DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Self-Adaptive Data Processing for the IoT Platform

Peeranut Chindanonda

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Self-Adaptive Data Processing for the IoT Platform

Selbstadaptive Datenverarbeitung für die IoT-Plattform

Author:	Peeranut Chindanonda
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Vladimir Podolskiy
Submission Date:	12.11.2019

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 12.11.2019 Peeranut Chindanonda

Acknowledgments

I would like to profoundly express my gratitude to my advisor Vladimir Podolskiy, and my supervisor Prof. Dr. Michael Gerndt for their guidance and assistance. Their advice and comments mean a lot to me and help me to follow the right path until the end of this research.

I also thank my parents and my friend for providing me with endless support and continuous encouragement throughout my years of study. This success would not have been possible without them. Thank you.

Abstract

Internet of Things (IoT) has become an integral part of our lives as we use it for connecting physical things together. With the increasing number of IoT devices, it is thus necessary to have a dynamically scalable system that adjusts computing resources according to different needs. A good scaling mechanism would have to maintain the *service-level objective (SLO)*, and while minimizing computing resource usage would be desirable.

Within the scope of this thesis, this study developed and evaluated various ways to calculate the desired number of *container* instances. The calculation of the desired number of instances is separated into scale-up calculation and scale-down calculation. If scale-up calculation is greater than or equal to the current number of instances, the scale-up calculation will then be used as the desired number of instances. Otherwise, the maximum between the scale-up calculation and scale-down calculation will then be used as the desired number of instances, but not greater than the current number of instances. Many calculations were evaluated to be used as scale-up calculation and scale-down calculation. This includes calculations based on the incoming message rate (*producing rate*), number of pending messages (*message lag*), and their forecast value using double exponential smoothing. In addition, the study evaluated different intervals to recompute the desired number of instances.

The study found that Kubernetes's built-in auto scaling solution has too many limitations that are hard to circumvent without rewriting the source code. Therefore, a self-adaptive agent for auto scaling was built from the ground up to run the defined scaling mechanisms. The developed self-adaptive agent can query and analyze complex structured data from the system by itself; it can also limit scale-down actions in a period of time.

Overall, the study found that calculations based on forecast *message lag* could offer more computing resource savings while most messages are within the *SLO* than other methods tested. This study found that using the forecast-*message-lag*-based calculation for scale-up and scale-down calculation could save about 10% of the computing resource usage from the *producing-rate*-based calculation, while about 96% of the messages are within SLO in the test with the generated seasonal load. In addition to this, separating the calculation of the desired number of instances into scale-up calculation and scale-down calculation also helps to control scaling behavior more precisely. Applying the self-adaptive agent for auto scaling and the scaling mechanisms with queue-based IoT solutions could help in terms of scalability and efficiency of the system.

Contents

1 Introduction

Internet of Things (IoT) is increasingly used for connecting physical things together, and it has become an integral part of human society. IoT allows devices to communicate, process, and interchange data with one another over a network infrastructure to make things smart. Moreover, IoT is widely used and well known in several areas, such as smart devices, smart homes, and smart buildings. The number of IoT devices grows at an increasing pace; according to [1], there were 500 million connected devices in 2003. However, while it is still less than the human population, the number of connected devices will reach 50 billion in the future by 2020 [1]. With a constantly growing number of IoT devices in the world, we would need to process a lot more data than we needed before. For the system to handle a high amount of traffic, it would need to be auto-scalable so that computing resources can shrink and expand according to the needs to minimize human intervention. The main focus of this thesis is the design and implementation of auto scaling mechanisms for the data processing part of the IoT platform, as the data processor would require computing resources the most.

When designing auto scaling mechanisms or any systems, there should be a measurable goal of how well the system should be performing; this target is known as the *service-level objective (SLO)*. For data processors, one of the desirable characteristics of the system is to start processing data as soon as possible so that clients can have a good user experience and that some actions can be performed on time. A simple example of this is the speed in which an e-mail is sent in response to critical events. Another example is the latency of sensors data information in a visualization user interface of an IoT system. The *message waiting time* could be used for specifying the *SLO*; this is the delay until the data start being processed. The *SLO* used in this thesis is the maximum *message waiting time*.

If the system is well designed for high scalability, conforming to the *SLO* with auto scaling is easy, as the auto scaling mechanism could simply allocate a lot more instances than needed to ensure that *SLO* is met. However, auto scaling mechanisms that use fewer computing resources would be preferable in many cases, especially the ones that prefer computing resource usage savings over a low *message waiting time*. To obtain the highest level of resource efficiency possible, the *SLO* has to be maintained while minimizing computing resource usage as much as possible.

In the 2018 summer semester at the Technical University of Munich, a group of students developed a prototype of an IoT platform for an IoT practical course [2]. The platform is made to be deployed on Kubernetes *container* platform. The architecture of the IoT platform

is used as the reference architecture for this thesis. The important part of that architecture is the usage of the Kafka streaming platform for storing incoming messages from IoT devices. Kafka could also be seen as a persistent message queue. By storing messages in Kafka, the messages can be stored in a fault-tolerant way over a cluster of servers. Moreover, it ensures that every message gets processed by the data processor with the help of Kafka's commit log. Usually, the CPU utilization or system memory usage is used as the metric for auto scaling calculations. However, as Kafka has a semantic of a queue, the number of pending messages in Kafka (*message lag*) is considered to be used for auto scaling calculations.

There are currently many kinds of auto scaling mechanisms in use. The most common type performs scaling by calculation based on the metric value and the target metric value, and it is implemented in many platforms and cloud providers, including Kubernetes (called *horizontal pod autoscaler (HPA)*), Amazon Web Services, and Google Cloud Platform. This auto scaling mechanism works by monitoring certain metrics such as average CPU utilization, memory utilization, network utilization, or custom metrics. The metric is then compared with the target value and the current number of instances proportionally to calculate the desired number of instances.

This work is a continuation of the previous work; in particular, the previous work proposes metrics to be used for auto scaling the data processor, namely CPU utilization, estimated *message waiting time*, and *processing capacity* [3]. The estimated *message waiting time* is an estimation of the time until the last message in the queue starts processing. This is calculated by dividing the number of pending messages in the queue (*message lag*) by the maximum message processing rate of the data processor (*max consuming rate*) [3]. The *processing capacity* is calculated by dividing the current message processing rate (*consuming rate*) by the maximum message processing rate of the data processor (*max consuming rate*) [3]. The metrics are used with the Kubernetes *HPA*, which is an auto scaling solution. The research found that using the estimated *message waiting time* as a metric for auto scaling the data processor could save more computing resources than other metrics tested [3]. However, the maximum *message waiting time* is usually higher than the target *SLO*. The reason for this is because the data processors need time to scale up. Mixed metrics are also tested in [3] by using the highest number of desired replicas among multiple metrics, which is what Kubernetes *HPA* does; but [3] found that mixing in this approach does not add anything useful, as the metric value of the metric that is more sensitive to scaling up will always be used. Combining metrics this way is still useful for combining multiple resource metrics together, as some resources could be fully utilized before another; an example of such a combination is the CPU utilization and memory usage. This new work proposes another approach of mixing multiple calculations by separating calculation for the desired number of instances into scale-up calculation and scale-down calculation instead.

The *HPA* comes with limitations that prevent new innovation of auto scaling mechanisms. This is because it is programmed to scale on fixed sets of rules and calculations—that is, to keep selected metrics below a target value. For example, to keep average CPU utilization over every instance under 70%. In this case, the user can only specify which metric or custom metric

should be below which value, and the user can only customize some runtime configurations. Ways to circumvent those limitations are also considered and described in Chapter 4. Overall, the *HPA* has too many limitations that are hard to circumvent without rewriting the source code. This research solves this issue by implementing a custom-made self-adaptive agent to be used for auto scaling. The concept of a self-adaptive system, an intelligent agent and autonomic computing is used for implementing the auto scaling mechanism as standalone software. It is designed to run the agent on the platform instead of being part of the platform; in this way, there could be more flexibility allowed in terms of calculations, as every line of code could be changed according to needs.

Different auto scaling strategies generate a different number of *SLO* violations and different computing resource consumption. The goal of this research is to minimize the computing resource consumption as much as possible; to do this, this study proposes a few auto scaling strategies for the IoT platform. Such strategies include calculations based on the *producing rate*, *message lag*, and their forecast version. In addition, this study discusses the advantages and disadvantages of each auto scaling strategy. To compare different auto scaling strategies in a fair manner, this study uses the performance indicators from the previous work [3] which are suitable for comparing auto scaling mechanisms in a queue-based IoT data processor. The solutions in this research are specifically made for the prototyped IoT platform, but they could also be adapted for other IoT platforms or other systems. Suggestions on how to apply the solution to other systems are also provided.

2 Background

2.1 Kubernetes

Kubernetes is an open-source *container* orchestration platform for the automation of *container* software deployment, scaling, and management.[1] The smallest computing unit that can be created and managed in Kubernetes is called a pod, and it consists of one or more *containers*.[2] *HPA* is the built-in auto scaling solution that comes with Kubernetes, and it works by recalculating the desired number of replicas repeatedly in a fixed interval. For every interval, the metrics are read from the Kubernetes API server for core system metrics such as CPU and memory usage. Custom metrics are read from an API adapter connected to an external monitoring system. Then the desired number of replicas is calculated. The *HPA* skips the scaling operation if the desired number of replicas is close to 1.0 in a configurable amount. If the desired number of replicas is greater than the current number of replicas, it will scale up at most either twice the current number of replicas or four replicas immediately, whichever is more. It will scale down to the maximum desired number of replicas within a stabilization period; the stabilization period defaults to 5 minutes.

2.2 IoT Platform Architecture

Figure 2.1 is an IoT platform architecture designed by students in the 2018 summer semester at the Technical University of Munich for the IoT practical course [2]. There are four key components for this platform, namely an IoT back-End, an IoT gateway, the Kafka streaming platform, and the data processor. Every component is designed to be run on a Kubernetes cluster.

2.2.1 IoT Back-End

The IoT Back-End is a full-stack web application for managing IoT devices and sensors. For each IoT device, a *JSON web token (JWT)* authentication and authorization token can be downloaded and used in the IoT devices. When a sensor is being created on the web dashboard, a Kafka topic and its corresponding data processor are then created.

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

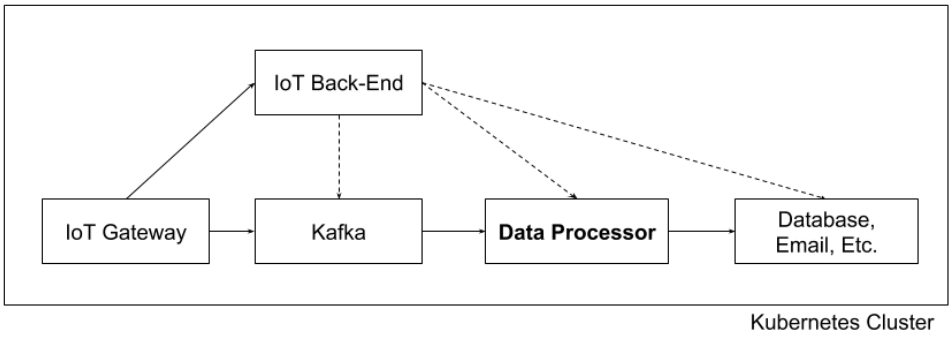


Figure 2.1: IoT Platform Architecture [2]

2.2.2 IoT Gateway

Messages from IoT devices are sent directly to the IoT gateway. There are multiple implementations of the gateway running concurrently to receive messages from different protocols such as MQTT, WebSocket, and HTTP.

Each message is a *JavaScript object notation (JSON)* object with three keys, namely the timestamp (in nanoseconds), sensor ID, and sensor readings. An example message is shown in Snippet 2.1.

Snippet 2.1: Example of the sensor data message

```
{
   "timestamp": 1570222085352146,
   "sensor_id": "5",
   "value": 22.5
}
```

After the gateway receives the message, it checks the *JWT* token set in the protocol for authentication and authorization. For HTTP and WebSocket, the token is placed in the HTTP authentication header; for MQTT, the token is set as the password. If the message is authorized, the gateway will append the message to the Kafka topic.

Apache Kafka

Apache Kafka[3] is a distributed streaming platform that can reliability store sequences of messages. The messages are stored in categories called topics, and a topic can have one or more partitions in which the partitions facilitate the distribution of work in a simple way. For each partition, a sequence id number called an offset is assigned to each message, where the

[3]https://kafka.apache.org

end offset refers to the offset of the last message plus one. When the client sends messages to a topic, the messages are queued to partitions in a round-robin fashion.

To consume messages from a topic, Kafka has a concept of consumer groups where each message in a topic is delivered to only one consumer in a consumer group. One partition is assigned to one consumer at a time, and partitions are assigned equally among consumers in the consumer group. This is very useful for load balancing works between consumers. Not only that, a single topic could be consumed by multiple consumer groups so that a single stream of data can be used for processing for many purposes concurrently. A Kafka consumer group keeps track of processed messages by using a commit log of message offsets. The consumer keeps committing the latest message offsets that it has processed plus one in its assigned partitions. The number of pending messages that have not been processed yet is called the *message lag*, which is the difference between the end offset of the topic and the committed offset.

Data Processor

The data processor is the component that acts on data; it is a stream processor that processes messages sequentially from one or more of the assigned partitions, and it then commits to the Kafka log when it finishes processing. The processing work can be anything such as doing calculations, performing conditional actions, or storing to the database. The data processor can be written in any language that the Kafka client is available in; the data processor needs to connect to Kafka and subscribe to the topic it is meant to process. After it finishes processing each message, a commit call should be made to the client to later inform Kafka that the message has been processed. The data processor program should be packaged as a docker image and be deployed on a Kubernetes cluster. The self-adaptation of the data processor is the part that this research focuses on.

2.3 Self-Adaptive System

A self-adaptive system is a system that can autonomously adapt itself to meet its desired function [4]. In [5], we see self-adaptive systems are classified into four types. Type I systems contain logic for adapting to the environment via an effector from the knowledge that is retrieved from the environment monitor. Type II systems are equipped with multiple strategies that are selected at the runtime by analyzing requirements. Type III systems can develop and construct a new strategy using the knowledge derived from sets of functionalities. Type IV systems are the smartest of every type as it can modify its own code. The idea from all the types of the self-adaptive system could be used to improve a self-adaptive system.

2.4 MAPE-K

A *monitor-analyze-plan-execute with shared knowledge (MAPE-K)* control loop is the design that was first introduced by IBM to be used for autonomic computing [6], which is a system that can manage itself [6]. Figure 2.2 shows the design of the *MAPE-K* reference architecture. It comprises four stages, namely monitor, analyze, plan and execute to control an environment in order to reach its desired goal [6].

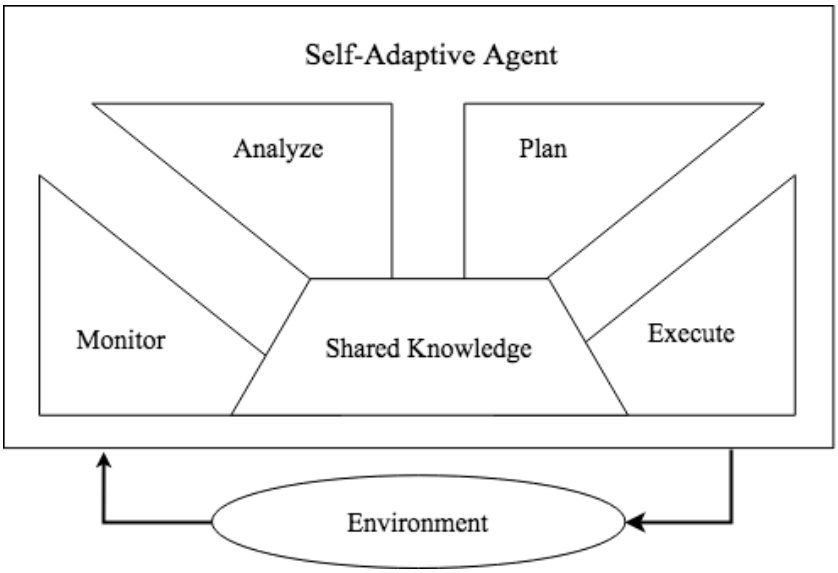


Figure 2.2: MAPE-K reference architecture [6]

2.5 Forecasting

As scaling up the data processor could not be done instantly, forecasting should be used to predict the necessary number of instances in advance so that the *SLO* will not be violated. Double exponential smoothing is used as a forecasting method in this thesis because it is lightweight to compute and takes a constant time to feed new data and forecast.

Double exponential smoothing is an exponential smoothing with trend [7]. The idea of exponential smoothing is to have a factor of how much the latest value should impact the smoothed value than the old value. Double exponential smoothing works by performing exponential smoothing twice—the first time with the value and the second time with the trend [7]. The equation of the smoothed value and the smoothed trend are respectively as follows [7]:

$$s_t = \alpha x_t + (1 - \alpha)(s_{t-1} + b_{t-1}) \tag{2.1}$$

$$b_t = \beta(s_t - s_{t-1}) + (1 - \beta)b_{t-1} \tag{2.2}$$

Where x is the raw value, s is the smoothed value, and b is the smoothed trend.

The initial values for the smoothed value and the smoothed trend are respectively as follows [7]:

$$s_1 = x_1 \tag{2.3}$$

$$b_1 = x_1 - x_0 \tag{2.4}$$

The forecast value at m period in advanced can be compute using the following equation [7]:

$$F_{t+m} = s_t + mb_t \tag{2.5}$$

3 Related Work

There are many areas in IoT where self-adaptation is applied. However, self-adaptation for streaming platforms or queue-based data processing for IoT still has not yet been widely researched. The goal of this related work is to first present the current state of research of the self-adaptive system. And then to explore the existing queue-based auto scaling solutions and the existing benchmark methodology for auto scaling.

The self-adaptive system is capable of improving itself by observing the environment and adapting itself to improve the result. To do this, many researchers use the *MAPE-K* feedback loop for self-adaptation (e.g., [8] and [9]). While using *MAPE-K*, there are many techniques used for analyzing and planning the action to be taken. For instance, the work in [8] uses a finite state machine (FSM) or an artificial neural network (ANN) for planning the adaptation while [9] uses a predefined adaptation plan to deal with the different uncertainties that occurred from the environment. This research chose to derive logic and calculations and used them for analyzing and planning the action.

There are many self-adaptive systems that serve different purposes. In [10], the authors proposed a method to solve the problem in high-performance computing (HPC) environments and cloud platforms. This method allows to partition computations into multiple tasks and assign the work to both HPC and cloud in an efficient and self-adaptive way. This works by monitoring data at runtime, predicting the total execution time, and choosing environments to execute. In the end, all the tasks are synchronized in order to obtain the final result.

In the area of IoT, an agent-based framework for self-adaptive IoT applications named FIoT was created by [8] with two example usages, smart city, and quantified things. For the smart city, the authors experimented with a car traffic light application to control traffic lights; the aim is to have the highest number of vehicles that concluded their route after the simulation ended and to have the application adapt based on the evolutionary algorithm [8]. For quantified things, the authors created an application to predict how many days the bananas will spoil under specific environmental conditions by using supervised learning to select an adaptation strategy [8].

Another example of an IoT-related self-adaptive system is DeltaIoT which applies self-adaptation to be used with multi-hop LoRa radio communication by adapting network settings such as transmission power and spreading factor in order to reduce energy consumption [9].

Forecasting can also play an important role in making self-adaptive systems run smoothly,

as the self-adaptive systems could use the forecast information to scale up or down at the right moment. The authors of [11] studied multiple forecasting models so that they could detect upcoming demand of requests to microservices in order to replicate the microservices in advance. Forecast models that are tested by [11] include various seasonal ARIMA models with GARCH modifications and outliers detection, exponential smoothing models, singular spectrum analysis (SSA), support vector regression (SVR), and simple linear regression. The test for the accuracy of the forecasting models conducted by [11] is evaluated by an interval score. Double exponential smoothing does not require a lot of computing resources to calculate, so it is selected to be used for forecasting the rate of incoming messages in this thesis.

For existing queue-based auto scaling solutions, there is currently a lack of research available. Regarding this topic, this research has only found guidelines that recommend using queue size or queue-size-based calculation as a custom metric for target-tracking-based auto scaling. One of the guides is from Amazon Web Services (AWS) [1] which tells how to use queue size as the metric. The guide recommended using queue size divided by the current number of instances as a metric and to target that metric to be within an acceptable latency (SLO) divided by the average processing time per message.

There is an existing benchmark methodology for auto scaling called BUNGEE [12]. The way this method work is by calculating a gold standard of resources over time and then comparing it with the actual resources over time. However, this method does not work for queue-based data processors. For queue-based data processors, the number of unprocessed messages, and the *message waiting time* of each message needs to be taken into account to calculate the best action at every point in time.

Thus, it can be seen that there is a wide range of self-adaptive systems for IoT as well as many different approaches in deciding how to adapt to the environment. However, the underlying concept of the control loop is nearly the same for many studies by using *MAPE-K* or something similar. The *MAPE-K* reference architecture is used for this study because it satisfies the needs of the system in this research. Lastly, there is the need to conduct research on a specialized solution for queue-based data processing for IoT in order to optimize the system for maximum efficiency.

[1] `https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-using-sqs-queue.html`

4 Motivation

The Kubernetes *HPA* has limitations that could not be circumvented, and for this reason, a custom self-adaptive agent is being implemented to serve the purpose instead. *HPA* has a limited number of configurable settings. This includes selections of metrics or custom metrics, their target metric value, and runtime configurations. There are mainly two problems with *HPA* that concern data gathering and calculation.

For data gathering limitations, only the HTTP pulling of numeric values in Prometheus format can easily be done. The design of the metric monitoring pipeline architecture of Kubernetes is that if a custom metric is needed, any monitoring system should be set up separately and an API adapter should be used to translate it to metric API in Kubernetes format.[1] The only monitoring system that has good integration with Kubernetes is Prometheus. Prometheus obtains metrics data by performing HTTP fetching in a fixed interval to retrieve metrics data in Prometheus format. If fetching metrics in another format is needed or if pushing data is preferred, another program could be implemented as an adapter, but doing this will decrease the data freshness.

For calculation limitations, *HPA* was not made to support a custom calculation of the number of desired replicas. The equation that calculates the number of desired replicas is as follows:[2]

$$desiredReplicas = \lceil currentReplicas \times \frac{currentMetricValue}{desiredMetricValue} \rceil \tag{4.1}$$

If custom calculation is needed, one way to circumvent this is to set *desiredMetricValue* to 1, calculate the desired number of replicas beforehand and provide the metric value to Prometheus (*currentMetricValue*) as *desiredReplicas* divided by *currentReplicas*. In this way, the *currentReplicas* will be canceled out and the chosen *desiredReplica* will be used. However, in order to get *currentReplicas*, we need to fetch some API and then push the metric value to the monitoring system. The time that the metric value is calculated and the time that the *desiredReplicas* are calculated are different. So the *currentReplicas* to be used for canceling the term out might not be equal to the *currentReplicas* used for calculating the *desiredReplicas*. This could lead to miscalculation where the calculation may not be as expected at all times.

[1] `https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/monitoring_architecture.md`

[2] `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`

The better design would be to allow full customization to the calculation of the desired number of replicas.

Furthermore, with the *HPA*'s equation, if the desired number of replicas is greater than the current number of replicas, Kubernetes will scale up at most twice the current number of replicas or four replicas immediately, whichever is more. It will scale down to the maximum desired number of replicas within a stabilization period; the stabilization period defaults to 5 minutes. These are hard-coded logic inside the *HPA* which are undesirable to be used with Kafka consumers. The reason for this is because the data processors will not be able to consume messages for a short moment when other data processor joins or leaves the consuming group, as they have to rebalance and reassign Kafka topic partitions. Scaling down rapidly will cause a lot of rebalances and the *message waiting time* could violate *SLO* easily. Seeing this issue, there should be a way to set up limits on how many instances can be scaled up or scaled down within a certain duration.

Because of the mentioned limitations, this study attempts to implement a custom-made self-adaptive agent for auto scaling to replace the *HPA*, which solves all the limitations as everything can be customizable. Furthermore, since the findings from the previous work show that scaling calculation based on *message lag* could violate *SLO* easily, this present work uses forecasting to prevent a late scaling action.

5 Approach

5.1 Architecture

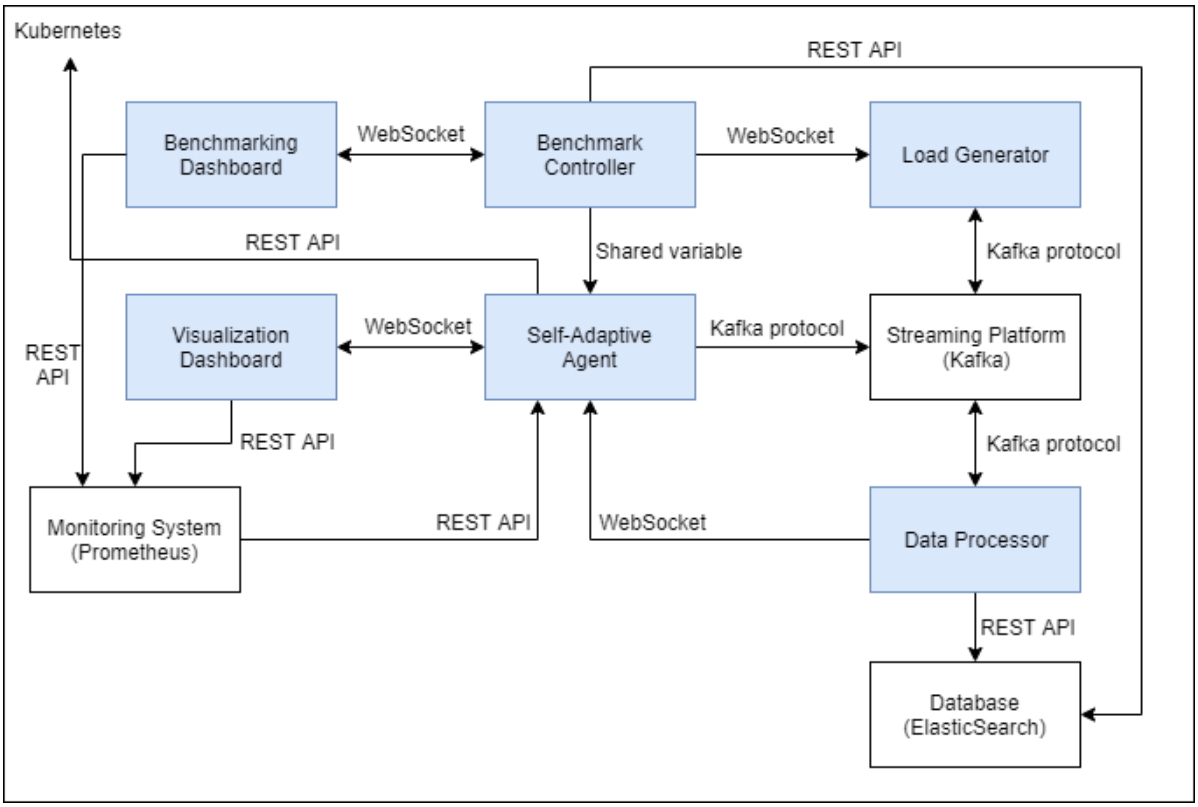


Figure 5.1: The architecture of the whole system

Figure 5.1 shows the architecture of the whole system. This architecture was made specifically for testing and for comparing scaling strategies. For the real IoT architecture, as mentioned in section 2.2, the load generator should be replaced with the IoT Gateway and the benchmark components should be removed.

To start the process of benchmarking, test cases need to be created and submitted in the benchmarker dashboard first. After the benchmark controller receives the benchmark information, it then conducts the benchmark by sending a load specification to the load generator. After this, the load generator will send messages to the Kafka streaming platform according to the load specification.

The data processor works by consuming messages from Kafka and storing them in the database. The self-adaptive agent is an agent that controls the number of data processor instances by creating or deleting the data processor's Kubernetes pods in order to reach the system goal. This is done by a calculation with information obtained from Kafka, Kubernetes, and the data processor.

The Prometheus monitoring system fetches metrics of every component in the self-adaptive agent every second; where metrics are used for displaying graphs on the visualization dashboard and the benchmarking dashboard. The visualization dashboard also shows the real-time state of the self-adaptive agent so that debugging and understanding of calculation could be done easily.

5.2 Self-adaptive agent

To make the system self-adaptive, the self-adaptive agent is made to react to the changes in the environment with *MAPE-K* architecture. The loop is made to run in a fixed interval to save CPU utilization. Moreover, any computations that rely on a uniform time interval could be calculated easily. An example of a calculation that would benefit from a fixed interval is time series analysis, as the time series is usually ordered sequentially and is taken from an equally spaced interval of time. The interval of the loop can be configured in the unit of seconds when enabling the agent. The shorter the interval, the more CPU will be utilized. Wider loop interval will utilize less CPU but will affect the reactivity of the system.

MAPE-K loop works by performing the steps of monitoring, analyzing, planning, and executing in order while having a shared knowledge. The custom-made agent is aimed for high reusability and extensibility. For this reason, the monitor, analyzer, planner, and executor are made as reusable components with predefined interfaces for each stage. This concept came from an idea in dataflow programming and visual programming where sets of reusable components are being connected together. For each stage, there can be multiple components used for performing their desired functionality.

5.2.1 Monitor

For monitoring, every monitor component is run asynchronously in an event loop, and the process of analyzing can start after all the monitor's work is completed. The monitored data is kept in variables that are accessible to every part of the agent as shared knowledge.

The monitor components that are implemented include Kafka consumer group monitor, Kafka offsets monitor, Kubernetes pod list monitor, and data processor information monitor.

Kafka consumer group monitor

Purpose: To query state and consumers' information of the specified consumer group
Parameters: Kafka topic name, Kafka consumer group name
Monitoring output: Object with two keys—state and members
State could be one of the following: COMPLETING_REBALANCE, DEAD, EMPTY, PREPARING_REBALANCE, STABLE, and UNKNOWN.
Members is a list of clients in the consumer group that are currently processing from Kafka. For each item, there are the client's id and list of assigned partitions.

Kafka offsets monitor

Purpose: To query end offsets of the specified Kafka topic and committed offsets of the specified consumer group
Parameters: Kafka topic name, Kafka consumer group name
Monitoring output: Object with two keys—current and consumed
Current is a list of end offsets for every partition.
Consumed is a list of committed offsets of the specified consumer group for every partition.

Kubernetes pod list monitor

Purpose: To query pods information with the specified Kubernetes label
Parameters: The Kubernetes resource's label selector
Monitoring output: Object with four keys—after time, pods, running count, and pending count
After time is a timestamp which ensures that the data is fetched after that time.
Pods contain a list of pod's names, creation time, and status/phase.
Running and pending count is the number of pods that are in each phase.

Data processor information monitor

Purpose: To query data processors and their statistical information about processing time per message
Parameters: Default message processing time
Monitoring output: Object with two keys—processors and sums
Processors contain objects that map the client id to an object with a histogram and percentile function.
Sums is an object with histogram and percentile function.
The histogram of processors is an object which maps message processing time to its frequency.
The histogram of sums combines the histogram of every processor.
The percentile function accepts percentile and returns the percentile value of the histogram.

5.2.2 Analyzer

For each analyzer component, the dependency of monitor values and the dependency of values from other analyzer components are specified so that the order of execution of the analyzer components can be scheduled by using a topological sort. The analyzer components are executed one by one sequentially and not asynchronously in an event loop; the reason for this is because the calculation part has no blocking I/O, and thus running asynchronously will not make the process faster. After all the analyses are finished, the planners will be run.

There are many sets of components that are used for the analysis phase as they are used for multiple tests. The high-level calculations are described later in subsection 5.3.1.

5.2.3 Planner

The planner decides which action to take based on the information from the analysis phase. The decided action is then stored in the shared knowledge so that the executor can execute these actions.

There is only one planner component implemented; this component decides which scaling action to take or it may choose to do nothing based on the desired number of instances from the analysis phase. It will scale up to the desired number of instances if the desired number of instances is greater than the current number of instances. It will scale down one instance if the desired number of instances is less than the current number of instances consecutive for a configurable number of seconds. It will also not scale outside of the range between configurable minimum instances and maximum instances.

5.2.4 Executor

The executor takes specified action from the planner that is stored in the shared knowledge. The executor that is implemented will scale up or scale down if requested from the planner by creating or deleting Kubernetes pods. Kubernetes replica set or deployment is not being used for flexibility in the future so that when scaling down, it will then be possible to choose which pod to delete. This can be useful when different data processors do not process messages at the same rate; for example, deleting the highest processing rate pod might be too much reduction in the total message processing rate.

5.2.5 Visualization

Every component can implement a function of an abstract class in order to provide textual data or metric data to be visualized. The textual visualization data is provided to the visualization dashboard in real-time via WebSocket. The web application was developed using ReactJS[1]

[1] `https://reactjs.org/`

library for building the user interface and MobX[2] for state management. The components are visualized as a dataflow diagram with their short information text from components. The dataflow diagram was drawn using dagre-d3[3] library. If the component is clicked, a dialog will show a full information text and the metric graphs of the component that is predefined for that component. The metric graphs are drawn using Rickshaw[4] library, and the dialog window is created using react-DnR[5] library. Figure 5.2 shows a sample screenshot of the visualization dashboard.

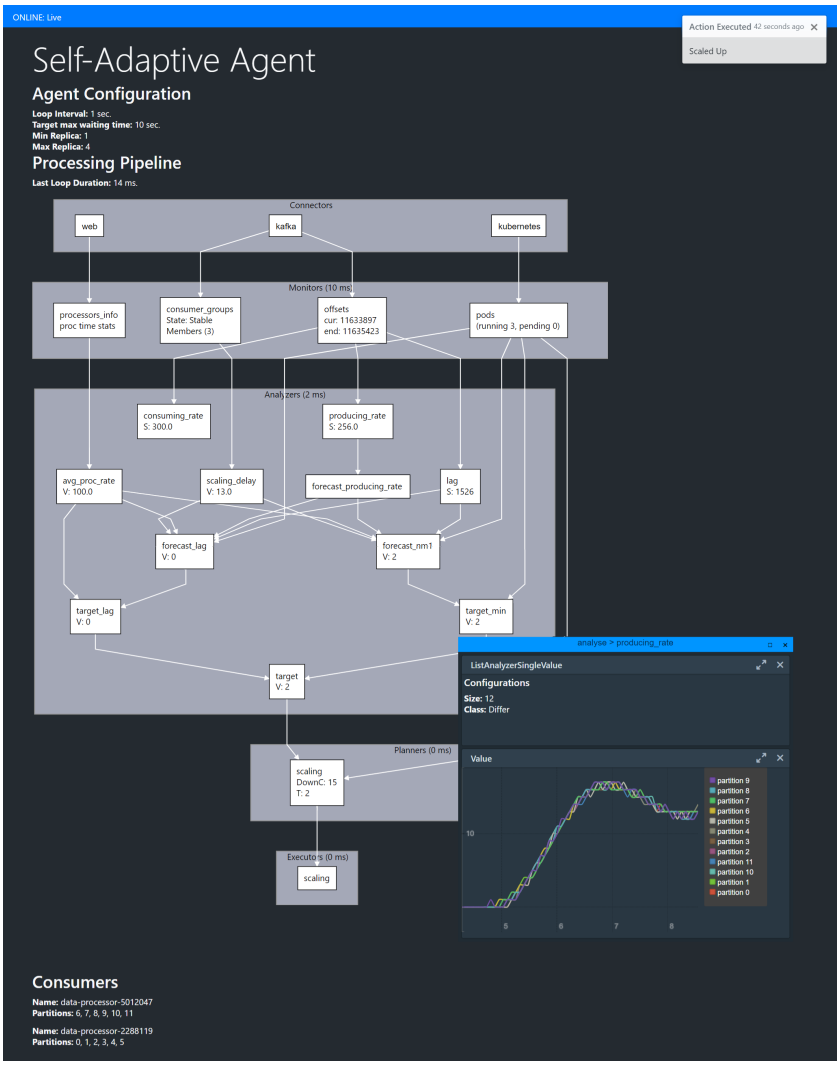


Figure 5.2: Screenshot of the visualization dashboard

This is useful for debugging and for developing scaling strategies because the clearer we

[2] https://mobx.js.org/
[3] https://github.com/dagrejs/dagre-d3
[4] https://tech.shutterstock.com/rickshaw/
[5] https://github.com/yongxu/react-DnR

understand the system, the better we can improve.

5.3 Calculation of the desired number of instances

This section describes the proposed different approaches to calculate the desired number of instances that are suitable for the IoT platform in use. This is the part of the analysis phase where the desired number of instances will be later used in the planner phase.

5.3.1 Calculation based on producing rate

The idea behind this approach is to try to keep the *processing capacity* usage to be within a certain percentage. The *producing rate* is the rate at which the message is pushed into the topic, and it is calculated in this way:

$$producingRate = \frac{Q_t - Q_{t-1}}{loopInterval} \tag{5.1}$$

where

$$Q_t = \text{sum of topic end offsets from every partition at the current loop.}$$
$$Q_{t-1} = \text{sum of topic end offsets from every partition at the previous loop.}$$

Accordingly, the desired number of instances is calculated as follows:

$$desiredInstances = \lceil \frac{producingRate}{DPCR \times percentage} \rceil \tag{PR-B}$$

where $DPCR$ stands for data processor *max consuming rate*, and it is a median of the *max consuming rate* from every data processor. This information comes from the data processor information monitor. *percentage* is the target percentage of the *processing capacity* of each data processor.

Here, *producing rate* is proposed for a few reasons. One reason is that resource-based scaling is used widely in the field of auto scaling, and it would be a good baseline. Another reason is that this method can be useful for a scenario where we want the messages to be processed as soon as they are pushed to the topic.

If the same calculation is to be used with Kubernetes, recall (4.1), which is for calculating the desired number of replicas of Kubernetes. The $currentMetricValue$ can be the percentage of how much the *processing capacity* is used, and it is calculated as follows:

$$currentMetricValue = \frac{producingRate}{consumingRate} \tag{5.2}$$

The *targetMetricValue* can be the target *percentage* of the *processing capacity* of each data processor. Thus, the whole equation becomes this:

$$desiredReplicas = \lceil \frac{currentReplicas \times producingRate}{consumingRate \times percentage} \rceil \tag{5.3}$$

We can see that the equation is similar; the $DPCR$ is replaced by the *consumingRate* divided by the *currentReplica* where *consumingRate* is the total sum of the *max consuming rate* from every replica. For the Kubernetes approach, there is a chance at some point in time that the *consumingRate* does not come from the data processors of *currentReplicas* instances as some replica might not be active and process data yet. This means that the *desiredReplicas* calculated from Kubernetes could be off for a brief moment. How long the value could be off depends on the loop interval as well as the ready status of the pods, as Kubernetes obtain the *currentReplicas* value by counting the pods with a ready status. The readiness probe could be used to specify when the pod is actually started working on its purpose. The readiness probe is checked every configurable period of seconds. This duration of the readiness probe period in seconds is the loophole that imposes miscalculation. From this issue, calculating the desired number of instances directly is better than setting *currentMetricValue* and *targetMetricValue* in the way that Kubernetes does, as there can be more flexibility, efficiency, and correctness.

5.3.2 Calculation based on message lag

For this calculation, the goal is to keep the *message waiting time* to be close to the *SLO* as much as possible and as long as possible in order to save computing resource usage. *Message lag* is the number of messages left unconsumed and is computed with the following equation:

$$messageLag = Q - S \tag{5.4}$$

where Q is the sum of topic end offsets from every partition; S is the sum of committed offsets of consumer group from every partition.

The desired number of instances is the number of instances needed in order to consume the oldest message of the *message lag* within the targeted max *message waiting time SLO*. The equation is defined as follow:

$$desiredInstances = \lceil \frac{messageLag}{DPCR \times maxWaitingTimeSLO} \rceil \tag{ML-B}$$

5.3.3 Usage of different calculation for scaling up and scaling down

The logic and calculations used for scaling up and scaling down should be different in order to have finer control in scaling. The reason that scale-up action is needed is that the current

number of instances is not enough for satisfying the *SLO*, and scale-down action is needed in order to save computing resources whenever possible. If the desired number of instances is set and the data processors become ready to process the messages immediately, we would have near perfect computing resource savings as the number of instances could be changed to the lowest while still conforming to the *SLO* at every point in time. However, this can not be true due to the time needed from scale-up action is being executed until the data processor starts processing is not instant. It takes roughly about 10 to 15 seconds with the environment used for the test.

If we scale down because the number of desired instances drops for a short moment, scaling down may not be worth it because the consumer group has to rebalance; this incurs a brief performance degradation. Another reason is that we might not be able to scale back up in time, and consequently, the *SLO* will be violated.

To separate the calculation into scale-up and scale-down calculation, the following equation can be used without the needs of the conditional statement:

$$desiredInstances = \max(calculation_{up}, \min(calculation_{down}, currentInstances)) \quad \text{(S-UD)}$$

For this equation, the $calculation_{up}$ will have a higher priority than $calculation_{down}$. If we wanted to test for only the scale-up actions, the $calculation_{down}$ can be substitute with ∞ to prevent $desiredInstances$ be decreased from $currentInstances$. If we wanted to allow only scale-down action, the $calculation_{up}$ can be substitute with $-\infty$.

5.3.4 Usage of forecast value

As said before, the data processor takes time to be ready after scaled up, and thus the scale-up action should be performed in advance. Forecasting is needed to achieve this effect. As this adaptive agent loop is designed to run often—as fast as a second to be responsive. For this reason, the forecasting method needs to be able to compute in a fraction of a second, and the double exponential smoothing method was chosen because it does not require a high computing power.

Forecasting for scale-up calculations

When scaling up, the desired number of instances should go up by $delay_{scale}$ seconds in advance where $delay_{scale}$ is the number of seconds that is needed from the scale-up action to reach the ready state of the data processor.

For the forecast scale-up version of (PR-B), this equation is used:

$$desiredInstances = \lceil \frac{producingRate_{t+delay_{scale}}}{DPCR \times percentage} \rceil \quad \text{(U-FPR-B)}$$

For the forecast scale-up version of (ML-B), only the *producingRate* is being forecast using double exponential smoothing, while the forecast of the *message lag* is derived from the forecast *producingRate*. The following is the equation for computing *desiredInstances*.

$$desiredInstances = \lceil \frac{messageLag_{t+delay_{scale}}}{DPCR \times maxWaitingTimeSLO} \rceil \tag{U-FML-B}$$

Here, $messageLag_{t+delay_{scale}}$ is computed by a simulation of the *message lag* over time with the information of *producingRate* over time and the total sum of *max consuming rate* from all the data processors. For each iteration of the simulation, this equation is used:

$$messageLag_t = \max(0, messageLag_{t-1} + producingRate_t - consumingRate) \tag{5.5}$$

Forecasting for scale-down calculations

When scaling down, the same calculation used for scale-up should not be used as we do not want to scale-down too early, as the instances may not be able to handle the current load. What should be done is to scale-down a single instance if the agent will not have to scale up for the next n seconds when given $currentInstances - 1$ instances. The n seconds should be greater than $delay_{scale}$ so that the next scale-up could be made in time. The higher the n is, the more stable the number of instances. Having a lower n will make the agent more eager to scale down and save computing resources. However, we should not make n too small as we have to account for the $delay_{scale}$ time that will consume computing resources without processing any messages. We scale down only a single instance at a time because Kafka topic rebalances cost a brief moment of performance degradation.

For the forecast scale-down version of (PR-B), the equation is as follow:

$$desiredInstances = \max_{\forall i \in \{0,1,2,\ldots,n\}} \lceil \frac{producingRate_{t+i}}{DPCR \times percentage} \rceil \tag{D-FPR-B}$$

The (D-FPR-B) will not make any difference to (PR-B) as scale-down calculation while using double exponential smoothing as the forecasting method. This is because the *producing rate* can only be forecast linearly with double exponential smoothing. If the *producing rate* goes down, the forecast slope of double exponential smoothing will be negative, and every $producingRate_{t+i}$ where $i > 0$ will be less than the $producingRate_t$ meaning that the behavior will be the same as (PR-B). However, using the forecast version may prevent unnecessary scale down if used with other forecasting methods.

For the forecast scale-down version of (ML-B), the equation is as follow:

$$condition_1 = \exists_{i \in \{1,2,\ldots,n\}} \frac{messageLagNm1_{t+i}}{DPCR \times maxWaitingTimeSLO} \geq currentInstances - 1 \tag{D-FML-B-1}$$

$$condition_2 = \frac{messageLag_t}{DPCR \times maxWaitingTimeSLO} \geq currentInstances - 1 \qquad \text{(D-FML-B-2)}$$

$$desiredInstances = \begin{cases} currentInstances & \text{if } condition_1 \wedge condition_2 \\ currentInstances - 1 & \text{otherwise} \end{cases} \qquad \text{(D-FML-B)}$$

where $messageLagNm1$ is the same as (5.5), but the $totalConsumingRate$ is $DPCR \times (currentInstances - 1)$.

5.4 Benchmarking

5.4.1 Load intensity model

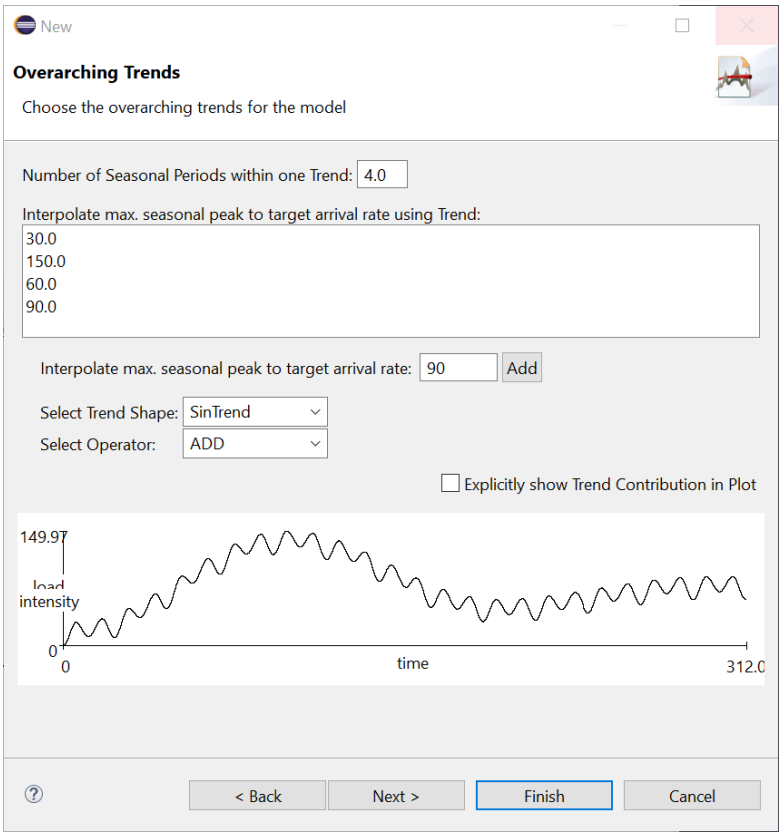


Figure 5.3: LIMBO user interface for trend configuration

LIMBO is a tool for creating load intensity models in a *Descartes load intensity model (DLIM)* structure [13]. It is made as a plugin for Eclipse Modeling Tools[6], and it is used for designing

[6] https://www.eclipse.org/modeling/

load models for benchmarks. *DLIM* describes models by combining trend, seasonal, burst, and noise functions with operators such as addition and multiplication [14]. As there could be so many ways to construct the model, a high-level *DLIM* limits the ways to model to a set of parameters [14], which is easily configurable via LIMBO. Figure 5.3 shows the user interface of LIMBO that is used for configuring the trend part of the model. After the model is constructed, it is being exported as an arrival rate file with a 1-second resolution. The arrival rate file will resemble Snippet 5.1.

Snippet 5.1: LIMBO's exported arrival rate file

```
0.5,0.004386450759028548;
1.5,0.10963721963409512;
2.5,0.3550428317535932;
3.5,0.7401727622845158;
4.5,1.2643513624606442;
5.5,1.9266590449002194;
6.5,2.725933896874551;
...
```

For each row, the first value is the duration since the start time, and the second value is the message arrival rate. The arrival rate file is then converted to the format in *JSON* format that is made for the custom implemented load generator. The *JSON* file will have the structure as shown in Snippet 5.2.

Snippet 5.2: An example of a load generator specification

```
[
   [1,4],
   [2,2],
   [3,1],
   ...
]
```

The left value is the message arrival rate, and the right value refers to how many seconds that arrival rate should last. The arrival rate is rounded up for the converted file so that it is simpler to generate the load and that any peak specified in the graph can be reached at some point in time. The load profiles are used in benchmarks to confirm the hypothesis about the pitfall or advantages of scaling mechanisms.

5.4.2 Load generation

The load generator is implemented to generate loads according to the specification in *JSON* format (e.g., Snippet 5.2). It is a list of message rates to be produced (in messages per second)

and their duration. The load generator will generate messages in the same format as if they were sent from the IoT gateway to a topic in Kafka, in which the messages are distributed to every partition in a round-robin fashion by Kafka. There is also a creation timestamp in the message.

At the beginning of the load generation process, the load generator will remember a start timestamp, and the first message will be sent out. It will then know that the next message should be sent out at $start + \frac{1}{rate}$ seconds. The load generator will wait until that time is reached before sending out the next message. It will keep adding $\frac{1}{rate}$ until every message has been sent out. In this way, we can be sure that the load is generated precisely to specification.

5.4.3 Benchmark controller and benchmarking dashboard

The benchmark controller is responsible for controlling the benchmark and for collecting results. The benchmarking dashboard is a web application that resembles Figure 5.4.

When the benchmark job is submitted through the dashboard, the self-adaptive agent will be enabled with the configured configuration. The old messages data will be cleaned up, and then the load generation job will be sent to the load generator to start generating loads. The benchmark controller will know that the benchmark is complete when the sum of consumer group offsets reaches the total number of messages defined in the load profile specification.

When the data processor starts processing the message, it will add a timestamp, and it will later store the message in Elasticsearch. The benchmark controller will dump all the messages from Elasticsearch in order to analyze the *message waiting time*; the *message waiting time* of each message can be computed by subtracting the timestamp by the creation timestamp. The scaling action history in the self-adaptive agent is also fetched so that it can obtain the number of pods count over time, find the *instance-seconds* used, and get the number of scaling actions executed.

The benchmark result consists of performance indicators and graphs. The main performance indicators include the *instance-seconds*, the scaling action count, and the histogram of *message waiting time*. The graphs are from metrics specified in each component, which are stored in Prometheus. The graphs that are useful for analyzing scaling calculations include instances by time, load by time, *consuming rate* by time, and *message lag* by time. All of the performance indicators and graphs mentioned are the same as the previous work [3].

5.4.4 Data processor

The data processor could be made to perform a variety of tasks, including both I/O intensive and CPU intensive works. For I/O intensive work, this could be file system access or modification; it could also make external calls to other systems such as email and messaging

Figure 5.4: Screenshot of the benchmarking dashboard

systems or cloud-based smart appliances. Cloud-based smart appliances can usually receive commands through its online API such as Remote Hue API[7]. For CPU intensive works, this could be analyzing, decrypting, or compressing data. Both CPU and I/O intensive tasks could take as little as nanoseconds to seconds. It is speculated that many IoT devices will use a third-party network for connectivity and cost-efficiency; one example of this is Sigfox[8]. Thus, end-to-end message encryption would be important in order for the data to be secure. The IoT platform would need to perform decryption of every message; this is ideal to be done by the data processor rather than the gateway in order to save gateway computing resources. It is tested on a laptop with Intel®Core™i7-7Y75 CPU that a Java program could decrypt 512-bit asymmetric encrypted data within about 5 to 15 milliseconds (around 66 to 200 decryption per second). The algorithm used is RSA in ECB mode with a 2048-bit key length. As decryption is likely to be done in the data processor, the data processor is programmed to process messages at a maximum rate of 100 messages per second for testing purposes. This is to make sure that each test will have a similar max message rate and thus can be compared fairly.

5.4.5 Test environment

The tests are conducted on a single-node Kubernetes deployed on RancherOS through KVM. The VM is assigned with all the available CPU cores and 24 GB of memory. The machine has a total of 2x Intel®Xeon®CPU E5-2640 v4 @ 2.40GHz and 128 GB of memory; the sum of the number of cores is 20 cores and 40 threads, which are enough for the test. One data processor pod should use at most two threads; the agent and benchmark controller together should use at most five threads. The load generator, Kafka and Elasticsearch would not over-utilize the rest of the threads.

5.4.6 Common configurations for benchmarks

For every test, the configurable values of the planner as described in subsection 5.2.3 is the same for every test. The minimum and the maximum number of instances is set to 1 and 4 respectively. The planner will scale down one instance if the target number of instances is less than the current number of instances consecutively for 15 seconds. It takes around 15 seconds from the scale-up action until the data processor can start process messages; because of that, n used for calculating (D-FML-B-1) and (D-FPR-B) is fixed to 15 intervals. The α and β for double exponential smoothing are set to 0.8 and 0.5 respectively. The β is set to a lower value than α because it is expected that the trend will not change rapidly nor frequently. These two values are used for demonstration only; for use in production, the values should be fine-tuned to obtain the best result for a certain load profile.

[7] `https://developers.meethue.com/`
[8] `https://www.sigfox.com/en`

5.4.7 Max tolerable linear increment of loads for producing-rate-based calculation (PR-B)

From the result of benchmark I at subsection 6.2.1, it is known that different loop intervals and *percent* result in different *message waiting time* for the same load. So for each configuration, the steepness or the increasing rate of the load that they can handle without violating the *SLO* would be different. *SLO* will be violated when the last message in the topic could not be processed within the defined *message waiting time SLO*. This is analyzed to be the following condition:

$$\frac{messageLag}{DPCR \times currentInstances} > maxWaitingTimeSLO \tag{5.6}$$

This can be rewritten as:

$$messageLag > currentInstances \times DPCR \times maxWaitingTimeSLO \tag{5.7}$$

We can see that the number of current instances also has an effect on how high *message lag* can reach before violating *SLO*. The *message lag* will increase when the *producing rate* is faster than the *max consuming rate* as the queue will be filled. If the *producing rate* is increasing linearly with a slope of m, the equation for the *producing rate* would be $y = mx$ where (0, 0) is the point where the *producing rate* is equal to the *max consuming rate*, as shown in Figure 5.5. After T seconds, the *message lag* will reach a certain amount which is equal to the area under the graph; the area under the graph can be calculated as follows:

$$\begin{aligned} \int_0^T mxdx &= \frac{mx^2}{2}\Big|_0^T \\ &= \frac{mT^2}{2} \end{aligned} \tag{5.8}$$

To prevent the condition (5.7) from happening, the *message lag* needs to be less than or equal to the right-hand side of the equation, which means that this condition must hold:

$$\begin{aligned} \frac{mT^2}{2} &\leq currentInstances \times DPCR \times maxWaitingTimeSLO \\ T &\leq \sqrt{\frac{currentInstances \times DPCR \times maxWaitingTimeSLO \times 2}{m}} \end{aligned} \tag{5.9}$$

This means that after T is greater than the right-hand side of the equation, the *SLO* would be violated, and thus the new data processor has to start processing messages within the time defined in the right-hand side of the equation in order to maintain the *SLO*. The time that the

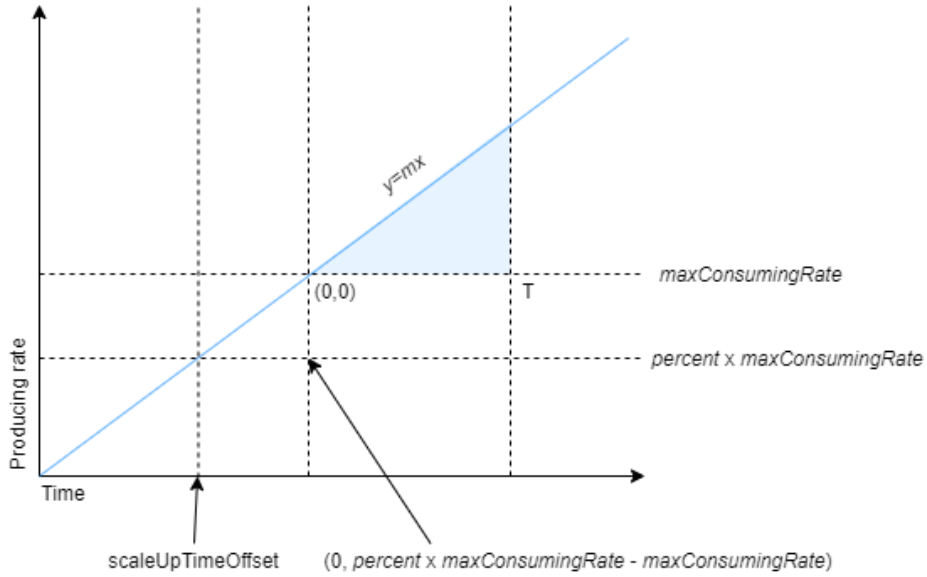


Figure 5.5: Calculation of message lag for linearly increasing producing rate

new data processor will start processing messages depends on three factors. The first is the monitoring delay of how late it knows that it needs to scale up, which depends on the loop interval. The second thing is the scaling delay of how long it takes after the scale-up action has been executed until the data processor is ready to process data. The third thing is the *percent* of (PR-B), which makes the scale-up action occur either earlier or later. This gives us the following equation:

$$T = monitoringDelay + scalingDelay + scaleUpTimeOffset \tag{5.10}$$

For $scaleUpTimeOffset$, we know that the agent will scale up when $producingRate > percent \times maxConsumingRate$, where $maxConsumingRate$ is $currentInstances \times DPCR$. The $y = mx$ where $y = 0$ is where the $producingRate$ is equal to $maxConsumeRate$, so y where the agent will scale up is at $percent \times maxConsumingRate - maxConsumingRate$ as visualized in Figure 5.5. The time that the agent scales up related to the time where $maxConsumingRate$ is equal to $producingRate$ is:

$$\begin{aligned} x &= \frac{y}{m} \\ scaleUpTimeOffset &= \frac{percent \times maxConsumingRate - maxConsumingRate}{m} \\ scaleUpTimeOffset &= \frac{maxConsumingRate \times (percent - 1)}{m} \end{aligned} \tag{5.11}$$

We can then derive the following:

$$T = monitoringDelay + scalingDelay + \frac{maxConsumingRate \times (percent - 1)}{m} \tag{5.12}$$

To get the full tolerable equation, we can substitute (5.12) into (5.9) as follows:

$$\begin{aligned} m &\leq \frac{currentInstances \times DPCR \times maxWaitingTimeSLO \times 2}{(monitoringDelay + scalingDelay + \frac{maxConsumingRate \times (percent-1)}{m})^2} \\ m &\leq \frac{currentInstances \times DPCR \times maxWaitingTimeSLO \times 2}{(monitoringDelay + scalingDelay + \frac{currentInstances \times DPCR \times (percent-1)}{m})^2} \end{aligned} \tag{PR-B-TOL}$$

which is the equation that needs to be true in order to maintain the *SLO*. The maximum m that still makes the equation true is the maximum slope that the (PR-B) can handle.

The solved equation for m is omitted because it is very long. However, if *percent* is 100%, solving for m is a lot simpler, as the $scaleUpTimeOffset$ which contains m is not on the right-hand side of (PR-B-TOL). The equation without $scaleUpTimeOffset$ would be as follows:

$$m \leq \frac{currentInstances \times DPCR \times maxWaitingTimeSLO \times 2}{(monitoringDelay + scalingDelay)^2} \tag{5.13}$$

From an experiment on the tested Kubernetes cluster, the scaling delay is between 10 to 15 seconds. The data processor is made to consume at most 100 *messages per second (MPS)*. Assuming that the max *message waiting time SLO* is 5 seconds and the loop interval is 1 second; if we substitute this into the (5.13), we will get the following:

$$\begin{aligned} m &\leq \frac{currentInstances \times 100 \times 5 \times 2}{(15+1)^2} \\ m &\leq currentInstances \times 3.90625 \end{aligned} \tag{5.14}$$

For the mentioned environment, when there is one instance, the (PR-B) with a *percent* of 100% would be tolerable up to m of 3.90625. This calculation assumes that the *message lag* is zero at the time where *producingRate* is equal to *maxConsumingRate*. This is not always true in the real system, as there will be some messages in the topic even when $maxConsumingRate > producingRate$ because of communication overhead. Moreover, the *message lag* will still be high after scaling, so if there will be a consecutive scale-up action, this needs to be taken into account. This hypothesis is tested for confirmation in subsection 6.2.2.

6 Evaluation

6.1 Self-adaptive agent for auto scaling

By having a custom self-adaptive agent for auto scaling, there is an opportunity to try out scaling mechanisms that were never easily possible with mainstream auto scaling solutions. This agent allows for the flexibility of monitoring, analyzing, planning, and executing that can be fully customized to needs.

6.2 Auto scaling calculation and strategy

Multiple benchmarks are conducted in this research to find the effectiveness of each scaling mechanism in various scenarios. Each benchmark, its configurations, and its reasons are described below. The underlined configuration is used to describe the baseline or control case.

6.2.1 Benchmark comparing different loop intervals and target processing capacities

Goal: To determine how much the different loop intervals affect the *message waiting time* and how helpful the low target percentage of *processing capacity* is to the long loop interval
Loop interval: 1, 2, 4 and 8 seconds
Calculation: (PR-B)
Target *processing capacity* (*percent*): 100%, 85% and 70%
Load: Start from a message rate of 0 *MPS*, increases linearly to 250 *MPS* in 120 seconds, and remains at 250 *MPS* for 120 seconds
Result: Table 6.1

For this benchmark, the loop interval of 1 second and the percent of 100% are used as a control. First, we checked how different target capacities affect the performance of the *message waiting time*. From this test, we can see that by lowering the target capacity to 85%, the computing resource usage in terms of *instance-seconds* increases by just about 4%. This only applies to the tested load where the message rate increases from 0 to 250 *MPS* in 120 seconds. By lowering the target capacity to 70%, the *instance-seconds* increased by about 33% because four instances are required when the *producing rate* goes above 210 *MPS*, and thus a lot more

computing resources are used than other cases. For the performance side, when using a target capacity of 100%, there was about 0.52% of the messages that their *message waiting time* is between 2 and 4 seconds; for other target capacities, *message waiting time* of every message was below 2 seconds. This is because the calculation with a lower target capacity will scales up faster.

For different loop intervals, a higher loop interval will likely have worse performance than a lower loop interval because a higher loop interval is likely to react to changes in the load more slowly. The result also confirms this hypothesis. By lowering the target capacity, it expedites the scale-up action to occur at an earlier point in time, which may account for a high loop interval; but doing so will consuming more computing resources in terms of *instance-seconds*.

However, as the CPU utilization is about 4% for the 1-second loop interval, which is considered to be cheap, it is recommended to use a 1-second loop interval so that the system can be highly responsive to a high increase in loads and thus do not violate the *SLO* easily. Other later benchmarks will be using a 1-second loop.

Table 6.1: Result of the benchmark comparing different loop intervals and target processing capacities

Loop interval	Target capacity	Instance seconds	Scales to	Waiting time (seconds)			
				[0,2)	[2,4)	[4,6)	[6,8)
1 second	100%	576	3	99.48%	0.52%	-	-
2 seconds	100%	578	3	98.85%	1.15%	-	-
4 seconds	100%	577	3	98.60%	1.40%	-	-
8 seconds	100%	566	3	93.56%	6.15%	0.28%	0.01%
1 seconds	85%	601	3	100.00%	-	-	-
2 seconds	85%	598	3	100.00%	-	-	-
4 seconds	85%	598	3	100.00%	-	-	-
8 seconds	85%	593	3	99.82%	0.18%	-	-
1 seconds	70%	768	4	100.00%	-	-	-
2 seconds	70%	761	4	100.00%	-	-	-
4 seconds	70%	755	4	100.00%	-	-	-
8 seconds	70%	735	4	100.00%	-	-	-

6.2.2 Benchmark testing for a tolerable incremental rate of message rate

In theory from subsection 5.4.7, the (PR-B) should be able to tolerate the message rate slope of up to 3.90625 when scaling up from 1 to 2 instances while the max *message waiting time SLO* is 5 seconds and the *percent* is 100%. A test has been conducted to confirm the hypothesis. Message rate with a slope of about 3.83 will be tested instead because it is easier to make the load generator specification with that slope.

Max *message waiting time SLO*: 5 seconds
Calculation: (PR-B)
Target *processing capacity*: 100%
Load: Start from a message rate of 0 *MPS*, increases linearly to 180 *MPS* in 47 seconds (with a slope of around 3.83), and remains at 180 *MPS* for 120 seconds
Result: Table 6.2, Figure 6.1

From the result, we can see that the max *message waiting time* was in the range of [4,5) seconds, which nearly violates the *SLO*. This confirms that the equation from subsection 5.4.7 is accurate to some extent. The peak *message lag* reached up to around 430 messages, which is slightly below the tolerable *message lag* ($currentInstance \times dataPorcssorConsumingRate \times maxWaitingTimeSLO$). The scale-up action is completed in time, thus not violating the *SLO*.

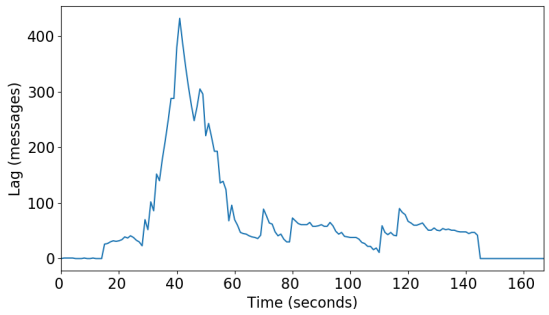


Figure 6.1: Message lag over time of the benchmark testing for a tolerable incremental rate of message rate

Table 6.2: Result of the benchmark testing for a tolerable incremental rate of message rate

Slope	Instance seconds	Waiting time (seconds)					
		[0,1)	[1,2)	[2,3)	[3,4)	[4,5)	[6,∞)
3.83	265	90.60%	6.40%	2.66%	0.29%	0.06%	-

6.2.3 Benchmark comparing different trend functions

Max *message waiting time SLO*: 10 seconds
Calculation: (PR-B)
Target *processing capacity*: 100%
Load: Start from a message rate of 0 *MPS*, increases either linearly, exponentially, logarithmically, or sinusoidally to 250 *MPS* in 120 seconds, and remains at 250 *MPS* for 120 seconds (Figure 6.2)
Goal: To determine how different trend functions affect the quality of the *message waiting time*

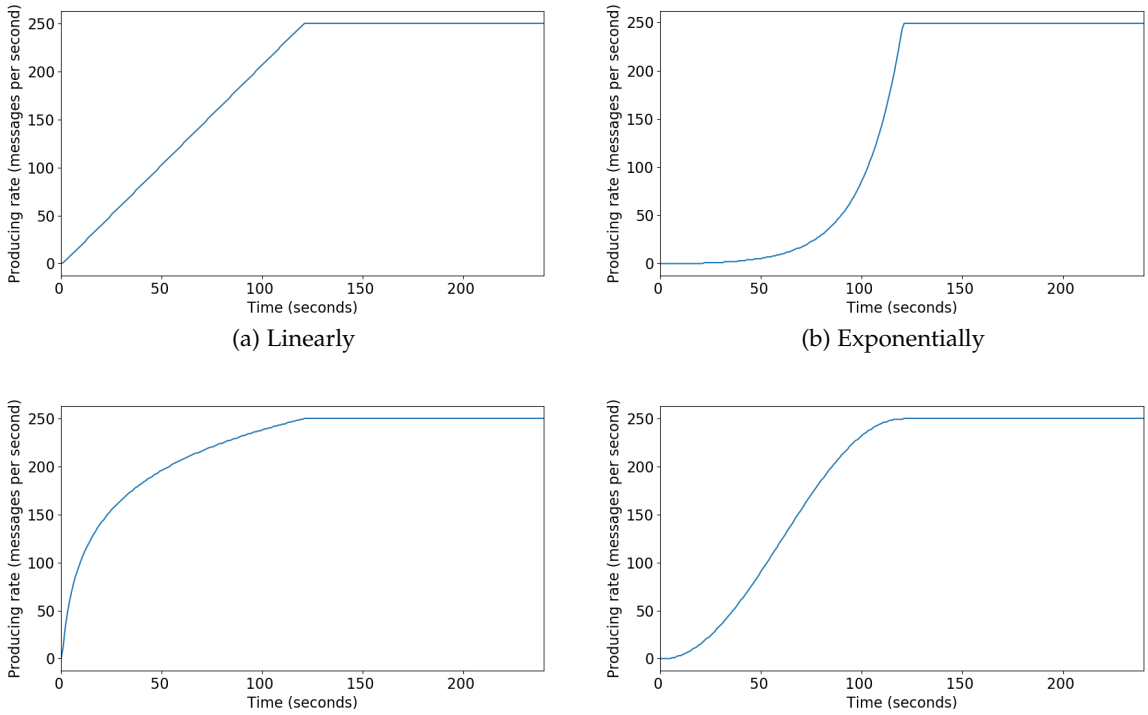
Result: Table 6.3

The slope of logarithmic function and sin function is not much steeper than the linear function, and thus the *message waiting time* lies mostly within 4 seconds. The message rate from the log function increases quickly at the start, so the data processor scales up earlier with this trend than with linear or sin trend, thereby consuming the most *instance-seconds*. The case where the exponential trend is used has the highest max *message waiting time* because of a very steep slope that occurs after about 90 seconds. The total number of messages generated for each test is not the same, so this result should not be used for comparing performances.

Overall, the linear trend and the sin trend are not much different; it can be seen that the slope of the sin trend is slightly steeper and each end of the trend is smoothened in its way. The exponential trend and the log trend also just have steep slope gathered in some parts of the trend. To estimate for the max tolerable increment of load for (PR-B), (PR-B-TOL) can be used for the linear trend. However, for the sin trend, the maximum slope in the trend could be used for making an estimation instead. For log and exponential trends, some parts of each trend could be very steep, so (PR-B) might not be able to handle this well. Forecasting method that supports exponential or log trends would be helpful in such cases, as the system would be able to know the accurate forecast of the *producing rate* in advance.

Table 6.3: Result of the benchmark comparing different trend functions

Trend incrementation	Instance seconds	Waiting time (seconds)					
		[0,2)	[2,4)	[4,6)	[6,8)	[8,10)	[10,∞)
Linearly	576	99.48%	0.52%	-	-	-	-
Exponentially	503	72.99%	13.64%	11.05%	2.28%	0.03%	-
Logarithmically	662	97.95%	1.98%	0.07%	-	-	-
Sinusoidally	588	97.34%	2.63%	0.03%	-	-	-



(a) Linearly

(b) Exponentially

(c) Logarithmically

(d) Sinusoidally

Figure 6.2: Load for different trend functions of the benchmark comparing different trend functions

6.2.4 Benchmark comparing different scale-up calculations

This benchmark tests for scale-up behavior only, and thus the study uses (S-UD), and $calculation_{down}$ is substituted with ∞ to prevent scale-down action.

Max *message waiting time SLO*: 10 seconds
Scale-up calculation:

- (PR-B) with *percentage* = 100%
- (ML-B) with $maxWaitingTimeSLO$ = 10 seconds
- (U-FPR-B) with *percentage* = 100%
- (U-FML-B) with $maxWaitingTimeSLO$ = 10 seconds

Load: Start from a message rate of 0 *MPS*, increases linearly to 250 *MPS* in 120 seconds, and remains at 250 *MPS* for 120 seconds (Figure 6.3)
Goal: To determine how different scale-up calculations affect the quality of the *message waiting time*
Result: Table 6.4, Figure 6.4

From the result, we can see that the *producing-rate*-based calculations (PR-B) or the forecast version of it (U-FPR-B) scales earlier than the *message-lag*-based calculations ((ML-B) and (U-FML-B)) as it scales up when the *producing rate* is greater than the *max consuming rate*. The forecast version (U-FPR-B) scales up to four instances as it forecast that the *producing rate* will go above 300 *MPS*, but it does not do that here and thus utilizes computing resources the most.

The *message-lag*-based calculations ((ML-B) and (U-FML-B)) delay the scale-up action until it will violate the *SLO*. The non-forecast version (ML-B) scaled up immediately when the estimated *message waiting time* is 10 seconds, but that is too late, and thus over 6% of the messages violate the *SLO*. On the other hand, the forecast version (U-FML-B) knows in advance and scales up beforehand, and thus only some messages violate the *SLO*. The *message-lag*-based calculation also uses fewer *instance-seconds*, whereas the forecast version uses the fewest.

Overall, (U-FML-B) would be the preferred choice as it has the greatest control over the performance of the *message waiting time*. However, if there is a limitation in the auto scaling software, both(PR-B) and (U-FPR-B) are suitable to be used. In comparison, (PR-B) would likely save more computing resource usages than (U-FPR-B), and thus it should be considered first by estimating whether it will violate *SLO* or not; this can be done by using (PR-B-TOL) to find the maximum tolerable slope. If it could not tolerate, the forecast version (U-FPR-B) should be used instead.

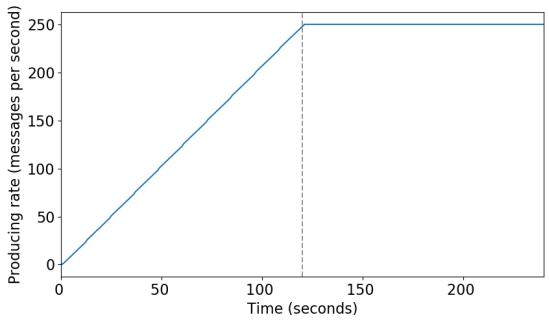
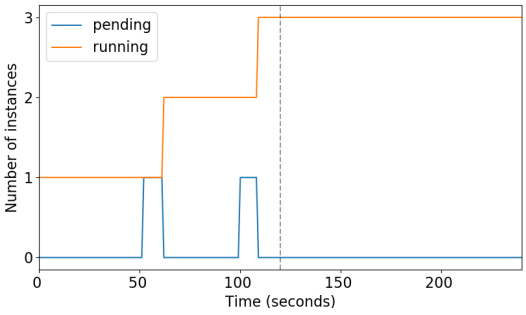


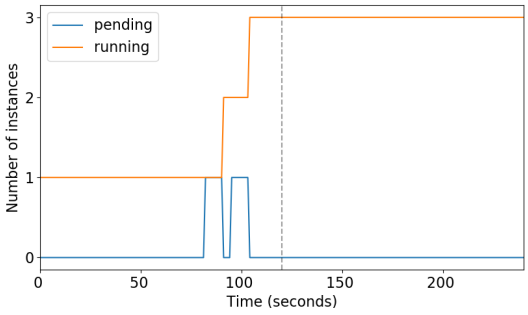
Figure 6.3: Load over time of the benchmark comparing different scale-up calculations

Table 6.4: Result of the benchmark comparing different scale-up calculations

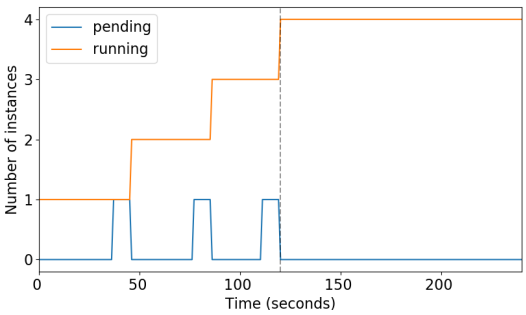
Scale-up calculation	Instance seconds	Waiting time (seconds)					
		[0,2)	[2,4)	[4,6)	[6,8)	[8,10)	[10,∞)
(PR-B)	581	99.14%	0.86%	-	-	-	-
(ML-B)	555	66.22%	9.39%	7.81%	5.75%	4.74%	6.09%
(U-FPR-B)	752	100.00%	-	-	-	-	-
(U-FML-B)	547	53.00%	11.58%	20.69%	11.28%	2.99%	0.46%



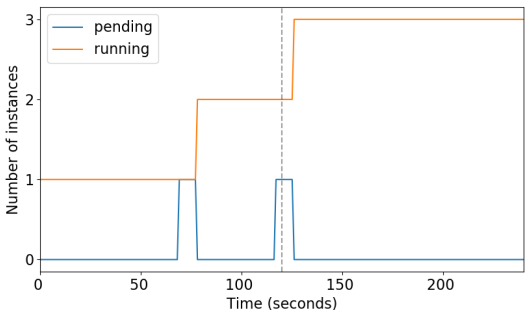
(a) (PR-B)



(b) (ML-B)



(c) (U-FPR-B)



(d) (U-FML-B)

Figure 6.4: Instances by time for each calculation of the benchmark comparing different scale-up calculations

6.2.5 Benchmark comparing different scale-down calculations

This benchmark tests for scale-down behavior only. The (S-UD) is used for separating calculation into scale-up and scale-down calculation, and as the test could only start with the minimum number of instances, we would need to scale up first before testing the scale-down calculations. At first, the same scale-up calculation will be used for every test to scale up to three instances so that there is little to no difference in the performance indicators at this time. The scale-up calculation will then be replaced with $-\infty$ so that only the scale-down calculation is being accounted for.

Max *message waiting time SLO*: 10 seconds
Scale-up calculation: (PR-B) with *percentage* = 100% in the first 120 seconds (until reaching 180 *MPS*), then $-\infty$ is used for the rest of the time
Scale-down calculation:

- (PR-B) with *percentage* = 100%
- (D-FPR-B) with *percentage* = 100%
- (ML-B) with $maxWaitingTimeSLO$ = 10 seconds
- (D-FML-B) with $maxWaitingTimeSLO$ = 10 seconds

Load: Start from a message rate of 0 *MPS*, increases linearly to 180 *MPS* in 120 seconds, remains at 180 *MPS* for 120 seconds, and decreases linearly to 0 *MPS* in 120 seconds (Figure 6.5)
Goal: To determine how different scale-down calculations behave
Result: Table 6.5, Figure 6.6

From the result, (ML-B) is not suitable to be used for any case at all, as it just scales down when the *message lag* is low, which happens easily when the total processing rate is greater than the *producing rate*; this is undesirable because it would scale down too often. For (PR-B) and (D-FPR-B), there is no different while using double exponential smoothing as a forecasting method as mentioned before in section 5.3.4; but using the forecast version may prevent unnecessary scale down if used with other forecasting methods, so (D-FPR-B) should always be used instead of (PR-B). For (PR-B) and (D-FML-B), we can see that (D-FML-B) scale down before (PR-B) for about 32 seconds; it forecast that it can scale down one instance for up to 15 seconds without violating the *SLO*, even before the *producing rate* goes below 100 *MPS*.

Overall, (D-FML-B) is the most suitable for cost-saving and control. However, if the platform could not support that calculation, (D-FPR-B) would be a good alternative. On the next benchmark, we test a scenario where we can easily see how (D-FML-B) can save more computing resources than (D-FPR-B) when it is used as the scale-down calculation.

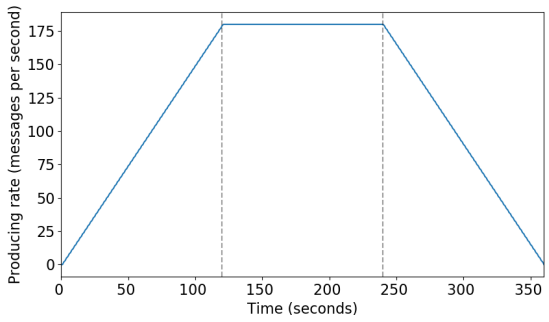
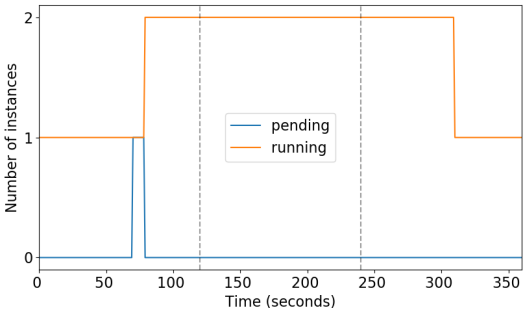


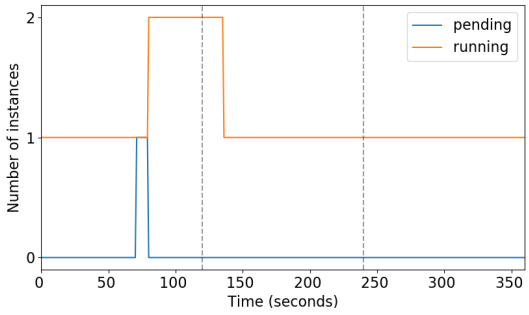
Figure 6.5: Load over time of the benchmark comparing different scale-down calculations

Table 6.5: Result of the benchmark comparing different scale-down calculations

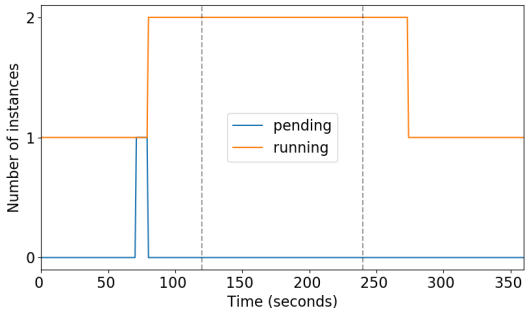
Scale-down calculation	Instance seconds	Waiting time (seconds)					
		[0,2)	[2,4)	[4,6)	[6,8)	[8,10)	[10,∞)
(PR-B)	601	99.93%	0.07%	-	-	-	-
(D-FPR-B)	— " —	— " —	— " —	— " —	— " —	— " —	— " —
(ML-B)	504	31.34%	1.05%	1.04%	1.00%	1.12%	64.44%
(D-FML-B)	563	92.77%	5.75%	1.48%	-	-	-



(a) (PR-B), (D-FPR-B)



(b) (ML-B)



(c) (D-FML-B)

Figure 6.6: Instances by time for each calculation of the benchmark comparing different scale-down calculations

6.2.6 Benchmark comparing forecast-producing-rate-based and forecast-message-lag-based scale-down calculation

Max *message waiting time SLO*: 10 seconds
Scale-up calculation: (U-FML-B) with $maxWaitingTimeSLO$ = 10 seconds
Scale-down calculation:

- (D-FPR-B) with *percentage* = 100%
- (D-FML-B) with $maxWaitingTimeSLO$ = 10 seconds

Load: Start from a message rate of 0 *MPS*, increases linearly to 110 *MPS* in 120 seconds, remains at 110 *MPS* for 480 seconds, and decreases linearly to 0 *MPS* in 120 seconds (Figure 6.7)
Goal: To determine how (D-FML-B) can save more computing resources than (D-FPR-B) when used as the scale-down calculation
Result: Table 6.6, Figure 6.8

Using (D-FML-B) for scale-down calculation saves over 25% of *instance-seconds* from (D-FPR-B), while only 2.06% of messages violate the *SLO*. This is because (D-FML-B) scales down when forecasting and simulation imply that it can scale down while the *SLO* will not be violated within 30 seconds after scaling down. We can confirm the hypothesis that (D-FML-B) can help save more computing resources than (D-FPR-B) when the equation is used for scale-down calculation with (U-FML-B) as the equation for scale-up calculation.

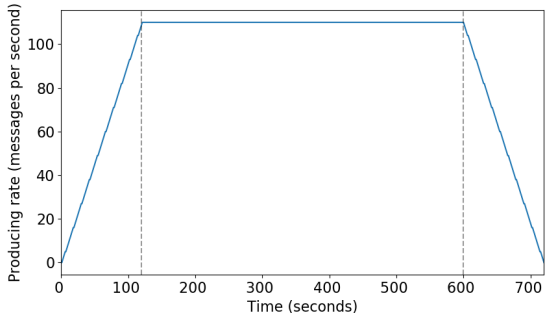
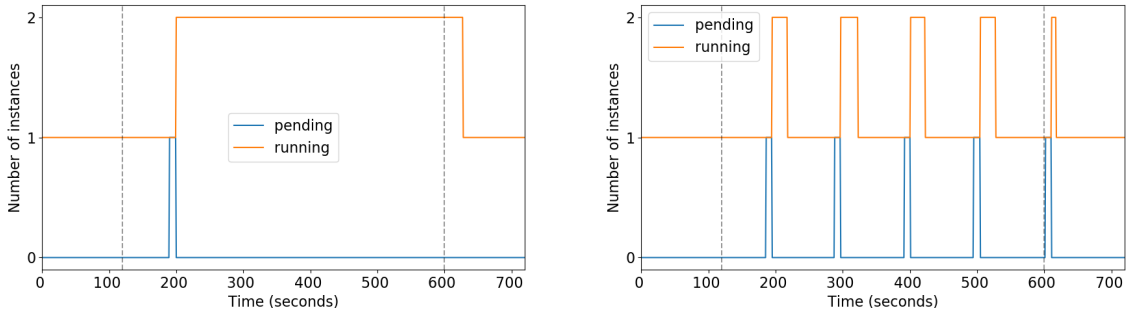


Figure 6.7: Load over time of the benchmark comparing forecast-producing-rate-based and forecast-message-lag-based scale-down calculation

Table 6.6: Result of the benchmark comparing forecast producing-rate-based and forecast-message-lag-based scale-down calculation

Scale-down calculation	Instance seconds	Waiting time (seconds)					
		[0,2)	[2,4)	[4,6)	[6,8)	[8,10)	[10,∞)
(D-FPR-B)	1157	86.80%	3.90%	3.76%	3.22%	1.76%	0.57%
(ML-B)	867	34.26%	19.79%	19.75%	16.29%	7.85%	2.06%



(a) (D-FPR-B) (b) (ML-B)

Figure 6.8: Instances by time for each calculation of the benchmark comparing forecast-producing-rate-based and forecast-message-lag-based scale-down calculation

6.2.7 Benchmark comparing multiple combinations of scaling calculations

Max *message waiting time SLO*: 10 seconds
Scale-up calculations:

- (PR-B) with *percentage* = 100%
- (U-FPR-B) with *percentage* = 100%
- (U-FML-B) with *maxWaitingTimeSLO* = 10 seconds

Scale-down calculations:

- (D-FPR-B) with *percentage* = 100%
- (D-FML-B) with *maxWaitingTimeSLO* = 10 seconds

Load: Seasonal load as in Figure 6.10; parameters are shown in Figure 6.9
Goal: To see how different combinations of scaling calculation behave and to see their resource usages
Result: Table 6.7, Figure 6.11

From the result, we can see that there is actually little to no difference for scale-up calculations of (PR-B) with the two scale-down calculations. This is because (D-FML-B) usually be less than (PR-B). As (S-UD) prioritizes the scale-up calculation, (D-FML-B) does not do anything in this test, and thus the second case will be removed from the evaluation.

This also applies to (U-FPR-B), but (U-FPR-B) is used instead of (PR-B) for scale-down calculation; this makes the scale-down action happen earlier than (PR-B) because it forecasts on the downtrend slope. This is considered to be undesirable behavior as it could scale-down too early and violate the *SLO*. Therefore, the fourth case is will also be cut from being evaluated.

For using (U-FML-B) as the scale-up calculation, the main purpose was to save *instance-seconds* as much as possible. Using (D-FML-B) for scale-down calculation helps to serve this purpose well, and thus this is considered a good match. There is no noticeable advantage for using (D-FPR-B) instead of (D-FML-B) as it is used more computing resources, and the *message waiting time* performance is not much different. Therefore, the fifth case will be cut from the evaluation.

Thus far, we have three configurations left for evaluation, namely the first, the third, and the last configuration. The first configuration could be used for systems where low *message waiting time* is preferred more than computing resource savings; while checking with (PR-B-TOL) for possible *SLO* violations. The third case should be used when the first configuration could violate the *SLO*. Finally, the last case would be suitable for a system that wants to optimize computing resource usage.

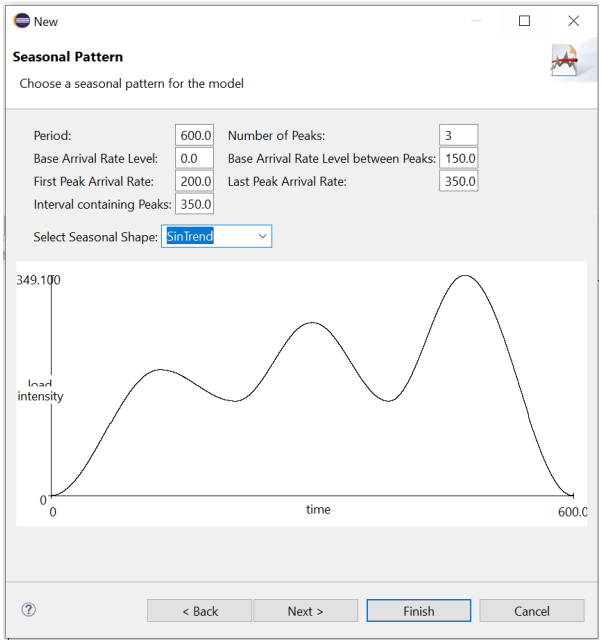


Figure 6.9: LIMBO Load model parameters of the benchmark comparing multiple combinations of scaling calculations

Table 6.7: Result of the benchmark comparing multiple combinations of scaling calculations

Scale-up calculation	Scale-down calculation	# Scaling actions	Instance seconds	Waiting time (seconds)					
				[0,2)	[2,4)	[4,6)	[6,8)	[8,10)	[10,∞)
(PR-B)	(D-FPR-B)	8	1416	93.23%	6.50%	0.27%	-	-	-
(PR-B)	(D-FML-B)	8	1418	97.65%	2.31%	0.04%	-	-	-
(U-FPR-B)	(D-FPR-B)	12	1547	100.00%	-	-	-	-	-
(U-FPR-B)	(D-FML-B)	12	1494	100.00%	-	-	-	-	-
(U-FML-B)	(D-FPR-B)	8	1350	43.53%	14.81%	20.91%	12.61%	5.68%	2.45%
(U-FML-B)	(D-FML-B)	8	1276	37.79%	17.09%	22.23%	12.64%	6.55%	3.69%

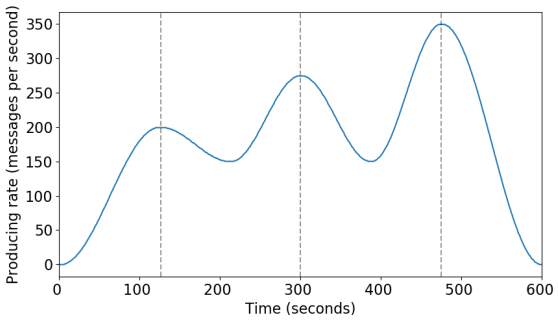
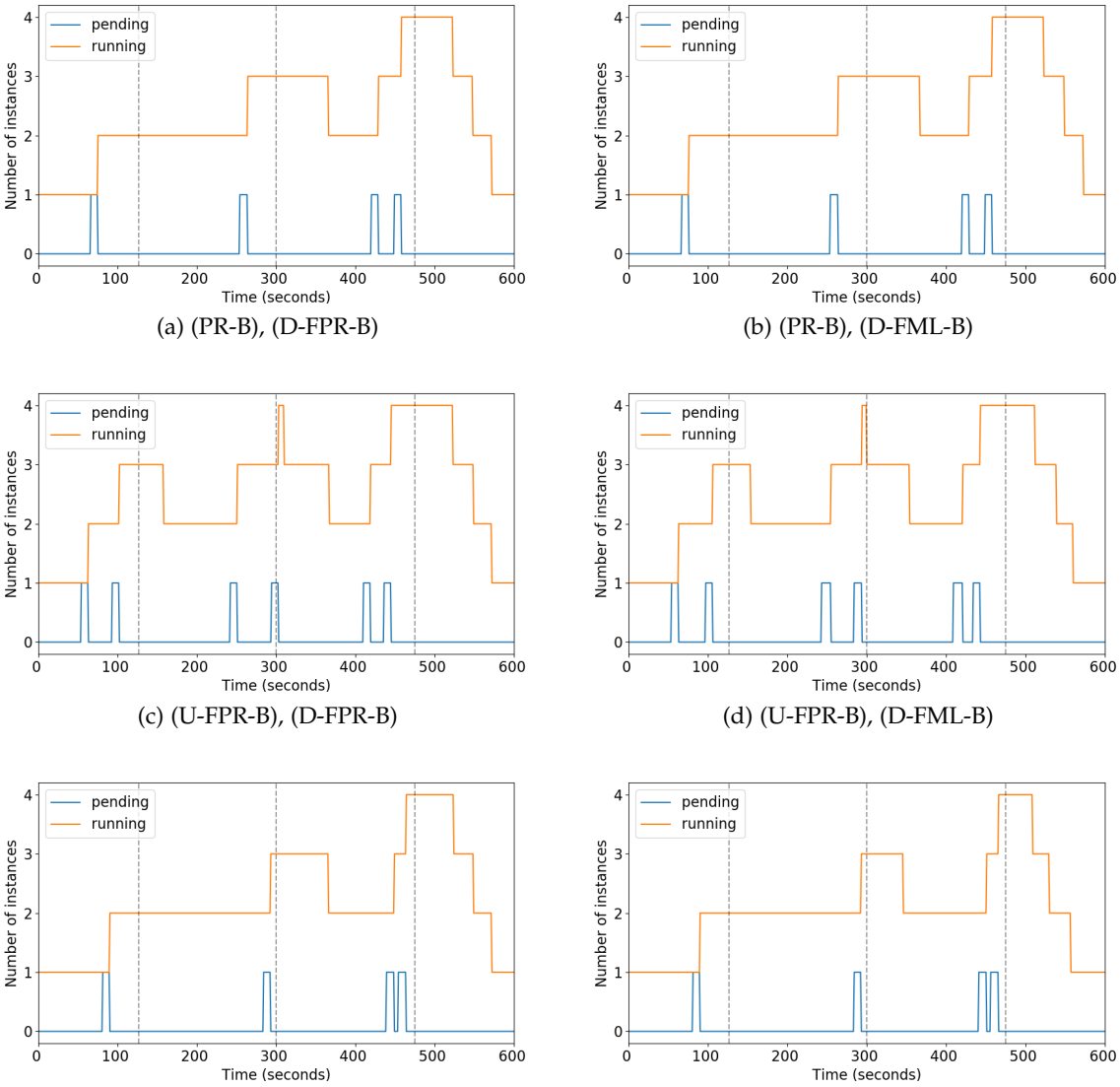


Figure 6.10: Load over time of the benchmark comparing multiple combinations of scaling calculations



(a) (PR-B), (D-FPR-B)

(b) (PR-B), (D-FML-B)

(c) (U-FPR-B), (D-FPR-B)

(d) (U-FPR-B), (D-FML-B)

(e) (U-FML-B), (D-FPR-B)

(f) (U-FML-B), (D-FML-B)

Figure 6.11: Instances by time for each calculation of the benchmark comparing multiple combinations of scaling calculations

6.2.8 Conclusion

According to subsection 6.2.1, the loop interval of 1 second should be used instead of the higher ones because it takes only about 4% of CPU utilization. From all the hypotheses and confirmation with benchmarks, if a low message waiting time is preferred, the (PR-B) with (D-FPR-B) should be used as scale-up calculation and scale-down calculation respectively, while taking (PR-B-TOL) into account. If the calculation could not tolerable the increasing of incoming message rate according to (PR-B-TOL), then (U-FPR-B) should be used instead of (PR-B) for scale-up calculation. However, if *instance-seconds* saving is preferred, using the forecast-*message-lag*-based calculation ((U-FML-B) with (D-FML-B)) for scale-up and scale-down calculation would be the choice. Moreover, if we desire more stability in the number of instances over time, the parameters for scale-up and scale-down calculation could be set differently. For example, we can scale up with a target *processing capacity* of 90% and scale down with a target *processing capacity* of 70%. This is easily possible because of the flexibility of choosing different scale-up and scale-down calculations.

6.3 IoT Platform

The IoT platform relies on the streaming platform to pass around messages. This has the benefit of message reliability and heterogeneity of data processors. The reliability is achieved by message persistency. For heterogeneity, this is because the data processors can be written in different programming languages, and can communicate together via the streaming platform (i.e., Kafka in this case). As Kafka supports multiple consumer groups per topic, multiple aspects can be processed separately and easily while using the same data. This is contrary to a message queue where the message is gone when some client fetches the message; to have a pattern similar to the consumer group, there must be some kind of processor to duplicate messages to multiple queues.

Some IoT platform implementation uses a publish-subscribe solution, and the publish-subscribe solution usually would lose messages when there is no subscriber listening. An example of those platforms is Mainflux[1], which uses the NATS[2] messaging system with no persistence; therefore, if all the subscribers are down, messages that are sent at that moment will be lost. NATS Streaming's queue group should be used instead in that case, as it provides persistence and has a concept similar to the Kafka consumer group. ThingsBoard uses gRPC in its design for monolithic architecture.[3] However, when the new microservices architecture for ThingsBoard was developed, they have decided to use Kafka for persistence and the reliability of communication.[4] The author is also considered to update monolithic architecture to use Kafka instead of gRPC.[5] Other IoT platforms that use microservices approach and use

[1] `https://www.mainflux.com/cloud.html`
[2] `https://nats.io/`
[3] `https://thingsboard.io/docs/reference/monolithic/`
[4] `https://thingsboard.io/docs/reference/msa/`
[5] `https://thingsboard.io/docs/reference/monolithic/#core-services`

Kafka for message passing include DeviceHive[6] and SiteWhere[7].

6.4 Adaptation to be used with other systems

The self-adaptive agent for auto scaling can be used for any kind of system that needs an auto scaling mechanism. A custom monitor, analyzer, planner, and executor components can be added to the agent easily. The executor could be programmed to control anything such as virtual machines, *containers*, and hardware. The calculations that are proposed in this research could be reused for scaling data processors that consume data from a streaming platform or queue if there is a way to monitor the *message lag* or queue length. Moreover, the maximum number of instances that can be scaled down in a certain duration should be defined to prevent performance impact if needed.

[6] https://docs.devicehive.com/docs/devicehive-architecture#section-message-bus-kafka-
[7] https://sitewhere.io/docs/2.1.0/platform/architecture.html#event-data-processing-pipeline

7 Future Work

7.1 Self-adaptive agent for auto scaling

There are many more areas that the self-adaptive agent could be improved. The agent is now adapted only to changes in scaling delay and changes in the data processor processing rate. Improvements could be made by making it a closed-loop system and by adapting it to the actual *message waiting time*. This could be done by sending the *message waiting times* information from the data processors to the self-adaptive agent and adjusting the forecasting time or scaling sensitivity (*percent* or *maxWaitingTimeSLO*) intelligently.

Another thing is to make the agent auto-tune parameters on the fly, such as the α and β values of double exponential smoothing. This could be done by forecasting with multiple α and β values while maintaining an accuracy score for each configuration by comparing the forecast value to the real data.

Lastly, at present, one agent could manage only the scaling of one data processor entity. The agent should be made to support the auto scaling of multiple groups of data processors, and thus a single agent will be enough for managing scaling of every data processor.

7.2 Benchmarking

Currently, the test is conducted by setting up everything in a real test environment and by generating simulated loads. It would be desirable to do everything in a simulation instead of real testing to save time and computing resources.

If a simulation is not possible, the benchmark controller should make multiple tests of the same configuration and combine results to obtain a more stable result.

7.3 IoT Platform

At the time of starting the thesis, the latest version of Kafka is 2.2. It was found out at the very end of this research that Kafka 2.3 was released where there is a new consumer group rebalancing protocol that could lower the performance impact when consumers join or leave.[1]

[1] `https://cwiki.apache.org/confluence/display/KAFKA/KIP-415%3A+Incremental+Cooperative+Rebalancing+in+Kafka+Connect`

This may mean that scaling down more than an instance at a time could be possible with less performance degradation. Therefore, the scale-down limit should be re-evaluated for Kafka 2.3.

8 Conclusion

This research shows that separating scale-up and scale-down calculations allow us to have finer control of the scaling strategy, and this can lead to more intended behaviors. Moreover, it can also help in testing different scale-up or scale-down calculations separately by inhibiting the other action. This study found that using the forecast-*message-lag*-based calculation for scale-up and scale-down calculation could save about 10% of the computing resource usage from the *producing-rate*-based calculation, while about 96% of the messages are within SLO in the test with the generated seasonal load. Overall, applying the self-adaptive agent for auto scaling and the scaling mechanisms with queue-based IoT solutions could help in terms of scalability and efficiency of the system.

List of Figures

List of Tables

List of Snippets

Glossary

consuming rate The rate where messages are fetched from Kafka topic. 2, 24

container An instance of OS-level virtualization. iv, 1, 4, 46

Descartes load intensity model see subsection 5.4.1. 22, 54

horizontal pod autoscaler A Kubernetes solution for auto scaling number of pods by targeting metric value. 2, 54

instance-seconds The sum of the number of seconds that the Kubernetes pod is running. 24, 30, 31, 33, 35, 39, 41, 45

JavaScript object notation A human-readable data-interchange format[1]. 5, 54

JSON web token A standard for signing and verifying *JSON* payload[2]. 4, 54

max consuming rate The maximum processing rate that a data processor can process. 2, 18, 19, 21, 27, 35

message lag The number of pending messages to be processed; for Kakfa, it is calculated by subtracting the end offset of a topic by the committed offset. iv, 2, 3, 6, 12, 19, 21, 24, 27, 29, 32, 35, 37, 45, 46, 49

message waiting time The delay until the message starts being processed. 1, 2, 10, 12, 19, 24, 27, 29–33, 35, 37, 39–41, 47

monitor-analyze-plan-execute with shared knowledge see section 2.4. 7, 54

processing capacity The current consuming rate divided by the max consuming rate. 2, 18, 19, 30, 32, 45

producing rate The incoming message rate to Kafka topic. iv, 3, 18, 21, 27, 30, 33, 35, 37, 49

service-level objective A measurable objective of a service. iv, 1, 54

[1] `https://www.json.org/`
[2] `https://jwt.io/`

Acronyms

DLIM Descartes load intensity model. 22, 23, *Glossary:* Descartes load intensity model

HPA horizontal pod autoscaler. 2–4, 11, 12, *Glossary:* horizontal pod autoscaler

JSON JavaScript object notation. 5, 23, 53, *Glossary:* JavaScript object notation

JWT JSON web token. 4, 5, *Glossary:* JSON web token

MAPE-K monitor-analyze-plan-execute with shared knowledge. 7, 9, 10, 14, *Glossary:* monitor-analyze-plan-execute with shared knowledge

MPS messages per second. 29, 30, 32, 35, 37, 39

SLO service-level objective. iv, 1–3, 7, 12, 19, 20, 27, 29, 31, 32, 35, 37, 39–41, *Glossary:* service-level objective

Bibliography

[1] D. Evans. "The internet of things: How the next evolution of the internet is changing everything". In: *CISCO white paper* 1.2011 (2011), pp. 1–11.

[2] V. Podolskiy, Y. Ramirez, A. Yenel, S. Mohyuddin, H. Uyumaz, A. N. Uysal, M. Assali, S. Drugalev, M. Gerndt, M. Friessnig, and A. Myasnichenko. "Practical Education in IoT through Collaborative Work on Open-Source Projects with Industry and Entrepreneurial Organizations". In: *2018 IEEE Frontiers in Education Conference (FIE)*. Oct. 2018, pp. 1–9. DOI: 10.1109/FIE.2018.8658377.

[3] P. Chindanonda, V. Podolskiy, and M. Gerndt. "Metrics for Self-Adaptive Queuing in Middleware for Internet of Things". In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. June 2019, pp. 130–133. DOI: 10.1109/FAS-W.2019.00042.

[4] D. Weyns. "Software engineering of self-adaptive systems: an organised tour and future challenges". In: *Chapter in Handbook of Software Engineering* (2017).

[5] L. Sabatucci, V. Seidita, and M. Cossentino. "The Four Types of Self-Adaptive Systems: a Metamodel". In: *International Conference on Intelligent Interactive Multimedia Systems and Services*. Springer. 2018, pp. 440–450.

[6] IBM. "An architectural blueprint for autonomic computing". In: *IBM White Paper* (June 2006).

[7] C. Croarkin, P. Tobias, J. Filliben, B. Hembree, W. Guthrie, et al. "NIST/SEMATECH e-handbook of statistical methods". In: *NIST/SEMATECH, July. Available online: http://www. itl. nist. gov/div898/handbook* (2006).

[8] N. M. do Nascimento and C. J. P. de Lucena. "FIoT: An agent-based framework for self-adaptive and self-organizing applications based on the Internet of Things". In: *Information Sciences* 378 (2017), pp. 161–176.

[9] M. U. Iftikhar, G. S. Ramachandran, P. Bollansée, D. Weyns, and D. Hughes. "DeltaIoT: A self-adaptive Internet of Things exemplar". In: *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press. 2017, pp. 76–82.

[10] K. Mantripragada, A. Binotto, and L. P. Tizzei. "A self-adaptive auto-scaling method for scientific applications on HPC environments and clouds". In: *arXiv preprint arXiv:1412.6392* (2014).

[11] V. Podolskiy, A. Jindal, M. Gerndt, and Y. Oleynik. "Forecasting Models for Self-Adaptive Cloud Applications: A Comparative Study". In: *2018 IEEE 12th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*. IEEE. 2018, pp. 40–49.

[12] N. R. Herbst, S. Kounev, A. Weber, and H. Groenda. "BUNGEE: an elasticity benchmark for self-adaptive IaaS cloud environments". In: *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. IEEE Press. 2015, pp. 46–56.

[13] J. von Kistowski, N. Herbst, and S. Kounev. "LIMBO: a tool for modeling variable load intensities". In: Mar. 2014, pp. 225–226. DOI: 10.1145/2568088.2576092.

[14] J. von Kistowski, N. Herbst, and S. Kounev. "Modeling variations in load intensity over time". In: Mar. 2014, pp. 1–4. DOI: 10.1145/2577036.2577037.