representation .