UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Rajan Raj Das

Towards Auto-Scaling of Serverless Data Pipelines

Master's Thesis (30 ECTS)

Supervisor(s): Shivananda R. Poojara (PhD)

Tartu 2023

Towards Auto-Scaling of Serverless Data Pipelines

Abstract:
The ever-increasing number of IoT devices generates massive data, and collecting data from heterogeneous sources and processing it without any bottleneck is challenging. Data pipelines are heavily used for automated data processing without any manual hassle. The traditional Data pipelines, such as Extract-Load-Transform, has its own challenges, which are difficult to scale and reduce the timeliness of data processing. It can be solved with the use of serverless computing. Serverless computing is a recent paradigm in cloud computing, It offers granular level scaling of the functions compared to the Virtual Machine (VM). With the increase of smart and Internet of Things(IoT) devices, the use of data pipeline is increased exponentially. However, stochastic IoT workloads and assuring Quality of Service metrics (Latency, throughput, etc.) impose several challenges, including scaling of the underlying infrastructure. Serverless Data Pipelines(SDP) can be designed to process high data volume with efficient resource usage. SDPs comprise several components like serverless functions, message queues, and queue connectors. Scaling the entire pipeline without leaving any bottlenecks is challenging. In our study, we created a serverless data pipeline for an Image Processing IoT application that uses serverless functions to execute the data operation tasks. We also applied different reactive scaling mechanisms, such as resource-based scaling and Workload based scaling, to measure the performance of the scalability on the serverless data pipeline. The reactive mechanisms consider single metrics to enforce auto-scaling configuration, i.e. CPU usage or Request rate. Therefore, we evaluated the use of multiple performance metrics of the Serverless data Pipeline to proactively predict the number of serverless functions in the data pipeline. To experiment with this, we collected data by configuring the reactive auto-scalers, cleaning them to remove outliers, and using them for training and testing the proactive auto-scaler. In this work, we used multioutput regression models, and the results show that the ExtraTreeRegressor algorithm has better efficiency in predicting the pods.

Keywords: Cloud Computing, Serverless Functions, Function as a Service(FaaS), Data Pipelines

CERCS: P170

Iseskaleeruvate serverita andmekonveierite poole

Võtmesõnad:

Tänapäeval kiirendab paljude ettevõtete tegevust andmepõhine lähenemine. Siiski on selliste andmete töötlemine reaalajas osategevuste nagu mitme andmeallika integreerimine, andmete puhastamine, andmete kvaliteedi kontrollimine ja teisendamine, filtreerimine jne tõttu keerukas ülesanne. Samamoodi tekitab üha suurenev IoT-seadmete arv tohutul hulgal andmeid ja heterogeensetest allikatest kogumine ning töötlemine on ilma robotita vaevarikas. Andmekonveiereid kasutatakse laialdaselt automatiseeritud andmetöötluseks. Traditsioonilisel andmekonveieril nagu ekstraktimine-laadimine-teisendamisel on omad väljakutsed - keeruline skaleeritavus ning vähe on võimalusi vähendada andmetöötluse ajakulu. Seda saab lahendada serverita andmetöötluse abil. Serverita andmetöötlus on hiljutine pilvandmetöötluse paradigma, mis pakub funktsioonide detailset skaleerimist virtuaalmasinale (VM). Function as a Service (FaaS) on hiljuti laialdaselt kasutusele võetud on serverita andmetöötluse paradigma. Tarkade- ja Asjade Interneti (IoT) seadmete arvu suurenemisega suureneb andmekonveierite kasutamine hüppeliselt. Asjade Interneti töökoormus on olemuselt juhuslik ning vajaliku teenusekvaliteedi (mõõdikud latentsus, läbilaskevõime jne) tagamisel on suureks väljakutseks aluseks oleva infrastruktuuri skaleerimine. Serverita andmekonveierid (SDP) suudavad tõhusalt töödelda suurt andmemahtu. SDP koosneb mitmest komponendist nagu serverita funktsioonid, sõnumijärjekorrad ja järjekorra ühendused. Kogu konveieri skaleerimine ilma kitsaskohti jätmata on keeruline. Lõime oma uuringus pilditöötluse IoT rakenduse jaoks serverita andmekonveieri, mis kasutab andmetöötluse ülesannete täitmiseks serverita funktsioone. Rakendasime ka erinevaid reaktiivsed skaleerimismehhanismid nagu ressursipõhine skaleerimine ja töökoormusel põhinev skaleerimine serverita andmekonveieri jõudluse skaleeritavuse mõõtmiseks. Reaktiivsed mehhanismid võtavad arvesse üksikuid mõõdikuid, et jõustada automaatse skaleerimise konfiguratsiooni, nt protsessori kasutust või päringute hulka sekundis. Seetõttu hindasime mitme toimivusmõõdiku kasutamist serverita andmetorus, et ennustada serverita funktsioonide arvu andmekonveieris. Selle katsetamiseks kogusime andmeid, konfigureerides reaktiivseid autoskalaatoreid, puhastasime neid kõrvalekallete eemaldamiseks ning kasutasime neid proaktiivse autoskalaatori treenimiseks ja testimiseks. Selles töös kasutasime mitme väljundiga regressioonimudeleid ja tulemused näitavad, et ExtraTreeRegressori algoritmil on ennustamisel efektiivsem.

Võtmesõnad:
Pilvandmetöötlus, serverita funktsioonid, funktsioon teenusena (FaaS), andmekanalid
CERCS: P170

Acknowledgements

I would like to thank my supervisor, Shivananda Rangappa Poojara, for providing constant support, direction, and material.

Also, I thank my parent, Raja Prasad Das and Nitu Das, and my brother, Rajeev Raj Das, for their motivation and energy.

Thanks to my cousin, Aaditya Raj Das, for always directing me in my career and always reminding me of the value of time.

I would like to thank The European Social Fund for partially supporting this work via the IT Academy program by providing the required hardware and especially thank Shivananda for providing access.

Contents

1 Introduction

Function as a Service has gained high popularity in recent years. It is the higher abstraction of cloud computing. It is the kind of cloud computing platform that allows developers to build, compute, run, and manage application packages as functions without having to maintain their own infrastructure[2]. A function is a computation unit running business logic on an operating system in a container. Applications can be composed of many functions to serve an independent purpose. Functions are primarily independent of each other. A function resource is provisioned when an event is triggered, and the resource is released when the function job is completed. This makes it an on-demand service, and the billing happens per the function's resource usage. This serverless computing benefit has attracted the developer's interest and widespread usage. Different fields are using it, for example, computing expensive operations like predicting stock price [Ver19a] that enabled faster processing than the on-premise computers[Ver19b]. According to the report by Reports-and-Data on the Function-as-a-Service market, it is predicted that the market size of the serverless platform can be $31.53 billion by 2026 [520]. It will be adopted in several markets, including Banking, Financial Services, Insurance, Telecommunications and IT, Government and Public Sector, Healthcare and Life Sciences, Retail and Consumer Goods, Manufacturing, and Media and Entertainment. Thousands of functions can run in parallel to solve a business problem. Advancements in 5G and other Telecommunication technologies lead to large-scale use of heterogeneous IoT applications in several domains such as healthcare and smart cities [KJZS17]. It is necessary to collect vast amounts of data in standard format for further analysis. Off-the-shelf data pipeline tools such as Apache Nifi and Apache Kafka are used to design complex data processing pipelines. In the data pipeline, the captured data start with the IoT edge devices, and data moves through processing, transformation, validation, aggregation and selection before moving to the data sink. Some of the operations can be complex and resource-demanding. The operation which needs more resources has to be scaled optimally to avoid bottlenecks in the data pipeline. The data pipelines need constant resource allocation and monitoring, increasing the operation cost. This can be overcome with the use of serverless technology in data pipelines. This is where we can harness the benefit of Serverless computing by combining serverless computing and data pipeline to form a Serverless Data pipeline.

Serverless Data Pipeline abstracts the need to maintain the platform from developers so that developers can focus only on writing core operational logic. Resource Management for data pipelines can be overhead for most developers as several components can be involved. Using a Serverless function can be simplified as function resources are not persistent, i.e. the resource is created on demand. The resource is released to the resource pool when the function finishes its job. The other advantage of using serverless functions in data pipelines is the management of pipeline tasks can be done efficiently. A function can execute each task, and tasks can be scaled by scaling the group of functions. Services

can acquire the computation resource when needed and release them when unused. Hence it improves energy and computing resource conservation by letting machines and storage go when they are no longer useful [M09]. It is estimated by statistically multiplexing the resources in large-scale economies, cloud computing uncovers factors of 5% to 7% decrease in the cost of electricity, network bandwidth, operations, software, and hardware available at this very large economies[LWW+10][M09].

Serverless Data Pipeline does help to abstract resource management. However, the bottleneck operation can still congest the pipeline when the data volume increases. Thus, the selected operational task in the Serverless Data Pipeline can be scaled to increase the pipeline's throughput. Especially in IoT applications, The buffer storage at the Edge and Fog is limited; processing the data helps us to reduce resource strain and decrease the latency of the applications.

1.1 Motivation

Manually managing the scaling policy can be time-consuming and error-prone, as many functions can have different workloads. A set of challenges are identified for introducing automation and optimizing the provisioning of resources while in parallel respecting the agreed Service Level Agreement between cloud and application providers [ZFFP22]. The auto-scaling mechanism can be the best way to increase the system's reliability. There are numerous auto-scaling mechanisms proposed for managing elasticity actions by serverless platforms. These mechanisms do not necessitate any action on behalf of the developer. However, they pose a management overhead on the cloud provider's part since there is still a need for proper configuration of the serverless platform [ZFFP22]. It is best to find the optimal scaling mechanism of the serverless FaaS platform with minimum latency, which will not ruin the user experience.

The configuration set will auto-scale the system based on the event, a reactive approach. Reactive auto-scaling has two classifications, namely, Resource-based and Workload-based scaling. Both of them use application metrics to make scaling decisions. Resource-based considers CPU and Memory Usage, workload-based uses request rate, concurrency, queue length, etc. On the other hand, the auto-scaling mechanism that happens based on historical data to predict the next state of the system by analysing the current state is called Proactive auto-scaling. In a proactive mechanism, we can predict the next state based on the pattern in historical data or use a machine learning agent. Figure1 shows the categorization of auto-scaling techniques based on steps taken to handle the number of running services.

There have been several studies done for auto-scaling serverless platforms, which are discussed in Section 3.1. The proposed solution in those research is to auto-scale the single pod in the serverless platform based on the CPU usage and use of reinforcement learning to obtain optimal configuration. In the literature survey, no approach included a serverless data pipeline involving Edge, Fog, and Cloud layers. Also, there is a limited

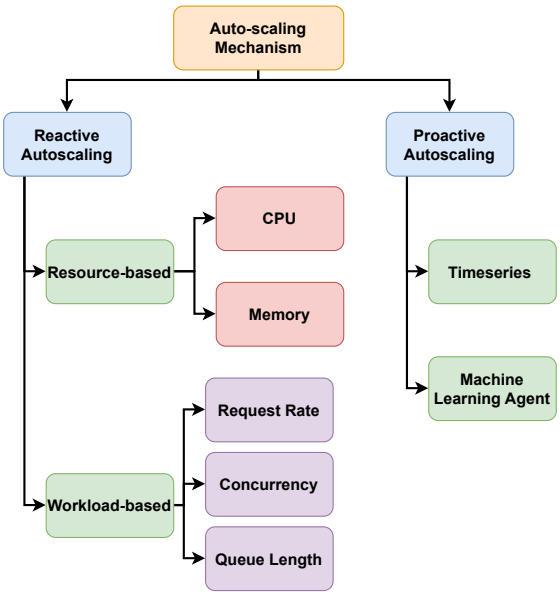


Figure 1. Classification of Auto-scaling Mechanism

study regarding the application metrics of the data pipeline, which combination of which can be used for proactive auto-scaling of application components.

1.2 Goal

Auto-scaling mechanisms can have a critical impact on the Internet of Things(IoT) industry to process large amounts of data [Ete21]. There are thousands of interconnected edge, fog, and cloud devices in a network. All these IoT applications generate heterogeneous raw data that needs to be filtered, transformed, selected, converted, and collect before storing in the cloud environment. This operation can be heavy because of resource constraints in a Fog or Edge environment. IoT applications can send a large number of requests with payload. It is necessary to handle those requests and optimize the use of resources and maintain Quality-of-service i.e. sending data from source to sink as fast as possible.

Due to this, we wanted to find the optimal scaling strategy among resource-based and workload-based scaling strategies and proactively predict the number of serverless functions (pods) in the data pipeline using machine learning by combining application metrics.

1.3 Contribution

Our contribution to this thesis work can be summarized in the following points.

- This study involved designing a Serverless Data Pipeline for Image Processing Data pipelines for IoT applications across Edge, Fog and Cloud environments.
- Applied resourced-based and workload-based auto-scaling to scale the component in the Image processing data pipeline and collected application metrics.
- A quantitative approach to analyze optimal scaling strategy, resource-based and workload-based auto-scaling, in the Serverless Data Pipeline.
- A comprehensive evaluation of machine learning algorithm to predict the number of serverless functions (pods) in the Serverless Data Pipeline using the data pipeline application metrics operated under different configurations.

1.4 Outline

In this chapter, we brief the emergence of serverless computing along with the use of serverless platforms in the data pipeline and the requisite of auto-scaling serverless data pipeline for better Quality-of-Service (QoS). In the next chapter 2, we discuss the background work and state-of-the-art data pipeline system. Chapter 3 describes the literature survey and introduces the research questions. In chapter 4, we discuss the system architecture of the proposed Serverless Data Pipeline with event-driven system architecture. Chapter 5 describes a detailed description of the proposed solution. In chapter 6, we talk about the result gathered and answer the research question. Finally, In chapter 7, we give a conclusion and future discussion of this work.

2 Background

In this section, we see the current state of the data pipeline system. The implementation of the system carried over the years to increase the efficiency of the Data pipelines. In Section 2.1, we discuss the evolution of the data collection for analytical derivation and the need for the data pipeline with the increase in data volume. In the following section 2.2, we describe the current state-of-the-art system and related work that is currently used in the industry and research development respectively.

2.1 Evolution of Data Pipeline

With the increase of the data-driven framework, to validate the hypothesis with data analysis, the industry started gathering a higher volume of data to derive knowledge [Sar21]. The derived knowledge is then used to make decisions. It started with the recording of the Online Transaction Processing(OLTP) application to store transaction data, where a huge amount of the transaction data is stored in the application persistence layer, i.e. hardware, which usually had limited capacity. The data were transferred to the data warehouse with higher storage capability and higher redundancy to store the transaction data. Transferring the data used to be manual, and the frequency of the backup used to be low. Sometimes hardware used to be transferred. In the later stage, the data transfer used to take place at a specific time or day of the week over the Internet. As the internet bandwidth used to be low. The data transfer normally happens when the network traffic was low. The manual intervention in the steps to transfer the data from one location to another result in fault or delay in the process. The risk of privacy and security was also a concern if it was handled manually. The industry moved to an automated approach to overcome this issue, Cron Jobs. These Cron Jobs made the process efficient and timely. The data transfer jobs have to be scheduled during a lower traffic period since it consumes mostly the bandwidth in the network. It has its own flaws and complexity overheads. If the data volume is higher, it consumes higher resources, and the Cron worker has to be configured with enough resources to perform optimally. This raises the overhead of managing and maintaining the service.

The source of data collection and data volume was also increasing exponentially as the use of the internet grew rapidly over the years. The need for faster data processing was a need and necessity in the industry. It was the need because the data collected used to become stale in the process of data analytics and the companies wanted to process the data faster to have a better understanding of the business and provide better service to the clients. The necessity in terms of data volume being generated was higher, and the storage of data became an extra challenge. Some of the data were noisy as it was not used to make decisions. The data has to be stored in the data warehouse along with the user data. Also, the complexity of the data structure increases to represent the real-world scenario of the system. The data need to be simplified so that data analytics and data

visualization can become easy [Den21].

With the emergence of the Internet of Things (IoT) application, the list of challenges has grown longer. IoT applications are distributed across the network i.e. the data are collected from different nodes in the network, and the nodes could be heterogeneous in nature. The IoT application has a heterogeneous protocol and framework [KJZS17]. The architecture of Edge, Fog and Cloud applications could be different based on the resource constraint and use case. Usually, all the layers are heterogeneous in nature. So data synchronicity and data flow emerged as a bigger challenge. This challenge is solved by Extract Load Transform (ELT) based architecture [PS21]. This architecture is very dominant in the industry, and it is still being used in the industry.

2.2 State of the Art

There are different approaches to solving problems faced during data transfer from source to sink. The solution has been developed and optimised over the last 2 decades based on the use cases. There is still research going on to further optimize the pipeline. There are several proprietary and open-source tools emerged to achieve this result. While some enterprises plan on developing and maintaining specialized internal tools, they can drain the organizations' resources to manage them, especially when data circulates in multi-cloud environments. As a result, some businesses will turn to third-party vendors to save these costs [Kni23][PS21]. Third-party vendors like AWS, Azure, Cloudera, etc. have their own flavour of data pipeline tools with different unique features. For example, Amazon offers AWS Glue as ELT based data pipeline with an add-on as a Machine Learning approach to duplicate and clean data. It uses an Apache Spark serverless environment to run the jobs. Likewise, Azure provided a service for orchestrating data movement and transforming data between Azure services called Azure Data Factory. Likewise, Cloudera provides a similar service based on Apache NiFi.

2.2.1 Serverless Data Pipelines

Data Pipeline is a series of synchronous or asynchronous chain of task that modifies the data along the chain, modified data of one task is fed to another task in the chain till all the data residue to the sink. It is the smooth automated flow of data from source to destination [RBOW20]. Data pipelines are widely used in the industry focused on data. Almost all of the Information technology industry uses a kind of data pipeline for monitoring, migration and reporting, with the use of a pipeline for machine learning, data governance, and data aggregation.

Serverless computing is a relatively new model of cloud computing. Serverless Architecture is built upon virtualization technology. The term serverless computing does not mean missing the server in the cloud environment. The resources for the computation are allocated on demand when the event is triggered. The core benefit of a

serverless platform is that it can spin up a function with provisioned resources and release the resource when the function execution is completed. This abstraction of serverless computing promotes decoupling function run.

Serverless data pipelines combine the capability of both serverless computing and data pipelines to deliver value efficiently. The task of data manipulation or data quality check in the chain of activity can be assigned to a single function with a relatively shorter life and efficient use of resources. The serverless function can be deployed in Cloud, Fog or Edge platforms, and data pipeline technologies are used for data transport, routing and function invocation [LKRL19].

2.2.2 Scaling Mechanism of Serverless Functions

Auto-scaling service in the cloud environment has been the topic of research for a long time now. The researchers are working on defining a general-purpose scaling mechanism for production-ready systems. Scaling rule deciding factor of scaling a service. Scaling laws describe the functional relationship between two resources that scale each other over a significant interval. The rule can be as simple as scaling service based on the CPU and Memory percentage. The scaling can be horizontal by changing the running function number for computation, or the scaling can be vertical by changing resources to a running function. For example, Kubernetes default scaling rule to calculate the instances of service (s) to be deployed in the next scaling interval.

$$n_{t+1}(s) = \lceil n_t(s).\frac{m_t(s)}{m^*(s)} \rceil \tag{1}$$

Here, n_{t+1} is the number of instances to be deployed in the next interval, while n_t is the number of currently deployed instances. The measured value of the scaling metric averaged over all instances of the service is denoted as m_t and the desired metric value is denoted as m^* [SGvK$^+$22].

Auto-scaling mechanisms can be classified as Reactive Mechanism and Proactive Mechanism. The Reactive mechanism defines the static rules for scaling the service based on resource consumption. The resource consumption can be CPU resources or Memory Resources. When the application exceeds the expected threshold of the resources, it scales the service horizontally or vertically. For example, The scaling rule of a service is defined as 80 % CPU usage implies it will scale the resource when the CPU usage of the service exceeds 80 %. It can allocate more CPU core in case of scaling vertically or it can scale horizontally by spinning identical instances of the service to manage the incoming load. It is a widely used scaling mechanism in the industry. As it is easy to configure the scaling rule. The Service Level Agreement(SLA) for these scaling mechanisms can be low because it takes time to allocate new instances in the cloud environment. Thus, the service faces delay.

The Proactive mechanism is the state-of-the-art auto-scaling system. It is an active research topic in the cloud computing community. Researchers are focusing on finding production-ready solutions for predictive mechanisms. Currently, There is timestamped-based auto-scaling which predicts the system's workload based on the pattern in history. The data of the system usage is stored in the time-stamped database, and decisions are made based on the history. Currency Proactive solutions are more application dependent than general-purpose solutions. Another proactive approach for auto-scaling is using a machine learning agent. The agent learns from the monitoring log.

3 Literature Survey

As a part of the literature survey, Scopus is chosen as the database for the research paper. It has good indexing capabilities with quick-to-find relevant and authoritative research. It also provides a good interface for the research paper query with easy-to-use filters. The literature review was done in order to answer the following question.

RQ1: What is the optimal auto-scaling strategy, from resource-based scaling or workload-based scaling, for serverless cloud functions in the data pipeline?

RQ2: How to proactively predict the number of dependent pods in-chain using a machine learning combined metrics approach?

Cloud computing has been a common terminology in many several fields like Healthcare, Telecommunication, Banking, etc. In order to answer the research question mentioned above the keyword from the research question is extracted to form a query string. The keywords such as serverless, auto-scaling, mechanism, configuration and throughput are searched against all metadata, full-text, author, publication, etc. In many papers serverless and FaaS could be used interchangeably with auto-scaling mechanisms, configuration and throughput are added to filter relevant and narrow down the scope of the research paper. The query string is formed as followed:

```
ALL (
   (serverless OR FaaS) AND
   ( data-pipeline OR data-analytics) AND
   (auto-scaling OR mechanism) AND
   (configuration OR optimise)
PUBYEAR > 2017 AND
PUBYEAR < 2023
)
```

There were 74 result papers from the above research paper. The quality of the research paper was determined by Inclusion and exclusion criteria. This also helped to reduce the number of research papers to read without losing any good quality research papers. The inclusion and exclusion criteria are as follows

Inclusion Criteria:

IC1: Papers related to FaaS serverless technology.

IC2: Papers related to data pipeline or data processing.

IC3: Papers related and resource management scaling mechanism.

Exclusion Criteria:

EC1: Papers before 2018.

EC2: Papers are not conference papers and articles.

EC3: Papers which are not in English.

After applying inclusion and exclusion criteria, we were left with ten research papers analysed. We also used the snowballing method to find more related papers with similar concepts.

3.1 Related Work

This section describes the current research in data pipelines and serverless computing platforms. We will analyse the state-of-the-art system for serverless computing environment and data pipeline with their goals.

Poojara et al. [PDJS22] in their research have data processing at the Edge, Fog and Cloud layer using serverless cloud functions. There is a demonstration of heterogeneous hardware used at different layers and the serverless function to streamline data processing. He compared the performance matrices of different tools and storage solutions for data processing at the fog layer to make data processing closer to the data source. Data Flow Tool (Apache Nifi), Object Storage Service(MinIO) and Message Queue (RabbitMQ) were used to channel the flow of data in the data pipeline. The performance matrices were measured using a variety of applications with different bandwidth consumption, like Aeneas for audio processing, PocketSphinex for audio processing and Yolo for video processing applications. The research concluded best-fit tool to channel for different applications based on resource utilization and processing time.

As the succession of the research related to measuring the performance metrics, Poojara et al.[PDJS22] have narrowed down the scope of research on the fog layer. In the later research, he analysed scaling approaches for serverless functions in the fog layer. In this research, Resource-based scaling and Work load based scaling was discussed. He conducted experiments to evaluate the performance of the serverless data pipeline under several CPU utilization configurations, Incoming Message Rate and Function Invocation Rate. The research concluded that workload-based scaling has higher throughput with higher resource utilization, whereas resource-based scaling has consistent throughput with moderate resource utilization.

Benedetti et al. [BFRS22] have researched the applicability of resourced-based auto-scaling in serverless edge applications using reinforcement learning. In the research, he explored the optimal CPU threshold configuration for resources-based auto-scaling. To

achieve that, They initially run the application in a serverless function under a selected CPU threshold ranging from 10-90% as the based experiment to compare the metrics and find the initial feed for the machine learning agent. They repeated the same experiment using a machine learning agent(Q-learning) to configure the CPU threshold after each experiment based on the feedback from the previous iteration. In this research, they concluded the optimal CPU configuration for the application based on the minimum latency of the application.

In this chapter, we look into different research that has been done in the field of serverless platform and research that are related to auto-scaling. In the next chapter, we shall see the system architecture of the proposed data pipeline.

4 System Architecture

We propose the high-level architecture of the Serverless Data Pipeline infrastructure distributed over different layers. For the study, we designed Image processing data pipeline application,

4.1 Event Driven System Architecture

Event-driven architecture (EDA) is a system design practice built to record, transmit, and process events through a decoupled architecture [1822]. Event-driven architecture is a widely adaptable system architecture. The component in the system is loosely coupled and starts the action when an event is triggered. The event manager manages each event and triggers the subscribed topic based on the event listener. The event manager is capable of distributing the incoming load based on configuration. In the image processing application, each component starts the operation when the event from the previous task is finished and passes the operation to the forthcoming component. This architecture helps us to make a scalable system. The system is scaled horizontally because the operation is independent of each other. In our application, we tried to scale the application based on different scenarios.

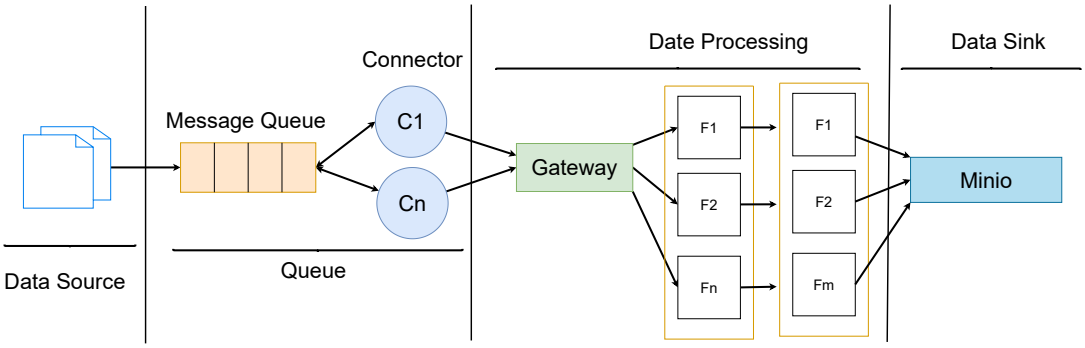


Figure 2. Event-Driven Architecture of Data processing pipeline

4.1.1 Data Source

In the image processing application, the data source is generated by a sensor in the edge device. The edge device sends the encoded image to the queue in a nearby fog environment. There could be thousands of edge devices connected to an edge device. This event-based approach was used. It makes the data source independent so that there are no persistent connections which waste up the resource. We also use the fire technique and forget so that the source does not have to wait for acknowledgement from the fog system. In our use case, we have used locust to mock the load generated by the edge

device, and the load runs for 100 seconds with an increasing spawn rate (makes the parallel request).

4.1.2 Queue

The queue mechanism is introduced in the early stage of the pipeline to streamline a load of requests from edge devices. The message arriving are assigned a designated topic in the message broker. A message queue supports several topics based on the use case. Each of these topics is connected to a queue connector which triggers configured endpoint based on the specified topic. There can be several instances of connectors running in the system to void the queue bottleneck. This also increases the consumption of the message in the queue.

4.1.3 Transformation

In most applications, transformation is done by single micro-services with several business logic layers. The micro-services keep blocking resources when the pipeline is not in use. In the image processing application presented in the research paper, we aim to optimise resource usage. To achieve this goal, we used serverless functions which encapsulate the business logic, and resource for calculation is allocated when the function is triggered. This helps to optimise the resource pool in the cloud and helps manage the code by managing atomic business code inside a function template.

The transformation stage has services of function which are triggered when the job of the prior function is finished. Since the function is stateless, the calculated result from a function is transferred to subsequent functions as REST API. During the data transformation stage, redundant data could be lost to achieve high-quality data at the sink.

4.1.4 Data Sink

The high-quality data is stored at this stage for future reference and analysis. The data sink can have a persistent connection or an on-demand connection. In image processing applications, the transformation stage triggers the mini object storage service with the data load that has to be stored. This decouples the application from the storage service. Also, the storage service can be scaled independently for higher availability and consistency.

4.2 Layered Approach

In IoT applications, data are gathered from different edge devices which can be present in different geographical locations. Hence, the resource bandwidth will be limited.

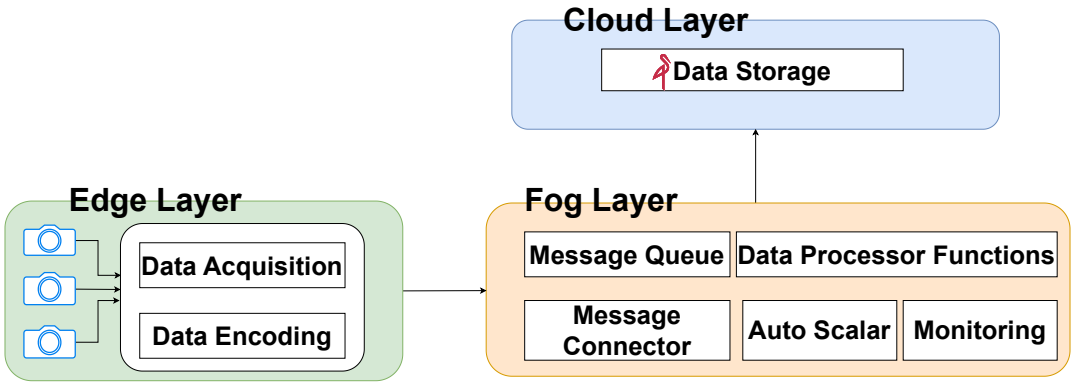


Figure 3. High-level Layered system architecture of application

Dividing the responsibility of processing data in a layered approach will help us to optimize resource consumption and ensure that any component will not be overused. In figure 3, we have described the high-level architecture of the data pipeline and the components involved in each layer.

4.2.1 Edge Layer

Edge computing Layer is an architectural paradigm of distributed computing. The edge computing layer encompasses that infrastructural component near the source and is dispersed from the central location of the data centre [1522]. The infrastructural component includes compute and storage capabilities, sensors and network connectivity back to the centre. This architectural paradigm benefits low latency, high bandwidth and real-time processing.

The edge layer has the basic responsibility of processing the captured image data. In this layer, the image is encoded to transfer over the network. Encoding makes it efficient to transfer over the network because transformed data makes it proper and safe to consume by a different type of system.

4.2.2 Fog Layer

The fog computing layer is a decentralized computing paradigm in which data, computing, storage and applications are located between the data source and the cloud [Pos]. The fog layer has more computing and storage capability than the edge. The purpose of the fog layer is to bring computation near the edge layer. It reduces the distance and latency of data processing. The fog layer also has a cloud-like environment with several fog environments that serve many edge devices. The fog environment comprises several infrastructure components that can be conceptually grouped as fog nodes.

In the image processing application, the fog layer is depicted by the Kubernetes cluster with a master node and worker node. This node can conceptually be considered a fog node. These nodes have the responsibility of maintaining the queue for asynchronous processing. The fog layer processes the image captured by edge devices. This processing makes the system more robust and scalable. It also helps to reduce network congestion in the cloud layer.

4.2.3 Cloud Computing Layer

Cloud architecture refers to deploying computing resources and data storage in a centralized data centre accessible over the Internet. Cloud computing has large computing functions distributed over different geographical locations called data centres [17]. The cloud layer has huge storage and computing resource that can be extended based on demand. Resource-intensive applications like data analytics, data lakes, and data warehouses can be built on a centralised cloud layer.

In image processing applications, the image processing in the fog layer is stored in the cloud layer, as it has a vast storage capacity that can be extended easily based on demand. The cloud layer is a central data centre used by several applications. It has an advantage when the data are processed and prepared by the lower layer, and clean data is available for analytics in the cloud layer.

In the chapter, We talked about the high-level design of the application, with different functions and non-functional expected from the system. The event-driven architecture also helped us to visualise the flow of the application. It gives the overall description of the system. In the following chapter, we will look at the detailed description of the system for a solution to the forwarded problem.

5 Proposed Approach

To achieve the solution of the problem, there are multiple approaches to reaching the solution. In this chapter, we describe our approach to solving the forwarded solution. Section 5.1 describes the test bed setup of the proposed system architecture in the cloud environment. In section 5.2, We discuss the auto-scaling configuration in the test-bed setup. Section 5.3 describes the data collection mechanism and the important metrics of the system. Section 5.4 discusses the Machine learning implementation and the stages involved in making an efficient machine model.

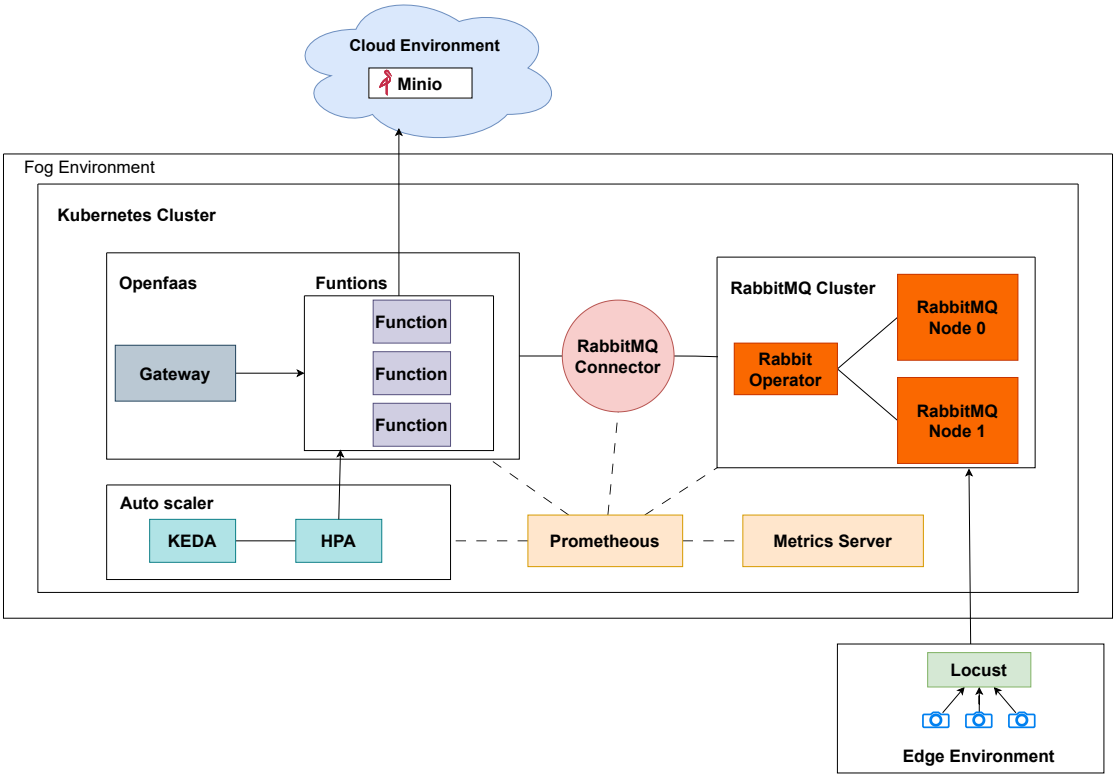


Figure 4. Experiment Setup of Image Processing application

5.1 Test-bed Setup

To realize an experiment for evaluating the performance of the data pipeline under different scaling strategies, the test bed is set up to combine all computing layers (edge, fog and cloud). Figure 4 depicts the setup of different tools and systems across respective layers.

The edge layer's primary job is sending images to the fog layer. The edge layer uses Locust[10](python tool) to simulate real-time load. The edge layer hardware unit consists of six Raspberry Pi(RPi) 4B models with specification of Quad-core Cortex-A72 (ARM v8) 64-bit 4 CPU core 8 GB RAM resembling with m2.medium size of AWS EC2 instance.

In the proposed design, the fog layer does most of the operation. The fog layer is set up with a Kubernetes cluster with two nodes, master and worker. The lightweight Kubernetes(K8s) engine called K3s is installed in this layer. OpenFaaS[1] serverless computing platform has been installed for orchestrating function deployment over the k3s cluster. RabbitMQ is installed as three pod clusters in the fog layer, with one virtual host as an operator, one exchange and one message queue. Prometheus[2] is installed as a pod to monitor and record application metrics in a time-series database. Kubernetes metrics server is installed as a pod to collect resource metrics from Kubelets. It exposes them in Kubernetes Apiserver through Metrics API for use by Horizontal Pod Autoscaler(HPA) and Prometheus. RabbitMQ connector, an open-source tool, is installed as a pod to act as an event bridge between the Message queue and OpenFaaS. The Kubernetes pods are scaled using Kubernetes Horizontal Pod Autoscaler(described in section 5.2) with Kubernetes-based Event-Driven Autoscaling(KEDA) to auto-scale based on RabbitMQ incoming message rate. The machine learning agent developed at the end of the base experiment is deployed as a pod to auto-scale the number of functions in the OpenFaaS. The fog layer hardware consists of 8 core CPU with 16GB RAM provisioned by the University OpenStack Cloud.

Finally, the Cloud layer is configured with Kubernetes K8s engine with MinIO object storage running as a pod.

5.2 Auto scaling mechanism

In the data pipeline systems, there can be the presence of a bottleneck in the chain of tasks which can bring down the performance of the system. This bottleneck has to be identified and removed to achieve optimal performance and low latency. Identification of bottlenecks is out of the scope of this research paper. In the image processing application designed for the experiment, The bottleneck identified is mainly the RabbitMQ connector and transformation function because a single instance of these services will increase the system latency. If the RabbitMQ connector has only one instance, there will be a high number of data in the queue to be processed. If the transformation function takes more time, it increases the overall execution time to send the data from source to sink.

[10] `https://locust.io/`

[1] OpenFaaS® makes it easy for developers to deploy event-driven functions and microservices to Kubernetes without repetitive, boilerplate coding.

[2] Prometheus application used for event monitoring and alerting

In this experiment, we proposed different approaches of auto-scaling to scale different components of the image processing data pipeline as depicted in Table 1

Mechanism	Approach	Scaling Metrics	RabbitMQ	Serverless functions
Reactive	Workload based	Message Rate	KEDA	KEDA
Reactive	Resource-based	CPU Usage	K8s HPA	K8s HPA

Table 1. Auto scaling mechanism

5.2.1 Kubernetes Horizontal Pod Autoscalar (HPA)

The Kubernetes Horizontal Pod Autoscalar is a component that periodically updates the workload resources in the Kubernetes cluster to match the required resource demand or configuration. The HPA exposes simple-to-use API to configure the workload resource configuration. HPA can make scaling decisions based on custom or externally provided metrics and works automatically after initial configuration. System Administrator or machine learning agent can configure this HPA configuration. HPA has a control loop where, at each time period, the HPA gets information regarding the metrics considered in the scaling decisions[ZFFP22]. The HPA make a scaling decision to scale the number of pods based on the equation 2

$$n_{t+1}(s) = \lceil n_t(s).\frac{m_t(s)}{m^*(s)} \rceil \tag{2}$$

The desired number of pods increases or decreases one step at a time and decreases one pod when the cool-down period is over. The HPA of the initial experiment ranges from 10% to 90% for all possible combinations of RabbitMQ pod count, Imageprocessor function pod count, and several functions to push processed image count to MinIO.

5.2.2 Kubernetes-based Event-Driven Autoscaling (KEDA)

KEDA[12] is a Kubernetes component that drives the scaling of any container in Kubernetes based on the number of events needing to be processed. KEDA work works with Kubernetes Horizontal Pod Autoscaler(HPA) by continuously polling the event source and adjusting the scale of deployed pods. This helps us to scale the processing capabilities on real-time events. In the proposed system, The KEDA service is deployed as a pod in the cluster, periodically monitoring the Message Queue system. The configuration for KEDA scaling metrics is message rate. It monitors the message published to the RabbitMQ service and simultaneously scales the number of pods. The configuration for the system is set from 0.5, 0.7, 0.9, and 1.1 messages per second.

[12] `https://keda.sh/`

5.3 Data Collection Mechanism

Data collection is the process of gathering, measuring, and analyzing accurate data from various relevant sources to find answers to research problems, answer questions, evaluate outcomes, and forecast trends and probabilities[3023]. There are several monitoring solutions to gather data in distributed system architecture. Some of the examples are Zabbix[3], Sentry[4], Nagios[5], Prometheus, and Ganglia[6]. Data collection is one of the core actions of the study. It helps to prepare machine learning models and analyse system performance. The gathered data can be used for learning by machine learning models. For our study, we have continued with Prometheus. The Prometheus monitoring service is widely used in a real-world production environment and adopted by most of the research. The popularity of Prometheus is because it is open source and can be easily extended. Prometheus is designed to monitor dynamic cluster systems, which fits our use case.

Prometheus logs a wide range of metrics. But not all recorded parameters can be used in every study. The selection of the metrics should be made based on their suitability, according to the study. It requires a review of the available metrics and their impact. Based on the detailed study of the available metrics and in-depth analysis of their effect on the system, careful selection of the metrics is shown in section 5.3.1.

5.3.1 Application Metrics of Serverless Data Pipeline

In this section, we have described the importance of each scrapped data point using Prometheus.

Workload Total Processing Time: The average workload total time processing time represents the average time taken to process a unit of work to flow through the data pipeline. In our application, the workload is the image captured by the camera. Average workload is more reliable data than individual processing time of individual units of work because it gives the overall insight of batch processing of images in one cycle. An individual unit of work can vary based on queue time and delay in processing by any function in the data pipeline.

Average Workload Processing Time: The average workload processing time represents the total time to process the workload under a specific configuration. It is calculated from the average workload processing time for a particular combination of scaling configurations.

Workload Size: It is the amount of work the locust system transfers during load testing. In our application, the amount of work is measured in terms of the number of

[3] https://www.zabbix.com/

[4] https://sentry.io/welcome/

[5] https://www.nagios.org/

[6] https://developer.nvidia.com/ganglia-monitoring-system

images transferred by the agent or user in the locust system. Word Size helps to determine the performance of the data pipeline. Image with bigger size takes more time to process, and image with smaller size takes less time. It also holds validity for transferring data from source to sink.

Workload Type: Workload type is a categorical string data that distinguish the type of scaling set for the pod function in the Kubernetes cluster. There are two scaling strategies: CPU utilization-based scaling by HPA and massage rate-based scaling by KEDA.

Average Function Execution Time: The Average Function execution time is the average time a series of functions runs to process a work unit. It is the average sum of the *Imageprocessor* function and to cloud function executed time. It gives insight into the time a particular unit of work (task) takes in the transformation phase in the data pipeline.

Average Queue Time: The average queue time represents the average time a task spends on the queue before it is taken out of the queue by the RabbitMQ connector for further processing. Average Queue Time is an inverse relationship between average workload processing time. The longer the average queue time, the larger the average processing time value. The average queue time should be minimum to increase the system's efficiency.

Average Imageprocessor function execution time: The Average image-processor function execution time is the average time an image-processor function runs to process an image. It gives insight into the time a particular unit of work (task) takes in the initial step of the transformation phase in the data pipeline. It helps to identify the bottleneck of the application.

Average Tocloud function execution time: The Average image-processor function execution time is the average time an image-processor function runs to process an image. It gives insight into the time a particular unit of work (task) takes in the initial step of the transformation phase in the data pipeline. It helps to identify the bottleneck of the application.

Number of Imageprocessor function count: The number of image-processor function counts is self-explanatory. It is the sum of all the functions run in parallel to process the image and forward it to the following function in the stage. The number of **Imageprocessor**s depends on CPU usage and request.

Number of to-cloud function count: It is the sum of all the functions run in parallel to process the image and forward it to the following function in the stage. The number of *Imageprocessor*s depends on CPU usage and request.

CPU requested for Imageprocessor pod: It is the number of CPU core requests by the OpenFaaS gateway from the resource schedule to run the function in a pod.

CPU used for Imageprocessor pod: It is the number of virtual CPU cores the OpenFaaS gateway uses to process the request in the data pipeline.

CPU requested for to-cloud pod: It is the number of virtual CPU core requests by the OpenFaaS gateway from the resource schedule to run the function in a pod.

CPU used for Imageprocessor pod: It is the number of CPU cores the OpenFaaS gateway uses to process the request in the data pipeline.

Memory requested for Imageprocessor: It is the number of memory requests by the OpenFaaS gateway from the resource schedule to run the function in a pod.

Memory used for Imageprocessor: It is the amount of memory the OpenFaaS gateway uses to process the request in the data pipeline.

Memory requested for Tocloud: It is the number of memory requests by the OpenFaaS gateway from the resource schedule to run the function in a pod

Memory Used for Imageprocessor: It is the number of memory requests used by the OpenFaaS gateway to process the request in the data pipeline.

Message Rate: It is the rate at with messages arrive in the message queue. Message rate depends on the configuration in locust based on concurrent users and time duration between request and user spawn rate.

5.3.2 Data Scraping Technique

After we finalized the vital data point to our study, we used the Prometheus tool to scrape data from the server. Prometheus provides a different way to scrape or push the metrics of a service. Service in the Kubernetes cluster can push metrics to Prometheus, and Prometheus store them in a time series database. On the contrary, Prometheus can be set to scrape the service metrics on every defined interval when the URI[7] endpoint for metrics is exposed in the service(for example, '/metrics'), which is namely called HTTP service discovery. HTTP service discovery enables us to discover targets over an HTTP endpoint[8]. Thus, it decreases complexity by removing each service's registry. Additionally, Prometheus exposes HTTP-based API to query from a time-series database (PromQL). Figure 6 shows the sample data of PromQL. As some service has no metrics endpoint, we introduce Kube-State-Metrics to monitor the Kubernetes cluster. It provides information about the state of a couple of Kubernetes objects by listening to Kubernetes API [9]. All the cluster metrics can be obtained from the Kube-State-Metrics service on /metrics URI endpoint. Thus, Prometheus is well-compatible with Kube-State-Metrics.

```
{'status': 'success', 'data': {'resultType': 'vector', 'result': [{'metric': {'job': 'kubernetes-pods'}, 'value': [1679125861,
'0.5089299663299663']}]}}
0.5089299663299663
```

Figure 5. Sample Response of PromQL

[7]URI - Uniform Resource Identifier

[8]`https://prometheus.io/docs/prometheus/latest/http_sd/`

[9]https://chrisedrego.medium.com/kubernetes-monitoring-kube-state-metrics-df6546aea324

The data from the application and Kube-State-Metrics services is scraped by Prometheus and stored in the time-series database as all the metrics that are scrapped is a raw metrics which are not metrics which is scrapped every 15-second time interval. We use HTTP-based PromQL to query the specific metrics. These metrics are aggregated in a CSV file using Python script. Figure 6 depicts this operation's flow block level diagram.

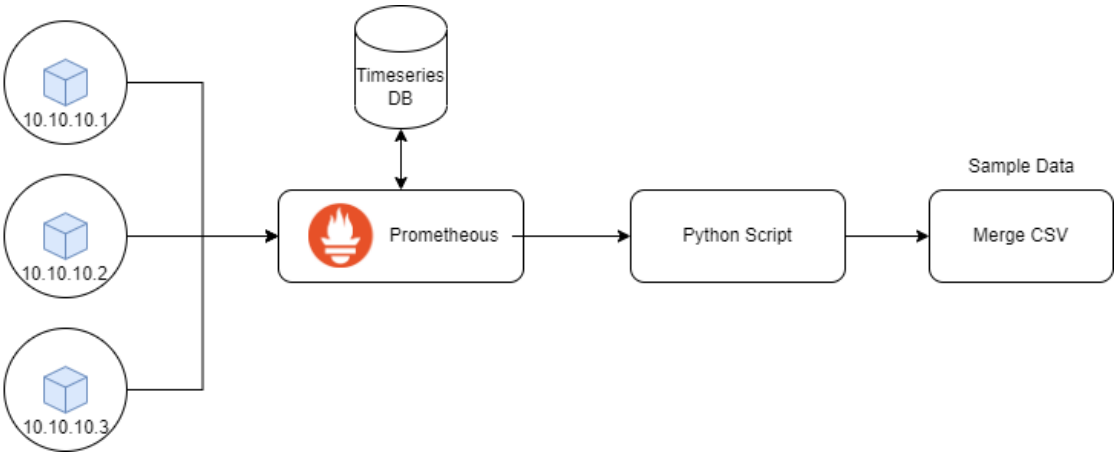


Figure 6. Data collection flow by Python script

The experiment has to be performed for different configurations of RabbitMQ connector, Imageprocessor function and to-cloud function. Initially, we set all the initial auto-scaling configurations using HPA, KEDA or Machine Learning Agent, depending on the experiment type. Then the Locust sends the load to the RabbitMQ queue service in the Kubernetes cluster; the load is sent for 100 seconds with a spawn rate of 0.05. A parallel user will send the load to the RabbitMQ queue service. Their load is processed in the data pipeline. While the load is processed in the data pipeline, the auto-scale continuously monitors and scales the target service. Prometheus and Kube-State-Metrics also monitor all the services and objects in the cluster and record the data in the database. At the end of a particular configuration, the Python script scrapes the data stored in the Prometheus time-series database and exports these data in a CSV file.

Scaling Mechanism	**RabbitMQ Connector**	**Imageprocessor**	**Tocloud**
HPA (CPU usage)	10,20,30,..,90	10,20,30,..,90	10,20,30,..,90
KEDA (message rate)	-	0.5, 0.7,0.9,1.1,1.3	0.5, 0.7,0.9,1.1,1.3

Table 2. Scaling Configuration Combinations based on scaling mechanism

This experiment is repeated for combinations mentioned in Table 2. After running all the configuration combinations, all these individual CSV files are merged into a single CSV file to get sample data as depicted in Figure 5.3.2. Finally, The merged CSV file has all the required headers valid for training machine learning models.

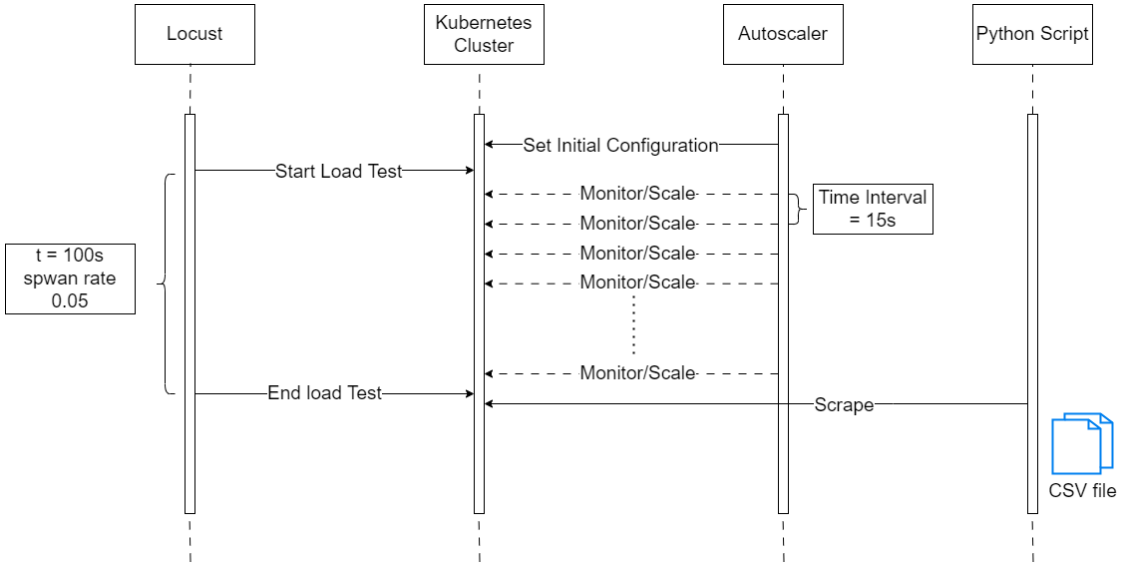


Figure 7. Experiment Iteration to collect server metrics

5.4 Machine Learning Implementation

In this section, We discussed the methodology and our approach to building an efficient machine-learning model. For the same purpose, we have chosen Python as a programming language. Many languages are available in the industry to analyse data and train machine learning models. One of the famous examples is R. We continued with Python because of many benefits like a ready-made library, active community support and an easy learning curve. The primary reason is Python's support of popular machine learning and deep learning frameworks and libraries like Tensorflow, Keras, and Scikit-Learn.

Machine learning-related tasks generally have standard steps that help validate continuous development. For our study purpose, we have followed the usual flow that involved data exploration, data cleaning, data transformation, feature selection, Model Development, Model Evaluation, and Model Deployment[Kap22]. Each step has its significance, which will be discussed in the subsection. We follow these steps to get higher accuracy by feeding higher-quality data to the machine learning algorithm.

5.4.1 Problem formulation, Data acquisition, Data exploration

This is the first and crucial step of the machine learning implementation. Here we define the problem and the objective we want to achieve using machine learning. This step also ensures we have a good quality data source to procure good data from the model. After procuring the data, the data needs to be examined to understand the context and identify the pattern in the dataset. The data has been acquired from the Kubernetes cluster using Prometheus. In section 5.3.1, we have discussed detail the data collection mechanism

so that it can be used to train machine learning models. These data are structured with a header described in section 5.3.1. It helps to understand the definition of the feature and helps us while working on the feature engineering. During the exploration phase, we found that the feature with time metrics has decimal points up to seven decimal places. The sample of a feature(Average Workload Total Processing Time) is shown in table 4. Similarly, there were some features with missing values or negative values. This initial exploration helps to clean data in the next phase, which is Data processing and Feature selection.

Average Workload Total Processing Time(second)
69.86374603
67.51003571
68.2645
67.68982258
64.96255357

Table 3. Sample Feature

We are building the machine learning model to understand the optimal number of OpenFaaS functions required in our image processing data pipeline. Since multiple pods are in the data pipeline chain, we have to scale them independently because scaling the *Imageprocessor* function and *Tocloud* function with the same number of pods will not be the optimal use of resources. The image processing function requires more resources to process an image. Hence there will be congestion in the data pipeline, and increasing the number of *Imageprocessor* functions will help to process images in the queue in parallel, and *Tocloud* job is to push the image to MinIO object storage. But, there can be another instance when the network between *Tocloud* and MinIO object storage have lower bandwidth. The could be congestion in pushing the image into the MinIO storage. We need more *Tocloud* functions in this scenario to push the processed images in parallel. Thus, We need a machine learning model with more than one independent variable called Multivariate Output. The equation below represents the mathematical model of the Multivariate Regressor model.

$$Y = \beta_0 X_0 + \beta_1 X_1 + ... + \beta_n X_n \tag{3}$$

where, $Y \epsilon Y_0, Y_1, Y_2, ..., Y_n$ is an dependent variable set. In our use case, $Y \epsilon Y_0, Y_1$ can be set of the number of *Imageprocessor* functions and the number of to-cloud functions, respectively. Similarly, $X_0, X_1, X_2, ..., X_n$ is an independent variable, for example, CPU Utilization and Memory utilization and $\beta_0, \beta_1, \beta_2, ..., \beta_n$ represent contribution of independent variable toward Y (dependent variable).

To explore the sample data, We use Python libraries such as Pandas and Matplotlib. This library helps to find the data distribution along with minimum and maximum values.

Similarly, we use Matplotlib to visualize the feature for uniform distribution. Table 3 shows an example of the Average Workload Processing Time in Seconds. From Figure 8 Quantile- Quantile plot(Q-Q plot), it is clear there is some outlier or erroneous value, i.e. (668 seconds). Also, plotting the histogram (Figure 9) shows that the uniform distribution graph is skewed toward the left. It means data is not uniformly distributed because of outlier values. In the next phase, we pre-process these data points to make the algorithm more efficient.

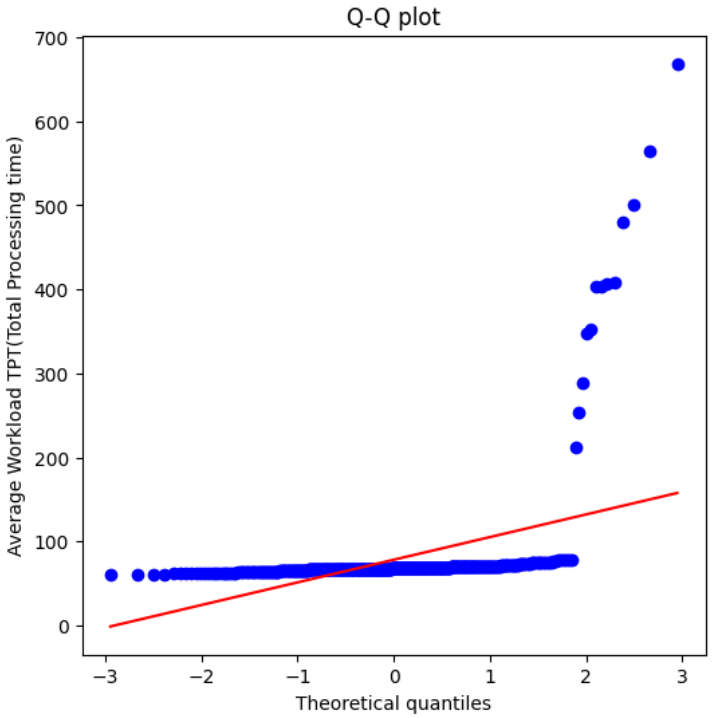


Figure 8. Q-Q plot of Average Workload Total Processing Time

5.4.2 Data pre-processing and Feature selection

The data extracted from the source may not always be reliable and ready to use. The data can have incomplete values and contains noisy and outlier value. This type of data has low quality. The lower quality of data results in lower accuracy and reliability in data while making decisions. Data pre-processing helps to make data more consistent and increases the data's algorithm readability.

To pre-process data, we have removed missing values using SimpleImputer provided Scikit-Learn Imputation Technique. This will fill the average value of the feature in the sample dataset. As we have less number of data points, thus removing an entire row

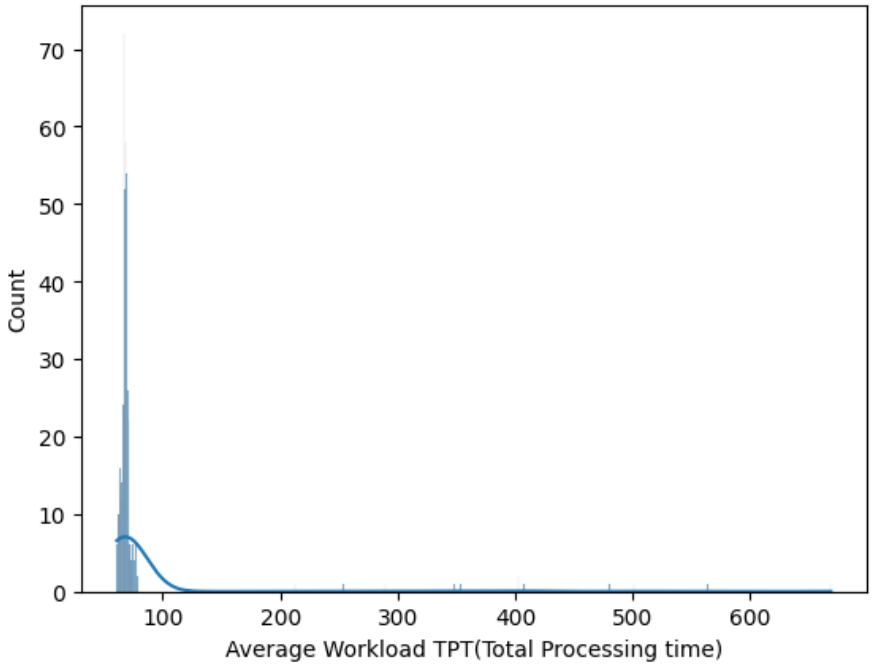


Figure 9. Histogram plot of Average Workload Total Processing Time

based on one missing value is not justified. Thus, SimpleImputer provides mean, median, most frequent, and constant imputation strategies. We use the mean imputation strategy, which is also the default strategy. This gives the advantage of being an unbiased dataset. Additionally, we remove the negative value from the dataset. This error caused due to a calculation error while scraping the data from the server. So we use algorithm 1 to impute the mean value and remove the negative from the dataset.

In addition to that, we have changed categorical data to numerical data using One Hot Encoding. This method of encoding the categorical data in a feature which does not have any hierarchical representation is transformed into multiple features, as shown in Figure 10. The transformed data can be easily fed to the machine learning algorithm, which generally takes numerical value rather than a categorical string value. The categorical value has only two possible values; Pandas Library creates two extra features. The feature with the HPA header has one value corresponding to the original feature's row. Similarly, The feature with KEDA has one value original feature's row. The remaining row is filled with 0 values.

After cleaning all the erroneous, negative and missing values, the data obtained can be fed into the machine learning model. For example, The skewed data shown in Figure 8 and Figure 9 can be visualized in Figure 11 and Figure 12 respectively. The data are now uniformly distributed, and there are fewer outliers.

For Feature selection, we tried to find the correlation between the selected feature

ScalerType
HPA
KEDA
HPA
HPA

Transformed

ScalerType_HPA	ScalerType_KEDA
1	0
0	1
1	0
1	0

Figure 10. One hot encoding of Categorical Data

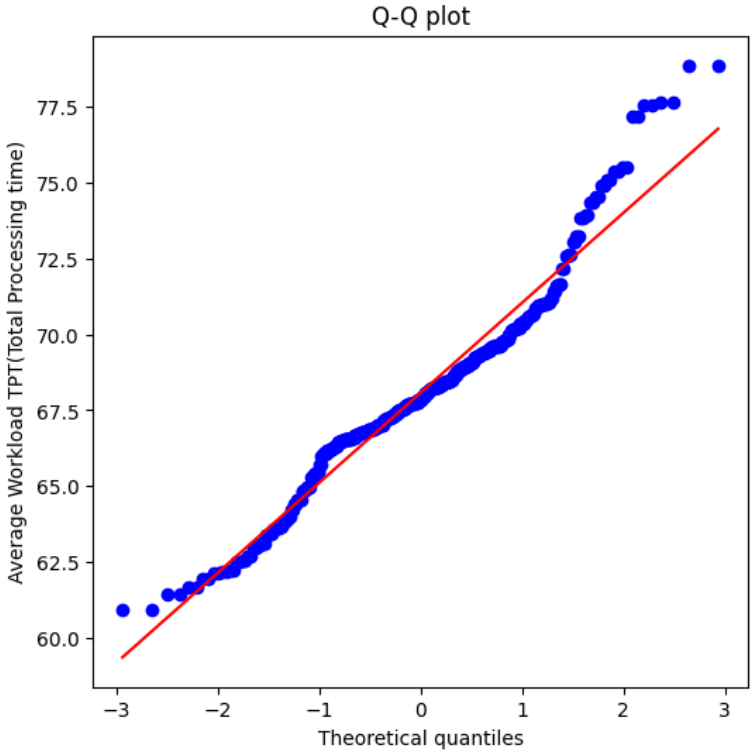


Figure 11. Uniform Distributed Q-Q plot of Average Workload Total Processing Time

Algorithm 1: removeNegativeValue

Input: ProcessingTimes ← Collection of Processing Time containing positive and negative values

Result: Collection of Processing Time containing positive value

```
data ← ProcessingTimes
result ← []
i ← 1
while i ≤ length(data) do
    if data[i] ≥ 0 then
        Append( result, data[i])
    i ← i + 1
return result
```

and the target feature. Figure 13 shows the correlation among selected features. We used Seaborn Library provided by Python. Correlation Value range from -1 to 1. When a feature highly depends on another feature, the correlation value is closer to 1. Similarly, If a feature is unrelated or inversely dependent on another feature, the value is close to -1. For example, the Pod count of the *Imageprocessor* is negatively related to Average Workload Total Processing Time(Average workload TPT). This implicates if the number of pods increases, Processing Time decreases and vice versa.

To build a machine learning model, we filtered features highly correlated to a number of the *Imageprocessor* and *Tocloud* functions.

5.4.3 Model Development

After analysing and pre-processing the data, we must build a suitable model that gives optimal output with minimum errors and higher accuracy. In our study, we want to predict the number of pod counts based on different metrics scraped from the proposed data pipeline. To solve this problem, we use the Regressor model to analyze the relationship between the dependent variable and one or more independent variables. We have to predict more than one number of dependent variables. This can be achieved with Multivariate output, which predicts one or more than one output given input variable.

1. Random Forest Regression: Random forests have emerged as a forefront classification and regression technique, enjoying exceptional accuracy and enabling interpretative insight for wide classes of problems, all with minimal tuning [MS11]. It has been used in different application areas like for Classification of Assessment of Carbon Stocks [JK16], Stock Prediction [LK16], and Remote sensing like Global

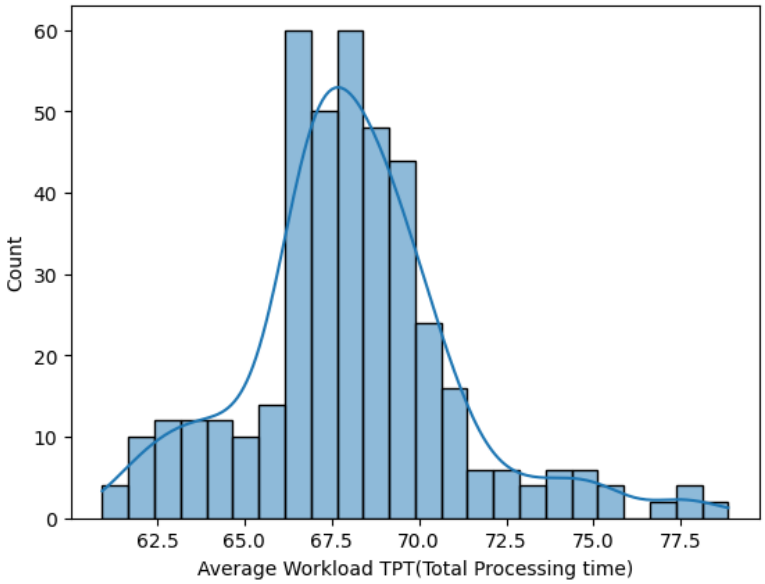


Figure 12. Uniform Distributed Histogram plot of Average Workload Total Processing Time

Burned Area Classification [RC17]. Because of its benefits, we have considered this Machine Learning Algorithm for our study.

Random Forest is a classifier consisting of a collection of tree-structured classifiers $h(x, \theta_k), k = 1, ...$ where the θ_k are independent identically distributed random vectors. Each tree casts a unit vote for the most popular class at input x [MS11]. In a nod,

- With training predictor features(X) and output feature vectors(Y),
- Node splitting is done to select a feature from a random set of m features and a threshold z to partition the node into two child nodes,
- Left node (with samples < z) and Right node (with samples > z).

The value of m for the regression model is $\frac{X}{3}$. The best branch split is by using the mean squared error (MSE) to how your data branches from each node.

$$MSE = \frac{1}{N} \sum_{i=N}^{N} (f_i + y_i)^2 \tag{4}$$

This formula calculates the distance of each node from the predicted actual value, helping to decide which branch is the better decision for your forest. Here, N

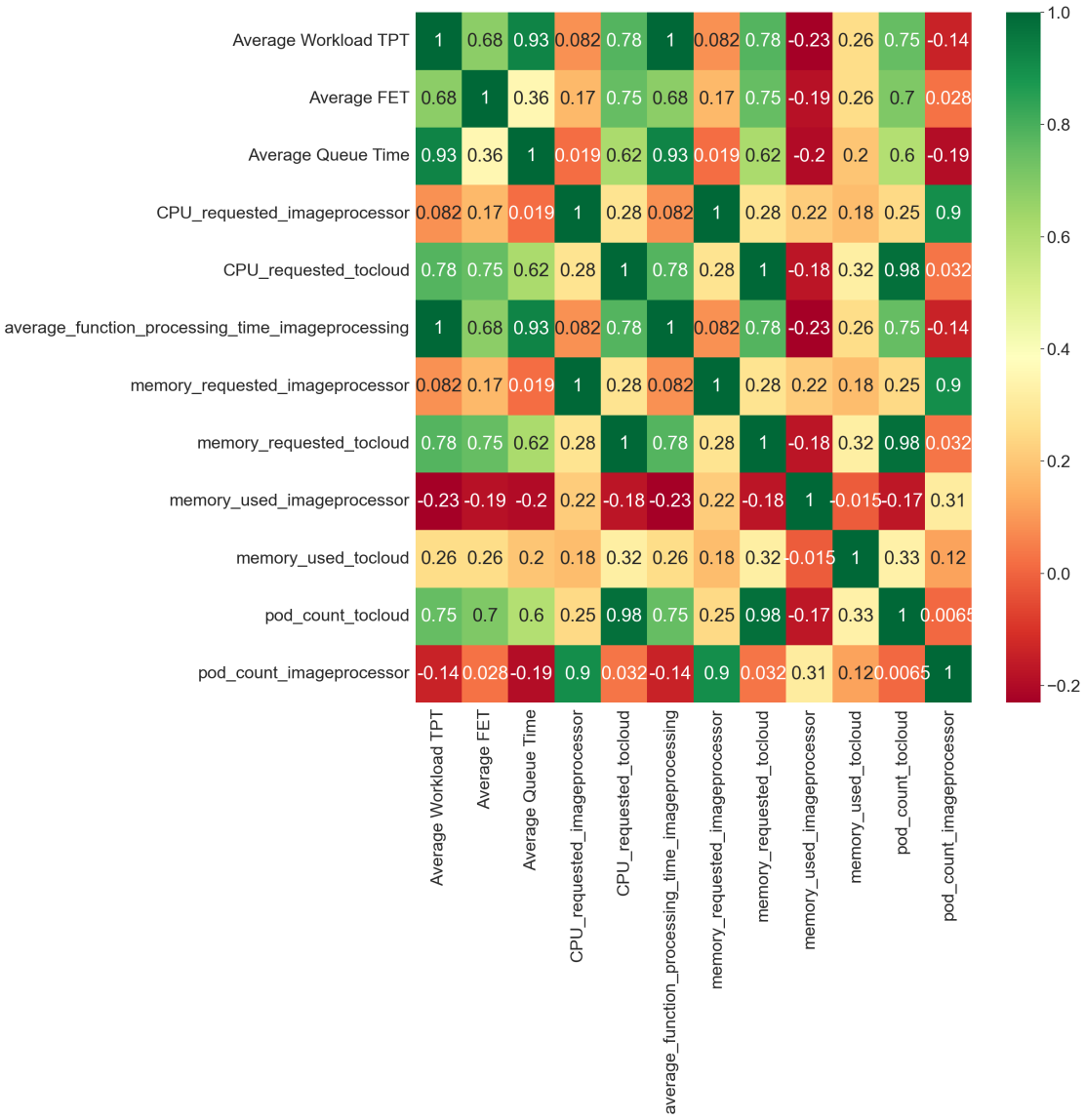


Figure 13. Correlation among features

is the number of data points, y_i is the value of the data point you are testing at a certain node, and f_i is the value returned by the decision tree [Sch19]. Table 4 shows the sample dataset with three features and four data points.

Workload	CPU Used	Pod Count
55	0.36	1
59	0.40	2
58	0.56	2
62	0.64	2

Table 4. Sample Data Set A

a. For the first tree, we take two instances from our data set

For the first tree, we have taken the 2 data points to build the tree. The best feature for the initial root is the one with minimum mean square error.

Workload	CPU Used	Pod Count
55	0.36	1
59	0.40	2

Table 5. Sample Data For Tree 1

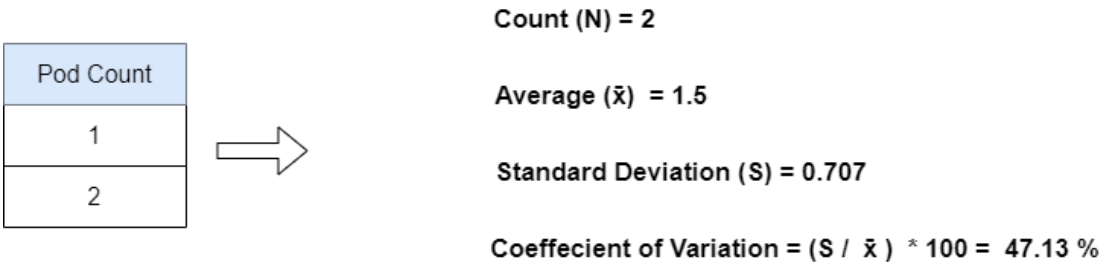


Figure 14. Base calculation for Tree 1

The splitting value for Workload is calculated as 57, whereas the splitting value of CPU used is calculated as 0.38. These values are outcomes with minimum Mean Square Error(MSE). Figure 15

b.The standard deviation for two attributes (target and predictor):

The splitting value for Workload is calculated as 57, whereas the splitting value of CPU used is calculated as 0.38. These values are outcomes with minimum Mean Square Error(MSE). Figure 16

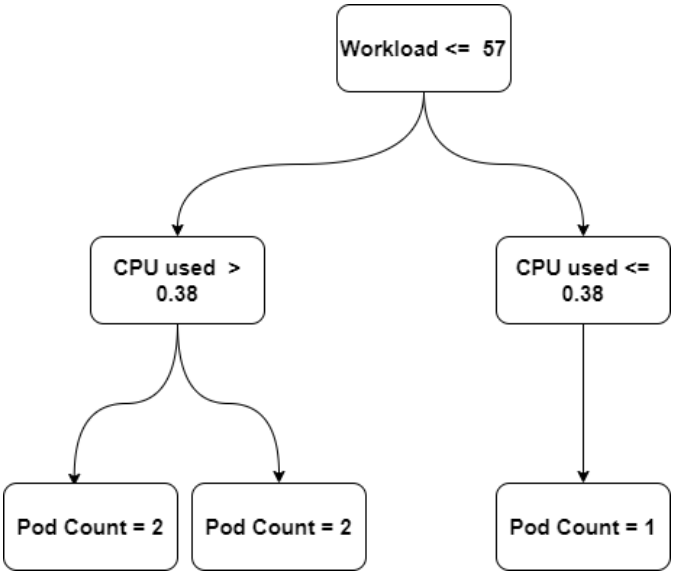


Figure 15. first random forest decision tree

Workload	CPU Used	Pod Count
58	0.56	2
62	0.64	2

Table 6. Sample Data For Tree 2

For the final prediction of the target value, we take the average of all the leaf values in a decision tree. In the table 7

square error = 0.222 sample = 2 value = 1.667

Table 7. Final Target Value Calculation

Similarly, We get the target value from another decision tree. The final predicted value will be the average value of all the decision trees.

2. Lazy Predict Package: Whenever we start building a machine learning regressor model, the initial solution would be to start with a simple model(Random Forest) to predict the initial performance of the model, and then we iterate over a different machine learning model that is provided by the Python Sci-kit package. This is an iterative and time-consuming process. To simplify this process, we use a ready-made Python package called LazyPredict. Lazy Predict helps to decide the

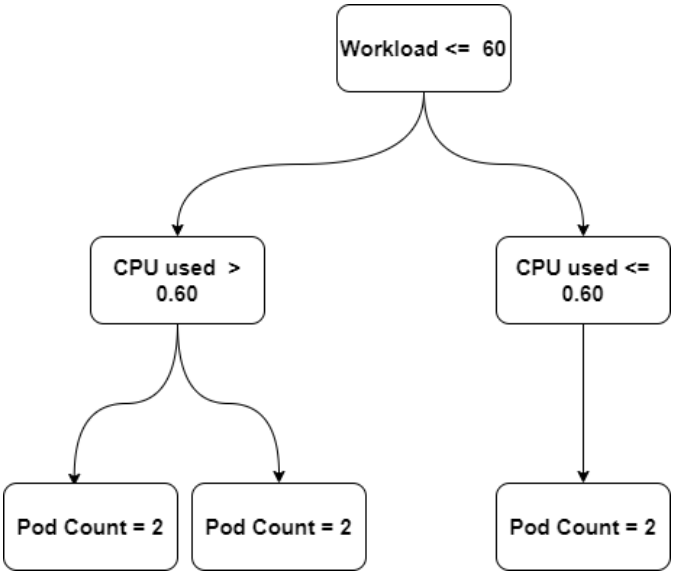


Figure 16. second random forest decision tree

selection of the regression method [Wil23].

LazyPredict is a Python package that aims to automate the machine learning modelling process. It works on both regression and classification tasks, with the key feature of automating the training and evaluation of machine learning models[1822]. It provides a simple interface for defining a range of hyperparameters and then trains and evaluates a model using a variety of different combinations of these hyperparameters [1822]. It is simple to use, and the sample code is shown in 17

```
from lazypredict.Supervised import LazyRegressor

reg = LazyRegressor(verbose=0, ignore_warnings=False)
models, predictions = reg.fit(X_train, X_test, y_train, y_test)
```

Figure 17. Python code to implement LazyPredict for regression

This package gives standard performance metrics for all the machine learning algorithms in Scikit learn package. The sample output is shown in Figure 18. It gives all the necessary metrics that help us to decide on good machine learning algorithms like adjusted R-squared, R-square, and RMSE. It ranks the best algorithm along with the time taken to train the model.

Model	Adjusted R-Squared	R-Squared	RMSE	Time Taken
ExtraTreesRegressor	0.95	0.96	0.09	0.23
OrthogonalMatchingPursuit	0.90	0.92	0.11	0.02
BaggingRegressor	0.90	0.92	0.12	0.09
KNeighborsRegressor	0.86	0.89	0.15	0.33
RandomForestRegressor	0.86	0.88	0.14	0.67
ExtraTreeRegressor	0.84	0.87	0.13	0.02
RidgeCV	0.81	0.84	0.14	0.02
Ridge	0.80	0.84	0.14	0.03
LinearRegression	0.79	0.83	0.15	0.02
TransformedTargetRegressor	0.79	0.83	0.15	0.03
XGBRegressor	0.77	0.81	0.15	0.21
DecisionTreeRegressor	0.58	0.66	0.23	0.02
MLPRegressor	0.52	0.61	0.22	0.80
LassoLars	-0.24	-0.01	0.41	0.01
ElasticNet	-0.24	-0.01	0.41	0.04
DummyRegressor	-0.24	-0.01	0.41	0.01
Lasso	-0.24	-0.01	0.41	0.02
RANSACRegressor	-0.35	-0.09	0.42	0.25
GaussianProcessRegressor	-3.61	-2.74	0.75	0.07
KernelRidge	-22.73	-18.26	1.56	0.08

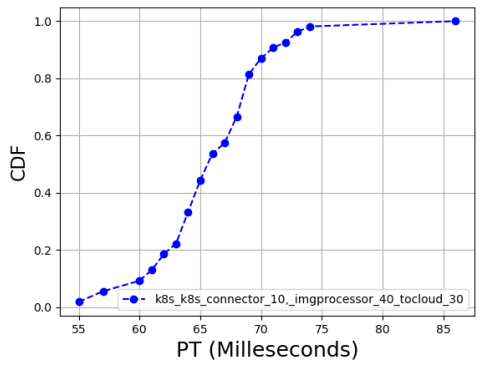
Figure 18. LazyPredict Output on collected server metrics

6 Results and Discussion

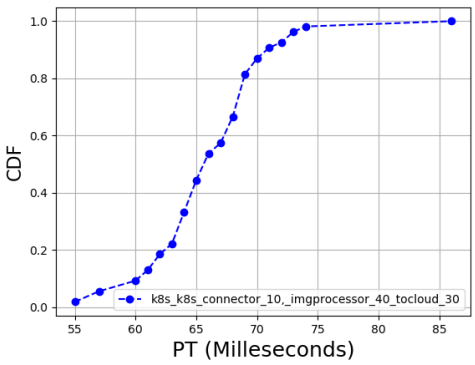
The CDF of the processing time of each workload under a specific CPU configuration using HPA is shown in Figure 19. Figure 19(a) shows the processing shows the CDF of processing time of each workload when the configuration of the HPA to scale Connector, Imageprocessor, and Tocloud pod when CPU usage is 10,10, and 10%, respectively. It has a P95 processing time is less than 78 milliseconds. With Figure 19(b) HPA configuration of the connector is set to scale when the CPU is 10%, Imageprocessor is set to scale when the CPU is 40%, and Tocloud is set to scale when the CPU is 30%. In this case, The P95 processing time is less than 73 milliseconds. For Figure 19(c), where the HPA configuration of connector is set to scale when CPU is 10%, Imageprocessor is set to scale when CPU is 50%, and Tocloud is set to scale when CPU is 40%. In this case, The P95 processing time is less than 75 milliseconds. Figure 19(d) HPA configuration of connector is set to scale when CPU is 10%, Imageprocessor is set to scale when CPU is 90%, and Tocloud is set to scale when CPU is 80%. The P95 processing time is less than 78 milliseconds. Besides these HPA configurations, The CDF of processing time follows a similar pattern as shown in Appendix 7. In our use case, The processing time of the data pipeline increases when the HPA CPU threshold is higher. When the CPU threshold is lower, the HPA threshold is exceeded, and newer functions are introduced in the OpenFaaS to process the payload in parallel from the Message Queue. Hence the processing time of the payload is increased.

The CDF of the processing time of each workload under a specific message arrival rate configuration using KEDA is shown in Figure 20. Figure 20(a) shows the processing shows the CDF of processing time of each workload when the configuration of the KEDA to scale Imageprocessor and Tocloud functions when the message arrival rate in Message Queue is 0.5m/s [12]. The P95 processing time is less than 480 milliseconds. Similarly, Figure 20(b) shows the processing shows the CDF of processing time of each workload when the configuration of the KEDA to scale Imageprocessor is to scale when the arrival message rate is 0.7. Tocloud is when the message arrival message rate in Message Queue is 0.5 messages per second. The P95 processing time is less than 1260 milliseconds. Likewise, Figure 20(c) shows the processing shows the CDF of processing time of each workload when the configuration of the KEDA to scale Imageprocessor is to scale when the arrival message rate is 0.9. Tocloud is when the message arrival message rate in Message Queue is 0.5 messages per second. The P95 processing time is less than 1120 milliseconds. And, Figure 20(d) shows the processing shows the CDF of processing time of each workload when the configuration of the KEDA to scale Imageprocessor is to scale when the arrival message rate is 1.1, and Tocloud is when messaging arrival message rate in Message Queue is 0.9 message per second. The P95 processing time is less than 560 milliseconds. Besides these KEDA configurations, The CDF of processing

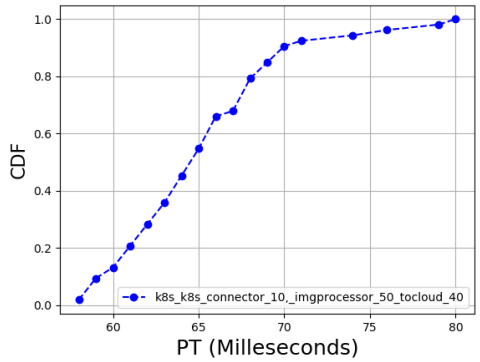
[12] m/s - message per second



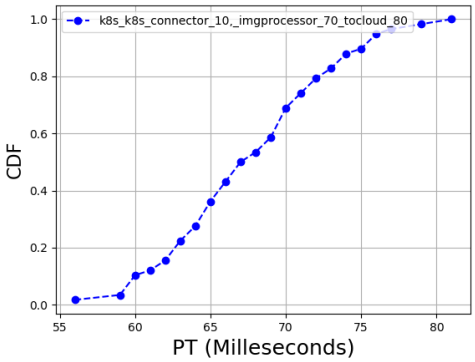
(a) K8s HPA (connector(10%), imgprocessor(40%), tocloud(30%))



(b) K8s HPA (connector(10%) imgprocessor(%40) tocloud(%30))

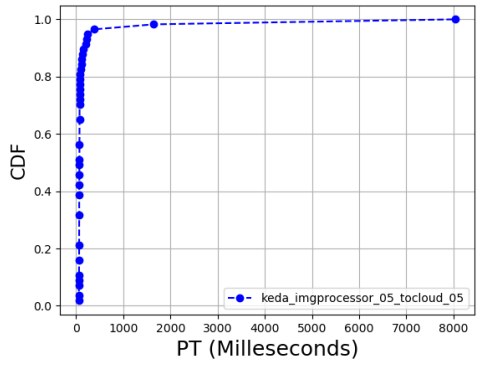


(c) K8s HPA (connector(10% imgprocessor(%50) tocloud(%40)

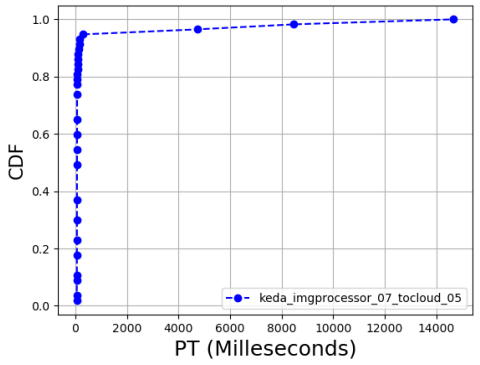


(d) K8s HPA (connector(%10) imgprocessor(%70) tocloud(%80)

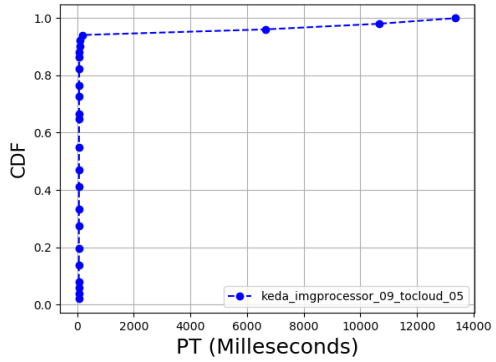
Figure 19. The CDF of processing time of each load in Resource-based scaling(HPA)



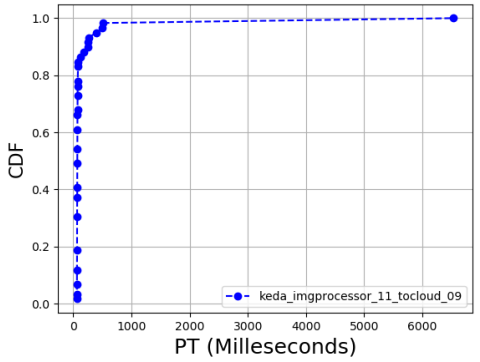
(a) k8s KEDA (imgprocessor(0.5m/s) to-cloud(0.5m/s))



(b) k8s KEDA (imgprocessor(0.7m/s) to-cloud(0.5m/s)



(c) k8s KEDA (imgprocessor(0.9 m/s) to-cloud(0.5m/s)



(d) k8s KEDA (imgprocessor(1.1m/s) to-cloud(0.9m/s)

Figure 20. The CDF of processing time of each load in Workload based scaling(KEDA)

time follows a similar pattern as shown in Appendix 7. The KEDA-based scaling has a higher processing time in general. The processing time after a specific configuration is saturated at a higher message arrival rate. Due to this, the KEDA auto-scaler is not triggered because the message arrival rate from edge devices remains stable for our image processing application. The processing time increases because the message has to wait in Message Queue.

Figure 21 shows the average of all the queuing time across different configurations of resource-based (HPA) and Workload based(KEDA). The queuing time for payload in resource-based is distinctly less than that of workload-based scaling. In Figure 21, The average queuing time for resource-based(HPA) is 31.08 milliseconds with a minimum queuing time of 1.75 milliseconds and maximum queuing time of 49.67 milliseconds,

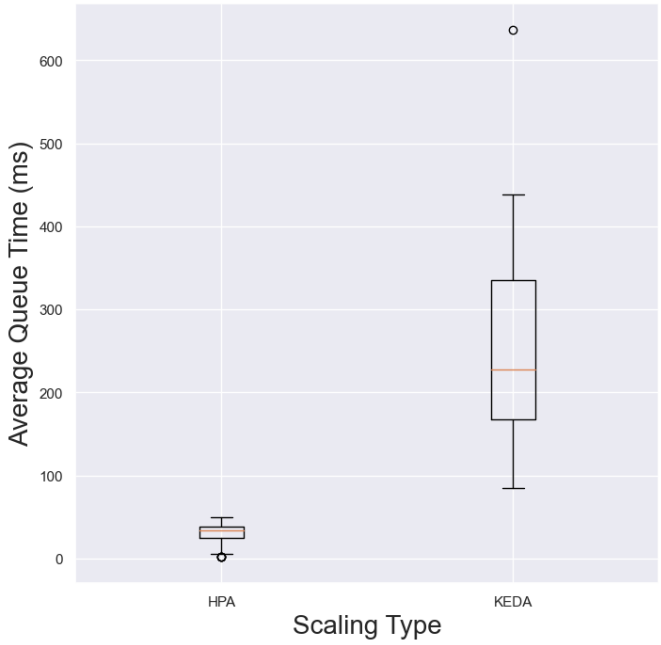


Figure 21. Average Queuing Time in data pipeline across all configuration

whereas the average queuing time for workload-based(KEDA) is 273.26 millisecond along with minimum average queuing time is 84.52 milliseconds and a maximum of 636.38 milliseconds. The payload stays in the Message Queue if there are insufficient computation resources in the fog or cloud environment.

The queue time has a direct effect on the processing time. As we can see in Figure 22. The average processing time of the data pipeline for workload-based(KEDA) is much higher than the average processing time of resource-based(HPA). The average processing time of resource-based scaling(HPA) is 68.03 milliseconds, the minimum average processing time is 60.92 milliseconds, and the maximum average processing time is 78.85 milliseconds, whereas workload-based scaling(KEDA) has a mean of 419.80 milliseconds, minimum of 212.29 milliseconds and maximum 668.58 milliseconds of average processing time.

The resource consumption for the serverless function is nominal, as depicted in Figure 23, as the serverless function is lightweight. Figure 23(a) compares the CPU utilization of resource-based scaling with workload-based scaling. The CPU utilization of the serverless function is approximately the same due to the same payload to process. The mean of CPU utilization in resource-based scaling(HPA) is 52 millicores, whereas in workload-based scaling(KEDA) is 50 millicores. Similarly, In figure 23(b), the memory used to process the payload of 13 Kb is 2.6 MB in resource-based scaling(HPA) and 1.79 MB in workload-based scaling.

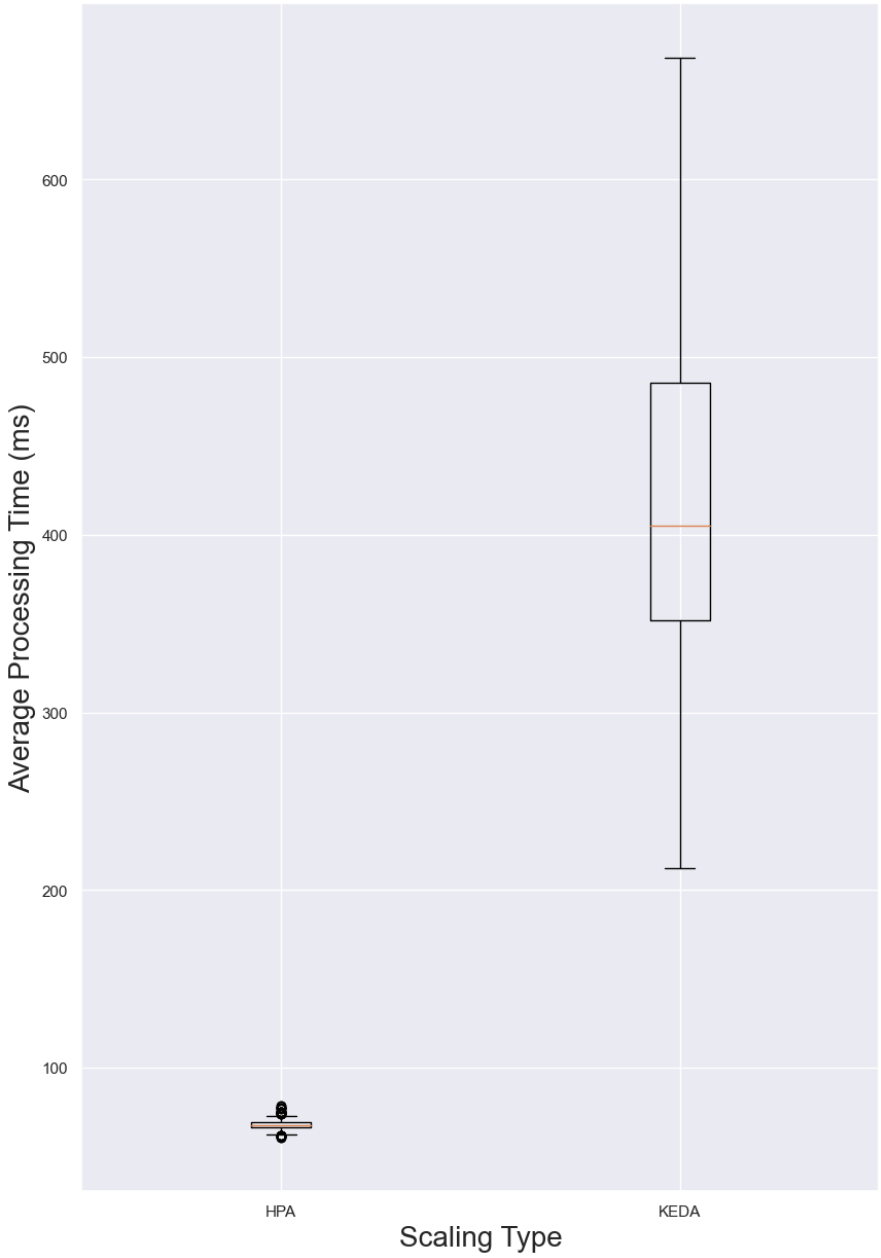
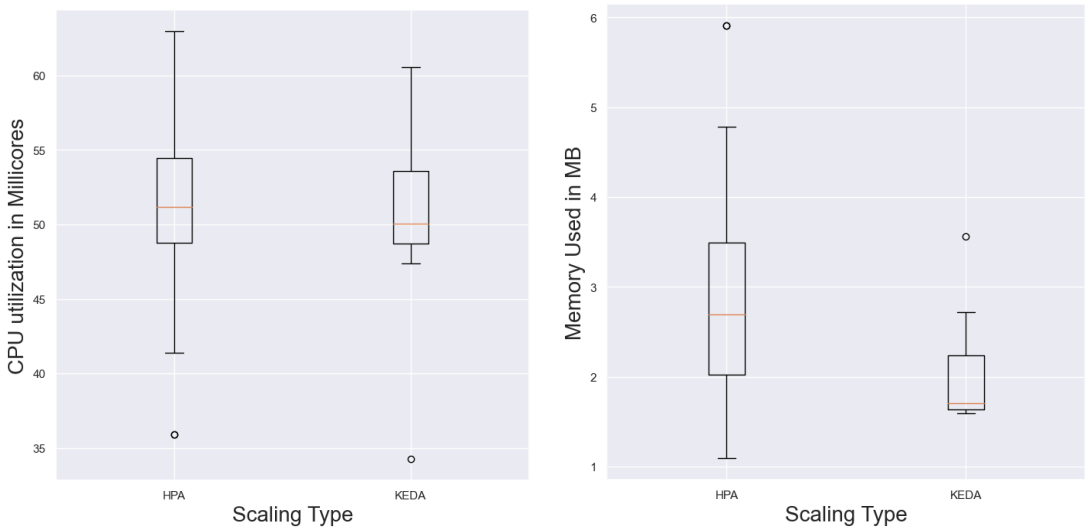


Figure 22. Average Processing Time in data pipeline across all configuration



(a) Average CPU utilization

(b) Average Memory utilization

Figure 23. Average Utilization

In our study, we also predicted the number of serverless functions required in a serverless data pipeline. We selected the Supervised Regressor Machine Learning model for this work. The most commonly used evaluation metrics for regressor models are R^2, adjusted R^2 and Root Mean Square (RMSE). The adjusted R^2 compares the descriptive power of regression models that include diverse numbers of predictors, whereas R^2 is a measure that provides information about the goodness of fit of a model. In an overfitting condition, an incorrectly high value of R^2 is obtained, which is not in the case of adjusted R^2. Sometimes, R^2 could be misleading. Figure 18 shows the most effective machine learning model in order. The highest R^2 is 0.96 for ExtraTreesRegressor followed by OrthogonalMatchingPursuit at 0.95, whereas RandomForestRegressor has 0.86. Similarly, Root Mean Square(RMSE) for the top five machine learning models is under 0.15, which indicates a better prediction.

7 Conclusion and Future Work

This study involved building a data pipeline using serverless functions for data processing. This approach becomes easy to handle the haphazard load coming from the edge devices. It minimizes resource wastage as the serverless functions are spin-off when the event is triggered. This also makes it easy to reuse and manage serverless functions in the data pipeline.

This thesis presents building a simple Image processing data pipeline to measure the performance metrics under different reactive auto-scaling mechanisms i.e. workload-based auto-scaling using KEDA and resource-based auto-scaling using HPA. Based on the data collected in the Image processing data pipeline, it reflects that resource-based auto-scaling gives higher throughput and lower processing time to process the payload whereas resource-based auto-scaling has a higher processing time and queuing time for the same payload.

However, We observed the limitation of the amount of data collected while running a serverless data pipeline, which is not enough for most of the machine learning models to predict and give higher accuracy. For Future work, we can overcome this limitation by using a reinforcement learning agent like Q-Learning which enact directly on the deployment environment to scale the serverless function. It will learn and improve over time using reward and penalty functions.

References

[1522] What is edge architecture? Redhat,Inc., November 2022.

[17] Cloud computing.

[1822] Lazypredict: Run all sklearn algorithms with a line of code, December 2022.

[2] Wikipedia function as a service (faas). `https://en.wikipedia.org/wiki/Function_as_a_service`.

[3023] What is data collection: Methods, types, tools, and techniques. Simplilearn,Inc., April 2023.

[520] Ict: Function-as-a-service (faas) market report id: Rnd_002444. Report and Data, 2020.

[BFRS22] Priscilla Benedetti, M. Femminella, G. Reali, and Kris Steenhaut. Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 674–679, 2022.

[Den21] James Densmore. Data pipelines pocket reference. O'Reilly Media, Inc., February 2021.

[Ete21] Ghobaei-Arani M. Shahidinejad A Etemadi, M. A cost-efficient auto-scaling mechanism for iot applications in fog computing environment: a deep learning-based approach. In *Springer*, May 2021.

[JK16] Sabine Grunwald Jongsung Kim. Assessment of carbon stocks in the topsoil using random forest and remote sensing images. J Environ Qual, November 2016.

[Kap22] Shashank Kapadia. 6 steps towards a successful machine learning project. Medium, April 2022.

[KJZS17] Aqeel Kazmi, Zeeshan Jan, Achille Zappa, and Martin Serrano. Overcoming the heterogeneity in the internet of things for smart cities. In Ivana Podnar Žarko, Arne Broering, Sergios Soursos, and Martin Serrano, editors, *Interoperability and Open-Source Solutions for the Internet of Things*, pages 20–35, Cham, 2017. Springer International Publishing.

[Kni23] Michelle Knight. Data pipelines: An overview, March 2023.

[LK16] Sudeepa Roy Dey Luckyson Khaidem, Snehanshu Saha. Predicting the direction of stock market prices using random forest. J Environ Qual, 2016.

[LKRL19] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. Understanding open source serverless platforms: Design considerations and performance. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 37–42, New York, NY, USA, 2019. Association for Computing Machinery.

[LWW+10] Marin Litoiu, Murray Woodside, Johnny Wong, Joanna Ng, and Gabriel Iszlai. A business driven cloud optimization architecture. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, page 380–385, New York, NY, USA, 2010. Association for Computing Machinery.

[M09] Armbrust M. Above the Clouds: A Berkeley View of Cloud Computing, Technical Repor. IBM Research, February 2009.

[MS11] Yuanyuan Xiao Mark Segal. Multivariate random forests. John Wiley Sons, Inc, February 2011.

[PDJS22] Shivananda R. Poojara, Chinmaya Kumar Dehury, Pelle Jakovits, and Satish Narayana Srirama. Serverless data pipeline approaches for iot data in fog and cloud computing. *Future Generation Computer Systems*, 130:91–105, 2022.

[Pos] Brien Posey. What is fog computing?

[PS21] Antreas Pogiatzis and Georgios Samakovitis. An event-driven serverless etl pipeline on aws. *Applied Sciences*, 11(1), 2021.

[RBOW20] Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Tian J. Wang. Modelling data pipelines. In *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 13–20, 2020.

[RC17] Rubén Ramo and Emilio Chuvieco. Developing a random forest algorithm for modis global burned area classification. *Remote Sensing*, 9(11), 2017.

[Sar21] I.H Sarker. Data science and analytics: An overview from data-driven smart computing, decision-making and applications perspective. In *Springer*, July 2021.

[Sch19] Madison Schott. Random forest algorithm for machine learning. Medium, April 2019.

[SGvK+22] Martin Straesser, Johannes Grohmann, Jóakim von Kistowski, Simon Eismann, André Bauer, and Samuel Kounev. Why is it not solved yet? challenges for production-ready autoscaling. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 105–115, New York, NY, USA, 2022. Association for Computing Machinery.

[Ver19a] Gil Vernik. Predicting the future with monte carlo simulations over ibm cloud functions. IBM Research, January 2019.

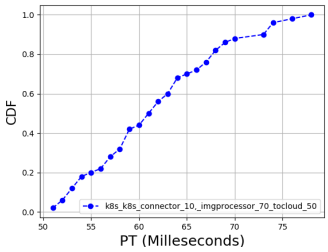
[Ver19b] Gil Vernik. Predicting the future with Monte Carlo simulations over IBM Cloud Functions. IBM Research, 2019.

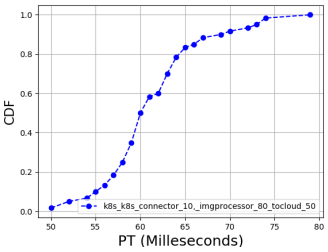
[Wil23] Kalinka K. Sanches R. Williams, T. Machine learning and metabolic modelling assisted implementation of a novel process analytical technology in cell and gene therapy manufacturing. In *Scientific Reports*, January 2023.

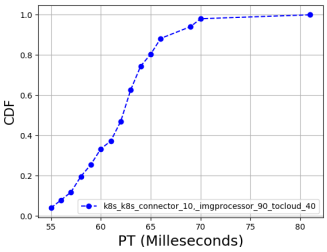
[ZFFP22] Anastasios Zafeiropoulos, Eleni Fotopoulou, Nikos Filinis, and Symeon Papavassiliou. Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms. *Simulation Modelling Practice and Theory*, 116:102461, 2022.

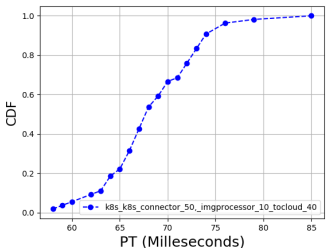
Appendix

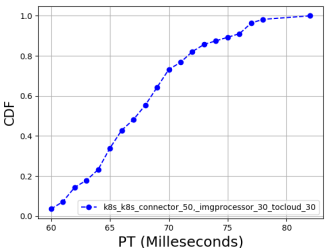
I. Diagram Resource-based scaling (HPA)

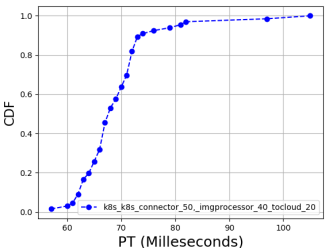


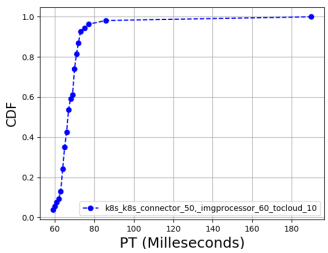


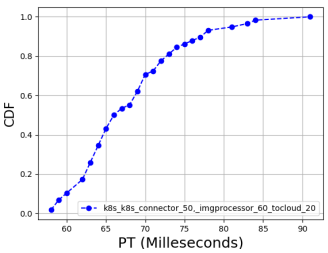


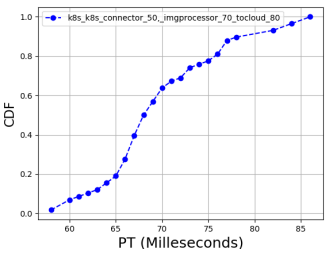


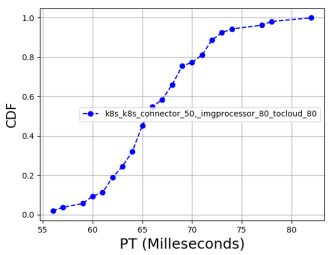


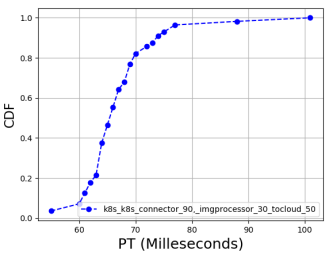


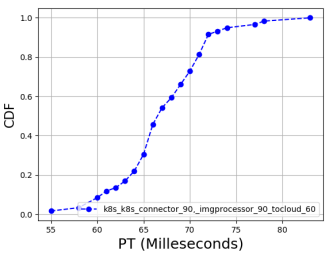




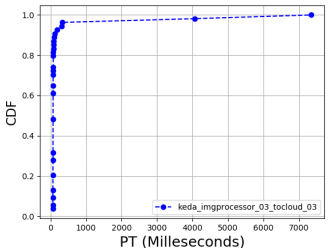


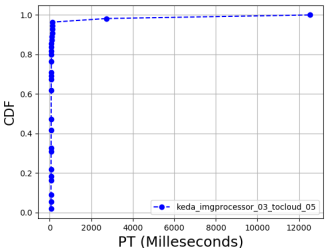


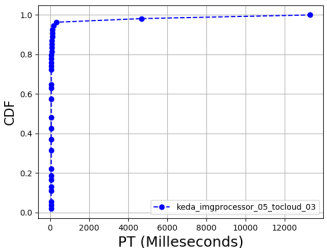


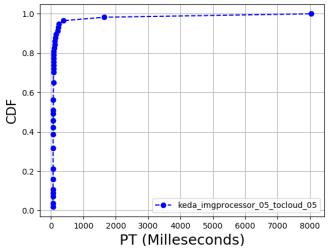


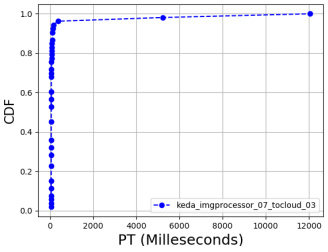
II. Diagram Workload-based scaling (KEDA)

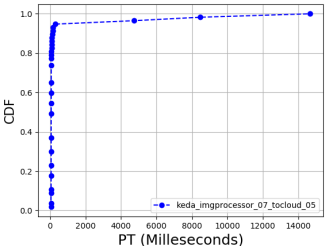


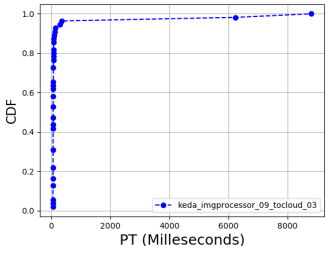


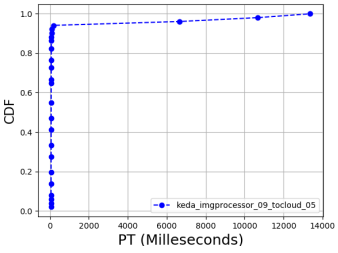


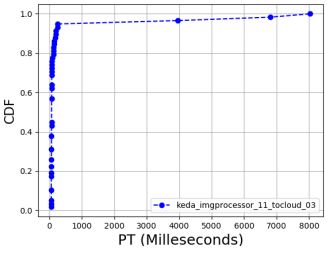


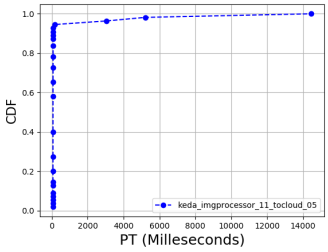


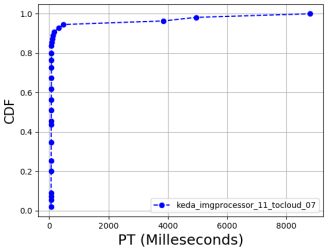


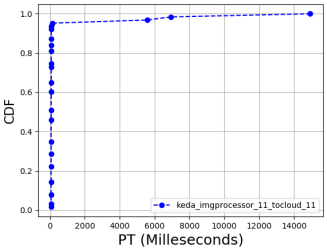












II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Rajan Raj Das**,
*(*author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Towards Auto-Scaling of Serverless Data Pipelines,
   *(*title of thesis)

   supervised by Shivananda R. Poojara.
   *(*supervisor's name)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Rajan Raj Das
08/05/2023