Scaling Approaches for Serverless Data Pipelines in Edge and Fog Computing Environments: A Performance Evaluation

SHIVANANDA POOJARA, Institute of Computer Science, The University of Tartu, Estonia
PELLE JAKOVITS, Institute of Computer Science, The University of Tartu, Estonia
RAJKUMAR BUYYA, The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems,The University of Melbourne, Australia
SATISH NARAYANA SRIRAMA*, School of Computer and Information Sciences, University of Hyderabad, India

The rise of Internet of Things (IoT) applications has led to massive data generation. However, dealing with such massive data is challenging. Nowadays, data pipelines are popular mechanisms used to properly deal with data operations at scale in the IoT continuum. Serverless data pipelines (SDP) are one such approach to performing event-driven data analysis on data streams. Data pipelines are composed of many components, and scaling the entire pipeline without leaving any bottlenecks is challenging. This study aims to assess the performance of scaling mechanisms in handling stochastic workloads efficiently and understanding critical resource utilization in fog environments. We applied workload-based techniques (Request per Second, Queue Length, Message Rate) and resource-based scaling (CPU) on SDP components of two IoT applications: Aeneas (long-running functions) and PuhatuMonitoring (short-running functions). Using Azure serverless workload patterns, we compared scaling approaches in real-time fog environments, evaluating QoS metrics like processing time and CPU utilization. Our analysis of suitability, using the weighted average scoring method on two QoS metrics, revealed that for compute-intensive tasks, the resource-based scaling approach works effectively for jump, steady, spike, and fluctuation workloads. For short execution time tasks, workload-based scaling suits all four workloads.

CCS Concepts: • **General and reference →Evaluation**; • **Computer systems organization →Distributed architectures, Self-organizing autonomic computing**; • **Computing methodologies → Distributed computing methodologies**;

Additional Key Words and Phrases: serverless computing, data pipeline, auto scaling, fog computing, edge computing

1 INTRODUCTION

The widespread use of IoT is enabled by the deployment of thousands of sensors that generate a huge amount of raw data. Artificial intelligence and machine learning techniques are used to gain insights from the data and generate the corresponding actions. This data analysis process involves several activities such as data extraction, transformation, filtering, loading, feature preparation, selection, and training [1]. Currently, data pipelines are widely used, which simplifies the design and deployment of such data processing activities.

*Corresponding author

Authors' addresses: Shivananda Poojara, poojara@ut.ee, Institute of Computer Science, The University of Tartu, Tartu, Tartu, Estonia, 51009; Pelle Jakovits, Institute of Computer Science, The University of Tartu, Tartu, Tartu, Estonia, 51009; Rajkumar Buyya, The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems,The University of Melbourne, Melbourne, Australia; Satish Narayana Srirama, School of Computer and Information Sciences, University of Hyderabad, Hyderabad, Telangana, India, satish.srirama@uohyd.ac.in.

ACM 1556-4703/2025/7-ART
https://doi.org/10.1145/3747186

Data pipelines (DP) are workflows with a series of data processing components [2], where the output of one component is an input to another component, and the data is moved and processed in a pipeline manner. Creating and configuring such data pipelines for IoT data analysis brings several benefits, such as accountability and guaranteed delivery [3–5]. In addition, data pipelines could be deployed in cloud environments that run seamlessly as software services or on premise servers. Some examples of data pipeline services include the AWS data pipeline and Azure Data Factory, and open source tools such as Apache NiFi, StreamSet, and Airflow. However, moving and processing the entire data from IoT devices to the cloud imposes other bottlenecks, such as high dependency on end-to-end tasks, higher response time, and more transfer and storage costs.

The emergence of Serverless computing has simplified the design of event based, real time, and scalable IoT data processing [6]. It is also known as Function as Service (FaaS) and have several advantages in terms of function design, reusability, and granular scaling allowing for greater efficiency. Combining the serverless computing and data pipelines have brought significant benefits to avoid the underlying challenges of the cloud-centric approach and reduce the complexity of designing multi layer (Edge, Fog, Cloud) IoT applications [7–9].

In the Serverless model, functions are individually deployed services that are triggered on certain events (e.g. new database record or REST request arrival), receive data, and produce output. It is also significantly easier to deploy individual functions in different locations closer to the data sources (e.g., Edge, Fog layers) as compared to more monolithic applications (e.g., when compared to Apache Spark data analytic applications). To combine both models (serverless and data pipelines), serverless data pipelines (SDP) [7] can be created where serverless functions are used as pipeline tasks and are seamlessly invoked while the data moves through the pipeline. Moreover, data pipeline technologies are used for data transport, routing, and function invocation. Our previous work [7] investigated the design and performance of such SDPs approaches in three real-time IoT fog applications (video processing, Aeneas, Pocketsphinx). Those approaches include Apache NiFi based SDP, Message Queue based SDP, and Object Storage based SDPs.

Message queues are widely used in IoT systems [10–12], to queue and route data. In this paper, we focus on Message Queue based SDP designed for two IoT applications as described in Section 4. The Message Queue based SDP focused on designing the pipelines using message queues as intermediate data carriers and serverless platforms for processing the data operations. This approach has three pipeline components: Message Queue (MQ), Message Queue Trigger (MQT), also known as Function Invoker/Trigger, and serverless functions. When data arrives in the pipeline into MQ on a specific queue, MQT triggers the corresponding serverless function; the function executes, produces, and publishes the output back to MQ on a specific queue. This sequence of data movement from and to MQ and serverless functions continues in the pipeline till the data sink.

Most of the IoT workloads are stochastic with critical QoS like latency and throughput [13, 44], such constraint specific workloads are needed with auto scaling nature to adapt to the fluctuating demand. Auto scaling has been categorized into two types: workload-based scaling, which focuses on user arrival rate or concurrency of processing, and resource-based scaling, which works on resource utilization metrics like CPU and memory. However, auto scaling in fog environments has to be optimal because over provisioning of resources (serverless function or MQT replicas) can lead to higher utilization, which can consume more energy and hinder the performance of other workloads; on the other side, under provisioning of resources may degrade the expected QoS of running workloads.

Auto-scaling is driven by a predefined set of configurations and threshold-based rules that determine scaling decisions. However, identifying appropriate scaling rules for each function or system component, such as a Message Queue Trigger, can be challenging for developers. For instance, the incoming message rate in a Message Queue might dictate the scaling of the Message Consumer, or the CPU utilization threshold could trigger server scaling. Designing effective scaling rules and identifying the relevant components are critical tasks, particularly in edge and fog environments, where improper scaling can result in resource over- or under-utilization, leading to increased costs or service latency issues. Additionally, scaling decisions are often influenced by the arrival or

event rate of inputs, which directly impacts service quality parameters [14]. This study leverages these input patterns to analyze the efficiency and behavior of scaling strategies in IoT applications within fog environments.

We have selected two SDP applications with distinct characteristics: one with a longer execution time (2145ms) and higher resource utilization, and another with a shorter execution time (986ms) and minimal resource utilization. Aeneas deals with audio data that requires more network bandwidth, whereas PuhatuMonitoring deals with text data that require less network bandwidth. This type of application is useful to understand the performance and behavior of autoscaling mechanisms [15], as discussed in Section 5. Therefore, the management of autoscale in serverless data pipelines can be challenging due to the need for synchronization between the serverless platform and the components of the data pipeline [16]. The proposed work explores scaling techniques using reactive mechanisms (workload- and resource-based) that can be applied on SDP components. In addition, this study highlights the essential threshold metrics or optimal configurations that are required to define scaling. Finally, we measure and compare the efficiency and behavior of workload-based and resource-based scaling approaches for Message Queue-based SDP components in fog environments using real-time applications.

Considering the above context, our work makes the following key contributions:

- We evaluated reactive scaling strategies, such as workload-based and resource utilization-based, on SDP components.
- We used two real-time fog computing workloads, the Aeneas and PuhatuMoniroting applications, to measure the performance, such as processing time, and resource utilization of the scaling methods for various user arrival patterns that mimic the Azure real-time serverless workloads.
- We offer insights on the suitability of scaling approaches, experience and challenges encountered during the implementation and evaluation of the scalability of serverless data pipelines in various configurations.

This study contributes to autonomous and intelligent systems by enabling adaptive, workload-aware scaling of serverless data pipelines in fog computing environments. It addresses the challenge of maintaining performance under dynamic and unpredictable IoT workloads. The findings offer practical insights for optimizing resource utilization and ensuring consistent quality of service for autonomous systems.

The rest of the paper is organized as follows. In section 2, we present a literature survey on the current state of the art autoscaling and technologies. We described the three tier architecture presented in section 3, and section 4 provides an overview of real-time fog applications. Following this, all auto scaling approaches and research questions are elaborated in section 5, and further, experiment setup, and results are outlined in section 6. In section 7 we share our experiences from this investigation and propose recommendations for practitioners. Finally, the concluding remarks along with proposed future work are discussed in Section 8.

2 BACKGROUND AND RELATED WORK

In this section, we discuss the reactive auto-scaling approaches and also provide an overview and comparison of the state-of-the-art works and the approaches used in this paper.

2.1 Auto scaling approaches

Auto scaling is a mechanism for dynamically increasing or decreasing of the resources based on demand to meet QoS expectations. Modern container orchestration tools such as Kubernetes (k8s) are equipped with easy, granular auto-scaling mechanisms[28] compared to virtual machine scalability. This enables serverless platforms to design with container technologies that make for fewer start-up delays by providing several instances at scale with minimum management. Auto scaling is handled with reactive mechanisms in off-the-shelf container orchestration tools and serverless platforms. Reactive scaling approaches are broadly classified into workload-based and resource utilization-metric-based, respectively [20]. The former focuses on the increase or decrease of the count of containers based on user traffic (mainly requests per second). In contrast, the latter focuses on

adding additional containers or deleting based on the resource utilization threshold of the running containers, such as CPU or memory. In the following we provide more insight into the scaling approaches used in this article.

Workload based scaling: The workload based scaling approach is heavily used in public cloud serverless platforms like AWS Lambda. For example, scaling is based on the concurrency limit. Concurrency is the number of in flight requests the AWS Lambda function is handling at the same time. If the function receives more requests then additional replicas are spawned if the concurrency limit is exceeded. The other approach commonly used in open source serverless platforms such as OpenFaaS is Request Per Second (RPS). Here, the function invocation rate decides the scaling up or down of the function replicas. Similarly in message queues, for illustration adding additional consumer instances dynamically when the number of the messages in the queue (RabbitMQ) exceeds the limit or the arrival rate of messages per second exceeds a certain threshold. The scaling of serverless functions involves adding additional replicas of function units, while in message queue scaling, it entails adding extra units of message queue consumers to process the messages.

Resource based scaling: In this scaling approach, the system tries to keep the metrics, such as CPU and memory, in a specified threshold limit. The scaling action is undertaken with the addition or deletion of resources if the threshold is reached. This is a commonly used approach for scaling the microservices and virtual machines in the cloud. The open-source serverless platforms like OpenFaaS, Fission, Nuclio, Knative, and other tools use this approach for scaling the function replicas. In most of the Kubernetes-based serverless platforms, the Kubernetes Horizontal Pod Autoscaler (HPA) is used to scale based on the CPU and Memory limits set to the functions.

2.2 Related Work

Auto scaling of micro services is a well investigated area of research [29–31]. However, in edge and fog computing environments, especially latency sensitive data processing is most critical, and scaling of the processing components is of considerable interest [17]. This section briefly summarizes the recent work done in the context of scaling the serverless data processing architectures and models.

Schuler et al. [32] proposed a proactive custom Kubernetes (K8S) based approach for adaptive auto scaling of serverless functions for various workload profiles. Their proposed system was tested using Knative with workload based scaling, i.e., based on concurrency limits. However, the focus was on a steady workload with latency as the highest priority. Our proposed approach utilizes four distinct workload profiles, as relying solely on a steady workload is not suitable for all scenarios. On a similar line, Benedetti et al. [20] proposed a custom K8S based scaling of the serverless function based on the CPU resource utilization metrics. Their system was tested using the OpenFaaS platform. The goal was to find the optimal CPU limit and configure the same in the auto scaler of the system to reduce the latency of processing the arriving IoT workload. Li et al. [18] proposed KneeScale algorithm based CPU utilization as a metric for scaling serverless functions. They investigated spikes based on user workload for verifying latency and throughput.

Resource based scaling of serverless functions was extensively studied by Zafeiropoulos et al. [21] and Tari et al. [41], and they proposed various RL approaches such as Q Learning, Deep Q Learning, and DynaQ for scaling serverless functions based on CPU as metrics considering the discrete and continuous state space. The approaches were simulated using an Open Gym environment and integrated into the Kubeless serverless platform to tune the CPU configurations to optimize the latency in serving the function invocations. Along the side, Junfeng Li et al. [23] investigated the performance of workload and resource based scaling of various serverless platforms by considering the steady and spikes workloads.

The workload-based approach was applied to message queues to scale the microservices and investigated the elasticity based on the thresholds of QueueLength and Message arrival rate for IoT steady workloads [26, 40]. Similarly, Mahmoudi et al. [27] investigated the concurrency threshold as a workload based approach for scaling the serverless functions for steady workloads to reduce latency, and cost in cloud environments.

Table 1. Overview of related works

Article	Application Domain	Environment	Auto Scaling Technique	Auto Scaler	Components	User patterns	Performance Metrics
Li et al. [18]	IoT	Edge	Resource Utilization (CPU)	Custom scaler using K8S	Serverless Functions	Spikes	Latency, Throughput
Jegannathan et al.[19]	Cloud	Cloud	Resource Utilization (CPU)	Custom scaler using K8S	Serverless Functions	Spikes	Cold Start Time, Latency, Pod Count
Benedetti et al.[20]	IoT	Edge	Resource Utilization (CPU)	Custom scaler using K8S	Serverless Functions	Steady	Latency
Zafeiropoulos et al.[21]	Cloud	Cloud	Resource Utilization (CPU)	Custom scaler using K8S	Serverless Functions	Fluctuations	Latency, Throughput, SLA Latency,Success Rate
Palade et al.[22]	IoT	Edge	Resource Utilization (CPU)	K8S HPA	Serverless Functions	Steady	Latency, Success Rate
Junfeng et al.[23]	Cloud	Cloud	Resource Utilization, Workload Based (RPS)	K8S HPA, RPS	Serverless Functions	Spikes, Steady	Latency
Zhou et al.[24]	Cloud	Cloud	Resource Utilization (CPU)	K8s HPA	Serverless Functions	Spikes	Latency
Trieu et al.[25] Priscilla et al. [42]	Edge	Edge	Resource Utilization (CPU)	K8s HPA	Serverless Functions	Steady	Latency, Pod Count
Gotin et al.[26]	IoT	Cloud	Resource Utilization (CPU), Workload Based (QueueLength, Message Rate)	Custom	Message Queue	Steady	CPU Utilization, Throughput
Arjona et al.[15]	Cloud	Cloud	Workload Based (Event Rate)	KEDA	Serverless Functions	Steady	Delay, Events processed
Mahmoudi et al.[27]	Cloud	Cloud	Workload Based (Concurrency)	Custom	Serverless Functions	Steady	Latency, Cost
Wen et al.[43]	Cloud	Cloud	Correlation based	Custom	Serverless Functions	Steady	Latency
Our work	IoT	Edge, Fog	Resource Utilization (CPU), Workload Based (QueueLength, RPS,Message Rate)	KEDA, K8S HPA, RPS	Serverless Functions, Message Queues	Jump, Steady, Spikes, Fluctuations	Processing Time (latency), Success Rate, Pod Count, CPU and Memory Utilization, Fairness Index

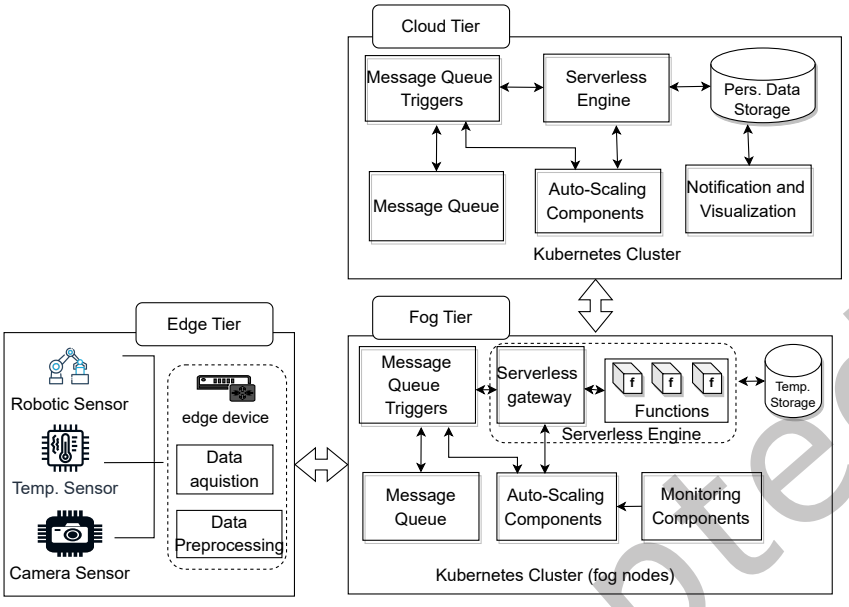


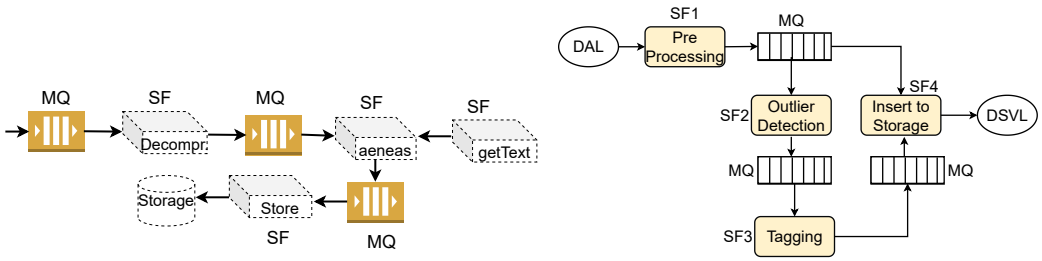
Fig. 1. Three tier System architecture containing SDP components

A comprehensive comparison of related works and our proposed work is presented in Table 1. The comparison is based on several characteristics such as the deployment environments (edge, fog, and cloud), the type of approach used for auto scaling, and whether the focus was on workload or resource based metrics. Further, we want to compare and understand which components (serverless, message queue) the authors focused on. The end user workload pattern is a key characteristic in investigating the scaling behavior, so we compared the user patterns considered with other works and also various performance metrics considered by the authors.

The comprehensive comparison shows that many of the works focused on scaling the serverless function using a resource utilization based approach, to improve the latency and throughput for steady workloads. The workload based approach was mostly used for scaling the message queue consumers and partly in serverless functions. However, most of the existing studies have focused on scaling individual serverless functions, without taking into account DAG based workflows, such as serverless data pipelines used for data processing in edge and fog environments. This is a crucial consideration, as scaling serverless functions heavily relies on intermediate data passing units (MQTs), which must be synchronized between the components. Apart from this, many of the works focused on only one type of workload as compared with our work. Our proposed performance evaluation work addresses this gap in the literature by taking into account the varying user patterns and function duration in various IoT applications.

3 SYSTEM ARCHITECTURE

Based on the related work, we described an overall three tier architecture, that includes the required services to accomplish the Message Queue based SDP at the fog and cloud tier. Further, we described essential services required for auto scaling the serverless and message queue components at the fog and cloud tiers. The system architecture in Figure 1 is composed of three layers with required services to handle the scalability of pipeline components for data processing. It contains three layers, namely, the Edge, Fog, and Cloud Tier, respectively.



(a) Aeneas (b) PuhatuMonitoring

Fig. 2. Two real-time applications and their pipelines

The data generated from the various end point IoT sensors gathered in the edge tier moved to the Fog tier and eventually to the Cloud tier for processing.

3.1 Edge Tier

The edge tier includes in situ IoT sensors deployed on the sensing environment, such as industrial floors, patients, and vehicles. Along with a set of sensors, tiny servers, network devices, and other computing devices are used to process the data with minimum operations. This tier manages the endpoint devices, collects and aggregates the data, and performs pre-processing operations such as compression, filtering, etc. We use custom Python services in this system to perform such data operations in the pipeline. The data pipeline initiated in this tier further continues into the fog tier.

3.2 Fog Tier

Fog Tier is mainly responsible for receiving data in the pipeline from the edge tier and process in a scalable manner. It constitutes a cluster of fog nodes configured with Kubernetes Engine. The Serverless platform, Object Storage services, and Message Queues are configured to store and process the data in the pipeline. Message Queue Triggers for Serverless functions and event-based scaling services are also part of the fog tier. The data received from the Edge tier is processed in a scaleable manner based on QoS demands and further moved into the Cloud tier for further processing and storage. Here, OpenFaaS is used as a Serverless Engine to create, deploy, and scale the serverless functions. Furthermore, we use RabbitMQ as a message queuing service to store intermediate data in the SDP. The RabbitMQ connector is used as an MQT for serverless functions whenever a message payload arrives in the queue on a specific topic with associated routing keys. We also use Kubernetes Event Driven Autoscaling (KEDA) for event-driven scaling. The metrics of the OpenFaaS gateway, RabbitMQ, CPU, and memory utilization of pods are scraped using the Prometheus monitoring service.

3.3 Cloud Tier

The cloud tier is mainly responsible for processing the data processing tasks (functions) forwarded from the fog tier. This tier constitutes the services for storing, visualization and generating alerts or notifications to the integrated business processes. The primary focus of the cloud tier is to provide persistent storage, which acts as data sink in the pipeline.

Table 2. Execution time, CPU, and Memory used by individual functions of two IoT applications

Application	Execution time (ms)	Functions	Function execution time (ms)	CPU consumed (milliCPU)	Memory used (MiB)
Aeneas	2145	Decompress	1	2	55
		aeneas	2000	546	243
		getText	100	5	4
		Store	44	6	7
PuhatuMonitoring	986	preProcess	6	6	97
		outlierIdentify	940	154	243
		tagData	5	3	32
		storeData	35	4	65

4 REAL TIME IOT APPLICATIONS

Considering the system architecture, in this section, we provide an overview of the IoT applications used in experiments. Aeneas [33] and PuhatuMonitoring [7] represent two IoT applications that take advantage of event-driven architecture to operate and manage data processing. The event-driven characteristic positions them as optimal candidates for including in a Message Queue-based SDP design. Furthermore, we present the realization of serverless data processing pipelines for these applications.

Aeneas application is a fog computing workload [33] and specialized in the automated synchronization of audio to a given text file also known as forced alignment. It automatically generates a synchronization map between a list of given text fragments and an audio file containing the narration of the text. The serverless data pipeline implementation derived from [7] is shown in Figure 2(a). An audio data forwarded from the edge tier is received in the fog tier and decompressed using *Decompress* function, further *aeneas* function retrieves the text data from the storage to synchronize with the input to a given user audio file. The data is generated and aggregated in the edge tier and forwared to the fog tier. The three functions *Decompress*, *aeneas*, *getText* resides in the fog tier. Finally, it processes the audio file and produces the alignment output in a JSON document and *store* function is used to store the final output to the cloud storage (data sink).

Aeneas based SDP contains four serverless functions and three message queues as described in Table 2 and Figure 2(a). The SDP works as follows, the compressed audio file arrives to the message queue in the fog tier, *Decompression* function is invoked, next output of the function is published into the queue. Next, *aeneas* function is triggered with audio file as an input and the output of *aeneas* which is a JSON document published to the queue and further *store* function is invoked to store it in the cloud tier. Here, *aeneas* function has a long running time with more CPU and memory utilization. The total execution time for processing a single data unit (audio file size of 512KB) by the SDP from source to sink was 2145ms and the execution time of the individual functions are mentioned in Table 2.

PuhatuMonitoring [7] is an IoT application for observing the water level changes in a wetland in the Puhatu Nature Protection Area (NPA), North West Estonia next to an open-pit oil-shale quarry. IoT devices are installed for monitoring the environment and sensor data is collected and analyzed by geologists to understand the effect of water level on the growth of the bog in the wetland. This system has several activities to perform such as data collection, analysis, detection of outliers using machine learning algorithms, tagging of outlier data, and storing the results. Figure 2(b) shows the message-queue-based SDP of the application. All the functions have small running times and minimum CPU and memory utilization as mentioned in Table 2. These functions reside in the fog tier, whereas data is compressed, aggregated and moved from edge tier.

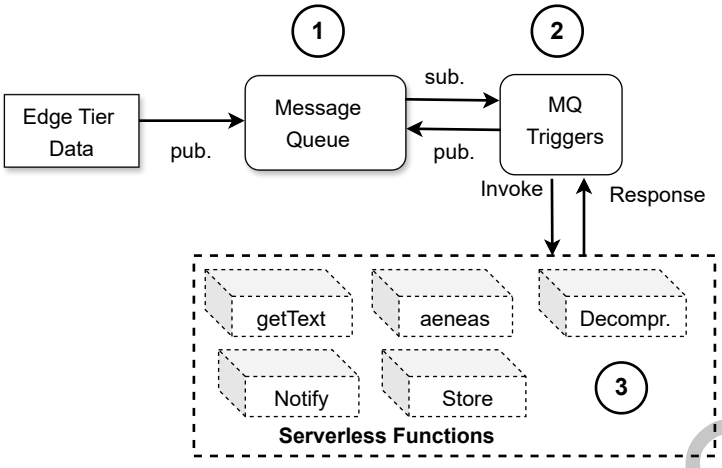


Fig. 3. Example of scalable components in the pipeline

5 AUTO-SCALING OF SERVERLESS DATA PIPELINES

By considering the three-tier system architecture and background knowledge, this section describes the auto scaling components of the Message Queue Based SDP and how those components are scaled using workload and resource based scaling approaches to provide seamless service to the stochastic IoT workloads.

The Message Queue based SDP contains mainly three components, namely, 1)*Message Queue*, 2)*Message Queue Server less Connector (Function Invoker or MQT)* and 3) *Serverless Functions* as shown in the Figure 3. We have considered two components to understand and investigate the scaling techniques based on workload and resource characteristics (MQ Trigger and Serverless Functions). The message queue is configured as a highly available cluster with multiple containers to handle the amount of workload used in the scalability testing.

Identifying the metric characteristics of the selected pipeline components is essential for defining scaling rules for various scaling approaches. As previously discussed, the two scaling approaches—workload-based and resource-based—are associated with distinct metrics. It is crucial to determine the relevant metrics for each pipeline component. Accordingly, we will define metrics for the Message Queue Trigger (MQT) and serverless functions (SF) below.

In **Message Queue Serverless Connector (Function Invoker or MQT)** component, as part of workload based scaling, two metrics are essential, namely, the number of messages (QueueLength) in the queue and the arrival rate of the message (incoming Message Rate, i.e., measured as messages per second). The scaling rule is defined as the threshold of QueueLength or Message Rate is met, then additional MQT instances are added to the system infrastructure. However, further investigation of choosing the optimal configuration is necessary because, for example, if we set QueueLength=1, then 100 messages arrive in the queue then 100 MQT instances will be spawned, which heavily hinders the system infrastructure; hence it's very crucial to decide the size of the metric in edge and fog computing environments. Here, MQT is a message subscribing service that triggers or invokes a serverless function, and the QueueLength metric decides the invocation rate of the functions. For resource based scaling, CPU and memory are the metrics used, with specific threshold values configured. CPU is considered a critical metric for scaling MQT instances, but identifying the optimal threshold value is crucial to avoid performance issues such as latency. Further investigation is necessary to determine the best threshold value [20] because that may hinder the latency and other performance issues.

Table 3. Scaling configurations of serverless functions and MQTs

Function belonging to	Functions/MQTs	K8s HPA						KEDA		RPS	
	Decompress	CPU threshold	Replica limit	cpu_limit	cpu_request	memory_limit	memory_request	QueueLength	MessageRate	rps threshold	Concurrency limit
Aeneas	aeneas	20	10	1000m	500m	512Mi	50Mi	3	0.3	0.2	20
	getText	50	15	500m	100m	100Mi	50Mi	4	0.3	0.2	50
	Store	50	15	500m	100m	256Mi	128Mi	4	0.3	0.2	50
PuhatuMonitoring	preProcess	50	15	500m	100m	256Mi	128Mi	4	0.3	0.2	50
	outlierIdentify	50	15	500m	100m	256Mi	128Mi	4	0.3	2	50
	tagData	50	10	1000m	500m	512Mi	50Mi	3	0.4	2	50
	storeData	50	15	500m	100m	256Mi	128Mi	4	0.3	2	50
MQT	MQT-aeneas	50	15	500m	100m	256Mi	128Mi	10	0.3	2	50
	MQT-store	50	15	500m	100m	256Mi	128Mi	10	0.3		
	MQT-outlierIdentify	50	15	500m	100m	256Mi	128Mi	10	0.3		
	MQT-tagData	50	15	500m	100m	256Mi	128Mi	10	0.3		
	MQT-storeData	50	15	500m	100m	256Mi	128Mi	10	0.3		

In **Serverless Functions** component, the scaling metric for the workload based approach is request per second (RPS) [23]. The RPS based approach is a default auto scaling mechanism in modern Serverless platforms. It's defined as the number of requests arriving at the function per second unit of time and is inherently correlated to the rate of invocation of the functions. To configure the scaling, the threshold value of the RPS metric needs to be calculated. This metric value varies for various functions, for example, the RPS values are significantly different from functions with long running and short running times. Our study focuses on both types of functions. Concurrency or capacity limit for each function instance or replica is another metric used in workload-based scaling [27]. Here, concurrency indicates the set of user requests processed by each function concurrently.

Table 3 provides an overview of the approaches, metrics, and corresponding SDP components. In the **workload-based approach**, we have considered RPS, Message Rate, and QueueLength as key metrics for scaling SF and MQT respectively. RPS metric is used as the default scaling approach in serverless platforms and we are not using any of the external services to perform the scaling. We used the Message Rate and QueueLength to scale the MQT. This is accomplished using an extra service known as Kubernetes Event driven Auto Scaler[1].

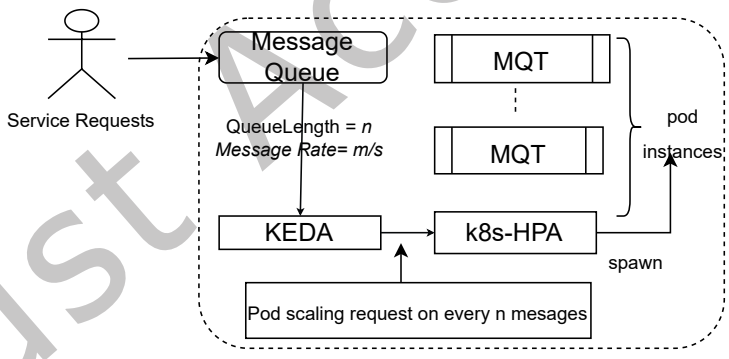


Fig. 4. KEDA based scaling architecture

Figure 4, shows the working details of KEDA and its integration with MQT to monitor the scaling metrics and apply the scaling decisions. KEDA can drive the scaling of the instances or pods in k8s cluster based on certain events such as topics in Apache Kafka, streams in Redis, or events in S3. KEDA has many scalers[2] that decide the activation and deactivation of the deployments. An example of KEDA based scaling of MQT is shown in Figure 4. The KEDA and MQT, both are configured as a service in the k8s infrastructure and the detailed experimental

[1]https://keda.sh/
[2]https://keda.sh/docs/2.9/scalers/

setup is described in Section 6. The internal architecture of KEDA has two components namely, *keda-metric API server* and *keda-scaler* respectively. Considering Figure 4, the *keda-metric API server* is used to monitor and pull the metrics (QueueLength and incoming Message Rate) on certain polling time intervals from the Message Queue. These metrics are used by *keda-scaler* to calculate the pod or instances count to scale and further send a signal to k8s Horizontal Pod Auto scaler to activate the deployment with the estimated count. The threshold values for the metrics such as QueueLength and Message Rate (MR) with associated queue names are configured in the deployment file and the corresponding ScaledObject is created in k8s cluster. The example of the ScaledObject file is shown below. The triggers section has two types of auto scaling triggers, firstly based on QueueLength and secondly on Message Rates. Here, queueName indicates the name of the queue to monitor, Value indicates the threshold to scale beyond this. The metric values are scrapped using the HTTP endpoint of the queue. The scaling decision is made based on the highest value achieved while scrapping the metrics.

```
- type: rabbitmq
  metadata:
    protocol: http
    mode: QueueLength
    queueName: aeneas
    value: "4"
```

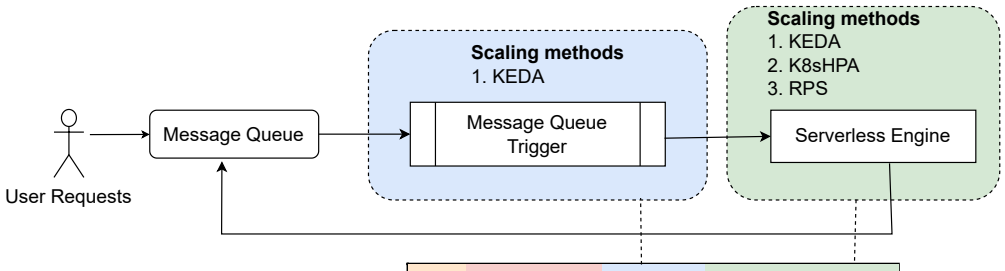
In the **resource based scaling approach**, we use Kubernetes Horizontal Pod Auto Scaler[3] (k8s-HPA) for both of the SDP components. The k8s-HPA is part of the K8s environment used for horizontal scaling, meaning adding extra pods or removing them based on metric values such as average CPU utilization, and average memory. In our experiments, the CPU is used as a scaling metric and configured into the auto scaler with a certain threshold.

Considering the Message Queue-based SDPs from the Figure 2(a) and 2(b), scaling metrics and approaches in Table 3 and further with the above questions, our objective is to investigate the performance of the scaling approaches on SDP components for various user workloads using performance metrics defined in Section 6. However, the question which arises is: ***Does scaling only Serverless Functions improve the efficiency of the pipeline or scaling both of the components***. To answer the question, our investigation focuses on using a combination of workload and resource based approaches to MQT and SF respectively.

Considering this, we define the six approaches for auto-scaling (see Figure 5) as follows:

1)KEDA and RPS(keda+rps) approach: In this approach, as shown in Figure 5, the combination of the KEDA and RPS is used to auto scale the SDP. KEDA is used to monitor the queues and further scale the consumers (Message Queue Triggers) that invoke the associated serverless functions. On the other side, the Request per second approach is used to monitor and scale serverless functions. In KEDA-based scaling, the MQT is created for each queue, for example, in the Aeneas application, three MQTs are created to invoke the functions. In ***2) KEDA and KEDA (keda+keda) approach*** as shown in Figure 5, the KEDA scalers are configured to watch the queue metrics and scale the function replicas based on the target values, for example, QueueLength or Message Rate metrics. Similar to approach 1, KEDA scalers are configured to monitor the queues and auto scale the queue consumers. However, In ***3) KEDA and K8s HPA (keda+k8shpa) approach*** as shown in Figure 5, the K8s Horizontal Pod Auto scaler is configured with target CPU on all serverless functions. The optimal target CPU is chosen for each function by running several experiments. Further, in ***4) No Scaling and KEDA (no+keda)***, ***5) No Scaling and K8sHPA (no+k8shpa)***, and ***6) No Scaling and RPS (no+rps)*** as shown Figure 5, scaling techniques are configured individually on serverless functions without scaling the MQTs. This approach enables the identification of key differences in the QoS of the SDPs.

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



Sr. No	Approaches	MQT	Serverless functions
1	keda+keda	KEDA	KEDA
2	keda+k8shpa	KEDA	K8sHPA
3	keda+rps	KEDA	RPS
4	no+keda	no	KEDA
5	no+k8shpa	no	K8sHPA
6	no+rps	no	RPS

Fig. 5. Approaches for auto-scaling the serverless data pipelines

6 PERFORMANCE EVALUATION

All the scaling mechanisms described in the previous section are realized, implemented for Aeneas and PuhatuMonitoring applications, and deployed on the system infrastructure as mentioned in Section 4. Further, experiments are conducted with real time workload patterns observed in Azure Cloud and provided a detailed analysis of the obtained results. We also provide underlying challenges and experiences learned during our experiments.

6.1 Performance Metrics

This subsection provides six potential performance metrics that are used to evaluate and check the efficiency of scaling SDP over variable user demands. The resource utilization metrics such as CPU, memory, and pod counts are collected using *Prometheus*, *cadvisor* software services. *PromoQL* (Prometheus Query Language) is used to calculate the metrics for the specific time period. Further, processing time and throughput are calculated using logs collected from the data source (Message arrived into the Queue) and data sink (final result stored in the MinIO storage).

6.1.1 Processing Time. In IoT applications latency is a primary concern, and it directly correlates to system scalability. In this regard, we calculated the processing time of user requests, received in the pipeline until the response reached back to the end user or storage unit. Processing time is measured in seconds in our experiments and it's the total time taken from the data source, processing units (serverless functions), intermediate storage units, and final data sink. The preliminary goal of our investigation is to see how the processing time is optimal by considering scaling approaches of various components in the SDP. We performed an extensive analysis of function execution and queuing time of user requests varies in scaling approaches. Further, we realized the processing time distributions using the Cumulative Distribution Function (CDF) to understand the efficacy of the scaling approaches.

6.1.2 Success Rate. The success rate is a metric used to find the number of users who succeeded to reach the data sink, alternatively, it measures the number of users processed on the given workload size. It is measured in percent and calculated by the ratio of users processed to the total number of users fed into the pipeline at a given time window.

6.1.3 CPU and Memory Usage. The resource utilization metrics such as CPU, memory, network, and disk, are essential parameters to compare the other scaling approaches because over and under provisioning of resources may lead to degrading the application and system infrastructure performance. In contrast, these are important metrics in IoT environments because of resource constraints in the fog infrastructure. We measure the average CPU and memory consumption over a range vector of 10s. The formula for CPU utilization of i^{th} user request in workload is as given below in equation 1 and is similar for memory utilization. We find the average CPU utilization by adding together the cpu usage of all the replicas of the serverless functions and MQTs, then dividing the sum by the total number of replicas.

$$CPU_t = \sum_{SF_k \in SF} (\frac{1}{j} * \sum_{FR_j \in FR} cpu(FR_{kj})) + \sum_{MQT_m \in MQT} cpu(MQT_m) \tag{1}$$

where $t = 10s$, $SF = \{SF_1, SF_2, \ldots, SF_k\}$ is a set of k number of serverless functions and $FR = \{FR_1, FR_2, \ldots, FR_j\}$ is a set of j replicas of k^{th} function and $MQT = \{MQT_1, MQT_2, \ldots, MQT_m\}$ is a set of m Message Queue Triggers.

6.1.4 Number of Pods. Whenever scaling activity happens in the system, an extra number of pods or replicas or instances are added to the system to accommodate the huge demand. However, comparing the pod counts between various scaling approaches provide insight into its over and under usage during its scaling lifetime. We sum up all the replicas of the serverless functions and MQTs used during the experiment for each workload.

$$Count(Replicas_t) = \sum_{\mathcal{SF}_{\|} \in \mathcal{SF}} Count(SF_k) + \sum_{\mathcal{MQT}_{\|} \in \mathcal{MQT}} Count(MQT_m) \tag{2}$$

where t with time window of $t = 10s$ step in metric collection, $SF = \{SF_1, SF_2, \ldots, SF_k\}$ is a set of k number of serverless functions and $MQT = \{MQT_1, MQT_2, \ldots, MQT_m\}$ is a set of m Message Queue Triggers.

6.1.5 Fairness Index. This metric is used to calculate the fairness index for the processing time of user requests in serverless data pipelines. The fairness index determines how user requests are served using auto scaling components in the SDP. We calculate using Jain's fairness [34] index (JFI) and values are between 0 to 1, where a values 1 represents good fairness, i.e, all user requests have lower processing time. The value 0 indicates the disproportion processing time. Overall, JFI provides insights into the level of variability in the processing time of user requests and can be used to compare the fairness of different scaling approaches. By analyzing the fairness index, we can identify potential improvements in optimizing the configurations and resources of serverless and message queue scaling components. The formula to calculate is:

$$JFI = \frac{(\sum PT_i)^2}{n \times (\sum_i PT_i^2)} \tag{3}$$

where PT_i is Processing Time (PT) calculated for i^{th} user request or workload.

6.2 Serverless Workload Patterns

In this section, we explore the workload patterns of serverless applications based on Azure Function Traces [35]. These traces are available along with invocation logs for a duration of two weeks. To analyze the invocation patterns, we considered two days of data consisting of 41407 functions triggered by various sources such as Event, HTTP, queue, storage, orchestration, and others. Out of these functions, we selected 230 functions that are triggered by queues, which may include Service Bus, Queue Storage, RabbitMQ, Kafka, and MQTT. We followed the approach described in ServiBench work [14] to select and classify the workloads. We categorized the invocation patterns of the 230 serverless functions into four types: Fluctuating, Spikes, Jump, and Steady workloads as depicted in Figure 6. Our chosen data set indicates that 5.2% of the functions exhibit *steady* workloads

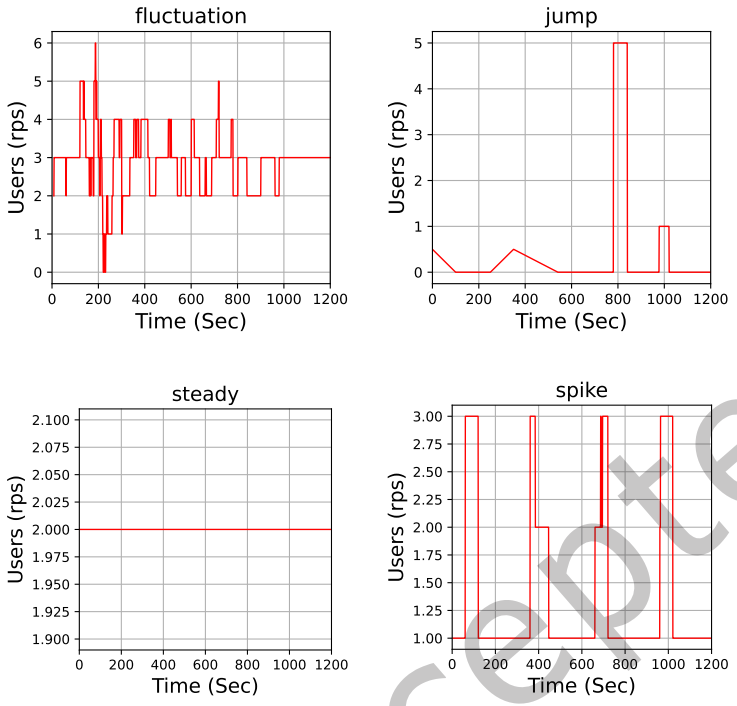


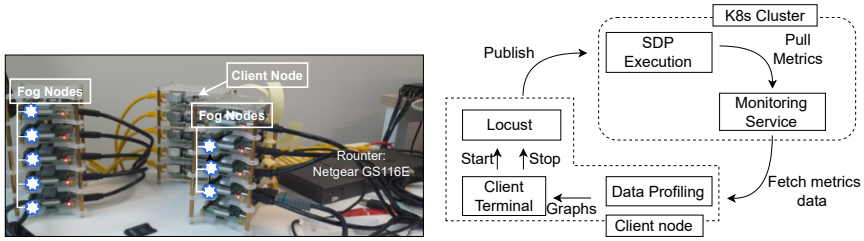
Fig. 6. Azure serverless workload invocations patterns

Table 4. Characteristics of the workload

Characteristics of the Workload	Fluctuation	Jump	Steady	Spike
Mean	2.9	0.8	2	1.87
Standard deviation	1.2	1.56	0	0.97
Minimum	0	0	2	1
25% of the workload	2	0	2	1
50%	3	0	2	2
75%	4	1	2	3
Maximum	6	5	2	4
Total requests generated	3442	1055	2390	1646

that remain stable throughout the invocations over time. The *fluctuating* workload (34.37%), shown in Figure 6, represents the load with constant fluctuations and frequent small bursts during invocations. The *spike* workload (27.08%) indicates occasional extreme bursts with or without a steady base load. Finally, the *jump* workload (30.5%) represents sudden load changes that occur only for a moment and may exist for a temporary period.

To use the four workload patterns, it was necessary to transition from per-minute to per-second invocation patterns. This transformation was achieved by leveraging fractional Brownian motion, as detailed in Scheuner et al.'s work [14], which enabled the synthesis of perturbations at the granularity of seconds while preserving the



(a) Hardware setup with client and fog nodes connected with router

(b) Process flow of experiments

Fig. 7. Hardware setup and process flow of experiments

overarching characteristics observed in the Azure traces. The adjustment also involved reducing the workload intensity; for instance, in the case of the spike workload, the maximum intensity originally was at 450 requests per second (rps), and this was lowered by 4 rps. Similar reductions were applied to the intensities of other workloads, as illustrated in Figure 6. The specific characteristics of the workload are described in Table 4, with values expressed in requests per second (rps). The fluctuation workload exhibited a peak intensity of 6 rps, while the steady workload maintained a constant intensity of 2 rps. The standard deviation metric signifies the extent of aggressive changes in workload intensity, with the jump workload displaying a deviation of 1.56 rps. In the case of fluctuation workload, 75% of the instances surpassed 4 rps. The duration of the experiment was set at 1200 seconds, during which the fluctuation workload generated 3442 user requests, while the jump workload produced 1055 user requests.

6.3 Experimental Setup

The fog tier includes a hardware unit consisting of nine Raspberry Pi (RPi) 4B model clusters with specifications of Quad-core Cortex-A72 (ARM v8) 64-bit 4 CPU cores, 8 GB RAM, and 60GB storage. Cloud tier with two virtual machines with 4 vCPUs, 8 GB RAM resembling the capacity of *m2.medium* size of AWS EC2 instance is provisioned from the University of Tartu's OpenStack cloud. The edge RPi 3B model is used as a gateway, and RPi 4B model is used as a client to simulate the user behavior. All the edge and fog tier nodes are connected over LAN using Netgear GS116E-200PES, 16-Port Smart Managed Gigabit Switch with speed up to 2000Mbps. Figure 7(a) shows the hardware setup with RPis stack connected to the router with the network system.

The lightweight Kubernetes (k8s) Engine k3s (v1.25.5+k3s1) is installed in the fog tier, and similarly, k8s engine v1.26 is configured in the cloud tier. OpenFaaS serverless platform *faas-netes*(0.23.0)[4] is installed in the fog and cloud tier as a serverless engine to create, manage and scale the serverless functions. RabbitMQ is installed as three pod clusters in the fog layer, with one virtual host, one exchange, and three message queues. The MinIO object storage service is configured on the cloud tier to store the results after processing the serverless data pipeline acting as a data sink. The serverless functions are created and deployed using the OpenFaaS Command line interface. Further, Python 3.9 run time is used to develop the serverless functions. The Prometheus, along with the c-advisor, captures and monitors the k8s service metrics. Further, Prometheus REST API is used to get the metrics after completing the experiments.

Since fog tier nodes are with ARM architecture, some of the services like KEDA, OpenFaaS-RabbitMQ-Connector (MQT), and RabbitMQ k8s operator are rebuilt (from other architecture-based deployments to ARM) and further

[4]https://github.com/openfaas/faas/releases/tag/0.23.0

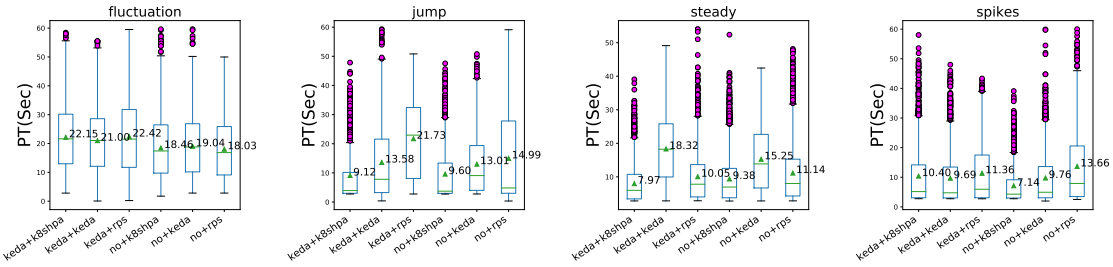


Fig. 8. Comparison of processing time for Aeneas application

deployed on the RPi-based k8s cluster. Those docker images can be found here in the docker hub[5]. After setting up hardware and software services, Jupyter Notebook is used to code and deploy the serverless data pipeline components of the Aeneas and PuhatuMonitoring applications. The process flow of experiments conducted is shown in Figure 7(b). The locust tool is used to publish the user requests with data according to Azure workload scenarios (Jump, fluctuation, steady, and spikes) to the message queue and initiate the pipeline. The data input size for the Aeneas application includes the range from 512KB to 1MB audio files considered from the projects[6, 7]. We use the real-time data collected in the Puhatu IoT application as input in the PuhatuMonitoring application. Once user requests were processed by SDP and final outputs were stored in the data sink (MinIO Storage in Aeneas and Influxdb in the PuhatuMonitoring application). After the SDP execution of each workload scenario, CPU, Memory, and Pod Count metrics were pulled from Prometheus using PromoQL Query as an HTTP invocation with step 10s. We collected Function Execution Time, and Queueing Time using custom services (python code) by scrapping from the OpenFaaS gateway. Finally, the Data Profiling service was used to parse and generate the required performance monitoring graphs. The source code of the two SDPs of the real-time applications and performance analysis notebooks can be found in GitHub[8].

6.4 Results and Discussion

An auto-scaling behavior largely depends on the arrival rate of user workloads. So, performance validation of auto-scaling needed to generate stochastic user workloads, and to mimic such user behavior, python-based locust[9] tool, which is mainly used for HTTP-based load testing, but we have written the python scripts[10] to publish the user workload into RabbitMQ channels. Several experiments are conducted to realize the results with a minimum error rate over all the experiments. The following subsections provide a detailed analysis of the Aeneas and PuhatuMonitoring application of the obtained results.

6.4.1 Aeneas Application. The processing time is shown in Figure 8 for the Aeneas application and is measured in seconds. These four box plots provide the overall processing time of four quartile groups for six scaling methods. The processing time for *fluctuation* workload is shown in Figure 8(a), the *no+rps* have a median value of 18.03s, as compared with others, which shows that with this approach most of the users have faster processing time, whereas *keda+k8shpa* has a highest average processing time of 22.15s with a maximum value of 56s. Figure 8(b)

[5]https://hub.docker.com/u/shivupoojar
[6]https://gitlab.doc.ic.ac.uk/st220/COSCO/-/tree/master/framework/workload/DockerImages/Aeneas/assets/audio
[7]https://github.com/readbeyond/aeneas
[8]https://github.com/shivupoojar/autoscalingsdp.git
[9]https://locust.io/
[10]https://github.com/shivupoojar/autoscalingsdp/../k6_locust.py

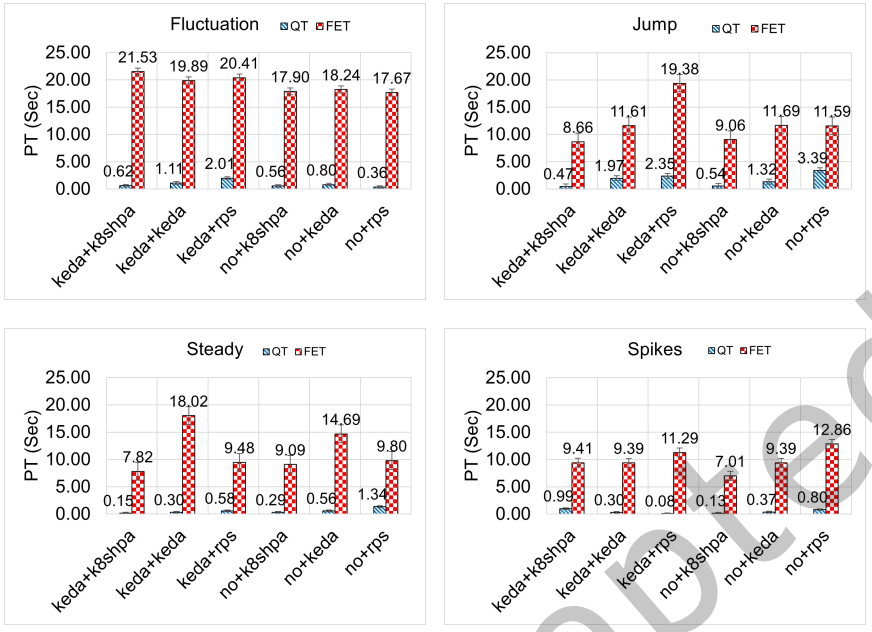


Fig. 9. Average Function Execution Time (FET) and Average Queuing Time (QT) of Aeneas application over scaling approaches

shows for *jump* workload, the *keda+k8shpa* experienced an average PT of 9.12s with outliers, whereas *keda+rps* had a maximum of 21.73s. This indicates that *keda+k8shpa* is capable to handle the jump workloads in a given time. Figure 8(c) and 8(d) show the processing times of steady and spikes workloads.

The *keda+k8shpa* and *no+k8shpa* experienced median values of 7.97s and 7.14s respectively, in steady and spikes workloads. This indicates that *no+k8shpa* can handle the spikes workload, this is because Aeneas has a long running function with compute intensive, and CPU-based autoscaling works satisfactorily here. In the *K8shpa*-based approaches, the response time and data loss for the long-running and compute-intensive function (aeneas) are lower than those of *rps* and *KEDA-based* approaches. In a serverless platform, the rps threshold value is the same for all functions, which does not provide optimal performance for long-running functions, resulting in more data loss. However, in the *K8shpa* approach, the optimal CPU threshold value of (20%) for Aeneas is configured, which reflects the optimal scaling decision and achieves a lower response time.

Further, the processing time of SDP is the sum of function execution time (FET) and queuing time (QT). To understand the correlation of queuing time and efficiency of scaling approaches, we show the comparison of the average FET and QT of the Aeneas application in Figure 9. In *fluctuation* workload, *no+rps* had the lowest QT of 0.36s and FET of 17.67s, whereas *keda+k8shpa* experienced 0.62s and 21.53s of QT and FET, respectively. This suggests that *rps* for serverless functions can handle frequent alterations in the event patterns of the workload, ranging from a minimum of 2.6rps to a maximum of 4.25rps of input. With *keda+k8shpa*, hpa configured on serverless functions wait for the cpu to reach the threshold before provisioning the replicas, which increases the time taken for the function to execute. However, in spikes and jump, the *keda+rps* and *keda+k8shpa* have lower QT and FET of 0.08s, 11.29s, and 0.47s, 8.66s, respectively. This result indicates that the rps approach in serverless functions can be flexible for frequent changing workloads, whereas the k8shpa approach can easily deal with

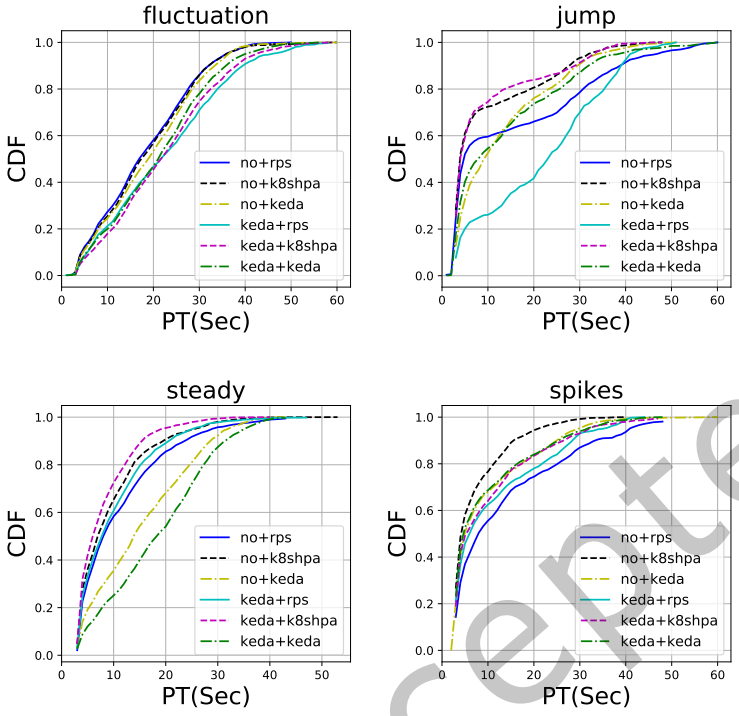


Fig. 10. Cumulative Distribution Function of Processing Time for Aeneas application over scaling approaches

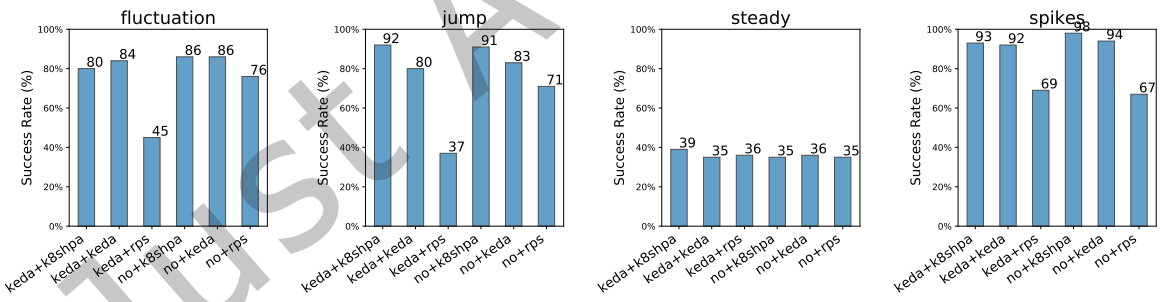


Fig. 11. Success rate of Aeneas application over scaling approaches

sudden and aggressive workloads. In *steady* workload, *keda+k8shpa* has minimum QT and FET compared to other approaches.

To understand the variance and distribution of processing time of all workloads, we calculated and plotted the Cumulative Distribution Function (CDF) for all workloads. The CDF for the Aeneas application for workloads is shown in Figure 10. It shows that, in fluctuation workload, 80% of the workloads have the PT of below the $\approx 28s$ in *no+rps*, whereas *keda+k8shpa* has $\approx 34s$. The RPS approach has a faster scaling activation than the k8shpa

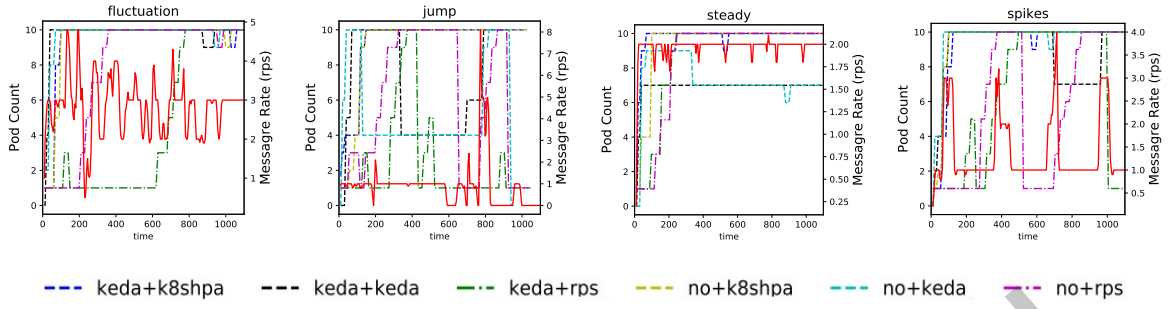


Fig. 12. Scaling pattern of Aeneas function w.r.t to message arrival rate

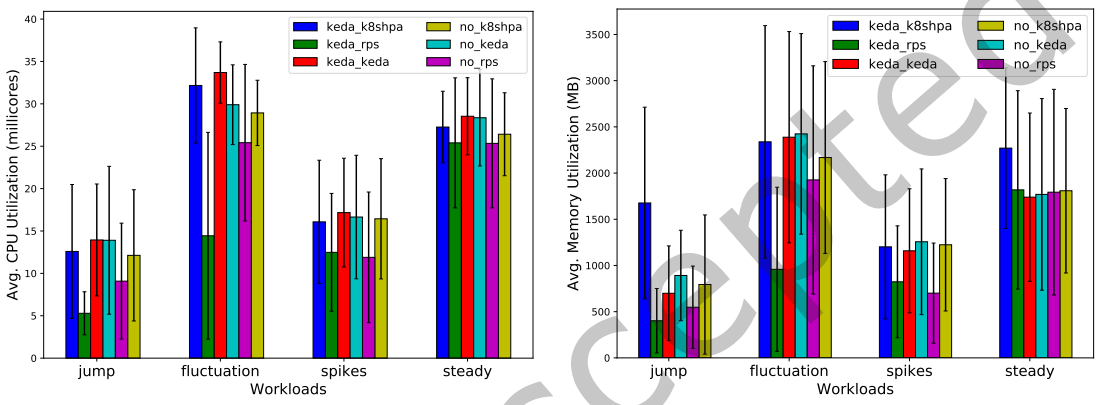
based approach configured for serverless functions, potentially reducing the PT. Surprisingly, *keda+k8shpa* have 80% of below 12s in steady workload. This is because the scaling decision in k8shpa is consistent w.r.t. steady workload. Similarly, in jump workload, *keda+k8shpa* have 80% have PT below 15s. However, for spikes, workload *no + k8shpa* 90% of them are within 27s compared to other approaches. The CDF results show that *no+rps* can deal efficiently with fluctuations, while *keda + k8shpa* works better for jump and steady workloads. However, *no+k8shpa* works satisfactorily to deal with spikes. The overall results indicate that the long-running functions of the Aeneas application were handled by *k8shpa* for jump,steady and spikes.

The success rate is a crucial performance metric that indicates the percentage of user requests or workloads that are processed and stored successfully in the data sink. It is used to evaluate the efficiency and reliability of different scaling approaches. Figure 11 presents a comparison of success rates for various scaling approaches used in the Aeneas application. The figure shows that *no+keda* and *no+k8shpa* achieved a similar success rate of 86%, indicating their inability to handle 14% fluctuating workloads. However, in steady workloads, *keda+k8shpa* had the highest success rate, processing 98% of users, while in jump workloads, it achieved a success rate of 91%. Conversely, *no+k8shpa* demonstrated the highest success rate of 98% in spike workloads. The *keda+rps* was least reliable in dealing with the fluctuation workload with 45%, and similarly in jump with 38%. In *keda+rps* approach, *keda-based* scaling approach in MQT dequeues user requests and invokes serverless functions, *rps* lead to data loss due to time out of the user requests queued in the serverless function internal queue. The maximum data loss appeared for the *aeneas* function which was with a long-running time.

In order to gain a deeper understanding of how the scaling approach, which relies on the stochastic arrival of messages in a queue, behaves, we conducted an analysis of the scaling patterns exhibited by the Aeneas function under different workloads. This analysis is presented in Figure 12. The x-axis represents the pod count, while the y-axis represents the message arrival rate measured in messages arrived per second. Our observations from Figure 12 reveal that the *no+rps* and *keda+rps* approaches progressively increase the replicas (pods) as the message arrival rate increases. On the other hand, the *keda+keda*, *no+keda*, *keda+k8shpa*, and *no+k8shpa* approaches aggressively react to workload spikes but maintain a constant number of replicas due to fluctuations in the arrival workload. In the case of a steady workload, the RPS based approaches react slowly and keep incrementally raising the replicas until saturation was reached. On the other hand, the k8shpa based approaches raise the replicas over the first few seconds, while KEDA based approaches raise the replicas instantly and then downscale to a steady count of 7 replicas. Interestingly, the KEDA and k8shpa based approaches were more volatile to changes in workloads, while the RPS based approaches show similar progressive behavior. However, *no+k8shpa* approach aggressively reaches a maximum count of replicas despite the load downscaling in the workload.

Table 5. Overview of performance metrics of scaling approaches in Aeneas application

Workload	jump				steady						spikes						fluctuation					
Message Queue Trigger	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda
Serverless	rps	rps	k8shpa	k8shpa	keda	keda	rps	rps	k8shpa	k8shpa	keda	keda	rps	rps	k8shpa	k8shpa	keda	keda	rps	rps	k8shpa	k8shpa	keda	keda
Count	1055	1055	1055	1055	1055	1055	2390	2390	2390	2390	2390	2390	1646	1646	1646	1646	1646	1646	3442	3442	3442	3442	3442	3442
Success Rate	72	38.1	91.5	**92.5**	83.8	80.8	88.8	90.4	89.6	**98.3**	92.2	87.3	67	69.3	**98.3**	93.6	94.2	92.5	76.6	45.4	**85.9**	79.9	86.2	84.4
PT Sum (sec)	11390	8736	9260.8	8905.4	11497	11574	23644	21714	20084	18737	33600	38229	15066	12965	**11549**	16030	15129	14764	47539	35059	54564	60936	56499	61005
PT Mean	15	21.7	9.6	**9.1**	13	13.6	11.1	10.1	9.4	**8**	15.2	18.3	13.7	11.4	**7.1**	10.4	9.8	9.7	18	22.4	**18.5**	22.2	19	21
PT Max	59.1	50.8	47.6	**47.8**	50.8	59.3	48.1	54	52.4	**39.1**	42.4	49.1	60	43.3	**39.1**	58	59.9	48	50	59.5	**59.6**	58.4	59.6	55.5
PT Min	0.4	2.8	2.7	**2.8**	2.8	0.4	2.8	2.9	2.8	**2.8**	2.8	2.8	2.5	2.7	**2.8**	2.8	2	2.8	2.8	0.2	**1.8**	2.8	2.8	0.1
PT STD	15.6	13.3	10.4	**10.1**	10.7	12.6	8.7	7.5	7.2	**5.7**	9.5	10	12.8	10.6	**6.2**	10.1	9.4	9.6	10.3	13	**10.5**	11.8	10.5	11.2
PT 90th	38.6	38.9	27.8	**28.6**	29.4	31.7	23.5	20.6	19.5	**15.4**	28.6	31.3	33.3	28.3	**15.5**	26.6	25	25.7	32	39.5	**32**	38	33.5	35.8
Avg FET	11.6	19.4	9.1	**8.7**	11.7	11.6	9.8	9.5	9.1	**7.8**	14.7	18	12.9	11.3	**7**	9.4	9.4	9.4	17.7	20.4	**17.9**	21.5	18.2	19.9
Avg QT	3.39	2.35	0.54	**0.47**	1.32	1.97	1.34	0.58	0.29	**0.15**	0.56	0.3	0.8	0.08	**0.13**	0.99	0.37	0.3	0.36	2.01	**0.56**	0.62	0.8	1.11
JFI	0.52	0.27	0.54	**0.55**	0.59	0.45	0.59	0.45	0.59	**0.45**	0.59	0.45	0.59	0.45	**0.59**	0.45	0.59	0.45	0.59	0.45	**0.59**	0.45	0.59	0.45
Avg CPU	9.08	5.29	12.13	12.59	13.91	13.94	25.34	25.4	26.41	27.26	28.36	28.54	11.89	12.48	16.44	16.08	16.65	17.17	30.42	19.43	33.93	37.17	34.9	38.7
Avg Memory	548	402	794	1676	892	699	1793	1818	1809	2270	1769	1739	701	823	1224	1202	1256	1159	1926	958	2168	2338	2424	2388
Pods Count	23	9	15	**15**	18	19	31	33	19	**19**	21	21	26	29	**18**	18	23	22	33	22	**19**	19	23	23



(a) CPU utilization (millicore)

(b) Memory utilization (MB)

Fig. 13. Comparison of CPU and Memory utilization of Aeneas application for various scaling approaches

We computed the Jain's fairness Index (JFI) for all workload patterns of the Aeneas application to assess the variation in processing time as compared to the expected average processing time. Table 5 presents the results. For jump workloads, all scaling approaches exhibited moderate fairness in processing user requests, but the *keda+rps* approach had a higher fairness index than the others, despite its low success rate and low CPU utilization. However, it processed 402 requests out of 1055 user requests, with a success rate of 38.1% which degrades the QoS. In steady workloads, both the *no+keda* and *no+rps* approaches had relatively higher fairness indexes compared to other scaling approaches. The fairness index for steady workloads indicates that all approaches were moderately fair in processing user requests. However, for fluctuation workloads, the fairness index is higher, suggesting that a larger proportion of user requests are processed close to the average processing time with less variance.

When processing IoT data in fog environments, the utilization of CPU and memory by application components is crucial for determining the effectiveness of scaling approaches. These metrics provide insights into the amount of resources consumed by application components and can aid in evaluating the efficiency of the scaling approach. Figure 13(a) and Figure 13(b) show the CPU and memory utilization of the Aeneas application in fog environments respectively. In Figure 13(a), the y-axis represents the average CPU consumption measured in millicore, with 1000m indicating 1 core, while the x-axis represents the workload patterns.

For jump workloads, the *keda+rps* scaling approach has a relatively low CPU consumption of around $\approx 19 millicore$, while the KEDA based approaches (*no+keda*, *keda+keda*) have maximum CPU consumption of around

$\approx 38millicore$. This indicates that the *no+rps* approach may be more efficient in terms of CPU utilization for this workload pattern. In spikes workload, similar to jump, *keda+rps* has a lower CPU utilization of $\approx 25millicore$, whereas *keda+keda* had a maximum CPU utilization. In steady workloads, both *keda+rps* and *no+rps* have a minimum CPU utilization of around $\approx 19.43millicore$, while KEDA based approaches consumed the maximum CPU. Considering the overall CPU utilization metrics, KEDA based approaches consume more CPU resources because the scaling decision is based on the QueueLength and hence more replicas are spawned, leading to more CPU consumption.

The memory consumption of scaling approaches across various workloads is shown in Figure 13(b). The y-axis represents the memory utilization measured in MegaBytes (MB). In jump workload, *keda+rps* consumes minimum memory, whereas the *keda+k8shpa* consumes more memory. However, *keda+rps* has a success rate of 38.2%, hence consuming less memory. The *keda+k8shpa* in steady workload utilized more memory of $\approx 2270MB$ as compared with other approaches. In spikes, *no+rps* had minimum memory utilization, whereas K8shpa based approaches have maximum memory utilization. Similarly for fluctuation, keda, and k8shpa based approaches have more memory utilization, *keda+rps* has minimum memory consumption, however, the success rate is 45%.

Aside from the previously mentioned performance metrics, there are several other crucial parameters that influence the efficacy of scaling techniques for the Aeneas application. These parameters include the Processing Time Standard Deviation (PT STD), the 90th percentile of PT, the Minimum of PT, and the Maximum of PT. PT STD is particularly important as it indicates the consistency of processing time across a given user workload. A lower PT STD implies strong consistency in the processing times, which is essential for the real time processing of audio data in Aeneas. Similarly, the 90th percentile of PT, Minimum of PT, and Maximum of PT are significant performance metrics that can help determine the best and worst case scenarios for the system's response time. Together, these performance parameters provide a comprehensive view of the system's efficiency, allowing developers to optimize scaling approaches for Aeneas.

Along the side, Table 5 provides an overview of all the performance metrics of all scaling approaches with different workloads. From Table 5 it is evident that ***keda+k8shpa*** is able to efficiently process **jump** workloads with a success rate of 92.5%, with minimum PT mean, PT STD, FET, and QT of 9.1s, 10.1s, 8.7s and 0.47s respectively. However, higher utilization of CPU and Memory utilization but a moderate replica count of 15. Considering PT (latency) as a key metric, the ***keda+k8shpa*** works a better scaling approach for auto scaling the message queue consumers (MQT) and serverless functions. Similarly in **steady** workload, ***keda+k8shpa*** processed adequately with a success rate of 98.3% and with a minimum in other performance metrics. However in **spikes**, without any scaling approach at MQT, and k8shpa scaling approach at serverless functions worked efficiently with a success rate of 98.3%. On the other side, ***no+rps*** and ***no+k8shpa*** could able to handle the fluctuation workload adequately, but ***keda+rps*** is not efficient in dealing workload. Overall results show that for the Aeneas application, ***k8shpa*** based scaling approach works better due to long running functions in the pipeline.

6.4.2 PuhatuMonitoring Application. Similar to the Aeneas application, the performance analysis of the PuhatuMonitoring (PM) application is described in this subsection. The PM application has a short running time and moderately low compute intensive functions as discussed in Section 4. The overview of processing time for various workloads is shown in Figure 14. In fluctuation workload, *no+rps*, *keda+k8shpa* and *keda+rps* approaches yielded lower processing time with a mean of $0.66s$. This shows that the *k8shpa* and rps based approach works well, however the success rate of *k8shpa* was higher, which means lower data loss. Similarly in jump workload, *no+rps*, *no+k8shpa* and *no+keda* processed PM data with mean of $\approx 0.70s$, however *keda+k8shpa* had maximum value of $7.3s$. Scaling approach configured serverless functions capable to deal the jump workload, where as no scaling of MQT was necessary. In steady workloads, *keda+rps* and *no+keda* processed data with a mean of $0.79s$, where as *no+rps* has a maximum value of $10.92s$. On the other hand, in spikes workload *no+rps*, *no+k8shpa* and *no+keda* similar minimum mean of $\approx 0.5s$ and *no+k8shpa* has maximum of $3.8s$.

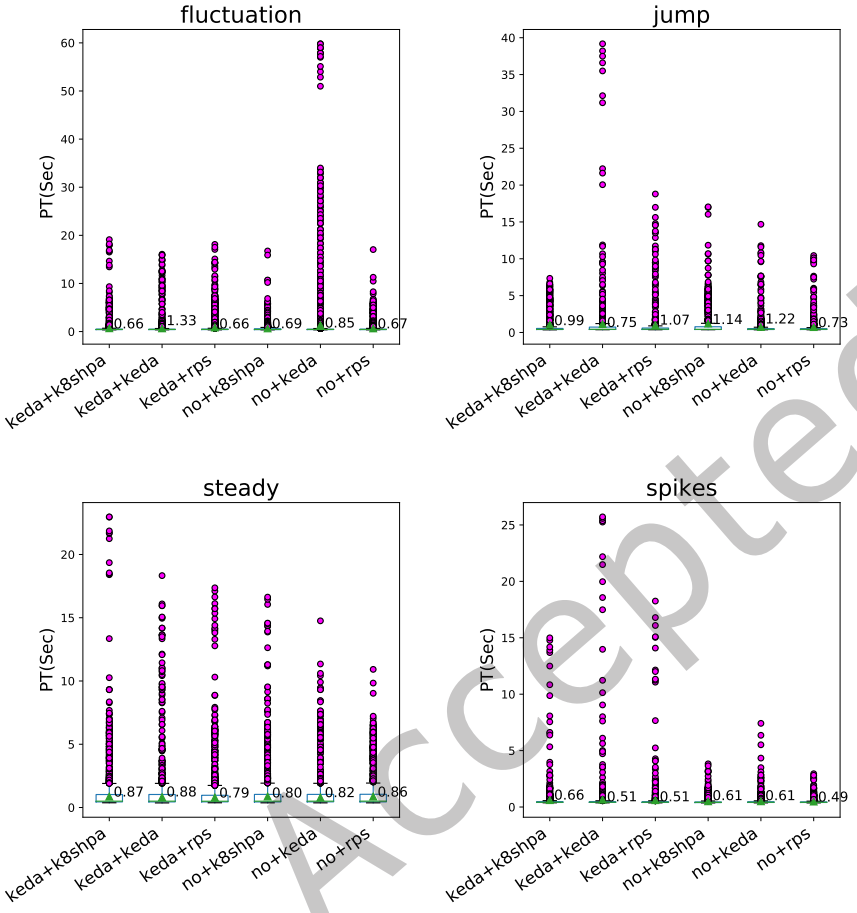


Fig. 14. Comparison of processing time of Puhatu application

Furthermore, Figure 15 shows the comparison of the average FET and QT of PM applications for various workloads. Three approaches (*no+rps,no+k8shpa* and *keda+k8shpa*) have average FET of 0.6*s* but *keda+keda* has minimum average FET of 0.56*s*, however the QT was maximum of 0.12*s*. Surprisingly, the QT mean was higher than FET, which indicates longer waiting in the queue. This is due to the longer waiting time in the message queue, where no scaling of MQT and *keda* scaling approach in serverless functions scale the functions based on the configured QueueLength, which is not sufficient to handle load. In steady workload, *no+rps* and *no+keda* have lower mean QT of 0.07 but *keda+rps* has lower FET as compared with the other approaches. For spikes workload, *no+k8shpa* has lower mean QT of 0.04*s* but moderately more FET time. The *no+keda* was efficient in handling the jump workloads with mean QT of 0.06*s*.

The CDF of PT of the PM application is shown in Figure 16, In fluctuation workload, the *no+k8shpa* has 86% of them are processed within 1s The *no+keda* and *keda+rps* have similar distributions initially. In the steady workload, *keda+rps* have 90% of them have PT of 1.1s, however, *no+rps* has 1.3s not performing better as compared to the others. Interestingly, *no+rps* performed well, with 90% of the workload's PT under 0.58s, whereas *keda+keda* had

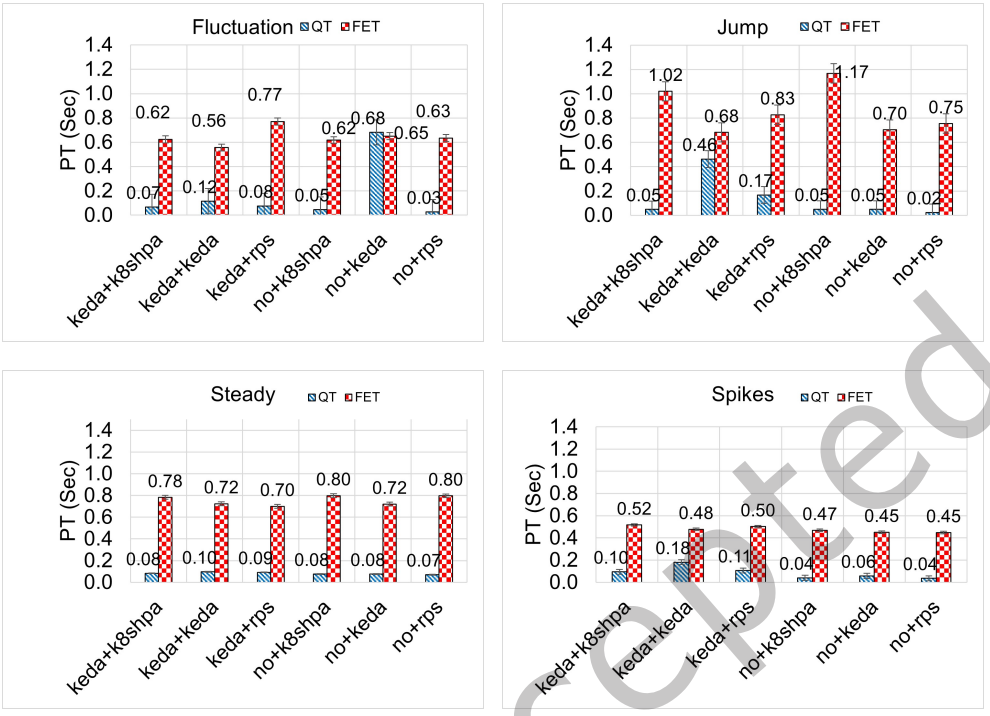


Fig. 15. Function Execution Time (FET) and Queuing Time (QT) of PuhatuMonitoring application over scaling approaches

1% of workload's PT over 25s as compared with other approaches. In jump workloads, the *no+rps*'s distributions show that 90% of them were in 1.2s and work better as compared with other approaches.

The success rate of various workloads for PuhatuMonitoring application is shown in Figure 17. For fluctuation workload, *no+k8shpa* has a higher success rate of 99% and *keda+rps* was less with 95%. However, in steady workload *no+keda* was with a high success rate of 98%, whereas *keda+rps* and *keda+k8shpa* were similar with 96%. The *no+rps* is capable to handle 99% user requests in jump workload. Interestingly, *keda+k8shpa* was able to succeed in processing 89%, whereas other approaches reached a success rate of 99% in spikes workload.

The scaling patterns in response to the arrival rate of messages of outlierDetection function are shown in Figure 18. In fluctuation workload, *keda+rps* and *no+rps* progressively increases the replicas based on the increase in message rate, however, *keda+rps* reached the maximum of 14 replicas due to parallel requests arriving from keda based MQT. The *keda+k8shpa* and *no+k8shpa* increase the replica initially, and further, tune to the arrival rate. All the scaling approaches become to stagnant to certain replicas over a period. However, in steady workloads, *keda+keda* and *no+rps* increases the replicas aggressively based on message rate and becomes stationary with certain replicas. But, RPS based approaches increase the replica progressively and reach the stationary value. Interestingly in jump workload, due to the sudden raise in arrival rate, all approaches took the decision for scaling, however, scale down for rps was faster. In spikes, *keda+keda* and *keda+rps* reach the stationary point after initial scaling, not eventually reactive to the arrival rate, whereas *keda+k8shpa* and *no+keda* reacts to the spike workload. Considering this scenario, *keda+keda* and *keda+rps* may not be suitable for fog environments due to the overconsumption of resources.

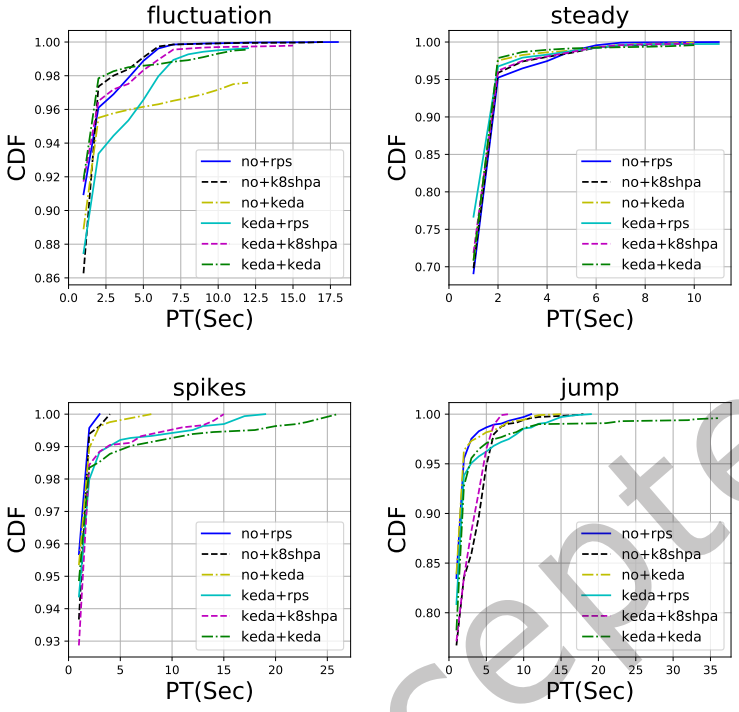


Fig. 16. CDF of processing time for puhatu application over scaling approaches

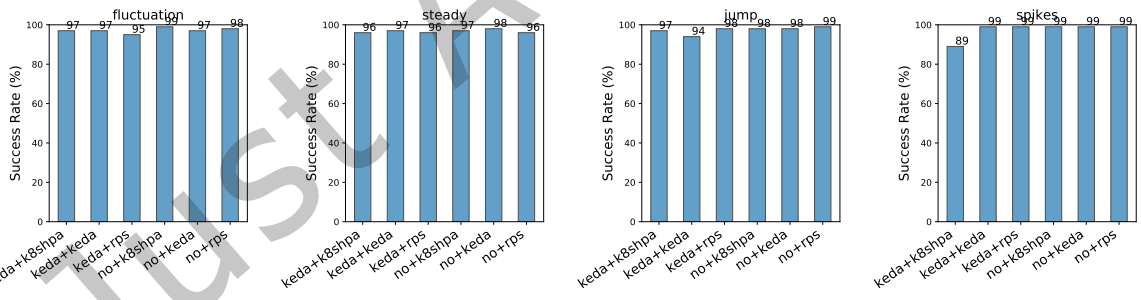


Fig. 17. Success rate of PuhatuMonitoring application over scaling approaches

The Jain's Fairness Index (JFI) for fluctuation workload, *no+k8shpa* is moderately higher as compared with other approaches with an index of 0.4, whereas *keda+keda* with a least of 0.23. This indicates that 40% of the fluctuation workload was processed up to the mean value of PT. Similarly in steady workload, *no+keda* and *no+rps* with higher indexes of 0.48 and 0.50 respectively. In jump workload, *no+rps* has higher fairness index of 0.39, however, in spikes workload, it has the highest fairness of 0.81.

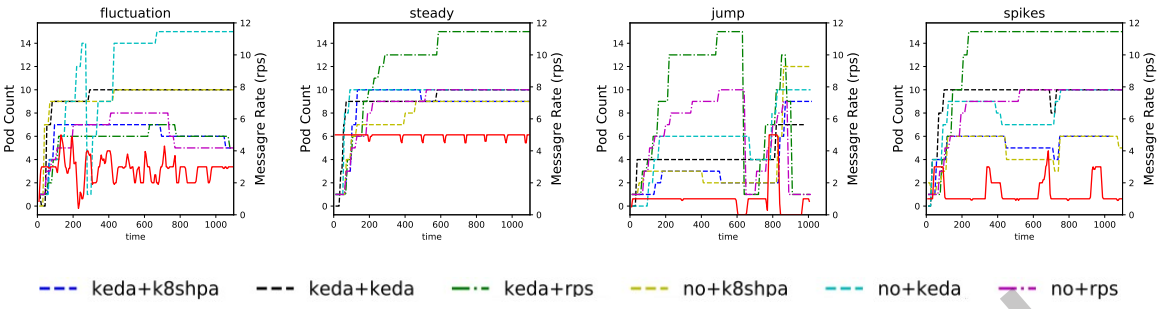
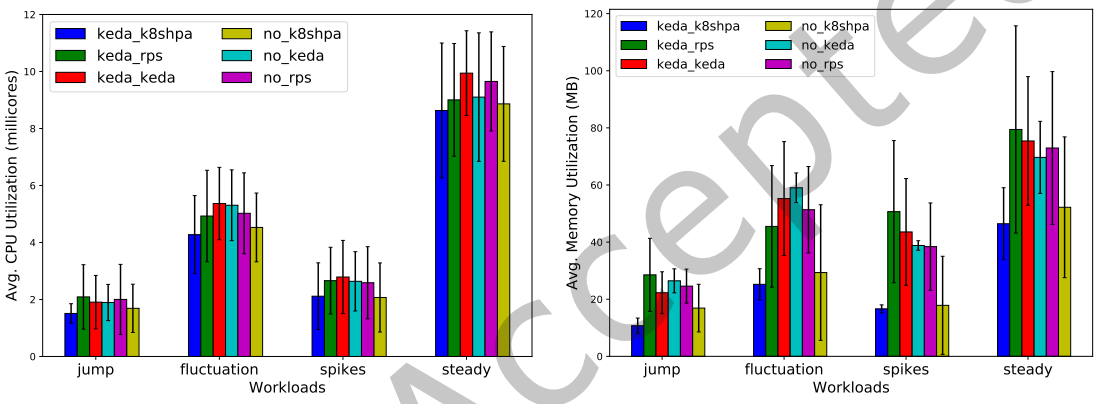


Fig. 18. Scaling pattern of outlierDetection function message arrival rate



(a) CPU utilization (millicore)

(b) Memory utilization (MB)

Fig. 19. Comparison of CPU and Memory utilization of PuhatuMonitoring application for various scaling approaches

Considering the CPU and memory utilization shown in Figure 19, for jump workload, *keda+k8shpa* used minimum average CPU (1.5 millicore) and memory (107.56 MB) as compared with other approaches. Similarly for steady workload, with 8.6 millicore and 464MB respectively. However, in spikes *no+k8shpa* with CPU of 2.07 millicore and memory of 178 MB. In fluctuation workload, *keda+k8shpa* had used minimum CPU and memory. Overall, K8shpa based approaches used minimum CPU and memory in puhatu monitoring application, this is due to the scaling decision of K8s HPA based cpu threshold neither on arrival rate.

Considering the performance metrics described above and other metrics in Table 6. We calculated PT Mean, Max, STD and 90th percentile, also pod count apart from the above described metrics. From the table values, it indicates that, For jump workload, all the approaches aggressively react to the arrival rate, and no approach is optimal to consider, however, based on the success rate and fairness index *no+rps* approach works better compared to other approaches. Further, in steady workload, *no+keda* works satisfactorily as all the metrics values are minimal as compared to other approaches. The primary reason is the KEDA service configured for the serverless functions spawns the replicas only according to message arrival and it tunes to optimal replica count over time. On the other side, KEDA may over provisioning the replica counts and degrade resource utilization.

Table 6. Overview of performance metrics of scaling approaches in PuhatuMonitoring application

Workload	jump						steady						spikes						fluctuation					
Message Queue Trigger	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda	no	keda
Serverless functios	rps	rps	k8shpa	k8shpa	keda	keda	rps	rps	k8shpa	k8shpa	keda	keda	rps	rps	k8shpa	k8shpa	keda	keda	rps	rps	k8shpa	k8shpa	keda	keda
Count	1060	1060	1060	1060	1060	1060	5958	5958	5958	5958	5958	5958	1646	1646	1646	1646	1646	1646	3442	3442	3442	3442	3442	3442
Sucess Rate	**98.9**	97.6	98	96.9	98.3	93.6	96.9	96.7	97.1	96.1	**98**	97.2	**99.9**	99.4	99.8	89.4	99.1	99.1	98.4	95.5	**99**	97.1	97.5	97.2
PT Sum (Sec)	**766.66**	1029	785.13	1098	785.1	1136	5006	4564	5063.3	4951.8	**4646.2**	4756.6	**798.3**	994.4	835.53	900.27	831.6	1074	2240	2785.1	2263.5	2312.4	4470	2244
PT Mean	**0.73**	0.99	0.75	1.07	0.75	1.14	0.87	0.79	0.88	0.86	**0.8**	0.82	**0.49**	0.61	0.51	0.61	0.51	0.66	0.66	0.85	**0.66**	0.69	1.4	0.67
PT Max	**10.44**	18.79	14.68	7.35	14.68	39.18	10.92	17.36	16.63	22.97	14.75	18.33	**2.95**	18.25	3.83	14.99	7.42	25.72	17.04	18.11	**16.75**	19.11	59.81	16.07
PT Min	0.37	0.37	0.36	0.36	0.36	0.37	0.37	0.37	0.37	0.37	**0.37**	0.36	**0.36**	0.37	0.36	0.37	0.36	0.37	0.36	0.36	**0.36**	0.36	0.36	0.36
PT STD	**1.04**	1.97	1.23	1.4	1.23	3.37	0.87	1.03	1.04	1.17	**0.83**	1.04	**0.24**	1.18	0.3	1.06	0.38	1.78	0.87	1.48	**0.81**	1.2	4.98	1.24
PT 90th	1.27	1.42	1.15	3.52	1.15	1.73	1.36	1.15	1.39	1.29	**1.17**	1.17	**0.58**	0.65	0.77	0.82	0.65	0.61	0.95	1.08	1.05	0.86	1.02	0.89
Avg FET	0.75	0.83	0.7	1.02	0.7	0.68	0.8	0.7	0.8	0.78	**0.72**	0.72	**0.45**	0.5	0.47	0.52	0.45	0.48	0.63	0.77	**0.62**	0.62	0.65	0.56
Avg QT	**0.02**	0.17	0.05	0.05	0.05	0.46	0.07	0.09	0.08	0.08	**0.08**	0.1	**0.04**	0.11	0.04	0.1	0.06	0.18	0.03	0.08	**0.05**	0.07	0.68	0.12
JFI	**0.39**	0.2	0.3	0.37	0.27	0.1	0.5	0.37	0.41	0.35	**0.48**	0.38	**0.81**	0.21	0.74	0.25	0.64	0.12	0.37	0.25	**0.4**	0.25	0.3	0.23
Avg CPU	2	2.09	1.89	1.51	1.89	1.91	9.65	9.01	8.86	8.63	9.1	9.94	2.59	2.66	2.07	2.12	2.64	2.79	5.02	4.93	4.53	4.28	5.31	5.37
Avg Memory	246	285	264	107	264	223	729	794	522	464	697	754	384	507	179	166	388	436	513	455	293	252	590	552
Pods Count	24	27	21	9	21	19	35	39	15	15	38	37	35	40	9	9	34	39	36	36	10	10	38	38

Table 7. Suitability analysis using weighted average scoring

	Weights		Jump		Spikes		Steady		Fluctuation	
α	Processing Time (Latency)	Resource Utilization	Aeneas (Compute)	Puhatu (Latency)	Aeneas (Compute)	Puhatu (Latency)	Aeneas (Compute)	Puhatu (Latency)	Aeneas (Compute)	Puhatu (Latency)
0	0%	100%	keda+rps	keda+k8shpa	no+rps	no_k8shpa	no+k8shpa	keda+k8shpa	keda+rps	keda+k8shpa
0.2	20%	80%	keda+k8shpa	no_k8shpa	no+k8shpa	no_k8shpa	keda+k8shpa	no+keda	no+keda	no+rps
0.4	40%	60%	keda+k8shpa	no+keda	no+k8shpa	no+rps	keda+k8shpa	no+keda	no+keda	no+rps
0.6	60%	40%	keda+k8shpa	no+keda	no+k8shpa	no+rps	keda+k8shpa	no+keda	no+keda	no+rps
0.8	80%	20%	keda+k8shpa	no+keda	no+k8shpa	no+rps	keda+k8shpa	no+keda	no+keda	no+rps
1	100%	0%	keda+k8shpa	no+keda	no+k8shpa	no+rps	keda+k8shpa	no+keda	no+keda	no+rps

6.5 Suitability Analysis of Scaling Approaches

In this subsection, an overview and suitability analysis of scaling approaches for two applications is provided. The applications were experimented with using four types of workloads, and the effectiveness of different scaling approaches. To understand the suitability of each workload, we used the weighted average scoring method to rank the scaling approaches. We used two criteria to choose the scaling approach, namely processing time (latency) and resource utilization. Latency is a key primary metric in IoT environments and is directly correlated with the scaling approach. Resource utilization is highly critical in fog computing environments due to resource constraints. Our result analysis in the previous section shows that resource utilization is directly proportional to the scaling approach with the decision of spawning replicas. Further, to decide the suitable *Scaling Approach (SA)* based on the weighted average for two criteria, we used the following equation 4.

$$SA_w = \alpha * Score(PT_w) + (1 - \alpha) * Score(RU_w) \tag{4}$$

where w is a set of workloads $\{jump, steady, spikes, fluctuation\}$ and PT_w is the score obtained by using the weighted average technique on PT metrics on w^{th} workload, similarly RU_w is the score for resource utilization metrics on w^{th}.

So, to calculate the score for each workload, using a weighted average scoring method, weights need to be assigned for selected performance metrics. To choose the critical parameters in our performance metrics from Table 5 and Table 6, we selected six metrics that are highly essential in estimating the processing time (latency) and three metrics that contribute to resource utilization. In addition, we also assign higher weights to the metrics that are important considering the critical requirements of IoT applications similar to the approach used in the article [39][7]. Such essential metrics for PT and their weights are (*Success Rate*, 0.5), (*PT Mean*, 0.1), (*PT 90th*, 0.2), (*FET*, 0.05), (*QT*, 0.05), and (*JEF*, 0.1). The resource utilization metrics and associated weights are (*CPU*, 0.5), (*Memory*, 0.2), and (*Pod Count*, 0.3).

We applied the weighted average scoring method with various values of α on all the scaling approaches with four workloads for two applications and the highest scored scaling approach was chosen for each value of α as shown in Table 7. When, $\alpha = 0$ it indicates that, resource utilization metrics have high priority than processing time and results show that, for jumps and spikes workload, RPS based approaches are efficient for Aeneas application because of less resource utilization, however, k8shpa based approach is better in PuhatuMonitoring.

This is because k8shpa makes the decision to spawn the replicas based on CPU usage and periodically scales the replicas.

Considering the results for different α values, applications, and workloads from Table 7, it indicates that as processing time becomes the highest priority, scaling approaches were consistent with ranks. In the Aeneas application, ***keda+k8shpa*** is highly suitable for Jump and Steady workloads. Similarly, ***no+k8shpa*** can able to handle the spikes in the workload, however, ***no+keda*** is well suited for Fluctuations. This is because KEDA spawns the replicas based on the message arrival rate into the queues and replicas were scaled in hand before the invocations. Since the Aeneas application has compute heavy functions and k8shpa is well-suited to handle such functions. Surprisingly, results show that the KEDA scaler configured for MQT improves the efficiency in Jump and Steady workloads, whereas not beneficial in Spikes and Fluctuations.

For the PuhatuMonitoring application, ***no+keda*** is well suited for Jump and Steady workloads, as mentioned earlier KEDA spawns the replicas instantly based on message arrival in the queues. Comparatively, ***no+rps*** adequately handles the spikes and fluctuations, this was because the PM application was not having the long running time function and easily scales up and down the replicas based on arrival rate. Interestingly, no scaling at MQT was necessary for PM application, however, minimizing the waiting time, and scaling of MQT is essential. Efficient resource consumption and low latency are crucial metrics in IoT applications in fog environments to facilitate real time data processing and delivery. Nevertheless, experimental findings reveal that scaling approaches encounter several bottlenecks that limit the ability to simultaneously optimize processing time and resource consumption.

7 EXPERIENCES AND RECOMMENDATIONS FOR PRACTITIONERS

A series of experiments and corresponding results, we discussed in previous subsections, highlight the considerable variation in the performance of scaling approaches depending on workload patterns and application diversity, such as those featuring long running functions (e.g., Aeneas) or short running functions (e.g., PuhatuMonitoring). Nonetheless, achieving optimal latency and resource consumption is significantly related to appropriate scaling decisions made by the algorithms. The aim of the proposed article is to gain a better understanding of the behavior of such scaling approaches and to offer insights to developers.

During the experiment design phase, significant time was devoted to brainstorming, experimenting, and analyzing scaling configurations for KEDA, K8s HPA, and RPS approaches across both the Message Queue Trigger (MQT) and serverless functions of two SDPs. Determining and fine-tuning the ideal scaling configurations is critical for administrators and developers to minimize costs, latency, and resource consumption. However, this process revealed numerous bottlenecks and challenges.

For instance, in the KEDA-based approach, selecting the target thresholds for scaling metrics such as Queue-Length and Message Rate proved difficult. The optimal values of these metrics are closely tied to the concurrency limits of serverless functions and the execution time of each invocation, both of which significantly influence the success rate. Similarly, in K8s HPA-based approaches, extensive experimentation was required to determine optimal CPU and memory requests and limits for individual functions, ensuring alignment with the desired concurrency limits. Additionally, identifying the appropriate CPU utilization thresholds for each function took considerable time, as incorrect configurations could lead to resource overuse—a significant drawback in fog environments.

In the RPS-based approach, selecting scaling rules posed unique challenges. Developers and administrators needed prior knowledge of workload patterns and user behaviors, as these factors critically influence scaling decisions. The scaling rules also varied for individual functions, depending on their execution characteristics (e.g., long-running versus short-running functions).

Ultimately, the performance of SDPs heavily depends on selecting optimal scaling configurations and settings. However, the complexity of choosing appropriate scaling rules and configurations for intricate applications underscores the need for automated solutions that leverage state-of-the-art techniques [37–39] to streamline this process.

The following key recommendations for practitioners derived from the experiences and outcomes of the experiments:

- **Understand the workload patterns:** IoT operations exhibit diverse arrival patterns, including steady, jumps, fluctuations, and spikes. To achieve desired Quality of Service (QoS), scalability algorithms need to adapt to these arrival patterns. So, recommendation is to consider such workloads and suitable scaling approaches in their design of the IoT system.
- **Know the characteristics of serverless functions:** SDP often consist of multiple functions, each with varying characteristics such as compute intensity, memory intensity, and potential involvement of external I/O operations, which can introduce contention during execution. As per the Universal Scalability Law, scalability becomes nonlinear due to factors like contention and coherence. Therefore, it's crucial to understand the characteristics of these functions for effective scalability management within serverless data pipelines.
- **Optimal scale configurations:** IoT developers have to focus on determining and fine-tuning the ideal configurations of scaling components to minimize costs, latency, and resource consumption.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we described workload and resource based scaling techniques such as KEDA, K8SHPA, and RPS scaling the Message Queue based serverless data pipelines. We applied these approaches to the Aeneas and PuhatuMonitoring IoT application and investigated their performance using the metrics such as processing time, and resource utilization (CPU, Memory) and rigorously analyzed the results by calculating the suitability index and it shows that workload based scaling is useful for faster response times, whereas resource based scaling is useful for consistent throughput and moderate CPU, memory utilization. However, an opportunity exist to test the approaches using long running time, and multiple applications such as compute intensive, bandwidth intensive, and memory intensive IoT applications. We also observed that the CPU, and memory provisioned were not fully utilized and this challenge helps to investigate further to provide optimal solutions using novel scaling algorithms.

Our future work will focus on two key aspects. The first aspect involves selecting resource configurations, concurrency limits, and other function settings in the SDP without incurring additional costs. This can be achieved through the use of statistical and machine learning based optimization techniques. Additionally, approximation of the QueueLength and Message Rate thresholds for scaling the MQT can be improved using these methods. Secondly, none of the current scaling approaches have achieved a 100% success rate or ideal SDP processing time for any single user workload, as shown in Table 2. For example, KEDA and RPS based approaches have scaled more replicas than necessary, potentially leading to higher resource utilization and hindering other applications. These approaches may also not perform optimal scaling for long running functions. While the K8shpa based approach performs better in this regard, it is harder to achieve a 100% success rate. Therefore, optimal scaling decisions for the MQT and serverless functions can be achieved through reactive mechanisms, which can be designed and implemented using state-of-the-art algorithms such as statistical modeling, machine learning, and other optimization techniques. The experimental results provide a road map to model the behaviour of the scaling approaches using well known scalability laws such as Amdahl's Law and Universal Scalability Law.

ACKNOWLEDGMENTS

This work is partially supported by the European Social Fund via IT Academy program. The research is also supported by SERB, India, through grant CRG/2021/003888. We appreciate financial support to UoH-IoE by MHRD, India (F11/9/2019-U3(A)) and Telia, Estonia for financing the hardware setup.

REFERENCES

[1] Hernandez, A., Xiao, B. & Tudor, V. Eraia-enabling intelligence data pipelines for IoT-based application systems. *2020 IEEE International Conference On Pervasive Computing And Communications (PerCom)*. pp. 1-9 (2020)

[2] Raj, A., Bosch, J., Olsson, H. & Wang, T. Modelling data pipelines. *2020 46th Euromicro Conference On Software Engineering And Advanced Applications (SEAA)*. pp. 13-20 (2020)

[3] Chatti, S. Using Spark, Kafka and NIFI for Future Generation of ETL in IT Industry. *Proceedings Of Journal Of Innovation In Information Technology*. (2019)

[4] Ravindra, P., Khochare, A., Reddy, S., Sharma, S., Varshney, P. & Simmhan, Y. Echo: An Adaptive Orchestration Platform for Hybrid Dataflows across Cloud and Edge. *Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13-16, 2017, Proceedings*. pp. 395-410 (2017)

[5] Truong, H. Integrated analytics for IIoT predictive maintenance using IoT big data cloud systems. *2018 IEEE International Conference On Industrial Internet (ICII)*. pp. 109-118 (2018)

[6] Baldini, I., Castro, P., Chang, K. & Others Serverless computing: Current trends and open problems. *Research Advances In Cloud Computing*. pp. 1-20 (2017)

[7] Poojara, S., Dehury, C., Jakovits, P. & Srirama, S. Serverless data pipeline approaches for IoT data in fog and cloud computing. *Future Generation Computer Systems*. **130** pp. 91-105 (2022)

[8] Cheng, B., Fuerst, J., Solmaz, G. & Sanada, T. Fog function: Serverless fog computing for data intensive iot services. *2019 IEEE International Conference On Services Computing (SCC)*. pp. 28-35 (2019)

[9] Poojara, S., Dehury, C., Jakovits, P. & Srirama, S. Serverless Data Pipelines for IoT Data Analytics: A Cloud Vendors Perspective and Solutions. *Predictive Analytics In Cloud, Fog, And Edge Computing: Perspectives And Practices Of Blockchain, IoT, And 5G*. pp. 107-132 (2022)

[10] Tran, M., Elsisi, M., Liu, M., Vu, V., Mahmoud, K., Darwish, M., Abdelaziz, A. & Lehtonen, M. Reliable Deep Learning and IoT-Based Monitoring System for Secure Computer Numerical Control Machines Against Cyber-Attacks With Experimental Verification. *IEEE Access*. **10** pp. 23186-23197 (2022)

[11] Ortiz, G., Boubeta-Puig, J., Criado, J., Corral-Plaza, D., Prado, A., Medina-Bulo, I. & Iribarne, L. A microservice architecture for real-time IoT data processing: A reusable Web of things approach for smart ports. *Computer Standards & Interfaces*. **81** pp. 103604 (2022)

[12] Sai Lohitha, N. & Pounambal, M. Integrated publish/subscribe and push-pull method for cloud based IoT framework for real time data processing. *Measurement: Sensors*. **27** pp. 100699 (2023)

[13] Adhikari, M., Mukherjee, M. & Srirama, S. DPTO: A deadline and priority-aware task offloading in fog computing framework leveraging multilevel feedback queueing. *IEEE Internet Of Things Journal*. **7**, 5773-5782 (2019)

[14] Scheuner, J., Eismann, S., Talluri, S., Van Eyk, E., Abad, C., Leitner, P. & Iosup, A. Let's Trace It: Fine-Grained Serverless Benchmarking using Synchronous and Asynchronous Orchestrated Applications. *ArXiv Preprint ArXiv:2205.07696*. (2022)

[15] Arjona, A., López, P., Sampé, J., Slominski, A. & Villard, L. Triggerflow: Trigger-based orchestration of serverless workflows. *Future Generation Computer Systems*. **124** pp. 215-229 (2021)

[16] Carver, B., Zhang, J., Wang, A. & Cheng, Y. In search of a fast and efficient serverless dag engine. *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. pp. 1-10 (2019)

[17] Srirama, S.N., A Decade of Research in Fog computing: Relevance, Challenges, and Future Directions. *Software: Practice and Experience*. **54**, 3-23 (2024)

[18] Li, X., Kang, P., Molone, J., Wang, W. & Lama, P. KneeScale: Efficient Resource Scaling for Serverless Computing at the Edge. *2022 22nd IEEE International Symposium On Cluster, Cloud And Internet Computing (CCGrid)*. pp. 180-189 (2022)

[19] Jegannathan, A., Saha, R. & Addya, S. A Time Series Forecasting Approach to Minimize Cold Start Time in Cloud-Serverless Platform. *2022 IEEE International Black Sea Conference On Communications And Networking (BlackSeaCom)*. pp. 325-330 (2022)

[20] Benedetti, P., Femminella, M., Reali, G. & Steenhaut, K. Reinforcement Learning Applicability for Resource-Based Auto-scaling in Serverless Edge Applications. *2022 IEEE International Conference On Pervasive Computing And Communications Workshops And Other Affiliated Events (PerCom Workshops)*. pp. 674-679 (2022)

[21] Zafeiropoulos, A., Fotopoulou, E., Filinis, N. & Papavassiliou, S. Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms. *Simulation Modelling Practice And Theory*. **116** pp. 102461 (2022)

[22] Palade, A., Kazmi, A. & Clarke, S. An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge. *2019 IEEE World Congress On Services (SERVICES)*, pp. 206-211 (2019)
[23] Li, J., Kulkarni, S., Ramakrishnan, K. & Li, D. Understanding open source serverless platforms: Design considerations and performance. *Proceedings Of The 5th International Workshop On Serverless Computing*. pp. 3742 (2019)
[24] Zhou, Z., Zhang, C., Ma, L., Gu, J., Qian, H., Wen, Q., Sun, L., Li, P. & Tang, Z. AHPA: Adaptive Horizontal Pod Autoscaling Systems on Alibaba Cloud Container Service for Kubernetes. *ArXiv Preprint ArXiv:2303.03640*. (2023)
[25] Trieu, Q., Javadi, B., Basilakis, J. & Toosi, A. Performance Evaluation of Serverless Edge Computing for Machine Learning Applications. *ArXiv Preprint ArXiv:2210.10331*. (2022)
[26] Gotin, M., Losch, F., Heinrich, R. & Reussner, R. Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments. *Proceedings Of The 2018 ACM/SPEC International Conference On Performance Engineering*. pp. 157-167 (2018)
[27] Mahmoudi, N. & Khazaei, H. Performance Modeling of Metric-Based Serverless Computing Platforms. *IEEE Transactions On Cloud Computing*. pp. 1-13 (2022)
[28] Mampage, A., Karunasekera, S. & Buyya, R. A holistic view on resource management in serverless computing environments: Taxonomy and future directions. *ACM Computing Surveys (CSUR)*. **54**, 1-36 (2022)
[29] Xu, M., Song, C., Ilager, S., Gill, S., Zhao, J., Ye, K. & Xu, C. CoScal: Multi-faceted scaling of microservices with reinforcement learning. *IEEE Transactions On Network And Service Management*. (2022)
[30] Lv, W., Wang, Q., Yang, P., Ding, Y., Yi, B., Wang, Z. & Lin, C. Microservice deployment in edge computing based on deep q learning. *IEEE Transactions On Parallel And Distributed Systems*. **33**, 2968-2978 (2022)
[31] Coulson, N., Sotiriadis, S. & Bessis, N. Adaptive microservice scaling for elastic applications. *IEEE Internet Of Things Journal*. **7**, 4195-4202 (2020)
[32] Schuler, Lucia, Jamil, S. & Kühl, N. AI-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments. *2021 IEEE/ACM 21st International Symposium On Cluster, Cloud And Internet Computing (CCGrid)*. pp. 804-811 (2021)
[33] McChesney, J., Wang, N., Tanwer, A., De Lara, E. & Varghese, B. Defog: fog computing benchmarks. *Proceedings Of The 4th ACM/IEEE Symposium On Edge Computing*. pp. 47-58 (2019)
[34] Jain, R., Chiu, D., Hawe, W. & Others A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*. **21** (1984)
[35] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M. & Bianchini, R. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *ArXiv Preprint ArXiv:2003.03423*. (2020)
[36] Raza, A., Akhtar, N., Isahagian, V., Matta, I. & Huang, L. Configuration and Placement of Serverless Applications using Statistical Learning. *IEEE Transactions On Network And Service Management*. pp. 1-1 (2023)
[37] Akhtar, N., Raza, A., Ishakian, V. & Matta, I. COSE: Configuring Serverless Functions using Statistical Learning. *IEEE INFOCOM 2020 - IEEE Conference On Computer Communications*. pp. 129-138 (2020)
[38] Eismann, S., Bui, L., Grohmann, J., Abad, C., Herbst, N. & Kounev, S. Sizeless: Predicting the Optimal Size of Serverless Functions. *Proceedings Of The 22nd International Middleware Conference*. pp. 248-259 (2021)
[39] AlQerm, I. & Pan, J. DeepEdge: A new QoE-based resource allocation framework using deep reinforcement learning for future heterogeneous edge-IoT applications. *IEEE Transactions On Network And Service Management*. **18**, 3942-3954 (2021)
[40] Choupani, A., Azizi, S. & Aslanpour, M. Joint resource autoscaling and request scheduling for serverless edge computing. *Cluster Computing*. **28**, 171 (2025)
[41] Tari, M., Ghobaei-Arani, M., Pouramini, J. & Ghorbian, M. Auto-scaling mechanisms in serverless computing: A comprehensive review. *Computer Science Review*. **53** pp. 100650 (2024)
[42] Benedetti, P., Femminella, M. & Reali, G. Management of Autoscaling Serverless Functions in Edge Computing Via Q-Learning. *Available At SSRN 5116838*
[43] Wen, L., Xu, M., Gill, S., Hilman, M., Srirama, S., Ye, K. & Xu, C. StatuScale: Status-aware and Elastic Scaling Strategy for Microservice Applications. *ACM Trans. Auton. Adapt. Syst.*. **20** (2025,3), https://doi.org/10.1145/3686253
[44] Ilager, S., Fahringer, J., Tundo, A. & Brandić, I. A Decentralized and Self-Adaptive Approach for Monitoring Volatile Edge Environments. *ACM Trans. Auton. Adapt. Syst.*. (2025,6), https://doi.org/10.1145/3736726, Just Accepted

Received 23 April 2024; revised 24 April 2025; accepted 27 June 2025