FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automate and Control an Event-Driven Real-Time Stream for Big Data

Diogo Landau

WORKING VERSION

Mestrado Integrado em Engenharia Eletrotécnica e Computadores

Orientador: Jorge Barbosa

January 9, 2022

Resumo

Abstract

Agradecimentos

Diogo Landau

“
”

Someone

Contents

List of Figures

List of Tables

Symbols and Abbreviations

ETL	Extract, Transform, Load
API	Application Programming Interface
ISR	In-Sync Replica
w.r.t	with respect to
VM	Virtual Machine
AWS	Amazon Web Services
GBQ	Google Big Query
CPU	Central Processing Unit
KEDA	Kubernetes Event Driven Autoscaling
EKS	Elastic Kubernetes Service
ECS	Elastic Container Service
QoS	Quality of Service
MOP	Multi-Objective Problem
CLI	Command Line Interface
BPP	Bin Packing Problem
SBSBPP	Single Bin Size Bin Packing Problem
WCPR	Worst Case Performance Ratio
APR	Asymptotic Performance Ratio
DE	Data Engineering
PV	Persistent Volume
PVC	Persistent Volume Claim

Chapter 1

Introduction

HUUB is a company that aims to revolutionize the supply chain of fashion companies with a platform (Spoke) that delivers end-to-end logistic support, to any company that wishes to have a chance to grow globally.

As expected, providing this kind of solution to an unlimited amount of companies, will imply a data-driven solution, to unyielding real-time and scalability constraints. As such, the company is transitioning from a monolithic design into a microservices approach which responds better to the fast growing needs of the company.

Currently, 75% of production data is still gathered through their legacy service, whereas the remaining data is already fetched using their 5 microservices. The distributed systems communicate with one another using the Kafka messaging system, a highly scalable and available, event-driven platform that simplifies decoupling services when correctly applied.

To provide the information required for an in-depth analysis on the logistic process of an individual company, the Data Engineering team is responsible for Extracting, Transforming and Loading (ETL) all the relevant data, into a "single source of truth" data warehouse.

It is within this department that the thesis was developed, and it entails 3 important component's of HUUB's technical architecture: Kafka Cluster; Data-Engineering Consumer Clients; Google BigQuery (GBQ).

Currently the data-engineering team uses a single consumer client to consume messages from a Kafka topic with the sole purpose of populating a table in GBQ. There are 5 topics of interest for the data engineering team, and as such, there are only 5 consumers running on a single VM hosted by AWS.

This is the architecture that will be thoroughly analyzed throughout the thesis with recommended changes to certain components' process and the actual architecture itself to provide a scalable solution to the data-engineering team.

1.1 Motivation

As the company grows and more fashion brands look for global reach through HUUB, more data flows through the messaging systems, making it harder to comply with the near real-time requirements HUUB aims to provide. It is also the case that the peaks and valleys related to the quantity of data being produced grow farther apart, making it harder to justify a static solution.

Therefore, the Data Engineering team decided that the best approach would be a dynamic system, capable of evaluating the state of data production and consumption, scaling its services as needed, always taking into account the expenses.

To summarize, the proposed solution must:

- Consume data in near real-time, regardless of the amount of data being produced;
- Be reliable;
- Be monitored;
- Comply with Microservices;
- Be cost efficient.

1.2 Contributions and Structure

This thesis aims to provide a comprehensive review of the technologies and architectural patterns referenced throughout the thesis. The work related with the objective's specified by the data-engineering team envolves a comprehensive analysis of the whole pipeline from the Kafka Cluster to the insertion of the records into GBQ. As such, the following topics are described in detail in the literature review, to provide the reader with the required knowledge of what will be referenced throughout the thesis:

- Microservices
- Distributed Messaging Systems
- Containers
- Deploying Containerized Applications
- Optimization Problems and its applications

Posterior to introducing the necessary background, the methods combine all of the previous topics, into a single solution to solve a common problem in distributed systems which use a technical architecture similar to the one of HUUB's.

The results of the proposed solution is then presented and compared to the previous implementation at the company, followed by the conclusions.

Chapter 2

Literature Review

This chapter has the sole purpose of introducing and explaining the necessary information for a reader to be within context of the problem at hand.

The first section is an explanation of the architecture, followed by the communication methods between components, service deployment and lastly optimization methods.

2.1 Monolith Architecture

Within this type of architecture, all application concerns are contained in a single deployment.

It is usually the first approach to most product development due to its simplicity, but it is hard to ignore its disadvantages as it can easily become the bottleneck of any system.

When developing for this kind of architecture, a lot more care is given to how each part of the system communicates with one another, as communication is achieved through method or function calls.

This leads to very tight coupling between different application components, which inevitably provides a lot of friction to updating the codebase due to the high risk of it disabling the service entirely.

Like with most applications, with every new feature, the monolith grows, also increasing the computing resources required to run a single instance. In times of traffic peak, it is common for an application to scale its service allowing for response speed to remain approximately constant at all times.

The bigger the deployment, the longer it takes to make a new instance available and the more computing resources are used, making it more expensive to provide availability in times of high traffic.

With this type of system, technological choices have a huge impact on the final product, and have to be thoroughly planned in the beginning to guarantee that the system fulfills its requirements. It also makes it harder to adopt a different technology further into the development as the technological cost and time invested would be too much to even consider this option. This point alone, makes it hard to accept this kind of architecture, especially considering the frequency of new advances in this field, and it is only natural that a new approach to system architecture were developed.

2.2 Microservices

Although there is no formal definition, [**newman2015building**] states that “Microservices are small, autonomous services that work together.”

Small is of course a subjective measure, but in fact there is no "theoretical" boundary on this quantity. It depends on the context of the application and the business, but considering the size of these microservices, it is only logical that each of the small services follows some kind of team structure allowing for parallel development, without any real coupling between the services.

It is also stated in [**MartinFowlerMicroservices**] that Microservices are built around different “business capabilities” which are “independently deployable by fully automated deployment machinery”. In other words, each microservice can be deployed on its own, hence scaling services and reliability is "included" with this kind of architecture.

This makes updating code very low risk. As an example, in the scenario there is some flaw in a new deployment sent to production, it will only affect that microservice in particular, making it easier not only to track where the fault lies, but also guaranteeing that the remaining services are not made fully unavailable.

On the topic of scaling, with smaller decoupled services, only the service that requires scaling is in fact replicated, which leads to less computer resources utilization thanks to the reduced overhead of the service, resulting in a considerate cost reduction, and faster response to scaling needs.

With the possibility of running decoupled services interacting through lightweight network calls, each microservice can be programmed in whichever programming language and run the datastore that fits its needs best. The reduced size of each service, also makes technological changes and code refactoring a lot simpler, and can be done in a lot less time, compared to its counterpart.

With smaller services, it is only logical that team separation follows the lines that separate each microservice (business capabilities), leading to more diverse qualities within a single team/service, as mentioned in [**MartinFowlerMicroservices**]. When a new feature is to be developed for the service, the requirements rarely cross between teams making it faster to develop a single feature.

It is also worth mentioning that the separation between each service allows several representations of the same data in different ways depending on the service that is storing it. The benefits of this approach is not only in a projection perspective where there isn’t a need to over-engineer how the data will be stored for future purposes that still are not implemented, but it also helps model the world in the way that fits each service best.

This of course does not come without hardships, as maintaining data consistency between each service is one of the hardest tasks in this distributed data storage approach. On this note, we introduce the next section, which attempts to tackle this issue.

2.3 Distributed Messaging Systems

With a microservices approach, rarely do two components communicate with one another directly or through synchronous communication. If this were the case, the communication would become too convoluted with the increase of instances and services, becoming a source of coupling.

As stated in [**sharvari2019study**], it is clear the market needs are pointing to a messaging system that has the following features:

- Scalable - the system provides tools with which services can process a bigger load of messages in a time of intense volume of traffic;
- Space decoupling – The receiving and sending entity do not need to “know” each other;
- Reliable – The system can guarantee that the receiver has received the message;
- Asynchronous – The sender and receiver do not have to be active at the same time, and as soon as a message is produced the producer can go back to its tasks;

To enable decoupling between services, most messaging systems leverage the use of a broker, providing with the space and time decoupling, with the added benefit of it also being asynchronous, in the sense that after the broker acknowledges the reception of the message sent by the producer, the producer does not need to worry about its consumption allowing for it to go back to the tasks it has at hand.

There are two main approaches at solving this problem, event and message driven paradigms.

2.3.1 Message Driven

This paradigm makes use of message queues where producers send the data for it to be consumed by a single consumer in the same order it was appended to the queue. When data is produced, in case there isn’t a consumer interacting with the queue, the message is stored until read.

RabbitMQ is a well known platform commonly used for this purpose. All the previous features are provided, and on the note of scalability, a given service can connect to a queue with one or more consumers to increase consumption rate. This is done using round-robin dispatching [**RabbitMQscale**] which simplifies parallelizing work between instances, always guaranteeing that each message in a queue will be read by a single consumer.

2.3.2 Event Driven (Publish/Subscribe)

Messaging system which allows a producer to send a message to a certain category, which can be posteriorly retrieved by multiple consumers which are subscribed to the same category. This architecture not only decouples the services, it also allows many-to-many communication.

A common application for the event driven paradigm is for event sourcing. This is the pattern of storing any event that changes a systems state via an event object, onto an event log that preserves the sequence in which the events occurred. This was presented as an alternative to storing structures that model the state of the world, into storing events that lead to the current state of the world [**nadareishvili2016microservice**].

Comparing this scenario to the one of using a database, in the latter, every time the main database is updated, a second database recording the historic state has to be updated as well, storing the past states the system has been through. With event sourcing, when an event changes the state of the system, it is simply appended to the sequential event log, which in turn makes it less expensive to make operations of this type.

For these reasons, HUUB opted for this kind of architecture for their system, resorting to Kafka as their messaging system.

2.3.3 Kafka

Kafka functions as a distributed log running in multiple brokers that coordinate with each other to form a cluster.

2.3.3.1 Broker

A Kafka cluster is made up of one or more brokers, which function as servers where the messages can be published to, and consumed from.

This is the entity each client has to connect to in order to interact with other services. After a client connects to a single broker, it will also be connected to the remaining elements of the cluster.

2.3.3.2 Topics and Partitions

When data is published, the producer has to specify to which topic the record goes to.

A topic contains several partitions, and each partition can be stored in different brokers. Message order is only guaranteed within a single partition, and to insert a record consistently to the same partition, the same key has to be provided when inserting the message.

This consistency is only guaranteed while the number of partitions remains constant. As soon as a partition is added or removed, a message with the same key might end up in a different partition, but order will be guaranteed while the structure of the topic remains the same.

At the time of topic creation, there are three parameters that have to be provided:

1. **topic** - The name of the topic to create;

2. **partitions** - The amount of partitions the topic is subdivided in;

3. **replication-factor** - The amount of replicas of each partition to guarantee a reliable service.

Although the first two parameters are self-explanatory, the third is one of the most important features to guarantee a reliable service to a producer.

Assuming the scenario of choosing the replication factor of 2, this means that for each partition in the created topic, there will be 2 partitions that represent the same log in different brokers. At any given time, there is only 1 partition leader (broker), which leaves the other partition trying to keep up with the main log. When a replica is up-to-date with the leader, it becomes an in-sync replica (ISR).

In the scenario the partition leader fails unexpectedly, the partitions it is leading have to be reassigned a different leader. For each partition, the new partition leader will be one of the brokers that contains an in-sync replica of the same partition.

2.3.3.3 Producer

To publish data, producers have to specify one of the brokers from the cluster - which automatically connects it to all brokers in the cluster - and to which topic the record is to be sent.

The partition that belongs to the partition leader is where the messages are appended and read. To guarantee a message is delivered safely to the cluster, a producer might want to wait for acknowledgement. This is one of the configurations that allows for reliability when adding a message to a topic [**KafkaProducer**].

There are 3 possible values for this configuration:

- $acks = 0$ - Producer does not wait for acknowledgement, making it an unreliable configuration;
- $acks = 1$ - Producer waits for acknowledgement from the partition leader;
- $acks = all$ - Producer expects an acknowledgement from leader and all of its replicas.

After appending data to a log, it can no longer be changed (immutability).

When producing a message, if the order of a group of messages is important, Kafka provides a feature that allows messages to be consumed in the same order as they were produced. This is done by setting the key of a record to the same value, as a hash function runs over the key, and the partition where the message goes to is consistently the same, as long as the number of partitions does not change. If the number of partitions changes, then it is no longer guaranteed that same key messages will end up in the same partition as prior to the parameter being changed, only guaranteeing same key messages go to the same log from the moment the configuration changes and until it is modified.

To improve throughput when the volume of traffic increases, Kafka provides other configurations which allow a single producer to send messages in a batch, opposed to sending a message at a time. These configurations are:

- `batch.size` - Before sending a single record to the topic, the producer batches the messages (by partition), and when it reaches the size defined by this parameter, it proceeds with appending the records to its respective partitions.
- `linger.ms` - How long the consumer waits before sending the messages it has already batched.

2.3.3.4 Consumer Group and Consumer Clients

When connecting a consuming client to the messaging system, the topic(s) it wishes to subscribe to, the consumer group and at least one of the brokers from the cluster have to be provided (a connection to a single broker connects the consumer to all the other brokers in the cluster).

Messages are then read in order w.r.t a single partition, and in parallel w.r.t. multiple partitions.

Each consumer from within a consumer group, reads from exclusive partitions. This means that two consumers belonging to the same consumer group cannot be responsible for reading messages from the same partition, which in turn implies that the parallelism Kafka provides, is through the partitions in a topic. The amount of active consumers a single consumer group can have is therefore limited to the number of partitions in a topic [**OreillyConsumer**].

In order to maintain a reliable consumption service, whenever a consumer successfully reads a message and commits its offset, kafka internally stores the offset in a topic with the name `__consumer_offset` [**KafkaConsumer**].

There is another important functionality of this internal topic, which is related to the consumer group management. For a group to know when to rebalance it is important for the system to monitor the state of each consumer in the group, and this is performed by a group coordinator. To elect a group coordinator, the consumer group's id is selected as the message key, which is then hashed into a partition of this internal topic. The partition leader of that partition becomes the group coordinator.

For a consumer to remain a member of the group, it must periodically send heartbeats with a predefined interval to the coordinator. When the maximum amount of time without receiving a heartbeat is exceeded or the coordinator is notified of a consumer leaving the group, rebalancing is triggered, reassigning the partitions the consumer was responsible for to the remaining elements of its group.

The same rebalancing is triggered when a consumer starts up and requests to join a group. Rebalancing is simply attempting to split the load as equally as possible between the active consumers in a group, which also means approximately assigning an equal amount of partitions to each consumer in the same group.

Depending on the chosen configuration for the parameter `auto.offset.reset`, when a consumer receives its assignment from the coordinator, if there are no committed offsets by the consumer group's id for the topic-partition pair, then this policy is selected to determine where to start consuming records from its partitions. If it is set to `earliest` it will start consuming from the first message in the partition, whereas set to `latest` it will start consuming from the last message appended to that partition's log.

When data is consumed in batch, it can only be guaranteed that the data is read at least once. The reason why this is the case is due to the fact that if a consumer fails while processing the messages, before committing the offsets, the partition is reassigned and the same messages will be read again.

This implies the batch size is an important factor to take into consideration: the bigger the size, the more messages can be read more than once in cases of consumer failure.

While rebalancing, it is important to note that the consumers are not capable of consuming data from the partitions being rebalanced, making it an expensive operation which is to be avoided. If a certain consumer stops unexpectedly, it is no longer consuming from its partitions, and the coordinator has to wait for as long as `session.timeout.ms` for it to trigger rebalancing. This is a configurable value, but there is of course a trade-off. The bigger the value set, the less rebalancing is performed, but it will also take the coordinator longer to determine whether a single consumer is unavailable before triggering a rebalance.

2.4 Containers

When internet services first started, it was common to have the services running on local hardware. To handle peak traffic, scaling was performed by purchasing more hardware, which would then become unused when the traffic was no longer as high [**smith2017docker**].

Virtual Machines (VMs) became the next advance in this industry, and it allowed to use the resources of pieces of hardware more efficiently as multiple VMs could run on a single machine as long as there was space. This is when cloud service providers like Amazon, Google and Microsoft also started renting out VMs. In moments of peak traffic, a company could scale services in minutes, and there was no need to waste hardware resources with unused instances, therefore paying only for what is used.

It became clear that Virtual Machines also came with disadvantages, as there was a considerable overhead of memory to support the operating system of this environment.

Enter containers. Running instances could be done in a matter of seconds without the overhead of an operating system, which allows applications to run consistently in different environments as long as the containers are created resorting to the same image.

As stated in [**DockerContainer**], an image is a bundle of code/software which contains all the dependencies and libraries a given instance might need to run reliably in different computing environments. Container images become containers on runtime.

2.4.1 Kubernetes

Kubernetes works with a cluster of distributed nodes, that interact with one another to work as a single unit.

This service allows for an automated distribution and scheduling of containerized application. The level of abstraction causes a deployment to have no ties to a specific machine.

A kubernetes cluster is formed by 2 entities, the control plane, and nodes. The first is responsible for coordinating all activities in a cluster, such as scheduling, scaling and rolling out new updates of an application. The second, contains a process named kubelet, which manages the communication of the node with the control plane. There are additional tools like containerd or docker to allow the node to deploy containerized applications.

"In practice, a node is simply a VM or a physical computer that serves as a worker machine in Kubernetes" [**CreateKubeCluster**].

As soon as the Kubernetes cluster is running, it is possible to deploy the containerized applications. This is performed by creating the Kubernetes Deployment configuration specifying the number of instances and the image(s) with which to create the instances, followed by sending it to the control plane.

At all times, there is a deployment controller that continuously monitors the instances. In the scenario the node running a given instance fails, the controller detects the failure and launches the same instance on another node that belongs to the cluster. This is a consequence of the control plane attempting to maintain the state of an application that was defined in the deployment configuration.

Kubernetes offers a CLI, kubectl, which gives a user the possibility to communicate with the cluster locally. The communication is performed via the Kubernetes API, which is an API server located on the control plane. This server exposes an HTTP API, which allows differing entities to interact with it to query or even change API object's states [**KubernetesAPI**].

The Pod is Kubernetes' atomic unit. When creating a deployment, it creates as many pods as specified in the `replicas` value defined in the deployment configuration, which in turn holds the containers specified in the configuration as well under the `template` value. This is the template to be used when creating any pod that belongs to the deployment.

2.4.1.1 Autoscaling

As described in [**KubernetesAutoscaling**], the algorithm works in loop, constantly evalutaing the performance of the pods measuring the average CPU usage in the last minute. The control is performed by default every 30s, which can be modified by changing the value of:

- `--horizontal-pod-autoscaler-sync-period`.

The following equation determines the appropriate amount of pods for the average CPUUtilization to be below the target:

$$numPods = ceil\left(\frac{\sum_{p\in Deployment} CPUUtilization_p}{Target}\right) \tag{2.1}$$

where $CPUUtilization_p$ is the average CPU utilization in the last minute of a pod p.

To reduce noise in the metrics, the autoscaler can only scale-up if there was no rescaling within the last 3 minutes. The same applies to scale-down but the waiting time is 5 minutes.

Assuming a pod represents a Kafka consumer client, and a deployment a consumer group, to achieve maximum parallelism of consumption from a topic, the amount of pods in the deployment would be the same as the amount of partitions in the topic. Hence, when evaluating the performance of this deployment, specifically related to the usecase presented by the data-engineering team, where each consumer has to read messages from the topic and insert them into a GBQ table, what delays the message consumption from the topic, is the synchronous request made to

BigQuery. Consequently, autoscaling cannot be performed on CPUUtilization, as it would not increase with the increase in lag of the current offset w.r.t. the last message inserted in the topic.

Kubernetes Event Driven Autoscaling (KEDA), specifically provides a scaler which allows to increase the amount of consumers based on the average lag of a consumer group. Although this is definitely better than scaling based on the CPU usage, it still lacks granularity on evaluating the groups performance based on other metrics, providing as an example consumer group throughput and production throughput to the same topic. It also does not take into consideration the cost of releasing another instance, which will be one of the goals of the work developed throughout the thesis.

2.4.1.2 Kubernetes Operating Modes

When running Kubernetes, there are several alternatives. The first is to have the cluster run in bare metal. This is the option that allows most control over which type of node is added or removed from the cluster.

As soon as a node is manually added to the cluster, the control plane can start assigning deployment instances - or pods - to run in the node. When there is no longer any more space in the cluster to run other instances, more nodes have to be added to the cluster to allow the deployments to be correctly scheduled to maintain the state described in the deployment.

With this type of configuration, the payment is usually associated with the nodes that are added to the cluster. This can represent renting out VMs, or it can also be the price of buying the actual physical hardware running the necessary software to be a part of the Kubernetes cluster.

Another possibility, is to use some kind of cluster auto-management which is already included in a few cloud provider's Kubernetes Engine. Using as an example Google's Kubernetes Engine (GKE), there is a cluster mode of operation which is the autopilot, which manages the nodes within the cluster automatically without having to interfere manually with the cluster.

This mode of operation allows the user to pay by pod instead of node [**GKEautopilot**], meaning that only the resources that are being used are being payed for, making it as efficient as it could be with regards to pricing. Autopilot allows a truly dynamic and hands-off experience when having to deal with dynamic scaling a certain deployment.

2.5 Bin Packing Problem

The bin packing problem (BPP), as defined in the literature, is a combinatorial optimization problem that has been extensively studied since the thirties. Due to its NP-hard nature, on the practical side, several heuristic and metaheuristic algorithms have been proposed, as well Integer Linear Programming to determine the optimal solution. The former benefits from lower time execution to reach a solution which may not be optimal, whereas the latter provides the optimal solution, but requires more time to do so.

As specified in [**wascher2007improved**], there are several categories with which to define a BPP. The problem at hand is the one of a Single Bin Size Bin Packing Problem (SBSBPP).

As described in [**delorme2016bin**], an informal definition of this BPP, is: provided there are n items, each with a given weight $w_j\,(j = 1,...,n)$ and an unlimited amount of bins with equal capacity C, the goal is to arrange the n items in such a way that the capacity of each bin is not exceeded, and to determine the minimum amount of bins required to hold the n items.

Formally, the problem can be summarized as the following optimization problem:

$$\text{min} \quad \sum_{i\in B} y_i \tag{2.2}$$

$$\text{subject to} \quad \sum_{j\in L} w_j x_{ij} \leq C. \quad \forall\, i \in B \tag{2.3}$$

$$\sum_{i\in B} x_{ij} = 1, \qquad \forall\, j \in L \tag{2.4}$$

$$y_i \in \{0,1\} \qquad \forall\, i \in B \tag{2.5}$$

$$x_{ij} \in \{0,1\} \qquad \forall\, i \in B, j \in L \tag{2.6}$$

x_{ij} represents whether an item j is packed in the bin i, whereas y_i determines whether bin i is used. As for the constraints, 2.3 assures that the sum of the items in a given bin, does not exceed it's capacity, and 2.4 makes sure that every item is assigned a bin.

Set B corresponds to the available bins, and L as the list of items to be arranged into different bins.

It is also common to represent the bin-packing problem in its normalized version, where all the weights are down-scaled by the total capacity of a bin, and the capacity of each bin is 1. This implies modifying 2.3 into:

$$\sum_{j\in L} \hat{w}_j x_{ij} \leq 1. \quad \forall\, i \in B, \tag{2.7}$$

where $\hat{w}_j$ represents the normalized weight of an item w_j/C.

This approach generalizes the analysis of the bin packing approach, and unless stated otherwise, it will be the model used throughout this analysis.

Another important note, is that "weight" and "size" will be used interchangeably when analyzing the following algorithms.

2.5.1 Linear Programming

When interested in achieving the optimal solution, the textbook implementation of the BPP model presented in (2.2) is not computationally efficient. A common approach is to study the Upper and Lower Bounds of the Algorithm, and to add valid additional constraints to restrict the search space.

In [**delorme2016bin**], there is an extensive review on the state of the art algorithms that have been developed to solve the ILP problem, comparing several models and their efficiency when solving the same set of problems.

Due to the dynamic nature of the problem that will be further described in 3, when solving the BPP, there will already exist a given arrangement as for which bin each item is assigned to, provided by previous iterations of the algorithm. Because the items change size between iterations, the algorithm has to be run again to determine whether there is a better configuration, not only to attempt to minimize the amount of bins used, but also to determine whether the capacity of a bin is being exceeded. For this reason, approaching this problem using the Linear Programming approach, would completely disregard whether a certain item is assigned to a different bin other than the one it is currently in. For this reason, the work elaborated throughout this thesis will have more emphasis on Approximate algorithms, where there is more control over the bin assignment rules.

2.5.2 Approximation Algorithms and Heuristics

During this section, the method presented in [**coffman2013bin**] to classify a BPP problem will be employed throughout the analysis of the algorithms.

There are 2 possible states a bin can find itself while executing the algorithm, *open* or *closed*. In the former, the bin can still be used to add additional items, whereas in the latter, they are no longer available and have already been used.

In the literature, it is common to present a parameter α which indicates the size limit of the weights within the list L, which can vary between $(0,1]$. Considering the problem that will be presented in 3 does not limit the size of the weights - other than being smaller or equal to the bins capacity -, we will consider $\alpha = 1$.

To compare the performance between algorithms, the Asymptotic Performance Ratio (APR), is what will be employed throughout this chapter. $A(L)$ denotes the amount of bins a certain algorithm makes use of for a configuration of items L. $OPT(L)$ is used to represent the amount of bins required to achieve the optimal solution. Defining Ω as the set of all possible lists, each with a different arrangement of their items, given

$$R_A(k) = \sup_{L \in \Omega} \left\{ \frac{A(L)}{k} : k = OPT(L) \right\} \tag{2.8}$$

the APR is

$$R_A^{\infty} = \limsup_{k \to \infty} R_A(k). \tag{2.9}$$

Algorithms can be classified into several classes, as clearly specified in [**coffman2013bin**], but the only classes that are of interest for this problem are:

1. online - Algorithms that have no holistic view over any other item in the list except the one it is currently assigning a bin to. It is also a requirement that a bin is assigned to the item as soon as it is analyzed.

2. bounded-space - Sub-class of online algorithms, where the amount of bins open at a single instance is limited by a constant provided prior to its execution.

3. offline - The algorithm is aware of all the items in the list before assigning bins to each item, and so their ordering does not directly impact the outcome as it is not necessary to respect the list's initial order.

2.5.2.1 Online Algorithms

As previously mentioned in 2.5.2, without an overview of all the items within a list, whenever an item is analyzed it has to be assigned a bin. This also implies that the bin in which the current item will be inserted is a function of the weights of all the preceding items in the list.

This section will analyze Any fit, Almost any fit, bounded-space and reservation technique algorithms.

Throughout the following sections, when *current item* is mentioned, it is referring to the next item to assign a bin to. After assignment, the item that follows it in the list, is then considered the *current item*.

Next Fit (NF) is one of the simplest algorithms developed to solve the bin packing problem heuristically, and it consists in the following procedure. The current bin is considered to be the last non-empty bin opened. If the current bin has space for the item then the item is inserted. Otherwise, the current bin is closed, a new one is opened, and the item is inserted. The next item on the list is then considered the current item.

With regards to time complexity, NF is $O(n)$. And because there is only one open bin at a time, this is a bounded-space algorithm with $k = 1$. As proven in [**johnson1973near**],

$$R_{NF}^{\infty} = 2. \tag{2.10}$$

Worst Fit (WF): For each element, it looks for the open bin that has the most space for the item where it fits, and inserts the item on that bin. If there is no open bin that can hold the item, then a new bin is opened, and the item is inserted. Having no closing procedure, this algorithm is unbounded, and it does not run in linear time.

Although it is expected that this algorithm would perform better than the NF, in the worst-case performance analysis it does not gain any benefits from not closing its bins, as stated in [**man1996approximation**]

$$R_{WF}^{\infty} = R_{NF}^{\infty}. \tag{2.11}$$

First Fit (FF) searches each open bin starting from the lowest index, and the first bin that has the capacity to fit the item, is where the item is inserted. If there is no open bin where the item can be inserted, then a new bin is opened where the item is inserted. [**johnson1974worst**] showed that

$$R_{FF}^{\infty} = \frac{17}{10}. \tag{2.12}$$

There is a more general class of algorithms presented as *Any Fit* (*AF*) in [**johnson1974fast**]. This class' constraint is that if BIN_j is empty, then no item will be inserted into this bin if there is any bin to the left of BIN_j that has the capacity to contain the item. In the same paper it is also shown that no algorithm that fits this constraint can perform worst than the WF, nor can it perform better than the FF, always with respect to the asymptotic performance ratio. Then

$$R_{FF}^{\infty} \leq R_{A}^{\infty} \leq R_{WF}^{\infty}, \, \forall \, A \in AF \tag{2.13}$$

Best fit (BF): Attempts to fit the item into any of the open bins where it fits the tightest. If it doesn't fit in any, then a new bin is created and assigned to the item. As can be seen, it is similar to the worst fit, with exception to the fitting condition, differing only in the fact that the item is inserted where it fits the tightest, and not where it leaves most slack.

As it happens, BF also belongs to the AF class, and it performs as well as the First Fit

$$R_{BF}^{\infty} = R_{FF}^{\infty}. \tag{2.14}$$

In [**johnson1974fast**], another class of algorithms presented, which is also a subclass of AF, is *Almost Any Fit* (AAF). This class has as constraints: *If BIN_j is the first empty bin, and BIN_k is the bin that has the most slack, where $k < j$, then the current item is not inserted into k if it fits into any bin to the left of k.* One of the properties proven in the same paper, is that

$$R_{A}^{\infty} = R_{FF}^{\infty}, \, \forall \, A \in AAF, \tag{2.15}$$

which is to say that any algorithm that fits the previous constraints, have the same APR as the FF algorithm. Focusing on the constraints, it is clear to see how BF and FF belong to this class.

Due to the constraints presented in the AAF class, an improvement for the WF algorithm arises wherein the current item is inserted in the second bin with most space, instead of the first. This algorithm is the Almost Worst Fit (AWF), and with this simple change, now belongs to the AAF class, having as APR

$$R_{AWF}^{\infty} = R_{FF}^{\infty} = \frac{17}{10}. \tag{2.16}$$

2.5.2.2 Bounded-space

Bounded-space algorithms have a predefined limit on the amount of bins that are allowed to be open at a given instance, which will be defined as k. It is also a subclass of the online algorithms.

An example of a bounded-space algorithm is NF, as it never has more than a single open bin at a given instant. The other presented online algorithms can also be adapted into a bounded space algorithm, simply by specifying a procedure as to which bin to close before exceeding the limit.

A bounded-space algorithm can be defined via their packing and closing rules [**coffman2013bin**]. A class that derives from rules based on the FF and BF can be defined in the following manner:

- Packing - When packing an item into one of the available open bins, the selected bin either follows a FF or BF approach.

- Closing - When choosing which bin to close, if following the FF approach, then the bin with the lowest index is closed. If following the BF it's the bin that is filled the most that is selected as the one to be closed.

The notation for this class of algorithms is AXY_k, where X represents the packing rule, and assumes the letters F or B, and Y which can either be a F or a B, refers to the closing rule. The k represents the maximum amount of open bins allowed.

Next-k-fit applies both packing and closing rules based on the first fit algorithm, and as expected when $k \to \infty$ it approximates the FF algorithm, having as APR $17/10$. For variable k, as shown in [**mao1993tight**]

$$R^{\infty}_{AFF_k} = \frac{17}{10} + \frac{3}{10k-10}, \quad \forall\, k \geq 2. \tag{2.17}$$

Best-k-fit, which is also known as the ABF_k algorithm, has been proven in [**mao1993besk**] to have

$$R^{\infty}_{ABF_k} = \frac{17}{10} + \frac{3}{10k}, \quad \forall\, k \geq 2. \tag{2.18}$$

As for AFB_k, [**zhang1994tight**] showed that this algorithm has

$$R^{\infty}_{AFB_k} = R^{\infty}_{AFF_k}, \quad \forall\, k \geq 2. \tag{2.19}$$

For the last possible combination of this class of algorithms, ABB_k has been proven in [**csirik2001bounded**] to have

$$R^{\infty}_{ABB_k} = \frac{17}{10}, \quad \forall\, k \geq 2 \tag{2.20}$$

which surprisingly indicates that the value of k (as long as it's bigger than two) has no effect on the APR metric of this algorithm, contrary to all the previous algorithms.

The next algorithms make use of a reservation technique that proved to be very useful to break the lower bound of the APR of the Any Fit class of algorithms. The first algorithm to be developed of this type was the Harmonic-Fit (HF_k) which makes use of k to split the sizes of items into k different intervals.

I_j denotes the interval of sizes with index j, and is within the range

$$\left(\frac{1}{j+1}, \frac{1}{j} \right], \quad \forall\, j \in \{1, ..., k-1\}. \tag{2.21}$$

I_k is defined as the interval from $(0, 1/k]$.

B_j is used to classify the bin type which is responsible for holding items of type I_j.

[**lee1985simple**] presents the APR of HF_k in the following manner. If k denotes the maximum amount of open bins allowed at once, then the asymptotic performance ratio can be shown with respect to k in the following manner:

$$t_1 = 1 \quad and \quad t_{i+1} = t_i(t_i+1) \quad \forall\, i \geq 1 \tag{2.22}$$

$$\frac{1}{t_i} = 1 - \sum_{j=1}^{i-1} \frac{1}{t_j+1} \tag{2.23}$$

$$t_i < k \leq t_{i+1} \quad for\, i \geq 1 \tag{2.24}$$

$$R^{\infty}_{HF_k} = \sum_{j=1}^{i} \frac{1}{t_j} + \frac{k}{t_{i+1}(k-1)} \tag{2.25}$$

When $k \to \infty$, then $R^{\infty}_{HF_k} \approx 1.6910$. It is also important to note than to obtain a better APR metric compared to the other online bin packing algorithms, HF_k achieves this with $k \geq 7$, as is shown by [**lee1985simple**].

Posterior to this technique being presented, it was clear that the reservation technique could be a good approach to improve the performance of the Any-Fit algorithms, and as such, several other algorithms have been created around this technique, that achieve even better APR's than HF, but because these techniques all involve assigning item types, based on their size, to their respective bin type, this approach is not applicable to the problem, as the control over item rebalancing is reduced, which will further be described in the following chapter.

2.5.2.3 Offline Algorithms

Comparing with the previous section, offline algorithms have the added benefit that all items are known prior to its execution. As long as the list of items remains the same, items can be grouped, sorted or anything that might be convenient for the algorithm that is going to execute over the list of items.

It is important to note that as soon as an algorithm chooses to sort the list of items, it automatically implies that the algorithm no longer runs in linear-time as the sorting algorithm would have a time complexity of $O(nlog(n))$.

Most of the Any fit algorithm, perform best if the list of items is sorted in a non-increasing order. The following three algorithms remain the same as for how the packing is done when analyzing the list of items, with the exception that before running the algorithm, the list is sorted.

Provided the list is sorted in the aforementioned manner, the **Next Fit Decreasing (NFD)** has a considerable improvement in terms of it's worst-case performance, and performs slightly better than FF and BF, as presented by [**baker1981tight**]

$$R^{\infty}_{NFD} \approx 1.6910. \tag{2.26}$$

As can be seen in [**johnson1974worst**], **Best Fit Decreasing (BFD)** and **First Fit Decreasing (FFD)** improve the APR metric with presorting compared to their online versions of the same algorithm

$$R^{\infty}_{BFD} = R^{\infty}_{FFD} = \frac{11}{9}. \tag{2.27}$$

Another algorithm which is worth mentioning within this class of offline algorithms is the **Modified First Fit Decreasing (MFFD)**. As shown in [**johnson19857160**], The APR is

$$R^{\infty}_{MFFD} = \frac{71}{60}, \tag{2.28}$$

which is achieved by initially presorting the items in a decreasing manner and grouping each item into seven distinct groups based on the item's size.

Table 2.1: Item size interval which guides the grouping for the MFFD algorithm.

Group	Item size interval
A	$(1/2, 1]$
B	$(1/3, 1/2]$
C	$(1/4, 1/3]$
D	$(1/5, 1/4]$
E	$(1/6, 1/5]$
F	$(11/71, 1/6]$
G	$(0, 11/71]$

After doing so, the algorithm then performs the following five steps:

1. For each item that belongs to group A, from biggest to smallest, assign it a bin with the same index as the item has within it's group. When this process terminates, there are as many bins as items in group A and the bins created are $B_1, ..., B_{|A|}$.
2. Iterating over the existing bins from left to right, for each bin, if any item in group B fits in the bin, insert the biggest such item in the current bin.
3. Iterating through the list of bins from right to left, for each bin, if the current bin contains an item that belongs to group B, do nothing. If the two smallest items in $C \cup D \cup E$ do not fit, do nothing. Otherwise insert the smallest unpacked items from $C \cup D \cup E$ combined with the largest item from $C \cup D \cup E$ that will fit.
4. iterate over the list of bins from left to right, and for each bin if any unpacked item fits, insert the largest such item, repeating until no unpacked item fits.
5. Lastly, the remaining items that did not fit in bins $B_1, ..., B_{|A|}$ are inserted in an FFD fashion starting with a new bin $B_{|A|+1}$.

2.6 Conclusion

As presented throughout this chapter, the work that will be developed within this context involves several technologies that are constantly evolving.

Kafka presents a great solution to decoupling services and allowing asynchronous communication between microservices. Other features that Kafka is designed for is scalability, which allows the services to create more replicas of consumers within the same consumer group for faster data consumption. It is important to note that Kafka does not provide autoscaling, just the support for distributed data consumption.

To enable the scheduling of distributed containers, Kubernetes is the most common tool for this purpose. The more support cloud providers develop for Kubernetes, the less manual interaction is required to add or remove nodes based on cluster use. This not only simplifies the cluster management, but it also automates cost optimization as the payment becomes per pod, instead of paying for the manually inserted node. For this reason, the autopilot tool provided by GKE is what will be used to provide a truly dynamic approach to consumer group autoscaling.

Kafka's rebalancing algorithm attempts to balance the load between the consumers within a group by assigning approximately the same amount of partitions to each consumer. Because the partitions assigned do not all have the same write speed ($bytes/s$), it is not guaranteed that the consumer will in reality have the load balanced between them.

This provides the basis for the work that will be developed. The objective is to attempt to minimize the cost of a consumer group (the number of instances running), while at the same time attempting to make sure that the group does not get behind in the messages it has to read from a topic.

This approach is only possible because there is a general view of the maximum data rate of consumption of a single consumer within the studied context - although it can also be applied in other scenarios as long as there is a predictable model for a single consumer -. The next chapter will specify the consumer model, and formulate the problem as a BPP.

In more detail, the aim is to try and minimize the amount of consumers active at once, while simultaneously making sure that the sum of the write speed of the partitions assigned to a single consumer does not exceed it's maximum data consumption capacity, and attempting to minimize partition redistribution between consumers. The last condition is due to the fact that assigning a partition to a new consumer briefly pauses the consumption so another consumer can pick up where the previous left off.

This consumer model is what allows to model the problem as a BPP. Approximation algorithms were considered the most appropriate approach as they are the algorithms that allow most control over the partition redistribution with least modification. This is also the reason why the Linear optimization approach will not be implemented.

Chapter 3

Kafka Consumer Group Autoscaler

The Data-Engineering (DE) team at HUUB is responsible for making a pipeline that inserts data from several data sources into a data lake so it can then be transformed and loaded into a single source of truth data warehouse.

HUUB's current architecture makes use of event sourcing to announce events that change the system's state within a microservice. Prior to this architecture, it was common to fetch the information from several different databases which were of interest for this department, periodically and in batches, which lead to tight coupling between the work performed by the different teams at the company.

Based on reporting and real-time data needs, with regards to the extraction process, if there is new data that has to be consumed from a service that isn't currently being consumed, Kafka and event sourcing allow this addition to be as simple as subscribing the DE consumer to the topic of interest.

The work developed throughout this thesis, is based on optimizing the pipeline from Kafka into bigquery, by providing a dynamic solution that makes use of Kafka's parallelism without disregarding the cost of the number of instances deployed.

The goal is to achieve a deterministic approach as for the number of consumers required working in parallel, so as to guarantee that the rate of production into a topic is not higher than the rate of consumption, which would lead to messages being accumulated in a topic making the lag increase between the last message inserted, and the last message read by the consumer group.

A Kafka topic is subdivided into several partitions, distributed within the several brokers of the Kafka cluster. When sending a message into a topic, if a key is provided, then the message will consistently end up in the same partition, whereas if none is provided, then the message is inserted in a round-robin fashion in one of the partitions of the same topic. Messages going into a topic may also have different sizes ($bytes$), and for these reasons, the write speed ($bytes/s$) to the several partitions in a topic is not guaranteed to be the same.

The following model, also assumes that the maximum consumption rate of a single consumer is constant, and if there is enough data to be consumed, this is the speed the consumer will find itself when working "full throttle". This is elaborated in section 3.2.5.1 based on several tests performed to determine this value for a consumer.

Based on the previous information, the model for this pipeline is considered to fit the constraints for the SBSBPP, where the consumer is the bin that has as capacity its maximum consumption rate, and the weights are the partitions and their respective write speeds. The problem then is to find the minimum amount of consumers where to fit all the partitions so as to make sure that the sum of the write speeds of the partitions assigned to a single consumer do not exceed its maximum capacity.

For the first iteration, the optimal solution finds the arrangement between the partitions and the consumers that minimizes the amount of instances, but this solution is not static, as the write speeds of the partitions change with time. As such, there is a second factor to take into consideration which is the partition reassignment.

When a partition is reassigned, because 2 consumers from the same group cannot be consuming from the same partition at the same time, when assigning a partition to another consumer, the one it is currently assigned to has to stop consuming in order to allow the new consumer to start. Due to this process, there is some downtime where data is not being consumed from a partition.

For this reason, making use of an optimal algorithm that also minimizes partition redistribution, is not feasible as it would not run within the necessary time requirements due to the NP hard nature of the problem.

To determine the arrangement of partitions and consumers, this work is based on the studied approximation algorithms which have already been mentioned in the literature review. As is clear, there is no approximation algorithm which considers item redistribution. The reason for this may lay on the fact that the context where this problem lies is not very common, and there is no mention of a stateful assignment where each item is already assigned a bin prior to executing the algorithm.

The following sections aim to present the details of each component that is required to model this problem as a BPP and make it into a dynamic system capable of adapting based on the amount of data being produced to the topics of interest.

3.1 Monitor

As mentioned in 3.3 the algorithm requires a measure of the write speed of all the partitions within the topics of interest.

If enabled, Kafka exposes JMX metrics for a java process to consume from, providing with a complete monitoring tool. Prometheus is a wrapper to these JMX metrics as it creates a unified model to query for the data, and it also registers a timeseries of the metrics, allowing for a historic view of any parameter, as well as creating more elaborate metrics from the exposed cluster metrics.

Due to the specificity of the required metric, which is the write speed to a partition of a single topic, to the best of my knowledge, neither of the aforementioned tools, nor any other tool available provides this information with regards to the cluster.

Algorithm 1: Monitor process pseudo-code.

```
input: admin - Kafka admin client to communicate with cluster,
       producer - Kafka producer client,
       topics - list of topic identifiers from which the monitor is to determine the speed of their partitions
measurementQueue ← Queue<Measurement>();
while true do
    measurement ← new Measurement();
    measurement.partitionSizes ← admin.getLogDirs(topics);
    measurement.timestamp ← currentTime();
    measurementQueue.add(measurement);
    if (measurementQueue.first.timestamp - measurement.last.timestamp) > 30s then
        sizeDiff ← measurement.last.partitionSizes - measurement.first.partitionSizes;
        timeDiff ← measurement.last.timestamp - measurement.first.timestamp;
        measurementSpeed ← sizeDiff / timeDiff;
        producer.producer("data-engineering-monitor", measurementSpeed) while
          (currentTime() - measurementQueue.first.timestamp) > 30s do
            measurementQueue.pop();
        end
    end
end
```

For this reason, this section aims to explain the process that is responsible for monitoring the write speed of the partitions of interest, so as to provide with the input data for the algorithm to run the heuristic or to be used to trigger the execution of the algorithm to reassign partitions within the group.

Kafka provides an Admin client, which can be used to administer the cluster, and also query information about it. This client/class exposes a method `describeLogDirs()` which queries the cluster for the amount of bytes each TopicPartition has. A TopicPartition is a string-integer pair, which identifies any partition (integer) within a topic (string).

Since the consumer only consumes from the partition leader, if the `replication-factor > 1`, then several partitions are excluded from the result of the previous method call, since this process is only interested in the partitions that belong to one of the topics of interest, and which are leaders.

The monitor process is responsible for periodically polling for the amount of bytes of each partition, and associating a timestamp with the data, which is then stored in a queue. Each time the partition size is queried by the admin client, a timestamp is appended to the measurement, and it is inserted to the back of the queue. Any query that is older than $30s$, which is guaranteed to be in the front of the queue, is removed. To obtain the write speed of a single partition, the last element of the queue and the first (representing the latest and the earliest measurement of the partition size within the last $30s$) are used to compute the ratio between the difference in bytes and the difference in time ($bytes/s$). This is also the average write speed over the last 30 seconds.

Every topic has the parameters `retention.ms` and `retention.bytes`, which determine how long a record remains in a partition before being deleted, or how many bytes a partition retains before removing old records. When a record is deleted, the partition size reduces as it no longer reserves space for the removed record. For this process to have an accurate write speed, it is important to set both of these parameter to `-1`, which means that the record is never deleted from the partition. Ideally, the admin client would be able query for the historic size of a partition, but since this information could not be retrieved, this approach was what was implemented.

After computing the write speed for all the partitions of interest, the message has to be communicated to the controller/orchestrator that runs the algorithm to assign the partitions to the consumers. To benefit from an asynchronous approach, this monitor process communicates with the controller/orchestrator process via a kafka topic named `data-engineering-monitor`. The data to be inserted is the write speed of the partitions of interest, which is then consumed by the controller/orchestrator to be used as input for the algorithm's execution, or to trigger the execution of the algorithm.

Two testing scenarios were developed to illustrate the measured partition write speeds by this method when a controlled producer is sending records at a predefined rate. The payload inserted into the partition is always 123 bytes.

The first scenario has the producer send the message at 3 different rates for approximately 35 seconds each. As can be seen in 3.1, the monitor increases the measured write speed rate slower than the actual speed as measured by the producer, since the monitor takes into consideration the first and last measurement made in the last 30 seconds, whereas the producer simply measures the produce rate since the last time it inserted a record into the partition. It takes the monitor 30 seconds to converge to the actual production speed where it then settles until the produce rate increases again.

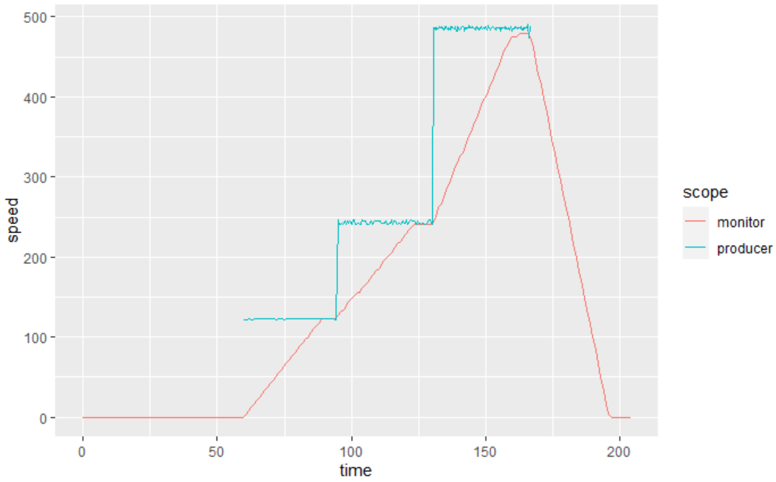


Figure 3.1: Monitor step response to three different controlled write speeds.

The second scenario has the producer wait a randomized time interval (between $0.01s$ and $2s$) before sending the message to the partition. Since the monitor takes into consideration the last $30s$ since it's latest measurement, it is not affected by the noisy write speed as measured from the producer. Since the implementation has the time interval as a uniform random variable, when the monitor process stabilizes, it should stabilize on a speed rate as computed using the time interval's expected value which is $\approx 1s$. This is in fact the case as can be seen in 3.2, since the write speed stabilizes around $123bytes/s$. The statistical description of the producer's measured write speed better illustrates the noise reduction due to the monitor computing the average over the last $30s$ of each measurement.

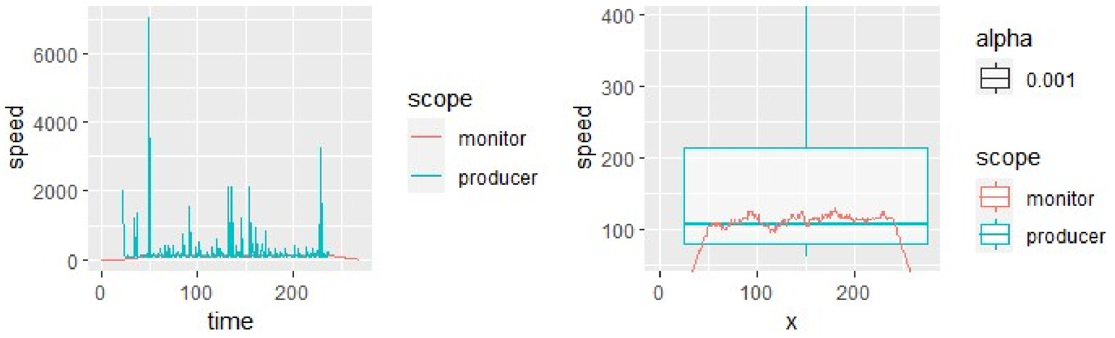


Figure 3.2: Monitor response to a producer waiting uniformely between [0.01,2]s every time it produces a record.

3.2 Consumer

The Consumer goes through four important phases within it's process, to approximate it's consumption rate to a constant value when being challenged to work at it's peak performance. These phases repeat cyclically until the consumer is terminated by an external termination signal, or in the event of an unexpected exception.

3.2.1 Phase 1: Gathering Records

The first phase is where data is gathered before being sent to the data warehouse.

The consumer is configured with two important parameters, `BATCH_BYTES` and `WAIT_TIME_SECS`, which indicate respectively, the amount of bytes the consumer waits to gather in a single iteration, and the amount of time it is allowed to wait to gather the information.

3.2.2 Phase 2: Processing

The second phase is where the data gathered in the previous phase is prepared to be sent into bigquery. Each record fetched from a Kafka topic, represents a new row in a bigquery table.

To insert data into bigquery, Google's Python Bigquery Client is used. Each post request made with this client, has a batch of rows sent to a single table, and it is also limited to no more than 10MB per request.

Since the gathered records originate from multiple topics, each row has to be batched with data of it's kind, which are rows that are intended to be sent to the same bigquery table.

Algorithm 2: Consumer Phase 2 algorithm

```
input :
        messages - List of Kafka messages,
        mapTopicsTable - Map that indicates which table a message from a topic is
inserted into
output: List<BatchList>

mapTableBatchlist ← new Map[String, BatchList]();
for msg in messages do
    table ← mapTopicTable.get(msg.topic());
    batchList ← mapTableBatchlist.get(table);
    if (batchList == None) then
        mapTableBatchlist[table] ← new BatchList();
        batchList ← mapTableBatchlist.get(table);
    end
    batchList.add(msg);
end
return mapTableBatchlist.values()
```

A single instance of a Batch, a data structure created to agglomerate rows intended for the same bigquery table, holds all the rows which will be sent in a single post request through Google's python bigquery client API. Due to the imposed limit of a post request, a Batch has a payload limit of $5Mbytes$.

A BatchList, is another data structure created to group instances of type Batch that refer to the same table.

To process the data, a map stores the link between a table's id and the BatchList instance assigned to it. When analyzing a message, it's topic metadata indicates which table it is directed to, which in turn points to the BatchList it must be added to.

When adding a message to a BatchList instance, this class controls to which Batch it is assigned, based on the size of the row, and the size of the last created Batch within this data structure.

3.2.3 Phase 3: Sending Data into Bigquery

This is the final stage of the consumer's insert cycle. There is a list of BatchList instances, each having to be sent to a single bigquery table. To optimize the insert cycle, this phase is done asynchronously with respect to all the batches within the list of BatchLists contained in the map specified in 3.2.2. In other words, each batch corresponds to an asynchronous request to insert rows into a table.

After successful insert of the rows into the bigquery table, the messages are then committed to Kafka.

Algorithm 3: Consumer Phase 4 algorithm

```
input  : consumer - consumer instance used to fetch information from the Kafka topics.
output:

Set<Partition> current ← consumer.getCurrentState();
Set<Partition> future ← current.copy();
Queue< Set<Partition> > changeStateQueue ← consumer.consumeMetadata();
while changeState ← changeStateQueue.pop() do
    if changeState.type == "StopConsumingCommand" then
        future ← future - changeState.partitions;
    else
        future ← future ∪ changeState.partitions;
    end
end
toAssign ← future - current;
toStop ← current - future;
consumer.incrementalAssign(toAssign);
consumer.incrementalUnassign(toStop);
```

3.2.4 Phase 4: Consumer Metadata

The consumer has to be informed by the Controller 3.3, as to which partitions it gets data from. For this purpose, to allow both the controller and the consumers to work asynchronously, a message queue is the ideal structure for this purpose.

Following event sourcing patterns, the current state of a single consumer should be attained by consuming every message that was directed to it, which enforced a change in state. With a slight modification, Kafka is used for this communication between the controller and the consumers by creating a single controller topic named `data-engineering-controller`.

Maximum efficiency in data conveyed between each process, occurs when a single process only reads messages which are relevant for it's functioning, without having to ignore messages or data that it receives. As such, each consumer has to be assigned a separate queue in the `data-engineering-controller` topic, which in Kafka represents a distinct partition per consumer.

As will be further described in 3.3, the consumer knows which partition to consume from through it's deployment's name. When the controller desires to communicate with this consumer, it simply has to send a record to it's partition.

Each record published into this topic has to have the same AVRO schema as specified by A.1. A command can be either a:

- StartConsumingCommand - The consumer is to start consuming from each of the partitions within the record.
- StopConsumingCommand - The consumer stops consuming from the partitions specified in the record.

The fourth phase starts with the consumer determining whether there are any messages in it's metadata queue (the partition it was assigned from the `data-engineering-controller` topic). If there are none, the consumer's state isn't changed, and the phase is finalized. Otherwise, if there are any messages in the queue, the consumer goes through the process of consuming all the messages in the metadata partition, and adds them to a Queue instance `changeStateQueue`.

The next step is to process the `changeStateQueue`. Initially, a set is created where each element represents a partition from a topic the consumer is currently consuming from. This set is stored in the variable `current`. A copy of this set is made and assigned to a variable `future`. While the queue is not empty, the front-most element is removed and assigned to a temporary variable `changeState`. Depending on the command, if the record requests the consumer to start consuming from it's set of partitions then a union operation is performed between the `future` and the `changeState`. In turn, if the record is of type "StartConsumingCommand", the difference between future and change state is computed.

Having processed the whole `changeStateQueue`, future holds the new state the consumer has to change into. Lines 11 to 14 change the consumer's assignment into future's state.

3.2.4.1 Persisting Metadata

The final stage of the consumer goes through persisting the consumer's metadata in case the consumer fails and has to pick up the work it was last performing. The data to be persisted is the set of partitions it is currently consuming from. This means that a successful change in consumer metadata would only occur after the consumer successfully persists the data to it's persistent volume.

Since the consumer is a pod within a deployment in a Kubernetes cluster, the container responsible for this component starts with a clean slate. When the container is terminated the data written to disk is cleaned, making it inaccessible for future reads outside of a single pods lifetime. This type of storage is named ephemeral as it has the same lifetime as the pod.

Kubernetes provides with volumes which can be of type ephemeral or persistent (it's lifetime is independent of the pod's).

A persistent volume (PV) can be created static or dynamically, and is a resource within the cluster. A persistent volume claim (PVC), is a method of abstraction which allows a user to request for storage. This request, then tries to match the claim to one of the available resources, and if there are none available, then a new persistent volume can be created dynamically if the Storage Class is defined. The mapping between PV and PVC is one-to-one.

This consumer uses both types of volumes, since the downwardAPI is of type ephemeral and it provides the consumer with it's context, giving it access to it's deployment's name (data the consumer requires for it to know which partition to consume from in the data-engineering-controller [metadata] topic). As for the persistent volume, the consumer will use this type of volume to persist the data for it's current consumption state.

If the consumer fails unexpectedly, then on startup, it just has to verify it's state on the volume, and in case it was performing any tasks, it picks up where it left off.

When stopping a consumer, to safely terminate it's tasks, the controller sends a StopConsumingCommand for all the partitions the consumer is currently assigned to. After the consumer acknowledges it acted to the command (it sends a StopConsumingEvent back to the controller 3.3.5.3), the controller then terminates the pod. The fact termination only occurs after the consumer updates it's metadata, implies the persistent volume is also updated to an empty set, which allows a new consumer of the same deployment to start off with a clean slate as should be the case, since the consumer was gracefully terminated.

3.2.5 Consumer Maximum Capacity

The way the problem is formulated assumes the consumer is capable, when required, to achieve a maximum data consumption rate, still to be defined, analogous to the capacity of a bin in a BPP.

With the goal of attaining a value for this maximum bin capacity, the consumer was tested in 3 different scenarios, each requiring the consumer to be working at peak performance.

Peak performance is defined as the case when the sum of the bytes still to be consumed from all the partitions the consumer is assigned to, is bigger than `BATCH_BYTES`.

The 3 testing scenarios aim to test the consumer's throughput while varying the number of tables it has to insert data into, the average amount of bytes available in each of the assigned partitions, and the number of partitions assigned to it.

The consequence of having more tables where data has to be inserted, directly affects the third phase by increasing the amount of asynchronous request that have to be made.

As for reducing the average amount of bytes available in the partitions assigned, but still being able to make the consumer work at full capacity, aims to test the first phase of the algorithm, where the data has to be consumed from the partitions assigned from Kafka.

This is the case since this high-level consumer is based on the Kafka client provided by the confluent_kafka package. When the client begins, it starts a low level consumer implemented in C that runs in the background. As the low-level consumer runs, it buffers messages into a queue until the high level python consumer requests for what it has consumed with either `poll()` or the `consume()` methods. The difference between these two methods is that the first returns a single message whereas the second returns a batch of messages defined by `num_messages`, from the kafka client's buffer.

To improve throughput, when a consumer requests for data from the broker, the broker attempts to batch data together before sending it back to the consumer. `fetch.max.bytes` and `fetch.max.wait.ms` define how a single request is handled by the broker, wherein the first determines the maximum amount of bytes returned by the broker, and the second, the amount of time the broker can wait before returning the data if fetch.max.bytes is not satisfied.

Reducing the average amount of bytes in each of the partitions assigned, leads to the consumer having to perform more requests to fetch the same amount of data defined by `BATCH_BYTES`.

The number of partitions assigned to the consumer is expected to influence the time it takes for the consumer to gather `BATCH_BYTES`, by increasing the amount of requests required to fetch the data, as assigning partitions increases the probability of the consumer having to communicate with more brokers.

Another important definition is that the consumer is said to have reached it's steady state, when it is capable of fetching `BATCH_BYTES` prior to `WAIT_TIME_SECS` being triggered in it's first phase. This reflects that the low-level consumer is capable of gathering the necessary amount of bytes into it's buffer, while the high-level consumer is running all the remaining phases except the first of the insert cycle.

For the following test cases, `BATCH_BYTES = 5000000` and `WAIT_TIME_SECS = 1`.

These values inherently reflect the time it takes for a consumer to go through each of it's phases, and the rate at which it can consume data.

Table 3.1: Testing conditions to obtain consumer maximum throughput measure.

Test ID	Total Bytes	Average Bytes	Number of Partitions	Number of Tables
Test 1	648058050	20251814	32	1
Test 2	100140747	863282	116	5
Test 3	678388069	4711028	144	5

Test 1 has the consumer fetching records which are all directed to the same table, and every partition it visits has enough bytes to satisfy the max.fetch.bytes condition, optimizing the throughput between the low-level consumer and the brokers, as the data is batched together. Since there is only one table to send the data to, the third phase is also optimized as there will only be a single BatchList instance.

As for Test 2, the increased number of partitions and the reduced average amount of bytes in each partition has the consumer polling more brokers to gather `BATCH_BYTES`, increasing the time the consumer takes in the first phase. Although the time taken in this phase increased The average time it takes the consumer to fetch `BATCH_BYTES` from the kafka cluster, indicates that the consumer still manages to reach its steady state.

Lastly, Test 3 combines both of the previous scenarios, where there is more data per partition to be sent back to the consumer, although some iterations require the consumer to send data to 5 different tables since the data originates from partitions that belong to different topics.

The metrics that were deemed important for the analysis of the consumer's behaviour, were: Time to go through Phase 1 (Δt_{P1}); Time to go through Phase 2 (Δt_{P2}); Time to go through Phase 3 (Δt_{P3}); Total cycle time (Δt); Measured cycle throughput.

Each of these parameters is statistically summarized in the following table:

Table 3.2: Statistical summary of the metrics

Test ID	Summary Measure	Δt_{P1}	Δt_{P2}	Δt_{P3}	Δt	Throughput
Test 1	Average	0.394	0.000775	1.69	2.08	2417215.33
	Standard Deviation	0.0338	0.00620	0.0941	0.101	99234.23
Test 2	Average	0.462	0.00684	1.72	2.18	2307501.21
	Standard Deviation	0.125	0.001600	0.0946	0.169	171231.27
Test 3	Average	0.397	0.00896	1.68	2.08	4218220.20
	Standard Deviation	0.0466	0.00619	0.105	0.121	132017.37

The following graph demonstrates how the consumer is capable of maintaining a stable speed above a threshold of $2Mbytes/s$, when it finds itself in it's steady state (capable of consuming $5Mbytes$ before `WAIT_TIME_SECS` is triggered), sharing a mode between the three different testing scenarios around $2.3Mbytes/s$.

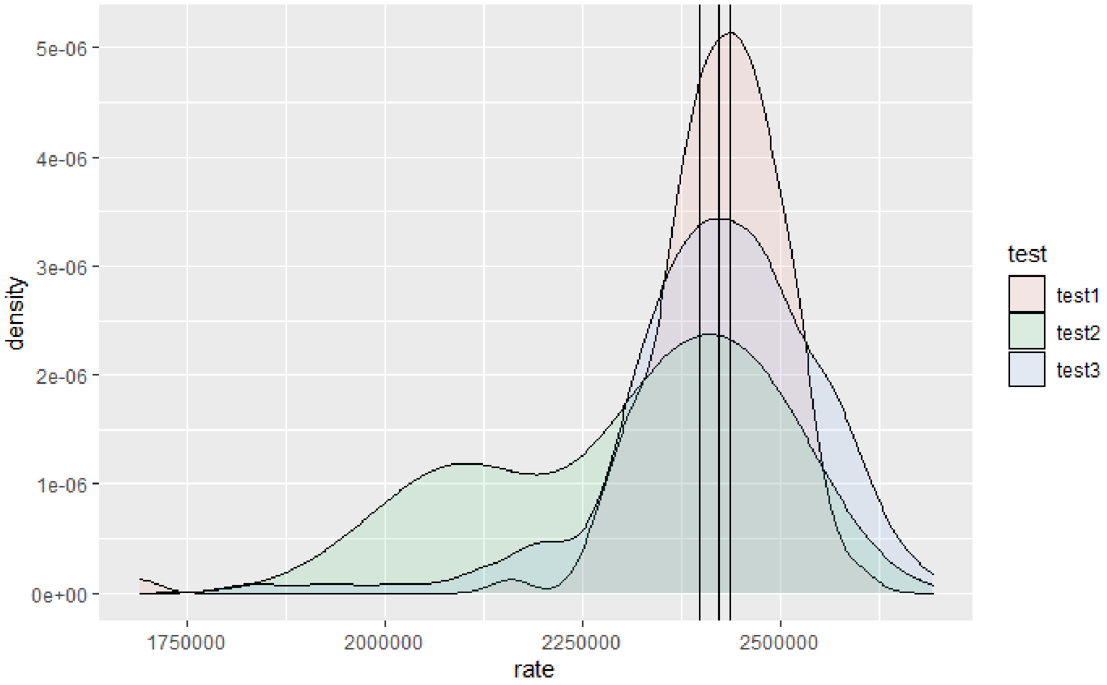


Figure 3.3: Density Plot for the consumer's measured throughput in the 3 testing conditions.

3.2.5.1 Bin Capacity

Two measures are defined which are to be used in the sections to come. The `ALGORITHM_CAPACITY`, is the constant bin size that will be used when running all the heuristic algorithms when solving the dynamic BPP, whereas the `CONSUMER_CAPACITY` is the consumer's maximum capacity, which if exceeded, must trigger a new configuration as for which partition should be assigned to which consumers.

The reason for having defined these two configuration parameters differently are due to the problem's dynamic nature. Since the measured write speed for each partition varies between each measurement, after executing the BPP algorithm for a single measurement, it is common to have bin's filled close to their capacity, which would lead to excessive rebalancing computations if the algorithm and consumer capacity were defined using the same value. Defining the Consumer's capacity to a value higher than the algorithm's perceived capacity, reduces the number of rebalances triggered, but also at the cost of the each configuration having more consumers than required.

From the graph 3.3, the Consumer's capacity is set to $2Mbytes$, whereas the Algorithm's capacity is defined as $1.5Mbytes$. These are the values used for the remaining part of the work.

It is important to note that these values are only valid when the consumers are deployed in the environment where they were tested. As long as the consumers share the same stable environment, the maximum capacity does not change with the parameters previously mentioned, and therefore the test can be run again to get these same baseline values.

3.3 Controller/Orchestrator

The controller is the component of the system which is responsible for orchestrating and managing the consumer group that is intended to consume the data from the partitions of interest.

The problem is modeled as a dynamic bin packing problem, where the size of each item is equivalent to the write speed of each partition, and the size of the container represents the maximum achievable speed of a consumer which has been provided by the data presented in 3.2.5.1.

For clarity, for the following sections, consumer, container and bin will be used interchangeably, representing a consumer instance described in 3.2.

This problem is different to the usual bin packing problem due to the changing write speed of each partition (the size of the items). This implies that a viable solution in one iteration, might not be viable in a future iteration when the sizes have changed.

At any given time, no consumer within the consumer group can have it's capacity exceeded by the cumulative write speed of the partitions assigned to it, and if it is, then a rebalance has to be triggered. The same applies in case a partition has not yet been assigned a consumer.

After the controller determines the state in which it wants its consumer group, it then creates the consumers that don't yet exist, communicates the change in assignment to each consumer in the group, followed by deleting the consumers that are not required in the new computed group's state.

While rebalancing any partition, during the time the controller is informing the current consumer to stop reading from it to inform the new consumer to start doing so, the messages belonging to this partition are not being read. This is the new cost that has to be taken into consideration when dealing with the rebalancing procedure.

3.3.1 System Design

As has been described, there are three different components in the system that work together to get the data from Kafka into bigquery. To do so, each component has to be able to communicate with one another to inform any change in state.

For this reason, the following diagram describes how the pieces fit together, and communicate with one another.

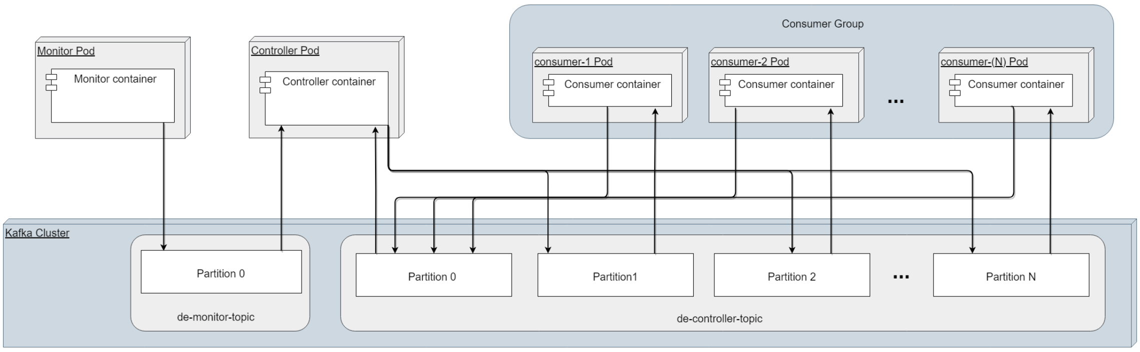


Figure 3.4: System architecture deployment diagram

As shown in the diagram, there are two main topics that will be responsible for providing an asynchronous communication between the 3 different entities of the system. The data-engineering-monitor topic, is where the monitor process sends the speed measurements for the controller to consult, whereas the data-engineering-controller topic is where the controller sends messages to each consumer to inform the change in their state. Partition 0 of this topic is reserved for the controller so the consumers always know how to acknowledge their change in state.

When using a Kafka topic to communicate between the consumers and the controller there are certain requirements that need to be complied with.

The first, is that when the controller communicates with a consumer from the consumer group, it has to be guaranteed that the message reaches the intended consumer.

Secondly, as will be common with this system, the controller will create and delete a single consumer multiple times in its lifecycle. The message offset a consumer starts on after restarting, has to be precisely the one where it left off before being shut down by the controller. This can be done leveraging kafka's `group-id` property, defined in each consumer client, and so the system just has to guarantee that the consumer gets the same id as before.

Thirdly, each consumer only reads messages which are intended to it. This would represent maximum efficiency in the information transmitted between both controller and consumer, since no process is reading messages that have to be ignored. To better understand this design requirement, the following scenario is described: It might happen that a certain number of consumers satisfies the controller's conditions, without any need to scale up or down the group. This would imply that the control messages sent by the controller are only read by the currently active consumers. When a new consumer is started by the controller, if the messages are shared, then there is a big queue of messages that have not yet been read by the new consumer since it was last up.

For the new consumer to be able to read new control messages sent by the controller, it would first have to read all messages that were not intended to it prior to it starting up.

Lastly, to guarantee temporal consistency in the control messages sent to a single consumer, it is clear that a control message can only be sent to a single partition, as kafka only guarantees message read order when the messages belong to the same partition. Kafka already does this by allowing a message to have a key attribute which is then hashed to determine the partition in which the message is to be inserted. The issue with this approach is if the number of partitions in the data-engineering-controller topic increases, the key might not send the messages to the same partition as before. As such, each consumer is given an incremental id $(1, 2, ...)$, which represents the partition where change in state information has to be sent for the consumer to read (both controller and consumer are aware of this id).

3.3.2 State Machine

The controller can be defined by a state machine, intended to continuously manage a group of consumers and their assignments. A group's assignment represents the same group's state.

The following diagram shows the high level architecture of the controller process:

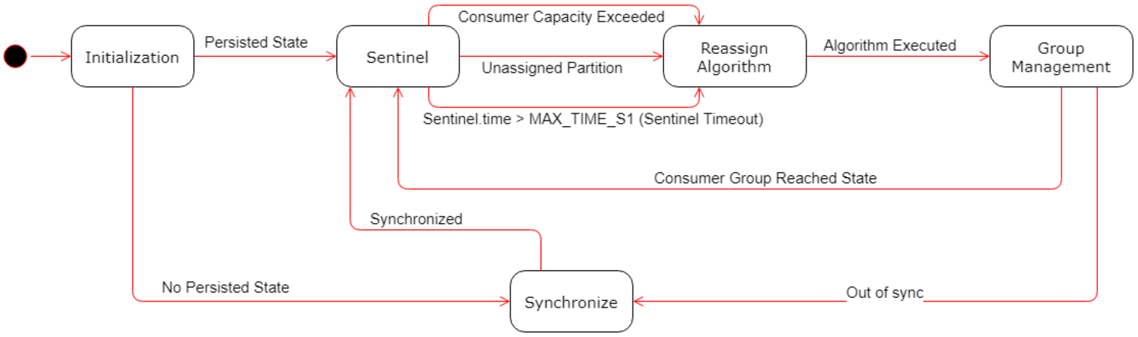


Figure 3.5: Controller State Machine.

3.3.3 State Sentinel

This state is where the controller uses the information provided by the monitor component (3.1), to determine whether it has to recompute the consumer group's assignment. The first step the controller goes through in this state is to read the last speed measurement the monitor component added to the data-engineering-monitor topic.

The controller then updates each of the partition's speeds, which in turn updates the cumulative speed of each consumer. As can be seen in figure 3.5, there are three transitions that can lead to the controller re-computing the current consumer group's assignment:

1. **Consumer Capacity Exceeded** - This transition is triggered if the cumulative speed of all the partitions assigned to any consumer, exceeds the maximum consumer capacity measured in 3.2.5.1.

2. **Unassigned Partition** - If there is any partition within the last speed measurement that is not currently assigned to a consumer, then this transition is triggered.

3. **Sentinel Timeout** - One of the controller's configurations, is a parameter `MAX_S1_TIME` which indicates the maximum amount of time the controller can spend in the sentinel state without triggering a rebalance. This trigger is to have a condition that runs the algorithm to verify if downscaling is viable, as the other triggers are directed to upscaling conditions.

3.3.4 State Reassign Algorithm

This state receives as input the current consumer group's assignment and the remaining unassigned partitions, and produces as output a new consumer group assignment, which is to be defined as the group's future state.

This state uses only approximation algorithms to determine the group's future assignment. The algorithms implemented are the already existing Next Fit, First Fit, Worst Fit, Best Fit, and each of the previous algorithms' decreasing versions, and modified versions of the Best and Worst Fit algorithms.

In the implementation of the existing approximation algorithms, another step was included during the creation of the bins which does not affect the outcome in terms of number of consumers. If the consumer that is currently assigned to the partition has not yet been created in the future assignment, this is the bin that is created, otherwise, the lowest index bin that does not yet exist is the one created.

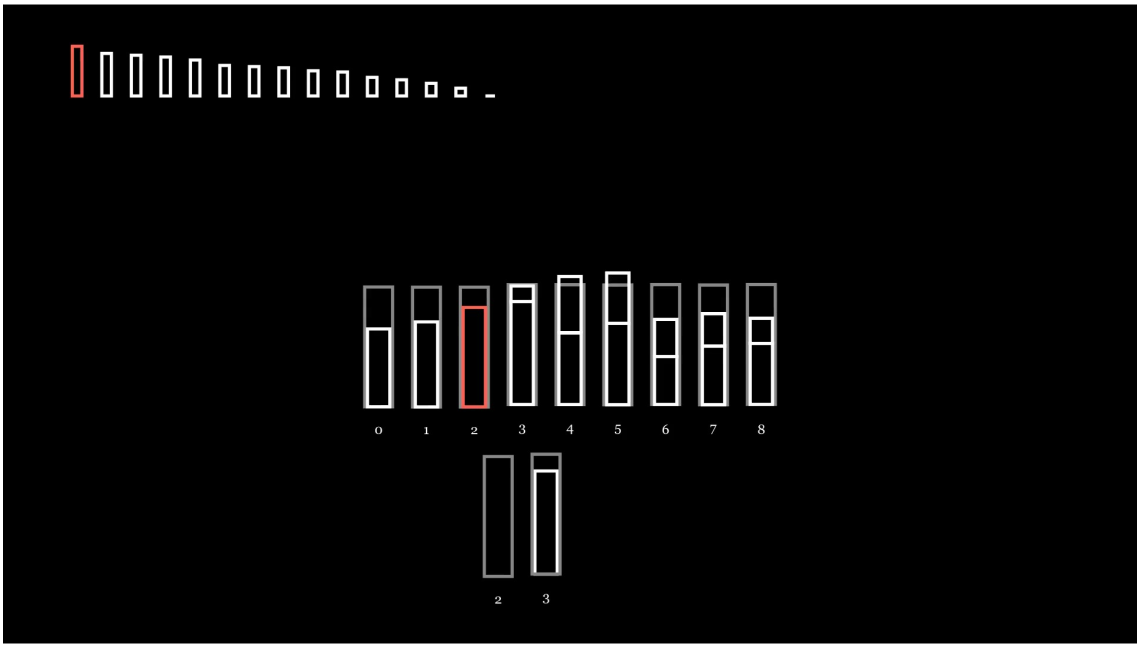


Figure 3.6: New bin creation procedure when performing an approximation algorithm.

Figure 3.6 illustrates this process, wherein the first row of consumer's (grey containers) represents the current consumer group's state, and the second the new consumer group assignment that is being computed by the controller. Partitions are represented in red and white, red being the partition currently being analyzed by the controller. In this example, the red partition currently finds itself in the consumer 2. When analysing the partition, the controller verifies if the partition fits in any of the existing partitions. Since it does not fit, and another consumer has to be created for this partition, the new consumer is the same as the one it is currently assigned, therefore occurring no rebalance for this partition.

This is not presented as a modification to these algorithms as their functionality does not change with this approach to for a bin's creation, as it simply adapts the algorithms to the rebalancing problem at hand, improving it as compared to always creating the lowest index bin, as that would imply rebalancing partitions more often.

3.3.4.1 Modified Any Fit Algorithms

The motivation to modifying the existing Any Fit algorithms, is that the item reassignment is not the problem they are trying to solve, they simply aim to reduce the amount of active bins for a certain list of items.

Within this problem's context, rebalancing inevitably implies having the consumer group to stop consuming from a partition, while the controller is reassigning the partition from one consumer to another (as there cannot be a concurrent read from the same partition).

A new measure is presented to compute the the total rebalance cost (Rscore), of a new group's configuration. This metric represents the rate at which information is accumulating for each second that passes, given in consumer iterations per second.

Table 3.3: Data to compute the Rscore for an iteration.

Symbol	**Description**
P_i	set of partitions that were rebalanced in iteration i
$s(p)$	outputs the write speed of partition p
R_i	Computed total Rscore for iteration i
C	Constant that represents the maximum consumer capacity from 3.2.5.1

The following equation then presents the cost of rebalancing for a single iteration i.

$$R_i = \frac{1}{C} \sum_{p \in P_i} s(p) \tag{3.1}$$

Since the controller has access to the speed measurements of all partitions and it does not require to read and assign each partition in any particular order, this bin packing problem can be categorized as offline.

Different from the previous decreasing versions of the Any Fit algorithms which have already been described in 2.5.2, this algorithm will not sort the partitions, but in turn the consumers (bins).

There will be 2 main approaches to sorting the consumers:

- Sorting each consumer based on their current cumulative speed (cumulative sort);
- Sorting each consumer based on the partition assigned to it that has the biggest measured write speed (max partition sort).

After sorting the current consumer group using one of the above strategies, the procedure is as follows:

Algorithm 4: Modified Any Fit Pseudo Code

```
input  : current consumer group C and set of unassigned partitions U
output: consumer group N which is the next state for the consumer group
N ← new ConsumerList(assign_strategy=("BF" | "WF"));
sorted_group ← sort(C, sort_strategy=("cumulative" | "max partition"));
for consumer ∈ sorted_group do
    Set<Partition> partitions ← consumer.assignedPartitions();
    List<Partition> sorted_partitions ← sort(partitions, reverse=True);
    for i ← length(sorted_partitions)-1 to 0 do
        partition ← sorted_partitions[i];
        result ← assignExisting(N, partition);
        if result == False then
            break;
        end
        remove(sorted_partitions, partition);
    end
    if sorted_partitions.length() == 0 then
        continue;
    end
    createConsumer(N, consumer);
    for partition ∈ sorted_partitions do
        result ← assignCurrent(N, partition, consumer);
        if result == False then
            break;
        end
        remove(sorted_partitions, partition);
    end
    extend(U, sorted_partitions);
end
sorted_unassigned ← sort(U, reverse=True);
for partition ∈ sorted_unassigned do
    assign(N, partition);
end
return N
```

For each consumer in the sorted consumer group, the partitions assigned to the consumer are sorted based on their write speed. From smallest to biggest, each partition is inserted into one of the bins that have already been created in the future assignment, based on one of the any fit strategies, which in the tested implementation, can either be with a Best or Worst Fit strategy. If the insert is successful, then the partition is removed from the sorted list of partitions, otherwise, if there is no existing bin that can hold the partition, then the current consumer assigned to it is created.

The remaining partitions in the sorted list are now inserted into the newly created bin, from biggest to smallest, and removed from the sorted list if successful. If a partition does not fit into the newly created consumer, then the remaining partitions in the list of sorted partitions are added to the set of unassigned partitions.

After performing the same procedure over all consumers, there is now a set of partitions which have not been assigned to any of the consumers in the future assignment. These partitions are first sorted in decreasing order (based on their measured write speed), and each partition is assigned using their respective any fit strategy.

Table 3.4: Modified implementations of the any fit algorithms.

Algorithm	**Assign Strategy**	**Consumer Sorting Strategy**
Modified Worst Fit	Worst Fit	cumulative write speed
Modified Best Fit	Best Fit	cumulative write speed
Modified Worst Fit Partition	Worst Fit	biggest partition write speed
Modified Best Fit Partition	Best Fit	biggest partition write speed

3.3.4.2 Testing

To evaluate the performance of each of these algorithms, two metrics were used: The relative difference in the number of consumers of each algorithm to the approximation algorithm that presented the least amount of consumers; The Rscore of the algorithm's new configuration.

Table 3.5: Data to create the testing environment for the approximation algorithms.

Symbol	**Description**
P	Set of partitions assigned to the consumer group.
s_p^i	Provides the speed for partition p in iteration i
$\phi(\delta)$	Uniform random function that selects a value between $[-\delta, \delta]$

Given the data provided by 3.5, the testing sequences for the algorithms was obtained as follows:

1. 200 different partitions are created and given an initial speed. Four different initial conditions for all partition were tested: Starting at 0% of the `ALGORITHM_CAPACITY`; Starting at 50% of the `ALGORITHM_CAPACITY`; Starting at 100% of the `ALGORITHM_CAPACITY`; Assigning each partition a uniform random value between $[0, 100]\%$ of the `ALGORITHM_CAPACITY`.

2. Given the initial conditions, a new measurement is created, where the speed in the next measurement is calculated using:

$$s_p^{i+1} = s_p^i + \phi(\delta) \times algorithm\_capacity, \quad \forall p \in P, \tag{3.2}$$

having $\phi(\delta)$ be evaluated every new computation.

3. The previous step is repeated for the next measurements until there are 500 measurements in a single sequence.

Using this procedure, six different sequences of measurements were created, each differing in the δ used to vary the speed. As such, for each initial condition, there is a sequence where δ is set to $0, 5, 10, 15, 20, 25$. In other words, there are $6(deltas) \times 4(starting\ conditions)$ different sequences, each representing a measurement instance provided by the monitor process. After reading a single measurement from a sequence, the controller runs the implemented algorithms to find the group's configuration in all scenarios.

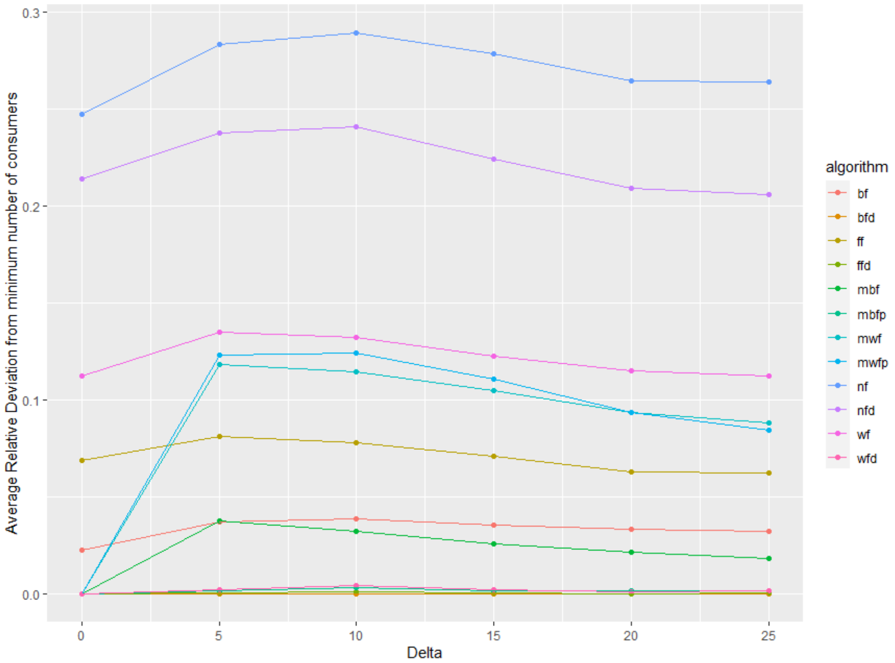


Figure 3.7: Average Relative Deviation from the configuration that provides the minimum number of consumers, with a random initial speed for each partition.

To better illustrate the performance of the algorithms for each measurement sequence, the metrics are statistically summarized using the average. The x axes refers to the 6 different measurement sequences for a given starting condition, and the value is the δ defined for that measurement. As for the y axes in 3.7, it indicates the algorithm's average relative deviation from the configuration that provides the minimum number of consumers.

In 3.7, each partition starts with a random speed between $[0, 100]\%$ of `ALGORITHM_CAPACITY`. The worst performing algorithm is the next fit followed by it's decreasing version, since every time a partition is to be assigned, the algorithm only verifies if it fits in the last created bin, as opposed to considering all the already existing bins. The remaining any fit decreasing algorithms, are the ones that perform the best, with the best fit decreasing consistently presenting the best results.

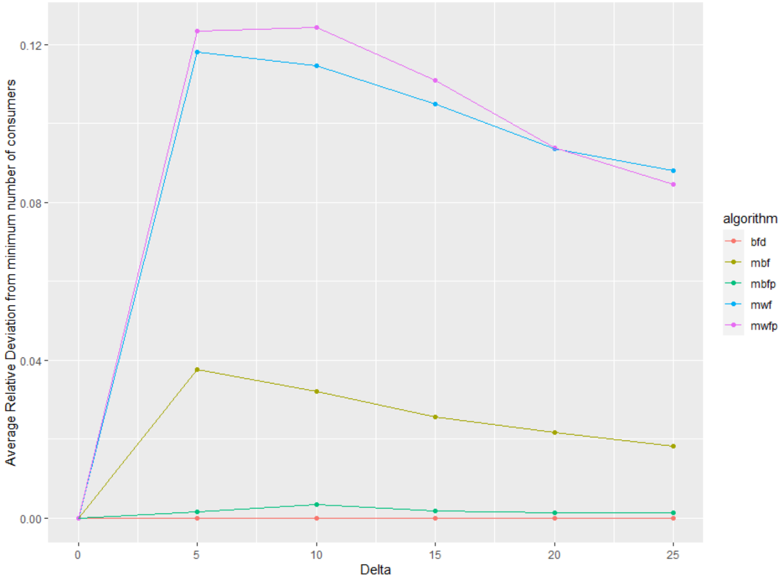


Figure 3.8: Average Relative Deviation from the configuration that provides the minimum number of consumers, with a random initial speed for each partition. (Filtered to present the modified algorithms and BFD)

As for the modified versions of these algorithms, due to its sorting strategy, MBFP shows the best results. It is also worth noting that for smaller variabilities, the modified algorithms behave similarly to the online versions of their any fit strategy, but, the higher the delta, the bigger the variability, which also leads to more rebalancing, having the modified algorithms behave more like the decreasing versions of their fit strategy.

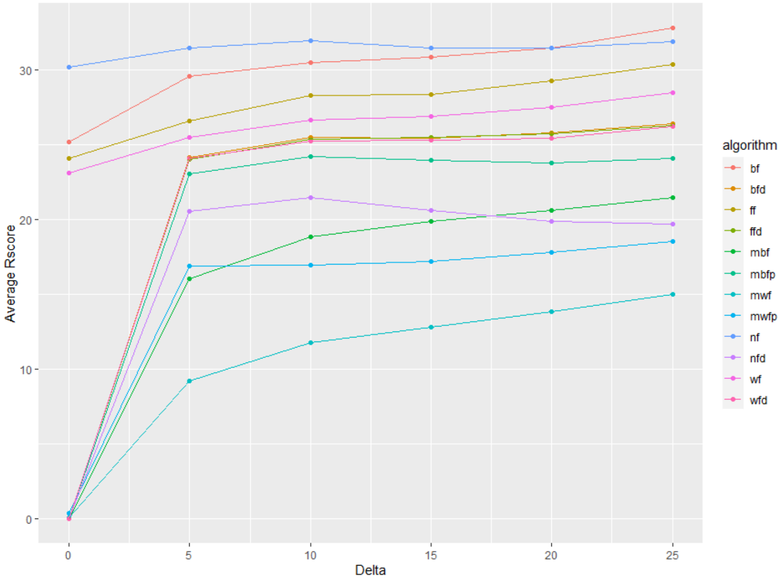


Figure 3.9: Impact on Rscore for different Deltas (random initial partition speed).

For the same starting conditions, 3.9 presents the average Rscore of each algorithm for all six measurement sequences.

As can be seen in 3.9, the modified algorithms present the best Rscore for each measurement along with next fit decreasing. The reason why the NFD presents such a result, is due to the increased amount of consumers it creates to assign new partitions, which at the moment of creation, will always be the same consumer as the one assigned to the partition being analyzed, as long as it has not yet been created (3.6).

For a similar reason, the modified algorithms that perform the best with regards to the Rscore, are also the ones that perform worst (compared to the remaining modified algorithms) when evaluating the relative number of consumers.

The remaining tests performed are presented in the appendix, and it can be concluded, that the initial conditions do not significantly affect the relative performance of the algorithms (**??**).

To select the algorithm to use within the controller, there is a trade-off between the aforementioned testing metrics. On account of the added re-balance concern within the modified algorithms, these present an improvement with regards to the Rscore compared to the approximation algorithms presented in the literature.

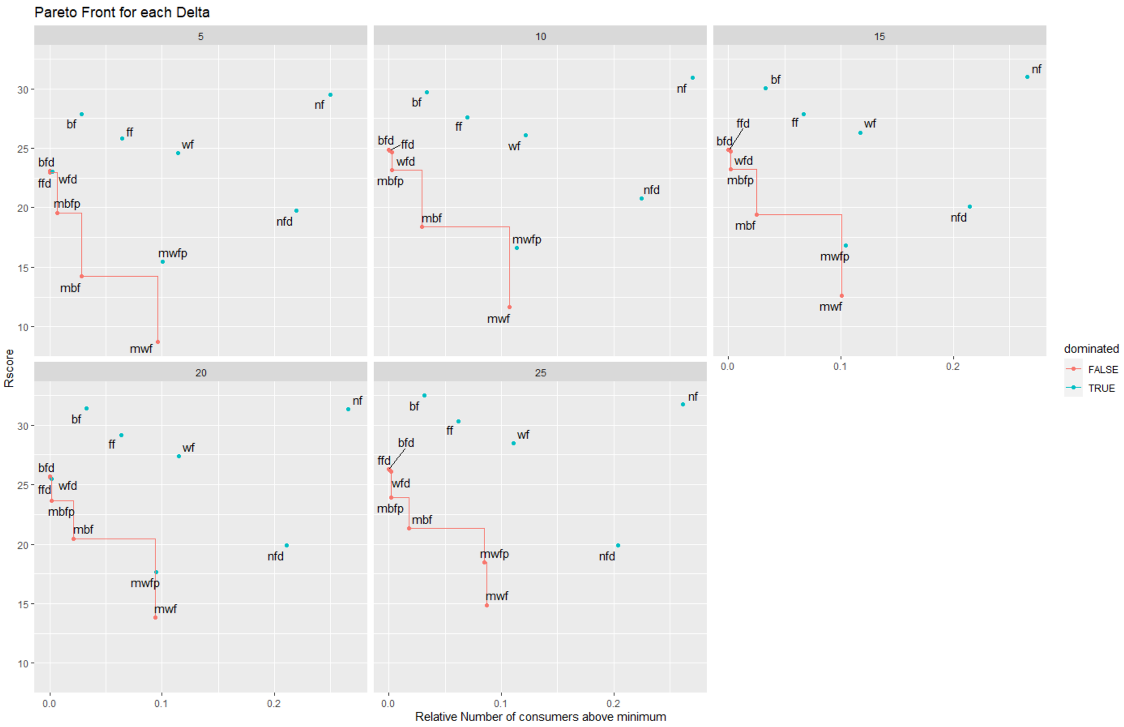


Figure 3.10: Pareto Front for each delta value used in the measurement sequences.

The pareto front is a way of evaluating the set of solutions that are most efficient, provided there are trade-offs within a multi-optimization problem.

Excluding MWFP, the modified algorithms are consistently a part of the pareto front 3.10, which implies these are a competitive option as to which algorithm to pick for the reassign algorithm to be executed in the controller. The algorithm that shows the best Rscore for the different variabilities is the MWF, whereas the modified algorithm that performs the best relative to the number of consumers used in each configuration is the MBFP.

3.3.5 State Group Management

This state is where the controller informs each consumer of their change in state. Since there cannot be any concurrent read of a partition by two consumers of the same consumer group, when rebalancing a partition, the controller first has to inform the consumer currently assigned to the partition to stop consuming from it, and only after the consumer acknowledges having acted to the request, can it inform the new consumer of it's new assignment.

This state not only handles this message exchange but it also creates and deletes consumer resources from the Kubernetes cluster.

3.3.5.1 Difference between current and future state

The current consumer group's state is represented by a list of consumers, each having their own assignment. The new computed group's state is defined as the next state, and is also a list of consumer's each having an assignment, but representing the desired state the controller will communicate to each consumer.

Similar to computing the difference between two sets, when computing the difference between 2 consumer lists the procedure involves iterating over each consumer (in both states), and calculate the difference between the two consumer assignments (set of assigned partitions). There are three scenarios that result in different actions the controller has to perform, for a given position *i* of each consumer list:

- next state has a consumer at *i* but the current state does not - This means that the consumer has to be created by the controller, and the partitions assigned to the consumer in position *i* of the next state, have to be associated with a StartConsumingCommand for this same consumer.

- next state does not have a consumer at *i* but the current assignment does - The partitions associated with the consumer at position *i* of the current state have to be associated with a StopConsumingCommand directed to this consumer, and the consumer has to be added to the list of consumers to remove.

- both next state and current state have a consumer at position *i* - This means that no creation or deletion operation has to be performed for this bin, and the operations to perform in this case are only communicating to the already existing consumer the difference in it's assignment.

  The same consumer has two different sets of partitions assigned to it represented in the two different group states. `next_assignment` will denote the set of partitions attributed to the consumer's state in the next group's context, and `current_assignment` is defined as the consumer's set of partitions in it's current state.

  ```
          partitions_stop = current_assignment - next_assignment
          partitions_start = next_assignment - current_assignment

  ```

  the resulting set of partitions within `partitions_stop` have to included in a StopConsumingCommand, whereas the partitions in `partitions_start` in a StartConsumingCommand, both message types directed to consumer *i*.

3.3.5.2 Managing the Consumer Group in the Kubernetes Cluster

Each active consumer in the current group's state, represents a deployment with a single replica (pod). The reason why this is the case is that each consumer requires a volume to persist its data, which implies having a different volume for each pod. This is also possible using stateful sets, but the constraint of a stateful set is that when removing a replica of the set, it only allows to remove the last added replica, and in this context, a more granular approach is required when removing an active consumer, as it does not necessarily have to be the highest indexed consumer.

Each consumer is given an individual id through its deployment's `metadata.name` property, a value that can be obtained within the pod using the downwardAPI, that provides a pod with its context. This individual id assigned to the deployment's name, is the one used to inform the pod of its metadata partition to consume from.

To allow the controller to create, list and delete the resources within the cluster, a Kubernetes service account is used to authenticate the controller, which is then given permissions for the aforementioned operations through a Kubernetes Role. In this scenario, the controller's service account is linked with a Role object that has permissions to create, list and delete, deployment and persistent volume claim resources.

Since all consumers have to be able to persist data, each consumer has to be mapped to a persistent volume using a persistent volume claim. If it is the first time a consumer with a given id is being spawned, the controller has to dynamically create and map a persistent volume claim to the consumer's deployment.

To simplify the process, two template yaml files (A.2, A.3) are used, one for creating persistent volume claims (PVC), and another used to create deployments. The controller is only responsible for changing the template PVC id when creating it. When the controller has to create the deployment, it has to reference the created PVC in the template deployment, and change the deployment's id to the incremental id attributed by the controller.

As an example, if the controller has to create a consumer whose id is 5, it would go through the following steps:

1. If the PVC with name `de-consumer-5-volume` does not yet exist, change the PVC metadata.name parameter to `de-consumer-5-volume`, and send the create request with the body containing the modified yaml file;
2. Change the template deployment's metadata.name parameter to `de-consumer-5`, and add a reference to the PVC created in the previous step.

3.3.5.3 Communication between controller and Consumer Group

Using the computed difference between the next and current group's state, each consumer has a set of start and stop messages that have to be sent by the controller, for the group to reach the intended state.

Each partition can have associated to it, at most two actions, which correspond to a start and/or a stop command.

Firstly, the controller prepares a batch of StartConsumingCommand messages, each directed to a single consumer. For each partition in the set of unassigned partitions, the partition id is added to the StartConsumingCommand message that is intended to the consumer that was assigned the partition.

Another batch of StopConsumingCommand messages is prepared, and for each partition that has to be either rebalanced or removed, that partition's id is added to the message which is intended to the consumer that is currently assigned to that partition.

Each message the controller sends out to a consumer, has to be acknowledged by the consumer with a corresponding event which can be one of two types, StartConsumingEvent, StopConsumingEvent. Each of these messages contain a set of partitions from which a consumer acted upon. For each partition contained within one of the events, the controller removes its corresponding actions from the set of actions that have to be deployed by the controller.

If the received event is of type StopConsumingEvent, a new batch of StartConsumingCommand records are prepared for the consumers that have to start consuming from the partitions that are being rebalanced. This means that for each partition referred in the StopConsumingEvent, if it has a corresponding start action, the partition is added to the StartConsumingCommand directed to the new consumer.

When the controller sends a message to a consumer, it sends it to the same partition as the consumer's id in the data-engineering-controller topic. As for the consumer, to communicate the event of having reacted to one of the controller's commands, the record has to be sent to the partition with id 0 of the same topic, as depicted in the diagram (3.4).

This process terminates when there are no more actions to perform over any partition, since the controller removes a start or stop action from a partition as soon as it receives the acknowledgement by the consumer that it has performed the corresponding command.

3.3.6 Synchronize

The controller enters this state if the state it holds of the consumer group, which is the state it has for each active consumer, is not synchronized with the state in each consumer.

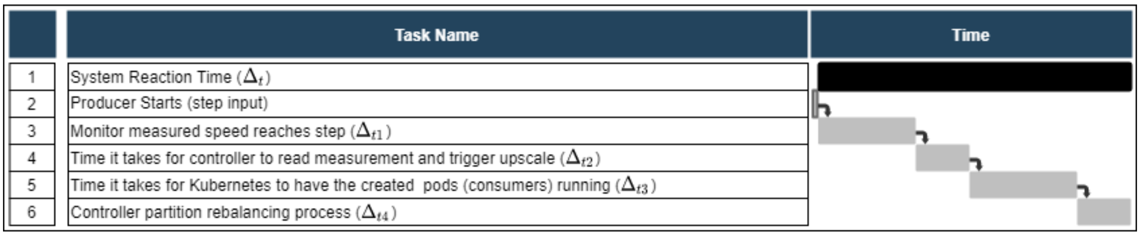
This state was then created to mitigate this problem, following a procedure which has the controller querying the kubernetes cluster to verify which are its active consumers. Given this, the controller then queries each consumer through their respective partitions for their current state, and only after all active consumers have responded does the controller proceed to its normal behaviour triggering the transition to the Sentinel state.

While testing the controller, this state would most commonly be triggered after the controller would unexpectedly fail and then restart.

Chapter 4

Integration Tests

The following chapter will focus on the main sequence of events that the autoscaler goes through to scale its consumer group.



	Task Name	Time
1	System Reaction Time (Δ_t)	
2	Producer Starts (step input)	
3	Monitor measured speed reaches step (Δ_{t1})	
4	Time it takes for controller to read measurement and trigger upscale (Δ_{t2})	
5	Time it takes for Kubernetes to have the created pods (consumers) running (Δ_{t3})	
6	Controller partition rebalancing process (Δ_{t4})	

Figure 4.1: System sequence of events.

The time it takes the controller to react to a certain measurement depends on all the events presented in 4.1. The following sections will delve into each of these events.

The data points presented for each Δ_{tn} $n \in \{1,2,3,4\}$, were obtained by testing the system with various simulations of step inputs, and randomly generated sequences of measurements similar to the sequences used in 3.3.4.2.

4.1 Monitor Measurement Convergence Time

To evaluate the monitor's response time, the data points were obtained by feeding the system a step input, which is obtained by starting a producer that sends data to one of the partitions the consumer group is consuming data from. The production rate is slightly higher than the maximum consumer's capacity obtained in 3.2.5.

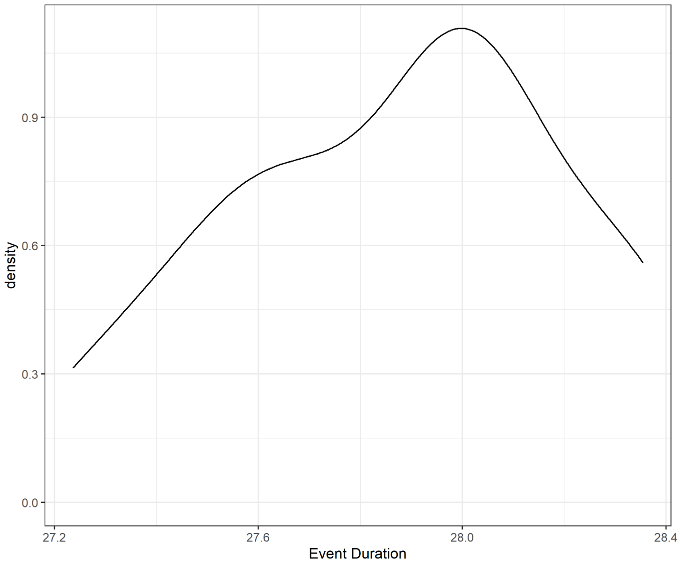


Figure 4.2: Distribution of the time it takes the monitor process to reach the step value for 33 different step inputs.

As such, this measure is the time it takes the monitor process to converge to the real production rate, which, as can be verified in 4.2, takes no more than 30*s*, similar to what had been obtained in 3.1.

4.2 Time to Trigger Scale-up

Measured as the time it takes the controller to compute the consumer group's new assignment, and to send an asynchronous request to the GKE cluster for every new consumer instance to be created. To obtain the distribution for this metric, the system was tested with the aforementioned step inputs and the randomly generated measurement sequences.

The time the controller takes with this procedure depends on the number of partitions to distribute between the consumer group, the algorithm the controller is executing to figure a new consumer group assignment, and the number of new consumer instances it has to create in the GKE cluster.

For the tested input data, where there were at most 32 partitions to rebalance and no more than 20 consumers to be created in a single instance. In these conditions, the event consistently takes less than 1 second to be executed (4.3).

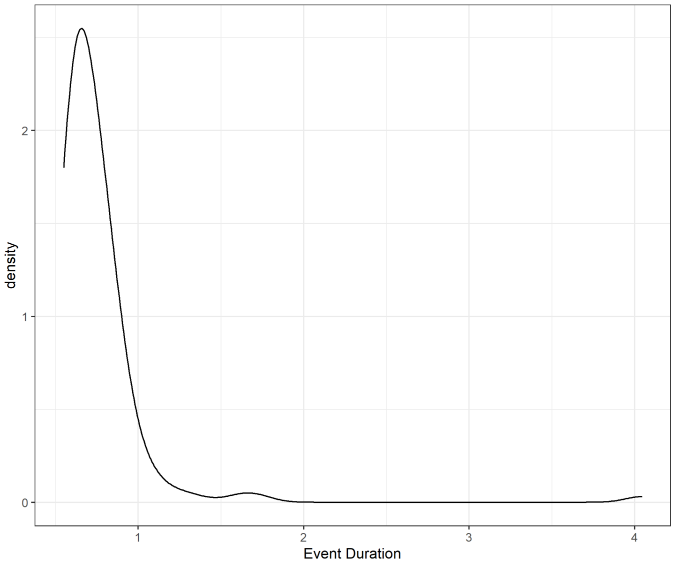


Figure 4.3: Distribution of the time it takes the controller to compute a new consumer group state and send the asynchronous request to create the new consumers to the kubernetes cluster.

4.3 Newly Created Deployments Ready

After making the asynchronous request to the GKE cluster, the control plane schedules the pods to a node, and within the node starts the containers that are to be executed.

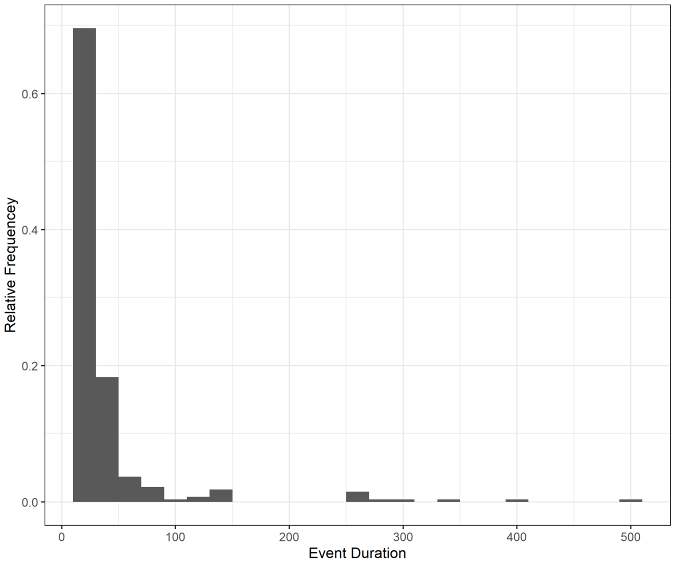


Figure 4.4: Relative frequency histogram of the time it takes the kubernetes cluster to have the newly created consumers "ready".

After discarding the data points that didn't have the controller creating any new consumer, there remain 273 data points, which show two clusters of data points that are justified by two different scenarios when creating a new pod.

The first, and most frequent (88%), has the GKE cluster taking between $[10,50]s$ to have a new consumer instance ready. This is usually the case when the controller requests for the creation of new consumer instances and the Kubernetes cluster has resources available to schedule the new consumer instance. As such, the events duration is related to the time it takes the scheduled node(s) to download the image from the container registry, and to start the containers.

The second group of data points, any time span greater than $50s$ (represents 12% of the data points), occurs when the Kubernetes cluster does not have any available resources. Here the actions the cluster undergoes are adding a new node, and only then scheduling the consumer instance into the new available node. Due to the autoscaling feature of the GKE cluster, this is done automatically but it is also more inconsistent, having data points taking up to $500s$, although very sporadically.

Although there isn't much control over the time it takes the Kubernetes cluster to schedule and start the pods, one variable that can be controlled is the size of the image which has to be downloaded by the nodes that were assigned the newly created consumer instances.

4.4 Communicating Change in State

Each partition that the controller is monitoring can either be associated with a start, stop, rebalance or a do nothing action.

A start action refers to the case where there is no consumer currently consuming data from the partition, and the controller has assigned that partition to a consumer in its newly computed consumer group state.

The stop action occurs when the controller no longer wants a consumer assigned to this partition therefore having no consumer fetching data from this partition in the group's future state.

Rebalancing happens when the controller is changing the partition from being assigned from one consumer to another.

Lastly, as the name suggest, do nothing is when there are no actions to be communicated for a partition by the controller to a consumer of the consumer group.

Since there is no concurrent read from the partition when the controller is analyzing a start or a stop action, the controller can send this message as soon as it enters this event, having to wait at most for 1 consumer cycle to receive the acknowledgement by the consumer.

When rebalancing, the controller has to first send out the stop command, wait for a response, send out the start command, and wait for the response. Without taking into account any processing and network delays, at worst, the controller might have to wait for 2 consumer cycles to be able to rebalance a partition.

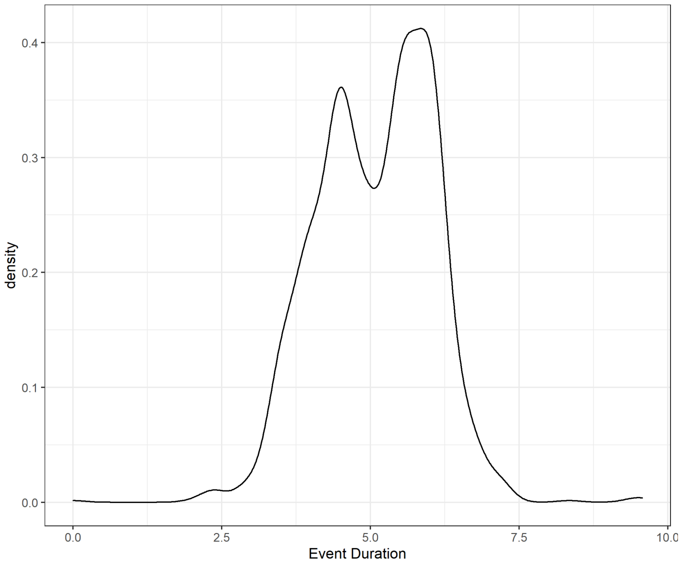


Figure 4.5: Distribution of the time it takes the controller to communicate the change in state to its consumer group.

A consumer's insert cycle goes through the different phases mentioned in 3.2, and only after performing it's tasks does it verify the metadata queue to check if it has received any change in state command. For this reason, between two metadata reads from the data-engineering-controller topic, it takes the consumer 1 whole insert cycle. Having defined in 3.2.5 that the consumer gathers at most $5Mbytes$ per cycle, and provided that the results from 3.3, which indicate the consumer has a data rate of approximately $2Mbytes/s$, each consumer cycle can take approximately $2.5s$. Since the controller has to wait for 2 consumer cycles, this would imply that change the group's state could take at least 5 seconds, as can be seen in 4.5

It is also worth noting that the more consumers there are in the group, the higher the probability that the consumer has to wait 1 whole cycle after sending out the stop command, and another whole cycle after a start command, as the communication is performed with more consumers.

Although this analysis was performed for a single partition, without any loss of generality and due to the fact that the controller sends out the change state commands in batch, the time it takes the controller to change a consumer group's state remains at 2 consumer cycles.

To vary the duration of this event, the consumer's cycle has to be altered using the `BATCH_BYTES` and `WAIT_TIME_SECS` parameters, which in turn have an effect on the time the consumer spends in a whole insert cycle.

Chapter 5

Conclusion and Future Work

5.1 Summary and Discussion of Thesis Results

The work developed in this thesis challenges the existing methods used to scale a Kafka consumer group and how the load should be distributed between the various consumers.

The goal was to deterministically solve the autoscaling problem related to a group of consumers, which in turn was modeled as a Bin Packing Problem where the items that have to be assigned a bin can change in size over time. Due to this particular feature, a new metric was introduced to determine a rebalance score (Rscore) for a given iteration.

The Rscore introduced a new concern when executing an algorithm to solve the bin packing problem, and for this reason 4 new algorithms were presented to heuristically solve the BPP, while at the same time taking into consideration how the items (partitions) were rebalanced to different bins (consumers). Three of the presented algorithms (MWF, MBFP and MBF), proved themselves to be a competitive alternative, consistently belonging to the pareto front 3.10 when comparing the multi-variate optimization with regards to the number of consumers and the Rscore.

To further consolidate the BPP Model, the consumer was tested in diverse conditions to determine whether a constant bin size model could be used for the scenario at hand. Provided the results in 3.2.5.1, the BPP was then modeled as its single bin size variant.

The different parts in the system interacted with one another as expected to deliver an autoscaling consumer group capable of adapting depending on the current load of the system as shown in 4.

5.2 Future Work

Given the multiple parts developed within this system, there is a lot of room for improvement. Most notably, due to the close relationship between the system and the Kubernetes cluster, the autoscaler could be wrapped as a Kubernetes Operator [**KuberenetesOperator**] as is common with systems of this kind ([**Kubegres**, **PulumiOperator**, **KEDA**]).

5.2.1 Monitor

To the best of my knowledge, for lack of a better metric being provided by the Kafka cluster, the value that was used to track the write speed of each partition was the number of bytes in each partition. For this reason, a process 3.1 was used to evaluate the speed of each partition by historically storing the amount of bytes in a partition at a certain timestamp.

A clear disadvantage of using this component is the fact that if `retention.ms != -1`, this means a record is deleted from its partition after `retention.ms` which leads to a reduction in the amount of bytes in the partition. The monitor process (3.1) then erroneously evaluates a negative partition write speed. To circumvent this behaviour, either one of the following metrics provided by Kafka would suffice: The current write speed (bytes/s) per partition; A historic amount of bytes that have been written to a partition.

Since the monitor process presents the write speed as the average data rate over the last 30 seconds, using a different strategy to evaluate the speed could lead to more robust measures.

5.2.2 Consumer

If the network is experiencing increased latency, the current implementation of the autoscaler does not account for changes in the size of the consumer capacity, therefore the controller assumes a maximum capacity of a consumer which is not accurate within this scenario. Metrics related to a consumer's performance provided by Kafka JMX metrics could be leveraged to obtain a more accurate dynamic representation of the consumer's current capacity. This could also lead to a variable bin size BPP.

5.2.3 Controller

Throughout this thesis, the controller process was presented with strict execution rules and actions to manage the Kafka consumer group. Evolving this component into a more abstract concept, could not only maintain its current functionalities, but it could also be used to more accurately solve load balancing within a consumer group without necessarily modeling the problem as a BPP.

In fact, the heuristic algorithms that make use of the worst fit rule when assigning partitions to consumers, are at the same time applying a load balancer rule between the available consumers.

5.3 Final Remarks

In spite of the fact that, as a whole, the thesis presents quite a specific use case, individually several of Kafka's functionalities are challenged and tested alternatives are also provided which can be used as a foundation so as to improve the way in which a consumer's load is modeled, and the manner in which the load is distributed between the elements of a consumer group.

Appendix A

Appendix

```
{
    "name": "DEControllerSchema",
    "type": "array",
    "items": {
        "name": "TopicPartition",
        "type": "record",
        "fields": [
            {"name": "topic_name", "type": "string"},
            {"name": "bq_table", "type": "string"},
            {
                "name": "topic_class",
                "type": {
                    "name": "topic_class",
                    "type": "record",
                    "fields": [
                        {"name": "module_path", "type": "string"},
                        {"name": "class_name", "type": "string"},
                    ]
                }
            },
            {
                "name": "partitions",
                "type": {
                    "type": "array",
                    "items": {
                        "name": "partition",
                        "type": "int"
                    }
                }
            },
            {
                "name": "ignore_events",
                "type": {
                    "type": "array",
                    "items": {
                        "name": "event_type",
                        "type": "string"
                    }
                }
            },
        ]
    }
}
```

Listing A.1: Avro Schema for records sent to the data-engineering-controller topic

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: de-consumer-1-volume
  namespace: data-engineering-dev
  labels:
    consumerGroup: de-consumer-group
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: de-consumer-volume
  resources:
    requests:
      storage: 500Mi
```

Listing A.2: Template Persistent Volume claim

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: de-consumer-1
  namespace: data-engineering-dev
  labels:
    consumerType: consumer-autoscaler
    consumerGroup: de-consumer-group
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels:
      app: de-consumer
  template:
    metadata:
      labels:
        app: de-consumer
    spec:
      containers:
        - name: consumer-capacity
```

```
          image: ...
          imagePullPolicy: Always
          resources:
            requests:
              memory: "0.4Gi"
              cpu: "500m"
            limits:
              memory: "0.6Gi"
              cpu: "500m"
          env:
            - name: CONSUME_ENV
              value: "uat"
            - name: WRITE_ENV
              value: "uat"
            - name: GROUP_ID
              value: "data-engineering-autoscaler"
            - name: BATCH_BYTES
              value: "5000000"
            - name: WAIT_TIME_SECS
              value: "1"
          volumeMounts:
            - name: podinfo
              mountPath: /etc/podinfo
            - name: podpvc
              mountPath: /usr/src/data
      volumes:
        - name: podinfo
          downwardAPI:
            items:
              - path: "pod_name"
                fieldRef:
                  fieldPath: metadata.name
        - name: podpvc
          persistentVolumeClaim:
            claimName: de-consumer-1-volume
```

Listing A.3: Template consumer deployment