DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
55

SHIVANANDA RANGAPPA POOJARA

Design and Orchestration of Scalable, Event-Driven Serverless Data Pipelines for Internet of Things (IoT) Applications

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
55

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
55

SHIVANANDA RANGAPPA POOJARA

Design and Orchestration of Scalable, Event-Driven Serverless Data Pipelines for Internet of Things (IoT) Applications

UNIVERSITY OF TARTU
Press
1632

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in Computer Science on September 3, 2024 by the Council of the Institute of Computer Science, University of Tartu.

Supervisors

PhD	Pelle Jakovits Institute of Computer Science University of Tartu, Estonia
Prof. PhD	Satish Narayana Srirama Institute of Computer Science University of Tartu, Estonia School of Computer and Information Sciences University of Hyderabad, India

Opponents

Prof. PhD	Mohammad Abdullah Al Faruque Electrical Engineering and Computer Science University of California Irvine, USA
Assoc. Prof. PhD	Nicolas Ferry I3S Lab, INRIA Université Côte d'Azur, France

The public defense will take place on October 7, 2024 at 10:15 in Narva Rd. 18-1018.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906 (print) ISSN 2806-2345 (pdf)
ISBN 978-9916-27-655-6 (print) ISBN 978-9916-27-656-3 (pdf)

University of Tartu Press
http://www.tyk.ee/

To my father Late. Shree Rangappa Poojara

ABSTRACT

With the ever-increasing use of Internet of Things (IoT) devices, there has been a massive influx of raw data. Managing such data involves complex tasks, including acquiring data from diverse devices in various formats, performing operations such as filtering and transformation, and executing machine learning operations. Effectively managing the flow and lifecycle of such data presents a significant challenge. To achieve low latency and other Quality of Service (QoS) metrics, edge and fog computing models are increasingly being adopted over cloud-based IoT data processing. This adds complexity to dynamically executing data analysis tasks across varying distances and on heterogeneous hardware devices.

One approach for realizing IoT data processing is using monolithic containerized applications that combine data operations into a single container. These containers can be migrated across the IoT continuum (edge, fog, cloud) to optimize user QoS metrics. Using containers can present challenges and complexities when developing multilayer data-driven IoT applications that require effective data management. Other challenges can arise in ensuring seamless end-to-end connectivity and scaling data operations at a granular level. The other existing solutions, like large data processing clusters (e.g., Apache Flink or Spark) and off-the-shelf tools, can be unreliable due to resource constraints (edge and fog devices) and the event-driven nature of IoT applications.

The hypothesis is that this can be simplified by using serverless computing and data pipelines. In Serverless computing, data analytic tasks can be created as individually scalable virtual functions and executed in an event-driven manner. Data pipelines enable composing individual data processing tasks into a large distributed data flow. By combining both models, Serverless Data Pipelines (SDP) can be created where serverless functions are used as pipeline tasks and seamlessly invoked while the data moves through the pipeline. Serverless functions can easily be deployed in edge, fog, or cloud environments, and data pipeline technologies are used for data transport, routing, and function invocation.

The goal of this thesis is to address critical aspects of data processing within the IoT environments, focusing on the transition from containers to serverless architectures. It first analyses the bottlenecks in traditional monolithic container-based approaches to IoT data processing. It then explores the application of serverless computing in IoT environments as a potential solution to overcome the challenges identified with monolithic architectures. Finally, it assesses the scalability of serverless data processing frameworks in managing stochastic IoT workloads.

This thesis makes three contributions. First is a novel simulator and framework for container orchestration in IoT environments, along with a gradient-based back propagation approach (GOBI and GOBI*) for scheduling, which outperforms existing schedulers. The second contribution comprises three design approaches for SDPs and their suitability analysis for various IoT applications. SDPs based on standard Data Flow Tool (DFT)s are unsuitable for compute-intensive

tasks such as video processing, but they are efficient for bandwidth-intensive applications. Object Service Storage (OSS) based SDPs are better suitable for compute-intensive tasks and MQTT-based SDPs are suitable for latency-sensitive tasks but not for compute and bandwidth-sensitive tasks due to higher CPU and memory utilization. The third contribution is the suitability analysis of reactive autoscaling mechanisms for SDP under four different workload patterns. 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.

Overall, this thesis addresses the complexities and challenges in the processing of IoT data while shifting from monolithic container architectures to serverless computing models for handling IoT data. The contributions assist IoT developers in selecting the most suitable data processing mechanism, considering factors such as computing resources, bandwidth, energy consumption, and latency while meeting sensitive QoS requirements.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Acronyms

ARPANET Advanced Research Projects Agency Network. 28

CERP Cluster of European Research Projects. 28

DARPA Defense Advanced Research Projects Agency. 28
DFT Data Flow Tool. 6

ESB Eneterprise Service Bus. 30

GPU General Processing Units. 16

ICT Internet and Communication Technology. 16
ITU International Telecommunication Union. 30

ML Machine Learning. 16

OSS Object Service Storage. 7

QoS Quality of Service. 6, 16

RFID Radio Frequency Identification. 28

S3 Simple Storage Service. 23
SDP Serverless Data Pipelines. 6, 17

TPU Tensor Processing Units. 16

LIST OF ORIGINAL PUBLICATIONS

Publications included in the thesis

1. S. Tuli, **Shivananda R. Poojara**, S. N. Srirama, G. Casale and N. R. Jennings,"COSCO: Container Orchestration Using Co-Simulation and Gradient Based Optimization for Fog Computing Environments," in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 1, pp. 101-116, 1 Jan. 2022, https://doi.org/10.1109/TPDS.2021.3087349.
2. **Shivananda R. Poojara**, Chinmaya Kumar Dehury, Pelle Jakovits, and Satish Narayana Srirama. "Serverless data pipeline approaches for IoT data in fog and cloud computing." *Future Generation Computer Systems* 130 (2022): 91-105, https://doi.org/10.1016/j.future.2021.12.012.
3. **Shivananda R. Poojara**, A. Jõeleht, P. Jakovits and S. N. Srirama, "Serverless Outlier Management for Environmental IoT Data - A Case Study of PuhatuMonitoring," IEEE 8th World Forum on Internet of Things (WF-IoT), Yokohama, Japan, 2022, pp. 1-7, https://doi.org/10.1109/WF-IoT54382.2022.10152102.
4. **Shivananda R. Poojara**, Dehury, C.K., Jakovits, P., Srirama, S.N. (2023). Serverless Data Pipelines for IoT Data Analytics: A Cloud Vendors Perspective and Solutions. In: Thakkar, H.K., Dehury, C.K., Sahoo, P.K., Veeravalli, B. (eds) Predictive Analytics in Cloud, Fog, and Edge Computing. Springer, Cham, https://doi.org/10.1007/978-3-031-18034-7_7.

Publications not included in the thesis

1. Chinmaya Kumar Dehury, **Shivananda Poojara**, Satish Narayana Srirama, Def-DReL: Towards a sustainable serverless functions deployment strategy for fog-cloud environments using deep reinforcement learning, Applied Soft Computing,Vol 152, 2024, https://doi.org/10.1016/j.asoc.2023.111179.
2. Dharwadkar, Nagaraj V., **Shivananda R. Poojara**, and Anil K. Kannur. "Risk Analysis of Diabetic Patient Using Map-Reduce and Machine Learning Algorithm." In Handbook of Research on Engineering, Business, and Healthcare Applications of Data Science and Analytics, edited by Bhushan Patil and Manisha Vohra, 307-329. Hershey, PA: IGI Global, 2021, https://doi.org/10.4018/978-1-7998-3053-5.ch014.
3. Patil, Manoj A., Khyamling Parane, **Shivananda Poojara**, and Ashwini Patil. "Internet-of-things and mobile application based hybrid model for controlling energy system." International Journal of Information Technology 13, no. 5 (2021): 2129-2138, https://doi.org/10.1007/s41870-021-00667-1.

1. INTRODUCTION

Advancements in Internet and Communication Technology (ICT) have proliferated the widespread deployment of IoT applications in several consumer domains such as smart health care [6], smart city [7], and industrial and transportation applications [8]. It has been predicted 3x rise from 7 billion devices to 25.44 billion connected devices by 2030 [9] . In such applications, thousands to billions of IoT devices generate massive amounts of raw data. Data processing and management in the IoT have complex activities that encompass data acquisition from heterogeneous devices, processing (for e.g, data cleaning, Machine Learning (ML) operations), and storage.

One such example would be, a scenario in a smart factory where video surveillance is utilized to detect incorrect worker poses while operating sensitive machinery. This involves collecting video streams, analyzing frames to identify worker poses and faces, annotating them with names, and issuing alerts to both administration and workers in case of potential harm. Managing the flow and life cycle of video data encompasses various tasks such as collection, routing, filtering, analysis, annotation, alerting, and storage, posing significant challenges. To simplify this process, pipelines are commonly employed to combine individual data processing tasks into cohesive services, allowing for the reuse and composition of common data handling processes into more complex data pipeline services.

Off-the-shelf data pipeline technologies like Apache NiFi, StreamSet, AirFlow, and cloud-based services such as Google DataFlow and AWS Data Pipeline are popular choices for data handling and processing. They enable the creation of advanced drag-and-drop-based workflows. Additionally, Apache Spark, Flink, and Storm are increasingly being used for IoT data processing, leveraging both on-premise servers and cloud resources for scalability [177]. However, using such tools in IoT contexts can have drawbacks, such as the need to create large and costly data processing clusters to manage IoT data. Many IoT applications require real-time actions and event-driven operations, making the configuration of large clusters less suitable and potentially costly. For instance, in video processing applications where IoT data like video streams must be sent to distant clouds for processing, it can lead to bandwidth consumption, increased latency, high dependency on end-to-end connectivity, higher transfer and storage costs, and other typical issues with centralized data collection and potential impacts on QoS.

To address this issue, a novel fog computing architecture was introduced bridging IoT devices and cloud servers. This involved transferring certain data analysis tasks from distant clouds to nearby fog nodes, enhancing real-time service performance. The fog devices advanced enough with substantial computing power, including Tensor Processing Units (TPU) and General Processing Units (GPU)s , along with increased storage capacities. These hardware improvements have simplified the adoption of edge and fog computing, enabling the deployment of compact containers on edge devices. CISCO pioneered fog computing in its network

devices, such as routers and switches, facilitating application hosting through docker containers. For example, CISCO 800 series [1] routers and CISCO Catalyst 9000 switches [2] support application hosting via the CISCO IOS XR platform using docker containers. Additionally, smaller devices such as edge gateways, such as InHand IG502[3] and IG902 IoT gateways, are now capable of running docker containers at the network's edge.

Containers offer a means of encapsulating applications, effectively isolating them from the underlying infrastructure, and enabling consistent deployment across diverse environments (edge, fog, and cloud). In the context of IoT data processing, the use of containers has been shown to be advantageous in various studies [49]–[51], [66]–[68] due to its lightweight and portability. Such features enable IoT application containers to migrate between various heterogeneous devices over the fog and edge layers [100]. In this process, it is essential to ensure that scheduling decisions are suitable to meet user expectations and achieve the desired QoS. However, this approach presents several challenges related to fine-tuning data operations, addressing latency concerns, and optimizing resources within resource-constrained edge and fog environments. For example, in container-based approaches, developers tend to encapsulate all data operations into a monolithic containerized application [49], [83] instead of creating fine-grained services for individual data operations that can be invoked in response to specific events.

Such kind of more advanced fine-grained individual data operation services can be streamlined by using serverless computing. Research studies [11], [23]–[30] conducted to investigate serverless architectures and frameworks for the IoT continuum have shown to be greater flexible in resource allocation, improved latency performance, and can lead to cost savings [13], [14]. Serverless computing, a novel cloud computing service model that leverages function-level billing and scaling, and design event-based, real-time, and scalable IoT data processing has been significantly simplified [13].

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. The serverless cloud model has several benefits including fine-grained auto-scaling and increased productivity gains due to reusable serverless functions deployed on-premise or in the clouds [14], [15]. It is also significantly easier to deploy individual functions in different locations compared to more monolithic applications (e.g. when compared to Apache Spark data analytic applications). SDP are created where serverless functions are used as pipeline tasks and are seamlessly invoked while the data are moved through the pipeline. Serverless functions can be deployed in cloud, edge, or fog environments, and data pipeline technologies are used for data transport, routing,

[1] https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000

[2] https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-9300-series-switches/white-paper-c11-742415.html

[3] https://www.inhand.com/en/products/iot-edge-gateways/

and function invocation. This is where combining a Serverless model and data pipelines can produce significant benefits to avoid some of the disadvantages of a cloud-centric approach and reduce the complexity of designing multilayer (edge, fog, cloud) IoT applications.

Considering the solutions for IoT data processing as described in the above context along with their challenges, the overall hypothesis of the thesis relies on three aspects:

- Despite the advancements offered by serverless data pipelines, we hypothesize that container-based data processing presents unique advantages in certain IoT data processing scenarios that serverless architectures cannot replicate due to its challenges in scheduling and orchestration.
- We hypothesize that serverless functions can be effectively made state-full by integrating specific state management techniques, enhancing their suitability for resource-constrained edge/fog environments in IoT applications.
- We hypothesize that adaptive scaling techniques can dynamically enable serverless data pipelines to adjust to the stochastic behavior of IoT workloads, significantly improving data processing efficiency and reliability.

1.1. Problem Statement and Research Goals

The applications outlined in Figure 1 are focused on event-centric and share some of the common characteristics such as few data operations require more resources (computation, memory), maximum bandwidth, and specific QoS expectations, particularly in terms of low latency and low energy footprints. As discussed earlier, data operations can be routed over edge, fog, and cloud layers, and vice versa, to optimize QoS requirements. However, as shown in Figure 1, the diversity of device architectures and resource capacities decreases as we move towards the cloud layer but increases as we move away from it. This presents challenges in terms of the execution of such data analysis operations across multiple layers.

As discussed in the Introduction, we have two approaches to deal with the data processing of these applications, first approach using containers by packing all the operations into single container application and schedule, and migrate the container applications between hosts in edge/fog to optimize the latency and energy consumption or other QoS parameters. The second approach, focused on event-driven data processing, using dynamic serverless data pipelines for scalable processing. Taking these aspects into account, the primary goals of the thesis are three-fold as described below.

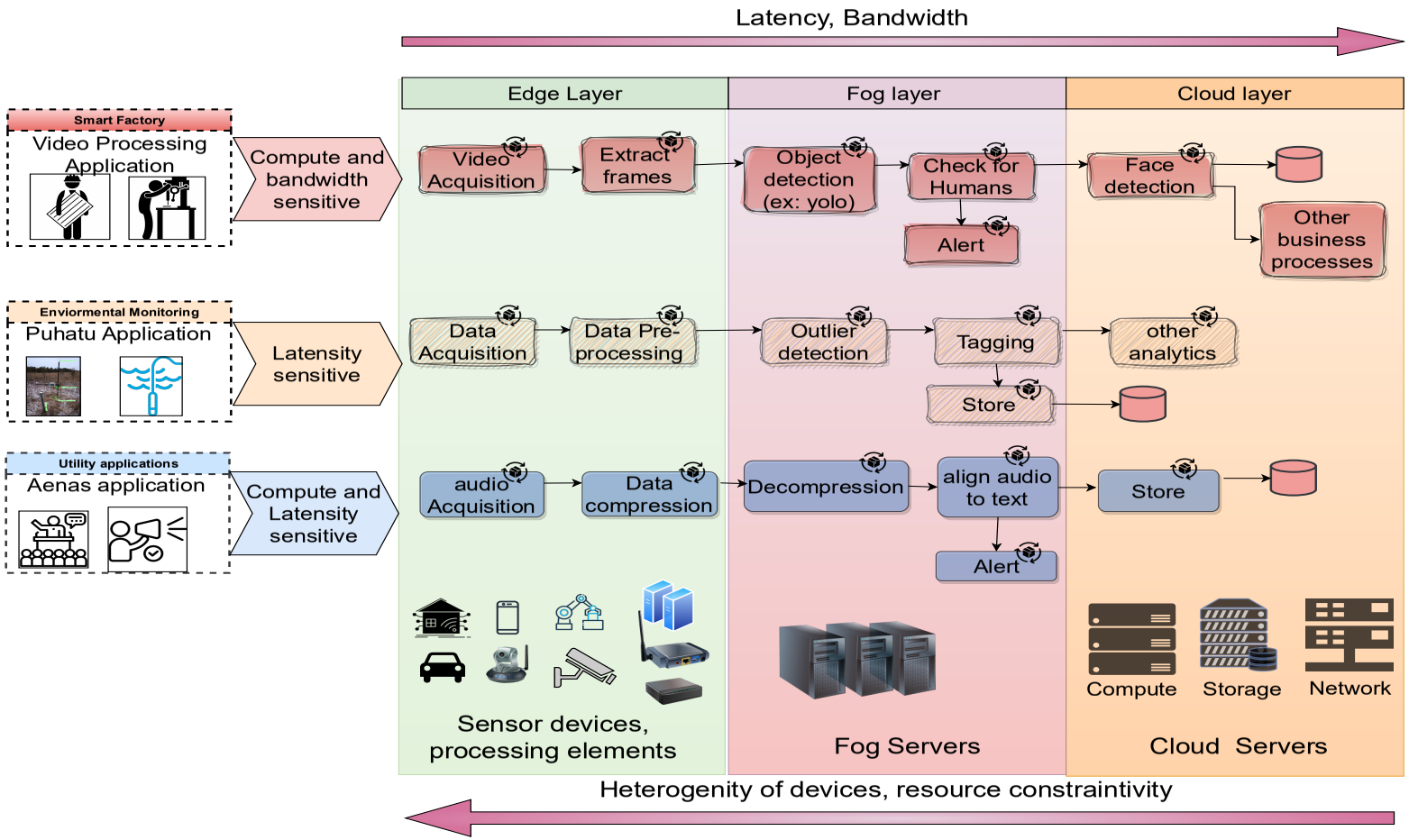


Figure 1. Motivating scenarios

Research Goals (RG)

RG1: To explore the challenges in existing container-based applications in IoT data processing.

RG2: Precise investigation of approaches for building scalable serverless data pipelines and checking their suitability for various IoT workloads.

RG3: Analyze and identify optimal reactive scaling methods for auto-scaling the serverless data pipeline components to effectively process stochastic workloads within the IoT continuum.

Regarding ***RG1***, we explore and outline the challenges in container-based IoT data processing, and the answer to ***RQ1*** will address the solution to RG1. Research studies [66]–[68] indicate that container-based data processing, applications bundled and executed as a monolithic containerized application. Over the years, researchers have extensively investigated how to place tasks or schedule containers in the three layers to meet the desired Quality of Service (QoS) requirements. However, the challenge of reaching low latency and energy consumption is further complicated by modern application workloads that are highly dynamic [69] and heterogeneity in resource capabilities [70]. To address this challenge, container migration and orchestration are the one approach to tune the performance. Several research studies have investigated container migration approaches to improve application execution and reduce response time and other QoS metrics [101], [102]. However, migration moves the entire application container to the scheduled fog node. If the number of migrations is more, then more bandwidth is consumed, which may hinder the performance of the IoT applications. So, the key challenge is to orchestrate all activities that adhere to the scheduling decisions offered by the scheduling algorithm, placing the containers on the appropriate hosts.

To provide quick and energy-efficient solutions, many previous work has focused on developing intelligent policies to schedule tasks (monolithic application containers) on fog hosts [71], [72]. These methods have been dominated by heuristic techniques [73]–[75]. Such approaches have low scheduling times and work well for general cases. For accurate and scalable modeling of the fog environment, there have been many works using local search based on deep learning or learning models with neural networks that approximate an objective function such as energy consumption or response time [69], [71], [76]. As these neural networks approximate objective functions of the optimization problem and share the same problem of high scheduling overhead. Such high scheduling times limit the extent of possible improvement of latency and subsequently SLO violations. This is not suitable for highly volatile environments where host and workload characteristics may suddenly and erratically change. Thus, there is a need for an

approach that not only can adapt quickly in volatile environments, but also has low scheduling overheads to efficiently handle modern workload demands. This problem motivates us to establish the research question as follows:

Research Question 1 (RQ1)

How feasible is the utilization and adoption of container-based monolithic applications for IoT data processing in fog environments without encountering QoS bottlenecks?

Regarding to ***RG2***, we highlight the key challenges by formulating with ***RQ2*** and the answer to the research question will provide a solution to ***RG2***. Serverless functions are stateless with high granular scaling, which introduces additional complexity [21] and challenges in data management between functions that reside in the edge, fog and cloud nodes [11], [168], [169], [176]. Some enterprise solutions such as Azure IoT or AWS Greengrass use serverless edge functions to pre-process and push data to enterprise clouds. Data movement between functions that reside on the edge and in the cloud is often handled by using object storage services such as AWS S3 [153]. However, it is challenging when a large set of functions are deployed in edge/fog infrastructure and data needs to be transferred on each function invocation [11]. The object storage may yield higher charges when more data and more function invocations occur. Although object storage attains the purpose of handling intermediate data, but costs, latency, etc., are challenging. There also exist off-the-shelf DP tools like StreamSet [172] and Apache NiFi [170], [171], which provide some support for edge/fog environments and can also be utilized to solve the issues, but they usually manage the flow of data in a more centralized manner and often require significant computing resources to run effectively.

Alternatively to object storage, its also possible to use message brokers (e.g. Apache Kafka, MQTT) as Message Queues [163], [165], [166] between serverless functions for designing serverless data pipelines. Compared to object-storage, they would require less storage and may be faster due to more extensive memory usage, which is highly desirable in edge/fog environments. However, compared to NiFi, it may be more difficult, for example, creating and designing the pipelines when there are a large number of functions and to control the precise execution flow of pipelines.

Despite the large number of research works published related to serverless IoT data processing [11], [23]–[25], [27], [28], [30], there is still a significant gap in the research literature in terms of creating and designing SDPs. The existing research literature is not convincing enough to choose a suitable approach for specific data and computing-intensive applications. This research gap motivates us to address the following research question.

Research Question 2 (RQ2)

How can Serverless Data Pipelines are created in fog and edge computing environments, that are suitable to specific requirements of diverse IoT applications?

Regarding ***RG3***, we explore scaling challenges of serverless data pipelines, and the answer to ***RQ3*** will address the solution to ***RG3***. In the scenarios discussed above, user workloads can often be stochastic [54] and require a faster response time [52], [53]. Therefore, such workloads need serverless data pipelines with auto-scaling capability to adjust to changing demand. However, the auto-scaling of SDPs in fog environments has to be optimal. Over-provisioning of resources (serverless function or other pipeline components) can lead to higher utilization, which can consume more resources and hinder the performance of other workloads [173]–[175]. Moreover, under-provisioning of resources may degrade the expected QoS of running workloads. We know that SDP has multiple components, for example, serverless functions and data handling and routing components. Now the question is which component to scale based on the workload pattern, for instance, scaling only serverless function will work, or both components should be scaled. Auto-scaling is heavily driven by a set of configurations and rules that act as threshold metrics for making scaling decisions [59], [60], [175]. Moreover, it is challenging for developers to identify such scaling rules on each function or pipeline component deployed in the system.

In addition, the scalability of SDPs can be achieved through available standard auto-scaling methods such as resource metrics and workload-based techniques. However, an investigation is needed to determine which approaches are effective for configuring SDP components under diverse workload patterns. Taking into account of workload behavior and auto scaling challenges, the research question that arises is the following.

Research Question 3 (RQ3)

How can we leverage the existing auto-scaling approaches for scaling the serverless data pipelines that are suitable for diverse and dynamic workload patterns?

1.2. Research Methodology

Considering the challenges and aligned research questions in previous section, we present methodology to answer the research questions.

To address ***RQ1***, we explored existing approaches to orchestrate container-based IoT applications in edge and fog environments. We found that the current state-of-the-art literature has focused on solving scheduling problems using sev-

eral optimization techniques, such as heuristics, evolutionary methods, and reinforcement learning. However, these approaches need a longer decision time to schedule. Furthermore, the migration of the container application was one of the optimal approaches used to reduce energy consumption and latency, using the scheduling decision offered by the optimization algorithms. To study the drawbacks of these approaches, we designed a simple Python-based discrete event simulation tool that uses the concept of container migration and scheduling for edge and fog computing. In addition, we designed a framework (real test bed environment) that is used to deploy and orchestrate the scheduling decisions offered by the scheduling algorithms in the simulator on real edge and fog devices. The docker container engine and its Checkpoint/Restore In User-space (CRIU) framework were used for container migration. We created docker images of real-time IoT applications such as the Aeneas, PockeSphinx, and Yolo applications described in the article [111]. We investigated six state-of-the-art scheduling approaches to measure energy consumption, number of migrations, latency, and scheduling decision time.

Furthermore, we collected the data set by running these schedulers and random schedulers. Using these data, we devised a neural network-based gradient-based back-propagation scheme. In addition, we extend the orchestration approach by using co-simulation to obtain an optimal and faster scheduling decision. However, our research investigation shows that a novel approach is efficient enough to schedule container applications but has several drawbacks. Here, the whole container is migrated instead of routing the user request along with the data. Further container migration consumes more bandwidth and is not focused on scalability aspects and event-driven architecture.

Taking into account the same IoT applications, we designed more advanced data processing technologies using serverless data pipelines that support the event-driven nature of IoT applications. To address the ***RQ2***, we have shown how SDP can be deployed in the three-layer IoT architecture. Aligned to the architecture, we proposed three approaches for designing Serverless Data Pipelines with three different data handling mechanisms such as Apache Nifi, MQTT Message Queue, and MinIO object storage service like AWS Simple Storage Service (S3).

Furthermore, we designed serverless data pipeline approaches using those three data handling mechanisms for three real-time fog computing workloads such as Aeneas, Pocket-Sphinx, and a custom video processing application. We selected applications with heterogeneous characteristics. For example, video processing using Yolo requires extensive computational power, Aeneas prioritizes minimal latency, and PocketSphinx is sensitive to bandwidth usage. We deployed SDPs on real-time edge and fog cluster consisting of 5 Raspberry Pi cluster and OpenStack clouds. In addition, we measure performance metrics, such as processing time and resource utilization of these different SDP approaches. From the performance metrics, we draw up suitability analysis that helps the IoT developers to choose the SDP approach to different types of fog computing workloads.

To address ***RQ3***, we first outline existing reactive scaling approaches such as workload-based and resource utilization-based along with their scale configurations and architecture. In addition, we show how such reactive approaches are configured on serverless data pipeline components. For this, we used six different scaling approaches that can be configured on SDP components. To validate their performance, we used two real-time fog computing workloads, the Aeneas and PuhatuMoniroting applications. We 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. Furthermore, we offer insights into the suitability of scaling approaches, experience, and challenges encountered during the implementation and evaluation of the scalability of serverless data pipelines in various configurations.

In summary, the thesis contributes to the field of IoT data processing in three key areas. Firstly, we developed a Python-based tool, integrated within container orchestration, aimed at aiding researchers in demonstrating scheduling algorithms for container orchestration. Through this tool, we conducted an in-depth analysis of container application scheduling and migration, uncovering both benefits and challenges. Secondly, we devised three serverless data pipeline approaches, showcasing their advantages and offering a suitability analysis for different applications. This examination serves as a valuable resource for IoT developers seeking guidance on selecting the most suitable data processing approach using serverless computing. Thirdly, we conducted an analysis on scaling serverless data pipelines utilizing existing auto-scaling mechanisms. Additionally, we conducted suitability assessments for four workload patterns against six scaling approaches. This comprehensive analysis provides invaluable insights for IoT developers in configuring scaling settings effectively.

1.3. Contributions

This thesis establishes three contributions that help IoT developers choose the appropriate data processing mechanism.

Contribution 1: *Container applications for IoT data processing*. We designed the COSCO framework for container application scheduling and orchestration that utilizes the migration approach. We implemented six state-of-the-art scheduling algorithms (Heuristic, Evolutionary approaches, Max-Weight-based approaches, Reinforcement Learning (RL) models) and two new algorithms (neural network-based approach), GOBI (Gradient-based back-propagation to input) and GOBI* (Uses co-simulation), which use back-propagation to input and are validated against the baseline algorithms. We evaluated algorithms against performance metrics such as the number of migrations, scheduling time, energy consumption, and response time. The experimental results show a significant improvement in terms of energy consumption, response time, service level objective, and scheduling

time by up to 15, 40, 4 and 82%, respectively, compared to the state-of-the-art algorithms. However, our experimental experience indicates that migration of the entire data processing application induces more bandwidth as the number of migrations increases.

Contribution 2: *Design approaches for Serverless Data Pipelines*. We designed three serverless data pipeline approaches (based on the standard data flow tool (DFT), based on object storage service (OSS), and based on MQTT), further implemented using three real-time fog applications. Furthermore, we evaluated performance using metrics such as end-to-end data processing processing processing time, resource utilization (CPU, memory, read/write network) in edge, fog, and cloud environments. Our results with suitability analysis indicate that DFT was not suitable for compute-intensive applications such as video or image processing, while OSS was better suited for this task. However, DFT was efficient for bandwidth-intensive applications due to the minimal use of network resources. On the other hand, MQTT-based SDP was observed with increase in CPU and Memory usage as the number of users rose, and experienced a drop in data units in the pipeline for PocketSphinx and custom video processing applications. However it performed well for Aeneas which had low size data units.

Contribution 3: *Auto-scaling of Serverless Data Pipelines* . To increase the efficiency of serverless data pipelines in terms of QoS metrics such as processing time and resource utilization, we used the auto-scaling mechanism. We investigated the scaling approaches that can adopt stochastic behavior of IoT workloads. Further, we evaluated the scaling approaches on real-time fog environments, using QoS metrics such as processing time and resource utilization. Our analysis of suitability, using the weighted average scoring method on two QoS metrics, revealed that for the Aeneas application, a combination of workload-based (QueueLength) scaling at Message Queue Trigger and resource-based (CPU) scaling on serverless functions was effective in dealing with Jump and Steady workloads, while only resource-based (CPU) scaling on serverless functions was sufficient to handle spikes and fluctuations. For the PuhatuMonitoring application, workload-based (QueueLength) scaling on serverless functions was effective for Jump and Steady workloads, while workload-based (RPS) scaling adequately managed spikes and fluctuations.

1.4. Thesis Structure

Sections 1.1 to 1.3 provided the context for the thesis, outlined the research goals, presented our research methodology, and described the contributions of the thesis. Overall structure of the thesis is organized as shown in Figure 2 and Table 1.

Chapter 2 offers a comprehensive background on essential concepts, including the Internet of Things (IoT) and its layered architecture, various data processing

Table 1. Overview of research goals and corresponding contributions along with publications

Research Goal	Research Question	Contributions
RG1: To explore the challenges in existing container-based applications in IoT data processing.	**RQ1:** How feasible is the utilization and adoption of container-based monolithic applications for IoT data processing in fog environments without encountering QoS bottlenecks?	**#1** - Container applications for IoT data processing **Publications:** - (2020) TPDS [1]
RG2: Precise investigation of approaches for building scalable serverless data pipelines and checking their suitability for various IoT workloads.	**RQ2:** How can Serverless Data Pipelines be created in Fog and Edge computing environments that are suitable to specific requirements of diverse IoT applications?	**#2** - Design of serverless data pipelines **Publications:** - (2021) FGCS [2] - (2022) Springer Book Chapter [3]
RG3: Analyze and identify optimal reactive scaling methods for auto-scaling the serverless data pipeline components to effectively process stochastic workloads within the IoT continuum.	**RQ3:** How can we leverage the existing auto-scaling approaches for scaling the serverless data pipelines that are suitable for diverse and dynamic workload patterns?	**#3** - Auto-scaling of serverless data pipelines **Publications:** - (2024) submitted to ACM TAAS [5] - (2022) WFIoT Conference [4]

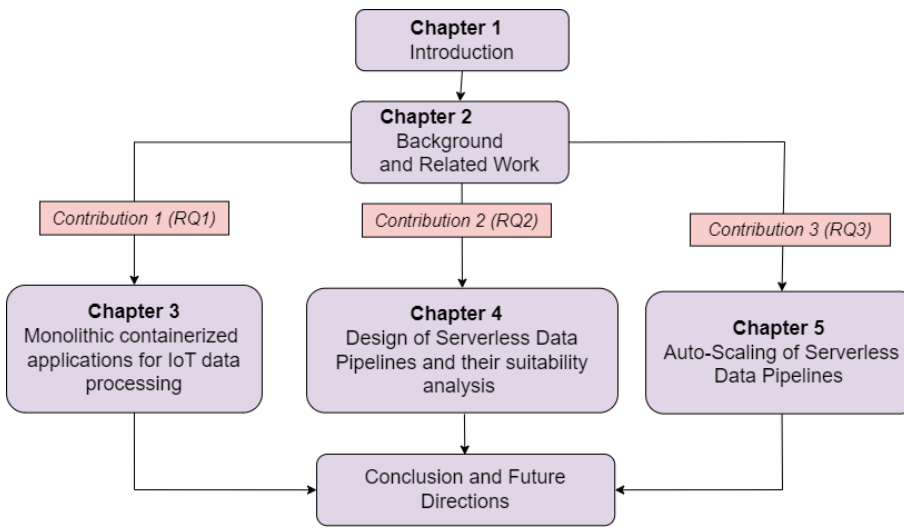


Figure 2. Structure of the thesis

methods, containers, serverless computing, and data pipeline mechanisms. In addition, this chapter provides a detailed overview of the real-time IoT applications featured in this thesis.

Chapter 3, tackles the ***RQ1*** of ***RG1***. This chapter investigates monolithic container-based data processing and presents our proposed container orchestration algorithm. This work was extracted from work [1] and published in *IEEE Transactions on Parallel and Distributed Systems*.

Chapter 4 tackles on addressing ***RQ2*** of ***RG2*** and is based on the work [2], [3]. This work was published in *Future Generation Computer Systems* and the Springer book chapter.

Chapter 5 addresses ***RQ3*** of ***RG3*** and is based on the works [4], [5]. This work [5] will be submitted for review. Article [4] presented and published in the proceedings of *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)* .

Chapter 6 serves as the conclusion of the thesis, summarizing key findings, and outlines potential directions for future research.

2. BACKGROUND

This chapter provides a comprehensive overview of key concepts and terminology used throughout the thesis, as well as a detailed explanation of the real-time applications used to illustrate the research objectives. Section 2.1 outlines the essential components of the three-tier IoT system, while Section 2.2 offers information on the architecture of IoT computing, including edge, fog, and cloud computing. Furthermore, Section 2.3 provides an in-depth look at cloud-centric data processing, along with its associated challenges, while also highlighting the advantages conferred by edge- and fog-centric data processing architectures.

Section 2.4 outlines containers and serverless computing as integral elements of data processing, accompanied by a comparative evaluation of container-based solutions versus serverless computing along with their respective merits and drawbacks. Furthermore, Section 2.5 dives into the architecture of serverless computing and its implementation in the design of serverless data pipelines for IoT data processing. Finally, in Section 2.6, we describe four real-time IoT applications used to support the demonstration of the research objectives outlined in this thesis.

2.1. Internet of Things

The "Internet" is a significant and revolutionary term in the current digital world. Way back in 1962, where research was started by Defense Advanced Research Projects Agency (DARPA) and later named Advanced Research Projects Agency Network (ARPANET) in 1969. The Internet became a significant component of the IoT, and history goes back to the 1980s when the first Internet-connected Coca-Cola machine used at Carnegie Mellon University, this IoT system was used to check to see if there was a drink available and if it was cold. The keyword *Internet of Things* was originally coined in 1999 by Kevin Ashton, who was MIT's Executive Director of Auto-ID Labs. He believed that Radio Frequency Identification (RFID) was a prerequisite for the Internet of Things, where all devices were tagged with RFID and computers could manage, track and inventory them.

A commonly known definition of IoT from the Cluster of European Research Projects (CERP) on the Internet of Things states that *"The Internet of Things allows people and things to be connected anytime, anywhere, with anything and anyone, ideally using any path / network and any service"*. This signifies two keywords, "Internet" and "Things", where Things encompass a more generic set of physical entities such as smart devices sensors and any other object that is aware of its context and can communicate with other entities, making it accessible at any time, anywhere over the Internet.

Generally, "things" refer to a physical object that can sense, process, and actuate, which is controlled and managed over the Internet. For example, controlling the heating system in the apartment based on external weather conditions and

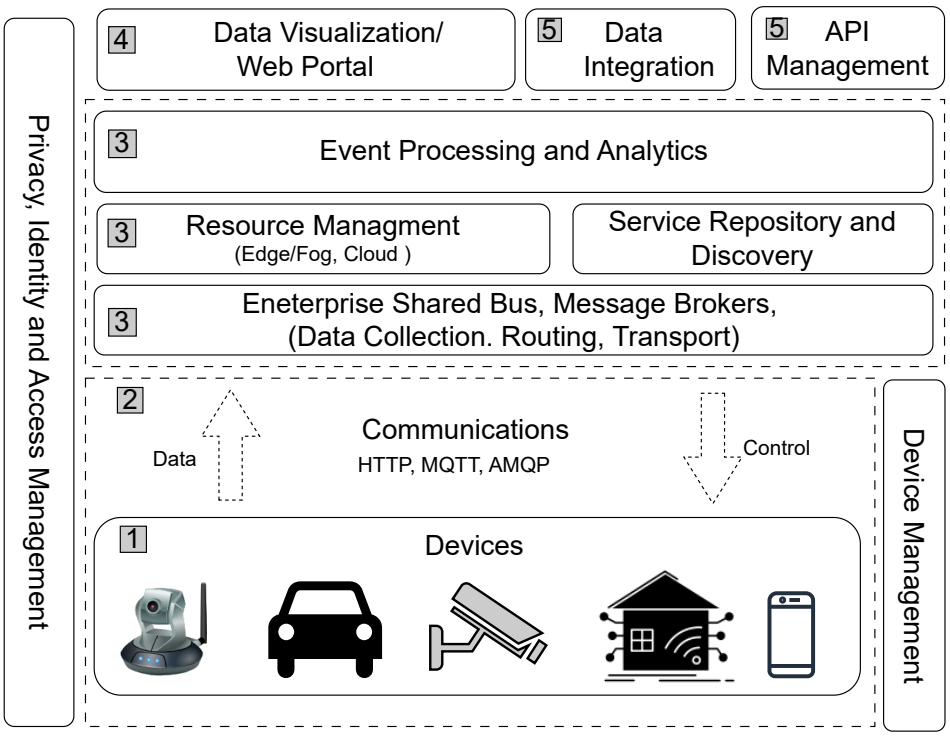


Figure 3. IoT Reference Architecture

human activities. Currently, such applications fall into several domains such as Smart City, Industrial, and Healthcare applications. Rapid innovations and advancements in Telecommunication technologies like 5G and other allied areas enabled significant transformation in the ubiquitous use of IoT applications achieving the "Internet of Everything" (IoE). The key term "IoE" is used by CISCO to describe things, places, and people that expose and consume services to each other entities, enabling machine-to-machine, human-to-machine and human-with-environment interactions.

The building blocks of an IoT system include several entities such as sensory embedded devices, communication networks, and application services. Over the years, IoT research has taken considerable interest in academia and research communities to realize the IoT concepts to reality, in essence, to provide a picture as a unified network of smart objects, people, and processes that are capable of universally and ubiquitously communicating with each other.

Several organizations such as the International Telecommunication Union (ITU), CISCO, IEEE, and the European Research Cluster on the Internet of Things devised open standard methodologies and architectures defining the components or service layers in the IoT systems. The consortium of industry parameters led by CISCO presented an IoT reference architecture, which focuses on a seven-layer architecture. Research teams in academia have suggested similar reference architectures. The benefit of these reference architectures and models is that they provide a broad overview of the entire system, hiding the particular limitations and implementation details. This gives them an advantage over other architectures.

Figure 3 shows the IoT reference architecture, which is a modified version of the additional services and presentation layers. The number box in each service and presentation layer indicates the mapping of the components from CISCO's 5-layer architecture. The service layer includes data processing and analytics, resource management and resource discovery, additional message brokers, and an Eneterprise Service Bus (ESB) built on top of the communication and physical layers. The presentation layers focus on data integration and API management for sharing the data, and visualization using custom dashboards on web portals. In addition, device and data security over all layers is provided by the privacy, identity, and access management layers. Device management is the main service that provides the configuration and management of IoT devices on physical objects.

Layer 1 comprises sensor devices, including video cameras, vehicles equipped with sensor arrays, and smart home devices. Sensors like those for humidity, temperature, and CO2 can be connected to microcontroller units (MCUs), system-on-chip (SoC) devices, or ARM-based devices such as Raspberry Pis. These sensor-equipped devices can communicate over various network channels such as Wi-Fi, BLE/Bluetooth, Zigbee, LoRaWAN (Long Range Wide Area Network), or NB-IoT (Narrowband IoT). IoT devices typically operate on limited power sources, such as batteries or energy harvesting. Therefore, low power consumption is critical. For instance, sensors and microcontrollers used in IoT devices are designed

to operate efficiently on minimal power. IoT devices often handle small amounts of data but may need to process or transmit this data frequently. Throughput requirements can vary; for example, video surveillance IoT devices need higher throughput compared to environmental sensors.

Further, these devices have the ability to perceive information and also transmit it through communication channels that support lightweight communication protocols such as MQTT, HTTP, and AMQP, which are part of Layer 2. This layer serves as the communication interface for transmitting and receiving data and control signals between the upper layer and the devices in Layer 1.

Layer 3 encompasses various elements that handle the collection, storage, data processing, routing, and supervision of relevant resources. Within this layer, message brokers or event channels play a pivotal role in collecting, routing, and transporting data through designated channels. Additionally, resource management includes tasks such as scheduling, scaling, and overseeing resources in continuous IoT clusters such as edge, fog, and cloud, all coordinated by the resource management component. Service discovery and recovery mechanisms automate the connection of data processing systems and data storage. In addition, the responsibility of processing IoT data, extracting insights, and executing actions based on specific business logic to interact with the remote IoT environment falls under the purview of the data processing and algorithmic control component. Layer 3 is of paramount importance within the context of this proposed thesis. In the following section, we will discuss further the compute continuum and the associated data processing approaches.

2.2. Compute Continuum in IoT

We extend the description of the layer 3 resource management component, layer 2 and layer 1 from Figure 3 in the context of the compute continuum. Typically, the compute continuum is dispersed into three layers, cloud, fog, and edge computing [77]–[79]. The computing continuum brings cloud data centers close to data sources located at the edge of the network by creating micro data centers located in various geographical locations with an increase in heterogeneity of devices that pose challenges in application deployment and data management [80]. Subsequently, our proposed thesis addresses the three research questions (RQ1, RQ2, and RQ3) in IoT data processing to achieve the desired QoS (such as low latency, optimal resource utilization) by leveraging the compute continuum.

2.2.1. Cloud Computing

Cloud computing offers the ability to seamlessly rent compute, storage, and network services through a pay-as-you-go model. It is supported by a virtual pool of limitless resources and clusters that exhibit scalability and elasticity. These resources are leased to users based on negotiated QoS metrics specified in Service

Level Agreements. Typically, cloud data centers with high-end servers with several thousands of CPUs, terabytes of memory and storage leased dynamically and reliably to multiple users by sharing the underlying infrastructure.

In the context of IoT and data processing, IoT developers can configure expensive distributed data processing clusters for scalable operations such as Apache Spark, Flink, or Storm. Additionally, the use of preconfigured data pipeline services, such as Google Data Flow and AWS Data Pipeline, to design end-to-end data migration and processing pipelines for IoT data.

2.2.2. Fog Computing

The first definition of fog computing proposed in 2012 by CISCO was "Virtualized platform that delivers the compute, storage, and networking service between end devices and traditional cloud data centers" [77]. It states that fog acts as an intermediate layer between the cloud and the terminal sensor devices. However, in a more diversified way, researchers [81], [82] defined- "Fog computing as a scenario where a large number of ubiquitous and ubiquitous heterogeneous (wireless and sometimes autonomous) devices communicate and potentially cooperate among them and with the network to perform storage and processing tasks). It means that fog computing ecosystem is made up of a large number of fog nodes such as high-end routers, gateways, switches, mobile base stations, and custom fog servers. These devices have compute, storage, and network capabilities like cloud data centers, but at nanoscale. They can act as nano-data centers between the cloud and terminal layer. These devices are located on streets, restaurants, mobile base stations, or other suitable areas. They can be static at a fixed location or mobile on a moving carrier. The end devices can be conveniently connected to the fog nodes to obtain services. They have the capability to compute, transmit, and temporarily store the received sensed data. Real-time analysis and latency-sensitive applications can be performed in the fog layer.

Fog servers are equipped with the ability to host container runtimes, such as Docker and Containerd, as part of the fundamental infrastructure. This also drives to create large-scale Kubernetes clusters by grouping the multiple fog servers. This provides IoT developers with the ability to package applications, execute them across multiple fog nodes, and scale them. Furthermore, Serverless Runtime Environments such as OpenFaaS or Fission essentially hosted on the K8s or container run times which provides the facility to create functions for data operations and scale the functions based on the IoT data arrival rate.

2.2.3. Edge Computing

Edge computing encompassed a set of end point devices such as sensing elements, surveillance cameras, high-end sensing devices, mobile phones with connected sensors, smart vehicles, health behavior measurement devices, and other smart devices. Edge devices usually with computation constraint devices can sense the

data and actuate based upon the control signal generated upon the data processing carried out at higher compute elements. They normally sense the data of physical objects and send them to the upper layers for processing. In general, these devices are geographically distributed in nature. Edge devices connected to near by routers or switches or IoT controller nodes, which can perform the simple data analytic operations such as filtering, error correction, or consolidation or other such operations.

2.3. Data processing in IoT

Taking into account Figure 3, the layer 3 component focuses mainly on event processing and data analytics, specifically tailored for IoT data processing. To align with Quality of Service (QoS) expectations and fulfill user Service Level Objectives, there are two key strategies: data can be transferred to the cloud or processed in close proximity to data sources within edge and fog environments. Moreover, the research literature highlights exploring two distinct approaches: a cloud-centric approach and an edge/fog-centric approach.

2.3.1. Cloud-centric approach

In the cloud-centric approach, all data must be transported to one central data center and processed using scalable data processing platforms (such as Apache Spark, Flint, or serverless data processing pipelines). Several solutions [114] exist for cloud-centered IoT platforms from different CSPs such as ThingsSpeak[1], Cayenne[2], etc, but such platforms are fully focused on collecting data from IoT devices and performing complete data processing and controlling device operations from centralized clouds. However, using only cloud-centric IoT platforms has multiple disadvantages, such as high end-to-end connectivity dependency, higher latency, higher transfer and storage costs, and other typical issues with centralized data collection. Its challenging due to that current IoT applications are latency agnostic and demand for huge bandwidth to upload live data streams to clouds for processing (for example, using thermal video cameras for predictive maintenance of heavy electrical machines or detecting safety position of workers using video cameras).

2.3.2. Edge/Fog-centric approach

Recently, edge / fog-centric data processing has gained considerable interest in industry and academia [43], [115]–[117] due to the need of immediate data processing solutions that take advantage of current advances in the Internet and telecommunication industry. The primary focus in this type of architecture is to perform preliminary operations near the source(in gateways, on-premise servers, or near by

[1] https://thingspeak.com/

[2] https://developers.mydevices.com/cayenne/features/

geographically distinct located servers), such as video compression or data aggregation, and further transported to cloud systems. State-of-the-art research on edge / fog computing-based data processing has been heavily investigated to solve interesting problems such as scheduling and placement of user tasks (data operations) based on user QoS expectations [118]–[120], tuning the infrastructure so as to reduce the energy consumption and maximum utilization of the fog resources [69], [121], [122], data management and orchestration in data analytic pipelines [123]–[125], further choosing the best approach for data processing when it comes to stream, batch or function-based processing [117], [126].

Previous work diversified with the use of monolithic containerized applications and serverless functions as part of data processing architecture. Some works demonstrates the advantages of using containerized applications, where fog clusters are configured with container engines that involve many fog nodes. The data operations are bundled in to container instance which is created in the fog cluster along with the data. Furthermore, the container migration approach was used in previous work [106] to tune the efficient use of fog resources and QoS metrics. The serverless edge/fog architecture provides better advantages over monolithic containerized applications in terms of scaling, agility, and function scheduling. Cheng et al. [21] proposed a serverless data processing architecture for fog-based data processing. The serverless engine is configured on fog nodes and data orchestration is enabled through data centric programming model called Fog Function. Similarly, Cassel et al. have implemented the serverless architecture for IoT data processing and show potential benefits.

We use both architectural approaches, container-based and serverless-based edge/fog data processing. In addressing RQ1 within Contribution 1, we leverage a monolithic container-based fog system in conjunction with a migration strategy to effectively schedule containers on the fog nodes, optimizing resource utilization, and ensuring quality of service metrics. Additionally, for RQ2 and RQ3, we adopt a serverless architecture for data processing, orchestrating the scaling of data operations by designing scalable event-driven serverless data pipelines.

2.4. Containers

In the past, most applications were configured and deployed on bare metal servers. This was the responsibility of the developer, who had to develop, configure, and manage the application's deployment on the server. This on-premise deployment of the application posed challenges such as underutilization of servers, scalability, and portability. Recent developments in virtualization have enabled faster deployment and configuration of applications through the creation of virtual machines and storage. However, virtual machines are complex in their architecture, which means that the images of the operating system are larger and take longer to boot. This results in longer application deployment times, which can range from a few minutes to several hours.

Over the years, the design and development of software applications have changed rapidly, focusing on agility and the delivery of software to meet changing business needs. Such a transformation evolved recommending ease of deployment and focus on the business logic rather than infrastructure management. This evolution has seen a progression from the use of bare-metal servers to virtual machines and further toward containerization.

Containers became the mainstream in the application configuration and deployment. In container-based applications, developers need to pack the application code, libraries, and system libraries by creating an application image to deploy it on the infrastructure. Containers are efficient self-contained packages of software that include the application code, its necessary libraries, and its dependencies. They can be run on a variety of platforms, including desktops, laptops, traditional systems, and cloud environments. Containers make use of operating system virtualization, taking advantage of the Linux Namespaces and Control Groups (cgroups) features of the OS kernel. Furthermore, containerization brings numerous advantages (such as fast time to market, manage scalability and portability, and quick deployment) in transitioning applications from a monolithic architecture to a light-weight container-based architecture, compared to traditional and virtual machine-based deployment methods.

2.4.1. Container Engines

Container engines are the technology used to create, manage, and deploy container applications and images. Container architecture is based on kernel features that utilize the cgroups and namespaces of the host operating system to create isolated containers by sharing host resources. Container engines are a ready-made solution for handling Linux features and offer a straightforward way to manage the container life cycle. *Docker* is a widely used tool for container runtime, and other popular tools like *Containerd* and *Podman* are some of the others.

As we move from the cloud to the fog and edge layers, the heterogeneity of hardware devices increases. For instance, ARM, x86, and x64 architecture devices, as well as ESP33 and PyCom boards, are all embedded devices. Container engines offer developers the opportunity to build container applications that are compatible with multiple architectures and can run on edge and fog devices with minimal effort.

Developers must package their application code, libraries, and system libraries into an application image to deploy it on the infrastructure for container-based applications. Additionally, they must also pay attention to scaling the application, organizing the network, and managing storage. This eventually combersome task for the developers, hence container orchestration tools provide feature to create multi node cluster and manage the large cluster up to a few hundred hosts. Additionally, Container orchestration simplifies the job of multi-tenant service scaling, configuration management, deployment, and security. The community has devel-

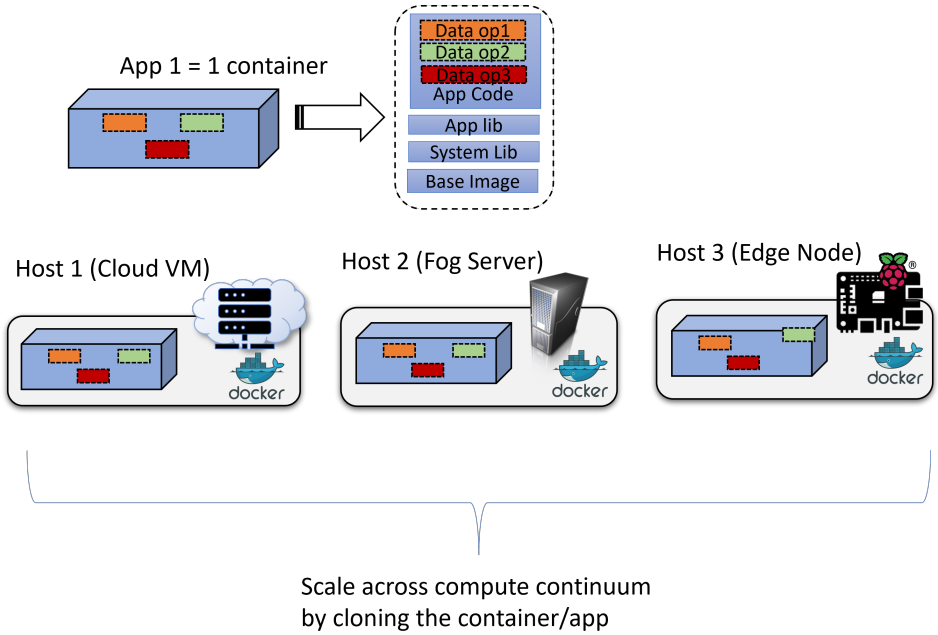


Figure 4. Containerized Monolithic application

oped several container orchestration tools and platforms; some of them are Docker Swarm, Kubernetes, and others.

Kubernetes have multiple versions and one of such is K3S which is light weight container orchestration specially for resource constraint edge devices. Using K3S like engine, its easy to create and manage the multi-node cluster using edge nodes and fog servers and run container-based applications across compute continuum.

2.4.2. Container Applications

Containerized applications are composed of the entire application itself with all the modules and functions that are expected, or they may not be internally monolithic but instead be structured as multiple components, layers, or a sequence of operations. However, externally, it is a single container, a single process, a single single service, or containing a set of operations.

A monolithic application has most of its functionality contained within a single process/container, which is composed of internal layers or libraries. In this context, a single container is deployed for each request, and based on the demand, the copies of the containers are added based on the load balancing of the host machines. The overall monolithic containerized application is shown in Figure 4. The colored blocks symbolize the operations or groups of functions that the container performs. Developers create a monolithic application and build a container image that is compatible with multiple architectures. Subsequently, containers are

generated across the compute continuum in accordance with the user demand and Quality of Service (QoS) expectations.

2.4.3. Container Migration

Container migration is a strategy used to move running containers from an existing container host to another container host. Container migration used by container engines during node failures or as container delegation due to resource pressure or other system failures. Container engines support migration functionality using the CRIU[3] (Checkpoint and Restore in User Space) utility. Docker offers a migration approach using the CRIU utility and provides docker native commands via CLI for example *docker checkpoint* for check pointing the running container and *docker start –checkpoint* to start the container from where its check pointed. This approach enbales the container to migrate from one docker host to another docker host. Previous research work[41], [102], [128] used this approach to migrate containers from overloaded fog nodes to under-loaded fog nodes without compromising user QoS metrics. In contribution 1, we utilize the container migration approach to tune the energy consumption and latency optimization of user workloads with SLAs.

2.5. Serverless Computing and Data Pipelines

2.5.1. Serverless Computing

Serverless had emerged as a new model of cloud computing to deploy and scale applications at the functional level. This provides an opportunity for developers to focus on business logic instead of focusing on scaling, configuring and managing [87], [88]. Developers are required to just submit a function written in a high-level language and the serverless systems will take care of everything else such as instance selection, auto-scaling, deployment, fault tolerance, etc. Serverless is also known as FaaS (Function as a Service), where a provider executes a piece of code (normally called a function) by dynamically allocating the resources. The code is typically run by provisioning the stateless containers and terminates after execution. Developers or end users are billed only for the execution of functions, unlike in the virtual machine billing model. This billing as opposed to traditional systems, where there is a need to pay for idle CPU and memory. Hence, serverless is advantageous and essential to run event-driven applications.

Event-driven systems consist of many short-running tasks expected to run in any sequence or in any combination. Serverless functions could be triggered by events such as data operations, http requests, querying services, update events in business process, and other events such as monitoring services. The serverless platform itself can be represented in the concept of an event processing system

[3]https://criu.org/

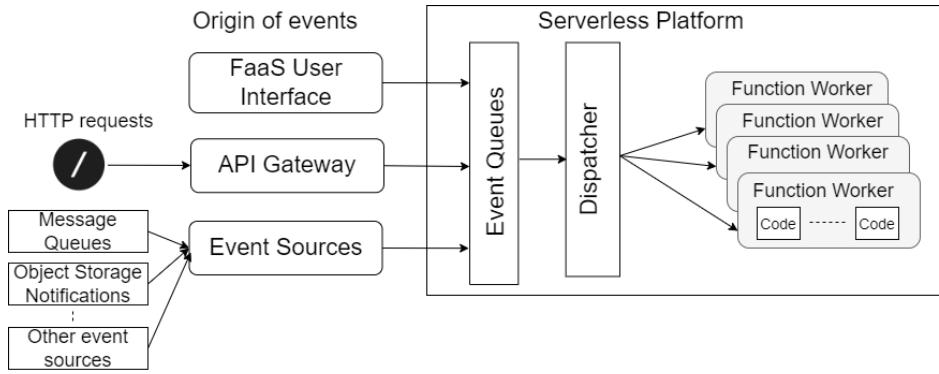


Figure 5. Serverless as an event processing system [84]

[13], [87]. The general architecture is represented in Figure 5. The events are associated with user defined functions. The event source triggers the event by sending web service http requests, and associated functions are executed by master by spawning workers with associated resource requirements based upon workloads. Events are managed by queues for entry and exit of requests.

The driving force is the benefits of serverless architecture, such as process agility and less operational cost, faster setup, and easier operational management. The evolution of serverless proliferated into two main serverless paradigms known as serverless runtime and serverless databases [84]. In this thesis, we only focus on serverless run-times. The first serverless platform powered by AWS released in 2014 is known as the Lambda service [95] and is motivated by the open Lambda project. In 2016, the serverless became part of the marketplace by the cloud giants Microsoft Azure functions [94], Google cloud functions [93], and IBM OpenWhisk [96]. Apart from the proprietary cloud providers, open source efforts lead to the birth of projects such as OpenWhisk [92], OpenFaaS [91] and so on. The following section provides an overview of several serverless platforms available in the marketplace.

Commercial Serverless Platforms:. Public cloud serverless frameworks (FaaS) enable developers to build, deploy functions, and applications using fully managed end-to-end serverless platform. Several cloud serverless platforms are available in the market, and we limit ourselves to platforms that have a large user base such as AWS Lambda, IBM Cloud functions, and Google Cloud Functions.

1. *AWS Lambda*: AWS Lambda [95] is a serverless, event-driven compute service provided by AWS. Moreover, developers can set up triggers for more than 200 software as a service applications (SaaS), including file processing, stream processing, web applications, IoT back-ends, and mobile back-ends. The run-times supported are Java, Go, PowerShell, Node.js, C#, Python, and Ruby code. The cost of using the Lambda service measured in invocation requests to function and memory duration (GB-Sec). However,

one million invocations are free per month and $0.20 per one million after exceeding the free limit. Taking into account memory limits, 400,000 GB-Sec of compute time per month and after that <$0.000017 per 1 GB-Sec.

2. *Microsoft Azure Functions*: This serverless platform was started by Microsoft Azure in 2016. Azure Functions is a serverless compute service that lets you run event-triggered code without having to explicitly provision or manage the infrastructure. It supports C#, JavaScript, F#, Java, PowerShell and Python run times. The price is calculated on the consumption and executions per second of the resource. Similarly like AWS, Azure functions have similar free limits, for example, one million invocations a month. After the free quota, $0.20 per million. For memory, Azure Functions have a limit of 400K GB-Sec per month. After this $0.000016 per GB-Sec.
3. *Google Cloud Functions [96]*: It is provided by Google Cloud Services. It is an event-driven compute platform that allows users to easily connect and extend Google and third-party cloud services and build applications that scale from zero to the planet scale.
4. *IBM Cloud functions [95]*: IBM offers this service based on the architecture of the Apache OpenWhisk open source framework, which IBM manages. This service finds applications in various use cases, serving as serverless back-ends implemented as micro-services with clearly defined endpoints. It is utilized for mobile backend, cognitive data processing, and data processing through the composition of serverless functions, forming serverless chains, commonly referred to as serverless data pipelines.

Opensource Serverless Platforms:. The following are the most popular open source serverless frameworks with maturity in development to a large extent.

1. OpenFaaS: Alex Ellis created OpenFaaS, which has grown into a collaborative project with a growing number of contributors. OpenFaaS is a framework for developing serverless functions using Docker containers. It allows developers to quickly create functions that contain their business logic without having to write a lot of extra code. These functions can be triggered by a variety of external events.
2. Apache OpenWhisk: Apache OpenWhisk is an open-source Serverless Computing platform that enables functions to be executed in response to incoming events at any scale. This architecture is based on Docker containers, which provide the infrastructure, servers, and scalability needed. Developers have a wide range of programming languages to choose from, such as Go, Java, .NET, PHP, Python, Ruby, Rust, Scala, and Swift. Functions can be implemented with the desired runtime and executed on OpenWhisk using Docker images. OpenWhisk can be deployed on both local and cloud infrastructures, with support for the Kubernetes, OpenShift, and Docker Compose frameworks.

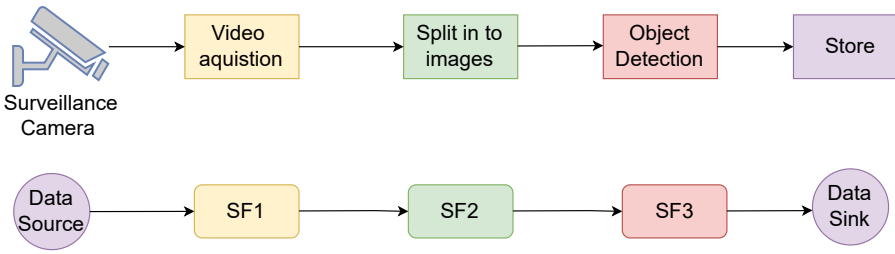


Figure 6. Example of Serverless data processing for IoT application

2.5.2. Serverless Data Pipelines

IoT data processing typically consists of several stages like raw data collection from IoT sensors, pre-processing, applying machine learning model for outlier detection, tagging outliers and storing the results. Such a sequence of stages is modeled in a pipeline manner with data processed at each stage and provided as input to next stages. *Data pipelines* (DP) are chains of workflows with a series of data processing components [97]. A DP tool/engine is a system with built-in components that captures, organizes/manipulates, and routes data to one or more required locations.

Modern DP tools are available as open source (Apache NiFi [98], StreamSet, Apache Airflow) or as commercial products (Google Data Flow). Leveraging such data pipeline engines in IoT data processing has several benefits, such as accountability and guaranteed delivery; however, such clusters often need more resources and are not efficient in resource-constraint and event-driven IoT systems.

A simple example of a serverless data pipeline is shown in Figure 6. Consider a use-case in smart factory, where video surveillance cameras are used to detect the false pose of factory worker while operating a sensitive device/machine; here, video streams are collected, split into frames (images) and detect the pose of the worker, finally alerting the administration and the worker about false pose that yield harm to person or to the machine. Each module depends on the previous module for data input, which resembles a data pipeline. The video data generated from video camera (data source) is processed in a sequence of modules and finally stored in a storage (data sink). Here, moving entire video streams for cloud-oriented data processing yields major disadvantages in bandwidth issues to upload video, latency, and cost of processing. So, essentially, part of data processing pipeline executed near the data source and further on to the cloud systems or entire pipeline deployed near the data source on the factory floor. However, due to advancements, cloud services can bring near to the device and then streamline from edge to cloud data processing.

Its challenging to configure compute and memory hungry data pipeline platforms in the factory floor. Therefore, due to the light weight and easy synchronization of serverless functions from cloud to edge and vice versa by cloud providers,

the data processing is efficient. Aligning to this, entire use-case modules are decomposed to individual serverless functions, where each function output is fed to another serverless function for processing data from the source sink by constructing pipelines, which are termed Serverless Data Pipelines. In Figure 6, SF1, SF2, and SF3 are serverless functions (SF) formed as data pipelines. Due to the non-stateless nature of serverless functions, intermediate storage units are used for a complete end-to-end data pipeline. Once SF is designed, it can reuse and deploy on platform-independent and heterogeneous hardware such as X86 or ARM systems. This drastically reduces the time to market and developers need not worry.

2.6. Real Time IoT applications

In this thesis, we realized the use of containers and SDPs using four real-time IoT applications that include video processing (yolo), Aeneas, Pockesphinx and Puhatu environmental monitoring applications. The first three applications belong to the set of standard fog computing workloads or applications that are considered from the article [85]. Applications are classified into latency critical (LC), bandwidth intensive (BI), location aware (LA), and computationally intensive (CI). The Puhatu monitoring application is an application with small data units and is a latency critical application, as considered in our article [4]. These applications are redesigned as a series of data flow pipelines that can spread across edge, fog, and cloud infrastructure for the implementation of SDPs. In Chapter 3, we designed these applications (excluding PuhatuMonitoring) as monolithic containers by packing all the data operations into a single container image. The detailed implementation of data flow pipelines is described in chapters 4 and 5.

2.6.1. Video processing application

The You Only Look Once (YOLO) makes predictions with a single network evaluation, unlike systems like Region-Based Convolutional Neural Networks (R-CNN), which require thousands of networks for a single image. Hence, its predictions are 1000x faster than R-CNN and 100x faster than Fast R-CNN. This fog application is the ideal candidate to consider because of bandwidth sensitiveness and requires high network bandwidth to send video stream from edge node to cloud and aiming for faster processing with immediate response. Most of the operations are compute-intensive and demand for more CPU and Memory resources. So, the application is designed to process the preliminary detection of objects in fog and send them to the cloud for further analysis. To understand precisely, we use a scenario of real-time object detection from the article [86]. The abstract flow of the data in this use case is shown in Figure 7. Here a drone is used to capture the video footage and to process this, sequences of operations are carried out as follows:

1. Drone captures video footage and forwards it to edge gateway using a communication protocol such as MQTT or HTTP or other protocols.

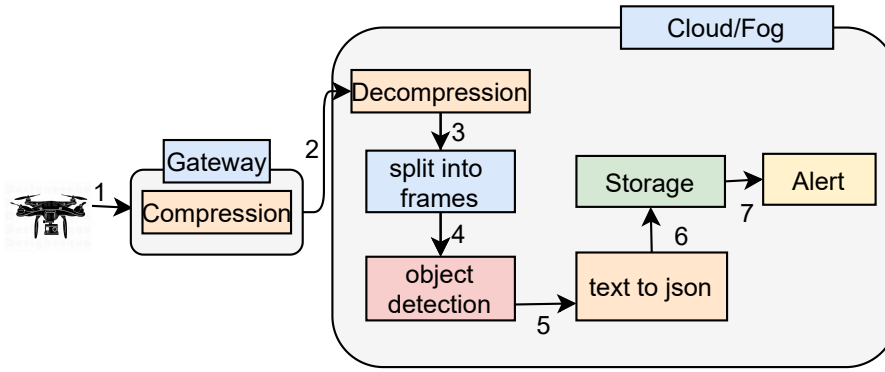


Figure 7. Abstract view of video processing data pipeline

2. Edge gateway compresses the video and forwards to fog nodes.
3. In a fog node, the video is decompressed using a set of tools (gzip or zip).
4. The video will be split into a number of frames that can be processed individually.
5. The frames are passed to YOLOv3 framework and objects are identified from respective frame.
6. The raw output generated from YOLOv3 is processed to json document. This will nicely arrange the raw text into json elements consisting of identified objects.
7. The json documents are stored in storage service for further analysis.

This application is used as a real-time workload in Chapters 3 and 4. The set of activities described above is encapsulated in a single monolithic container in Chapter 3. However, in Chapter 4, each task is an individual service and is designed as a data pipeline. To design the above example in traditional computing platforms, the developer needs to specify the necessary input, configuration, and run-time environment to perform the required data operation seamlessly by provisioning resources on the fly. However, it adds an issue of over-provisioning, rather than demand. For example, short-running tasks like triggering an alert message to the end-user when a human/animal object is identified don't require heavy computation. Thankfully, the serverless platform can subsidize this issue by invoking functions whenever events are triggered by consuming less computation resources.

In this application, the entire video processing application is decoupled into a set of OpenFaaS-based serverless functions as shown in Table 2. Among these, two major functions are (a) *Split* and (b) *Yolo* as described in the following.

- *Split*: Splits the video into multiple frames using ffmpeg, an image/video editing tool. The number of splits depends on the value given to the fps argument in ffmpeg command
- *Yolo*: Yolo is an object detection framework that has a darknet library.

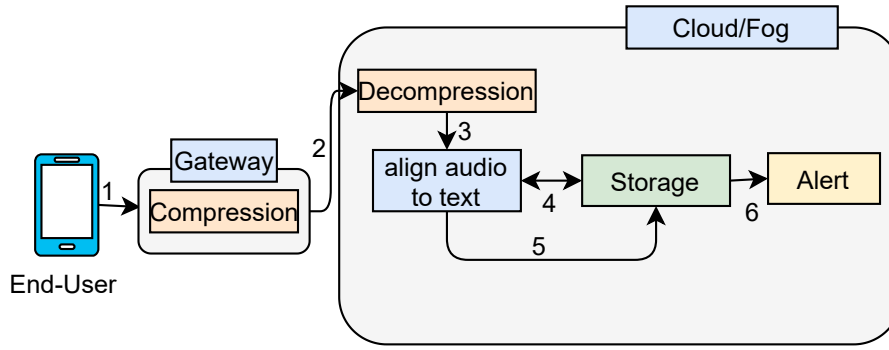


Figure 8. Abstract view of Aeneas data pipeline

The rest of the functions mentioned in Table 2 are used as subsidiary functions in the pipeline. This video processing application is implemented in all three SDP approaches, as described in Section 4.3.

Table 2. Number of serverless functions used

SDP/Application	Aeneas	Pocketsphinx	Video proc.
DFT based SDP	2	3	3
OSS based SDP	5	6	6
MQTT based SDP	2	3	3

2.6.2. Aeneas: A text-audio synchronization

The Aeneas tool is specialized for 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.

This fog application is the ideal candidate to consider because of bandwidth-intensive and consumes huge network bandwidth to stream audio files from a large set of end-user devices to the edge node and cloud node. End users aim for faster response times, and to achieve this, few of the operations are performed in fog nodes. We show the scenario of an Aeneas-based real-time application and its abstract flow in Figure 8.

Here, end-user devices, such as mobile devices or other devices, are used to stream.wav or.mp3 audio files. Then the following operations are carried out as follows:

1. End user offloads a .wav file to edge gateway using a mobile device.
2. Edge gateway compresses the audio and forwards to fog nodes.
3. In a fog node, audio is decompressed using a set of tools (gzip or zip).
4. The file (.xhtml) is downloaded from the cloud or other repository that is used as input to the Aeneas tool. The file contains text that is used for alignment in the audio file.

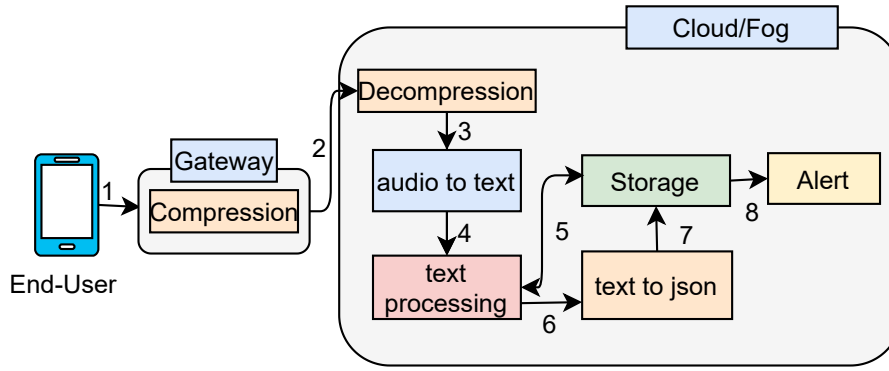


Figure 9. Pocketsphinx application data pipeline

5. The raw .mp3 or .wav is processed using the Aeneas tool along with the given file (.xhtml). The Aeneas tool has the facility to generate the alignment output in json.
6. The json documents are stored in a storage service for further analysis.

In Chapter 3, we use the Aeneas application composed of a monolithic container with all the sets of activities described above. However, in Chapter 4, the set of activities composed into SDP by designing the individual activity in to service. The above traditional data pipeline is redesigned to SDP by composing these operations into serverless functions. It has one major function:

- *aeneas*: Aeneas is python library for forced alignment of given text in a audio file and is configured with python3 run time. This function accepts two input files text file (.txt, .xhtml), audio file (.wav,.mp3) and generates the aligned output (.json, .mile). It generates .json as a HTTP response to the invocation.

2.6.3. PocketSphinx: A Speech-to-text conversion

It is a software engine specialized for speaker-independent continuous speech recognition [85]. An audio file (.wav) is converted to a defined language in text form using a pre-trained acoustic model to determine the source and destination language for speech-to-text conversion. The sample audio files are taken from the large-scale speech repository [4].

In this fog application, we consider a scenario where end user submits a.wav file via mobile phone and it needs to be processed (either fog/cloud) to find a given text in the audio file. This application is bandwidth and compute intensive and requires higher network bandwidth to offload the audio files to the cloud. Hence, it is feasible to process near the data source in the fog nodes to achieve higher response times. Figure 9 shows the abstract view of the application with the following set of operations:

[4]http://www.repository.voxforge1.org/downloads/SpeechCorpus

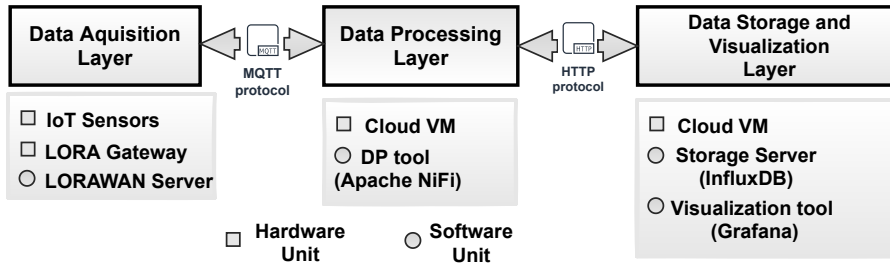


Figure 10. Abstract view of Puhatu-Monitoring IoT system

1. End user offloads a .wav file to edge gateway node
2. It will be compressed and forwarded to fog/cloud infrastructure
3. Decompress the audio file
4. Process an audio file to text format using PocketSphinx tool
5. Get the text to search in output produced from PocketSphinx operation
6. Perform text search operation
7. Convert into proper JSON document
8. Store into storage service for further analysis or usage and send corresponding alerts.

In Chapter 3, we use the PocketSphinx application composed of a monolithic container with all the sets of activities described above. However, in Chapter 4, the set of activities composed into SDP by designing the individual activity in to service. The general flow of the PocketSphinx application mentioned above is designed to a set of serverless functions as shown in Table 2. It has one major function:

- *pocketsphinx*: Pocketsphinx is a Python library for speech-to-text conversion using a predefined acoustic model. This function accepts one audio file (.wav,.mp3) and generates the aligned output (.json). It generates .json as a HTTP response to the invocation.

In Chapter 3 Section 3.2.2 and Section 3.5.2 will describe the workload model that represents the task considered to be a monolithic container of such real-time applications along with the data. In Chapter 4, we represent $F = \{f_1, f_2, \ldots, f_m\}$ as a set of serverless functions for each application, for example, Aeneas for DFT based SDP has $m = 2$. In Section 4.3 of Chapter 4, we describe the specific implementation of SDP approaches and three associated use cases that are implemented according to the design of the proposed SDPs.

2.6.4. Puhatu Monitoring Application

The wetland environmental monitoring takes place in the northern part of the Puhatu Nature Protection Area (NPA), NW Estonia, next to the open-pit oil-shale quarry. Actual mining occurs mostly 1 km north of the NPA boundary, but underground parts of the mine, groundwater pumping and draining channels are only

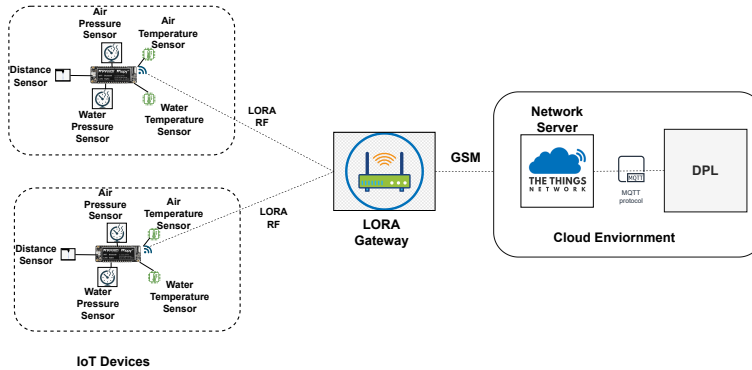


Figure 11. Data Acquisition Layer

a couple of hundred meters from the protection area. The aim of the monitoring network is to observe if groundwater lowering affects the surface water hydrological regime in the wetland. Two main parameters are monitored. First, fluctuations in water level with respect to the ground surface occur in shallow wells that are penetrated by peat. Second, changes in the ground surface altitude of peat must be measured relative to the mineral soil below the peat because the highly porous water-saturated peat itself can compact due to the lowering of groundwater in the long term. There are 15 monitoring sites, grouped into five profiles that are roughly perpendicular to the Puhatu NPA / quarry boundary.

An abstract view of Puhatu monitoring is shown in Figure 10. This includes three layers in the system and is responsible for collecting and storing the sensory data from the monitoring system to cloud storage. It consists of three layers: Data Acquisition Layer (DAL), Data Processing Layer (DPL), and Data Storage and Visualization Layer (DSVL). The DAL is a primary layer that collects raw streaming data (payload) from a set of environmental monitoring sensors and decodes the payload to extract the sensor data and metadata. Furthermore, the DPL is the intermediate layer responsible for collecting the data from the DAL and passing through the data pipeline to convert the data formats and extract the required features to store in a time-series database. Finally, the DSVL is responsible for storing and generating visualization charts to show the data stream measurements over the timestamps. It also has the alert mechanism to inform an organization of a major change in the readings of the data stream. The detailed description of individual layers is given in the following subsections.

Data Acquisition Layer. The data acquisition layer consists of hardware components such as IoT devices connected with in situ sensors, the LORA gateway, and the network server. The monitoring area is very remote and LTE does not extend to all sites. One of the monitoring IoT devices is shown in Figure 12. IoT devices are attached to mineral soil-anchored steel tubes that also serve as poles for the LORA antenna. The devices are powered by a LiPo battery that is charged using a $2W$ solar panel.

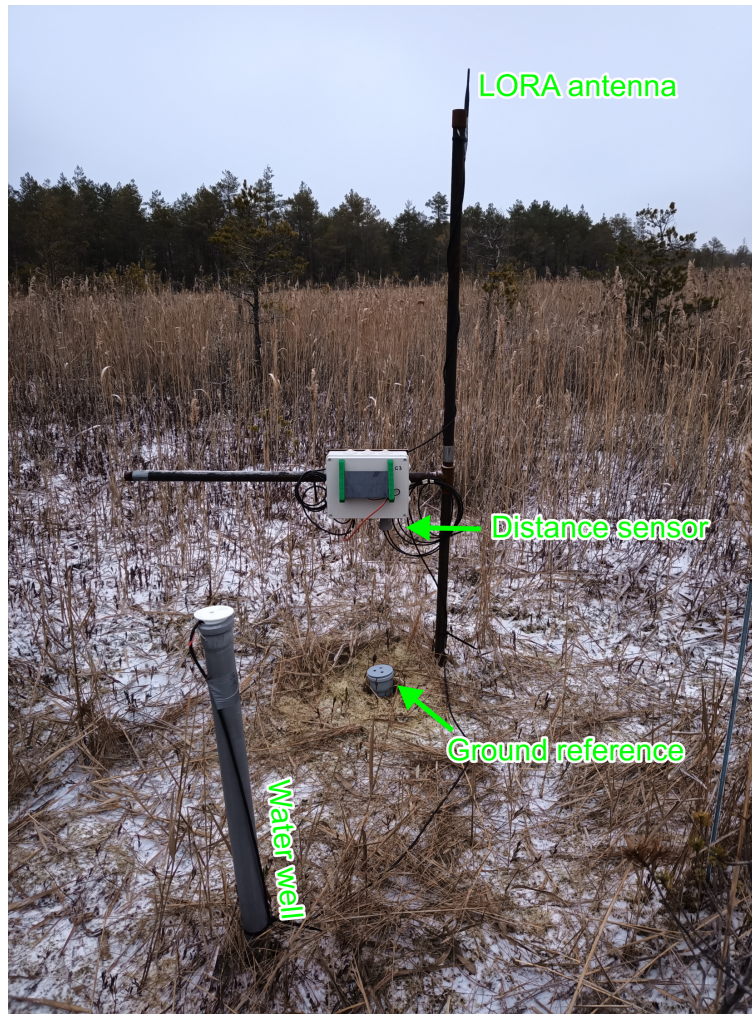


Figure 12. Puhatu IoT device in winter

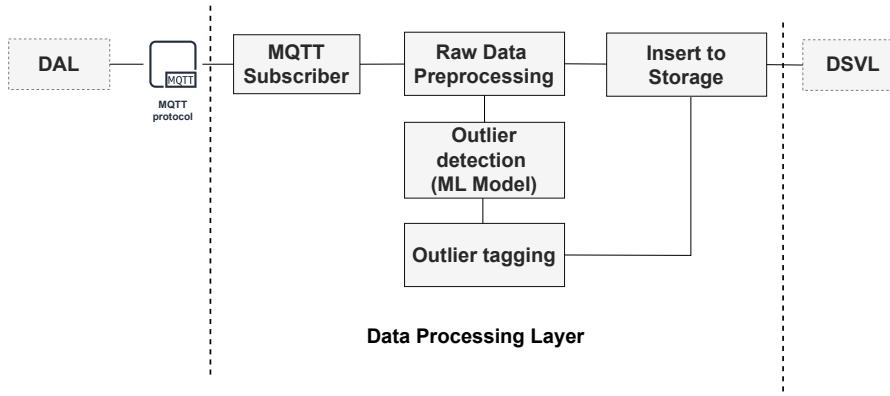


Figure 13. Data Processing Layer

IoT devices have an MS5803-07 pressure sensor to measure absolute pressure and temperature in the water well, while the same type of sensor or MS6805 pressure sensor is in the device box to measure air pressure and temperature. The water level in the well is calculated as the pressure difference between these two sensors, expressed as meters of H2O. The IoT device is controlled by Pycom FiPy on an expansion board. It has an ESP32 chip that is programmable with micropython. It supports five networks, including LoRa.

The elevation changes on the ground surface were measured with ultrasonic distance sensors (MAXBOTIX MB7360) that were located at the base of the device box. Since the steel tubes and the device box are anchored to the mineral soil, the change in distance indicates the rise or lowering of the peat surface. Initially, measurements were made on natural ground, but moss and grass form a rough surface from which ultrasound reflections are weak and distances vary. Therefore, the short plastic tube was inserted into the peat soil so that its top surface extended 20 cm above the ground. Having a reference above the actual ground surface also helps to get true ground surface fluctuations because in rainy periods the area may be flooded.

Data Processing Layer. The aim of the DPL is to pre-process the raw data, detect the outliers, tag the data and make it ready to store in the storage server in the DSVL layer as shown in Figure 13. The DPL consists of software services such as Apache NiFi, a serverless runtime engine, and a message queue broker (MQTT) based on the type of SDP design.

Figure 13 illustrates the working of the Data processing Layer. The Network Server (TheThingsNetwork Server) receives the raw payload from the IoT device and decodes it and then publishes it to various endpoints such as MQTT or HTTP APIs. The decoded payload is subscribed from the MQTT broker. The MQTT subscriber is connected to the broker over a TLS encryption channel. Here, the MQTT subscriber subscribes to the topic */up*, which means that data will be received from all active devices in the Network Server. The data consumed are in json format and passed to the next task to pre-process the data. The raw data (json)

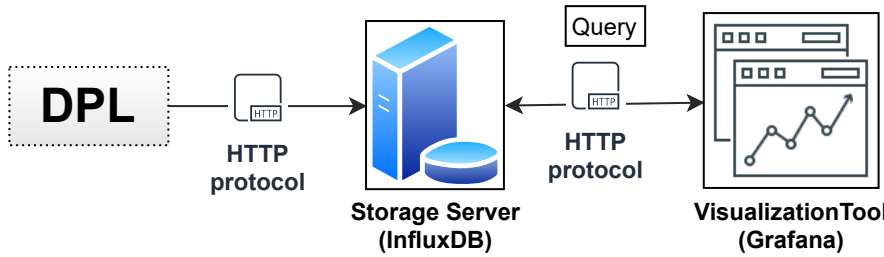


Figure 14. Data Storage and Visualization Layer

received includes sensor readings, metadata, and other gateway-related information. The pre-processing task is to extract the required json fields and normalize the sensor readings.

Further, parsed data are moved to the next task to prepare these data as per InfluxDB line protocol format and also forward to the outlier detection task to identify the anomaly in the data using the Machine Learning model. The training, testing, and exporting ML algorithm is explained in the article[4]. The outlier tagging task is to tag the data that the data stream has anomalous sensor readings. Furthermore, it is also to be stored in the storage server. The *Insert to storage* task invokes the HTTP API of influx to insert data into InfluxDB in the DVSL layer.

Data Processing Layer. The primary function of the DSVL as shown in Figure 14 is to store historical data and to visualize the sensory readings to end-users. It consists of InfluxDB scalable time-series database to store the parsed data based on time intervals and Grafana visualization tool to generate interactive charts.

2.7. Summary

This chapter offers a comprehensive overview of concepts, architectures, and preliminary aspects of technologies associated with IoT data processing. We describe the modified architecture of a seven-layered IoT system, focusing on Layer 3, which aligns with the scope of the proposed thesis, specifically focusing on data processing. In addition, we describe the compute continuum within the IoT system, which includes cloud, fog, and edge computing. The proposed thesis focuses on the compute continuum architecture, with a particular emphasis on edge/fog-based data processing. Further, we described container applications, container engines, and container migration concepts that are heavily used in Chapter 3. Serverless computing architecture along with open source and cloud vendor solutions are briefly described, and we also provide hints on serverless data pipelines with an example, where in this we provide background to Chapter 4 and Chapter 5. Finally, we precisely made an attempt to describe real- time IoT applications which are used in Chapter 3, Chapter 4, and Chapter 5 for evaluation of proposed concepts.

3. CONTAINER APPLICATIONS FOR IOT DATA PROCESSING

This chapter examines the use of monolithic containerized applications in IoT data processing in fog environments. The objective is to explore the use of containerized applications and scheduling techniques that can improve Quality of Service (QoS), such as latency and energy consumption. Additionally, a novel gradient-based optimization strategy is proposed that uses backpropagation of gradients with respect to input using coupled simulation. To demonstrate this, a simulation and testbed framework known as COSCO is designed. The overall contribution of this chapter provides answers to Research Question 1 (RQ1).

3.1. Introduction

In the context of IoT data processing, the use of containers has been shown to be advantageous in various studies [66]–[68]. For example, Pankaj et al. [83] employed a container-based architecture to illustrate a video-based analytics use case for IoT data processing. Additionally, Ahmed et al.[49] proposed a container-based resource management system for data processing on IoT gateways. Casalicchio et al.[99] conducted an investigation to adapt container applications and their challenges in geodistributed applications (Edge, Fog devices). They have described several use cases such as video streaming and multiplayer gaming applications, and mobile edge computing applications.

Data processing on the edge of the network using containers is beneficial due to its light weight and portability and can be migrated, moved between various heterogeneous devices over the fog and edge layers [100]. As described in the background section, data operations are consolidated within a containerized application designed to execute on the underlying infrastructure, which could be edge or fog devices. When users make requests for data processing, these requests are scheduled to execute on designated hosts or nodes within the infrastructure. When the user request is scheduled, meaning that the processing logic encapsulated in the container image is brought to host, where a container is instantiated, and data processing is initiated. In this workflow, the crucial aspect is to ensure that scheduling decisions are appropriate to meet user expectations and achieve the desired Quality of Service (QoS).

However, scheduling user requests (also known as tasks) and managing in large-scale fog environments is challenging due to the highly volatile nature of modern workload applications and sensitive user requirements such as low response times and low-energy footprints.

Container orchestration involves a variety of activities, such as making container images available, scheduling, placing containers on hosts, and migrating containers (application migration) to meet quality of service (QoS) expectations.

Several research studies have investigated container migration approaches to improve application execution and reduce response time and other QoS metrics. Carlo et al. investigated the performance of fog applications based on container migration to reduce response time [101]. Similarly, Kaur et al. [102] conducted a survey to explore how container placement and migration approaches can benefit and improve QoS parameters in the execution of containerized applications. The key challenge is orchestrating all activities that adhere to the scheduling decisions offered by the scheduling algorithm, placing the containers on the appropriate hosts. To our knowledge, such orchestration platforms to simulate and test on real platforms (including migration modules) for research and development are not available.

As part of scheduling algorithms, current state-of-the-art algorithms focus on heuristic, evolutionary, max-weight-based and reinforcement learning. For example, Several studies [105], [178], [179] have shown how heuristic based approaches perform well in a variety of scenarios for optimization of diverse QoS parameters. However, such heuristic-based approaches do not model stochastic environments with dynamic workloads [69]. Other prior work has shown that evolutionary-based methods, and generally gradient-free approaches, perform well in dynamic scenarios [68], [72], [181]. They take much longer to converge [182] and are not as scalable [183] as gradient-based methods. However, MaxWeight policies can exhibit poor delay performance, instability in dynamic workloads, and spatial inefficiency [184], [185]. To overcome this, GOBI and GOBI* are able to adapt to dynamic scenarios by constantly learning the mapping of scheduling decisions with objective values. This makes them both robust to diverse, heterogeneous, and dynamic scenarios.

Given the challenges outlined earlier, this chapter makes a triple-fold contribution toward addressing Research Question 1 (RQ1). First, we formulate the problem of containerized application scheduling in dynamic workload environments and further investigate state-of-the-art container scheduling methods and their associated challenges. Second, we introduce a tool known as the Coupled Simulation and Container Orchestration Framework (COSCO) designed to simulate and test scheduling approaches employing container migration in real test environments. Lastly, we developed an intelligent scheduling algorithm utilizing a back propagation-based approach informed by coupled simulation.

3.2. System model and Problem formulation

3.2.1. System Model

We consider standard distributed and heterogeneous devices in a three-tier architecture, as shown in Figure 1. Our system model has three layers such as the edge layer with sensors, the fog, and the cloud layer). We consider tasks to be monolithic containerized application instances with input data that are being generated

by sensors and other IoT devices, and results received by end users or actuators that initiate controlling actions. These devices constitute the edge layer and send and receive all data from the fog gateway devices. The fog broker or controller is responsible for the management of tasks and hosts and performs activities such as task scheduling, data management, resource monitoring, and container orchestration. The focus of this chapter is on improving the Fog Broker scheduler by providing an interface between container orchestration and resource monitoring services with a simulator. Our work proposes discrete-time controllers, which are commonly adopted in the literature [69], [71].

We consider communication between end-users and the fog broker to be facilitated by gateway devices. The compute nodes in the fog layer are referred to as *hosts*, with various compute configurations. Hosts at the edge of the network are resource constrained and have low communication latency with the broker and gateway devices. On the other hand, cloud hosts are several hops from the users and have much higher communication latency and are computationally more powerful. We assume that there are a fixed number of host machines in the fog resource layer and denote them by $H = \{h_0, h_1, \ldots, h_{N-1}\}$. We denote the collection of time series utilization metrics, which include CPU, RAM, Disk and Network Bandwidth usage of host h_i as $U(h_i^t)$. The collection of maximum capacities of CPU, RAM, Disk, and Network Bandwidth with the average communication latency of host h_i is denoted as $C(h_i)$.

3.2.2. Workload Model

The workload model represents the characteristics of the requests generated by IoT devices. We divide the timeline of user requests generated into equal-sized scheduling intervals of equal duration Δ. The t-th interval is denoted by I_t and starts at $s(I_t)$, so $s(I_0) = 0$ and $s(I_t) = s(I_{t-1}) + \Delta \; \forall t > 0$. In the interval I_{t-1}, N_t IoT devices create new tasks. The gateway devices then send batches of these new tasks N_t with their SLO requirements to the Fog Broker. The SLO violations are measured corresponding to the response time metrics of task executions as a fraction of them that exceeds the stipulated deadlines. The set of active tasks in the interval I_t is denoted as $A_t = \{a_0^t, a_1^t, \ldots, a_{|A_t|}^t\}$ and consists of $|A_t|$ tasks. Here a_j^t represents an identifier for the j-th task in A_t.

The Fog Broker schedules the new tasks to the compute nodes and decides which of the active tasks need to be migrated. At the end of interval I_{t-1}, the set of completed tasks is denoted as L_t, hence the tasks that carry forward to the next interval is the set $A_{t-1} \setminus L_t$. Thus, the broker takes a decision $D^t = D(Y_t)$, where Y_t is the union of the new (N_t), active ($A_{t-1} \setminus L_t$) and waiting tasks (W_{t-1}). This decision is a set of allocations and migrations in the form of ordered-pairs of tasks and hosts (assuming no locality constraints). At the first scheduling interval, as there are no active or waiting tasks, the model only takes allocation decision for new tasks (N_0). Here, D denotes the scheduler such that $D : Y_t \mapsto Y_t \times H$. Only

if the allocation is feasible, *i.e.*, depending on whether the target host can accommodate the scheduled task or not, is the migration/allocation decision executed. Initially, the waiting queue and the set of active and leaving actions are empty. The executed decision is denoted by $\hat{D}$ and satisfies the following properties

$$\hat{D}(A_t) \subseteq D(Y_t), \text{where} \tag{3.1}$$

$$\forall\ (a_j^t, h_i) \in \hat{D}(A_t),\ U(a_j^t) + U(h_i^t) \leq C(h_i). \tag{3.2}$$

We use the notation $a_j^t \in \hat{D}(A_t)$ to mean that this task was allocated/migrated. The set of new and waiting tasks $\hat{N}_t \cup \hat{W}_{t-1} \subseteq \hat{D}(A_t)$, where $\hat{N}_t \subseteq N_t$ and $\hat{W}_{t-1} \subseteq W_{t-1}$, denotes those tasks which could be allocated successfully. The remaining new tasks, *i.e.* $N_t \setminus \hat{N}_t$ are added to get the new wait queue W_t. Similarly, active tasks are denoted as $\hat{A}_{t-1} \subseteq A_{t-1}$. Hence, for every interval I_t,

$$A_t \leftarrow \hat{N}_t \cup \hat{W}_{t-1} \cup A_{t-1} \setminus L_t \tag{3.3}$$

$$W_t \leftarrow (W_{t-1} \setminus \hat{W}_{t-1}) \cup (N_t \setminus \hat{N}_t), \text{where} \tag{3.4}$$

$$W_{-1} = A_{-1} = L_{-1} = \emptyset. \tag{3.5}$$

We also denote the utilization metrics of an active task $a_j^t \in A_t$ as $U(a_j^t)$ for the interval I_t. We denote a simulator as $S : \hat{D}(A_t) \times \{U(a_j^t) | \forall a_j^t \in A_t\} \mapsto \{U(h_i^t) | \forall h_i \in H\} \times P$, which takes scheduling decision and container utilization characteristics to give host characteristics and values of QoS parameters. Here $P_t \in P$ is a set of QoS parameters such as energy consumption, response times, SLO violations, etc. Similarly, execution on a physical fog framework is denoted by F. Hence, execution of interval I_t on a physical setup is denoted as

$$(\{U(h_i^t) | \forall h_i \in H\}, P_t) \equiv F(\hat{D}(A_t), \{U(a_j^t) | \forall a_j^t \in A_t\}). \tag{3.6}$$

3.2.3. Problem Formulation

After the execution of tasks A_t in the interval I_t, the objective value to minimize is denoted as $O(P_t)$. $O(P_t)$ could be a scalar value which combines multiple QoS parameters. To find the optimum schedule, we need to minimize the objective function $O(P_t)$ over the entire execution duration. Therefore, we need to find the appropriate and feasible decision function D such that $\sum_t O(P_t)$ is minimized. This is subject to the constraints that at each scheduling interval the new tasks N_t, waiting tasks W_{t-1} and active tasks from the previous interval $A_{t-1} \setminus L_t$ are allocated using this decision function. The problem can then be concisely formulated as:

$$\begin{aligned} \underset{D}{\text{minimize}} \quad & \sum_{t=0}^{T} O(P_t) \\ \text{subject to} \quad & \forall\, t, Eqs.(3.1) - (3.6). \end{aligned} \tag{3.7}$$

Table 3. Symbol Table

Symbol	Meaning
I_t	t^{th} scheduling interval
A_t	Active tasks in I_t
W_{t-1}	Waiting tasks at the start of I_t
L_t	Tasks leaving at the end of I_t
N_t	New tasks received at the start of I_t
H	Set of hosts in the Resource Layer
h_i	i^{th} host in an enumeration of H
a_j^t	j^{th} task in an enumeration of A_t
$U(h_i^t)$	Utilization metrics of host h_i in I_t
$U(a_j^t)$	Utilization metrics of task a_j^t in I_t
$Y_t = N_t \cup W_{t-1} \cup A_{t-1} \setminus L_t$	Scheduler input at the start of I_t
$D(Y_t)$ or D	Scheduling decision at start of I_t
$\hat{D}$	Feasible sub-set of scheduling decision D
$\bar{D}$	Scheduling decision of GOBI in GOBI* loop
$O(P_t)$	Objective value at the end of I_t
S	Execution of an interval on simulator
F	Execution of an interval on physical setup

Further *Scheduler* module of the Fog Broker will optimize an objective function $O(P_t)$ at every interval I_t via taking an optimal action in the form of scheduling decision D. This D is then used by the framework/simulator to execute tasks as described in optimization program (3.7). We extend the objective function to (3.7) to make the appropriate scheduling decisions in a fog environment that focuses on energy consumption and response time, as QoS metrics which are most crucial metrics for fog environments [103], [104]. For interval I_t,

$$O(P_t) = \alpha \cdot AEC_t + \beta \cdot ART_t. \tag{3.8}$$

Here, AEC and ART ($\in P_t$) are defined as follows.

1. *Average Energy Consumption* (AEC) is defined for any interval as the energy consumption of the infrastructure (which includes all edge and cloud hosts) normalized by the maximum power of the hosts.
2. *Average Response Time* (ART)[1] is defined for an interval I_t as the average response time for all leaving tasks (L_t), normalized by maximum response time until the current interval.

3.3. Scheduling algorithms

Considering the objective function from the equation 3.8, in the following subsection we describe the proposed Gradient Based Optimization using Back-propagation to Input (GOBI) and GOBI*. Further we elaborate the state of art scheduling algorithms which are used as baseline for the proposed approach.

3.3.1. Proposed algorithms

We present two novel algorithms in this work: GOBI and GOBI*. GOBI uses a neural network as a surrogate model and gradient based optimization using back-propagation of gradients to input. With advances like cosine annealing and momentum allow us to converge to an optima quickly. Moreover, GOBI* leverages a coupled simulation engine like a digital-twin to further improve the surrogate accuracy and subsequently the scheduling decisions.

GOBI scheduler. In GOBI scheduler, We optimize the objective function $O(P_t)$ within each interval I_t by selecting an optimal scheduling decision D. The chosen D is subsequently used by the framework or simulator to execute tasks as described in the optimization program (3.8).

To train the model, input parameters are considered as follows: We consider a finite maximum number of active tasks as M. For any given interval, $|A_t| \leq M$. The feature vector of size F comprises utilization metrics of Instructions per second (IPS), RAM, Disk, and Bandwidth consumption. Taking into account this, the task utilization matrices $\{U(a_j^{t-1}) | \forall j, a_j^{t-1} \in A_{t-1}\}$ as an $M \times F$ matrix denoted as

[1]Both AEC and ART are unit-less metrics and lie between 0 and 1

$\phi(A_{t-1})$, where the first A_{t-1} rows are the feature vectors of tasks in A_{t-1} in order of increasing creation intervals, and the rest of the rows are 0. Similarly, feature vector of host utilization with IPS, RAM, Disk and Bandwidth consumption and communication latency of each host each of size F'. Thus, at each interval I_t for $|H|$ hosts, we form a $|H| \times F'$ matrix $\phi(H_{t-1})$ using host utilization metrics of interval I_{t-1}. Finally, as we have N_t, W_{t-1}, A_{t-1} and L_t at the start of I_t, the decision matrix $\phi(D)$ is an $M \times |H|$ matrix with the first $|N_t \cup W_{t-1} \cup A_{t-1} \cup L_t|$ rows being one-hot vector of allocation of tasks to one of the H hosts. The remaining rows being 0.

We now describe how a neural model can be trained to approximate $O(P_t)$ using the input parameters $[\phi(A_{t-1}), \phi(H_{t-1}), \phi(D)]$. Consider a continuous function $f(x;\theta)$ as a neural approximator of $O(P_t)$ with the θ vector denoting the underpinning neural network parameters and x as a continuous or discrete variable. Here, x is the collection of utilization metrics of tasks and hosts with a scheduling decision. The parameters θ are learnt using the dataset Λ which is defined as $\Lambda = \{[\phi(A_{t-1}), \phi(H_{t-1}), \phi(D)], O(P_t)\}_t$ such that a given loss function L is minimized for this dataset. We form this dataset by running a random scheduler and saving the traces generated. The loss L quantifies the dissimilarity between the predicted output and the ground truth. We use Mean Square Error (MSE) as the loss function as done in prior work [69]. Hence,

$$L(f(x;\theta), y) = \frac{1}{T}\sum_{t=0}^{T}(y - f(x;\theta))^2, \text{ where} (x,y) \in \Lambda.$$

Thus, for datapoints $(x,y) \in \Lambda$, where y is the value of $O(P_t)$ for the corresponding x, we have $\theta = \arg\min_{\hat{\theta}} \sum_{(x,y)\in\Lambda}[L(f(x;\hat{\theta}), y)]$. To do this, we calculate the gradient of the loss function with respect to θ as $\nabla_\theta L$ and use back-propagation to learn the network parameters.

Now, with this, we need to find the appropriate decision matrix $\phi(D)$, such that $f(x;\theta)$ is minimized. We formulate the resulting optimization problem as

$$\begin{aligned} \underset{\phi(D)}{\text{minimize}} \quad & f(x;\theta) \\ \text{subject to} \quad & \text{each element of } \phi(D) \text{ is bounded} \\ & \forall\, t, Eqs.(3.1) - (3.6). \end{aligned} \tag{3.9}$$

This is a reformulation of the program (3.7) using the neural approximator of $O(P_t)$ as the objective function. Note that, because there is a bounded space of inputs, there must be an optimal solution, *i.e.*, $\exists\ \hat{\phi}(D)$ such that

$$\begin{aligned} f([\phi(A_{t-1}), \phi(H_{t-1}), \hat{\phi}(D)]; \theta) \leq \\ f([\phi(A_{t-1}), \phi(H_{t-1}), \phi(D)]; \theta),\ \forall \text{ feasible } \phi(D). \end{aligned}$$

We solve this optimization problem using gradient-based methods, for which we need $\nabla_x f(x;\theta)$. Once we have the gradients, we can initialize an arbitrary input x and use gradient descent to minimize $f(x;\theta)$ as

$$x_{n+1} \leftarrow x_n - \gamma \cdot \nabla_x f(x_n;\theta), \tag{3.10}$$

where γ is the learning rate and n is the iteration count. We apply Eq. (3.10) until convergence. The calculation of $\nabla_x f(x_n;\theta)$ in the above equation is carried out in a similar fashion as back-propagation for model parameters.

After training the model f, at start of each interval I_t, we optimize $\phi(\mathscr{D})$ by the following rule (γ is the learning rate) until the absolute value of the gradient is more than the convergence threshold ε, *i.e.*,

$$\phi(D)_{n+1} \leftarrow \phi(D)_n - \gamma \cdot \nabla_{\phi(D)} f(x_n;\theta). \tag{3.11}$$

In dynamic scenarios, as the approximated function for the objective $O(P_t)$ changes with time we continuously fine-tune the neural approximator as per the loss function $MSE(O(P_{t-1}), f([\phi(A_{t-2}), \phi(H_{t-2}), \phi(D^{t-1})];\theta))$ (line 20 in Algorithm 1). Equation 3.11 then leads to a decision matrix with a lower objective value (line 6 in Algorithm 1). Hence, *gradient based iteration with continuous fine-tuning of the neural approximator allows GOBI to adapt quickly in dynamic scenarios.* When the above equation converges to $\phi(D^*)$, the rows represent the likelihood of allocation to each host. We take the arg max over each row to determine the host to which each task should be allocated as D^*. This then becomes the final scheduling decision of the GOBI approach. The returned scheduling decision is then run on a simulated or physical platform (as in Eq. (3.6)) and tasks sets are updated as per Eqs. (3.3)-(3.5). To find the executed decision $\hat{D}$, we sort all tasks in D^* in descending order of their wait times (breaking ties uniformly at random) and then try to allocate them to the intended host in D^*. If the allocation is not possible due to utilization constraints, we ignore such migrations and do not add them to $\hat{D}$. At run-time, we fine tune the trained neural model f, by newly generated data for the model to adapt to new settings.

We now discuss the novelty of the COSCO framework. As all scheduling algorithms and simulations are run on a central fog broker [108], it can be easily run in fog environments which follow a master-slave topology. As discussed in the architecture, a simulator can execute tasks on simulated host machines to return QoS parameters P. To execute an interval I_t, the simulator needs utilization metrics $U(a_j^t), \forall a_j^t \in A_t$, and a scheduling decision $\bar{D}^t$ to be executed on the simulator. Thus, at the beginning of the interval I_t, we have L_t, N_t, W_{t-1} and A_{t-1}. After checking for the possibility of allocation/migration, we get $\hat{\bar{D}}(A_t)$, $\hat{N}_t$ and $\hat{W}_{t-1}$. Now, using Eq. (3.3), we find A_t. Using given utilization metrics $U(a_j^t)$ and $\hat{\bar{D}}(A_t)$, we execute A_t tasks on the simulator to get $S(\hat{\bar{D}}(A_t), \{U(a_j^t)|\forall a_j^t \in A_t\}) = \{U(h_i^t)|\forall h_i \in H\}, \bar{P}_t$. This means that at the beginning of interval I_t, the COSCO framework allows simulation of the next scheduling interval (with an action of

Algorithm 1 The GOBI scheduler

Require:

Pre-trained function approximator $f(x;\theta)$
Dataset used for training Λ; Convergence threshold ε
Iteration limit σ; Learning rate γ
Initial random decision D

1: **procedure** MINIMIZE(D, f, z)
2: Initialize decision matrix $\phi(D)$; $i = 0$
3: **do**
4: $x \leftarrow [z, \phi(D)]$ ▷ Concatenation
5: $\delta \leftarrow \nabla_{\phi(D)} f(x;\theta)$ ▷ Partial gradient
6: $\phi(D) \leftarrow \phi(D) - \gamma \cdot \delta$ ▷ Decision update
7: $i \leftarrow i + 1$
8: **while** $|\delta| > \varepsilon$ and $i \leq \sigma$
9: Convert matrix $\phi(D)$ to scheduling decision D^*
10: **return** D^*
11: **end procedure**
12: **procedure** GOBI(scheduling interval I_t)
13: **if** (t == 0)
14: Initialize random decision D
15: **else**
16: $D \leftarrow D^*$ ▷ Output for the previous interval
17: Get $\phi(A_{t-1}), \phi(H_{t-1})$
18: $D^* \leftarrow$ MINIMIZE$(D, f, [\phi(A_{t-1}), \phi(H_{t-1})])$
19: Fine-tune f with loss =
20: $MSE(O(P_{t-1}), f([\phi(A_{t-2}), \phi(H_{t-2}), \phi(D^{t-1})]; \theta))$
21: **return** D^*

interest $\bar{D}$ and predicted utilization metrics) to predict the values of the QoS parameters in the next interval $\bar{P}_t$. This single step look-ahead simulation allows us to take better scheduling decisions in the GOBI* algorithm, as discussed below.

Now that we have a scheduling approach based on back-propagation of gradients, we extend it to incorporate simulated results. To do this, we use the following components.

1. *GOBI Scheduler*: We assume that we have the GOBI scheduler which can give us a preferred action of interest $\bar{D}$. Thus, at the start of a scheduling interval I_t, we get $\bar{D} = GOBI(I_t)$ (line 8 in Algorithm 2). D now denotes the decision of GOBI*.
2. *Utilization prediction models*: We train a utilization metric prediction model, such that using previous utilization metrics of the tasks, we get a predicted utilization metric set for the next interval. We use a Long-Short-Term-Memory (LSTM) neural network for this and train it using the same Λ dataset that we used for training the GOBI neural approximator. Thus, at

Algorithm 2 The GOBI* scheduler

Require:

Pre-trained function approximator $f^*(x;\theta^*)$
Pre-trained LSTM model $LSTM(\{U(a_j^{t'})\forall t' < t\})$
Dataset used for training Λ^*; Convergence threshold ε
Learning rate γ; Initial random decision $\mathscr{D}$

1: **procedure** GOBI*(scheduling interval I_t)
2: **if** (t == 0)
3: Initialize random decision D
4: **else**
5: $D \leftarrow D^*$ ▷ Output for the previous interval
6: Get $\phi(A_{t-1}), \phi(H_{t-1})$
7: $\bar{U}(a_j^t)\forall a_j^t \in A_t \leftarrow LSTM(\{U(a_j^{t'})\forall t' < t\})$
8: $\bar{D} \leftarrow GOBI(I_t)$
9: $\{U(h_i^t)|\forall h_i \in H\}, A\bar{E}C_t, A\bar{R}T_t \leftarrow$
10: $S(\hat{\bar{D}}(A_t), \{\bar{U}(a_j^t)\forall a_j^t \in A_t\})$
11: $O(\bar{P}_t) \leftarrow \alpha \cdot A\bar{E}C_t + \beta \cdot A\bar{R}T_t$
12: $D^* \leftarrow$ MINIMIZE$(D, f, [\phi(A_{t-1}), \phi(H_{t-1}), O(\bar{P}_t)])$
13: Fine-tune f^* with loss =
14: $MSE(O(P_{t-1}), f([\phi(A_{t-2}), \phi(H_{t-2}), \phi(D^{t-1})];\theta)) + 1(O(\bar{P}_{t-1}) < O(P_{t-1})) \times MSE(\bar{D}^{t-1}, D^{t-1})$
15: **return** D^*

the start of interval I_t, using $\bar{D}$ from GOBI and checking allocation possibility, we get $\hat{N}_t$ and $\hat{W}_{t-1}$. Using (3.3), we get A_t. Then, we predict $\bar{U}(a_j^t), \forall a_j^t \in A_t$, using the LSTM model (line 7 in Algorithm 2). Hence, we get the $\phi(A_{t-1}), \phi(H_{t-1})$ matrices.

3. *Simulator*: Now that we have an action $\bar{D}$ and predicted utilization metrics $\bar{U}(a_j^t), \forall a_j^t \in A_t$, we can use the simulator to predict the QoS parameters at the end of I_t as described in lines 9 and 10 in Algorithm 2.

Since at the start of the interval I_t, we do not have utilization models of all containers and subsequently hosts for I_t, our GOBI model is forced to predict QoS metrics at the end of I_t using only utilization metrics of the previous interval, *i.e.*, I_{t-1}. This can make GOBI's predictions inaccurate in some cases. However, if we can predict with reasonable accuracy the utilization metrics in I_t using the utilization values of the previous intervals, we can simulate and get an improved estimate of the QoS parameters during the next period I_t. Adding these as inputs to another neural approximator for $O(P_t)$ denoted as $f^*(x, \theta^*)$, we now have $x = [\phi(A_{t-1}), \phi(H_{t-1}), O(\bar{P}_t), \phi(D)]$.

Again, using a random scheduler and pre-trained GOBI model, we generate a dataset Λ^* to train this new neural approximator f^*. At execution time, we freeze the neural model f of GOBI and fine tune f^* with the loss function. This loss function is the MSE of objective values. We add the MSE between the predicted

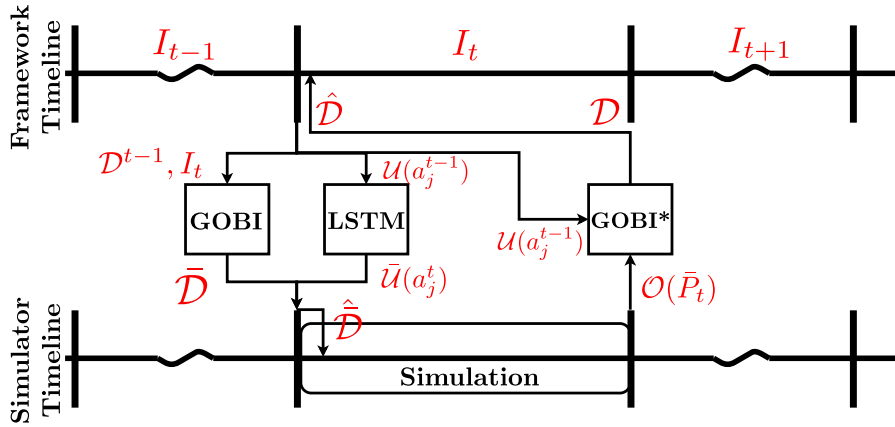


Figure 15. Iteration of GOBI* approach at the start of interval I_t

action D by GOBI* and $\bar{D}$ by GOBI in case the estimated objective value of GOBI is lower than that of GOBI*. Thus,

$$L = MSE(O(P_{t-1}), f^*([\phi(A_{t-2}), \phi(H_{t-2}), \phi(D^{t-1})]; \theta^*)) + 1(O(\bar{P}_{t-1}) < O(P_{t-1})) \times MSE(\bar{D}^{t-1}, D^{t-1}). \quad (3.12)$$

Using this trained model, we can now provide the schedules as shown in Algorithm 2 and Figure 15. At interval I_t, the previous output of GOBI*, *i.e.*, D^{t-1} is given to the GOBI model as the initial decision. The GOBI model uses utilization metrics to output the action $\hat{D}$. Using a single-step simulation, we obtain an estimate of the objective function for this decision. Then, GOBI* uses this estimate to predict the next action D. Based on whether GOBI*'s decision is better than GOBI or not, GOBI* is driven to the decision which has lower objective value. *This interactive dynamic should allow GOBI* to make a more informed prediction of QoS metrics for I_t and hence perform better than GOBI.*

3.3.2. Baseline algorithms

In this subsection, we describe an existing state of the art optimization approaches used in containerized scheduling in IoT environments, which are used as baseline with our proposed approach.

- *Heuristic approaches*: Research literature shows that heuristics methods perform well in optimization of various QoS metrics in diverse scenarios[104], [105]. Approach LR-MMT proposed by Beloglazov et.al[105], Linear regression (LR) was used to predict which host will be overloaded in near future and Minimum Migration Time (MMT) was used to select the containerized applications/tasks which can be migrated. Together LR-MMT was used to schedule the container/task dynamically, where LR was used for overload detection and MMT for selection of the tasks. Similarly, MAD-MC heuristic algorithm calculates the mean attribute deviation of the

CPU utilization of the hosts for overload detection and uses the minimum correlation to select the containerized applications for migration [105].

- *Evolutionary approaches*: This approaches are gradient free optimization for example Genetic algorithm uses the neural model for object value approximation and optimization to reach the optimal value and this makes to perform well in dynamic scenarios[72].
- *Max-Weight based approaches*: Max-weight approaches employ the technique of assigning priorities based on certain criteria, and the task/resource with the highest weight is allocated or scheduled at each time interval. This theoretical approach has the capacity to reduce resource contention in decision-making, which is why it is so popular. As a baseline , we used POND[107] by Liu et al. which is a variant of Max-Weight approach.
- *Reinforcement Learning (RL) models*: Recently RL based approaches have taken considerable interest in optimizing the edge and fog systems. The RL approaches uses the decision making process using Markovian assumption to obtain the optimal and desirable state for each interval of scheduling process by observation of current state and reward for optimal decision. A recent method, *DQLCM*, models the container migration problem as a multi-dimensional Markov Decision Process (MDP) and uses a deep-reinforcement learning strategy, namely deep Q-Learning to schedule workloads in a heterogeneous fog computing environment [106]. The state-of-the-art method, *A3C*, schedules workloads using a policy gradient based reinforcement learning strategy which tries to optimize an actor-critic pair of agents [69]. We will use these two approaches as a baseline algorithms to compare the RL and proposed approaches.

3.4. COSCO Architecture

Coupled or symbiotic simulation and model based control have long been used in the modelling and optimization of distributed systems [108]–[110]. Many prior works have used hybrid simulation models to optimize decision making in dynamic systems. To achieve this, they monitor, analyze, plan and execute decisions using previous knowledge-base corpora (MAPE-k) [113]. However, such works use this to facilitate search methods and not to generate additional data to aid the decision making of an AI model. COSCO is developed to leverage a seamless interface between the orchestration framework and simulation engine to have a interactive dynamic between AI models to optimize QoS.

We present the design of the COSCO simulator and the framework together with the details of the interface. COSCO is an event-driven simulator and framework for container orchestration in fog environments. It is a simple Python-based software solution that researchers can use to simulate, test, and deploy their scheduling algorithms for containerized applications by modeling user workloads.

It contains several modules classified into simulator, framework, common modules, along with controller, and fog node modules, as shown in Figure 16. The *COSCO modules* that are running on the controller node can be a developer machine that is capable of simulation of discrete events and should be easy to connect to the fog nodes through passwordless ssh. Fog nodes configured with the COSCO agent service, which is used to push and pull the instructions and data from the COSCO framework module. The fog nodes could be LAN connected machines, cloud based virtual machines located at different availability zones, or could be RPi cluster. The following section will provide details of each of the modules in the COSCO ecosystem.

Common Modules and Controller: The *controller module* is the main thread, where it provides an interface for developers to create the simulated and framework environment, setup and model the workloads, choose the scheduling algorithm, and can also initiate the experiments. In simple terms, main entry point to the implementation which runs the simulation or execution of tasks in a physical environment. It uses preimplemented schedulers, data center configurations, workload models, and database modules to simulate/execute containerized applications. The *Stats* module monitors and logs information on workloads, hosts, containers, and workloads for complete execution (simulation or physical execution). The *Grapher* generates comparative graphs for the raw traces saved from the Stats module. *Utility* module is used for the automated installation script used by the controller module, which installs the required packages to configure the fog nodes by installing the agent service. *Scheduler* module has the implementation of various compatible schedulers and is used by both the Simulator and the Framework modules. Moreover, the *Scheduler* class allows taking scheduling decisions in the form of task selection *getSelection()* and placement *getPlacement()*. Task selection function selects the containers that need to be migrated to another host; task placement function returns the target host for each selected or new container. *Metrics* module is used to store CPU(IPS), RAM, disk, and bandwidth consumption of tasks/containers at the end of each scheduling interval. In addition, common modules are shared between the simulator and the framework modules.

Simulator modules: Environment is the implementation of the Simulator class which allocates new workloads, determines if allocation/migration is possible, executes event-driven simulation and destroys completed workload. The controller modules initialize with the required data for Environment. The *Container* is the implementation of a simulated container application with CPU(IPS), RAM, Disk and Bandwidth time series models and it depicts the real time application instance. The *Host* is the implementation of simulated host objects with IPS, RAM, Disk, Bandwidth capacities and utilization values. The host depicts the real time fog nodes in the IoT enviornment. The *workload* module is used for generating container-based workloads or user requests using traces of static/bitbrain applications. The number of workloads being generated at each interval can be set by the

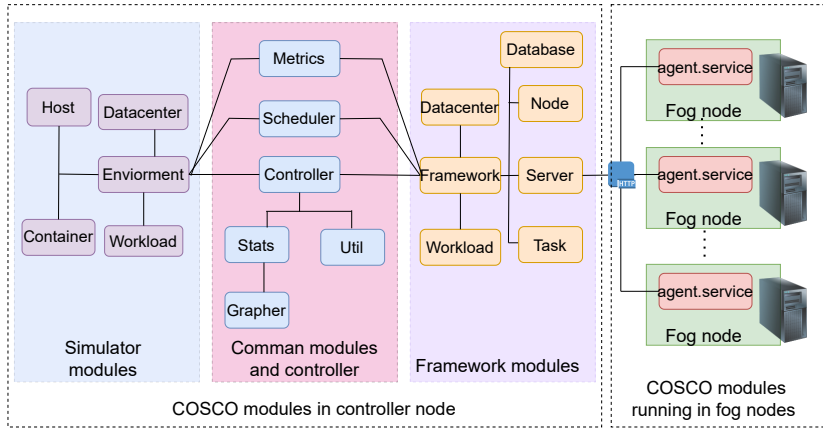


Figure 16. COSCO architecture

user. The hosts are created/simulated using the *Datacenter* class which provides a list of *Host* objects to the simulator or framework.

Framework modules: The framework modules are used in implementation of the physical container management framework, which monitors hosts and containers, determines if container placement is possible, checkpoints/migrates/restores containers, and destroys completed containers. *Task* is the identifier object for a container that runs in the framework with IPS, RAM, disk and bandwidth utilization values. *Node* is the identifier object for a host with IPS, RAM, disk, bandwidth and utilization values. *Database* is the InfluxDB client which maintains time-stamped information of all hosts and containers. *Datacenter* is the implementation of Datacenter class which configures host machines in the physical testbed and obtains the IPS, RAM, Disk and Bandwidth capacities. *Server* enable interaction with the individual agents for container management scripts based on Python DockerClient API. *Agent.service* recieves the request from Server and execute the request and sends the response which runs as a flask service.

The following are the key features of COSCO:

- Easy to model the AI based scheduling algorithms for integrated edge, fog and cloud infrastructures, because the researchers have to only extend one to their custom algorithms.
- Provides seamless integration of scheduling policies with simulated backend for enhanced decision-making along with coupled simulation to test in real-time environments.
- Supports container migration on physical deployments (not supported by other frameworks) using the CRIU utility.
- Developers can deploy multiple solutions depending on their requirements. These include a Vagrant virtual machine testbed, a VLAN fog environment, and cloud-based deployment using Azure, AWS, or OpenStack.
- Equipped with smart real-time graph generation of utilization metrics using

Table 4. Host characteristics of Azure fog environment.

Name	Quantity	Core count	MIPS	RAM	RAM Bandwidth	Ping time	Network Bandwidth	Disk Bandwidth	Cost Model	Location
Edge Layer										
Azure B2s server	4	2	4029	4295 MB	372 MB/s	3 ms	1000 MB/s	13.4 MB/s	0.0472 $/hr	London, UK
Azure B4ms server	2	4	8102	17180 MB	360 MB/s	3 ms	1000 MB/s	10.3 MB/s	0.1890 $/hr	London, UK
Cloud Layer										
Azure B4ms server	2	4	8102	17180 MB	360 MB/s	76 ms	1000 MB/s	10.3 MB/s	0.166 $/hr	Virginia, USA
Azure B8ms server	2	8	2000	34360 MB	376 MB/s	76 ms	2500 MB/s	11.64 MB/s	0.333 $/hr	Virginia, USA

InfluxDB and Grafana.

- Real-time metrics monitoring, logging, and consolidated graph generation using custom Stats logger.

3.5. Evaluation

To test the efficacy of the proposed approaches and compare them with baseline methods, we performed experiments on both simulated and physical platforms. As COSCO has the same underlying models for task migration, workload generation, and utilization metric values for both the simulation and the physical test bench, we can test all models in both environments with the same underlying assumptions. We use $\alpha = \beta = 0.5$ in (3.8) for our experiments.

3.5.1. Experimental setup

The physical environment used in the COSCO framework deployment was leased from the Microsoft Azure cloud provisioning platform. We created a test bed of 10 VMs located in two geographically distant locations with Intel Haswell 2.4 GHz E5-2673 v3 processor core architecture. The gateway devices are part of the same LAN as the server that was hosted in London, United Kingdom. In the set of virtual machines, six are leased from London availability zone and four are from Virginia, United States. The capacities of virtual machines or hosts $C(h_i) \forall h_i \in H$ are shown in Table 4. The power consumption models are taken from the SPEC benchmark repository. We run all experiments for 100 scheduling intervals, each interval being 300s long, giving a total experiment time of 8 hours and 20 minutes. We averaged over five runs and used various types of workload to ensure statistical significance.

In the simulation environment, we consider the setup with 50 host machines as a scaled-up version of the 10 machines from the last subsection. As we cannot place the simulated nodes in geographically distant locations, we model the latency and networking characteristics of these nodes in our simulator according to Table 4.

3.5.2. Workloads

To generate workloads to train the GOBI and GOBI* models and test them against baseline methods, we use two workload characteristics *Bitbrain traces* and *DeFog*

applications. These were chosen because of their non-stationarity, highly volatile workloads, and similarity to many real-world applications. In simulation enviormnet, we used *BitBrain traces* to generate dynamic workloads for cloudlets based on the real-world, open-source Bitbrain dataset [112][2]. The *DeFog applications*[111] are a fog computing benchmarks which consist of six real-time heterogeneous workloads such as Yolo, Pocketspinx, Aeneas, FogLamp and iPokeMon. We use three specific heterogeneous applications of DeFog: Yolo (Memory, Bandwidth and Compute Intensive benchmark), PocketSphinx (Compute Intensive) and Aeneas (Bandwidth Intensive). These applications explained in the Chapter 1 in Section 2.6.

3.5.3. Evaluation metrics

We assess the baseline algorithms and compare them with our proposed GOBI and GOBI* scheduling algorithms using the following evaluation metrics, which have been considered in previous articles [69], [71], [113].

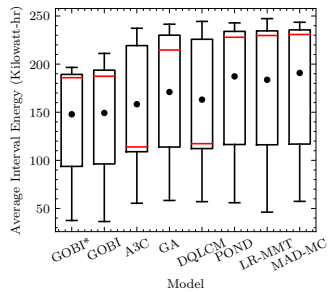
- SLO Violations: We compute SLA violations for DeFog and Bitbrain workloads based on the percentile of response time of the completed tasks, which should not be lower than 95% and is inspired from [31].
- Average Migration Time: It is calculated on the sum of the average migration time of all the tasks over all the intervals in the run.
- Scheduling Time: It is measured as the average time to reach the scheduling decision over all intervals in the run.
- Average Wait Time: It is the average time the container or task waits in the queue before it starts its execution.

3.5.4. Results

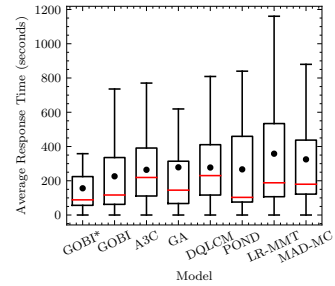
In this section, we provide comparative results showing how GOBI and GOBI* perform against other baselines as described in Section 3.3.2. We compare the proposed approaches with respect to the evaluation metrics described in Section 3.5.3. The graphs in Figure 17 show the results for runs with length of 100 scheduling intervals viz 8 hours 20 minutes using the DeFog workloads on 10 physical Azure machines. Figure 18 shows similar trends in the result for 50 hosts in a simulated environment.

Figure 17(a) shows the energy consumption of Azure host machines for each scheduling policy. Among the baseline approaches, *A3C* consumes the least average interval energy of 221.63 KW-hr. Running the *GOBI* approach on the Azure platform consumes 193.11 KW-hr, 12.86% lower than *A3C*. Further, *GOBI** consumes the least energy 188.26 KW-hr (15.05% lower than that of *A3C*). The major reason for this is the low response time of each application, which leads to more tasks being completed within the same duration of 100 intervals. Therefore, even

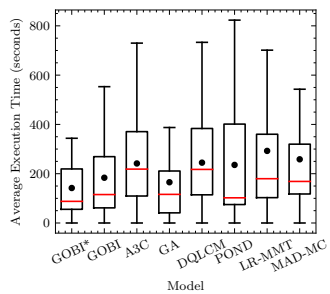
[2] `http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains`



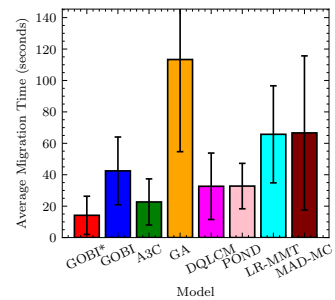
(a) Average Energy Consumption



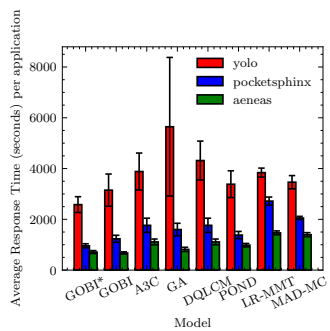
(b) Average Response Time



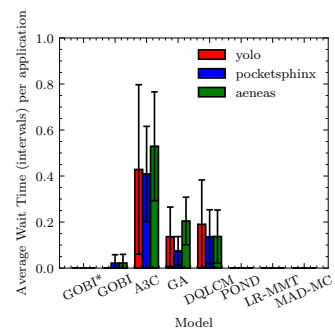
(c) Average Execution Time



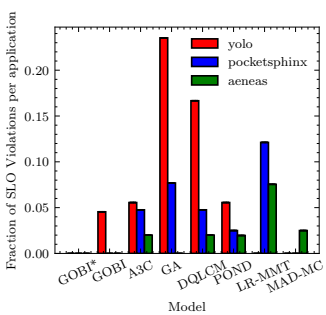
(d) Average Migration Time



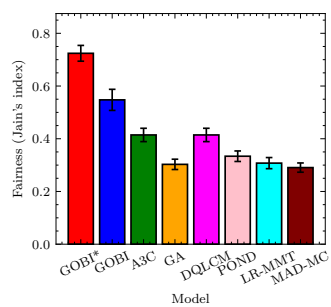
(e) Average Response Time (per application)



(f) Average Wait Time (per application)

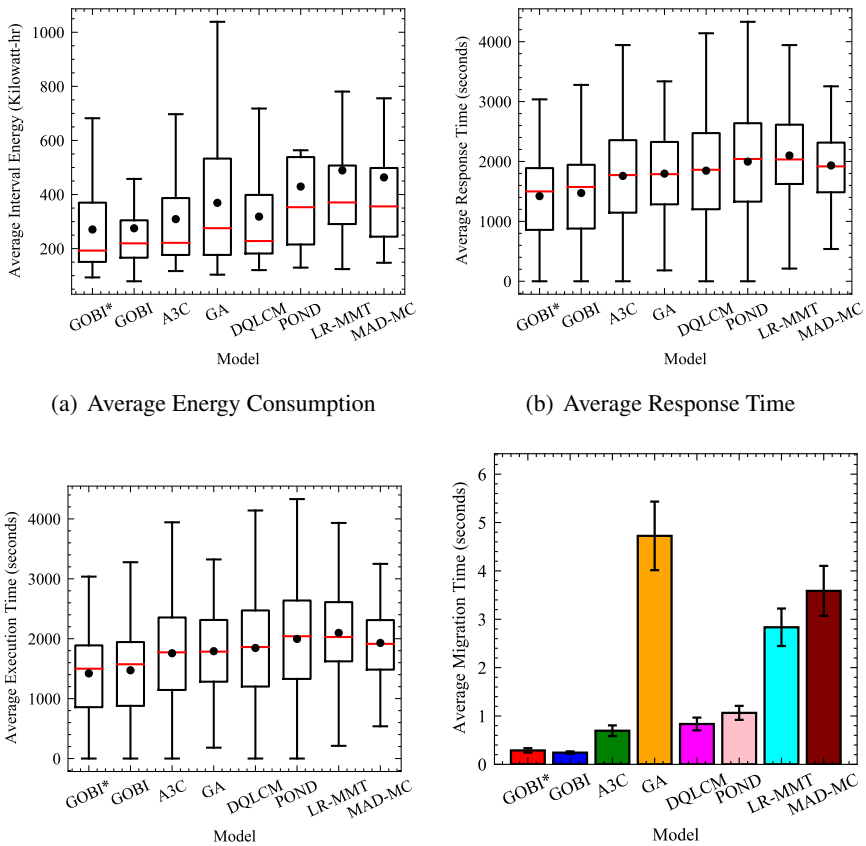


(g) Average SLO Violations (per application)



(h) Fairness

Figure 17. Comparison of GOBI and GOBI* against baselines on framework with 10 hosts



(a) Average Energy Consumption

(b) Average Response Time

(c) Average Execution Time

(d) Average Migration Time

Figure 18. Comparison of GOBI and GOBI* against baselines on simulator with 50 hosts

with similar energy consumption across all policies, *GOBI* and *GOBI** have a much higher task completion rate of 112-115 tasks in 100 intervals compared to only 107 tasks. Similarly, when we consider 50 host machines in a simulated platform (Figure 18(a)), *GOBI** still consumes 9.39% lower than *A3C*. The energy consumption with *GOBI* is very close to *A3C*.

Figure 17(b) shows the average response time for each policy. Here, the response time is the time between the creation of a task from an IoT sensor and the gateway that receives the response. Among the baselines models, *POND* has the lowest average response time of 266.53 seconds. Here, *GOBI* and *GOBI** have 226.04 and 156.09 seconds ($15.19\% - 41.43\%$ better than *POND*). For the three applications, the average response time for each policy is shown in Figure 17(e). Clearly, among the DeFog benchmark applications, Yolo takes the maximum time. The highest response time for Yolo among all policies is for *GA* as it prefers scheduling shorter jobs first *i.e.* Pocketsphinx and Aeneas to reduce response time. This is verified by the results which show that Aeneas has the lowest response time when the *GA* policy is used. Another disadvantage of this preference is that it leads to high wait times when running the *GA* policy. However, in larger scale experiments on 50 hosts in simulation (Figure 18(b)), *GA* has much lower wait times leading to the best response times among the baseline algorithms. Here, compared to *GA*, *GOBI* and *GOBI** have $17.98\% - 20.87\%$ lower average response times, respectively.

Figure 17(f) shows the average waiting time (in intervals) for tasks running on the Azure framework with each policy. As prior works have established that RL approaches are slow to adapt in highly volatile environments [69], [145], the *A3C* and *DQLCM* approaches are unable to adapt when a host is running at capacity. This means that for a large number of scheduling intervals (47.72%), they predict an overloaded host. Hence, the task cannot be assigned in the same interval and has to wait until either it is assigned to another host, or the resources of this host are released. This is also reflected in the wait time per application (Figure 17(f)). However, as *GOBI* and *GOBI** are able to adapt quickly to the environment, because of the neural model update in each iteration, they do not face such problems even on larger scale experiments.

For the experiments of physical hosts as shown in 17(g), as *POND* has the lowest response time, the SLO violation rate is also the lowest among the baselines (2.75%). *GOBI* has only 0.9% SLO violations and *GOBI** has none. This is due to the back-propagation approach which minimizes the response time, as well as avoiding local optima using the adaptive moment estimation approach (Adam optimizer). For *GOBI*, the SLO violations are only for Yolo tasks (4.5%). In the simulation environment, due to the high wait and response times of the *A3C*, *DQLCM* and *POND* schedulers, their SLO violation rates are highest among all methods. Here, *GOBI* has only 1.1% SLO violations. Due to the high response times of *A3C* and *DQLCM* for 50 hosts, the SLO itself is much higher giving the *GA* approach with low response times only 0.2% SLO violations. *GOBI**

has no SLO violations in this case as well. The main reason for low response times in *GOBI* and *GOBI** is their low scheduling time. Even with single-step simulation, the back-propagation based optimization strategy with the AdamW optimizer *converges faster than the gradient-free optimization* methods like *A3C*, *DQLCM GA* and *POND*.

Figure 18(d) shows the average migration times for all polices. Here, *GA* has the highest migration time due to the largest number of migrations. This is as expected due to the *non-local jumps* in the *GA* approach which lead to a high number of migrations. Approaches like the back-propagation method (*GOBI* and *GOBI**) and RL based approaches (*A3C* and *DQLCM*) have comparatively low migration times. Due to the convergence to better optimization, the execution times for iterative optimization approaches including *GOBI*, *GOBI** and *GA* are the lowest among all approaches (Figures 17(c) and 18(c)). Finally, Figures 17(h) show that *GOBI* and *GOBI** are the most fair schedulers.

However, running monolithic containerized applications in event-driven IoT architectures presents its own set of challenges. To optimize QoS metrics such as latency and energy consumption, we employed the concept of container migration. This involves moving application containers from overloaded hosts to underloaded hosts during the scheduling and placement phase. Instead of relocating entire application containers, it would be more beneficial for the individual data operations service hosted on the hosts and routed based on QoS requirements, which is more beneficial in resource-constrained edge/fog architectures. Over experiments show that scheduling decisions definitively can yield optimal results; however, the migration approach needs extra bandwidth to move the entire encapsulated container application between nodes, which is challenging in edge/fog environments. Other challenges related to granular scaling of data operations, not all application containers, and billing based on data operation execution provide more benefits in even driven architectures. These challenges can be streamlined by unpacking a monolithic application into granular data operations by designing data pipelines along with serverless technology. Some key advantages of serverless-based data processing over monolithic containers are:

1. Resource management: Containers often required a level of infrastructure management; for example, we demonstrate the scheduling and placing of containers on a set of hosts for optimal resource usage and latency requirements. However, serverless computing fully abstracts the responsibility from the developers.
2. Scalability: Containers needed intervention to scale the more instances of containerization application, where serverless functions are built to scale on demand based on the incoming requests or load.
3. Code Reusability: Containers consolidate multiple data operations or processing logic for an entire application, while serverless architectures break down functionality into granular-level functions. These functions can be

reused across multiple applications or in various data processing scenarios.

4. Support for event-driven architecture: Containers support event-driven with external management to control the trigger and action, which become complex in the container environment. However, serverless applications are itself event-driven and react to events or triggers. This encourages the creation of reusable decoupled components that respond to specific events.

3.6. Summary

We have presented a coupled-simulation approach to leverage simulators to predict the QoS parameters and make better decisions in a heterogeneous fog setup. The presented COSCO framework allows for the deployment of a holistic platform that provides an easy-to-use interface for scheduling policies to access simulation capabilities. Moreover, we have presented two scheduling policies based on back-propagation of gradients with respect to input, namely, GOBI and GOBI*. Comparing GOBI and GOBI* against state-of-the-art schedulers using real-world fog applications, we see that our methods are able to reduce energy consumption, response time, SLO violations, and scheduling time. Between GOBI and GOBI*, GOBI* gives better QoS values; however, GOBI is more suitable for resource-constrained servers.

The primary objective was to explore the use of monolithic container applications to efficiently schedule tasks in fog environments through the container migration strategy. However, the proposed system has limitations in accommodating workflow models for IoT data processing, which can become complex to manage within monolithic containers. The system aims to decrease scheduling time, without emphasizing fine-grained autoscaling, increase productivity, and improve flexibility of Iot data processing.

4. DESIGNING SERVERLESS DATA PIPELINE FOR IOT DATA PROCESSING

This chapter examines the use of serverless data pipelines for IoT data processing. The objective is to investigate the various approaches in designing serverless data pipelines using different intermediate data storage units (off-the-shelf tools, object storage, and message queues) in the pipelines. We investigate the performance of such approaches using real-time IoT applications (video processing, Aeneas, and PocketSphinx) described in Section 2.6 and then provide a suitability analysis for IoT developers to choose the appropriate SDP according to their type of application. The overall contribution of this chapter provides answers to Research Question 2 (RQ2).

4.1. Introduction

Nowadays, data pipelines are widely used for data processing, especially in the context of machine learning-based data analytics for Internet of Things (IoT) applications. Data pipelines are constructed using data pipeline engine or tools often configured on cloud or on-premise services with huge amounts of resources. For example, using large and expensive data processing clusters (e.g. Apache Spark, Flink, Storm) [12] may not be optimal because many IoT applications are event-driven and require performing actions in real-time [11]. As we discussed in background section about three tier architecture and edge/fog based IoT data processing, in such scenarios, the use of resource hungry data pipeline mechanisms poses a challenge due to the allocation of excessive resources, surpassing the actual demand. Considering the design and performance bottlenecks of container based monolithic applications, described in the Chapter 3, this Chapter discusses the use of serverless approach.

Leveraging serverless computing simplifies the design of event-based, real-time, and scalable IoT data processing [13]. By integrating serverless modelsc along with data pipelines, it becomes feasible to develop multilayer (edge, fog, cloud) IoT applications, offering significant advantages in both design and deployment in multilayer environments. Here, serverless functions are used to create pipeline tasks (data operations) and are invoked as the data move through the pipeline. This provides an advantage in deploying edge, fog and cloud functions, while data pipeline technologies can be used for data transport, routing, and function invocation.

However, there are challenges in using the SDP model for IoT data processing that need to be investigated. Heterogeneity in hardware resources available at the edge, fog, and cloud presents an interesting challenge in terms of how to allocate and prioritize data flow between functions located at the fog and cloud to meet the desired Quality of Service (QoS). Furthermore, serverless functions are

stateless, and their frameworks only manage the runtime of functions, completely separating them from data management [21]. This separation simplifies serverless computing, but has drawbacks for data-intensive and stream processing pipelines [22], which can pose challenges when dealing with intermediate data between functions of the pipeline.

The heterogeneity in hardware resources available at the edge, fog and cloud presents an interesting challenge in terms of how to allocate and prioritize data flow between functions located at the Fog and Cloud to meet the desired Quality of Service (QoS). In addition, there are other issues that must be taken into account and evaluated. Serverless functions are stateless and their frameworks only manage the runtime of functions, completely separating them from data management [21]. This separation simplifies serverless computing, but it has drawbacks for data-intensive and stream processing pipelines [22], which can pose challenges when dealing with intermediate data between functions in the pipeline. Data movement between functions residing at the edge and cloud is often handled by using object storage services like AWS S3. However, challenging when a large set of functions are deployed in edge/fog infrastructure and data needs to be transferred on each function invocation. The object storage may yield higher charges when more data and more function invocations occur. Even though object storage attains the purpose of handling intermediate data, cost, latency, etc., are challenging.

There also exist off-the-shelf DP tools like StreamSet and Apache NiFi, which provide some support for edge/fog environments and can also be utilized to solve the issues, but they usually manage the flow of data in a more centralized manner and often require significant computing resources to run effectively. Alternatively to object storage, its also possible to use data brokers (e.g. Apache Kafka, MQTT) as Message Queues between serverless functions for designing serverless data pipelines. Compared to object-storage, they would require less storage and may be faster due to more extensive memory usage, which is highly desirable in edge/fog environments. However, compared to NiFi, it may be more difficult to control the precise execution flow of pipelines.

Public cloud service providers such as *AWS greengrass* [161], *Google Cloud IoT* [157] and Microsoft- *Azure IoT Edge* [162] have typical IoT data pipeline solutions for industrial, healthcare, smart city and other real-time use cases. For example, consider AWS IoT Greengrass, where the Lambda service will be executed at the edge layer for data acquisition and pre-processing. Later data is forwarded to the cloud by edge devices, and then it passes through pipeline of activities for post processing, and finally is delivered to the data sink.

Valeria et al. proposed a solution of IoT data stream processing in distributed fog and edge computing environments with decentralized scalable manner [129] and further extended to how data processing operators were placed in computing nodes considering the efficiency, application topology and resources configurations [167]. These works provide hint that off-the-shelf data stream processing

tools such as Apache Storm can be used for the task. However, these stream processing tools require huge computing clusters and in IoT deployments more often devices are heterogeneous with limited computing capacity. More often IoT workloads are event and time driven which motivates us to investigate the serverless based data processing pipelines. Further, SDPs easily been deployed at various levels in the IoT hierarchy (edge, fog and cloud Infrastructure) with efficient granular scaling of the serverless functions.

Das et al. [153] proposed a model for efficient execution of user tasks as serverless functions in edge/cloud environments and designed a set of data pipelines using AWS Greengrass on edge devices along with Lambda capabilities. Our approach looks similar to this model, however it lacks fog based processing pipeline model.

Dehury et al. [154] designed a framework known as CCoDaMiC, which aims to ensure data accuracy, trustworthiness, and validation in SDP. This work directly relates to our proposed DFT based SDP approach. However, CCoDaMiC mainly focuses on data accuracy and trustworthiness and not on the performance of the applications. Lixiang et al. [155] designed a framework for video processing using serverless lambda functions known as *Sprocket*. Authors demonstrated the efficiency of serverless functions for faster execution by constructing pipeline of activities for video handling. However, their primary focus is to reduce latency and cost by using the techniques of parallelism. Interestingly, this work motivated us to consider the complex video processing use case in our proposed research.

Several techniques and methods have been proposed that illustrate the use of MQTT for data acquisition from different data sources using the publish / subscribe model [163], [165], [166]. MQTT brokers can act as data carriers and can store data until subscribers consume it. This approach is well suited for storing temporary or intermediate data between processing elements in SDP. In our work, one of the approaches uses MQTT together with serverless framework to construct data pipelines from data source to sink.

Taking into account the challenges mentioned above and in the Introduction Chapter 1 Subsection 1.1, it motivates us to investigate the advantages and disadvantages of different mechanisms by answering Research Question 2 (RQ2). The following are contributions to this chapter.

- We illustrate how SDP can be deployed in a three-tier IoT architecture.
- We propose three approaches for designing Serverless Data Pipelines with three data handling mechanisms (Apache Nifi, Message Queues and object storage services such as AWS S3).
- We use real-time fog computing workloads, such as Aeneas, PocketSphinx, and custom video processing applications, to compare the performance (e.g. processing time) and resource utilization of various SDP approaches.
- We offer insights on the suitability of these SDPs for different types of fog computing workload.

- We also describe insights on public cloud service provider solutions to design serverless data pipelines.

The remainder of the chapter is organized as follows. Section 4.2 provides an SDP architecture in a three-layer IoT system. Following this, three novel SDP approaches are designed, articulated, and implemented for real-time fog computing use cases in Section 4.3 and compared with different performance metrics in Section 4.4. Finally, the concluding remarks and future work are discussed in Subsection 4.5.

4.2. System Architecture

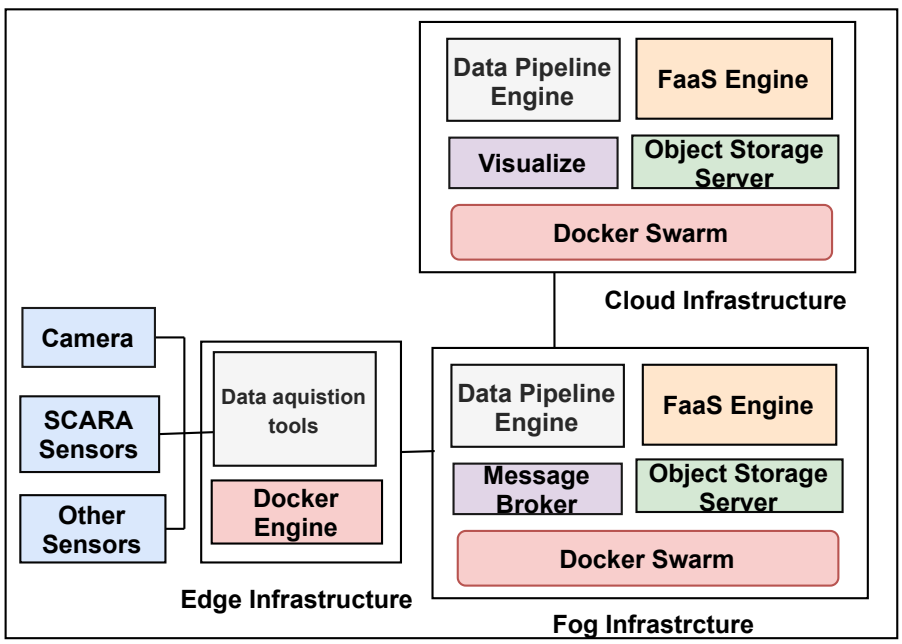


Figure 19. Proposed SDP architecture

We present a system architecture that consists of the required software services to handle the flow and execution of data in a pipelined manner, as shown in Figure 19. Data are generated by IoT devices (such as surveillance cameras, SCARA (Selective Compliance Assembly Robot Arm) robot sensors, and health monitoring sensors) and are eventually sent to edge/fog infrastructure for processing and then to cloud for storage. The following are typical functions of each layer:

4.2.1. Edge Infrastructure:

The edge layer is primarily focused on receiving data from IoT devices and performing preprocessing operations. For this purpose, we use data acquisition tools or software solutions such as MiNiFi, custom services such as Python, or other

runtime services, as shown in Figure 19. The use of MiniFi or Python services depends on the data pipeline mechanism used to construct the SDP. For example, in Apache NiFi based SDP, MiniFi is used. The detailed description of the services used in edge infrastructure is described below.

- *MiniFi* [1]: Apache MiNiFi is a super light-weight version of NiFi made for the edge devices. It can run as a system service, and it is centrally managed using Apache NiFi. Developers can easily design the pipelines using a set of processors in Apache NiFi and push them into the MiniFi service. These pipelines can handle preliminary data operations near to the source, e.g., compressing a video recorded by drone before sending to cloud/fog to reduce bandwidth consumption, etc. We use this MiniFi service in the implementation of DFT tool based SDP as described in 4.3.1.
- *Custom services*: Custom services are similar to the MiniFi processors. Such services need to be created from scratch using a specific programming language. E.g., A Python program can be created to collect and compress the video. Python-based custom services are created and used in the implementation of OSS and MQTT-based SDP as described in 4.3.2 and 4.3.3, respectively.

4.2.2. Fog Infrastructure

Fog Infrastructure is mainly responsible for processing the data received from Edge infrastructure, for which, Data pipeline engine (Apache NiFi[2]), FaaS engine (OpenFaaS[3]), MinIO[4], and MQTT[5] services are used. The fog infrastructure includes a group of fog servers deployed in a cluster with a set of particular software services using Docker Container Engine. The processed data are then forwarded to the cloud infrastructure to further process, store, and generate alerts and notifications. The use and necessity of software services are described below:

FaaS Engine: FaaS Engine is primarily one of the core components of this proposed work. OpenFaaS serverless platform is used in this work, as it is lightweight and easy-to-configure over other alternative solutions. OpenFaaS can be installed atop of Docker or Kubernetes platform. Docker containers are used to host and execute the serverless functions. The functions can be invoked using HTTP endpoints with the necessary data. The function invocation is performed in the pipeline by Data pipeline engine, MinIO event notification system, and MQTT event notification service.

Data pipeline engine: As discussed in Section 4.1, we use Apache NiFi, a data pipeline processing platform, which manages the data flow between the systems. This provides a set of independent processors with specific functionalities

[1] https://nifi.apache.org/minifi/
[2] https://nifi.apache.org/
[3] https://www.openfaas.com/
[4] https://min.io/
[5] https://mqtt.org/

to process and manage the data. The data flow between processors is managed via scalable queues. Developers can easily design custom data pipelines using a flexible user interface and automatically configure, control, and deploy the pipelines in Edge infrastructure using MiniFi service. This MiniFi-Nifi integration efficiently manages the orchestrated IoT data processing from the edge, fog, and cloud, and vice versa.

Message Queue: Message queues are published/subscribe protocol-based data carriers between source and sink. We demonstrate the use of Message Queues to build data pipelines integrated with the OpenFaaS serverless platform. A lightweight messaging protocol, MQTT, is ideal for small sensors and mobile devices and is suitable for high-latency or unreliable networks. MQTT uses different data types such as UTF-8 encoded string, bit/byte integer, binary data, and UTF-8 string pair. A serverless function can publish the processed data to MQTT, and in turn, functions are invoked when data need to be subscribed using web hooks. The flow of data between MQTT and the serverless platform builds consistent and reliable SDP.

Object storage service: An open source cloud-based storage solution, MinIO, compatible with Amazon S3, stores IoT data. This provides a RESTful API to access/insert/remove buckets and objects. Moreover, triggers are set to bucket when its content is accessed/written/removed, and corresponding event notifications are generated using techniques such as web-hooks and Message Queues. This is advantageous in IoT applications for handling event-driven data. It is configured as high-availability cluster using docker swarm.

4.2.3. Cloud Infrastructure

Cloud infrastructure is mainly responsible for processing heavy computation data received from Fog infrastructure and storing the data. This is also responsible for generating alerts and notifications to activate other business processes whenever required. To perform this, a set of services from cloud providers or user-configured open-source services are used, such as Data pipeline engine (Apache NiFi, AWS data pipeline, Google Cloud pipeline, etc.), Object storage service (MinIO/AWS S3/ Google object storage), Message queues (MQTT/AWS SNS) and Faas engine (OpenFaaS/AWS Lambda/Google Functions).

The setup and configuration of the FaaS engine, the data pipeline engine, and the object storage service are the same as those of the fog infrastructure. Along with this, the cloud infrastructure is also responsible for Visualization and Reporting. The primary job of visualization is to display processed data using visualization tools. The Grafana visualization tool is used to measure Edge/Fog and cloud nodes' performance metrics in this work. Additionally, the Prometheus time series database is used where performance metrics are collected.

4.3. Serverless Data Pipeline (SDP) approaches

Considering the general architecture mentioned above and the challenges mentioned in Chapter 1 in Section 1.1 and in this chapter in Section 4.1, the serverless data pipelines are designed using different approaches. In this section, we introduce three approaches: (a) Off-the-shelf data flow tool (DFT) based SDP, (b) Object storage service-based SDP, and (c) MQTT-based SDP. Furthermore, we implemented the proposed SDPs for real-time IoT use cases as described in Section 2.6.

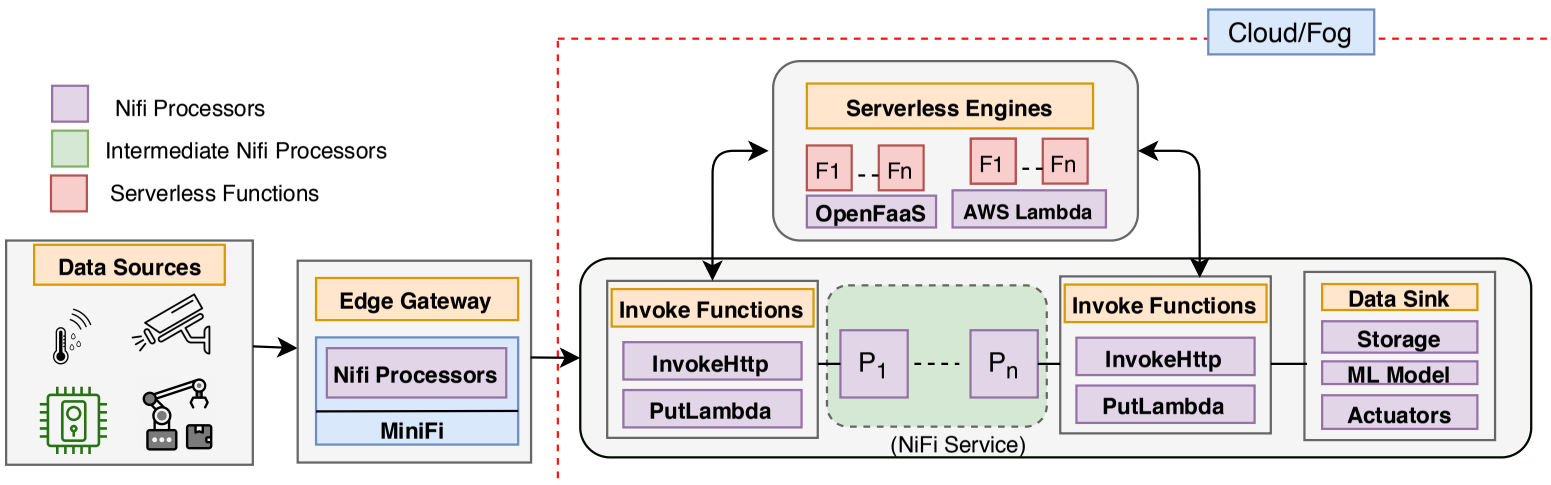


Figure 20. DFT based SDP approach using Apache Nifi and OpenFaas.

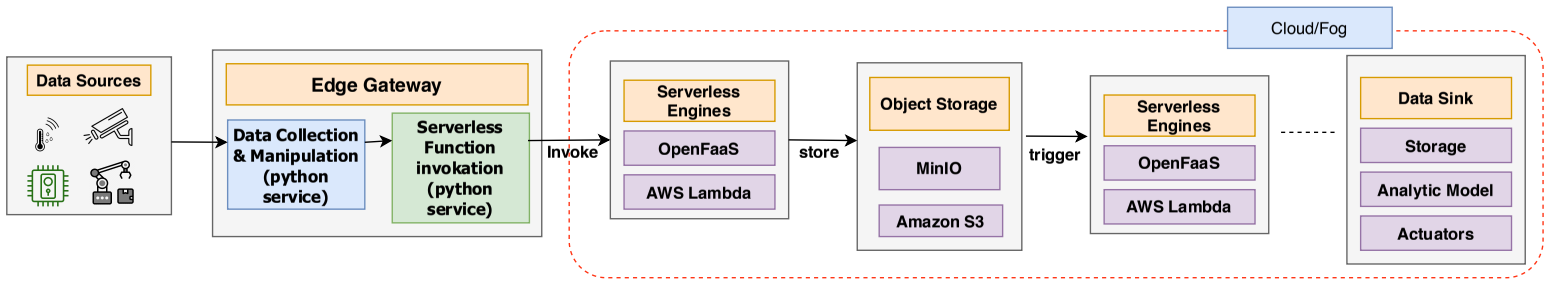


Figure 21. OSS based SDP approach using MinIO.

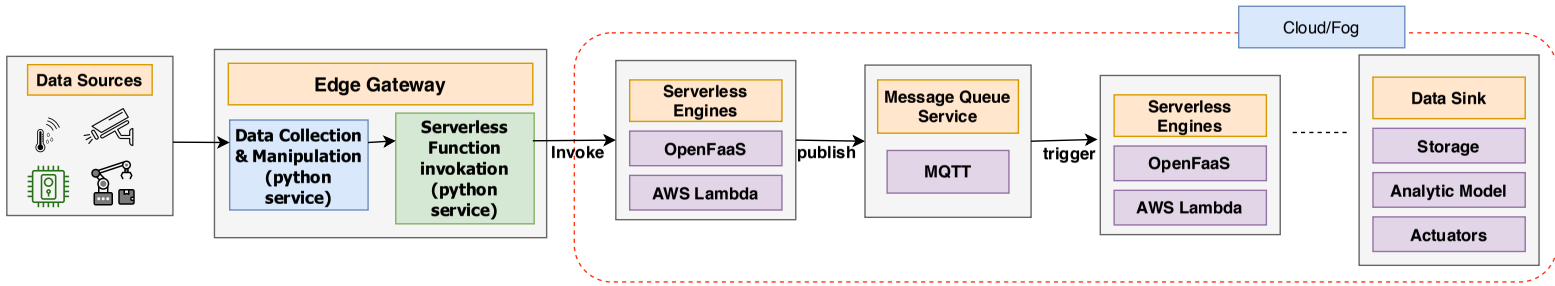


Figure 22. MQTT-based SDP Architecture

4.3.1. Off-the-shelf data flow tool (DFT) based SDP

In DFT based SDP, the developer does not need to maintain the queue, rather we use the queuing capability of Apache Nifi, which manages the centralized management of data and with integration of a serverless platform.

In this SDP approach, we have used Apache MiniFi in the Edge infrastructure to receive and preprocess the data, as shown in Figure 20. As discussed in Subsection 4.2, MiniFi service can run on devices with restricted resources and is managed autonomously using the central Apache NiFi from Fog Infrastructure. Here, sensed data from IoT devices is received into MiniFi using processors such as *ConsumeMQTT* or *ListenHTTP*, based on the communication protocol used between edge infrastructure and IoT devices. Some data preprocessing processors, such as data compression, filtering, and aggregation, are also configured but with limited computing resources and capabilities. MiniFi does not have a GUI and is developed with Java/C++ libraries and can be quickly started as a system service. The data flow with processor groups is designed in Apache NiFi and is automatically pushed into the MiniFi service. MiniFi performs the data operations and pushes the flow file containing the data to the Apache NiFi service configured in the fog.

Apache NiFi is used to handle data flow in fog and cloud infrastructure. Apache NiFi provides a flexible set of processors for data operations and integration between cross-platform systems. This gives the capabilities to seamlessly integrate the serverless platform through specific Apapche NiFi processors, such as *InvokeHttp*, *PutLambda*, etc. Such multiple NiFi processors can be connected to others, allowing the developer to invoke multiple serverless functions. The key benefit is that Apache NiFi facilitates queued data that can lend back pressure when limits are attained during data flow processing. Another benefit is that it has priority queuing to set single/multiple prioritization schemes that dictate how data is retrieved from a queue. This allows the developer not to pay much attention to maintaining or implementing the queue between each pair of serverless function invocations.

For the serverless platform, OpenFaaS is used in both fog and cloud infrastructures. However, public cloud serverless services can also be used in the cloud infrastructure such as Amazon Lambda. The serverless function receives the data flow file as input from *invokedHTTP* request and sends the processed data as an HTTP response body. Every serverless function is invoked using an HTTP endpoint with respective HTTP methods (POST or GET). We implemented the DFT based SDP using real-time IoT use cases as described in the following paragraphs.

In **Custom video processing application**, the Drone sends the video footage to the edge node. At the edge node, the MiniFi service is configured with *GetFile* MiNiFi processor to read the video file from disk and forward the video to Nifi in the fog node. The Nifi in fog infrastructure consists of three main processors to invoke three different OpenFaaS serverless functions (FFmpeg, Yolo, convert-

Tojson). On the other hand, the Apache NiFi in cloud infrastructure consists of multiple processors to store the data (received from Fog infrastructure) into MinIO bucket.

Similarly, in the **Aeneas** application the end-user mobile device sends an audio file to the edge node. At the edge node, the MiniFi service is configured with *GetFile* MiniFi processor to read the audio file received and compressed using gzip processor and forward it to NiFi configured in the fog node. The NiFi in fog infrastructure consists of three main processors, first is the decompress processor, second is to invoke two different OpenFaaS serverless functions (aeneas, getFile(.xhtml)), and third is creating JSON from the output. Finally, these data are stored in the MinIO bucket using multiple NiFi processors.

The **PocketSphinx** has similar implementation as above, but Nifi in fog infrastructure consists of three main processors; first is the decompress processor, second is to invoke two different OpenFaaS serverless functions (pocketsphinx, text processing), third is creating JSON from the output.

4.3.2. Object Storage service based SDP

In an OSS based SDP, the object storage service will resemble a persistent queue where the developer can visualize the data stored on the storage server as a queue. The capability of the object storage service is to trigger an event notification using webhooks, making the integration of object storage with the serverless platform flexible. In this approach, we have used the Python service on the edge infrastructure to receive, pre-process, and transport data from the IoT devices to the fog infrastructure, as shown in Figure 21. Here, the Python *requests* library is used to invoke serverless functions from the edge Python service to functions that reside in fog.

MinIO is used to handle data flow in fog and cloud infrastructure. MinIO is an open-source, scalable high-performance storage service, as described in Section 4.2. The flexible bucket notifications are a set of events such as inserted, accessed, deleted, and copied. The corresponding events are triggered using web hooks. This flexibility allows for the seamless integration of storage service and serverless platform invocations. Apart from this, the MinIO Python client library makes it easy to code the functions to access the object data from a specific bucket.

For the serverless platform, OpenFaas is used in both fog and cloud infrastructure. Here, the serverless functions are invoked from the gateway node with data or from the events triggered in MinIO buckets. Furthermore, the serverless function may store processed data into buckets using the MinIO client library. Again, events may trigger to invoke functions and continue until the fog node forwards data to cloud infrastructure. In the cloud infrastructure, data flows in a similar fashion over MinIO and OpenFaas serverless platform. This constitutes an SDP, where object storage with persistent mode acts as an intermediate data handling mechanism between serverless functions. The following paragraph will describe

the use of OSS based SDP in designing the real time use cases.

In **Custom video application**, we use a Python service as a drone simulator to send the video file to the gateway node. The gateway node is configured with a Python service to read a video file and send it to the fog node for processing. In fog node, MinIO is configured with two buckets: (a) *unprocessed* to store raw images and (b) *processed* to store processed video in JSON format. These buckets are set with web-hook event notifications to trigger serverless functions when a new data object is inserted. This implementation uses five serverless functions, as given in Table 2. However, in **Aeneas application**, in fog node, MinIO is configured with two buckets: (a) *raw-audio* to store raw images and (b) *syncmap* to store synchronization map generated to audio file in JSON format. This implementation uses six serverless functions, as given in Table 2. Finally, the **PocketSphnix application** in the fog node, MinIO is configured with two buckets: (a) *raw-pocketsphinx* to store raw images and (b) *processed-pocketsphinx* to store the converted audio file to text, (c) *output-pocketsphinx* to store text-processed data. This implementation uses 6 serverless functions, as given in Table 2. In the cloud infrastructure, two MinIO buckets are created, (a) *success-pocketsphinx-* to store success results from text processing, (b) *failure-pocketsphinx* to store failure results from text processing. These buckets are responsible to store the processed audio files and output is mainly in text format.

4.3.3. MQTT-based SDP

In this proposed SDP approach, MQTT is used as a queue to store the data, different queues are represented by different MQTT topics and serverless functions can be triggered when new data objects are published into a specific topic. The gateway node receives data from IoT device using the custom Python service (Python code is written to perform specific operation). The received data are preprocessed and published to the MQTT broker with a topic name, as shown in Figure 22.

The topic names need to be subscribed by the OpenFaaS serverless functions, to consume the data in the queue. For this, Serverless frameworks should be built in with connectors between itself and message broker to subscribe the topics and invoke corresponding functions. Apart from this, MQTT does not have the capability to directly trigger an HTTP endpoint upon the arrival of new data. Thanks to the OpenFaaS community for developing the *openFaas-mqtt* connector, which runs as a service to invoke serverless functions by subscribing to MQTT topics. OpenFaas has multiple connectors supported for different Message Queues. The serverless function processes the data and publishes the output again to the Message Queue. This process continues until the data from the source reaches to the data sink, as shown in Figure 22. The following paragraphs describe how the use cases were implemented using the MQTT based SDP.

Similar to OSS based SDP approach, all the three applications are configured with python service in the edge node. In **Custom video processing application**,

the Python service publishes the video to MQTT broker with topic name. The *openfaas-mqtt* connector running in fog node subscribes to the topic name and invokes the functions. Here, we use three serverless functions, as given in Table 2 and one *openfaas-mqtt* connector service. Similarly, in the **Aeneas application** the Python service publishes the audio file to MQTT broker with topic name. The *openfaas-mqtt* connector running in fog node subscribes to the topic name and invokes the functions. Here, we use three serverless functions, as given in Table 2 and one *openfaas-mqtt* connector service. In **PocketSphinx application**, we use three serverless functions, as given in Table 2 and one *openfaas-mqtt* connector service.

For calculating the evaluation metrics in Section 4.4, we represent storage units as a set $S = \{S_1, S_2, \ldots, S_k\}$. The storage unit could be processors in NiFi, MQTT queues or MinIO buckets. Next section will describe about experiment details.

4.4. Experiment and results

All the proposed SDP approaches are implemented on three applications as described in the Chapter 2 Section 2.6. Further, the goal is to measure the performance w.r.t metrics to understand and investigate efficiency of those SDPs on various applications (text, audio, video and image applications). In the following section, we will discuss the metrics used to measure the performance and analysed the results, further outlined the experience on the SDP implementation and provided future directions.

4.4.1. Performance metrics

In this subsection, we will describe various performance metrics along with their mathematical formulae. The resource utilization metrics are measured cumulatively on all the three layers of infrastructure that consists of utilization of edge (Gateway) resources, fog resources, and cloud infrastructure resource. Here, concurrent user requests are generated in the sensor nodes in all the three use cases with corresponding data such that it mimics the real-time application. Monitoring software solutions such as *Prometheus* and *Node Exporter* are used to collect such metrics. *PromoQL* (Prometheus Query Language) is used to calculate the metrics for a specific time period.

- **Processing Time:** In IoT environments, computation time and latency are very crucial. In this regard, the SDP *processing time* is directly proportional to response time of the user requests and therefore we note that these metrics as native pipeline performance metrics. Processing time is measured in seconds, which is defined as the total time taken to process a data in a pipeline from source to destination. The source is a sensor node and sink is a storage/other end point in Cloud Infrastructure as described in Section 4.2. The processing time is addition of both **communication** and **computation**

Table 5. List of Notation.

Notation	Description
P	Computation time
C_T	Communication time
F	Set of serverless functions
R	Set of n number of concurrent users' requests $R = \{r_1, r_2, \ldots, r_n\}$
D	Duration of the user request from source to destination (data sink)
D_{at}	Timestamp recorded at arrival from the source
D_{ct}	Timestamp recorded at destination
DAT	Disk Access Time
NCT	Network Communication Time
S	Set of intermediate storage units $S = \{S_1, S_2, \ldots, S_k\}$
DU_{at}	Timestamp recorded when data unit arrived in the storage unit
DU_{dt}	Timestamp recorded when data unit departed from the storage unit

time (latency). In the below paragraphs, we formulate the mathematical equations used to calculate these metrics. These metrics are calculated using logs of MinIO, MQTT, Apache NiFi and OpenFaaS gateway.

Computation time : The computation time is calculated as summation of time required to compute a data unit by individual processing units (serverless functions) in a data pipeline. Let R be the set of n number of concurrent users requests $R = \{r_1, r_2, \ldots, r_n\}$. In our experiments the value of n is considered from 10 to 300 and each user request carries a data unit to be processed by serverless function in the pipeline. The computation time of individual i^{th} user request is

$$P(r_i) = \sum_{\forall f_j \varepsilon F} P(r_i, f_j) \tag{4.1}$$

where $F = \{f_1, f_2, \ldots, f_m\}$ is a set of m number of functions and $P(r_i, f_j)$ represents the time taken by the function $f_j \in F$ to execute or process the request $r_i \in R$.

Communication time: The communication time denoted as $C_T(r_i)$ of i_{th} user request is the time required to move data unit from source to sink in pipeline excluding the computation time. It is the summation of **disk access time** (for intermediate storage units) and **network communication time.**

Let $D_{at}(r_i)$ be the arrival time at source and $D_{ct}(r_i)$ be the completion time of the i^{th} user request at sink. The total duration of serving the user request $D(r_i)$ is measured as

$$D(r_i) = D_{ct}(r_i) - D_{at}(r_i) \tag{4.2}$$

From equation (4.1) and equation (4.2), the communication time is calculated as

$$C_T(r_i) = D(r_i) - P(r_i) \tag{4.3}$$

Further, In the SDP approaches the data units are stored in the intermediate storage units and served to serverless functions for processing. The total time that data unit resides in the storage unit is considered as **disk access time** denoted as $DAT(r_i)$ is inclusive of time required to store and access the data units. Let $DU_{at}(r_i, S_j)$ and $DU_{dt}(r_i, S_j)$ be the arrival time and departure time respectively of the i^{th} user requests' data unit in the storage unit $S_j \in S$. The total duration of disk access time is calculated as

$$DAT(r_i) = \sum_{S_j \in S} DU_{dt}(r_i, S_j) - DU_{at}(r_i, S_j) \tag{4.4}$$

where $S = \{S_1, S_2, \ldots, S_k\}$ is a set of k number of storage units.
Finally, **network communication time** denoted as $NCT(r_i)$ is the summation of time required to move a data unit (of a user's request) in network from user's device to the sequence of processing units (serverless function) and intermediate storage units until the final data sink. Now from the equation (4.3) and equation (4.4) network access time is calculated as

$$NCT(r_i) = C_T(r_i) - DAT(r_i) \tag{4.5}$$

- **Average CPU utilization:** The average CPU utilization is measured in percentage (%) and is calculated over time period from pipeline invocation till the data is received in the final destination.
- **Average memory utilization:** It is measured in percent (%) and is calculated as sum of total free memory, cache memory, memory in buffer and divided by total memory. Similarly, average disk utilization is measured in percentage (%).
- **Network received:** This is calculated as bytes per second and is calculated as sum of bytes received on the network over a period of time.
- **Network transmitted:** It is calculated as bytes per second and is calculated as sum of bytes uploaded on the network over a period of time.
- **Disk I/O Read:** This is measured in kilobytes and is calculated as sum of bytes read from file system over a period of time.
- **Disk I/O Write:** It is measured as bytes per second and is calculated as sum of bytes written in to the file system over a period of time.

4.4.2. Experimental Setup

The Docker Container Engine v19.03.12 is installed in both fog and cloud infrastructure in swarm mode. The OpenFaaS serverless platform is used as a FaaS engine configured in fog and cloud. OpenFaaS functions are developed using the programming language templates (bash streaming and Python 3.7). OpenFaaS

Table 6. Hardware configuration for experimental setup

Device name	Configuration (Processor, RAM)	Quantity	Node type
RPi 4B model	Quad-CoreCortex A72, 4GB LPDDR4	2	Fog node
RPi 3B model	Quad-CoreCortex A53, 4GB LPDDR4	1	Gateway node
Virtual machine	4-Core, 8GB DDR4	1	Cloud node
Minix Neo Z64-W10	Quad Core Z3735F (64 bit), 2GB DDR3	1	Fog node
Router	Inteno DG200 model with 1000Mbps full duplex	1	Network layer

command line interface (CLI) is used to build and deploy the functions into the OpenFaaS gateway. The Apache NiFi v1.3.2 is used in both fog and cloud as container service. The Apache NiFi user interface is used to design the data flow and monitor the flow files. The MinIO is deployed using docker compose service and volumes are mounted in the host machines. The MinIO client is used to create and configure the settings for event notifications on bucket.

A set of hardware devices and cloud resources are used to deploy and setup application services, as shown in Table 6. Three Raspberry Pi 4B models and MiniX NEO Z83-4U Intel Mini PC are used for setting up fog infrastructure. For cloud infrastructure, the virtual machines of size *m2.medium* with vCPU and 8GB RAM resembling similar capacity as AWS are provisioned from the University's private OpenStack cloud. The Raspberry Pi 3B model is used as a gateway node, and all the edge and fog devices are connected in a LAN with 1000 Mbps network bandwidth using Inteno DG200 router. The fog devices are connected to cloud services via 1000 Mbps network bandwidth. The network setup used for interconnection between edge, fog and cloud environments are dedicated to these experiments.

Upon setting up of necessary hardware and application services, the use cases (Aeneas, PocketSphinx and custom video processing) are deployed. The corresponding performance metrics are measured, and results are discussed in below subsections.

4.4.3. Results and Discussion

We considered scaling the number of users as a parameter to measure the performance of the approaches because the rate of concurrent arrival of user requests heavily impacts the pipeline performance. To measure the performance of all the metrics, several users are scaled from 1 to 15 for video applications (we used a chunk of video file as one user request) and 10 to 300 for Aeneas, PocketSphinx applications, and the corresponding SDP performances were measured. However, for calculating the processing time we considered 100 users in Aeneas and PocketSphinx due to data units were started dropping in MQTT based SDP.

Performance metrics observed with Aeneas application. The processing time was studied in all the three SDP approaches, as shown in the Figure 23. Here, the y-axis represents processing time in seconds, and the x-axis shows the # of users. The OSS requires a maximum of 540*s* to complete the 100 users requests, whereas MQTT based SDP and DFT processed in 330*s* and 324*s*, respectively.

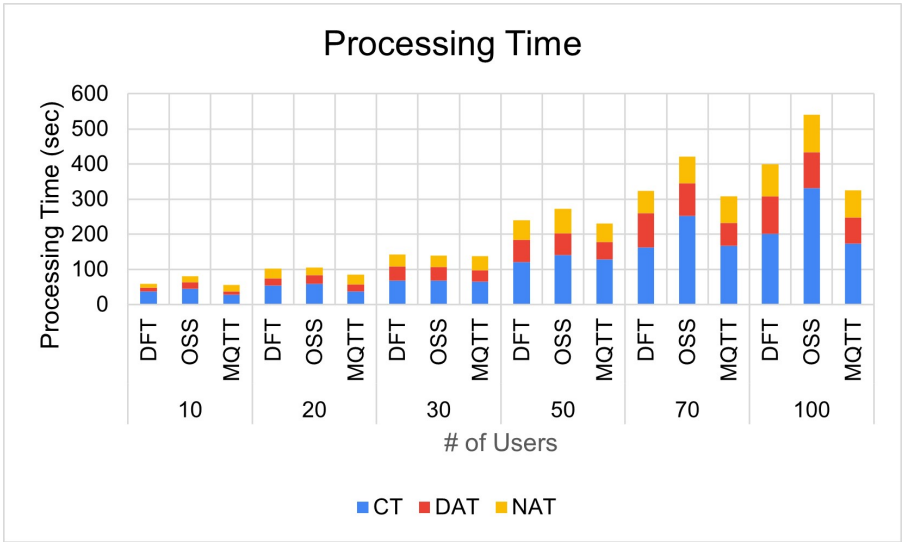


Figure 23. Aeneas application - Processing time measured in seconds

The OSS had more processing time as the number of users increases, because MinIO notification invocations are synchronous. This pipeline had around five serverless functions as shown in Table 2 and two functions were extra to facilitate for retrieving the object data from MinIO buckets and this can lead to extra processing time. The computation time in OSS and MQTT based SDP was higher, where as disk access time was more in DFT. The internal queues in DFT manages efficient flow of data that makes to stay the data units in the queue that increase the DAS time. In OSS and MQTT based SDP, events were triggered as the data units arrived in to storage units which makes openfaas gateway to push these asynchronous user requests to NATs queue that increases the overall function execution time. The average computation time was highest in OSS with 351s but DFT had lesser computation with more disk access time of 280s.

The average CPU utilization and Memory utilization were measured, as shown in the Figure 24. The primary vertical y-axis shows an average CPU utilization measured in percentage (%) and the secondary y-axis shows the Memory utilization. The DFT consumed highest CPU of 36%, whereas MQTT based SDP consumed a lesser CPU of 21.06% and OSS had moderate CPU utilization of 31%. But MQTT based SDP started using more CPU after 300 users request and further the data units in the pipeline started dropping.

The Object Store and MQTT-based SDP used more number of lightweight python-based serverless functions, and DFT had a higher CPU utilization due to the set of Apache NiFi processors used in the pipeline that require extra computation power apart from serverless functions.

The average Memory utilization at the secondary y-axis is measured in percent (%). The MQTT-based SDP approach has the highest memory usage footprint of an average 45.92%, whereas DFT and MQTT based data pipelines used an

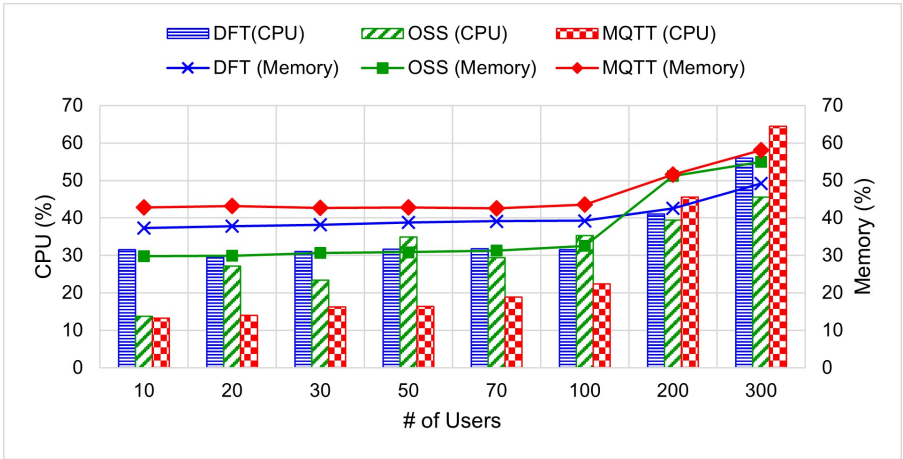


Figure 24. Aeneas application - Average CPU and Memory utilization

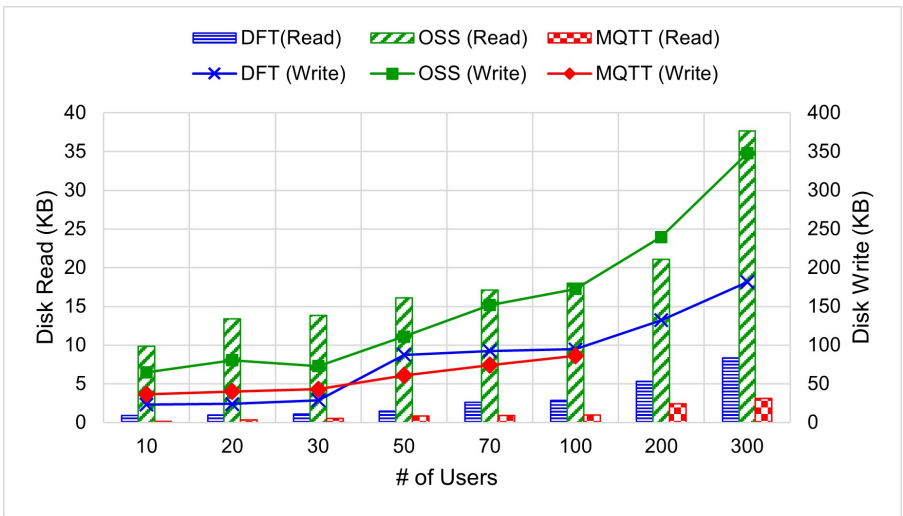


Figure 25. Aeneas application - Average Disk Reads and average Disk Writes measured in Kilobytes

average of 40.29% and 36.42% of memory, respectively.

In the Figure 25, the primary y-axis represents disk I/O read and the secondary y-axis shows a disk I/O writes measured in Kilo Bytes (KB). In the case of the OSS SDP approach, 18.38KB disk reads which is maximum as compared to DFT and MQTT with 2.9KB and 1.17KB, respectively for 300 users.

Similarly, OSS had a higher disk writes of 155KB as compared with DFT and MQTT with 83KB and 96KB respectively for 300 users. The OSS has more disk read/writes due to the read/write of bucket values based on each trigger. While in Apache NiFi, data flow is through the queue and doesn't had sever disk read/writes.

Network performances of the SDPs were measured as Network receive and

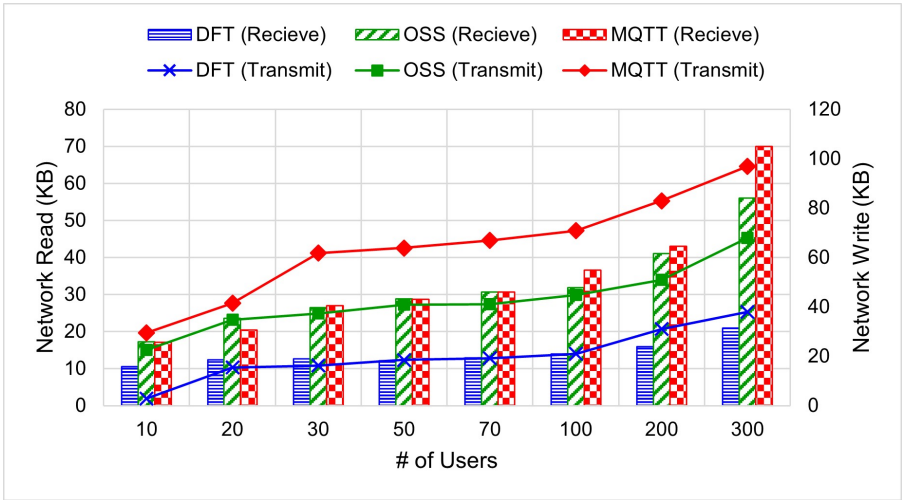


Figure 26. Aeneas application- Average Network Transmit and Average Receive data and measured in Kilo Bytes

transmit bytes as shown in Figure 26. The OSS and MQTT has highest Network receive bytes calculated as average overall users 32KB and DFT performed well with 34KB. But for Network transmit bytes, MQTT had the highest reading with 64.3KB for 300 users. While, DFT had less network transmit bytes of 20KB over 300 users. MQTT recorded with highest values in terms of network performance due to each data flow with topic will be published and subscribed over the network, even in OSS most of the network operations recorded on buckets with event triggers.

In this application, MQTT based SDP had a lowest computation time with minimum processing time as compared with DFT and OSS. Further, MQTT based SDP did not experience any drop of data units in the pipeline as compared with PocketSphinx and Custom video application. Moreover, CPU, Memory consumption and data Read/Writes metrics were also lowest, but there was raise in Network Receive/Transmit but it was negligible since the data unit size in the pipeline was very minimum. Considering the above metrics and associated SDP performances for Aeneas application, it is evident that MQTT SDP worked better over OSS and DFT, as shown in suitability table Table 7.

Performance metrics of the PocketSphinx application. The processing time of the PocketSphix application over all the proposed SDP's were measured as shown in Figure 27. The OSS had the highest processing time of 851s, whereas DFT had 663s with minimum processing time. This application had large a set of functions in the pipeline and OSS event notifications are set to three buckets to store intermediate results. All the events are triggered asynchronously and lead to a larger processing time. As similar to Aeneas, DFT shared highest disk access time, whereas OSS and MQTT had maximum computation time. In MQTT, major challenge was the data unit drop rate increased as the number of user requests

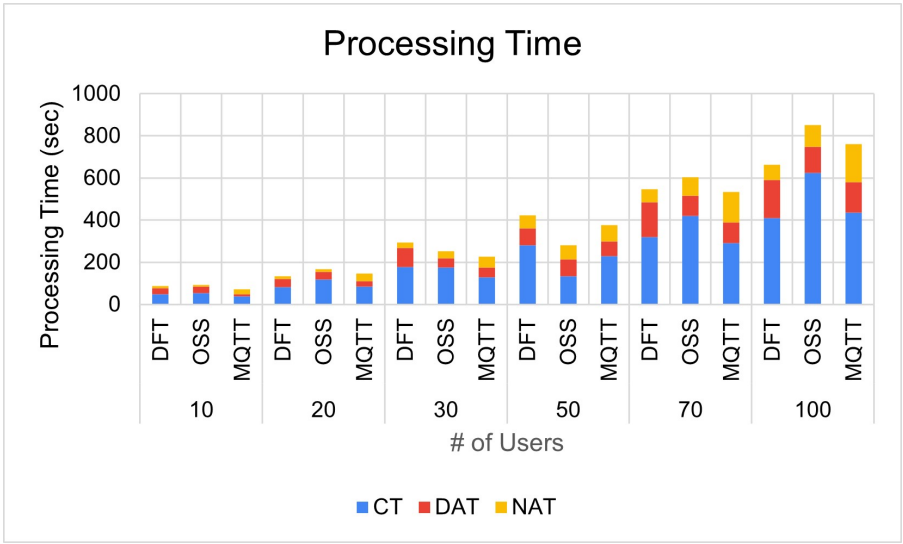


Figure 27. PocketSphinx application- Processing time measured in seconds

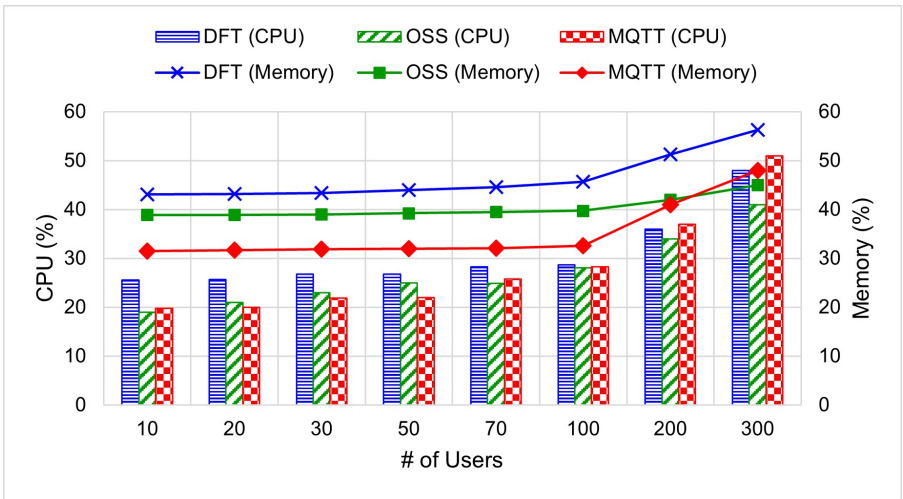


Figure 28. PocketSphinx application- Average CPU utilization and average Memory utilization and measured in

increased and here, drop rate was approximately 2%.

The CPU utilization and Memory utilization were observed in % as shown in Figure 28. The OSS and MQTT equally consumed the CPU of 28% calculated over 300 users as shown in the primary y-axis, whereas DFT consumed highest CPU 31%. However, MQTT had more CPU usage when user requests increased which is not suitable in terms of compute intensive and heavy compute bounded workloads. Similarly as in Aeneas, DFT uses Apache NiFi and required more CPU to execute processors concurrently. The memory utilization shown in the secondary y-axis, DFT had highest average memory utilization of 46% whereas

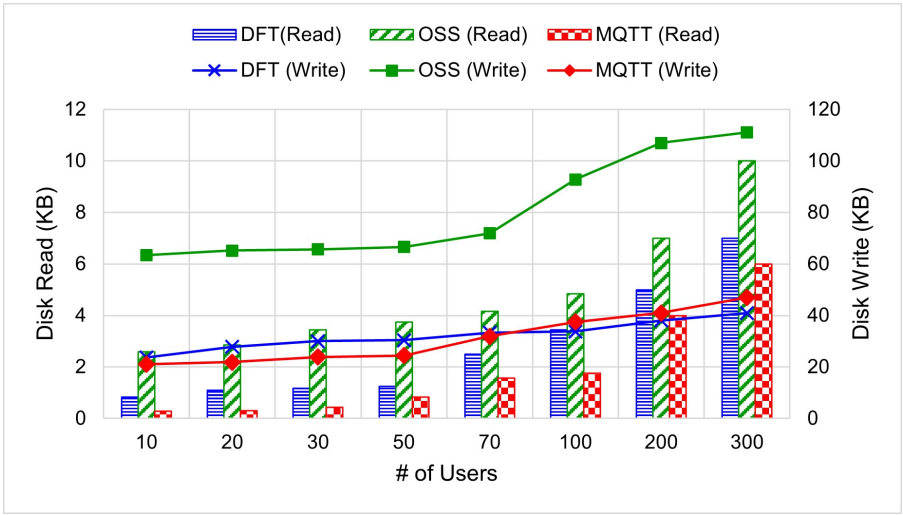


Figure 29. PocketSphinx application - Average Disk Reads and average Disk Writes and measured in Kilo Bytes

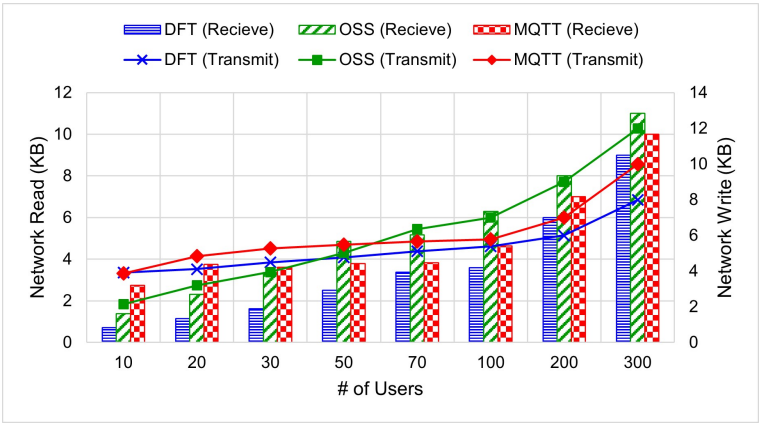


Figure 30. PocketSphinx application - Average Network receive and transmit and measured in Kilo Bytes

MQTT used the highest memory of 48% after 300 users.

The disk I/O read and writes are measured in KB as shown in Figure 29. The OSS leads to more disk read 5KB as compared to DFT with 2KB and least with MQTT-based SDP of 1KB over 300 users. Similarly as in Aeneas, OSS utilizes the MinIO (S3) storage and at each event triggers on bucket notification, data would read from the disk leading to higher disk reads. The disk writes were shown in secondary y-axis, as similar OSS had significant disk writes of 70KB because the number of buckets used were more as compared with Aeneas and Video processing application. The least disk writes by MQTT-based SDP and moderately by DFT is due to data pushed and pulled in the queue neither directly written nor read from the disk.

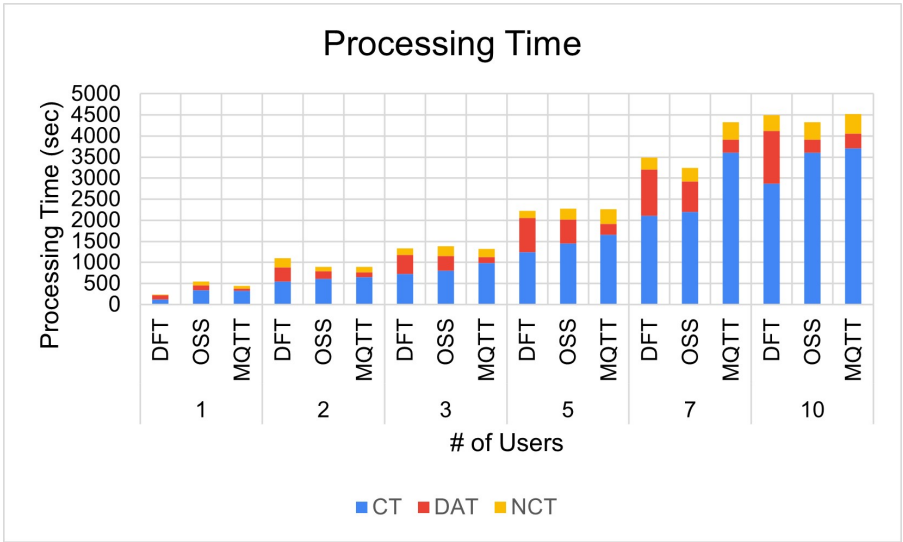


Figure 31. Video processing application - Processing time measured in seconds

The network receives and transmits were measured in KB as shown in Figure 30. Here, the network received bytes performance of OSS had highest as 6KB while DFT had least with 2KB. The secondary y-axis represents the Network transmit bytes, similarly as above OSS had the highest transmit bytes with 7KB and DFT had least with 5KB over 300 users. Push events and notifications events of MinIO make OSS to consume more Network receive and transmit bytes. Evermore, Pocketsphinx uses more buckets as compared with other applications.

Considering the above performance metrics, its observed that DFT and MQTT performed equally better on Pocketsphinx application as compared with OSS. However, Pocketsphinx required more bandwidth to transfer audio files over the fog network and this motivates to consider DFT as suitable SDP shown in Table 7.

Performance metrics observed for custom video application. The video processing application naturally demands huge computation power and more bandwidth to process and offload the video files. The quality of the video is determined by frame rate. A frame per second (fps) is the speed at which individual still images, known as frames, are displayed in the video. The higher value of fps in the video requires more resources to process. So, it is significantly necessary to investigate the performance metrics based on the change of the fps values. In our work, we considered fps values scaling from 1 to 15 and measured the performance. Along with this, its essential to investigate the rate of arrival of such user videos as measured in earlier applications. So, in this section, we will describe the performance metrics collected based on the change in fps values and arrival rate of user videos.

The processing time was measured in seconds (s) as shown in Figure 31. Here, the primary x-axis shows the number of users The primary y-axis represents the Processing time measured for number of users. The DFT worked better with

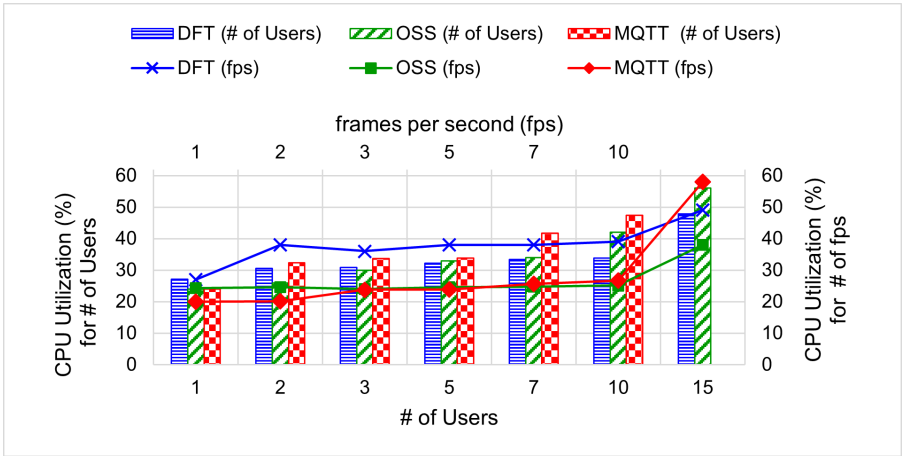


Figure 32. Video processing application- CPU utilization measured in percent

4500*s* as compared with OSS and MQTT based SDP with 5520*s*, 4515*s* respectively over 10 users. As like other applications MQTT had major issue of dropping the data units intermediate pipeline and it was approximately 28%. The other challenge was, MQTT openfaas-connector invokes the function carrying heavy data input, which makes maximum openfaas NATs queue memory utilization and this raises an exceptions from openfaas gateway. Similar to other applications, computation time was maximum in OSS where as DFT has minimum disk access time. The event triggers in MinIO and topic publish and subscription with huge multi-media (audio) data took quite higher processing time.

The Figure 32 represents the CPU utilization across scaling of number of users and fps values. Based on scale of users, DFT performed better with CPU utilization of 32%, but MQTT consumed more CPU with 35% over 15 users. However, MQTT worked better based on the scale of fps values with 23% and DFT consumed more CPU with 36%. Considering both of the scenarios, OSS worked well. Even though the MQTT-based SDP works well in case of fps based scenario but consumed more CPU in another scenario.

The memory utilization was shown in Figure 33, MQTT based SDP consumed less memory as compared to OSS and DFT with 34%, 46% and 44% respectively over 15 users. Interestingly, OSS consumed less memory with 35% as compared with MQTT and DFT with 46% and 49% respectively. The MQTT-based SDP and OSS were good in terms of memory consumption considering both of the scenarios.

Disk Read for both of the scenarios shown in Figure 34, OSS had very few disk reads in both of the scenarios with average values of 1KB, 6KB respectively. MQTT-based SDP had more disk reads and DFT moderately worked better in both of the scenarios.

Figure 35 shows the Disk Writes, MQTT has minimum disk write in both scenarios with an average value of 966KB, 33KB respectively. OSS had the highest

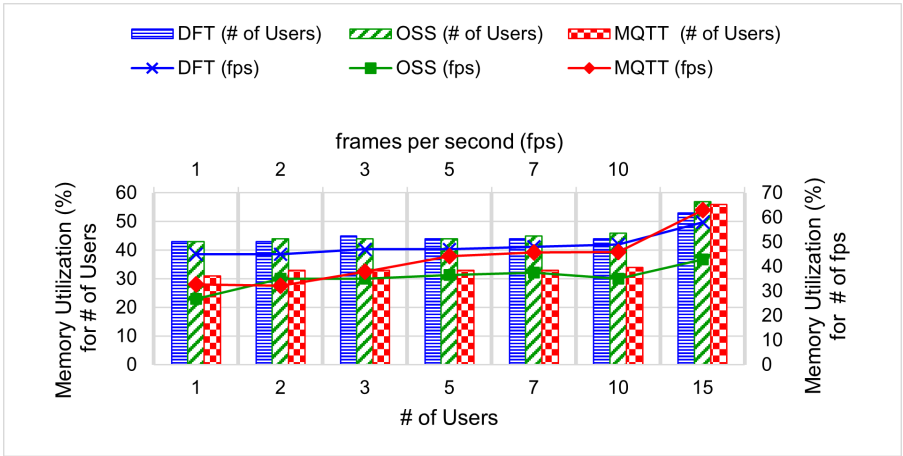


Figure 33. Video processing application- Memory utilization measured in percent (%)

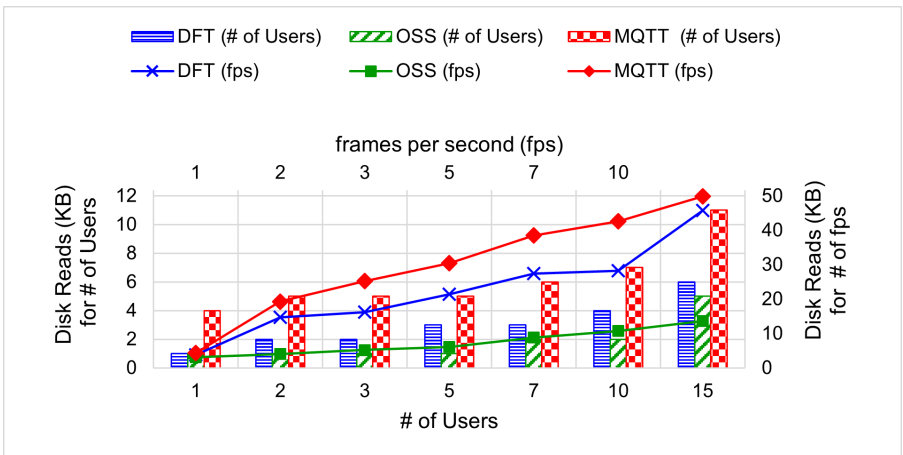


Figure 34. Video processing application- Disk Reads measured in Kilo Bytes (KB)

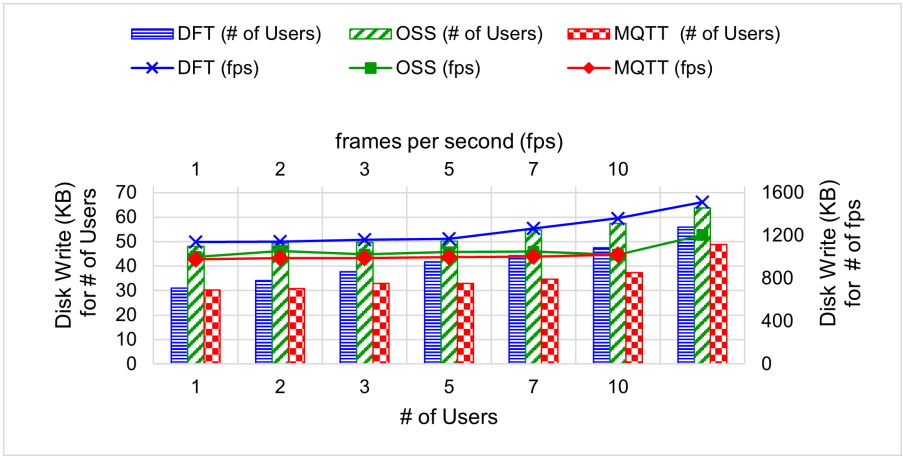


Figure 35. Video processing application- Disk Writes measured in Kilo Bytes (KB)

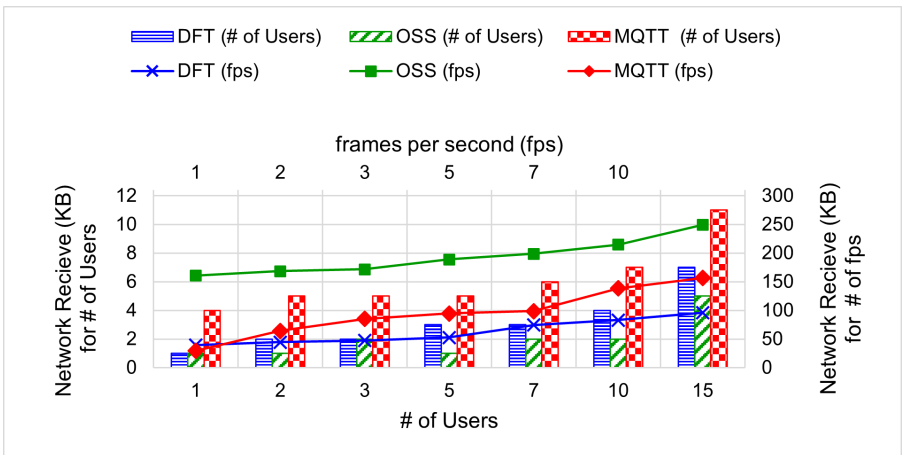


Figure 36. Video processing application- Network receive bytes measured in Kilo Bytes (KB)

disk writes based on the number of users while DFT had more based on the fps. The OSS will have obviously higher disk writes due to objects stored on the disks and DFT had more disk writes due to interaction of NiFi processors with file system.

The Figure 36 and Figure 37 represent the network performance measurement for both of the scenarios in Kilo Bytes (KB). In terms of network receive bytes in scaling of users scenario, OSS worked very well with 2KB and DFT moderately better with 4KB, but MQTT had more network receive bytes with 5.3KB considering 15 users. However, the scaling of fps values scenario, DFT worked well with 59KB, but OSS had maximum values over all the SDPs with 215KB. In network transmit bytes, DFT performance was moderately good with 6KB, 47KB but OSS consumed minimum network resources(receive bytes and transmit bytes)

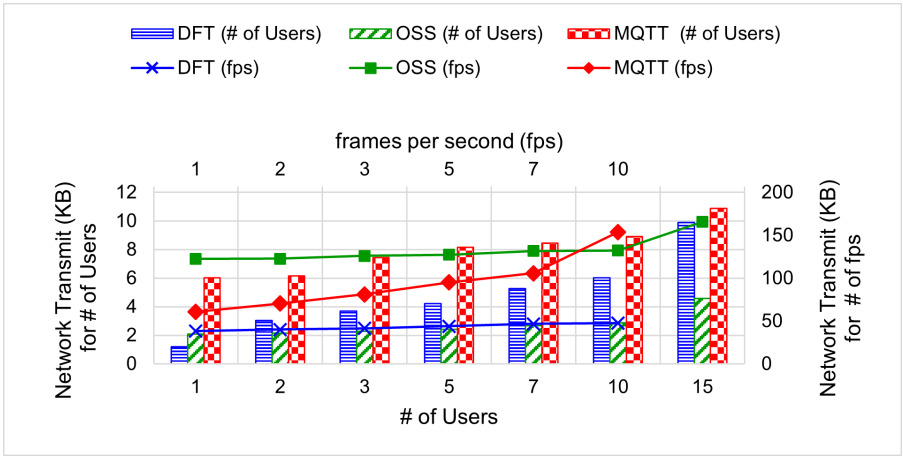


Figure 37. Video processing application- Network Transmit bytes measured in Kilo Bytes (KB)

in terms of scaling the users whereas MQTT consumed more network resources. The MQTT based SDP consumed more network bandwidth due to publish of the multi-media data over the tcp network and mqtt-openfaas clients always listen to this topic leading to higher network consumption.

Considering the various performance metrics based on scaling of users and fps values, results that Video processing application consumes more CPU due to ffmpeg and YOLOv3 tools and even demand more bandwidth to offload the multi-media files between edge/fog/cloud infrastructure. However, DFT consumes less network resources but more CPU and disk resources, rather OSS uses less CPU but required more network resources. MQTT-based SDP performance shows that not suitable for Video processing due to the heavy usage of resources.

Suitability analysis: . All the proposed SDPs were implemented for three fog computing applications, the observed performance of all metrics vs all applications are reported in Table 7. The focus of suitability analysis was to extract insights from observed experimental results in the selection of best-fit SDP for Aeneas, PocketSphinx, and Video processing applications. The overall results were summarized with the suitability index and the corresponding suitable SDP for each application was presented in Table 7.

In this Table 7, average performance metric values were calculated by averaging the recorded values of SDPs performance across individual applications, and then minimum and maximum of such average values were noted and corresponding SDPs were chosen, as mentioned in the Table 7. Further, the suitability index was calculated by counting the SDP names across each application on both the Minimum and Maximum of average columns, and then the percentage of their contribution, over all the metrics was calculated. Because this helps to decide at what percent the SDP is suitable (Minimum of average column) or not suitable (Maximum of average column). Finally, according to the suitability index, the

Table 7. Average performance metric values and suitability index calculated across each application

Metrics/Application	Aeneas		Pocketsphinx		Video Processing application			
					# of Users		fps	
Average Metric Values[6]	Min of Avg.	Max of Avg.	Min of Avg.	Max of Avg.	Min of Avg.	Max of Avg.	Min of Avg.	Max of Avg.
Processing Time (s)	MQTT (MQ)	OSS	DFT	MQ	DFT	OSS	OSS	MQ
CPU Utlization (%)	MQTT (MQ)	DFT	OSS/MQ	DFT	OSS	MQ	OSS	DFT
Memory Utilization (%)	OSS	MQ	MQ	DFT	MQ	OSS	OSS	DFT
Disk Read (KB)	MQTT (MQ)	DFT	MQ	OSS	OSS	MQ	OSS	MQ
Disk Writes (KB)	OSS	DFT	MQ	OSS	MQ	OSS	OSS	MQ
Network Receive (KB)	DFT	OSS	DFT	MQ	OSS	MQ	DFT	OSS
Network Transmit (KB)	DFT	MQ	OSS	MQ	OSS	MQ	DFT	OSS
Suitability index (%)	MQ (43%)	DFT (43%) OSS (43%)	MQTT (57 %)	MQTT (43 %)	OSS (57%)	MQTT (57%)	OSS (71%)	MQTT (43%)
Suitable SDP	MQTT		DFT		OSS		OSS	

Table 8. Average performance metric values across three applications

Metric/Application	DFT	OSS	MQTT
Processing Time (m)	20.97	23.97	21.77
CPU Utilization (%)	69	61	52
Memory Utilization (%)	97	85	86
Disk Read (KB)	4	19	3
Disk Writes (KB)	102	197	95
Network Receive (KB)	15	31	33
Network Transmit (KB)	22	43	63

well-suited SDP for each application was noted as mentioned in Table 7.

As mentioned earlier, Aeneas is a BI application and according to the suitability index, MQTT-based SDP is well suited for this application with 43%, in-spite it has huge network consumption which is not acceptable for BI applications. However, it had good performance in other metrics. The DFT is not suited due to higher processing time and disk utilization with 48%. In the PocketSphinx application, MQTT-based SDP has a suitability index of 57%, whereas it also has the highest not suitability index with 43%. The OSS had poor performance in all aspects, but DFT has 0% index for not suitability, this motivates to consider the DFT as the best-suited SDP for PocketSphinx. Finally, for the Video processing application, the performance of OSS was significant with a suitability index of 51% and 71%.

Experience and future directions. The set of experiments and associated results in earlier subsections show that SDPs performance varies significantly according to end user application (CI, BI). An IoT has a stochastic and heterogeneity (latency intensive, CI, BI) nature of workloads. To process such data oriented workloads the placement and design of the data pipeline mechanism on fog/cloud is quite necessary where our research fills this gap.

However, while designing and implementing these SDP approaches significant portion of time was consumed in designing and developing the serverless functions in various proposed SDPs. The OSS consumed more time to design as it required more than five number of serverless functions as shown in Table 2, whereas DFT is least with maximum of three functions because of a set of built-in processors in Apache NiFi that could handle necessary utility operations such as *PUTS3* to store a data in MinIO were used. But in MQTT based SDP and OSS we need to write them as functions. The DFT was best in designing and implementing, because state of art Apache NiFi data pipeline tool was used, which basically reduced efforts and easily integrated with serverless frameworks. The function templates and associated Docker files of serverless functions including other utility source files are available in GitHub [7].

Apart from the design experience, the resource utilization metrics such as CPU, Memory, Disk Reads and Writes are important in serving the demands of IoT applications, because the resource demand from end-user requests vary with

[7]https://github.com/shivupoojar/ServerlessDataPipelines

respect to the type of the application. For example, video application demands for maximum compute and bandwidth resources, whereas text processing application demands only for bandwidth. So to investigate and analyze the resource utilization metrics and processing time, we calculated the average over all the three applications (on performance metrics) as shown in the Table 4.4.3.

The DFT consumed highest CPU (69%) and Memory (97%), whereas least in network utilization (15KB, 22KB) and processing time (20.97m), this indicates that DFT is best suitable for applications with huge bandwidth demand such as text processing. Further, OSS consumed moderate CPU (61%), Memory (85%) and network utilization (31KB, 43KB), but had large number of disk read and writes (19KB, 197KB). These results show that OSS is best fit for video or image processing applications (bandwidth and compute-intensive) due to lesser CPU, Memory, and network utilization. On the other side, MQTT-based SDP utilized the highest network resource (63KB, 33KB) and lesser CPU (52%), disk, and moderate memory resources (86%). So, MQTT-based SDP is best suitable for compute-intensive applications, but not suitable for bandwidth-sensitive applications.

Even though the SDPs were designed and investigated significantly based on different performance metrics, certain challenges still exist and the key improvements can be undertaken. The future directions should focus on two aspects. Firstly, on how to minimize the processing time and resource utilization. In our analysis, a synchronous mechanism was used to invoke the serverless functions leading to larger processing time and they should be tested with asynchronous mode. The serverless function scaling mechanism can also substantially reduce the processing time and using intelligent scaling mechanisms could overcome this issue. Secondly, how can SDPs be executed on fog/cloud in terms of dynamic and stochastic workloads? Serverless operations were focused to fog infrastructure in this work, however, to achieve user QoS expectations and to fulfill the dynamism nature, a part of the pipeline could be dynamically executed in fog and rest in the cloud. So, several opportunities exist to implement such intelligent decision mechanisms.

4.5. Summary

In this chapter, we proposed three approaches to the Serverless Data Pipeline (SDP). DFT, OSS, and MQTT-based SDP using Apache NiFi, MinIO, and MQTT services, respectively. We applied these approaches to three different fog computing applications, namely Aeneas, PocketSphinx, and Video processing application. We investigated their performance using metrics such as processing time (computation time, disk access time, and network access time) and resource utilization (CPU, memory, network, and disk utilization) and rigorously analyzed the results by calculating a suitability index for each of them. The results show that the MQTT based SDP works best for Aeneas, DFT performs better for Pocket-

Sphinx and for video processing applications, the OSS performance was good as compared with SDPs.

The proposed serverless data pipelines (SDPs) can be improved in terms of performance by scaling their components.This autoscaling process includes serverless functions and intermediate data storage units such as message brokers, triggers, and object storage. In contribution 3 (Chapter 5), we explore approaches to auto-scaling, such as reactive methods, and provided hints to proactive approaches. We use a message queue-based SDP to investigate how the components can be autoscaled and how suitable they are for dynamic and stochastic workloads.

5. AUTO-SCALING OF SERVERLESS DATA PIPELINES

This chapter examines auto-scaling mechanisms for serverless data pipelines that are designed to handle dynamic and stochastic workloads. We will explore various approaches, such as reactive methods, to scale the SDP components and assess their suitability for four types of dynamic workloads in two real-time IoT applications. We provide a suitability analysis of the six scaling approaches using the weighted scoring method for two different applications with four workload patterns. Furthermore, we discuss the challenges of reactive mechanisms based on extensive experiments and provide future directions.

5.1. Introduction

Auto-scaling is a mechanism for dynamically increasing or decreasing resources based on demand to meet QoS expectations. Modern container orchestration tools such as Kubernetes (k8s) are equipped with easy, granular auto-scalability mechanisms [15] compared to virtual machine scalability. Reactive scaling approaches are broadly classified into workload-based and resource utilization metrics-based, respectively [61]. The former focuses on increasing or decreasing 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. The following provides more information on the scaling approaches used in this chapter.

Workload based scaling: The workload-based scaling approach is widely 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.

Resource based scaling: In this scaling approach, the system tries to keep the metrics, such as CPU and memory, in a specified threshold limit. Scaling action is taken with the addition or deletion of resources if the threshold is reached. This is a commonly used approach for scaling microservices and virtual machines in the cloud. Open-source serverless platforms such as OpenFaaS, Fission, Nuclio, Knative, and other tools use this approach for scaling function replicas. In most of the Kubernetes-based serverless platforms, the Kubernetes Horizontal Pod Au-

toscaler (HPA) is used to scale based on the CPU and Memory limits set to the functions.

Auto scaling of microservices is a well investigated area of research [56]–[58]. 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 [59]. This section briefly summarizes the recent work done in the context of scaling the serverless data processing architectures and models.

Lucia et al. [60] 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. On a similar line, [61] 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.[62] 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. [151], and 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. [152] investigated the performance of workload and resource based scaling of various serverless platforms by considering the steady and spikes workloads.

The workload-based [64] 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. Similarly, Mahmoudi et al. [65] 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.

A comprehensive comparison of related works and our proposed work is presented in Table 9. 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.

Table 9. Overview of related works

Article	Application Domain	Environment	Auto Scaling Technique	Auto Scaler	Components	User patterns	Performance Metrics
Li et al. [62]	IoT	Edge	Resource Utilization (CPU)	Custom scaler using K8S	Serverless Functions	Spikes	Latency, Throughput
Jegannathan et al. [63]	Cloud	Cloud	Resource Utilization (CPU)	Custom scaler using K8S	Serverless Functions	Spikes	Cold Start Time, Latency, Pod Count
Benedetti et al. [61]	IoT	Edge	Resource Utilization (CPU)	Custom scaler using K8S	Serverless Functions	Steady	Latency
Zafeiropoulos et al. [151]	Cloud	Cloud	Resource Utilization (CPU)	Custom scaler using K8S	Serverless Functions	Fluctuations	Latency, Throughput, SLA Latency,Success Rate
Palade et al. [187]	IoT	Edge	Resource Utilization (CPU)	K8S HPA	Serverless Functions	Steady	Latency, Success Rate
Junfeng et al. [152]	Cloud	Cloud	Resource Utilization, Workload Based (RPS)	K8S HPA, RPS	Serverless Functions	Spikes, Steady	Latency
Zhou et al. [188]	Cloud	Cloud	Resource Utilization (CPU)	K8s HPA	Serverless Functions	Spikes	Latency
Trieu et al. [189]	Edge	Edge	Resource Utilization (CPU)	K8s HPA	Serverless Functions	Steady	Latency, Pod Count
Gotin et al. [64]	IoT	Cloud	Resource Utilization (CPU), Workload Based (QueueLength, Message Rate)	Custom	Message Queue	Steady	CPU Utilization, Throughput
Arjona et al. [190]	Cloud	Cloud	Workload Based (Event Rate)	KEDA	Serverless Functions	Steady	Delay, Events processed
Mahmoudi et al. [65]	Cloud	Cloud	Workload Based (Concurrency)	Custom	Serverless Functions	Steady	Latency, Cost
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

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 compared to 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.

We will use reactive scaling approaches to auto-scale the SDP components. In Chapter 4, we investigate the performance of three SDPs approaches in three real-time IoT fog applications (video processing, Aeneas, and Pocketsphinx). Message queues are widely used in IoT systems [135]–[140], to queue and route data. In this chapter, we focus on SDP based on Message Queues designed for two IoT applications as described in Chapter 2 Subsection 2.6.

The Message Queue-based SDP approach has three pipeline components: Message Queue (MQ), Message Queue Trigger (MQT), also known as Function Invoker/Trigger, and serverless functions. Here, data processing tasks are designed as serverless functions, while MQ and MQT are the data carriers in the pipeline. When data arrive 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 until the data sink.

We have chosen SDP applications with different characteristics, one having a longer execution time (2145ms) with more resource intensiveness and another with a shorter execution time (986ms) with minimum resource utilization. The Aeneas application deals with audio data that need more network bandwidth, whereas PuhatuMonitoring deals with text data that needs minimum network bandwidth. Such a category of applications helps to understand the efficacy and behavior of auto-scaling mechanisms described in Subsection 5.4. Therefore, the management of auto-scale in serverless data pipelines can be challenging due to the need for synchronization between the serverless platform and the components of the data pipeline [55]. The proposed work explores the components of serverless data pipeline approaches that can be scaled. In addition, it examines when to scale the components and select essential threshold metrics or optimal configurations that are required to define scaling. We also describe how to scale the components based on workload or resource scaling metrics. Finally, we measure and compare the efficiency and behavior of workload-based and resource-based scaling approaches in fog environments using real-time applications with different

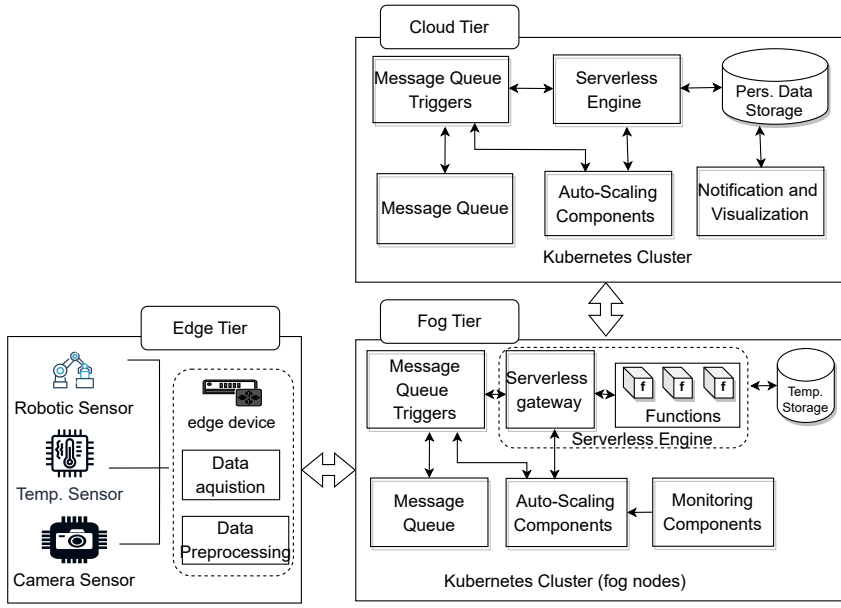


Figure 38. Three tier System architecture containing SDP components

execution times of functions.

Overall contributions in this Chapter are as follows:

- We applied and evaluated reactive scaling approaches, namely workload based (RPS, Message rate) and resource utilization (CPU threshold) based techniques on data pipeline components (Message queue trigger and serverless functions).
- We used two real-time fog computing workloads, the Aeneas and Puhatu-Moniroting 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, experiences, and challenges encountered during the implementation and evaluation of the scalability of serverless data pipelines in various configurations.

The rest of the chapter is organized as follows. We described the three-tier architecture in Subsection 5.2. Following this, all auto-scaling approaches and research questions are elaborated in Subsection 5.4, and further, experiment setup and results are outlined in Subsection 5.5. In Subsection 5.6 we share our experiences from this evaluation investigation and propose future research directions. Finally, the concluding remarks along with the proposed future work are discussed in Section 5.7.

Table 10. Execution time, CPU, and Memory used by individual functions of two IoT applications

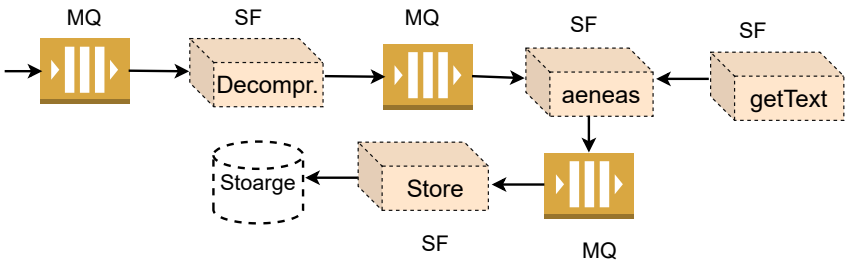
Application	Execution time (ms)	Functions	Function execution time (ms)	CPU consumed (milliCPU)	Memory used (MiB)
Aeneas	2145	Decompress	1	2	55
		aeneas	2000	546	243
		getText	100	5	4
		Store	44	6	7
PuhatuMonitoring	986	preProcess	6	6	97
		outlierIdentify	940	154	243
		tagData	5	3	32
		storeData	35	4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

5.2. System Architecture

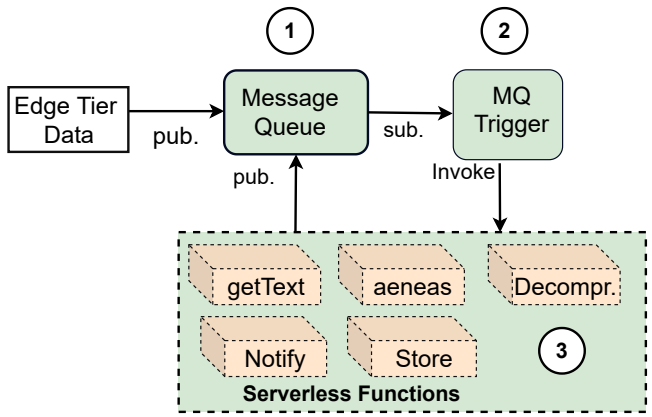
In this section, we described an overall three tier architecture, that includes the required services to accomplish the Message Queue based SDP at the fog and cloud tier. Further, we described essential services and tools required for auto-scaling the serverless and message queue components at the fog and cloud tiers. The system architecture in Figure 38 is composed of three layers with the required services to handle the scalability of the pipeline components for data processing.

The edge tier manages the endpoint devices, collects and aggregates the data, and performs pre-processing operations such as compression, filtering, etc. Fog Tier is mainly responsible for receiving the data in the pipeline from the edge tier and processing 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. Here, OpenFaaS is used as a Serverless Engine to create, deploy, and scale the serverless functions. Furthermore, we use RabbitMQ[1] as a message queueing 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. 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, visualizing 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 a data sink in the pipeline.

[1]https://www.rabbitmq.com/



(a). Aeneas application serverless data pipeline



(b). Scaleable components in the serverless data pipeline

Figure 39. Aeneas serverless data pipeline and its scalable components.

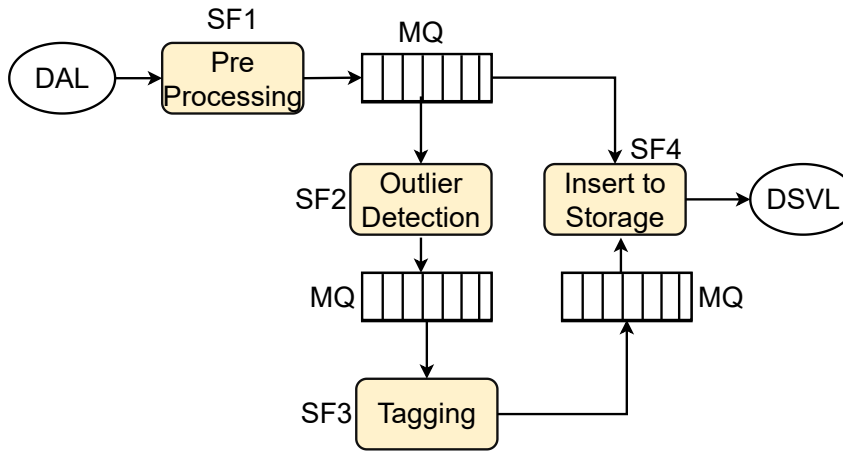


Figure 40. PuhatuMonitoring serverless data pipeline

5.3. Real Time IoT Applications

Considering the system architecture, in this section we provide more insight into the IoT applications described in Chapter 2 Subsection 2.6.

Aeneas SDP contains four serverless functions and three message queues as described in Table 10 and Figure 39a. Here, *aeneas* function has a long running time with more CPU and memory intensive. The total execution time for processing a single data unit (audio file size of 512KB) by the SDP from source to sink was 2145 ms and the execution time of the individual functions are mentioned in Table 10.

PuhatuMonitoring is an IoT application to observe changes in water level in a wetland in the Puhatu Nature Protection Area (NPA), north-west Estonia, next to an open-pit oil-shale quarry. 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 40 shows the message-queue based SDP of the application. All functions have a short execution time and minimum CPU and memory utilization, as mentioned in Table 10.

5.4. Scaling approaches

This section outlines the auto-scaling components of the Message Queue Based SDP within the three-tier system architecture. It discusses how these components are scaled using workload and resource-based approaches to ensure seamless service for stochastic IoT workloads.

The Message Queue based SDP contains mainly three components, namely, 1)*Message Queue*, 2)*Message Queue Serverless Connector (Function Invoker or MQT)* and 3) *Serverless Functions* as shown in the Figure 39. We have considered two components to understand and investigate the scaling techniques based

Table 11. Scaling configurations of serverless functions and MQTs

Applications	Functions/MQTs	K8s HPA						KEDA		RPS	
Aeneas	Decompress	CPU threshold	Replica limit	cpu_limit	cpu_request	memory_limit	memory_request	QueueLength	MessageRate	rps threshold	Concurrency limit
	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		

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.

It is important to identify the metric characteristics of the selected pipeline components in order to define scaling rules for different scaling approaches. The two scaling approaches, workload-based and resource-based, will be used as discussed previously. Each approach is linked to different metrics, and it is critical to identify the relevant metrics for the selected pipeline components. Therefore, we will define metrics for 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. 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 [61] because that may hinder latency and other performance issues.

In **Serverless Functions** component, the scaling metric for the workload based approach is request per second (RPS) [152]. 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.

Table 11 provides an overview of the approaches, metrics, and corresponding SDP components. In the **workload-based approach**, we have considered Request per second (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. The Message Rate and QueueLength to scale the MQT is accomplished using an extra service known as KEDA[2].

Figure 41, 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

[2]https://keda.sh/

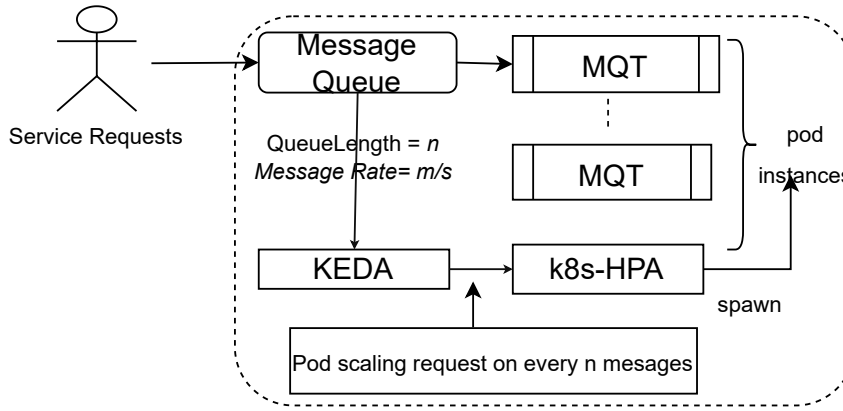


Figure 41. KEDA based scaling architecture

scalers[3] that decide the activation and deactivation of the deployments. An example of KEDA based scaling of MQT is shown in Figure 41. The KEDA and MQT, both are configured as a service in the k8s infrastructure and the detailed experimental setup is described in Section 5.5. The internal architecture of KEDA has two components namely, *keda-metric API server* and *keda-scaler* respectively. Considering Figure 41, 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[4] (k8s-HPA) for both of the SDP components. The k8s-HPA is part

[3]https://keda.sh/docs/2.9/scalers/

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

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 SDP from the Figure 39 (a), scaling metrics and approaches in Table 11 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 5.5. 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.

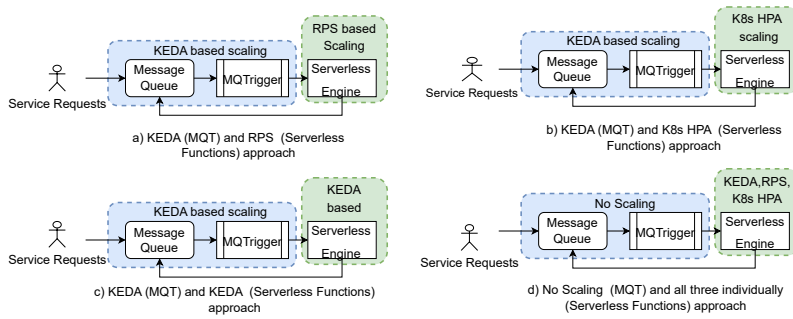


Figure 42. Approaches for auto-scaling the serverless data pipelines

Considering this, we define the six approaches for auto-scaling (see Figure 42) as follows:

1. ***KEDA and RPS(keda+rps) approach:*** In this approach, as shown in Figure 42.a, 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.
2. ***KEDA and KEDA (keda+keda) approach :*** This is shown in Figure 42.c, 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. Similarly to approach 1, KEDA scalers are configured to monitor the queues and auto-scale the queue consumers.
3. ***KEDA and K8s HPA (keda + k8shpa)*** is as shown in Figure 42.b, the K8s Horizontal Pod Auto scaler is configured with the target CPU in all serverless functions. The optimal target CPU is chosen for each function by running several experiments.

4. ***No Scaling and KEDA (no+keda)*** is shown in Figure 42.d, only KEDA is configured on serverless functions.
5. ***No Scaling and K8shpa (no+k8shpa)*** is similar to Figure 42.d, only K8shpa is configured on serverless functions.
6. ***No Scaling and rps (no+rps)*** is similar to Figure 42.d, only rps is configured on serverless functions.

5.5. Performance Evaluation

All the scaling mechanisms described in the previous section are realized, implemented for the Aeneas and PuhatuMonitoring applications, and deployed on the system infrastructure as mentioned in Section 5.2. 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.

5.5.1. Performance Metrics

This subsection provides six potential performance metrics that are used to evaluate and check the efficiency of scaling SDPs in response to variable user demands. 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).

Processing Time. In IoT applications, latency is a primary concern, and it directly correlates with 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.

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 as percent and calculated by the ratio of users processed to the total number of users fed into the pipeline at a given time window.

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 5.1 and is similar for memory utilization.

$$CPU_t = \sum_{SF_k \in SF} (\frac{1}{j} * \sum_{FR_j \in FR} cpu(FR_{kj})) + \sum_{MQT_m \in MQT} cpu(MQT_m) \tag{5.1}$$

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.

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.

$$Count(Replicas_t) = \sum_{SF_k \in SF} Count(SF_k) + \sum_{MQT_m \in MQT} Count(MQT_m) \tag{5.2}$$

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

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 [144] 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. The formula to calculate is:

$$JFI = \frac{(\sum PT_i)^2}{n \times (\sum_i PT_i^2)} \tag{5.3}$$

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

5.5.2. Serverless Workload Patterns

In this section, we explore the workload patterns of serverless applications based on Azure Function Traces [146]. These traces are available along with invocation

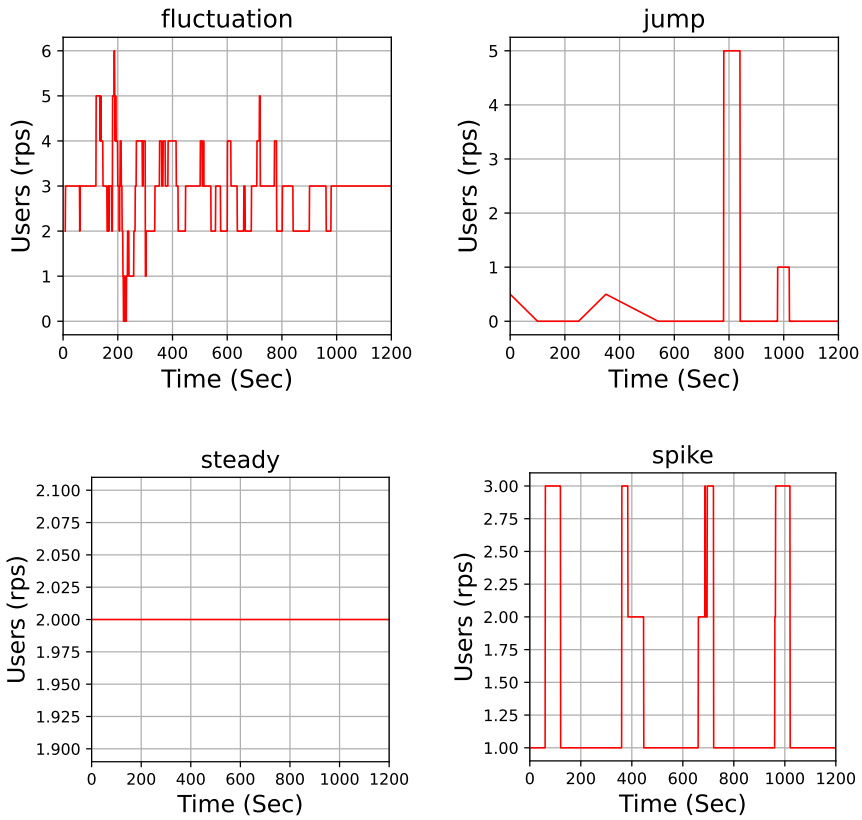


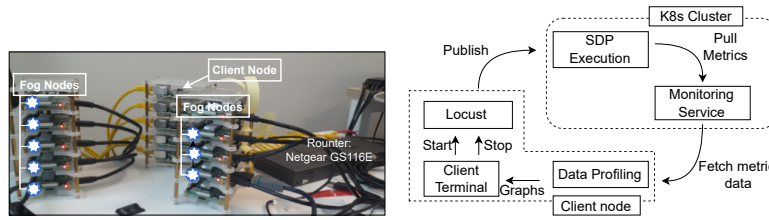
Figure 43. Azure serverless workload invocations patterns

Table 12. 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

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 [54] 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 43. Our chosen data set indicates that 5.2% of the functions exhibit *steady* workloads that remain stable throughout the invocations over time. The *fluctuating* workload (34.37%), shown in Figure 43, 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 [54], which enabled the synthesis of perturbations at the granularity of seconds while preserving the 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 43. The specific characteristics of the workload are described in Table 12, 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.



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

(b) Process flow of experiments

Figure 44. Hardware setup and process flow of experiments

5.5.3. 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 44(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) was installed in the fog tier, and similarly, k8s engine v1.26 was configured in the cloud tier. OpenFaaS serverless platform *faas-netes*(0.23.0)[5] is installed in the fog and cloud tier as a serverless engine to create, manage and scale the serverless functions. RabbitMQ was installed as three pod clusters in the fog layer, with one virtual host, one exchange, and three message queues. The MinIO object storage service was 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 was used to develop the serverless functions. The Prometheus, along with the c-advisor, captures and monitors the k8s service metrics. Moreover, the Prometheus REST API is used to obtain 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 deployed on the RPi-based k8s cluster. Those docker images can be found here in the docker hub[6]. 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

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

[6]https://hub.docker.com/u/shivupoojar

is shown in Figure 44(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[7, 8]. 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.

5.5.4. 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 was used. Locust is mainly used for HTTP-based load testing, and we have written Python scripts[10] to publish user workload on RabbitMQ channels. Several experiments are conducted to realize the results with a minimum error rate over all of the experiments. The following subsections provide a detailed analysis of the Aeneas and PuhatuMonitoring application of the obtained results.

Aeneas Application. The processing time is shown in Figure 45 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 45(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 45(b) 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 45(c) and 45(d) show the processing times of steady and spikes workloads.

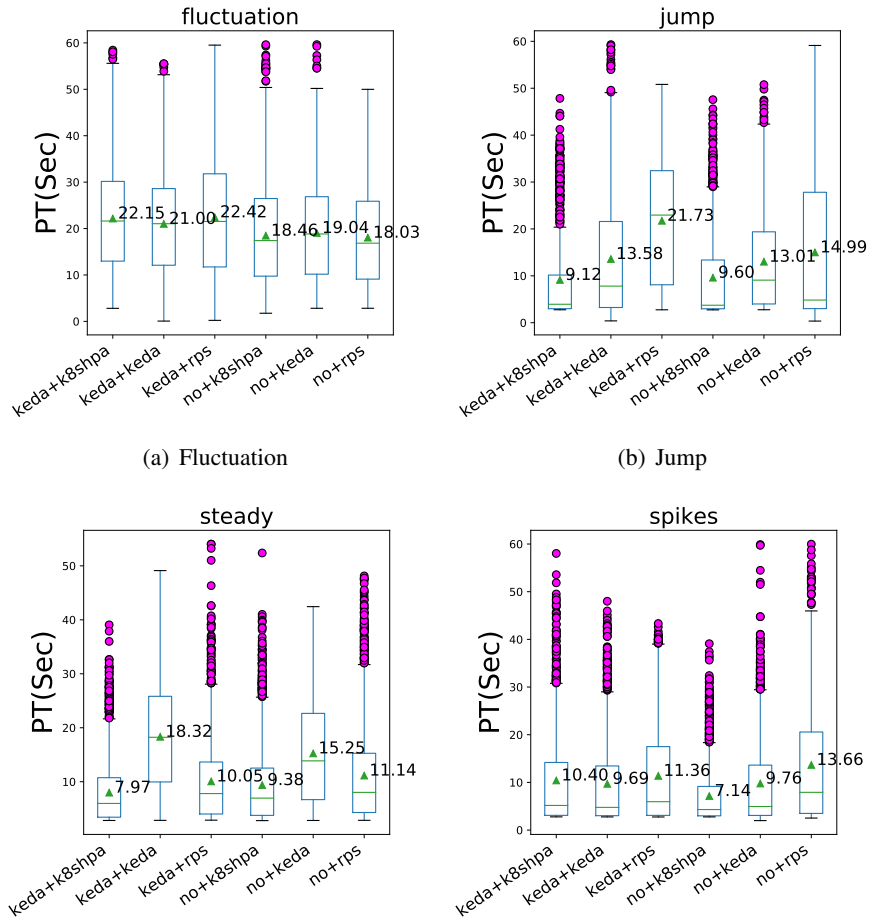
The *keda+k8shpa* and *no+k8shpa* experienced median values of 7.97s and 7.14s respectively, in steady and spikes workloads. This indicates that *no+k8shpa* can handle the spikes workload, this is because Aeneas has a long running function with compute intensive, and CPU-based autoscaling works satisfactorily here.

[7] https://gitlab.doc.ic.ac.uk/st220/COSCO/-/tree/master/framework/workload

[8] https://github.com/readbeyond/aeneas

[9] https://locust.io/

[10] https://github.com/shivupoojar/autoscalingsdp/../k6_locust.py



(a) Fluctuation

(b) Jump

(c) Steady

(d) Spikes

Figure 45. Comparison of processing time for Aeneas application

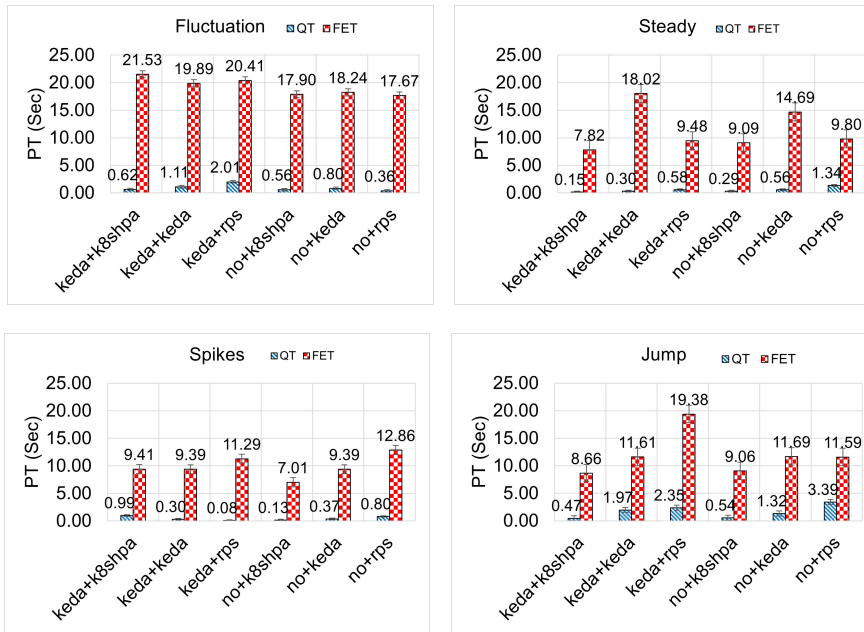


Figure 46. Average Function Execution Time (FET) and Average Queuing Time (QT) of Aeneas application over scaling approaches

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 46. 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 sudden and aggressive workloads. In *steady* workload, *keda+k8shpa*

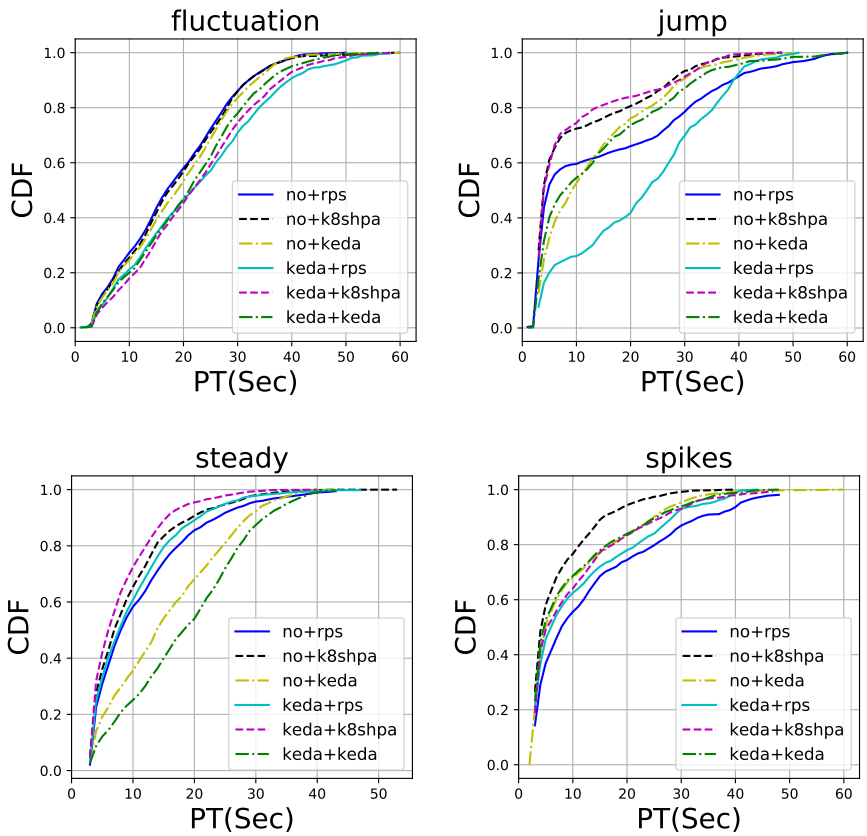


Figure 47. Cumulative Distribution Function of Processing Time for Aeneas application over scaling approaches

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 47. 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 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

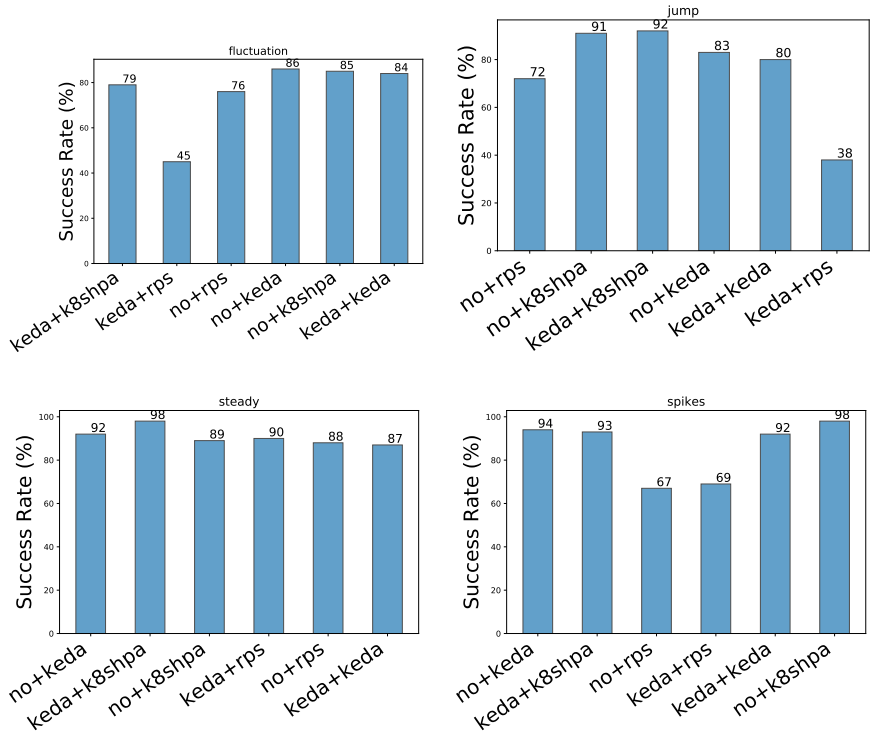


Figure 48. Success rate of Aeneas application over scaling approaches

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 48 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 user requests queued in the internal queue of serverless functions. 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 49. The x-axis represents

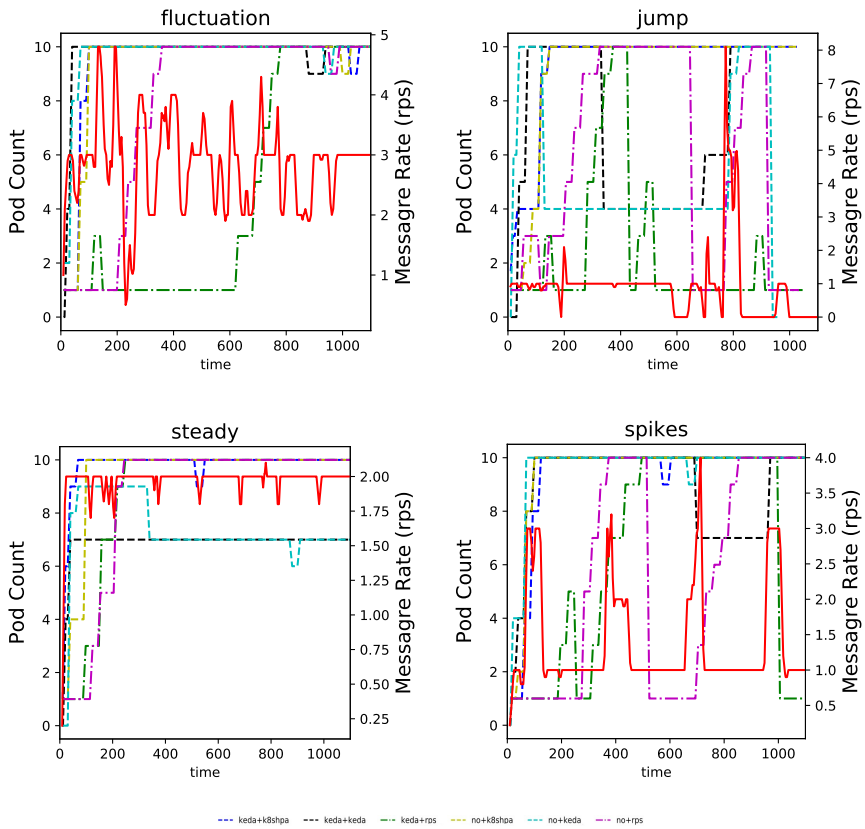
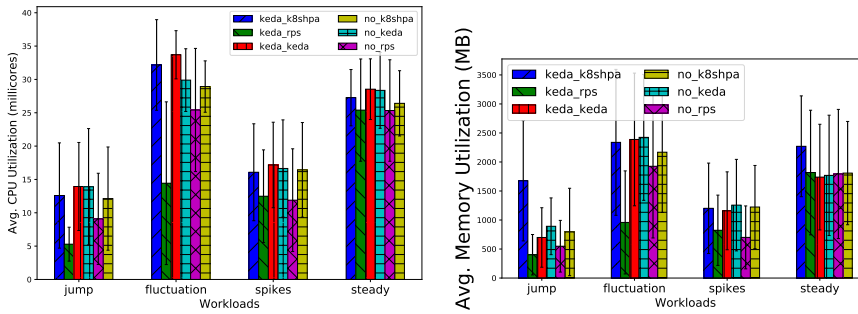


Figure 49. Scaling pattern of Aeneas function w.r.t to message arrival rate



(a) CPU utilization (millicore) (b) Memory utilization (MB)

Figure 50. Comparison of CPU and Memory utilization of Aeneas application for various scaling approaches

the pod count, while the y-axis represents the message arrival rate measured in messages arrived per second. Our observations from Figure 49 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.

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 13 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

Table 13. Overview of performance metrics of scaling approaches in Aeneas application

Workload	**jump**						**steady**						**spikes**						**fluctuation**					
Message Queue	**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
Metric																								
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
Succeded	760	402	965	976	884	852	2123	2160	2141	2350	2204	1523	1103	1141	1618	1541	1550	1523	2637	1564	2956	2751	2967	2905
Suceess 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	11390	8736	9260.8	8905.4	11497	11574	23644	21714	20084	18737.1	33600	38229	15066	12965	11549	16030	15129	14764	47539	35059	54564	60936	56499	61005
PT Mean	15.0	21.7	9.6	**9.1**	13.0	13.6	11.1	10.1	9.4	**8.0**	15.2	18.3	13.7	11.4	**7.1**	10.4	9.8	9.7	**18.0**	22.4	**18.5**	22.2	19.0	21.0
PT Max	59.1	50.8	47.6	**47.8**	50.8	59.3	48.1	54.0	52.4	**39.1**	42.4	49.1	60.0	43.3	**39.1**	58.0	59.9	48.0	**50.0**	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	2.0	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	10.0	12.8	10.6	**6.2**	10.1	9.4	9.6	**10.3**	13.0	**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	25.0	25.7	**32.0**	39.5	**32.0**	38.0	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	18.0	12.9	11.3	**7.0**	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	0.30	0.80	0.08	**0.13**	0.99	0.37	0.30	**0.36**	2.01	**0.56**	0.62	0.80	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	25.40	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	34.90	38.70
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

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 50(a) and Figure 50(b) show the CPU and memory utilization of the Aeneas application in fog environments respectively. In Figure 50(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 approaches (*no+keda*, *keda+keda*) have maximum CPU consumption of around $\approx 38 millicore$. 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 $\approx 25 millicore$, whereas *keda+keda* had a maximum CPU utilization. In steady workloads, both *keda+rps* and *no+rps* have a minimum CPU utilization of around $\approx 19.43 millicore$, 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 50(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 $\approx 2270 MB$ 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 13 provides an overview of all the performance metrics of all scaling approaches with different workloads. From Table 13 it is evident

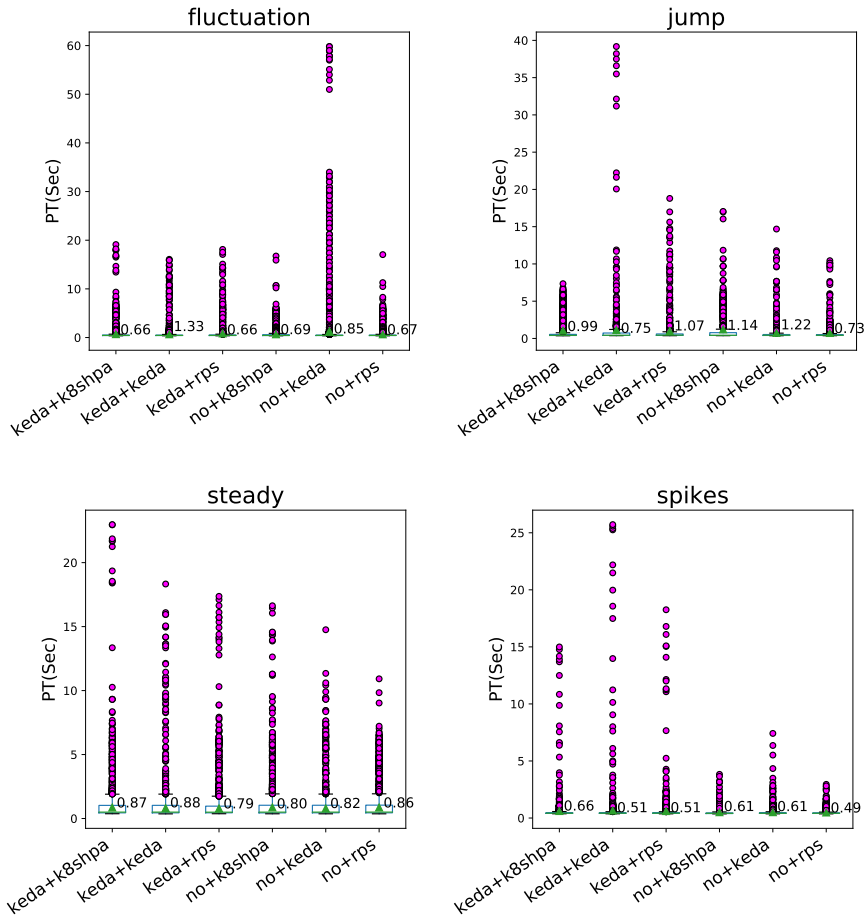


Figure 51. Comparison of processing time of Puhatu application

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.

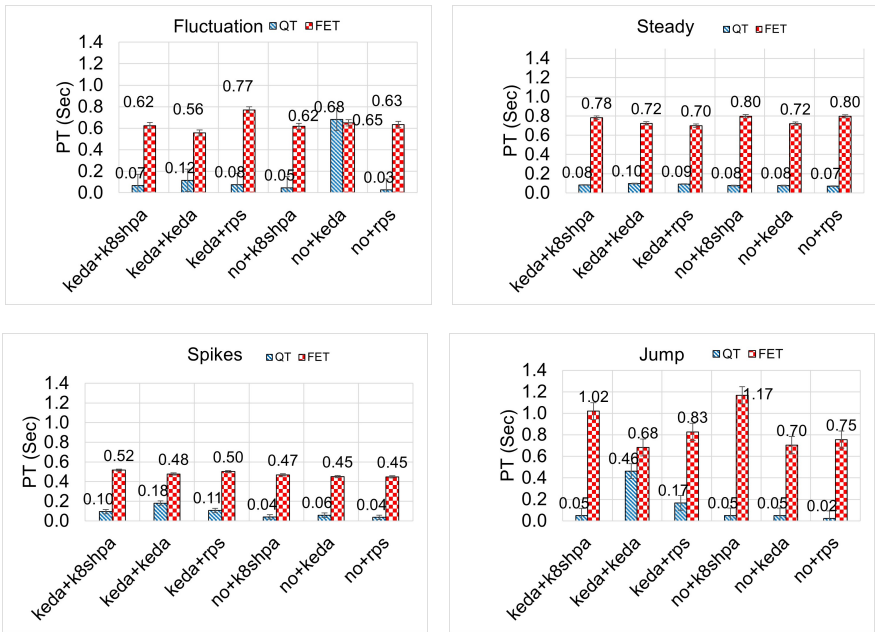


Figure 52. Function Execution Time (FET) and Queuing Time (QT) of Aeneas application over scaling approaches

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 Subsection 5.3. The overview of processing time for various workloads is shown in Figure 51. In fluctuation workload, *no+rps*, *keda+k8shpa* and *keda+rps* approaches yielded lower processing time with a mean of 0.66*s*. 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 7.3*s*. 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 0.79*s*, where as *no+rps* has a maximum value of 10.92*s*. 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 3.8*s*.

Furthermore, Figure 52 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

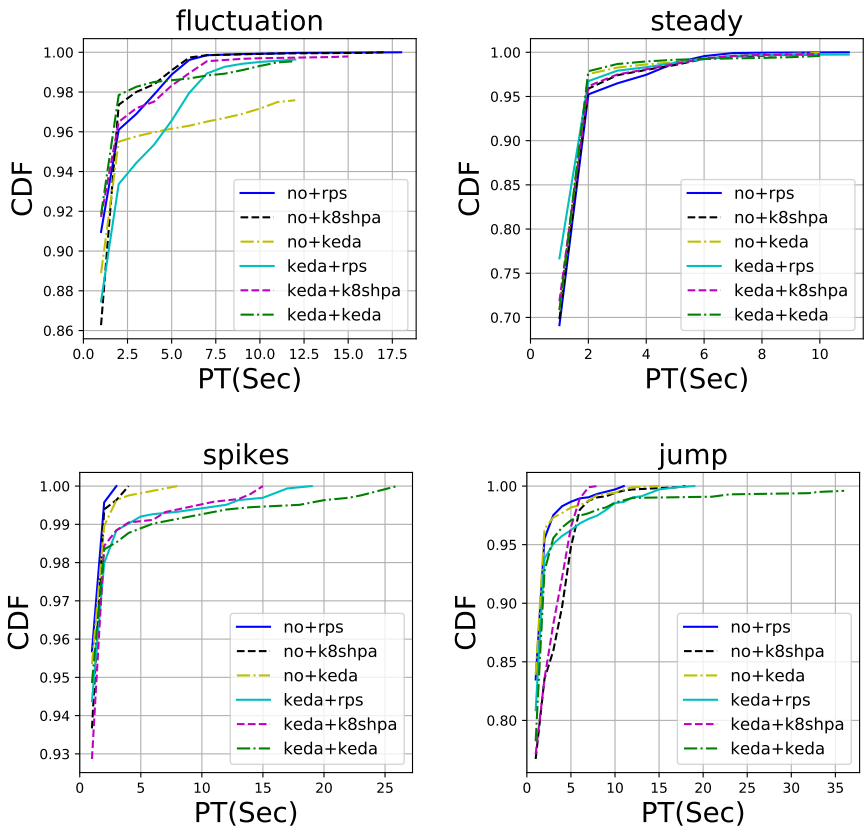


Figure 53. CDF of processing time for puhatu application over scaling approaches

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 53, 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 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 54. For fluctuation workload, *no+k8shpa* has a more success rate of 99% and *keda+rps* was less with 95%. However, in steady workload *no+keda*

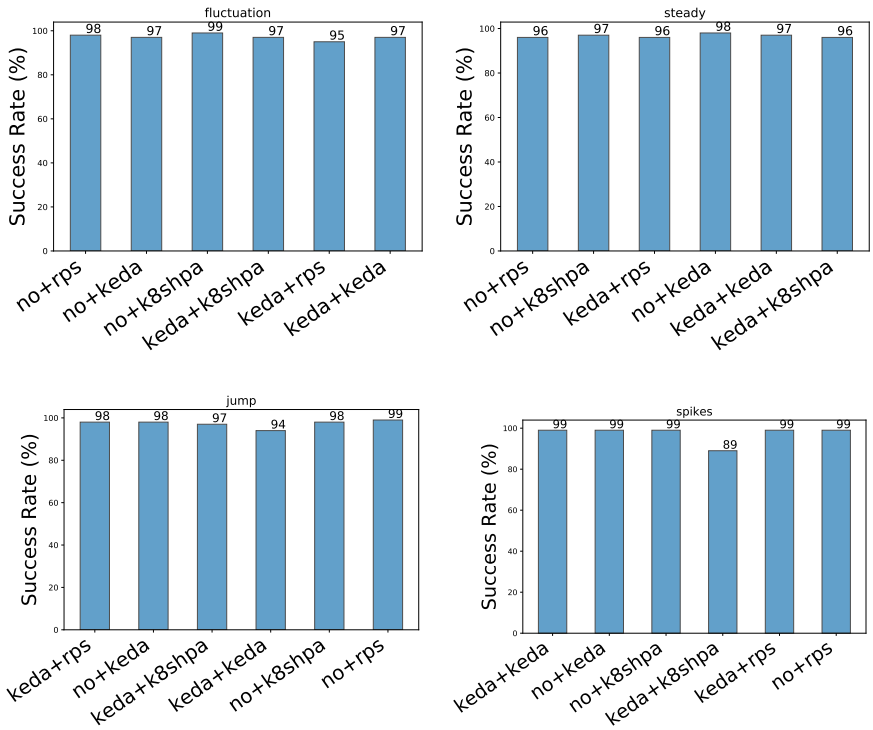
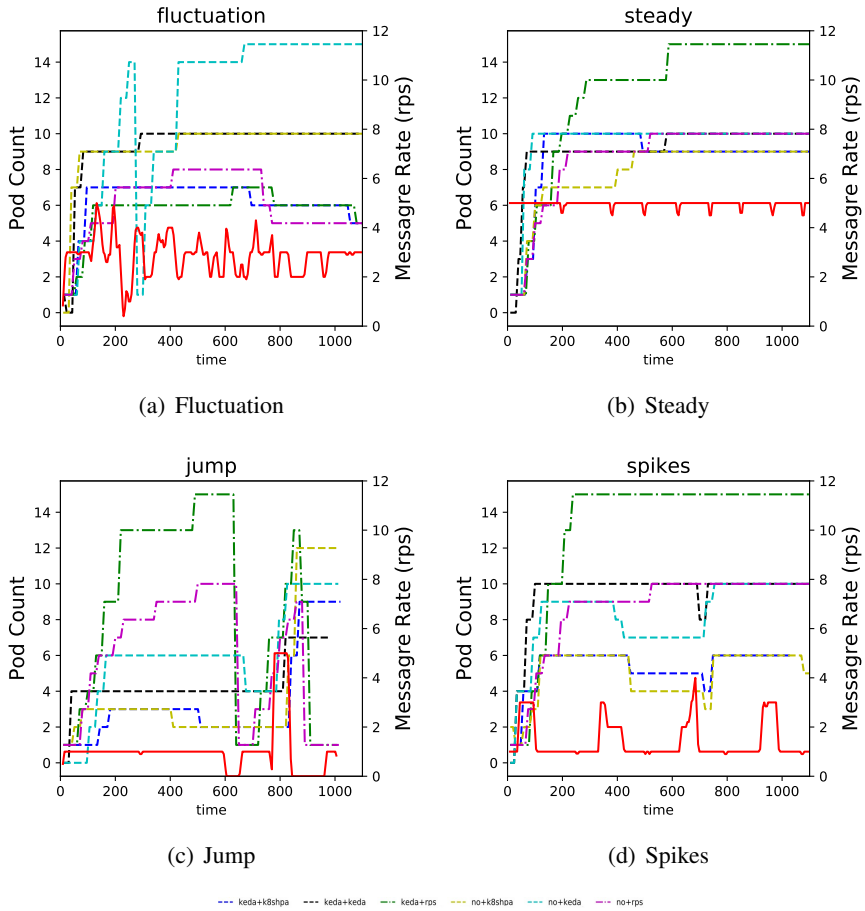


Figure 54. Success rate of PuhatuMonitoring application over scaling approaches



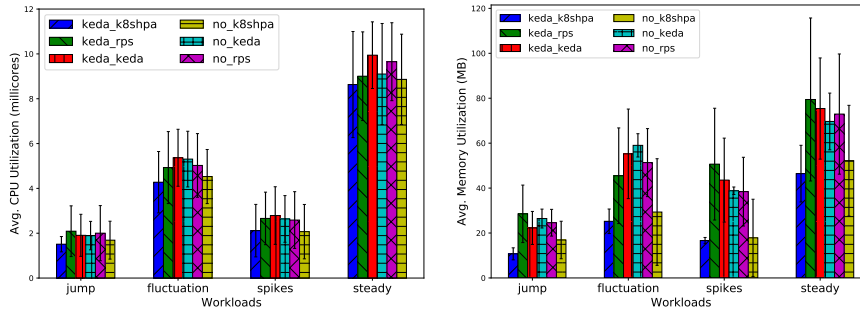
(a) Fluctuation (b) Steady

(c) Jump (d) Spikes

Figure 55. Scaling pattern of outlierDetection function message arrival rate

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 55. 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



(a) CPU utilization (millicore) (b) Memory utilization (MB)

Figure 56. Comparison of CPU and Memory utilization of PuhatuMonitoring application for various scaling approaches

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.

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.

Considering the CPU and memory utilization shown in Figure 56, for jump workload, *keda+k8shpa* used minimum average CPU (1.5 millicore) and memory (107.56MB) 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 178MB. 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 14. 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

Table 14. Overview of performance metrics of scaling approaches in PuhatuMonitoring application

Workload	**jump**						**steady**						**spikes**						**fluctuation**					
Message Queue	**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**
Metric																								
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
Succeded	**1048**	1035	1039	1027	1042	992	5775	5759	5785	5726	5840	5793	1644	1636	1642	1472	1632	1632	3386	3288	3409	3342	3356	3344
Suceess Rate	**98.9**	97.6	98.0	96.9	98.3	93.6	96.9	96.7	97.1	96.1	**98.0**	97.2	**99.9**	99.4	99.8	89.4	99.1	99.1	98.4	95.5	**99.0**	97.1	97.5	97.2
PT Sum	**766.66**	1029	785.13	1098	785.1	1136	5006	4564	5063.3	4951.77	**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	**0.80**	0.82	**0.49**	0.61	0.51	0.61	0.51	0.66	0.66	0.85	**0.66**	0.69	1.33	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	1.40	1.23	3.37	0.87	1.03	1.04	1.17	**0.83**	1.04	**0.24**	1.18	0.30	1.06	0.38	1.78	0.87	1.48	**0.81**	1.20	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	0.70	1.02	0.70	**0.68**	0.80	**0.70**	0.80	0.78	**0.72**	0.72	**0.45**	0.50	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**	0.10	**0.04**	0.11	0.04	0.10	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	2.00	2.09	1.89	**1.51**	1.89	1.91	9.65	9.01	8.86	**8.63**	9.10	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

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.

5.5.5. 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 the processing time (latency) and resource utilization. The latency is a primary key metric in IoT environments and is directly correlated to 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 based on the weighted average for two criteria, we used the following equation 5.4.

$$ScalingApproach_w = \alpha * Score(PT_w) + (1-\alpha) * Score(RU_w) \qquad (5.4)$$

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}.

Therefore, 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 13 and Table 14, we selected six metrics that are highly essential to estimate the processing time (latency) and three metrics that contribute to resource utilization. In addition, we also assign higher weights to metrics that are important considering the critical requirements of IoT applications similar to the approach used in the article [186][2]. 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 15. 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.

Table 15. Suitability analysis using weighted average scoring

	Weights		**Jump**		**Spikes**		**Steady**		**Fluctuation**	
α	Processing Time (Latency)	Resource Utilization	**Aeneas**	**Puhatu**	**Aeneas**	**Puhatu**	**Aeneas**	**Puhatu**	**Aeneas**	**Puhatu**
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
1.0	100%	0%	keda+k8shpa	no+keda	no+k8shpa	no+rps	keda+k8shpa	no+keda	no+keda	no+rps

Considering the results for different α values, applications, and workloads from Table 15, 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, in ***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. In the following section, we will discuss the experiences encountered during the experiments and corresponding future directions.

5.6. Experiences and Future Research Directions

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 Chapter is to gain a better understanding of the behavior of such scaling approaches and to offer insights to developers.

During the experiment design phase, a substantial amount of time was dedicated to brainstorming, experimenting, and analyzing the scaling configurations for KEDA, K8shpa, and RPS approaches for both MQT and serverless functions of two SDPs. It is crucial for administrators and developers to determine and finetune the ideal configurations of scaling components in order to reduce costs, latency, and resource consumption. This process presented numerous bottlenecks and challenges. For example, in the **KEDA-based** approach, deciding the target threshold of scaling metrics such as QueueLength and Message Rate was chal-

lenging. Because the optimal value of such metrics directly correlates to the concurrency limit of serverless functions and execution time for each invocation, this eventually can impact the success rate. In **K8shpa-based** approaches, several experiments were conducted to find the optimal CPU and memory request and limits for individual functions that fit the optimal concurrency limit. Further, choosing the target CPU threshold for each function in K8S HPA took moderate time. Otherwise, can lead to the use of more resources than required, and it's a disadvantage in fog environments. The decision of choosing the scaling rules **RPS-based** is quite critical, developers and administrators should have prior knowledge of the workload patterns and user behaviors, and it's challenging. The other issue was, the scaling rule differs for each individual function (for example, long running and short running time). Eventually, the performance of SDPs precisely depends on choosing the optimal scaling configurations and settings. Choosing scaling rules and configuration settings is challenging for complex applications, and needs an automated solution using state-of-the-art techniques [141]–[143] for choosing optimal configurations.

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 10. 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.

5.7. Summary

In this Chapter, 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.

6. CONCLUSION AND FUTURE DIRECTIONS

This chapter concludes the thesis by summarizing the key contributions. Further, it lists the limitations and proposes potential future research directions.

6.1. Summary of contributions

This thesis addresses the challenges of data processing over the IoT continuum. IoT data processing involves complex tasks from data acquisition to processing. To achieve low latency and meet other quality of service (QoS) metrics, edge and fog computing models are increasingly being adopted over cloud-based IoT data processing. This adoption adds complexity for dynamically executing data analysis tasks across varying distances and on heterogeneous hardware devices. Industries require lightweight approaches or frameworks to run data operations close to the data sources without compromising QoS demands. To address this issue, the goal of the thesis was threefold that address the three research questions, the first being to investigate how monolithic containers can be realized in IoT data processing with their drawbacks and benefits. Second, to explore serverless-based data pipeline-based approaches exclusively used for IoT continuum-based data processing. The third is concerned with the auto-scaling of serverless data pipelines. To achieve the three research goals, we have addressed the research questions as outlined below.

RQ1: How feasible is the utilization and adoption of container-based monolithic applications for IoT data processing in fog environments without encountering QoS bottlenecks?

To answer this RQ1, we formulated a strategy for designing data operations within monolithic containers and efficiently scheduling them across the IoT continuum, taking into account QoS parameters such as latency and energy consumption. The primary challenge was to achieve optimal scheduling decisions without compromising QoS metrics. Managing workloads intelligently in large-scale edge and fog platforms proves challenging due to the dynamic nature of modern workload applications and the specific responsiveness requirements of users. Our literature review in Chapter 3, indicates that existing state-of-the-art algorithms take more decision time and are not optimal in a volatile environment. So, we proposed a gradient-based backpropagation approach (GOBI) and GOBI* for fast and scalable scheduling and have shown that it outperforms state-of-the-art schedulers. To demonstrate this, we developed a framework[1], COSCO, which is the first to allow coupled simulation and container orchestration in IoT environments.

However, deploying monolithic containerized applications in event-driven IoT architectures presents its own set of challenges. To optimize QoS metrics such

[1] https://gitlab.doc.ic.ac.uk/st220/COSCO

as latency and energy consumption, we employed the concept of container migration. This involves moving application containers from overloaded hosts to under-loaded hosts during the scheduling and placement phase. Instead of relocating entire application containers, it would be more beneficial for the individual data operations service hosted on the hosts and routed based on QoS requirements, which is more beneficial in resource-constrained edge/fog architectures. Our experiments show that scheduling decisions definitively can yield optimal results; however, the migration approach needs extra bandwidth to move the entire encapsulated container application between nodes, which is challenging in edge/fog environments. Other challenges related to granular scaling of data operations, not the entire application containers, and each data operation execution-based billing provides more benefits in even-driven architectures. These challenges can be streamlined by unpacking a monolithic application into granular data operations by designing the data pipelines along with serverless technology.

RQ2: How can Serverless Data Pipelines are created in Fog and Edge computing environments, that are suitable to specific requirements of diverse IoT applications?

The answer to this question introduces a three-layered architecture that demonstrates the construction of serverless data pipelines (SDPs) for IoT data analytics. A significant challenge in this approach is managing intermediate data in the pipeline due to the stateless nature of serverless frameworks. Our literature review suggests various mechanisms to address this issue, such as object storage, message queues, or off-the-shelf data pipeline tools. However, IoT applications have diverse workload demands, including compute, bandwidth, and latency sensitivities. To tackle this, we designed and experimented with three different SDP approaches (Object Storage, Message Queues, and Data Flow Tools) using three different IoT workloads (compute-intensive, latency-sensitive, and bandwidth-intensive).

Furthermore, we proposed a suitability index that will guide the IoT developers in choosing the appropriate SDP based on the specific type of IoT application. Our experimental results, utilizing the suitability index, indicate that SDPs based on standard data flow tools (DFTs) are unsuitable for compute-intensive tasks like video processing but are efficient for bandwidth-intensive applications. Conversely, object storage service (OSS)-based SDPs are better suited for compute-intensive tasks, while message queue-based SDPs are appropriate for latency-sensitive tasks but not for compute and bandwidth-sensitive tasks due to higher CPU and memory utilization. Object storage service (OSS) based SDPs are better suitable for compute-intensive tasks and Message Queues based SDPs are suitable for latency-sensitive tasks but not for compute and bandwidth-sensitive tasks due to higher CPU and memory utilization.

RQ3: How can we leverage the existing auto-scaling approaches for scaling the serverless data pipelines that are suitable for diverse and dynamic work-

load patterns?

On one hand, IoT workloads are stochastic, volatile, and have diverse resource demands, reflecting the varying requirements of different IoT applications. On the other hand, our experimental analysis carried out in RQ2, indicates the need for auto-scaling SDPs to meet lower processing time and other QoS expectations. This is achieved through fine-grained scaling of serverless functions and effective handling of intermediate data. For example, SDPs based on message queues are composed of multiple components, such as serverless functions, message queues, and message queue triggers. Scaling the entire pipeline without leaving any bottlenecks is challenging.

To answer the RQ3, we applied two reactive methods of auto-scaling techniques on the SDP components. We considered stochastic workloads for three real-time applications with different resource demands. Further, we proposed a suitability index used to choose the auto-scaling method for the particular intensity of the workload. Our suitability analysis of reactive auto-scaling mechanisms for SDP under four different workload patterns indicates 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.

6.2. Future directions

Experiments in the thesis were carried out within a controlled real-world setting, however, can be extended to large-scale IoT setup which can improve the accuracy of the system. The proposed methodology described in the COSCO framework is limited to CPU and memory requirements of the applications, and bandwidth is not considered for scheduling optimization. Bandwidth-aware optimization offers a broader scope in scheduling, especially for IoT applications. Another limitation is that the workflow or dependency of IoT tasks is not considered in this study. The thesis focuses on only three designs of SDPs, narrowing the scope to specific applications. However, other approaches, such as in-memory-based pipelines, may be quite efficient in IoT data processing. We assumed the single serverless service provider, however federation of serverless provider ecosystems can yield more benefits in terms of availability, cost, and other aspects. The thesis did not count the cost as one of the quality parameters. The thesis limits only three auto-scaling mechanisms applied on message-queue-based SDP, but this limits the scope.

Based on the limitations as discussed, in the following we walk through the future directions that can potentially be taken forward in the IoT data processing research.

1. Alternative SDP design approaches

In the design of serverless data pipelines, three approaches have been explored— DFT, Object Storage, and Message Queue. The DFT-based approach

employs Apache NiFi, a centralized data pipeline engine, to manage controlled data movement centralized within Apache NiFi nodes. However, this method may consume more bandwidth as it involves moving the entire data instead of controlling or controlling the location of the function. Object storage, on the other hand, may experience performance challenges due to increased read and write operations on the disk. Message queues, while effective for small data units, have limitations. This prompts an opportunity to investigate the use of in-memory databases as intermediate data storage units and explore other lightweight, low-bandwidth consumption approaches. For instance, utilizing function chaining by managing function invocation through a master function or main thread could be explored.

2. Reactive scaling methods

The auto-scaling of the serverless data pipeline has provided multiple research directions based on the experimental analysis in Contribution 2. Optimal resource configurations, concurrency limits, and other function settings in the SDP scaling have been chosen by running several experiments, which can lead to extra costs in the design. It is typically challenging for IoT developers to select the optimal resource configurations, concurrency limits, and other function settings in the SDP during their design. This can be accomplished through the use of statistical and machine learning-based optimization techniques. Furthermore, the QueueLength and Message Rate thresholds for scaling the MQT can be improved by utilizing these methods. As a result, optimal scaling decisions for the MQT and serverless functions can be achieved through reactive mechanisms, using MAPE-K based approach, machine learning, and other optimization techniques.

3. SDP Modelling and dynamic deployment

Serverless data pipelines offer an efficient approach for IoT data processing in resource-constrained and heterogeneous IoT environments. Our experience in the design of serverless data pipelines (SDPs) provides research opportunities to model these pipelines using TOSCA-based service templates. Furthermore, dynamic deployment and orchestration on edge/fog environments present a significant potential, reducing design time and enabling the reuse of serverless data pipeline components by IoT developers. In our experiments, we focused on a single serverless provider, which can be monotonous. However, exploring multiple serverless service providers adds complexity to the context. This introduces key challenges related to dealing with federation and interoperability of multiple serverless providers in edge/fog environments, particularly when leveraged in Serverless Data Pipelines (SDP). In such scenarios, functions can be executed based on factors like cost, reliability, fault tolerance and other QoS parameters.

4. Alternative to serverless approach

Recently, there has been considerable interest in Web Assembly (Wasm) due

to its runtime performance and lower latency compared to applications running in serverless environments. This opens up a new research direction, exploring the potential of Web Assembly-based pipelines and serverless-based pipelines for efficient IoT data processing.

6.3. Final Remarks

Overall, this thesis addresses the complexities and challenges in the processing of IoT data while shifting from monolithic container architectures to serverless computing models for handling IoT data. The contributions assist developers, researchers, practitioners, and other associated stakeholders in selecting the most suitable data processing mechanism, considering factors such as computing resources, bandwidth, energy consumption, and latency while meeting sensitive QoS requirements. The thesis produced three contributions that solve the problem of container and serverless based IoT data processing approaches. The first part of our contribution resulted in the creation of a Python-based simulator and framework. This tool is designed for the scientific research community and developers to demonstrate novel scheduling algorithms for container-based data processing in edge and cloud environments.

The second part of the contribution resulted in a lightweight, serverless-based data processing and pipeline framework featuring three data handling mechanisms suited for compute, bandwidth, and memory-intensive applications. For IoT developers, this research contribution offers a suitability analysis to help choose the reliable data processing approach for their specific use cases, such as video analysis or smart environment monitoring. We demonstrated the research contribution through its application to an environmental monitoring project deployed in the Puhatu region in Northwest Estonia. The third part of the contribution guides the IoT developers to choose the suitable scaling approach based on IoT use case requirements for stochastic workloads (steady, jump, fluctuations).

BIBLIOGRAPHY

[1] S. Tuli, S. R. Poojara, S. N. Srirama, G. Casale, and N. R. Jennings, "Cosco: Container orchestration using co-simulation and gradient based optimization for fog computing environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 101–116, 2021.

[2] **Shivananda R Pojara**, C. K. Dehury, P. Jakovits, and S. N. Srirama, "Serverless data pipeline approaches for iot data in fog and cloud computing," *Future Generation Computer Systems*, vol. 130, pp. 91–105, 2022.

[3] S. Poojara, C. K. Dehury, P. Jakovits, and S. N. Srirama, "Serverless data pipelines for iot data analytics: A cloud vendors perspective and solutions," in *Predictive Analytics in Cloud, Fog, and Edge Computing: Perspectives and Practices of Blockchain, IoT, and 5G*, Springer, 2022, pp. 107–132.

[4] S. Poojara, A. Jõeleht, P. Jakovits, and S. N. Srirama, "Serverless outlier management for environmental iot data-a case study of puhatumonitoring," in *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, IEEE, 2022, pp. 1–7.

[5] **Shivananda R Pojara**, C. K. Dehury, P. Jakovits, and S. N. Srirama, "Scaling approaches for serverless data pipelines in fog computing environments: A performance evaluation,"

[6] V. K. Quy, N. V. Hau, D. V. Anh, and L. A. Ngoc, "Smart healthcare IoT applications based on fog computing: Architecture, applications and challenges," en, *Complex & Intelligent Systems*, Nov. 2021, ISSN: 2198-6053. DOI: 10.1007/s40747-021-00582-9. [Online]. Available: https://doi.org/10.1007/s40747-021-00582-9 (visited on 07/18/2022).

[7] P. Bellini, P. Nesi, and G. Pantaleo, "IoT-Enabled Smart Cities: A Review of Concepts, Frameworks and Key Technologies," en, *Applied Sciences*, vol. 12, no. 3, p. 1607, Jan. 2022, Number: 3 Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2076-3417. DOI: 10.3390/app12031607. [Online]. Available: https://www.mdpi.com/2076-3417/12/3/1607 (visited on 07/18/2022).

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

[9] F. B. Insights, *Internet of Things (IoT) Market to Exhibit 25.4% CAGR till 2028; Critical Need to Virtually Monitor Operations to Boost Growth: Fortune Business Insights™*, en, Aug. 2021. [Online]. Available: https://www.globenewswire.com/en/news-release/2021/08/23/2284666/0/en/Internet-of-Things-IoT-Market-to-Exhibit-25-4-CAGR-till-2028-Critical-Need-to-Virtually-Monitor-

Operations - to - Boost - Growth - Fortune - Business - Insights . html (visited on 07/18/2022).

[10] A. Hernandez, B. Xiao, and V. Tudor, "Eraia-enabling intelligence data pipelines for iot-based application systems," *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pp. 1–9, 2020, Publisher: IEEE.

[11] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, *et al.*, "Serverless edge computing: Vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[12] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic Data Stream Processing," *Future Generation Computer Systems*, vol. 87, pp. 171–185, 2018, ISSN: 0167-739X. DOI: https : / / doi . org / 10 . 1016 / j . future . 2018 . 05 . 025. [Online]. Available: https : / / www . sciencedirect . com / science / article / pii/S0167739X17326821.

[13] I. Baldini, P. Castro, K. Chang, and others, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, Springer, 2017, pp. 1–20.

[14] G. Casale, M. Artač, W.-J. van den Heuvel, *et al.*, "RADON: Rational decomposition and orchestration for serverless computing," *SICS Software-Intensive Cyber-Physical Systems*, pp. 1–11, 2019, Publisher: Springer.

[15] A. Mampage, S. Karunasekera, and R. Buyya, "A Holistic View on Resource Management in Serverless Computing Environments: Taxonomy and Future Directions," *ACM Computing Surveys*, Jan. 2022, Just Accepted, ISSN: 0360-0300. DOI: 10 . 1145 / 3510412. [Online]. Available: https://doi.org/10.1145/3510412 (visited on 07/18/2022).

[16] R. Buyya and S. N. Srirama, *Fog and edge computing: principles and paradigms*. John Wiley & Sons, 2019.

[17] C. Chang, S. N. Srirama, and R. Buyya, "Internet of Things (IoT) and new computing paradigms," *Fog and edge computing: principles and paradigms*, vol. 6, pp. 1–23, 2019, Publisher: Springer.

[18] R. Buyya, S. N. Srirama, G. Casale, and others, "A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade," vol. 51, no. 5, 2019.

[19] A. Taherkordi, F. Eliassen, and G. Horn, "From IoT big data to IoT big services," in *Proceedings of the Symposium on Applied Computing*, 2017, pp. 485–491.

[20] C.-H. Hong and B. Varghese, "Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms," *ACM Computing Surveys*, vol. 52, no. 5, 97:1–97:37, Sep. 2019, ISSN: 0360-0300. DOI: 10.1145/3326066. [Online]. Available: https://doi.org/10.1145/3326066 (visited on 07/18/2022).

[21] B. Cheng, J. Fuerst, G. Solmaz, and T. Sanada, “Fog function: Serverless fog computing for data intensive iot services,” in *2019 IEEE International Conference on Services Computing (SCC)*, IEEE, 2019, pp. 28–35.

[22] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, *et al.*, “Serverless Computing: One Step Forward, Two Steps Back,” *CoRR*, vol. abs/1812.03651, 2018, arXiv: 1812.03651.

[23] G. A. S. Cassel, V. F. Rodrigues, R. da Rosa Righi, M. R. Bez, A. C. Nepomuceno, and C. A. da Costa, “Serverless computing for Internet of Things: A systematic literature review,” *Future Generation Computer Systems*, vol. 128, pp. 299–316, 2022, Publisher: Elsevier.

[24] V. Kjorveziroski, S. Filiposka, and V. Trajkovik, “IoT Serverless Computing at the Edge: A Systematic Mapping Review,” *Computers*, vol. 10, no. 10, p. 130, 2021, Publisher: MDPI.

[25] N. E. Ioini, D. Hästbacka, C. Pahl, and D. Taibi, “Platforms for serverless at the edge: A review,” in *European Conference on Service-Oriented and Cloud Computing*, Springer, 2020, pp. 29–40.

[26] X. Yao, N. Chen, X. Yuan, and P. Ou, “Performance Optimization in Serverless Edge Computing Environment using DRL-Based Function Offloading,” in *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, IEEE, 2022, pp. 1390–1395.

[27] A. Bocci, S. Forti, G.-L. Ferrari, and A. Brogi, “Secure FaaS orchestration in the fog: How far are we?” *Computing*, vol. 103, no. 5, pp. 1025–1056, 2021, Publisher: Springer.

[28] N. P. Shah, “Design of a Reference Architecture for Serverless IoT Systems,” in *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, IEEE, 2021, pp. 1–6.

[29] D. Bermbach, S. Maghsudi, J. Hasenburg, and T. Pfandzelter, “Towards auction-based function placement in serverless fog platforms,” in *2020 IEEE International Conference on Fog Computing (ICFC)*, IEEE, 2020, pp. 25–31.

[30] X. Yao, N. Chen, X. Yuan, and P. Ou, “Performance Optimization in Serverless Edge Computing Environment using DRL-Based Function Offloading,” in *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, IEEE, 2022, pp. 1390–1395.

[31] K. Boloor, R. Chirkova, Y. Viniotis, and T. Salo, “Dynamic request allocation and scheduling for context aware applications subject to a percentile response time sla in a distributed cloud,” in *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, IEEE, 2010, pp. 464–472.

[32] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18, New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 263–274, ISBN: 978-1-4503-6011-1. DOI: 10.1145/3267809.3267815. [Online]. Available: https://doi.org/10.1145/3267809.3267815 (visited on 07/18/2022).

[33] P. Grzesik, D. R. Augustyn, Wyciślik, and D. Mrozek, "Serverless computing in omics data analysis and integration," *Briefings in Bioinformatics*, vol. 23, no. 1, bbab349, Jan. 2022, ISSN: 1477-4054. DOI: 10.1093/bib/bbab349. [Online]. Available: https://doi.org/10.1093/bib/bbab349 (visited on 07/18/2022).

[34] C. Dehury, P. Jakovits, S. N. Srirama, V. Tountopoulos, and G. Giotis, "Data Pipeline Architecture for Serverless Platform," en, in *Software Architecture*, H. Muccini, P. Avgeriou, B. Buhnova, *et al.*, Eds., ser. Communications in Computer and Information Science, Cham: Springer International Publishing, 2020, pp. 241–246, ISBN: 978-3-030-59155-7. DOI: 10.1007/978-3-030-59155-7_18.

[35] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "{SONIC}: Application-aware Data Passing for Chained Serverless Applications," 2021, pp. 285–301, ISBN: 978-1-939133-23-6.

[36] T. Elgamal, "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct. 2018, pp. 300–312. DOI: 10.1109/SEC.2018.00029.

[37] M. Zhang, F. Wang, Y. Zhu, J. Liu, and Z. Wang, "Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines," in *Proceedings of the 12th ACM Multimedia Systems Conference*, ser. MMSys '21, New York, NY, USA: Association for Computing Machinery, Jul. 2021, pp. 80–93, ISBN: 978-1-4503-8434-6. DOI: 10.1145/3458305.3463377. [Online]. Available: https://doi.org/10.1145/3458305.3463377 (visited on 07/18/2022).

[38] E. Al-Masri, I. Diabate, R. Jain, M. H. L. Lam, and S. R. Nathala, "A Serverless IoT Architecture for Smart Waste Management Systems," in *2018 IEEE International Conference on Industrial Internet (ICII)*, Oct. 2018, pp. 179–180. DOI: 10.1109/ICII.2018.00034.

[39] T. Pfandzelter and D. Bermbach, "IoT Data Processing in the Fog: Functions, Streams, or Batch Processing?" In *2019 IEEE International Conference on Fog Computing (ICFC)*, Jun. 2019, pp. 201–206. DOI: 10.1109/ICFC.2019.00033.

[40] S. Nastic, T. Rausch, O. Scekic, *et al.*, "A Serverless Real-Time Data Analytics Platform for Edge Computing," *IEEE Internet Computing*, vol. 21,

no. 4, pp. 64–71, 2017, Conference Name: IEEE Internet Computing, ISSN: 1941-0131. DOI: 10.1109/MIC.2017.2911430.

[41] S. C. Palepu, D. Chahal, M. Ramesh, and R. Singhal, "Benchmarking the Data Layer Across Serverless Platforms," in *Proceedings of the 2nd Workshop on High Performance Serverless Computing*, ser. HiPS '22, New York, NY, USA: Association for Computing Machinery, Jun. 2022, pp. 3–7, ISBN: 978-1-4503-9311-9. DOI: 10.1145/3526060.3535460. [Online]. Available: https://doi.org/10.1145/3526060.3535460 (visited on 07/18/2022).

[42] A. Sabbioni, L. Rosa, A. Bujari, L. Foschini, and A. Corradi, "A Shared Memory Approach for Function Chaining in Serverless Platforms," in *2021 IEEE Symposium on Computers and Communications (ISCC)*, ISSN: 2642-7389, Sep. 2021, pp. 1–6. DOI: 10.1109/ISCC53001.2021.9631385.

[43] R. Buyya and S. N. Srirama, *Fog and edge computing: principles and paradigms*. John Wiley & Sons, 2019.

[44] C. Chang, S. N. Srirama, and R. Buyya, "Internet of things (iot) and new computing paradigms," *Fog and edge computing: principles and paradigms*, vol. 6, pp. 1–23, 2019.

[45] R. Buyya, S. N. Srirama, G. Casale, *et al.*, "A manifesto for future generation cloud computing: Research directions for the next decade," vol. 51, no. 5, 2019.

[46] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, *et al.*, "Serverless computing: One step forward, two steps back," *CoRR*, vol. abs/1812.03651, 2018. arXiv: 1812.03651. [Online]. Available: http://arxiv.org/abs/1812.03651.

[47] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto, "Comparison of faas orchestration systems," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, IEEE, 2018, pp. 148–153.

[48] V. Ivanov and K. Smolander, "Implementation of a devops pipeline for serverless applications," in *International Conference on Product-Focused Software Process Improvement*, Springer, 2018, pp. 48–64.

[49] B. Ahmed, B. Seghir, M. Al-Osta, and G. Abdelouahed, "Container based resource management for data processing on iot gateways," *Procedia Computer Science*, vol. 155, pp. 234–241, 2019.

[50] P. Mendki, "Docker container based analytics at iot edge video analytics usecase," in *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, IEEE, 2018, pp. 1–4.

[51] L. Sanabria-Russo, D. Pubill, J. Serra, and C. Verikoukis, "Iot data analytics as a network edge service," in *IEEE INFOCOM 2019-IEEE Confer-*

ence on Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2019, pp. 969–970.

[52] M. Adhikari, M. Mukherjee, and S. N. Srirama, "Dpto: A deadline and priority-aware task offloading in fog computing framework leveraging multilevel feedback queueing," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 5773–5782, 2019.

[53] A. Hazra, M. Adhikari, T. Amgoth, and S. N. Srirama, "Intelligent service deployment policy for next-generation industrial edge networks," *IEEE Transactions on Network Science and Engineering*, 2021.

[54] J. Scheuner, S. Eismann, S. Talluri, *et al.*, "Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," *arXiv preprint arXiv:2205.07696*, 2022.

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

[56] M. Xu, C. Song, S. Ilager, *et al.*, "Coscal: Multi-faceted scaling of microservices with reinforcement learning," *IEEE Transactions on Network and Service Management*, 2022.

[57] W. Lv, Q. Wang, P. Yang, *et al.*, "Microservice deployment in edge computing based on deep q learning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2968–2978, 2022.

[58] N. C. Coulson, S. Sotiriadis, and N. Bessis, "Adaptive microservice scaling for elastic applications," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4195–4202, 2020.

[59] S. N. Srirama, "A Decade of Research in Fog computing: Relevance, Challenges, and Future Directions," *Software: Practice and Experience*, 2023, ISSN: 1097-024X. DOI: 10.1002/SPE.3243.

[60] L. Schuler, S. Jamil, and N. Kühl, "Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, IEEE, 2021, pp. 804–811.

[61] P. Benedetti, M. Femminella, G. Reali, and K. Steenhaut, "Reinforcement learning applicability for resource-based auto-scaling in serverless edge applications," in *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events*, 2022, pp. 674–679. DOI: 10.1109/PerComWorkshops53856.2022.9767437.

[62] X. Li, P. Kang, J. Molone, W. Wang, and P. Lama, "Kneescale: Efficient resource scaling for serverless computing at the edge," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 180–189. DOI: 10.1109/CCGrid54584.2022.00027.

[63] A. P. Jegannathan, R. Saha, and S. K. Addya, "A time series forecasting approach to minimize cold start time in cloud-serverless platform," in

2022 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom), IEEE, 2022, pp. 325–330.

[64] M. Gotin, F. L¨osch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in cloudiot-environments," in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18, Berlin, Germany: Association for Computing Machinery, 2018, 157–167, ISBN: 9781450350952. DOI: 10.1145/3184407.3184430. [Online]. Available: https://doi.org/10.1145/3184407.3184430.

[65] N. Mahmoudi and H. Khazaei, "Performance Modeling of Metric-Based Serverless Computing Platforms," *IEEE Transactions on Cloud Computing*, pp. 1–13, 2022. DOI: 10.1109/TCC.2022.3169619.

[66] K Aruna and G Pradeep, "Performance and scalability improvement using iot-based edge computing container technologies," *SN Computer Science*, vol. 1, pp. 1–7, 2020.

[67] M. Al-Rakhami, M. Alsahli, M. M. Hassan, A. Alamri, A. Guerrieri, and G. Fortino, "Cost efficient edge intelligence framework using docker containers," in *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, IEEE, 2018, pp. 800–807.

[68] Z. Wang, M. Goudarzi, J. Aryal, and R. Buyya, "Container orchestration in edge and fog computing environments for real-time iot applications," in *Computational Intelligence and Data Analytics: Proceedings of ICCIDA 2022*, Springer, 2022, pp. 1–21.

[69] S. Tuli, S. Ilager, K. Ramamohanarao, and R. Buyya, "Dynamic Scheduling for Stochastic Edge-Cloud Computing Environments using A3C learning and Residual Recurrent Neural Networks," *IEEE TMC*, 2020.

[70] N. Wang, M. Matthaiou, D. S. Nikolopoulos, and B. Varghese, "DYVERSE: dynamic vertical scaling in multi-tenant edge environments," *Future Generation Computer Systems*, vol. 108, pp. 598–612, 2020.

[71] D. Basu, X. Wang, Y. Hong, H. Chen, and S. Bressan, "Learn-as-you-go with megh: Efficient live migration of virtual machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1786–1801, 2019.

[72] K. Han, Z. Xie, and X. LV, "Fog computing task scheduling strategy based on improved genetic algorithm," *Computer Science*, vol. 4, p. 22, 2018.

[73] O. Skarlat, M. Nardelli, S. Schulte, M. Borkowski, and P. Leitner, "Optimized IoT service placement in the fog," *Service Oriented Computing and Applications*, vol. 11, no. 4, pp. 427–443, 2017.

[74] X.-Q. Pham, N. D. Man, N. D. T. Tri, N. Q. Thai, and E.-N. Huh, "A cost- and performance-effective approach for task scheduling based on collaboration between cloud and fog computing," *International Journal of Distributed Sensor Networks*, vol. 13, no. 11, p. 1 550 147 717 742 073, 2017.

[75] T. Choudhari, M. Moh, and T.-S. Moh, "Prioritized task scheduling in fog computing," in *The ACMSE 2018 Conference*, 2018, pp. 1–8.

[76] S. Liu and N. Wang, "Collaborative optimization scheduling of cloud service resources based on improved genetic algorithm," *IEEE Access*, vol. 8, pp. 150 878–150 890, 2020.

[77] M. De Donno, K. Tange, and N. Dragoni, "Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog," *IEEE Access*, vol. 7, pp. 150 936–150 948, 2019. DOI: 10.1109/ACCESS.2019.2947652.

[78] L. Bittencourt, R. Immich, R. Sakellariou, *et al.*, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3, pp. 134–155, 2018.

[79] P. Kochovski, R. Sakellariou, M. Bajec, P. Drobintsev, and V. Stankovski, "An architecture and stochastic method for database container placement in the edge-fog-cloud continuum," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 396–405. DOI: 10.1109/IPDPS.2019.00050.

[80] D. Kimovski, R. Mathá, J. Hammer, N. Mehran, H. Hellwagner, and R. Prodan, "Cloud, fog, or edge: Where to compute?" *IEEE Internet Computing*, vol. 25, no. 4, pp. 30–36, 2021.

[81] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM computer communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

[82] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *2015 Third IEEE workshop on hot topics in web systems and technologies (HotWeb)*, IEEE, 2015, pp. 73–78.

[83] P. Mendki, "Docker container based analytics at iot edge video analytics usecase," in *2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU)*, 2018, pp. 1–4. DOI: 10.1109/IoT-SIU.2018.8519852.

[84] T. Zois, *The Evolution of Serverless Services*. Feb. 2021. DOI: 10.13140/RG.2.2.33924.86407.

[85] J. McChesney, N. Wang, A. Tanwer, E. De Lara, and B. Varghese, "Defog: Fog computing benchmarks," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 47–58.

[86] E. G. Renart, D. Balouek-Thomert, and M. Parashar, "Edge based data-driven pipelines," *arXiv preprint arXiv:1808.01353*, 2018.

[87] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The Serverless Computing Survey: A Technical Primer for Design Architecture," *ACM Computing Surveys*, Dec. 2021, Just Accepted, ISSN: 0360-0300. DOI: 10.1145/3508360. [Online]. Available: https://doi.org/10.1145/3508360 (visited on 07/21/2022).

[88] H. B. Hassan, S. A. Barakat, and Q. I. Sarhan, "Survey on serverless computing," *Journal of Cloud Computing*, vol. 10, no. 1, p. 39, Jul. 2021, ISSN: 2192-113X. DOI: 10.1186/s13677-021-00253-7. [Online]. Available: https://doi.org/10.1186/s13677-021-00253-7 (visited on 07/21/2022).

[89] V. M. Research, *Serverless Architecture Market size worth $ 36.84 Billion, Globally, by 2028 at 21.71% CAGR: Verified Market Research®*, en. [Online]. Available: https://www.prnewswire.com/news-releases/serverless-architecture-market-size-worth--36-84-billion-globally-by-2028-at-21-71-cagr-verified-market-research-301479695.html (visited on 07/22/2022).

[90] *The Serverless Computing Survey: A Technical Primer for Design Architecture | ACM Computing Surveys*. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3508360 (visited on 07/21/2022).

[91] O. Ltd, *Home*, en. [Online]. Available: https://www.openfaas.com/ (visited on 07/22/2022).

[92] *Apache OpenWhisk is a serverless, open source cloud platform*. [Online]. Available: https://openwhisk.apache.org/ (visited on 07/22/2022).

[93] *Cloud Functions*, en. [Online]. Available: https://cloud.google.com/functions (visited on 07/22/2022).

[94] ggailey777, *Azure Functions documentation*, en-us. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/ (visited on 07/22/2022).

[95] *Serverless Computing - AWS Lambda - Amazon Web Services*, en-US. [Online]. Available: https://aws.amazon.com/lambda/ (visited on 07/22/2022).

[96] *IBM Cloud Functions*. [Online]. Available: https://cloud.ibm.com/functions/ (visited on 07/22/2022).

[97] A. Raj, J. Bosch, H. H. Olsson, and T. J. Wang, "Modelling data pipelines," in *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2020, pp. 13–20. DOI: 10.1109/SEAA51224.2020.00014.

[98] A. S. Foundation, *Apache nifi*, 2020 (accessed December 13, 2020). [Online]. Available: https://nifi.apache.org/.

[99] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 17, e5668, 2020.

[100] R. Morabito and N. Beijar, "Enabling data processing at the network edge through lightweight virtualization technologies," in *2016 IEEE International Conference on Sensing, Communication and Networking (SECON Workshops)*, 2016, pp. 1–6. DOI: 10.1109/SECONW.2016.7746807.

[101] C. Puliafito, A. Virdis, and E. Mingozzi, "The impact of container migration on fog services as perceived by mobile things," in *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2020, pp. 9–16. DOI: 10.1109/SMARTCOMP50058.2020.00022.

[102] K. Kaur, F. Guillemin, and F. Sailhan, "Container placement and migration strategies for cloud, fog, and edge data centers: A survey," *International Journal of Network Management*, vol. 32, no. 6, e2212, 2022.

[103] M. M. Mahmoud, J. J. Rodrigues, K. Saleem, J. Al-Muhtadi, N. Kumar, and V. Korotaev, "Towards energy-aware fog-enabled cloud of things for healthcare," *Computers & Electrical Engineering*, vol. 67, pp. 58–69, 2018.

[104] A. A. Mutlag, M. K. Abd Ghani, N. a. Arunkumar, M. A. Mohammed, and O. Mohd, "Enabling technologies for fog computing in healthcare iot systems," *Future Generation Computer Systems*, vol. 90, pp. 62–78, 2019.

[105] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 13, pp. 1397–1420, 2012.

[106] Z. Tang, X. Zhou, F. Zhang, W. Jia, and W. Zhao, "Migration modeling and learning algorithms for containers in fog computing," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 712–725, 2018.

[107] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[108] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya, "Fogbus: A blockchain-based lightweight framework for edge and fog computing," *Journal of Systems and Software*, vol. 154, pp. 22–36, 2019.

[109] S. Bosmans, S. Mercelis, J. Denil, and P. Hellinckx, "Testing iot systems using a hybrid simulation based testing approach," *Computing*, vol. 101, no. 7, pp. 857–872, 2019.

[110] B. S. Onggo, N. Mustafee, A. Smart, A. A. Juan, and O. Molloy, "Symbiotic simulation system: Hybrid systems model meets big data analytics," in *2018 Winter Simulation Conference (WSC)*, IEEE, 2018, pp. 1358–1369.

[111] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese, "Defog: Fog computing benchmarks," in *The 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 47–58.

[112] S. Shen, V. van Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE, 2015, pp. 465–474.

[113] S. S. Gill, S. Tuli, M. Xu, *et al.*, "Transformative effects of IoT, Blockchain and Artificial Intelligence on cloud computing: Evolution, vision, trends and open challenges," *Internet of Things*, vol. 8, pp. 100–118, 2019.

[114] P. P. Ray, "A survey of iot cloud platforms," *Future Computing and Informatics Journal*, vol. 1, no. 1-2, pp. 35–46, 2016.

[115] S. Narayana Srirama, "A decade of research in fog computing: Relevance, challenges, and future directions," *arXiv e-prints*, arXiv–2305, 2023.

[116] S. P. Singh, A. Nayyar, R. Kumar, and A. Sharma, "Fog computing: From architecture to edge computing and big data processing," *The Journal of Supercomputing*, vol. 75, pp. 2070–2105, 2019.

[117] T. Pfandzelter and D. Bermbach, "Iot data processing in the fog: Functions, streams, or batch processing?" In *2019 IEEE International conference on fog computing (ICFC)*, IEEE, 2019, pp. 201–206.

[118] M. Goudarzi, M. Palaniswami, and R. Buyya, "Scheduling iot applications in edge and fog computing environments: A taxonomy and future directions," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–41, 2022.

[119] P. Varshney and Y. Simmhan, "Characterizing application scheduling on edge, fog, and cloud computing resources," *Software: Practice and Experience*, vol. 50, no. 5, pp. 558–595, 2020.

[120] J. Mass, C. Chang, and S. N. Srirama, "Context-aware edge process management for mobile thing-to-fog environment," in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, 2018, pp. 1–7.

[121] E. Ahvar, A.-C. Orgerie, and A. Lebre, "Estimating energy consumption of cloud, fog, and edge computing infrastructures," *IEEE Transactions on Sustainable Computing*, vol. 7, no. 2, pp. 277–288, 2019.

[122] S. K. Mishra, D. Puthal, J. J. Rodrigues, B. Sahoo, and E. Dutkiewicz, "Sustainable service allocation using a metaheuristic technique in a fog server for industrial applications," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4497–4506, 2018.

[123] V. Moysiadis, P. Sarigiannidis, I. Moscholios, *et al.*, "Towards distributed data management in fog computing," *Wireless Communications and Mobile Computing*, vol. 2018, 2018.

[124] H. Watanabe, T. Sato, T. Kondo, and F. Teraoka, "Afc: A mechanism for distributed data processing in edge/fog computing," in *2021 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2021, pp. 01–07.

[125] S. Mirampalli, R. Wankar, and S. N. Srirama, "Evaluating nifi and mqtt based serverless data pipelines in fog computing environments," *Future Generation Computer Systems*, vol. 150, pp. 341–353, 2024.

[126] S. Yang, "Iot stream processing and analytics in the fog," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 21–27, 2017.

[127] G. A. S. Cassel, V. F. Rodrigues, R. da Rosa Righi, M. R. Bez, A. C. Nepomuceno, and C. A. da Costa, "Serverless computing for internet of things: A systematic literature review," *Future Generation Computer Systems*, vol. 128, pp. 299–316, 2022.

[128] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the fog: A performance evaluation," *Sensors*, vol. 19, no. 7, p. 1488, 2019.

[129] V. Cardellini, F. Lo Presti, M. Nardelli, and G. Russo Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Generation Computer Systems*, vol. 87, pp. 171–185, 2018, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.05.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17326821.

[130] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, *et al.*, "Serverless edge computing: Vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, 2021, pp. 1–10.

[131] A. Hernandez, B. Xiao, and V. Tudor, "Eraia - enabling intelligence data pipelines for iot-based application systems," in *2020 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2020, pp. 1–9. DOI: 10.1109/PerCom45495.2020.9127385.

[132] I. Baldini, P. Castro, K. Chang, *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, Springer, 2017, pp. 1–20.

[133] J. M. Hellerstein, J. M. Faleiro, J. E. Gonzalez, *et al.*, "Serverless computing: One step forward, two steps back," *CoRR*, vol. abs/1812.03651, 2018. arXiv: 1812.03651.

[134] B. Cheng, J. Fuerst, G. Solmaz, and T. Sanada, "Fog function: Serverless fog computing for data intensive iot services," in *2019 IEEE International Conference on Services Computing (SCC)*, IEEE, 2019, pp. 28–35.

[135] V. E. Alvear-Puertas, Y. A. Burbano-Prado, P. D. Rosero-Montalvo, P. Tözün, F. Marcillo, and W. Hernandez, "Smart and portable air-quality monitoring iot low-cost devices in ibarra city, ecuador," *Sensors*, vol. 22, no. 18, 2022, ISSN: 1424-8220. DOI: 10.3390/s22187015. [Online]. Available: https://www.mdpi.com/1424-8220/22/18/7015.

[136] M. Syafrudin, G. Alfian, N. L. Fitriyani, and J. Rhee, "Performance analysis of iot-based sensor, big data processing, and machine learning model for real-time monitoring system in automotive manufacturing," *Sensors*,

vol. 18, no. 9, 2018, ISSN: 1424-8220. DOI: 10.3390/s18092946. [Online]. Available: https://www.mdpi.com/1424-8220/18/9/2946.

[137] M.-Q. Tran, M. Elsisi, M.-K. Liu, *et al.*, "Reliable deep learning and iot-based monitoring system for secure computer numerical control machines against cyber-attacks with experimental verification," *IEEE Access*, vol. 10, pp. 23 186–23 197, 2022. DOI: 10.1109/ACCESS.2022.3153471.

[138] M. P. Loria, M. Toja, V. Carchiolo, and M. Malgeri, "An efficient real-time architecture for collecting iot data," in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2017, pp. 1157–1166. DOI: 10.15439/2017F381.

[139] "A microservice architecture for real-time iot data processing: A reusable web of things approach for smart ports," *Computer Standards Interfaces*, vol. 81, p. 103 604, 2022, ISSN: 0920-5489. DOI: https://doi.org/10.1016/j.csi.2021.103604. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0920548921000994.

[140] N. Sai Lohitha and M. Pounambal, "Integrated publish/subscribe and push-pull method for cloud based iot framework for real time data processing," *Measurement: Sensors*, vol. 27, p. 100 699, 2023, ISSN: 2665-9174. DOI: https://doi.org/10.1016/j.measen.2023.100699. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2665917423000351.

[141] A. Raza, N. Akhtar, V. Isahagian, I. Matta, and L. Huang, "Configuration and placement of serverless applications using statistical learning," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2023. DOI: 10.1109/TNSM.2023.3254437.

[142] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020, pp. 129–138. DOI: 10.1109/INFOCOM41043.2020.9155363.

[143] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21, Québec city, Canada: Association for Computing Machinery, 2021, 248–259, ISBN: 9781450385343. DOI: 10.1145/3464298.3493398. [Online]. Available: https://doi.org/10.1145/3464298.3493398.

[144] R. K. Jain, D.-M. W. Chiu, W. R. Hawe, *et al.*, "A quantitative measure of fairness and discrimination," *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA*, vol. 21, 1984.

[145] C. Findling, V. Skvortsova, R. Dromnelle, S. Palminteri, and V. Wyart, "Computational noise in reward-guided learning drives behavioral vari-

ability in volatile environments," *Nature neuroscience*, vol. 22, no. 12, pp. 2066–2077, 2019.

[146] M. Shahrad, R. Fonseca, I. Goiri, *et al.*, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," *arXiv preprint arXiv:2003.03423*, 2020.

[147] F. Scarselli and A. C. Tsoi, "Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results," *Neural networks*, vol. 11, no. 1, pp. 15–37, 1998.

[148] A. Taherkordi, F. Eliassen, and G. Horn, "From iot big data to iot big services," in *Proceedings of the Symposium on Applied Computing*, 2017, pp. 485–491.

[149] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 ieee 14th international symposium on network computing and applications*, IEEE, 2015, pp. 27–34.

[150] M. Park, K. Bhardwaj, and A. Gavrilovska, "Toward lighter containers for the edge," in *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*, 2020.

[151] A. Zafeiropoulos, E. Fotopoulou, N. Filinis, and S. Papavassiliou, "Reinforcement learning-assisted autoscaling mechanisms for serverless computing platforms," *Simulation Modelling Practice and Theory*, vol. 116, p. 102 461, 2022.

[152] J. Li, S. G. Kulkarni, K. Ramakrishnan, and D. Li, "Understanding open source serverless platforms: Design considerations and performance," in *Proceedings of the 5th international workshop on serverless computing*, 2019, pp. 37–42.

[153] A. Das, S. Imai, S. Patterson, and M. P. Wittie, "Performance optimization for edge-cloud serverless platforms via dynamic task placement," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 41–50.

[154] C. K. Dehury, S. N. Srirama, and T. R. Chhetri, "CCoDaMiC: A framework for Coherent Coordination of Data Migration and Computation platforms," *Future Generation Computer Systems*, vol. 109, pp. 1–16, 2020, ISSN: 0167-739X.

[155] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 263–274.

[156] C. Dehury, P. Jakovits, S. N. Srirama, V. Tountopoulos, and G. Giotis, "Data pipeline architecture for serverless platform," in *European Conference on Software Architecture*, Springer, 2020, pp. 241–246.

[157] Google, *Google cloud iot*, 2020 (accessed December 13, 2020). [Online]. Available: `https://cloud.google.com/solutions/iot`.

[158] *Google data flow*, 2020 (accessed December 13, 2020). [Online]. Available: https://cloud.google.com/dataflow.

[159] *Aws glue*, 2020 (accessed December 13, 2020). [Online]. Available: https://aws.amazon.com/glue/.

[160] *Aws datapipeline*, 2020 (accessed December 13, 2020). [Online]. Available: https://aws.amazon.com/datapipeline/.

[161] *Aws greengrass*, 2020 (accessed December 13, 2020). [Online]. Available: https://aws.amazon.com/greengrass/.

[162] *Azure iot edge*, 2020 (accessed December 13, 2020). [Online]. Available: https://azure.microsoft.com/en-us/services/iot-edge/.

[163] M. Helu, T. Sprock, D. Hartenstine, R. Venketesh, and W. Sobel, "Scalable data pipeline architecture to support the industrial internet of things," *CIRP Annals*, 2020.

[164] P. O'Donovan, K. Leahy, K. Bruton, and D. T. O'Sullivan, "An industrial big data pipeline for data-driven analytics maintenance applications in large-scale smart manufacturing facilities," *Journal of Big Data*, vol. 2, no. 1, p. 25, 2015.

[165] J. Ronkainen and A. Iivari, "Designing a data management pipeline for pervasive sensor communication systems.," in *MobiSPC*, 2015, pp. 183–188.

[166] O. Akin, H. F. Deniz, D. Nefis, A. Kiziltan, and A. Cakir, "Enabling big data analytics at manufacturing fields of farplas automotive," *arXiv preprint arXiv:2004.11682*, 2020.

[167] M. Nardelli, V. Cardellini, V. Grassi, and F. L. Presti, "Efficient operator placement for distributed data stream processing applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 8, pp. 1753–1767, 2019. DOI: 10.1109/TPDS.2019.2896115.

[168] S. Nastic, T. Rausch, O. Scekic, *et al.*, "A serverless real-time data analytics platform for edge computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[169] R. Hetzel, T. Kärkkäinen, and J. Ott, "μActor: Stateful serverless at the edge," in *Proceedings of the 1st Workshop on Serverless mobile networking for 6G Communications*, 2021, pp. 1–6.

[170] R. Young, S. Fallon, and P. Jacob, "Dynamic collaboration of centralized & edge processing for coordinated data management in an iot paradigm," in *2018 IEEE 32nd international conference on advanced information networking and applications (AINA)*, IEEE, 2018, pp. 694–701.

[171] S. Sankaranarayanan, J. J. Rodrigues, V. Sugumaran, S. Kozlov, *et al.*, "Data flow and distributed deep neural network based low latency iot-edge computation model for big data environment," *Engineering Applications of Artificial Intelligence*, vol. 94, p. 103 785, 2020.

[172] G. Li, J. Lan, and Q. Li, "Online monitoring of self-elevating leveling ship based on edge computing," in *2020 IEEE 3rd International Conference on Electronics Technology (ICET)*, IEEE, 2020, pp. 546–551.

[173] M. Etemadi, M. Ghobaei-Arani, and A. Shahidinejad, "Resource provisioning for iot services in the fog computing environment: An autonomic approach," *Computer Communications*, vol. 161, pp. 109–131, 2020.

[174] S. Taherizadeh and V. Stankovski, "Auto-scaling applications in edge computing: Taxonomy and challenges," in *Proceedings of the International Conference on Big Data and Internet of Thing*, 2017, pp. 158–163.

[175] M.-N. Tran and Y. Kim, "Optimized resource usage with hybrid auto-scaling system for knative serverless edge computing," *Future Generation Computer Systems*, vol. 152, pp. 304–316, 2024.

[176] F. M. Awaysheh, "From the cloud to the edge towards a distributed and light weight secure big data pipelines for iot applications," in *Trust, Security and Privacy for Big Data*, CRC Press, 2022, pp. 50–68.

[177] F. M. Awaysheh, M. Alazab, S. Garg, D. Niyato, and C. Verikoukis, "Big data resource management & networks: Taxonomy, survey, and future directions," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2098–2130, 2021.

[178] H. Nashaat, N. Ashry, and R. Rizk, "Smart elastic scheduling algorithm for virtual machine migration in cloud computing," *The Journal of Supercomputing*, vol. 75, no. 7, pp. 3842–3865, 2019.

[179] A. Samanta, Y. Li, and F. Esposito, "Battle of microservices: Towards latency-optimal heuristic scheduling for edge computing," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, IEEE, 2019, pp. 223–227.

[180] B. M. Nguyen, H. Thi Thanh Binh, B. Do Son, *et al.*, "Evolutionary algorithms to optimize task scheduling problem for the iot based bag-of-tasks application in cloud–fog computing environment," *Applied Sciences*, vol. 9, no. 9, p. 1730, 2019.

[181] S. Tuli, S. S. Gill, G. Casale, and N. R. Jennings, "iThermoFog: IoT-Fog based automatic thermal profile creation for cloud data centers using artificial intelligence techniques," *Internet Technology Letters*, vol. 3, no. 5, e198, 2020.

[182] L. Bogolubsky, P. Dvurechenskii, A. Gasnikov, *et al.*, "Learning supervised pagerank with gradient-based and gradient-free optimization methods," in *Advances in neural information processing systems*, 2016.

[183] L. M. Rios and N. V. Sahinidis, "Derivative-free optimization: A review of algorithms and comparison of software implementations," *Journal of Global Optimization*, vol. 56, no. 3, pp. 1247–1293, 2013.

[184] P. van de Ven, S. Borst, and S. Shneer, "Instability of maxweight scheduling algorithms," in *IEEE INFOCOM 2009*, IEEE, 2009, pp. 1701–1709.

[185] van de Ven and et el., "Inefficiency of maxweight scheduling in spatial wireless networks," *Computer Communications*, vol. 36, no. 12, pp. 1350–1359, 2013.

[186] I. AlQerm and J. Pan, "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*, vol. 18, no. 4, pp. 3942–3954, 2021.

[187] A. Palade, A. Kazmi, and S. Clarke, "An evaluation of open source serverless computing frameworks support at the edge," in *2019 IEEE World Congress on Services (SERVICES)*, IEEE, vol. 2642, 2019, pp. 206–211.

[188] Z. Zhou, C. Zhang, L. Ma, *et al.*, "Ahpa: Adaptive horizontal pod autoscaling systems on alibaba cloud container service for kubernetes," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, 2023, pp. 15 621–15 629.

[189] Q. L. Trieu, B. Javadi, J. Basilakis, and A. N. Toosi, "Performance evaluation of serverless edge computing for machine learning applications," in *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*, IEEE, 2022, pp. 139–144.

[190] A. Arjona, P. G. López, J. Sampé, A. Slominski, and L. Villard, "Triggerflow: Trigger-based orchestration of serverless workflows," *Future Generation Computer Systems*, vol. 124, pp. 215–229, 2021.

ACKNOWLEDGEMENTS

First and foremost, I am immensely grateful and hold deep respect for both of my supervisors, Prof. Satish Narayana Srirama and Dr.Pelle Jakovits. They have consistently supported me throughout my journey, attentively listened to my queries, identified the intellectual potential within me, and provided unwavering support. Ultimately, without them, this journey would not have been possible.

I would like to express my gratitude to my family—my wife Geetha and my son Rishaank—for their unwavering support, enabling me to pursue my PhD work.

I extend my heartfelt thanks to my mother, brother, and sisters for their unconditional love and steadfast support throughout this journey. Additionally, I acknowledge the kindness of Shree J. Basappa and his family for their love and support in hosting my family during my absence.

I express my gratitude to my mentor, Prof. Rajkumar Buyya from the University of Melbourne, for his invaluable guidance and support throughout this journey.

I extend my thanks to my friends and colleagues at the Institute of Computer Science who played a crucial role in my PhD journey. Thanks to Dr. Chinmaya Kumar Dehury and Dr. Jakob Mass for their valuable suggestions and comments.

Finally, I acknowledge the funding support received from the European Social Fund through the IT Academy program. Additionally, my thanks go to Telia Estonia for their hardware support and stipendium. Special thanks to Prof. Pelle Jakovits and the University of Tartu for believing in me and extending my funding beyond the nominal period.

SISUKOKKUVÕTE

Sündmustepõhiste, skaleeruvate ja serverivabade andmetorude disain ja orkestreerimine asjade Interneti rakenduste jaoks

Asjade Interneti (IoT) seadmete üha suureneva kasutamisega on toimunud tohutu toorandmete kasv. Selliste andmete haldamine hõlmab aga keerulisi ülesandeid, sealhulgas andmete hankimist erinevatest seadmetest erinevates vormingutes, filtreerimine ja teisendamine, ning masinõppe rakendamine. Selliste andmevoogude voo ja elutsükli tõhus haldamine on märkimisväärne väljakutse. Selleks, et saavutada madal latentsus ja muud teenusekvaliteedi (QoS) mõõdikud, võetakse üha enam kasutusele pilvepõhise IoT andmetöötluse asemel serva ja udu arvutusmudeleid. See muudab andmeanalüüsi ülesannete dünaamilise täitmise keerukamaks erinevatel kaugustel ja heterogeenses riistvaras.

Üks lähenemisviis Asjade Interneti andmetöötluse realiseerimiseks on monoliitsete konteinerrakenduste kasutamine, mis kondavad andmetoimingud ühte konteinerisse. Selliseid konteinereid saab migreerida üle IoT kihtide (serv, udu, pilv), et optimeerida teenuse kvaliteedi (QoS) mõõdikuid. Monoliitsete konteinerite kasutamine võib tõhusat andmehaldust nõudvate andmepõhiste Asjade Interneti rakenduste väljatöötamisel tekitada väljakutseid ja keerukust. Sujuva ühenduvuse tagamisel ja andmetoimingute skaleerimisel võib tekkida ka muid probleeme. Teised olemasolevad lahendused, nagu suured andmetöötlusklastrid (nt Apache Flink või Spark) ja valmistööriistad, võivad ressursipiirangute (serva- ja uduseadmed) ja asjade Interneti-rakenduste sündmustepõhise olemuse tõttu olla ebausaldusväärsed.

Hüpoteesiks on, et seda saab lihtsustada serverivabade arvutuste ja andmekonveierite kasutusele võtuga. Serverivabade arvutuste kasutamisel saab andmeanalüütilisi ülesandeid luua individuaalselt skaleeritavate virtuaalsete funktsioonidena ja neid sündmustepõhiselt täita. Andmekonveierid võimaldavad koondada üksikud andmetöötlusülesanded suureks hajutatud andmevooks. Mõlema mudeli kombineerimisel saab luua serverivabad andmekonveierid (SDP), kus serverivabu funktsioone kasutatakse konveieriülesannetena ja neid saab sujuvalt välja kutsuda, kui andmed konveieri kaudu liiguvad. Servervabu funktsioone saab lihtsasti käivitada pilve-, serva- või udukeskkondades ning andmeedastuseks, marsruutimiseks ja funktsioonide kutsumiseks kasutatakse andmekonveieri tehnoloogiaid.

Selle väitekiri eesmärk on adresseerida andmetöötluse kriitilisi aspekte asjade Interneti (IoT) keskkondades, keskendudes üleminekule konteineritelt serverivabale arhitektuuridele. Esmalt analüüsitakse kitsaskohti traditsioonilistes monoliitsetes konteineripõhistes IoT andmetöötluse lähenemisviisides. Seejärel uuritakse serverivaba andmetöötluse rakendamist asjade Interneti keskkondades kui potentsiaalset lahendust monoliitsete arhitektuuridega seotud väljakutsete ületamiseks. Lõpuks analüüsitakse serverivabade andmetöötlusraamistike skaleeritavust asjade

Interneti stohhastiliste töökoormuste haldamisel.

Sellel väitekirjal on kolm panust. Esimene on uudne simulaator ja raamistik konteinerite orkestreerimiseks IoT keskkondades koos gradiendipõhise tagasilevitamise lähenemisviisiga (GOBI ja GOBI*) ajastamiseks, mis on effektiivsem olemasolevatest planeerijatest. Teine panus hõlmab kolme disaini lähenemist serverivabade andmekonveierite (SDP) loomiseks ja nende sobivuse analüüsi erinevate asjade Interneti rakenduste jaoks. Standardsetel andmevootööriistadel (DFT) põhinevad SDP-d ei sobi arvutusmahukate ülesannete jaoks, nagu videotöötlus, kuid need on tõhusad laia ribalaiust vajavate rakenduste jaoks. Objektisalvestusteenusel (OSS) põhinevad SDP-d sobivad paremini arvutusmahukate toimingute jaoks ja MQTT-põhised SDP-d sobivad latentsustundlike toimingute jaoks, kuid mitte arvutus- ja ribalaiustundlike ülesannete jaoks, kuna protsessori ja mälu kasutus on suurem. Kolmas panus on reaktiivsete automaatse skaleerimise mehhanismide sobivuse analüüs SDP jaoks nelja erineva töökoormuse mustri korral. Arvutusmahukate ülesannete puhul töötab ressursipõhine skaleerimise lähenemisviis tõhusalt hüppelise, püsiva, järsu ja kõikuvate töökoormuste korral. Lühikese täitmisajaga ülesannete jaoks sobib töökoormusepõhine skaleerimine kõigi nelja töökoormuse korral.

See väitekiri käsitleb IoT andmete töötlemise keerukust ja väljakutseid üleminekul monoliitsetelt konteineriarhitektuuridelt serverivabadele pilvearvutusmudelitele asjade Interneti andmete töötlemisel. Töö väljundid aidatvad asjade Interneti arendajatel valida kõige sobivaamad andmetöötlusmehhanismid, võttes arvesse selliseid tegureid nagu vabad arvutusressursid, ribalaius, energiatarbimine ja latentsus, täites samal ajal tundlikke QoS nõudeid.

CURRICULUM VITAE

Personal data

Name:	Shivananda Rangappa Poojara
Date of Birth:	01.01.1987
Nationality:	Indian
Language skills:	Hindi, English and Kannada
Email:	shivu.poojar@gmail.com

Education

2019–2024	University of Tartu, Tartu, Estonia, Doctoral studies Specialization: Computer Science
2010–2012	University of Vishveshwarya College of Engineering, Karnataka, India, Master of Engineering Specialization: Software Engineering
2005–2009	Vishveshwarya Technological University, Karnataka,India, Bachelor of Engineering. Specialization: Computer Science and Engineering

Employment

2019–2023	Junior Research Fellow, University of Tartu, Estonia
2013–2019	Assistant Professor, Rajarambapu Institute of Technology, Maharashtra, India
2012– 2013	Salesforce Developer, Uohmac Technologies, Bangalore, India
2011–2012	Intern, Nokia India Private Ltd, Bangalore, India

Scientific work

Main fields of interest:

- IoT
- Cloud Computing
- Edge analytics

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Shivananda Rangappa Poojara
Sünniaeg:	01.01.1987
Kodakondsus:	India
Keeleoskus:	hindi, inglise ja kannada
E-post:	shivu.poojar@gmail.com

Haridus

2019–2024	Tartu Ülikool, Tartu, Eesti, Doktoriõpe Spetsialiseerumine: arvutiteadus
2010–2012	University of Vishveshwarya insenerikolledž, Karnataka, India, Magister Spetsialiseerumine: tarkvaratehnika
2005–2009	Vishveshwarya Tehnoloogiaülikool, Karnataka, India, Inseneri bakalaureus. Spetsialiseerumine: arvutiteadus ja tehnika

Teenistuskäik

2019–2024	nooremteadur, Tartu Ülikool, Eesti
2013–2019	abiprofessor, Rajarambapu tehnoloogiainstituudi, Maharashtra, India
2012–2013	Salesforce'i arendaja, Uohmac Technologies, Bangalore, India
2011–2012	praktikant, Nokia India Private Ltd, Bangalore, India

Teadustegevus

Peamised uurimisvaldkonnad:

- IoT
- Pilvandmetöötlus
- Edge analüütika

DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.
8. **Toomas Krips.** Improving performance of secure real-number operations. Tartu 2019, 146 p.
9. **Vijayachitra Modhukur.** Profiling of DNA methylation patterns as biomarkers of human disease. Tartu 2019, 134 p.
10. **Elena Sügis.** Integration Methods for Heterogeneous Biological Data. Tartu 2019, 250 p.
11. **Tõnis Tasa.** Bioinformatics Approaches in Personalised Pharmacotherapy. Tartu 2019, 150 p.
12. **Sulev Reisberg.** Developing Computational Solutions for Personalized Medicine. Tartu 2019, 126 p.
13. **Huishi Yin.** Using a Kano-like Model to Facilitate Open Innovation in Requirements Engineering. Tartu 2019, 129 p.
14. **Faiz Ali Shah.** Extracting Information from App Reviews to Facilitate Software Development Activities. Tartu 2020, 149 p.
15. **Adriano Augusto**. Accurate and Efficient Discovery of Process Models from Event Logs. Tartu 2020, 194 p.
16. **Karim Baghery.** Reducing Trust and Improving Security in zk-SNARKs and Commitments. Tartu 2020, 245 p.
17. **Behzad Abdolmaleki.** On Succinct Non-Interactive Zero-Knowledge Protocols Under Weaker Trust Assumptions. Tartu 2020, 209 p.
18. **Janno Siim.** Non-Interactive Shuffle Arguments. Tartu 2020, 154 p.
19. **Ilya Kuzovkin.** Understanding Information Processing in Human Brain by Interpreting Machine Learning Models. Tartu 2020, 149 p.
20. **Orlenys López Pintado.** Collaborative Business Process Execution on the Blockchain: The Caterpillar System. Tartu 2020, 170 p.
21. **Ardi Tampuu.** Neural Networks for Analyzing Biological Data. Tartu 2020, 152 p.

22. **Madis Vasser.** Testing a Computational Theory of Brain Functioning with Virtual Reality. Tartu 2020, 106 p.
23. **Ljubov Jaanuska.** Haar Wavelet Method for Vibration Analysis of Beams and Parameter Quantification. Tartu 2021, 192 p.
24. **Arnis Parsovs.** Estonian Electronic Identity Card and its Security Challenges. Tartu 2021, 214 p.
25. **Kaido Lepik.** Inferring causality between transcriptome and complex traits. Tartu 2021, 224 p.
26. **Tauno Palts.** A Model for Assessing Computational Thinking Skills. Tartu 2021, 134 p.
27. **Liis Kolberg.** Developing and applying bioinformatics tools for gene expression data interpretation. Tartu 2021, 195 p.
28. **Dmytro Fishman.** Developing a data analysis pipeline for automated protein profiling in immunology. Tartu 2021, 155 p.
29. **Ivo Kubjas.** Algebraic Approaches to Problems Arising in Decentralized Systems. Tartu 2021, 120 p.
30. **Hina Anwar.** Towards Greener Software Engineering Using Software Analytics. Tartu 2021, 186 p.
31. **Veronika Plotnikova.** FIN-DM: A Data Mining Process for the Financial Services. Tartu 2021, 197 p.
32. **Manuel Camargo.** Automated Discovery of Business Process Simulation Models From Event Logs: A Hybrid Process Mining and Deep Learning Approach. Tartu 2021, 130 p.
33. **Volodymyr Leno.** Robotic Process Mining: Accelerating the Adoption of Robotic Process Automation. Tartu 2021, 119 p.
34. **Kristjan Krips.** Privacy and Coercion-Resistance in Voting. Tartu 2022, 173 p.
35. **Elizaveta Yankovskaya.** Quality Estimation through Attention. Tartu 2022, 115 p.
36. **Mubashar Iqbal.** Reference Framework for Managing Security Risks Using Blockchain. Tartu 2022, 203 p.
37. **Jakob Mass.** Process Management for Internet of Mobile Things. Tartu 2022, 151 p.
38. **Gamal Elkoumy.** Privacy-Enhancing Technologies for Business Process Mining. Tartu 2022, 135 p.
39. **Lidia Feklistova.** Learners of an Introductory Programming MOOC: Background Variables, Engagement Patterns and Performance. Tartu 2022, 151 p.
40. **Mohamed Ragab.** Bench-Ranking: A Prescriptive Analysis Approach for Large Knowledge Graphs Query Workloads. Tartu 2022, 158 p.
41. **Mohammad Anagreh.** Privacy-Preserving Parallel Computations for Graph Problems. Tartu 2023, 181 p.
42. **Rahul Goel.** Mining Social Well-being Using Mobile Data. Tartu 2023, 104 p.

43. **Anti Ingel.** Algorithms using information theory: classification in brain-computer interfaces and characterising reinforcement-learning agents. Tartu 2023, 142 p.
44. **Shakshi Sharma.** Fighting Misinformation in the Digital Age: A Comprehensive Strategy for Characterizing, Identifying, and Mitigating Misinformation on Online Social Media Platforms. Tartu 2023, 158 p.
45. **Kristiina Rahkema.** Quality Analysis of iOS Applications with Focus on Maintainability and Security Aspects. Tartu 2023, 182 p.
46. **Ivan Slobozhan.** Studying Online Social Media Engagement in CIS Countries during Protests, Mass Demonstrations and War. Tartu 2023, 81 p.
47. **Nurlan Kerimov.** Building a catalogue of molecular quantitative trait loci to interpret complex trait associations. Tartu 2023, 248 p.
48. **Pavlo Tertychnyi.** Machine Learning Methods for Anti-Money Laundering Monitoring. Tartu 2023, 117 p.
49. **Abasi-amefon Obot Affia.** A Framework and Teaching Approach for IoT Security Risk Management. Tartu 2023, 180 p.
50. **Raimond-Hendrik Tunnel.** Video Game Design and Development Bachelor's Curriculum for Estonia. Tartu 2024, 137 p.
51. **Ahto Salumets.** Bioinformatics analysis of various aspects in immunology. Tartu 2024, 198 p.
52. **Mohammed Abdulhameed Shaif Ali.** Deep Learning Methods for Cell Microscopy Image Analysis. Tartu 2024, 143 p.
53. **Pille Pullonen-Raudvere**. Foundations of Efficient and Secure Algorithm Development for Secure Multiparty Computation. Tartu 2024, 265 p.
54. **Marili Rõõm.** Multiple approaches to learners' success and factors affecting it in computer programming MOOCs. Tartu 2024, 170 p.