A DECENTRALIZED FRAMEWORK WITH DYNAMIC AND EVENT-DRIVEN CONTAINER ORCHESTRATION AT THE EDGE

by

Umut Can Özyar

B.S., Computer Engineering, Istanbul Technical University, 2017

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2022

ACKNOWLEDGEMENTS

I would like to express my most profound appreciation to my advisor, Prof. Dr. Arda Yurdakul, for her guidance throughout my thesis development. I will forever be grateful for her continuous support, patience, and immense knowledge as a mentor and a role model. I also appreciate the valuable life lessons she has taught me along the way.

I would like to extend my deepest gratitude to my thesis defense jury members not only for their time and patience but also for their insightful advice and invaluable feedback.

Special thanks to my friends at our IoT-BC research group for providing a circle of science flourishing with constructive discussions, especially during the early years of my research.

Finally, I am indebted to my friends and family for the everlasting support and understanding they have shown throughout these stressful times.

ABSTRACT

A DECENTRALIZED FRAMEWORK WITH DYNAMIC AND EVENT-DRIVEN CONTAINER ORCHESTRATION AT THE EDGE

Persistent advancements are being made at a rapid pace on enabling edge computing for the Internet of Things technology to capitalize on. Vendors and developers are exploring new techniques to smooth out the process of navigating through the inherent heterogeneity of the edge networks. However, application delivery, resource allocation, fault tolerance, and security issues are yet to be fully solved while also providing a seamless experience for consumers. With virtualization and lightweight container management platforms providing an abstraction layer, it is possible to deploy the same application on devices with different architectures and achieve uniformity.

Towards a fully decentralized edge, the framework proposed in this thesis lays down the groundworks for dynamic container orchestration. It provides a blockchain-based delivery platform for container applications with their updates and resource specifications through a registry on a distributed file system, namely InterPlanetary File System (IPFS). Then, enabled by the operating system virtualization, the framework handles resource allocation, container availability and scaling. A self-adaptive resource manager running on the metrics scraped from the host and the virtualization platform, i.e. Docker in our implementation, dynamically optimizes the resources allocated to each container. The framework ensures that variable workloads of a heterogeneous environment can co-exist on an edge device that is designed to be further extended to multiple devices. To achieve a truly distributed system, an event-driven architecture is built over a lightweight messaging protocol, MQTT, capitalizing on the asynchronous and distributed nature of the publish/subscribe pattern.

ÖZET

KENARDA DİNAMİK VE OLAY GÜDÜMLÜ KONTEYNIR ORKESTRASYONLU DAĞITIK BİR ÇERÇEVE

Kenar hesaplamada Nesnelerin İnterneti teknolojisinin önünü açacak ilerlemeler kaydedilmektedir. Firmalar ve geliştiriciler, kenar ağların heterojen yapısında çalışmayı kolaylaştırmak için yeni teknikler araştırmaktadırlar. Uygulama teslimi, kaynak tahsisi, hata toleransı ve güvenlik sorunları henüz kenarda tüketicilere pürüzsüz bir deneyim sunacak şekilde çözülmüş değildir. Soyutlama ve hafif konteynır yönetim platformları ile sağlanan sanallaştırmayla aynı uygulamayı farklı mimarilere sahip cihazlara dağıtmak ve tekdüzelik sağlamak mümkün olmuştur.

Bu tezde önerilen çerçeve, tamamıyla merkezî olmayan bir kenara doğru, dinamik konteynır orkestrasyonu için temelleri ortaya koymaktadır. Dağıtık bir dosya sistemi olan Gezegenler Arası Dosya Sistemi (IPFS) üzerindeki bir kayıt sistemi aracılığıyla konteynır uygulamaları, güncellemeleri ve kaynak özellikleri için blokzincir tabanlı bir dağıtım platformu sağlamaktadır. Çerçeve kaynak tahsisini, ölçeklendirmeyi ve konteynır kullanılabilirliğini işletim sistemi sanallaştırmasıyla yönetmektedir. Ana bilgisayardan ve Docker sanallaştırma platformundan alınan metrikler üzerinde çalışan, kendi kendini uyarlayan bir kaynak yöneticisi konteynırlara tahsis edilen kaynakları dinamik olarak optimize etmektedir. Birden fazla cihaza genişletilmeyi desteklemek üzere tasarlanmış çerçeve, heterojen bir ortamın değişken iş yüklerinin aynı kenar cihazda bir arada var olmasını sağlar. Hafif bir mesajlaşma protokolü olan MQTT üzerinde yayınla/abone ol modelinin zaman uyumsuz ve dağıtılmış yapısından yararlanan olay güdümlü bir mimari oluşturularak tamamıyla dağıtık bir sistem elde edilebilmiştir.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

$C_i^{throttle}$	CPU throttling percentage of container C_i
$C_{i,j}^{util}$	Utilization of resource R_i by container C_j
C_i	Container i
I_i	Image i
$L_{i,j}^{base}$	Base limit definition of resource R_i for image I_i
L_i^{buffer}	Buffer ratio for scaling of resource R_i
$L_{i,j}^{current}$	Current limit definition of resource R_i for container C_j
$L_i^{default}$	Default limit definition of resource R_i for all images
$L_{i,j}^{request}$	Request limit definition of resource R_i for image I_i
$L_{i,j}^{scale}$	Limit scaling amount of R_i for $j \in \{up, down\}$
$L_{i,j}^{target}$	Target limit definition of resource R_i for container C_j
$L_{i,j}^{threshold}$	Threshold limit of resource R_i for $j \in \{min, max\}$
$L^{throttle}$	Limit CPU throttling percentage
$L_{i,S}^{total}$	Total limit definition of resource R_i of the system S
$P_{i,S}^{avail}$	Predicted availability of resource R_i of the system S
$P_{i,S}^{max}$	Max predicted utilization of resource R_i by the system S
$P_i^{throttle}$	Predicted CPU throttling percentage of container C_i
$P_{i,j}^{util}[t]$	Predicted utilization of resource R_i by container C_j at time t
$P_{i,S}^{util}[t]$	Predicted utilization of resource R_i by the system at time t
R_i	Resource $i \in \{cpu, mem\}$
S	System
S_i^{avail}	Availability of resource R_i of the system S
S_i^{total}	Total resources R_i of the system S
S_i^{util}	Utilization of resource R_i of the system S

LIST OF ACRONYMS/ABBREVIATIONS

API	Application Programming Interface
ARIMA	Autoregressive Integrated Moving Average
ARIMA-BP	ARIMA with Back Propagation
CPU	Central Processing Unit
CSV	Comma-Separated Values
CoAP	Constrained Application Protocol
CRIU	Checkpoint-Restore in Userspace
DAG	Directed Acyclic Graph
DDS	Data Distribution Service
DL	Deep Learning
DRF	Dominant Resource Fairness
IEC	International Electrotechnical Commission
IP	Internet Protocol
IPDR	IPFS-Backed Docker Registry
IPFS	InterPlanetary File System
IPNS	InterPlanetary Name System
JSON	JavaScript Object Notation
MAPE-K	Monitor-Analyze-Plan-Execute-Knowledge
MARL	Multi-Agent Reinforcement Learning
MEC	Mobile Edge Computing
ML	Machine Learning
MQTT	MQ Telemetry Transport
OBS	Open Broadcaster Software
OS	Operating System
Pub/Sub	Publish-Subscribe Pattern
REST	Representational State Transfer
RTMP	Real-Time Messaging Protocol
RL	Reinforcement Learning

SARSA	State-Action-Reward-State-Action
SLA	Service-Level Agreement
SLO	Service-Level Objective

1. INTRODUCTION

The term "Internet of Things (IoT)" describes a sophisticated system of heterogeneous devices, dynamic environments, and complex sub-systems. This heterogeneous characteristic of the IoT domain is the root cause of its complexity. Limited resources, the volume of data, and variable conditions all contribute to this complexity. A system without any governance with this many variables cannot sustain itself. Manual intervention could be a remedy in the short term. Still, in a complex system, a smart mechanism should be set in place to establish long-lasting stability.

The heterogeneous nature of IoT also opens up the door for a multitude of opportunities. Existing architectures rely upon a joint solution spanning multiple fields of computer systems. However, the primary defining factor is always data. From its generation to the unit where it ends up for processing, its volume, velocity, and variety shape the requirements. Everything else is built around it to maintain this flow to extract meaning from raw data. Changing requirements across different steps of this flow often means that it should pass through numerous devices and networks. To simplify this flow and gather dependent steps together, these architectures are often split into layers. These layers are ordered according to the data flow from the source where it is generated to its destination. These destinations are usually processing units, which make inferences from data, storage units, or simply repeaters that relay the data to the upper layer. Although distinct in nature, layers may share some capabilities. For example, an intelligent IoT end node can fulfill some of the tasks that would typically be assigned to the devices at the upper layers. What determines the boundaries of these layers can be narrowed down to the capabilities of the devices being used and the nature of the surrounding environment.

The steady flow of data is established by a series of components placed across the network navigating data to its destination. These components can adopt different technologies, and their positioning depends on the objective, environment, and require-

ments. This is why there is not just one solution that can be applied to every problem. However, these solutions share some characteristics and harbor some recurring patterns. As problems continue to present themselves, past knowledge can be used as building blocks to build novel solutions for ever-changing requirements.

As the knowledge of IoT technology broadens, the requirements of an IoT network also evolve. Networks utilize emerging technologies whenever possible as the demands increase. Cloud systems have become especially prominent in the last decade as they could solve multiple inherent problems with relative ease. With their high storage capacity, tremendous processing power, and ease of access, they have become the norm for data processing in IoT quickly. Since then, they are being utilized by most modern architectures. Yet, they still have their flaws. The abundance of data brings forth concerns about storage costs. With latency constraints in some domains like autonomous vehicles, it is no longer feasible to use the cloud so loosely. Hence, the next generation of IoT architectures have focused on these limitations and started employing newer technologies such as edge computing. As a paradigm that challenges the necessity of processing and storing data on the cloud, edge computing tries to find solutions on devices much closer to the data source.

Edge computing also comes with its own set of challenges because of its heterogeneous and resource-constrained nature. Yet, edge solutions can benefit from the knowledge accumulated by tackling similar challenges on the cloud leading to the recent advancements in software abstraction provided by virtualization and the availability of lightweight and distributed alternatives of technologies used on the cloud.

Recently, capitalizing on operating system (OS) level virtualization, applications packaged with their own dependencies called containers have proven themselves to be suitable for the edge. Container applications provide means of sharing host resources, virtualization of services, and modern delivery options. However, the deployment of applications on the devices at the edge has to be handled very carefully since on-device resources are scarce and diverse. An orchestrator's presence is necessary to solve issues

with fault tolerance, monitoring, container availability, resource allocation, and scaling. Yet, a fair system design becomes essential to have a solution for the applications of different vendors and developers to co-exist on the same hardware. This can be achieved by decentralization. Utilizing technologies like blockchain-assisted backends, distributed file storages, and publish/subscribe patterns for asynchronous and event-driven communication across the network can aid in building a decentralized solution. Hence, within the scope of this thesis, we will build a decentralized framework with a self-adaptive resource manager for container-based applications on the gateway devices. The main contributions of this thesis can be summarized as follows:

- A decentralized application delivery platform for container applications with their updates and resource specifications is implemented with a smart contract, a computer program stored and run on the Ethereum blockchain to ensure their authenticity. The platform depends on a decentralized registry for the storage and distribution of applications backed with an OS-level virtualization platform called Docker and a distributed file system, InterPlanetary File System(IPFS).
- A decentralized framework is designed for orchestrating multiple container applications at a resource-constrained edge gateway. time series forecasting and rule-based temporal analysis capabilities are built into the framework for self-adaptive resource management. A set of message schemas are defined to establish asynchronous communication between distributed components of the framework using a publish/subscribe network protocol, namely MQTT. A second smart contract is written for framework's internal long term data management to archive, backup, and restore the collected system metrics.

Experiments are carried on a resource-constrained edge gateway by scheduling applications with various workload patterns each representing different types of IoT tasks. The device used in the experiments is a Raspberry Pi with its resources constrained by configuring OS-level resource limits. The framework is put through different use cases with ideal or misconfigured resource specifications and various scheduling sequences and constraints. The framework is able to dynamically allocate resources, ensure con-

tainer availability and vertically scale active containers in each scenario to provide a platform for applications to co-exist by sharing the resources of an edge gateway. A case study is also made on the load balancing capabilities of the framework when it is deployed as a cluster on multiple edge devices in the same network.

The remainder of the thesis is organized as follows. In Section 2, previous work on virtualization, resource allocation, and orchestration for IoT systems is discussed. Section 3 introduces the proposed framework and explains the technology and design choices for data storage, application delivery, container orchestration, and decentralization problems. Section 4 analyzes the behavior of the framework under different experimental setups. The final section presents the conclusion and future work.

2. RELATED WORK

An IoT system is molded by the data circulating through its components. The characteristics of data determine the requirements of an end-to-end solution. Zubair et al. describe the life cycle of IoT data in five steps: Creation, curation, transformation, collection, deletion [1]. An architecture proposal missing support for any of these steps will be incapable of covering the problem as a whole. Data has inherent and domain-specific characteristics. These distinctive properties should be examined and incorporated into the system design. Inherent characteristics include the volume, velocity, and heterogeneity of data, as well as collection and storage. Domain-specific characteristics are more complex. For example, transactional data influences the system directly by introducing variation over time. Especially mobility, combined with a dynamic environment, demands solutions that can withstand the changing conditions. For the completeness of data, correct labeling has utmost importance.

2.1. IoT System Architecture

The properties above have given birth to unique architectures over the years. Cloud technologies make storage and processing cheaper. Also, end devices getting more powerful make diverse designs possible. As modeled by Sarkar and Misra, fog computing starts processing right after IoT data is transmitted to the system by the end devices [2]. This ensures that quick action can be taken when there are latency constraints. In fog computing, data is processed and filtered before being sent to the cloud. In addition to its clear performance in service latency, total energy consumption can be improved by offloading some content delivery tasks typically handled by the cloud [3]. One further step is to move the computing to the IoT data generating device and include it in the data processing. Another popular option is forwarding data all the way to the cloud, where storage and computing power are abundant. System designs proposed in [4, 5] aim to utilize the full potential of cloud computing. This behavior, although proven effective, leads to the creation of data silos [6]. As the size of data has

grown, financial concerns with ingesting data have turned out to be more apparent. Furthermore, there are contracts between service providers and clients called Service Level Agreements (SLA) which define a commitment on the quality and performance of the provided services. In some use cases like autonomous vehicles, cloud solutions are unable to honor the latency SLAs [7]. In such situations, services have to be located closer to the client to avoid problems in performance.

2.2. Virtualization

Virtualization is a concept for creating an abstract layer over the computing resources of the host by packaging a separate operating system with its own hardware functionalities. Virtual machines, container technologies like LXD and Docker are examples of operating system level virtualization [8]. This paradigm makes it possible to run software packaged with all necessary dependencies and configurations in complete isolation from each other. Recently, container-based architectures at the edge layer have gained popularity. Muralidharan et al. utilize a centralized but proven container management platform called Kubernetes [4]. Each container is directly connected to a central node which is responsible for orchestration and scaling. Rusek et al., on the other hand, try to decentralize these commands by capitalizing on the strengths of swarm algorithms inspired from pheromone robots [5].

Following the success of virtualization, architectural patterns such as microservices with loosely coupled and independent applications have rapidly expanded to the IoT [9]. As the system becomes more distributed, [10] relies on message queues to handle deployments on a heterogeneous system of independent applications. The reason behind this quick adoption is its resilience and robustness in large scale systems as well as the benefits it provides in deployment and development. These loosely coupled applications can be brought closer to IoT devices and applied at the edge layer. However, performance and overhead issues are more apparent when these services are deployed at the edge as Buzachis et al. analyze several network overlays to find the best fit for this environment [11]. The resilience factor opens up opportunities for fault tolerance

at the edge. Celeti et al. use passive and active containers simultaneously. Observing the system with a watchdog service, they determine when an active container goes down and activate the passive replica of the active container to achieve close to zero downtime [12].

Containers are built according to a set of configurations and commands on a file defined as *an image*. These image files can also be used for upgrading applications systematically. They can keep the system updated by efficiently adding new capabilities. Dolui and Kiraly suggest using a layered approach to reduce download sizes by using the old image as the base image and only downloading the upgrade [13]. Introducing a new update or upgrade to the system and distributing it is a security challenge in itself. Lee and Lee propose using blockchains to manage and distribute firmware updates securely. In their proposal, all validation and download occur between peers in an IoT environment [14]. Westerlund et al. take advantage of the domain name service of IPFS called *InterPlanetary Name System* (IPNS) to manage versioning by pointing an existing domain name to the new version's hash via an Ethereum smart contract to make secure delivery of updates possible [15]. Another possible approach is making use of the blockchain and letting nodes handle the updates between themselves [14]. Both systems can ensure that only the developers themselves can publish updates, and the update is tamper-proof.

Extending the decentralization and tamper-proof concepts back onto the data plane, there are several solutions that can prove to be helpful in handling IoT data [14, 16, 17]. IPFS is one such technology [18] which is a peer-to-peer distributed file system. It defines a decentralized file storage, transfer, and communication network. This network also provides a framework for decentralized applications as another option to the aforementioned smart contracts [19], and makes it possible to check the integrity of data with a checksum control [18]. An additional capability of IPFS is its support for publish/subscribe patterns by flooding messages [20]. However, protocols like CoAP, MQTT, and DDS offer features more befitting for IoT tasks [21, 22]. These protocols constitute an essential part of data transmission in various IoT systems [23, 24].

2.3. Orchestration

IoT systems, now thriving with virtualization and a multitude of tools for data management, face the problem of allocating the host system's resources to applications correctly [25]. Although it has become a challenge for all layers, the allocation of resources at the edge becomes a tougher challenge due to their scarcity at the edge [26–29]. The scarcity caused by limited resources leaves little room for errors. For sustainable environments at the edge layer, building smart mechanisms is a requirement. A layer-independent and proven algorithm for the distribution of resources is the Dominant Resource Fairness (DRF) proposed by Ghodsi et al. [30]. The resource allocation using DRF takes memory, CPU, and bandwidth requirements into consideration and tries to share available resources according to diverse resource requests while maximizing fairness. In the literature, there are resource allocation algorithms specifically focused on the edge layer as well. Liu et al. tackle the same challenge, specifically in IoT where communication is the main constraint [27]. They formulate a utility maximization problem and propose a greedy solution. Such methods favor cost-efficiency based on the utility of the task. Also, they rely on pair-to-pair communication between devices since their use cases are all decentralized IoT networks with no central controller [16]. Yigitoglu et al., on the other hand, relies on predefined metrics like priority, computation, and latency for orchestration [31].

The aforementioned container orchestration tool, Kubernetes, as well as other orchestration tools like *docker compose* and *docker swarm*, also depend on predefined requests and limit resources for each application for resource management. In [32], several container orchestration tools are compared based on the functionalities they offer, and the popularity of Kubernetes is emphasized. However, these tools fall short of satisfying the requirements of an edge system where the environment is out of reach of the developer maintaining the application. Therefore, a self-adaptive orchestrator has been built in [33] for better utilization of the resources at hand and maintaining the deployments. The framework is able to learn and make decisions on its own. These decisions can be based on the metrics collected from the system and the application

such as latency, CPU utilization and response times [34]. A survey [35] on different machine-learning-based orchestration methods presents a taxonomy of all the ML methods available in the literature. The four main categories are regression, classification, reinforcement learning, and time series analysis.

2.3.1. Regression

Logistic regression is used in [36] to make predictions on the expected loads of individual nodes. Resources are then allocated to applications across these nodes according to a multicriteria algorithm based on these predictions. Ajila and Bankole tries to predict the future resource demand by using Support Vector Machine (SVM) and Linear Regression (LR) [37].

2.3.2. Classification

The premise of using classification is that different applications can have similarities in their workload characteristics. In [38], K-means++ is used to identify these patterns on CPU and memory usage of applications and set scheduling policies for their co-location. This ensures that any new application can be categorized swiftly to make confident scheduling decisions early on. For smart manufacturing use cases, Jiang et al. propose an improved K-means algorithm to quickly scale the total number of nodes in the system based on network delay and available resources. [39].

2.3.3. Reinforcement Learning

Reinforcement learning is another popular method for resource allocation. Unlike the models discussed in the previous two categories, reinforcement learning does not always pick the most appropriate action. Actions taken differ vastly based on two concepts called exploration and exploitation [40]. Exploration allows the model to learn more about the environment and extend its knowledge base. On the other hand, exploitation is when the model acts according to past knowledge towards a positive

outcome. However, this means that the model has to make some arbitrary decisions to understand the environment better. Q-Learning is a reinforcement learning algorithm that is seeing application in edge systems. Q-learning uses a reward function and keeps a table with Q values representing the state-action pairs. Chien et al. improves upon this reward function to design a cache allocation mechanism suitable for edge networks [41]. Liu et al. challenges the usage of Q-tables as their size can grow increasing the need for the required computing resource and proposes a solution based on Deep Neural Networks [27]. Edsinger explores the different implementations of various reinforcement learning algorithms and points out that although deep Q-learning based algorithm can have better performance, State-Action-Reward-State-Action (SARSA) is the most stable [42]

The procedure of training the model beforehand is called offline training. In [43], a DNN is pre-trained with existing data to determine Q values accurately and make decisions afterward. However, a dataset may not always be readily available where the model has to consult online training, meaning that the model is trained as the data comes in. In order to reduce the negative impact of exploration on the stability of a newly introduced application, in [40], Dyna-Q and a model-based approach are preferred for orchestration.

The reinforcement learning solutions discussed up to this point are modeled after a single actor's interaction with the dynamic environment. In [44], the resource allocation problem is formulated as a non-cooperative game across computation tasks identified as individual agents. Zhao proposes a multi-agent based RL approach with multiple schedulers for large scale GPU cluster, where a single agent is insufficient to answer the requirements under heavy workload [45].

2.3.4. Time Series Analysis

The last branch in the taxonomy tree is time series forecasting. For the resource allocation problem, metrics used for forecasting are sets of data points collected over

time. Predictions made by [46] depend on this nature of the collected metrics for cloud service orchestration using Autoregressive Integrated Moving Average (ARIMA) modeling. Li et al. depends on the predictions made with ARIMA to make the final load prediction by using a neural network [47]. Tan et al. uses Vector Autoregression Model (VAR) to predict the load on edge data centers (EDC) [48]. In mobile edge computing (MEC), [29] bases their selective offloading algorithms on the predictions made by the ARIMA with Back Propagation (ARIMA-BP) model for energy optimization.

Predictive analytics is only a part of the solution for time series analysis. Temporal characteristics of IoT data open up the way for behavioral analytics, allowing [49,50] to base their algorithms on hourly aggregates of data. For MEC, [51, 52] exploit the daily recurring patterns of time series data and point out how identifying peak and off-peak hours are crucial for their predictions. Tom et al. applies the same on smart energy management and finds patterns in the day-to-day usage of IoT devices [53]. These daily patterns identified closely represent the human schedule, and [54] makes use of hourly buckets to make predictions in the long term.

2.4. Review on Edge Container Orchestration Frameworks

A review of edge container orchestration frameworks in the recent literature has been made. In this review, various aspects of edge container orchestration frameworks that are important in an edge computing scenario are presented. The full list of evaluation criteria used in this review can be found below:

- *C1 - Auto-scaling*: The container scaling options supported by the framework. If supported by the system, the options are horizontal (H) or vertical (V). While vertical scaling describes adapting the resources of an existing container, horizontal scaling refers to changing the number of containers of the same application to meet the varying load on an application.
- *C2 - Predictive Scaling*: The autonomous scaling decisions can be made on predictive system load. These predictions can be based on methods such as ma-

chine/deep learning, rule-based, or time series forecasting.

- *C3 - Software Heterogeneity*: Some orchestration frameworks offer limited support for the type of applications that they can host. The others can handle the heterogeneity of deployments and accommodate different kinds of applications like microservices, batch jobs, and streaming applications.
- *C4 - Resource Constrained Hardware*: Edge computing solutions can be designed for a multitude of devices from IoT devices to powerful cloudlets. These orchestration solutions are designed to be deployed on resource constrained devices such as IoT gateways.
- *C5 - Guaranteed Resource Allocation*: An orchestration framework can allocate resources and guarantee minimum and/or maximum amount of resources for the span of an application's execution, controlled through a set of resource constraints.
- *C6 - Optimization Metrics*: Auto-scaling decisions in each orchestration framework are based on some set of metrics. The two main categories are application (A) and system (S) level metrics. Application level metrics can be different based on the application. Some common examples are response time and error rate. System level metrics are resources observed from the system where the application runs, such as CPU and memory.
- *C7 - Monitoring Metrics*: In order to evaluate the operation of the system and, for some orchestration frameworks, take scaling actions most frameworks include metrics collection and monitoring solutions. These metrics can be application (A) and/or system (S) level metrics. Some frameworks include monitoring but do not declare specifically which metrics are collected.
- *C8 - Security*: There are security concerns on IoT edge computing systems regarding network (N), data (D), and application authenticity (U). An orchestration framework can incorporate solutions for these security concerns.
- *C9 - Multi-layer*: This literature review only covers frameworks that can be deployed on the edge. However, some orchestration framework can span both edge (E) and cloud (C) layers.
- *C10 - Event-driven Communication*: Event-driven communication is utilized by

some designs to benefit from asynchronous communication pipelines.

- *C11 - Decentralized Application Delivery*: Application delivery is a part of every orchestration framework but mostly handled through a centralized entity.
- *C12 - Decentralized Load Balancing*: With multiple devices connected to the system, the deployments can be distributed across all devices. The selection of a device to host the application can be a decentralized decision depending on the framework instead of relying on a centralized control plane.

Table 2.1. Literature Review of Edge Orchestration Frameworks.

	Criteria[1]											
Authors	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
[55]	H/V	✗	✓	✗	✓	S/A	S/A	✗	E/C	✗	✗	✗
[56]	H	✗	✗	✗	✓	S	S/A	✗	E	✗	✗	✓
[57]	✗	✗	✗	✗	✗	✗	S	D/U	E	✓	✗	✗
[58]	✗	✗	✓	✗	✓	S	S	✗	E	✗	✗	✗
[59]	✗	✗	✓	✓	✓	✗	✓	N	E	✗	✗	✗
[60]	H	✗	✓	✓	✓	✗	✓	N	E	✓	✗	✗
[13]	✗	✗	✗	✓	✗	✗	✗	✗	E	✗	✗	✗
[26]	H	✓	✗	✗	✗	✗	✓	✗	E/C	✗	✗	✗
[61]	H	✓	✗	✓	✓	A	A	✗	E	✗	✗	✓
[4]	H	✗	✓	✓	✓	S	S	N	E	✓	✗	✗
[62]	✗	✗	✓	✗	✓	✗	S	U	E/C	✗	✗	✓
[63]	✗	✗	✓	✗	✗	✗	✓	N	E	✓	✗	✗
This work	V	✓	✓	✓	✓	S	S	U	E	✓	✓	✓

[1] Description of Abbreviations: C1: Auto-scaling, C2: Predictive Scaling, C3: Software Heterogeneity, C4: Resource Constrained Hardware, C5: Guaranteed Resource Allocation, C6: Optimization Metrics, C7: Monitoring Metrics, C8: Security, C9: Multi-layer, C10: Event-driven Communication, C11: Decentralized Application Delivery, C12: Decentralized Load Balancing, H: Horizontal, V: Vertical, A: Application, S: System, E: Edge, C: Cloud, N: Network, D: Data, U: Application Authenticity

Selected studies on edge orchestration frameworks in Table 2.1 show where our framework stands in the literature. Most of these frameworks offer auto-scaling (C1) solutions, but almost all of them only support horizontal scaling. This type of scaling makes auto-scaling decisions more straightforward to manage by updating the deployments' replica count. However, frameworks offering only horizontal scaling require extra configuration and deployments for load management. They typically only support a single type of application. The scaling type selection is tightly correlated with software heterogeneity (C3). Our framework focuses on vertical scaling to satisfy the scaling needs of different types of applications without any extra configuration. We explore predictive scaling (C2) with vertical scaling of containers, while some works utilize predictive methods with horizontal scaling. The heterogeneity aspect of the edge also applies to the choice of hardware (C4). Most frameworks focus on the resource-constrained devices as we do, while the others are deployed on more powerful devices on the edge. This variance depends on which aspect or problem of orchestration each framework focuses on, such as availability, load-balancing, and scaling. Guaranteed resource allocation (C5) is one of the core aspects of orchestration frameworks. Its significance can be seen from the attention given by a wide range of frameworks. The same guarantee is a staple of our framework as well.

Optimization (C6) and monitoring (C7) metrics are closely associated as optimization metrics are limited by those being monitored. In our framework, we limited the collected metrics to the metrics that can be scraped from the system to be all-inclusive. Application metrics differ based on the application and do not fit with application heterogeneity. Although limited, application metrics can provide additional information for more accurate optimization. Hence, some frameworks collect application metrics and optimize deployments with them. Some of the works explore the security (C8) aspects alongside operational aspects. We emphasize the authenticity of applications as the delivery process (C11) is decentralized, unlike the rest of the reviewed works. Although only edge orchestration frameworks are compared, some have support for multiple layers (C9) spanning the cloud layer alongside the edge layer. As resources are abundant in the cloud, we have limited our framework to the edge to

concentrate on resource-constrained environments. Ours and several other works rely on event-driven communication (C10), as its adoption is crucial for any framework that aims for distributed deployments. It is also strategic for decentralized load balancing (C12) across a cluster of edge devices autonomously, which is offered by some of the reviewed orchestration frameworks as well as ours.

3. FRAMEWORK

The framework developed in this thesis is responsible for enabling resource-constrained edge systems to run IoT applications backed with a decentralized and dynamic orchestration platform. The framework manages the lifecycle of containerized applications, including design, delivery, execution, and optimization steps. The proposed design provides solutions for data storage, resource management, monitoring, and application registry and delivery which includes service upgrades and software updates. Individual components of the framework are designed with resource constraints, the authenticity of the applications, and autonomy in mind. Detailed explanations of the concepts and design choices are presented in the upcoming subsections.

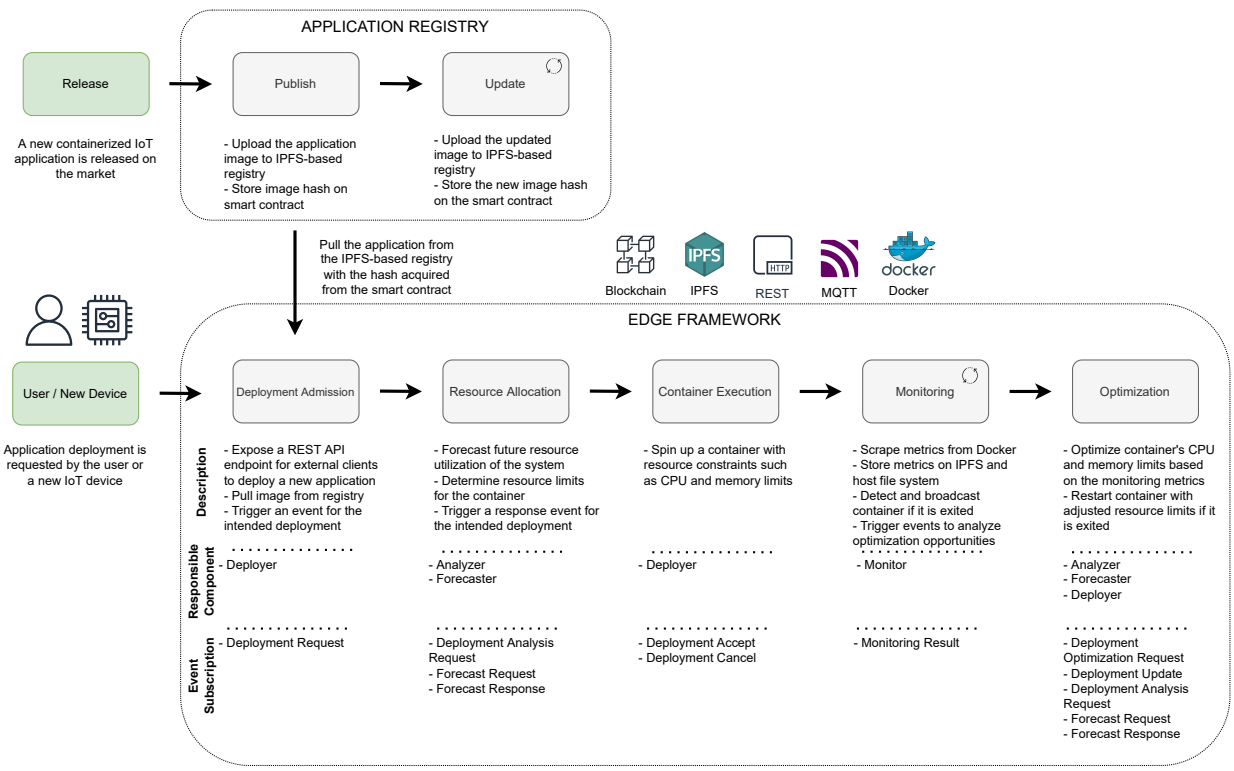

Figure 3.1. Architectural Overview of the Framework and Task Flow.[1]

[1]Licenses for the icons are as follows: IPFS Logo trademark of Juan Benet, CC BY-SA 3.0; Docker Logo trademark of dotCloud, Inc. Apache License 2.0; MQTT Logo, OASIS, Public domain

Two separate task flows are illustrated in Figure 3.1. The first task flow is the application delivery process through the application registry. The flow is triggered by the release of a new IoT application represented as *Release*. The *Publish* and *Update* steps represent the application delivery lifecycle. The *Update* step is a recurring process for each service upgrade and software update. The released or updated application is uploaded to an IPFS-based registry and published by the developer via a smart contract. The developer can also define a set of resource limits on CPU and memory utilization on the same smart contract executed outside of the edge network.

The second task flow takes place on the edge and is triggered by a *User* through the exposed REST API endpoint or a *New IoT Device* connecting to the network. This step fires a *Deployment Request* event through MQTT, which initiates a sequence of processes and events to fulfill this request. In the *Deployment Admission* step, the application and resource limits are pulled from the registry. The *Resource Allocation* step predicts the future availability of the system to check whether the device can host the new application and determines the CPU and memory limits based on the available resources of the device and provided configuration. Then, in the *Execution* step, the container is deployed with the assigned limits. After its execution, the *Monitoring* step starts checking its status and scraping system-level metrics. These metrics are initially stored locally but archived on IPFS based on the framework's configured retention time. *Optimization* step updates container limits periodically based on the knowledge accumulated from the previous steps.

The framework consists of four separately deployed components: *Deployer*, *Analyzer*, *Forecaster*, *Monitor*. The components responsible for each step are defined in Figure 3.1. Steps and components do not have a one-to-one relationship; a group of components can carry out a single step. Communication between these components is event-driven to ensure a more resilient asynchronous flow. Components are subscribed to MQTT topics and start consuming messages based on their subscription listed in the bottom row of Figure 3.1. This task flow is repeated whenever a new application is deployed on the system.

3.1. Preliminaries

3.1.1. Data Storage and Distribution

The challenges with data vary based on its characteristics. Methodologies that are able to provide solutions to multiple problems are given preference. Otherwise, these challenges have to be addressed individually. For example, application data and metrics require both short and long-term storage. A specific protocol should be chosen with multiple devices in the picture for data transmission. Data integrity should also be ensured no matter where it is stored. An end-to-end IoT system has to find answers to all of these problems.

It is possible to evaluate the value of the stored data to a degree. Some data could be discarded, but most have to be retained. This could purely be due to regulations or for further processing. Historical data is vital in making inferences on future behavior and improving current systems. However, being bound by limited space will probably cause the unnecessary disposal of valuable data. As a cheaper alternative, local drives could be used, but collected data far exceeds the capabilities of a local storage solution. It does not seem feasible to use a centralized approach to tackle this challenge with all these in mind.

Using a decentralized solution opens up a couple of possibilities. Firstly, any device with some storage can be used for data collection, reducing costs. A single local drive may be limited and packed silos in a cloud may be expensive, but a network of these units has a vast storage capacity. There is also the issue of reliability. There needs to be a system with replication features that would typically bloat up central storage to keep data safe.

IPFS is a protocol developed for similar challenges, and it can fulfill the storage requirements of the system. The following properties of IPFS make it a desirable choice. The first property is the data structure used by IPFS. IPFS creates a graph

structure from the hashes of the content stored in the network called a Merkle Directed Acyclic Graphs (DAG). This structure can be used to validate the authenticity and integrity of data. Therefore, any data pushed into the IPFS network is permanent, and it cannot be altered. Thus, it is safe to say that IoT data stored on IPFS is tamper-proof. Devices connected to the network, called IPFS nodes, form a peer-to-peer network. Data replication is another critical feature of IPFS. Data is divided into separate chunks, replicated, and spread across the network. Therefore, it can be considered as resilient as cloud-provided storage, if not more. The files are stored as objects in IPFS. These objects are replicated between nodes to provide fault tolerance as an inherent feature of IPFS. Therefore, files will be retained forever, even if a node goes down.

3.1.2. Blockchain Technology and Smart Contracts

Blockchain technology provides a platform for writing and executing applications without a central authority. Blockchain is a distributed database of transactions shared amongst the network participants. These participants, called nodes, use secure mechanisms provided by the underlying blockchain technology to achieve consensus by themselves. By validating new blocks, an immutable chain can be formed that can be used to provide a trusted platform without relying on a centralized third party. Such a platform enables developers to write smart contracts to build decentralized applications. Decentralized applications run on the blockchain and trigger autonomously when specific criteria are met. These criteria are defined by smart contracts stored and executed on the blockchain. Hence, trusted and transparent programs can be written and executed without a central authority.

The problem with a distributed file system like IPFS is the missing ownership of data. Data stored on IPFS cannot be traced to its owner unless proof of ownership is deliberately included. However, IoT application developers can use smart contracts to prove ownership and manage publicly disclosed data through decentralized applications. It should be noted that not all blockchains support smart contracts. Based

on the survey conducted in [64], Ethereum with the Solidity programming language is determined to be superior to other blockchain-language pairs because of its adoption rate and extensive documentation. Therefore, Ethereum is chosen to fulfill the role of data authentication for the framework.

3.1.3. Event-Driven Communication with MQTT

Message Queueing Telemetry Transport (MQTT) is a prominent lightweight protocol for IoT networks that can transport messages across devices and different components of a framework [65]. It is ideal as a messaging protocol for edge IoT applications with low resource and power consumption. MQTT is also reliable enough to be popular in the cloud layer for use cases, such as queuing and exchanging messages between nodes without any prior knowledge to achieve temporal and spatial decoupling where there are a large number of nodes [66].

MQTT adopts the publish/subscribe communication model. This model enables applications to send and receive messages and data asynchronously, supporting many-to-many communication. MQTT brokers are responsible for receiving and transmitting these messages and managing the connected applications called clients. This model enables devices to solely focus on transmitting the data and offload the delivery task to the brokers. The brokers make use of topics to separate data and link connections together. Data being broadcasted to a specific topic can be forwarded to multiple clients at once. Clients can subscribe to multiple topics that enable a steady data flow and open up many opportunities within the framework.

As the MQTT broker, the Mosquitto broker is chosen in this thesis work because of its small footprint [65]. One downside of MQTT is that it does not store data locally, and a new client does not receive previously published data. Kafka is a popular framework that offers these capabilities but leads to a higher overhead than MQTT while also depending on good infrastructure, and a stable network [67]. MQTT has some means of achieving this persistence with retained messages. Retained messages

are kept by the broker and delivered to the clients immediately as they establish a connection. This feature is essential for sharing the state between components of our framework as they are spun up on new devices joining the system or after a restart.

3.1.4. RESTful Web Services

An asynchronous communication protocol is not the correct solution for every scenario. A client can communicate with a server synchronously through an Application Programming Interface (API). Representational State Transfer (REST) defines a set of constraints for these APIs for a uniform method of communication in a heterogeneous edge environment. Thus, RESTful web services can handle the synchronous communication between our framework and users or IoT devices. These external clients rely on HTTP-based REST API calls to send requests to our framework, such as an application deployment request. The same APIs are not used within the framework to keep internal communication asynchronous, albeit being accessible by all components.

3.1.5. Virtualization

Another vital component of each IoT system is the applications producing data or consuming the generated data. Enhancing the capabilities of the edge layer requires modularity, and applications can reside independently from the underlying structure by employing such a strategy. To pursue this modular approach, the virtualization of applications is imperative.

In this thesis, we achieved virtualization by building an environment over the Docker platform and delivering applications as containers. A containerized application is packaged with all of its dependencies. Also, the resources of each container can be adjusted during runtime. Therefore, the containerized approach lays down the groundwork for edge computing and orchestration. Most applications do not consume a constant amount of resources, and some are temporary batch jobs. On a regular IoT device, resources would be dedicated to a single application that may not be able

to utilize all available resources continuously. However, other applications can use idle resources on-demand with virtualization and smart orchestration. Allocating the excess resources of an idle device to different applications allows multiple applications to share the same device.

Another opportunity with virtualization is that it makes it possible to ship not just applications but also components of the framework as containers. It is essential that the framework and the dependent tools can benefit from the Docker platform. IPFS is available as a desktop client via command line and a Docker image. Thus, the IPFS image satisfies the requirements set for components of the framework. MQTT brokers are also similarly available as Docker images.

The virtualization support comes in handy for data storage management as well. The container typically has a file system separate from the underlying operating system. However, folders of the host system can be mounted and used as storage for the node. As a result of this, the storage limits of a node can be adjusted when necessary.

The framework developed in this thesis interacts with the Docker daemon with its REST API endpoints [68]. These endpoints provide the full capabilities of the CLI commands and allow components of the framework to execute commands with HTTP requests. The RESTful nature of these endpoints makes implementation with all languages possible, provided that they support HTTP clients. A list of the endpoints utilized by the framework can be found below:

- *ContainerCreate*: This endpoint is used upon an incoming deployment request to create a new container by defining the image, name, and resource limits accordingly.
- *ContainerStart*: Containers created can be stopped and restarted on demand. Therefore creating a container does not automatically deploy a container, and this endpoint starts the created container.
- *ContainerUpdate*: Containers configuration can be updated during their runtime.

This API is used to update resource limits by the orchestration system without bringing the application down.

- *ContainerList*: Monitoring is an essential part of the whole system. This endpoint returns a list of running and stopped containers with their hashes. Following endpoints can then be used to get extra details by providing the hashes acquired.
- *ContainerStats*: This endpoint returns the resource usage metrics for the running containers. These metrics provide a snapshot of the resource usage of CPU, memory, bandwidth, etc., at the time of the request.
- *ContainerInspect*: The inspect endpoint provides detailed information for the requested container. Inspect is required when the information returned by the previous containers is not detailed enough.

Depending on the OS of the host, redirection of the sockets might be necessary to expose the API to the host network. Socket redirection can be achieved by running a relay as a container. Implementation details are discussed in Appendix A.

Another OS-dependent issue is the Docker platform itself, as it has to be versatile enough to be installed on the gateway devices. As discovered by [69], most gateway devices are built on similar operating systems based on Linux or Android. Since Docker can run natively on Linux on CPU architectures like x86-64, ARM, and more, the framework can rely on Docker for virtualization.

3.1.5.1. Docker Resource Management. Applications deployed on Docker do not have any constraints enforced by the daemon. The natural limits for each container are the constraints of the host system. However, Docker API provides parameters on multiple endpoints to control the resource allocation of deployments during their lifetime. *DockerCreate* can be used to define the resource limits during container creation, and *DockerUpdate* can modify the resource limits of a running container. Following parameters are set on various stages for resource management of containers. [70]:

- *memory*: This parameter is used to define a hard limit on the amount of memory a container can use. If the limit is exceeded, the container exits with an OOMKilled error message, meaning that the memory requested by the application has exceeded the defined limits.
- *memory-swap*: Swap usage allows the container to use the host system's storage to extend the defined memory limits. The storage device hardware can vary significantly between systems. Excessive swap usage can result in performance degradation, and the daemon could be competing for resources with other applications. Also, this storage usage is not monitored by the framework. Thus, the *memory-swap* parameter is always set equal to memory to disable swap usage.
- *cpu*: The amount of CPU guaranteed for a container's usage. This parameter can be configured based on the CPU count of the host system. Furthermore, Docker provides two additional parameters for fine-grained control directly on the scheduler with *cpu-period* and *cpu-quota* parameters.
- *cpu-period*: This parameter is used to define the period of the *Completely Fair Scheduler* (CFS), which is the default Linux kernel scheduler [70]. Unless the use case necessitates a different period, the default value of 100000 microseconds is used.
- *cpu-quota*: The quota parameter is used in conjunction with the period. It defines a limit on the CPU time allocated to a container within the specified period before the application is throttled. The framework adapts this parameter to manage the CPU limits of the deployments.

3.1.5.2. Decentralized Docker Registry. Containerization concepts touched upon until this point in this thesis book are based on a set of instructions defined as a Docker image. These images are built by the developers and distributed over a Docker registry. The Docker registry, however, is a centralized platform. Whether it is Docker's own registry called Docker Hub or a privately hosted one, they are all centralized solutions [71]. However, images can be regarded as just another piece of data, and as previously stated, any file can be stored on IPFS as an object. Therefore, IPFS can be used as the file system for a decentralized Docker registry.

Docker registry is more than a simple file storage system. It is a tool for storing and distributing images. Thus, a Docker registry proxy called IPFS-Backed Docker Registry (IPDR) is utilized alongside IPFS in this thesis [72]. With IPDR in place, the framework can fully benefit from the REST API of the Docker registry. Versioning, updates, and upgrades can be managed on top of IPFS using IPDR as a decentralized and agile solution. Docker images are composed of layers, and these layers can be downloaded and cached individually. IPDR offers the advantage of downloading image layers separately, which is necessary for using less bandwidth and local storage overall. Since all application deployments are containerized, updates and upgrades in the form of additional layers can minimize their impact on bandwidth usage.

3.2. Edge Framework Architecture

The orchestration flow of a deployment can be represented with a series of tasks as shown in Figure 3.1. Each task is carried out by one or more components of the framework. It is crucial to understand the role of each component and their subscription of events before moving onto the orchestration steps.

3.2.1. Framework Components on the Edge

The four components of the framework with the above-mentioned tools and technologies forming the framework deployed on the edge gateway as shown in Figure 3.2:

- *Monitor*: Scrapes metrics from the host system and shares them with the rest of the framework.
- *Deployer*: Responds to application deployment and resource optimization requests. New deployment requests are accepted through a REST endpoint.
- *Analyzer*: Evaluates incoming requests with the data made available by the other components and makes predictive analysis
- *Forecaster*: Provides time series forecasting capabilities to the framework
- *Docker*: Provides an engine with virtualization capabilities to the edge and means

to execute and monitor containerized applications

- *IPFS*: Grants the distributed file storage for the Docker registry and long term storage for application metrics
- *Blockchain*: Smart contracts connect the application delivery process with the edge framework
- *MQTT Broker*: Provide an event-driven communication channel between components
- *Host File System*: A mounted volume through Docker grants short term storage for application metrics on the host device

The four components of the framework rely on the tools and technologies above to access the application registry and establish asynchronous communication on the edge using the publish/subscribe pattern. The dependencies between each component are also represented in Figure 3.2.

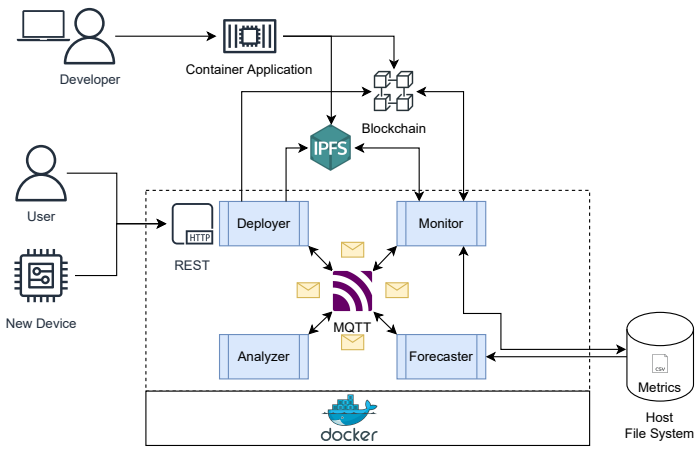

Figure 3.2. Overview of the Framework's Architecture on the Edge.[2]

[2]Licenses for the icons are as follows: IPFS Logo trademark of Juan Benet, CC BY-SA 3.0; Docker Logo trademark of dotCloud, Inc. Apache License 2.0; MQTT Logo, OASIS, Public domain

3.2.2. Messaging and Subscriptions

As an event-driven architecture, the messages in circulation are used to communicate between the framework's components with the publish and subscribe pattern. Components publish messages on specific topics to pass events and data for subscribed components to pick up. Components subscribed to these topics, called consumers, interpret the received messages and act upon the received payload. Messages use the JSON format. This format includes both the field names called the schema and the values in a single message. Although the schema increases the overhead of each message, considering the low amount of messages generated by our framework, the compromise does not cause a latency overhead. There are schemaless options like AVRO or a schemaless JSON as well. However, these alternatives require a separate schema registry. Since the number of nodes and message traffic is limited, there are no major latency concerns with a JSON implementation. Thus, a schemaless alternative is unnecessary, especially considering the added complexity of maintaining a registry. In Figure 3.3, an example message generated by the framework can be seen, with the schema as part of its payload.

```
{
  "trace_id": "123e4567",
  "action": "deployment_request",
  "image_name": "busybox",
  "message": "This is a request for the deployment of busybox"
}
```

Figure 3.3. Example Message Format.

All orchestration messages exchanged over MQTT must conform to a set of rules to ensure uniformity across the pipeline. These rules cover the field names, allowed values for these fields, and the hierarchy of nested objects. A malformed structure may result in incoherent, rejected, or misrouted messages. The following fields are supported:

- *trace_id*: A unique id generated for each new sequence of events for the observability of individual task flows.
- *action*: The identifier for specifying the purpose of the message with the action set listed in Table 3.1 .
- *image_name*: A list of image names corresponding to the applications currently running on the framework.
- *containers*: A list of container ids of running applications on the framework.
- *predictions*: A list of the expected resource utilization values of a container.
- *forecast_fields*: The framework gives the flexibility to work on specific metrics. Currently, supported values are CPU, memory, and throttling; the orchestrator can be configured to ignore some metrics by omitting them from the messages. The framework can be further extended to take network and block i/o utilization into account.
- *resources*: This field defines the CPU and memory limits enforced on an application. The values can be acquired from the smart contract provided by the developer or set by the orchestrator.
- *stats*: The field shares monitoring results by publishing metrics scraped from the host system.
- *retry_count*: Deployment sequence of initially rejected requests or exited applications can be retriggered up to a certain threshold. Some components may adjust their behavior based on the number of retries. When the maximum number of retries are exceeded a *deployment cancel* event is published for reporting.
- *availability*: Available resources of the host system. This field is used in cluster deployment to share idle resource information across heterogeneous devices. Supported values are CPU and memory.
- *ip_address*: The IP address of the edge device. This field is used in cluster deployment to identify the owner of events received through MQTT bridge configuration.
- *message*: An optional field that can be used for logging purposes.

The complete list of potential messages can be found in Appendix B.

Table 3.1. Supported Orchestration Actions.

Action	**Description**	**Topic**
Deployment Request	A declaration of intention to deploy a new application to the platform	Deploy
Deployment Analysis Request	An analysis request for the platform's availability of resources to host a new application	Analyze
Deployment Optimization Request	A request for optimization of resource limits imposed on an existing application	Analyze
Forecast Request	A request to forecast selected applications' future resource utilization	Forecast
Forecast Response	Forecasted resource utilization values	Forecast
Deployment Accept	Approval of a deployment request	Deploy
Deployment Cancel	Rejection of a deployment request	Deploy
Deployment Update	Approval and new limits for an optimization request	Deploy
Monitoring Result	Container metrics scraped from the host device	Monitor

3.3. Application Delivery

The application delivery process provides a pipeline for developers to publish and deliver applications to their clients. This process requires a decentralized application registry and means to publish and update applications. The flow is initiated by the *Release* step as shown in Figure 3.1.

A decentralized registry based on IPFS and IPDR is used to store and deliver applications. An interface for this decentralized Docker registry is developed as a decentralized application. In this thesis, a smart contract is written and deployed to provide this functionality. Developers start the initial application delivery step of an

application, called the *Release* process. The application release flow consists of *Publish* and subsequent *Update* processes. Both processes manipulate the application metadata stored on the blockchain through the smart contract. In the meantime, Docker images are stored in IPFS. The metadata primarily consists of the image hash. However, developers can also publish recommended and baseline resource limits. The smart contract allows developers to prove the application's authenticity and deliver updates and upgrades.

The developers provide the image hash and resource limits on the smart contract. The *image_hash* field is used to store the IPFS hash of the image uploaded on IPFS via IPDR. Updating this value ensures that only the updated image can be pulled from the registry. The *limits* field has two subfields: *base* and *request*. The application developer can define CPU and memory limits with these fields. These subfields are suitable for defining SLAs over the availability of resources in the form of minimum resource commitments. The framework can then take these values as a reference while setting the resource limits of the deployed containers under diverse conditions. If no resources have been defined by the developer, the framework will assign a set of default values during deployment. The framework optimizes the resources allocated to each container over time. Developers are encouraged to provide these limits to smooth out the process, decreasing the number of retries and minimizing the potential disruptions to the system.

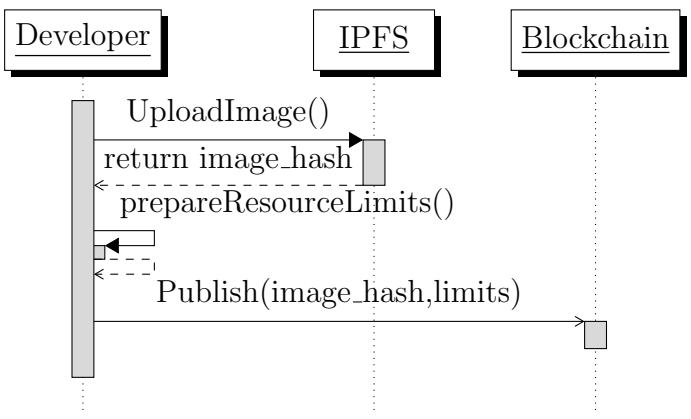

Figure 3.4. Developer Publish Sequence Diagram.

The *Publish* step is illustrated in Figure 3.4. The application developer first uploads the Docker image on IPFS and receives the *image_hash* in return which can be used to locate the image. Then the developer may define the limits. Afterwards, both the *image_hash* and, if defined, *limits* should be put on the blockchain via the provided smart contract.

3.4. Orchestration Flow

The orchestration flow is a step-by-step process executed on the edge to handle the lifecycle of deployments. The flow is initiated on the edge framework by the *User/New Device* as shown in Figure 3.1. It is responsible for managing incoming application deployments and dynamically adapting allocated resources for each container. The self-adaptive properties are modeled after Monitor-Analyze-Plan-Execute-Knowledge (MAPE-K) loops [73]. The framework components listed in Section 3.2.1 carry out tasks on the edge. They asynchronously pass messages between each other for separation of concerns and to provide a dynamic orchestration [10].

3.4.1. Deployment Admission

This step handles the application deployment requests by acquiring the information stored on the blockchain and interacting with the Docker API. The sequence of actions taken in this step are illustrated in Figure 3.5. The deployment request is initiated by a request that is made externally using the REST API provided by the *Deployer*. The actor invoking the request can either be the user or an IoT device connected to the network. The API is only responsible for publishing the request to the *deploy* topic as a new event. Publishing such events is essential for notifying all subscribers about the incoming deployment requests. The only mandatory information for a deployment request is the application name which can be matched with an application on the blockchain. If a matching entry for the application can be found on the contract, the IPFS hash for the image and optional base and request limits can be acquired. The high level interaction between the *Deployer* and the external technolo-

gies are shown in Figure 3.2. If the developer provides no limits, the *Deployer* uses a set of default resource values as a fallback. Then, if another container is deployed with the same image name, indicating that the application was previously deployed, its execution is stopped to open up room for redeployment. Finally, a deployment analysis request with the request limits is published on the *analyze* topic to be analyzed in the *Resource Allocation* step.

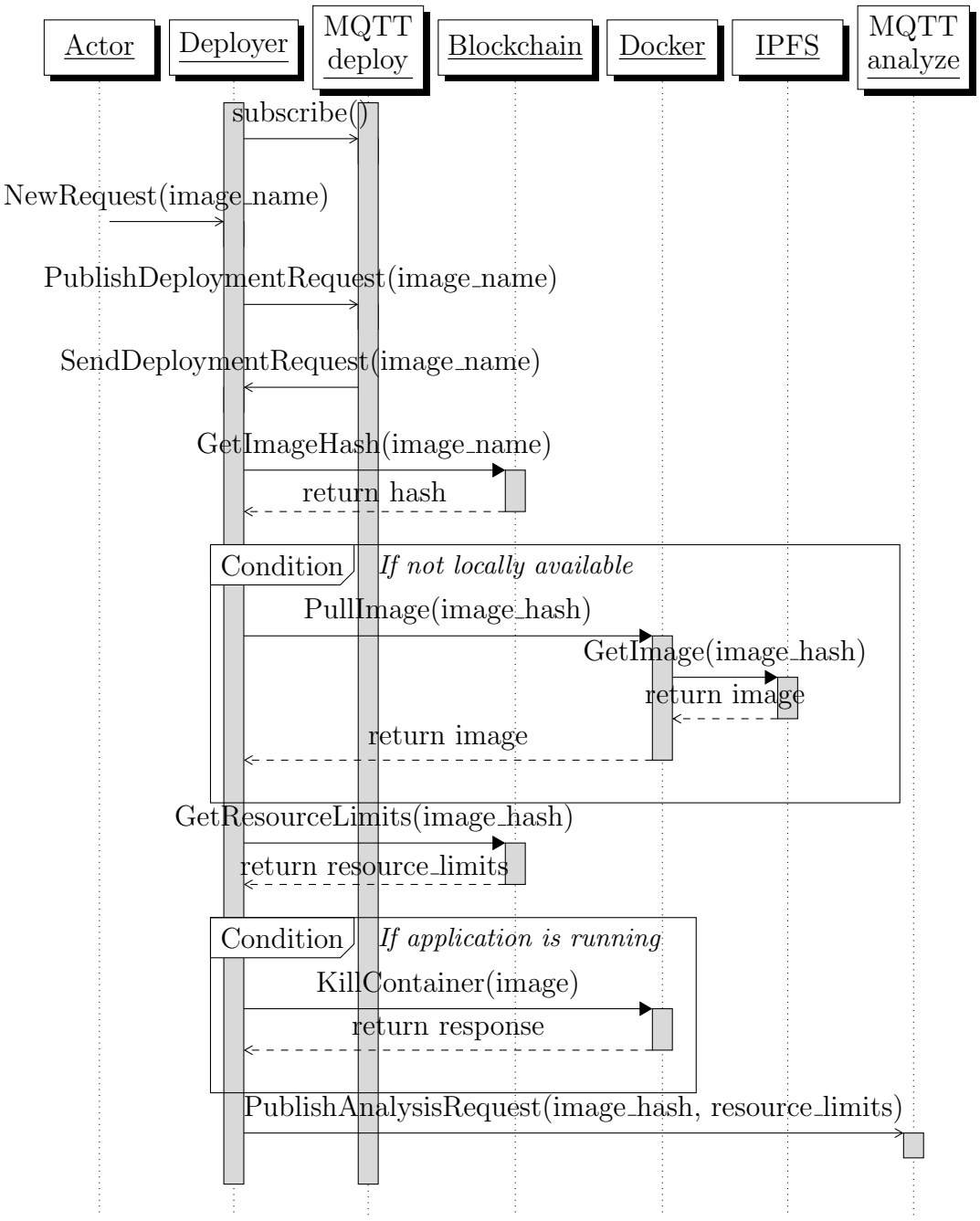

Figure 3.5. Deployment Request Sequence Diagram.

The deployment request triggers a sequence of events which is called the *deployment workflow*. The workflow starts with the request picked up by the *Deployer* and includes the steps until the application deployment is rejected or executed in *Container Execution*. The events, topics, and components responsible for concluding the deployment workflow are depicted in Figure 3.6.

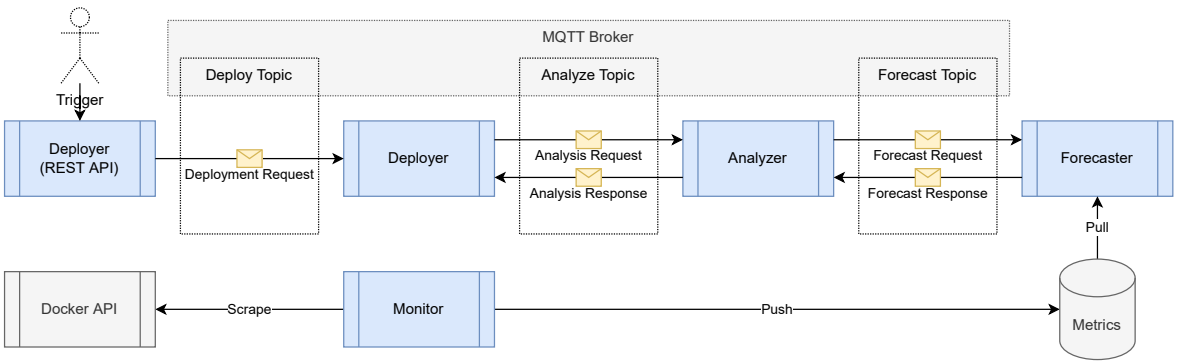

Figure 3.6. Deployment Workflow.

3.4.2. Resource Allocation

Each application is deployed with resource limits imposed on its CPU and memory usage by the framework. Otherwise, they might use as much CPU or memory available on the host. The *Resource Allocation* step is responsible for assigning appropriate limits to each application based on the status of other active deployments and given base and request resource limits. A deployment request message contains the image and resource limits of the application. The *Analyzer* is the component where all the decisions are made in the framework. A knowledge base in the form of the metrics and status reports are provided by the *Monitor* and resource utilization predictions are made by the *Forecaster*. The *Analyzer*, with the data provided by *Monitor* and *Forecaster*, evaluates the available information and returns an approval response. This response includes the target resource limits, $L_{i,j}^{target}$, for the container scheduled for deployment where i in $\{cpu, mem\}$, for CPU and memory resources, respectively, of container j. Otherwise, the deployment is rejected by publishing a *deployment cancel* event.

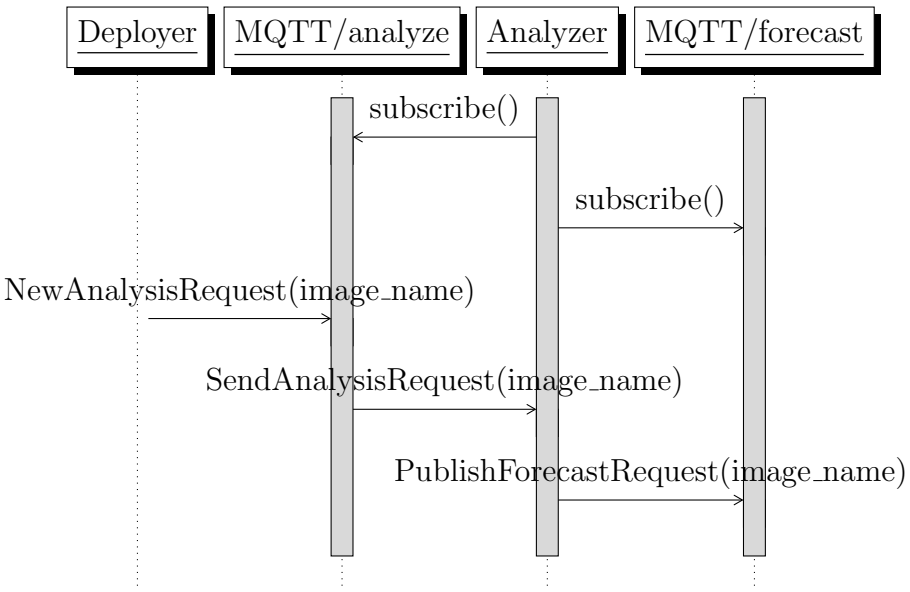

Figure 3.7. Deployment Analysis Sequence Diagram.

3.4.2.1. Deployment Analysis and Forecasting Requests. The *Analyzer* is subscribed to two different topics: *analyze* and *forecast*. It listens for analysis requests and publishes the responses on the *analyze* topic as shown in Figure 3.7. The analysis is for the feasibility of a new deployment with its resource requirements. A deployment analysis request is picked up by *Analyzer* through the *analyze* topic.

Forecasting provides insight into the future resource usage of active deployments. Whenever forecasting on future resource utilization is necessary, a request is published to the *forecast* topic. *Analyzer* is subscribed to the *forecast* topic to pick up the responses which contain future resource utilization predictions for all active containers as seen in Figure 3.8 and continue with the rest of the analysis.

3.4.2.2. Forecasting. The *Forecaster* uses the time series metrics data made available by the *Monitor* to predict the future resource utilization of requested containers. *Forecaster* accesses this locally stored data, also known as the knowledge base for the given container. If the container is a fresh instance, knowledge belonging to the past container instances of the same image is used instead. This enables the *Forecaster* to use past knowledge aggregated from previous instances of the same container that have died recently or recurring batch jobs.

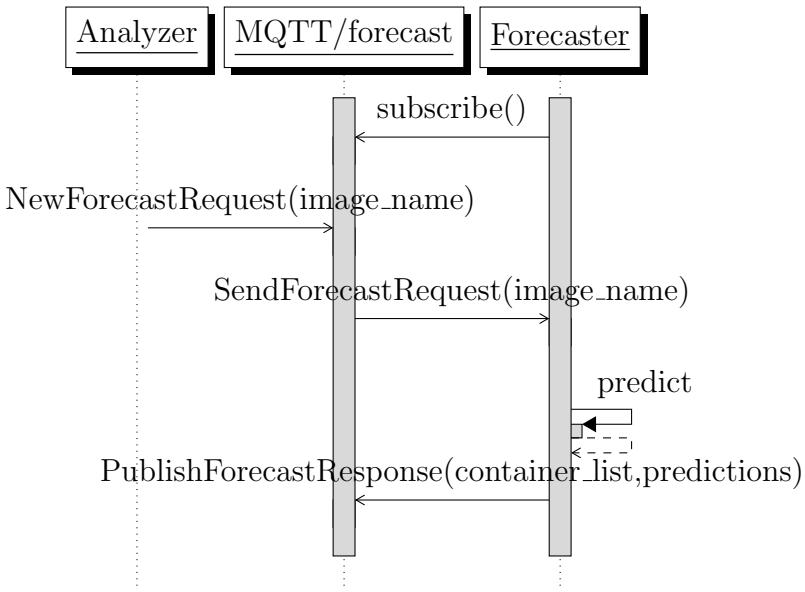

Figure 3.8. Forecast Sequence Diagram.

Forecaster can also be configured to skip some of the available metrics. The forecasting will only be based on the values provided in the *forecast_fields* field of the request message in Appendix B. The current implementation of the framework is configured to predict on three metrics only: CPU, memory, and throttling. However, any combination of the seven metrics collected by the *Monitor* can be used. These metrics are explained in Section 3.4.4.1.

The *Forecaster* conducts a time series analysis as shown in Figure 3.9. The ARIMA model is used to interpret the trends in time series data, and predict the future values with statistical analysis [74, 75]. The historical $C_{r,c}^{util}$ data stored by our framework is used as the input data. The *Auto-Regression* and *Integrated* models of ARIMA are configured together to capture non-stationary patterns seen in each analyzed metric and forecast their values for the next optimization cycle. The ARIMA model is used with an order of (5,1,0), which is able to capture the short time trends in available metrics data. This configuration sets up a lag order of 5 to smooth the timeseries data and a degree of differencing of 1 represented as

$$y'_t = c + \phi_1 y'_{t-1} + \phi_2 y'_{t-2} + \phi_3 y'_{t-3} + \phi_4 y'_{t-4} + \phi_5 y'_{t-5}, \tag{3.1}$$

where $y'_t = y_t - y_{t-1}$. However, this model does not capture trends that can span a longer period, like daily patterns. Seasonal ARIMA could potentially fulfill this role given enough data points, but it proved lackluster because of short retention times.

Input: $R, C, history(C^{util}_{r,c})$, r ∈ R and c ∈ C
Output: $P^{util}_{r,c}$
for $c \in C$ **do**
  for $r \in R$ **do**
    $\mathbf{P}^{util}_{r,c} := ARIMA.forecast(history(\mathbf{C}^{util}_{r,c}))$
    $hourly\_max := map()$
    for $h \leftarrow 0, 24$ **do**
      $hourly\_group[h] := group\_by(history(\mathbf{C}^{util}_{r,c}), 'H24')$
      $hourly\_max[h] := max(hourly\_group[h])$
    end for
    $predictionLength := sizeOf(\mathbf{P}^{util}_{r,c})$
    for $t \leftarrow 0, predictionLength$ **do**
      $h := getTimestamp(\mathbf{P}^{util}_{r,c}[t]).hour$
      $adj\_hourly\_max := max(hourly\_max[h-1 : h+1])$
      $\mathbf{P}^{util}_{r,c}[t] := max(\mathbf{L}^{current}_{r,c}, adj\_hourly\_max)$
    end for
    return $\mathbf{P}^{util}_{r,c}$
  end for
end for

Figure 3.9. ARIMA Forecasting and Time Series Analysis.

Retention time of historical data is managed by the *Monitor* as discussed in Section 3.4.4. When the use case scenarios are examined in this thesis, it becomes apparent that the daily patterns are a dominant factor. In order to take this temporal characteristic into account, data points are aggregated hourly. Then, the maximum value observed in the same or adjacent hours' aggregate is taken. This value is then compared with each prediction result. Any prediction lower than the max observed

value is replaced with the maximum observed value. This step is necessary to lower the risk of optimization disrupting the lifecycle of a running container, especially with memory adjustment.

The forecasting process is not responsible for collecting the data or making decisions based on the predictions. It simply publishes the forecasting results back to the *forecast* topic.

3.4.2.3. Deployment Analysis Response. Once the forecasting results, $P_{r,c}^{util}(t)$, are received, a response can be derived for the analysis request. The resource allocation for a new deployment should not be based on the amount of resources currently available on the system, denoted by S_r^{avail}. If other active deployments on the system need to be upscaled, their resource allocation requests should be prioritized to maximize availability and uptime. Deployment analysis should exclude the amount of resources needed for upscale optimizations in its resource allocation calculations. Therefore, the future availability of the system's resources, P_r^{avail}, is calculated first as shown in the algorithm in Figure 3.10. Then, P_r^{avail} value is used for all subsequent calculations during the analysis of this particular deployment instead of S_r^{avail}.

The calculation of P_r^{avail} is based on the forecasted utilization of resources $P_{r,c}^{util}$ where r in $\{cpu, mem\}$, for CPU and memory resources respectively of each container c. The aim of this calculation is to capture upwards trends in resource utilization of existing containers. Any prediction below $L_{r,c}^{current}$ can stay at the current limit value. Thus, each prediction in $P_{r,c}^{util}[t]$ for time t is compared with $L_{r,c}^{current}$ and updated with the maximum of these two values. This ensures that no new deployment can allocate resources predicted to be allocated to an existing container. Using $P_{r,c}^{util}$ ensures that an existing container will not end up being *OOMKilled* or with excessive throttling. Finally, the maximum $P_{r,c}^{util}[t]$ is subtracted from the system's total available resources, S_r^{total}, to find the predicted availability of resources, $P_{r,S}^{avail}$, for the system S.

Input: $\mathbf{S}, \mathbf{R}, \mathbf{C}, \mathbf{L}_{r,c}^{current}, \mathbf{P}_{r,c}^{util}$ r $\in \mathbf{R}$ and c $\in \mathbf{C}$

Output: $\mathbf{P}_{r,S}^{avail}$

$\mathbf{P}_{r,S}^{avail} := \mathbf{S}_{i}^{total}$

for $r \in \mathbf{R}$ **do**

  for $c \in \mathbf{C}$ **do**

    $predictionLength := sizeOf(\mathbf{P}_{r,c}^{util})$

    for $t \leftarrow 0, predictionLength$ **do**

      $\mathbf{P}_{r,c}^{util}[t] := max(\mathbf{L}_{r,c}^{current}, \mathbf{P}_{r,c}^{util}[t])$

    end for

    $\mathbf{P}_{r,S}^{avail} := \mathbf{P}_{r,S}^{avail} - max_t(\mathbf{P}_{r,c}^{util}[t])$

  end for

end for

Figure 3.10. Prediction of the Resource Availability of the System.

The analysis request also contains the resources that should be allocated to the new deployment indicated by a set of target resource limits, $L_{r,c}^{target}$ on the container. $L_{r,c}^{target}$ is provided with the deployment analysis request message as shown in Figure B.2. The message contains one of the resource limit values $L_{r,c}^{request}$, $L_{r,c}^{base}$ or $L_{r}^{default}$ as $L_{r,c}^{target}$. If neither $L_{r,c}^{request}$ nor $L_{r,c}^{base}$ was provided on the smart contract, the default resource limits, $L_{r}^{default}$ is sent instead. The handling of rejected deployments are explained in Section 3.4.3.

The *Analyzer* compares the previously calculated $P_{r,S}^{avail}$ with the $L_{r,c}^{target}$ value as shown in the algorithm in Figure 3.11. The deployment request is approved if the $L_{r,c}^{target}$ is less than $P_{r,S}^{avail}$, meaning that the system will have enough resources to accommodate the incoming application deployment. Otherwise, the request is rejected. Either way, responses are published on the *analyze* topic, and the *Analyzer* does not take any further action.

Input: $\mathbf{P}_{r,S}^{avail}$, $\mathbf{L}_{r,c}^{target}$ where r ∈ R and c ∈ C
  for $r \in R$ **do**
    if $\mathbf{L}_{r,c}^{target} < \mathbf{P}_{r,S}^{avail}$ **then**
      continue
    else
      publishRejectDeployment() ▷ Reject
      return
    end if
  end for
  publishApproveDeployment() ▷ Approve

Figure 3.11. Deployment Availability Analysis.

3.4.3. Container Execution

The execution of a container is handled by the *Deployer* but depends on an approval message from the *Analyzer*. Deployment analysis responses received by the *Deployer* can be twofold, corresponding to either an approved deployment or a rejected one as shown in Figure 3.12. If a deployment is tried with $L_{r,c}^{request}$ and rejected, the *Deployer* sends a second analysis request with $L_{r,c}^{base}$. For accepted deployments, a container is executed with $L_{r,c}^{target}$, successfully allocating the required amount of resources.

3.4.4. Monitoring

The monitoring process involves collecting metrics of the application running on the framework from the underlying host system and keeping the rest of the framework up to date about the status of deployed applications. The monitoring sequence is shown in Figure 3.13. The responsibility of the monitoring components consists of three individual steps:

- Metrics collection
- Application failure detection
- Optimization request

These three steps are triggered periodically. The trigger intervals have default values, which can be overridden during the initial setup. The default value for the first two monitoring steps is one minute, and the optimization request step is five minutes. Optimization is set to run less frequently to ensure that the system is not disrupted unnecessarily and there is time for collecting enough metrics between cycles for the forecasting process.

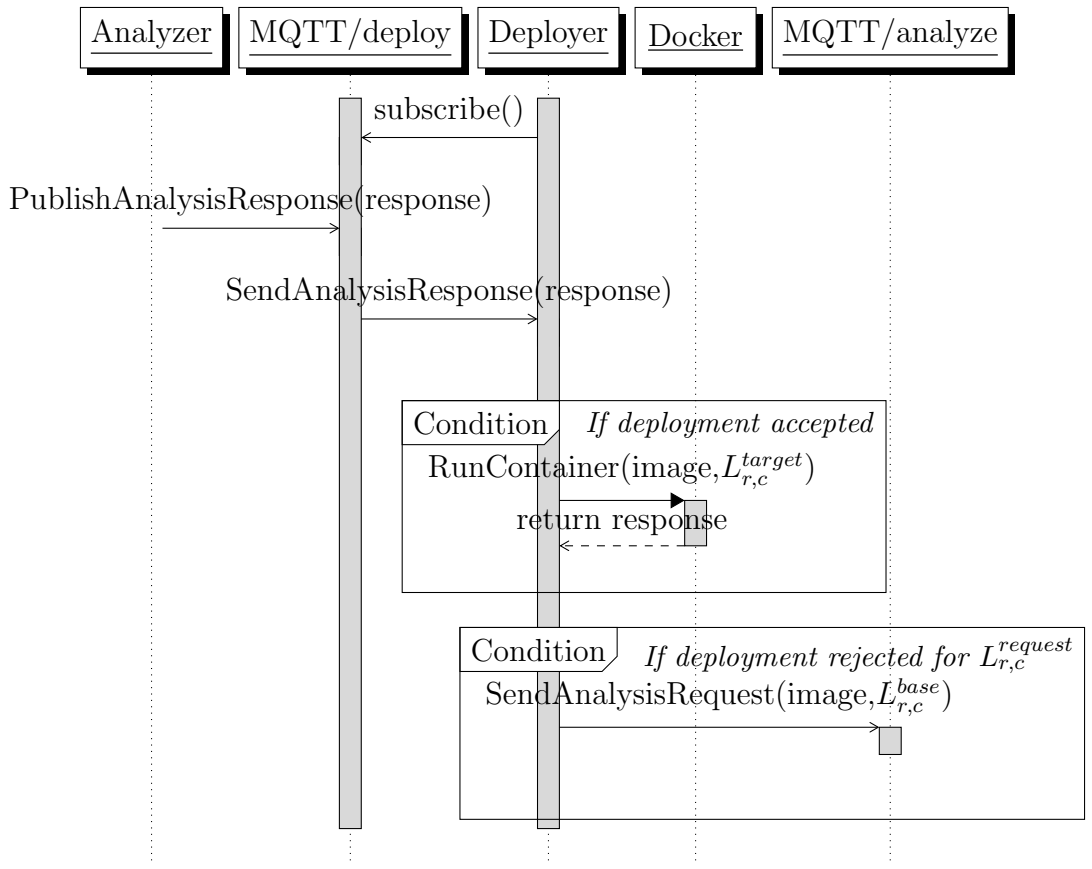

Figure 3.12. Deployment Response Sequence Diagram.

3.4.4.1. Metrics Collection. Metrics collection is where the application resource utilization metrics are scraped from the host system. Metrics are acquired from Docker API by calling the relevant endpoints and *cgroup* stats in Linux systems. First, the

running containers are identified with the *ContainerList* endpoint, which returns details about active containers such as image_id, image_hash, command, etc. Only the *container_id* information is of particular interest at this point. Their *id* is then passed to the stats endpoint for each active container to retrieve the metrics. This endpoint is very verbose, with a long list of container statistics. It does not make sense to store each metric returned by this endpoint. Therefore, the response is filtered by the *Monitor* to reduce the dataset to 6 metrics:

- CPU utilization percentage
- Memory usage
- Number of bytes transferred from the system (Block read)
- Number of bytes transferred to the system (Block write)
- Number of bytes received from the network
- Number of bytes transmitted to the network

There is one more metric crucial for making forecasts and optimizations, which is the *cpu_throttling* percentage. This value cannot be retrieved from the Docker API and should be derived from the raw CPU throttling stats of that particular container's scope found in Linux *systemd*. The following metrics can be acquired from *cgroups* [76]:

- *nr_period*: Number of periods that the container was runnable
- *nr_throttled*: Number of periods that the whole available cpu-quota was used
- *nr_throttled_time*: The total amount of time the container was throttled

The throttling percentage denoted by $C^{throttle}$ of a container in the current interval can then be acquired with

$$\begin{aligned} nr\_throttled_{diff} &= nr\_throttled - nr\_throttled_{prev} \\ nr\_period_{diff} &= nr\_period - nr\_period_{prev} \\ C^{throttle} &= \frac{nr\_throttled_{diff}}{nr\_period_{diff}}, \end{aligned} \tag{3.2}$$

where *prev* subscript indicates the values scraped in the previous interval [76]. Then these seven data points are stored in Comma-separated values (CSV) format on the local disk with the *image_name* and *container_id*. CSV format is chosen because it is universal, has minimal overhead, has a static schema, and is lightweight.

Based on the configured retention time of metrics with a default of two days, historical values are aggregated in a single file, uploaded to IPFS, and removed from local storage as shown in Figure 3.13. The default retention time is set as two days to favor recent daily trends, but it can be prolonged to capture weekly or monthly trends if storage space is abundant by increasing the built-in retention time. As shown in Figure 3.2, the *Monitor* depends on IPFS and smart contracts for archive and backup processes.

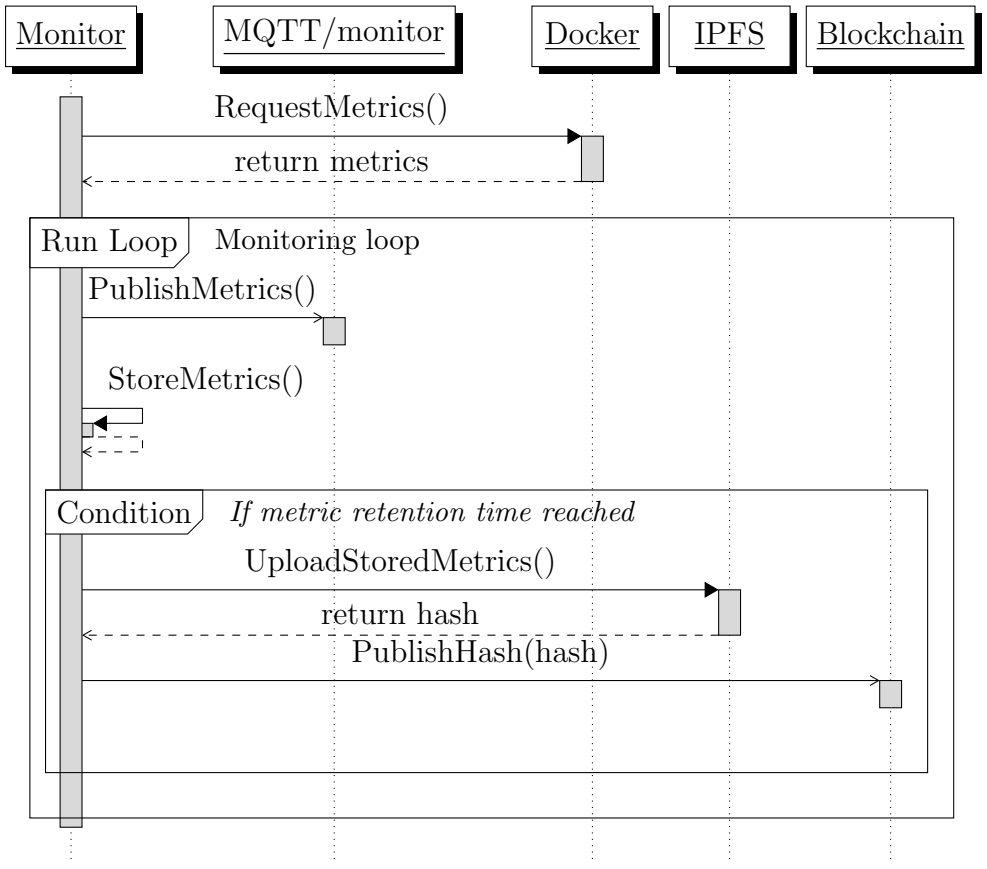

Figure 3.13. Monitoring Sequence Diagram.

A smart contract is provided on the blockchain to store the IPFS hashes of the archived data. These hashes are stored on the blockchain with the user's unique id for

retrieval in the following scenarios; if the gateway device is replaced or another device is added to the same edge network for the past knowledge to be shared across devices.

3.4.4.2. Container Availability. One of the responsibilities of the *Monitor* is identifying applications that have stopped prematurely. Docker keeps track of such containers called *exited* containers and provides a list of them through the *ContainerList* endpoint. Then filtering is applied client-side only to keep containers exited in the last hour. The information acquired from this endpoint is not enough to deduce the reason behind the container's termination. Another request must be made to the *DockerInspect* endpoint to check whether the container exited abruptly or not. Here, information about whether the container was killed because of running out of memory can also be found. In such cases, the container is flagged with an *OOMKilled* exit code. This bit of information is necessary for determining the resource limits of the container during the subsequent deployment requests. After the acquired list is filtered, it is grouped by *image_name* to determine how many times the deployment has failed in the last hour to determine the *retry_count* set in the deployment request messages. This metric is also used to determine the resource limits for the next deployment attempt. Similar to the sequence in Figure 3.5, for each identified application, a deployment request is published by the *Monitor* on the *deploy* topic to be handled by the *Deployer*.

3.4.4.3. Optimization Request. The *Monitor* is not directly responsible for optimizing running applications; however, the *Monitor* initiates the optimization process. In a longer interval, as discussed earlier, an optimization analysis request for an active container is published on the *analyze* topic for the other components of the framework to handle.

3.4.5. Optimization

During the optimization step, the framework dynamically adjusts the resource limits of an active container. In the optimization workflow depicted in Figure 3.14, the

Deployer has responsibilities similar to that in the deployment workflow. It interacts with the Docker engine to update the current resource limits of the container, $L_{r,c}^{current}$ if there is a need for optimization. The *Analyzer* determines if an optimization is necessary. The target resource limits, $L_{r,c}^{target}$ are calculated based on the predictions provided by the *Forecaster*. Then, an analysis response message is published on the *deploy* topic for the container if it is scheduled for resource optimization. Finally, the *Deployer* calls the *DockerUpdate* endpoint to update $L_{r,c}^{current}$ with the $L_{r,c}^{target}$ values provided in the analysis response.

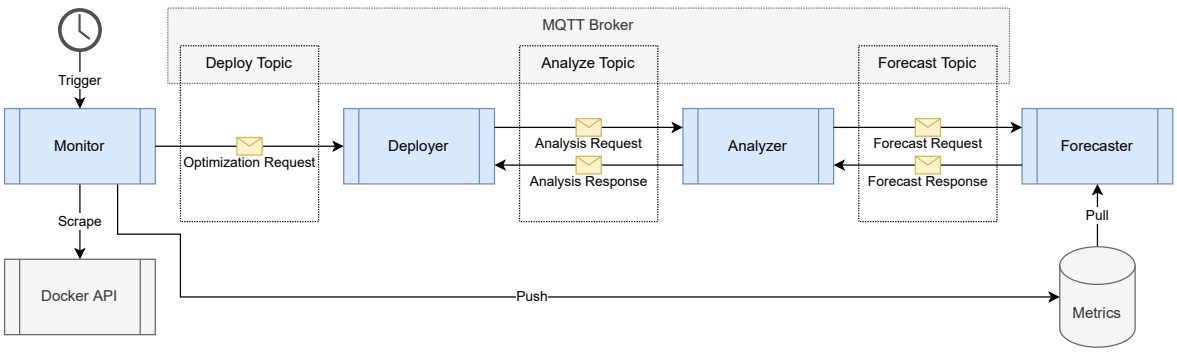

Figure 3.14. Optimization Workflow.

3.4.5.1. Optimization Analysis. Optimization analysis is the process of identifying containers that require resource optimization and determining a $L_{r,c}^{target}$ value satisfying their scaling needs. The aim is to downscale underutilized containers' limits, and upscale containers with increasing resource utilization approaching the limits. The *Analyzer* publishes $L_{r,c}^{target}$ given that optimization is necessary and enough resources are available on the system as explained in the algorithm in Figure 3.16.

The analysis starts with the calculation of $P_{r,S}^{avail}$ as the initial predicted availability of the system's resources. This calculation was explained previously in the algorithm in Figure 3.10. Then, the *Analyzer* evaluates the scaling needs of each container sequentially. After every approved optimization response, $P_{r,S}^{avail}$ is updated accordingly before the next container is evaluated. $P_{r,S}^{avail}$ is adjusted by the scaling amount of the optimized container. Each published response is unique to the container that is

scheduled for optimization and published individually on the *deploy* topic. In order to proceed with the optimization analysis, current resource limits on each container should be known. $L_{r,c}^{current}$ is requested for each active container from Docker API by calling the *DockerInspect* endpoint. Then, the target limits $L_{r,c}^{target}$ are calculated. Their calculation is explained in Section 3.4.5.2 and Section 3.4.5.3 for CPU and memory, respectively. The process of calculating the target limits differs for each resource, the reason being twofold:

- *OOMKilled Error*: When a container runs out of memory, it is killed by the Docker engine. Therefore, while downscaling memory, a more conservative approach is employed.
- *CPU Throttling*: Unlike memory, an attempt to surpass the defined CPU limits will not kill the container. Instead, it will result in throttling. Thus, the predicted throttling of a container, $P_c^{throttle}$, is taken into account to optimize the output further.

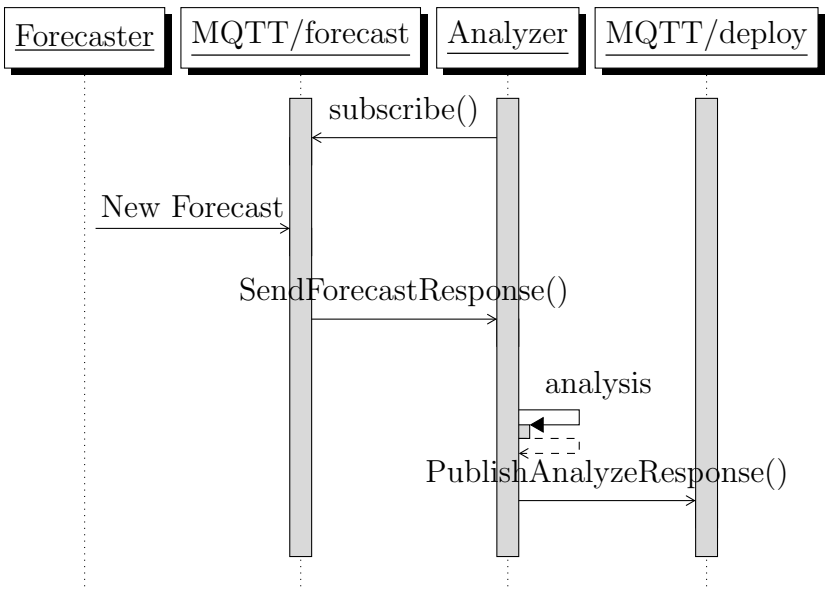

Figure 3.15. Analysis Response Sequence Diagram.

The analysis in the algorithm in Figure 3.16 can continue once $L_{r,c}^{target}$ for each active container is calculated. If there is a difference, $\Delta L_{r,c}$, between $L_{r,c}^{target}$ and $L_{r,c}^{current}$, the *Analyzer* can make a decision to either downscale or upscale. If $\Delta L_{r,c}$ value equals 0, no optimization is necessary. Positive and negative values for delta represent a need

for upscaling and downscaling, respectively. Upscaling demands are rejected if $\Delta L_{r,c}$ exceeds $P_{r,S}^{avail}$ meaning that the system will not be able to satisfy the upscaling demand. Otherwise, the *Analyzer* publishes an approval to the *deploy* topic to be fulfilled by the *Deployer*.

Input: C, $\mathbf{P}_{r,S}^{avail}$, $\mathbf{P}_{r,c}^{util}$ where r ∈ R, c ∈ C
  for $c \in C$ **do**
    $\mathbf{L}_{R,c}^{current} := dockerInspect(c)$
    $\mathbf{L}_{R,c}^{target} := calculateTargetLimits(R, c, \mathbf{L}_{R,c}^{current})$
  end for
  for $c \in C$ **do**
    $deployFlag :=$ **True**
    for $r \in R$ **do**
      $\Delta\mathbf{L}^{r,c} := \mathbf{L}_{r,c}^{target} - \mathbf{L}_{r,c}^{current}$
      if $(\Delta\mathbf{L}^{r,c} \neq 0)$ and $(\mathbf{P}_{r,S}^{avail} > \Delta\mathbf{L}^{r,c})$ **then**
        continue
      else
        $deployFlag :=$ **False** ▷ Reject
      end if
    end for
    if $deployFlag$ **then**
      $publishOptimizationAnalysisResponse(c, \mathbf{L}_{R,c}^{target})$ ▷ Approve
      $\mathbf{P}_{R,S}^{avail} := \mathbf{P}_{R,S}^{avail} - \Delta\mathbf{L}^{R,c}$
    end if
  end for

Figure 3.16. Optimization Availability Analysis.

3.4.5.2. Memory Limit Optimization Analysis. The optimization analysis for memory limit determines the $L_{mem,c}^{target}$ as shown in the algorithm in Figure 3.17. Afterwards, the $L_{mem,c}^{target}$ value is returned to the algorithm in Figure 3.16 to continue with the optimization analysis. Optimizing memory limits is a task with little room for error,

especially for downscaling. If $C^{util}_{mem,c}$ tries to pass the memory limit defined by $L^{current}_{mem,c}$, the container will be killed, i.e., *OOMKilled*. Thus, if the framework anticipates that container's memory utilization will pass $L^{current}_{mem,c}$, the container should be scaled up to $L^{target}_{mem,c}$ which is higher than $L^{current}_{mem,c}$. A buffer ratio of L^{buffer}_{mem} is applied in this comparison to always give memory utilization a buffer between $P^{util}_{mem,c}$ and $L^{current}_{mem,c}$ until the next optimization cycle. The buffer is configured as 10% margin in our framework. This ensures that the framework will still be inclined to scale up the container as early as possible when $P^{util}_{mem,c}$ is approaching $L^{current}_{mem,c}$. If the prediction is less the memory utilization will go down, the framework will lower $L^{current}_{mem,c}$ by $L^{scale}_{mem,down}$ to scale down the container. Both $L^{scale}_{mem,down}$ and $L^{scale}_{mem,up}$ values are constant values and can be configured while setting up our framework.

Input: C, $\mathbf{L}^{current}_{r,c}$, $\mathbf{P}^{util}_{r,c}$ where r ∈ R, c ∈ C
Output: $\mathbf{L}^{target}_{mem,c}$
for $c \in C$ **do**
  $prediction := \max(\mathbf{P}^{util}_{mem,c})$
  if $prediction > \mathbf{L}^{current}_{mem,c} / \mathbf{L}^{buffer}_{mem}$ **then**
    $\mathbf{L}^{target}_{mem,c} := \mathbf{L}^{current}_{mem,c} + \mathbf{L}^{scale}_{mem,up}$ ▷ Scale up
  else
    $\mathbf{L}^{target}_{mem,c} := \mathbf{L}^{current}_{mem,c} - \mathbf{L}^{scale}_{mem,down}$ ▷ Scale down
    if $\mathbf{L}^{target}_{mem,c} < prediction * \mathbf{L}^{buffer}_{mem}$ **then**
      return $\mathbf{L}^{current}_{mem,c}$ ▷ Reject Scaling
    end if
  end if
  if $\mathbf{L}^{target}_{mem,c}$ **between** $\mathbf{L}^{min}_{mem}$ **and** $\mathbf{L}^{max}_{mem}$ **then**
    return $\mathbf{L}^{target}_{mem,c}$ ▷ Approve scaling
  end if
  return $\mathbf{L}^{current}_{mem,c}$ ▷ Reject scaling
end for

Figure 3.17. Memory Optimization Analysis.

After the $L_{mem,c}^{target}$ is set, in case of down scaling, the new limit is once again compared with the max prediction amount to make sure that the the container is not scaled below the allowed buffer room. Otherwise, $L_{mem,c}^{current}$ is returned, and memory is not scaled down any further.

Finally, for either type of scaling, it should be confirmed that $L_{mem,c}^{target}$ is between some default minimum and maximum resource values: L_{mem}^{min} and L_{mem}^{max}. These limits define a general lower and upper limit no matter the container's workload and hardcoded in the framework. The upper limit ensures that a single container cannot allocate a very high share of a system's total resources, S_{mem}^{total}. The lower limit protects the container from scaling down indefinitely. The resulting $L_{mem,c}^{target}$ value is returned to the algorithm in Figure 3.17 for further analysis.

3.4.5.3. CPU Limit Optimization Analysis. The calibration for $L_{cpu,c}^{target}$ depends on $P_{cpu,c}^{util}$, and $P_{c}^{throttle}$ and follows the steps in the algorithm in Figure 3.18. As previously discussed, if $C_{cpu,c}^{util}$ reaches $L_{cpu,c}^{current}$ the container is not killed, unlike an *OOMKilled* response in the memory case. However, this does not mean that the container runs in an optimal condition. In our framework where application metrics such as response time are not available, it is not possible to deduce any performance degradation from memory and CPU utilization alone. However, there is a reliable system metric which is $C_{c}^{throttle}$. A high amount of $C_{c}^{throttle}$ will jeopardize the application performance. Thus, extra precautions are taken while down scaling CPU for containers experiencing throttling.

First, singular prediction values are taken from $P_{cpu,}^{util}$ and $P_{c}^{throttle}$ by getting their maximums. Then, if the CPU utilization prediction is greater than $L_{cpu,c}^{current}$, $L_{cpu,c}^{target}$ is calculated by increasing $L_{cpu,c}^{current}$ with a fixed amount of $L_{cpu,up}^{scale}$. In the opposite case where $P_{cpu,c}^{util}$ is greater than $L_{cpu,c}^{current}$, the downscale target limit is calculated similarly; $L_{r,c}^{current}$ is lowered by the amount of $L_{r,down}^{scale}$ to get $L_{cpu,c}^{target}$.

There are two extra controls to prevent excessive throttling in CPU optimization analysis. The predicted throttling amount is compared against a maximum allowed throttling limit value, $L^{throttle}$, which is a constant defined in our framework. If maximum predicted throttling is greater than $L^{throttle}$, the allowed throttling threshold, the $L^{target}_{cpu,c}$ is recalculated. $L^{scale}_{cpu,up}$ is lowered by adjusting it with the predicted throttling value. Then the $L^{target}_{cpu,c}$ is recalculated with the lowered $L^{scale}_{cpu,up}$ for a minor scale up. The second control is before setting the final scaled down $L^{target}_{cpu,c}$. While scaling down, there exists a possibly that with the $L^{target}_{cpu,c}$, the container can start to experience throttling. In order to prevent sudden throttling, instead of using a constant $L^{scale}_{cpu,down}$, $L^{target}_{cpu,c}$ is calculated with the maximum predicted utilization. $L^{target}_{cpu,c}$ is scaled by increasing the maximum predicted utilization by the L^{buffer}_{cpu} ratio. Similar to memory optimization, L^{buffer}_{cpu} is configured as a 10% margin in our framework. This buffer ensures that the application is given enough room for unexpected load spikes until the next optimization without causing unnecessary throttling. After this adjustment, it is possible that $L^{target}_{cpu,c}$ can now be above $L^{current}_{cpu,c}$ which would scale up the container instead. In that case, any scaling operations are overturned by returning $L^{current}_{r,c}$ as the throttling was already below $L^{throttle}$.

A final control is made to check if $L^{target}_{cpu,c}$ is within the allowed boundaries of L^{min}_{cpu} and L^{max}_{cpu} similar to the memory optimization. If the $L^{target}_{cpu,c}$ satisfies the final condition, it is returned back to the algorithm in Figure 3.16 to continue with the rest of the optimization analysis.

3.5. Cluster Deployment

The event-driven architecture makes it possible to deploy our framework on multiple edge devices as a cluster in the same network. This deployment strategy supports load balancing across the connected hosts and decentralized orchestration of deployments. The setup depends on MQTT bridges. In a bridge setup, a broker can automatically broadcast monitoring and deployment messages to all hosts in the network. A setup with three connected edge devices is illustrated in Figure 3.19.

Input: C, $\mathbf{P}_{c}^{throttle}$, $\mathbf{L}_{r,c}^{current}$, $\mathbf{P}_{r,c}^{util}$ where r ∈ R, c ∈ C

Output: $\mathbf{L}_{cpu,c}^{target}$

for $c \in C$ **do**

$prediction := \max(\mathbf{P}_{cpu,c}^{util})$

$throttle\_prediction := \max(\mathbf{P}_{c}^{throttle})$

if $prediction > \mathbf{L}_{cpu,c}^{current}$ **then**

$\mathbf{L}_{cpu,c}^{target} := \mathbf{L}_{cpu,c}^{current} + \mathbf{L}_{cpu,up}^{scale}$ ▷ Scale up

else

if $throttle\_prediction > \mathbf{L}^{throttle}$ **then**

$adj\_scale := (\mathbf{L}_{cpu,up}^{scale} \times throttle\_prediction)/100$

$\mathbf{L}_{cpu,c}^{target} := \mathbf{L}_{cpu,c}^{current} + adj\_scale$ ▷ Minor scale up

else

$\mathbf{L}_{cpu,c}^{target} := \mathbf{L}_{cpu,c}^{current} - \mathbf{L}_{cpu,down}^{scale}$ ▷ Scale down

if $\mathbf{L}_{cpu,c}^{target} \leq prediction \times \mathbf{L}_{cpu}^{buffer}$ **then**

$\mathbf{L}_{cpu,c}^{target} := prediction \times \mathbf{L}_{cpu}^{buffer}$ ▷ Adjust

end if

if $\mathbf{L}_{cpu,c}^{target} > \mathbf{L}_{cpu,c}^{current}$ **then**

return $\mathbf{L}_{cpu,c}^{current}$ ▷ Reject scaling

end if

end if

end if

if $\mathbf{L}_{cpu,c}^{target}$ **between** $\mathbf{L}_{cpu}^{min}$ **and** $\mathbf{L}_{cpu}^{max}$ **then**

return $\mathbf{L}_{cpu,c}^{target}$ ▷ Approve scaling

end if

return $\mathbf{L}_{cpu,c}^{current}$ ▷ Reject scaling

end for

Figure 3.18. CPU Optimization Analysis.

The bridging feature of MQTT connects two brokers in such a way that each event automatically shares messages between them, as illustrated in Figure 3.20. This

setup can be configured to share all published messages in selected topics with another MQTT broker. There are no limits on the number of bridges that can be configured on a broker.

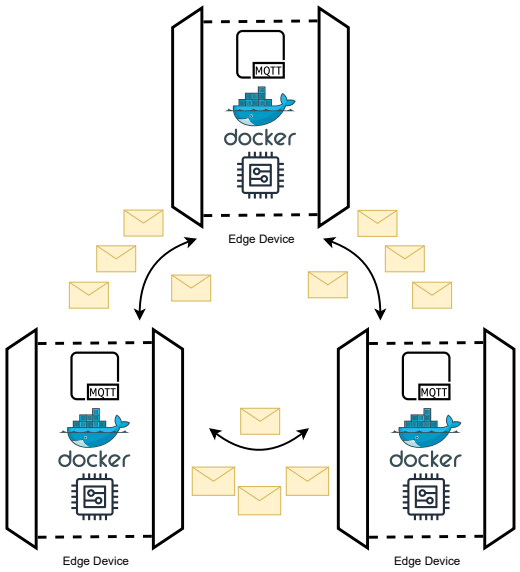

Figure 3.19. Cluster Deployment with Three Devices.

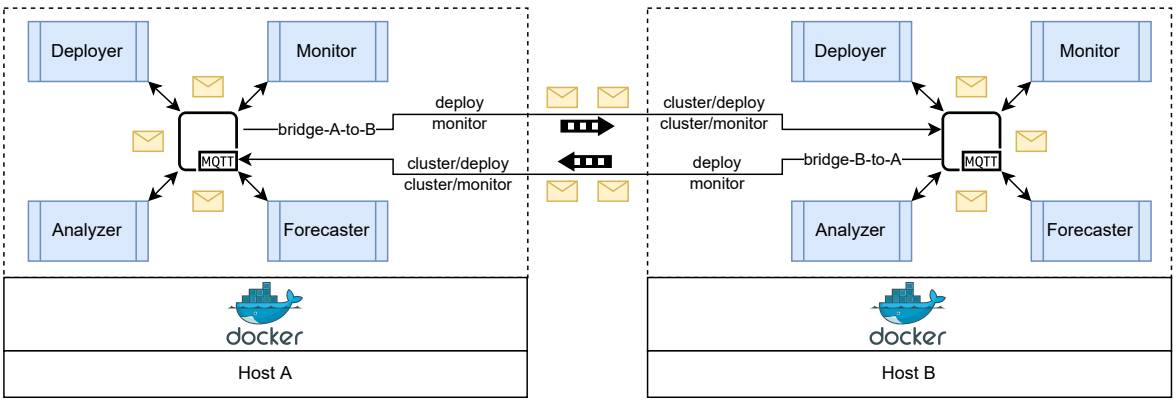

Figure 3.20. Two MQTT Brokers Connected with Bridges.

The cluster deployment requires some extra configuration steps on the MQTT brokers. In order to set up a bridge with another broker, the MQTT broker should know the IP address of the target host. In this thesis, we manually provide each device's IP address to each other broker. However, it is also possible to set up an MQTT device discovery mechanism to automatically find other brokers on the network [77]. The

second configuration is about the shared topics. It is possible to add prefixes to topics in a bridge configuration. All shared topics of our framework are prefixed with the *cluster* keyword. This prefix allows each component to identify whether an event is generated internally or coming from an external host from the topic name.

In the cluster setup, only the *monitor* and *deploy* topics are shared between devices through *cluster/monitor* and *cluster/deploy* topics respectively as shown in Figure 3.20. The *Deployer* on each device is subscribed to both *monitor* and *cluster/monitor* topics. The *Deployer* keeps an hash map with IP addresses. For each IP address, a resource pair for CPU and memory resource availability is kept. The hash map is generated with the information received from both topics. When a monitoring message is received from a device for the first time, an entry with that device's IP address is created in the hash map. The available resource information is also extracted from the same message and stored alongside the IP address. With every new monitoring result, the resource availability information is updated. The *Deployer* uses this hash map to stay up to date about the available resources of all devices in the network.

The *Deployer* is also subscribed to the *deploy* and *cluster/deploy* topics. On these topics, *Deployer* receives deployment requests made on all devices. When a new request is received, each *Deployer* selects the device with the highest amount of available resources from the hash map. The selected device is the most suitable candidate for the new deployment. Then, only the device that determined itself as the best candidate continues with the deployment by publishing a deployment analysis request. The deployment follows the same steps as the regular deployment workflow. If the available resources, S_r^{avail}, of multiple devices are the same, each device marks the device with the smallest IP address for deployment. Then, only the device that has marked itself proceeds with the deployment. As the responsibility of accepting the deployment is given to each device, it is assumed that they will act fairly. A device could potentially publish malicious messages on the system. However, the framework is designed for private networks, and all devices are configured and connected by the same user. Therefore, the environment in which the framework is deployed is trusted.

4. Experiments and Results

4.1. Experimental Setup

All experiments are carried out on a Raspberry Pi 4 Model B [78]. The device has an 8GB LPDDR4-3200 SDRAM and Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit processing unit. Using systemd, a system and service manager for Linux, a slice unit is created with 1 CPU and 1GB memory with swap usage disabled. Docker daemon's cgroup parent is assigned to this slice to limit the maximum allowed resource usage [79].

Experiments are based on three collected metrics: memory usage, CPU utilization, and CPU throttling. These values are calculated from the usage and limit values stored in the host system's control group (cgroup) files [80]. The memory usage is directly acquired from *memory.usage_in_bytes*. The memory limit is stored in *memory.stat* instead, as the *hierarchical_memory_limit*. CPU utilization is not as straightforward. How the limit value was set was explained in Section 3.4.2 through the Docker API. The CPU quota value that was parameterized by Docker, *cpu-quota*, can be acquired from *cpu.cfs_quota_us*. The CPU time value in nanoseconds found in *cpuacct.usage* is used to calculate CPU utilization with

$$\begin{aligned} cpuacct.usage_{diff} &= cpuacct.usage - cpuacct.usage_{prev} \\ time_{diff} &= time\,ns - time_{prev}\,ns \\ C^{util} &= \frac{cpuacct.usage}{time_{prev}} \end{aligned} \tag{4.1}$$

by making two observations. This formula returns the time the CPU was active in a given timeframe matching the same granularity of nanoseconds. The activity is limited to the cgroup, which corresponds to a single container. The calculation of $C_i^{throttle}$ is previously explained in Section 3.4.4.1 and given as Equation (3.2).

In order to create a controlled environment for the tests, two sets of containers are prepared each set corresponding to a resource type R_i. The requirement for these containers is that their CPU and memory utilization could be controlled through a set of variables to create different workload patterns. For memory, a container's memory usage should be adjustable which is achieved by setting blocks of memory or freeing them up. For CPU, mocking the CPU utilization proved to be rather challenging. In order to create a controllable CPU load, mini units of maximum 1 second sorting tasks are created. When put together back to back, these tasks are able to replicate a certain percentage of CPU utilization. As shown in Figure 4.1, these 1 second long tasks perform a short sorting task using 100% of the CPU and then sleep for the remaining time. In the examples given in Figure 4.2 first a 30% CPU utilization is achieved by running the sorting task for 0.3 seconds and sleeping for the rest of the time. Afterward, the task is set to run for 0.5 seconds to achieve an overall 50% CPU utilization.

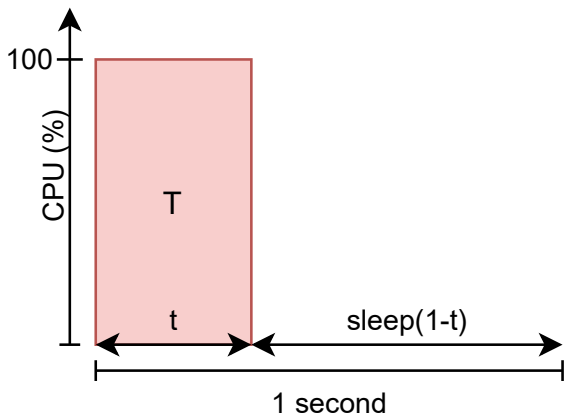

Figure 4.1. A Single Unit of Task.

However, there is no guarantee that a particular sorting task will complete in the expected 1 second. This is an indicator of CPU throttling caused by the limited quota assigned to the container. If there is heavy throttling, the individual tasks will exceed the 1 second window, causing the following tasks to be delayed.

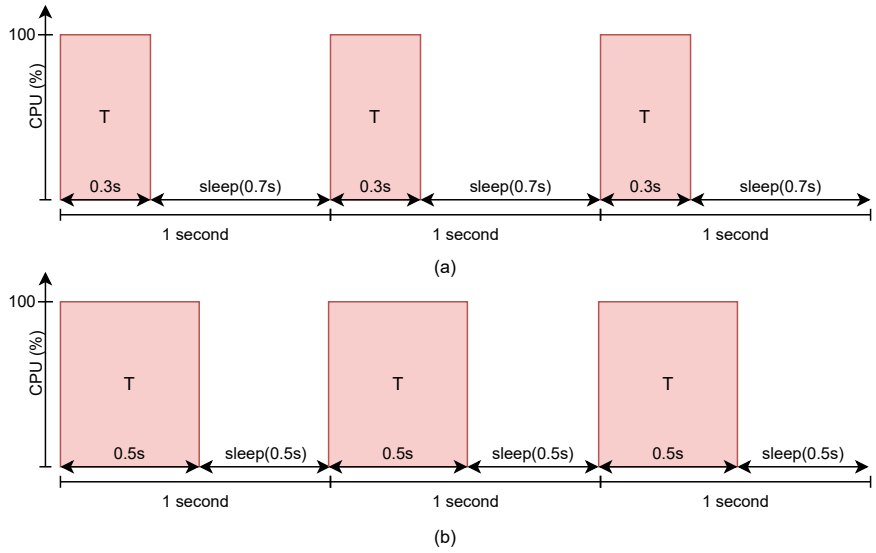

Figure 4.2. A Sequence of Unit Tasks with Different CPU Utilization.
(a) A sequence of unit tasks with 30% utilization of a single core.
(b) A sequence of unit tasks with 50% utilization of a single core.

To better understand the delay caused by throttling, 100 sorting operations each with a duration of approximately 0.6 seconds are performed in Figure 4.3 sequentially. These operations are normal sorting operations with no sleeps in between, different from the tasks explained above. Given that the whole CPU is dedicated to the sorting operation, it completes in the expected 60 seconds. The second row of graphs in Figure 4.3 also shows that the same operation takes double the amount of time if half the resources are given. It should also be noted that the process is completely throttled for the whole duration in 50% CPU limit.

Now that it is clear how the CPU limits and throttling works, the unit tasks can be explored further. In the first row of Figure 4.4, 100 unit tasks are performed sequentially where the sorting is performed in each unit which takes 0.6s, leaving the CPU idle in the remaining 0.4s. This results in 60% CPU utilization, and the task completes in 100 seconds. In the second and third sets of graphs in Figure 4.4, the container's CPU resource is constrained by 75% and 60%, respectively. Since the CPU

utilization was found to be 60%, a 75% limit does not affect the completion time of the process as it once again completes in 100 seconds. However, for the duration of the sorting operation, which uses all available CPUs typically, there is constant throttling, resulting in 60% throttling throughout the process. If the limit is set to 50%, then there are not enough resources for the sorting operation in a unit task to be completed in 1 second. Therefore, it takes 20 seconds longer to complete the same number of tasks as it is being throttled constantly.

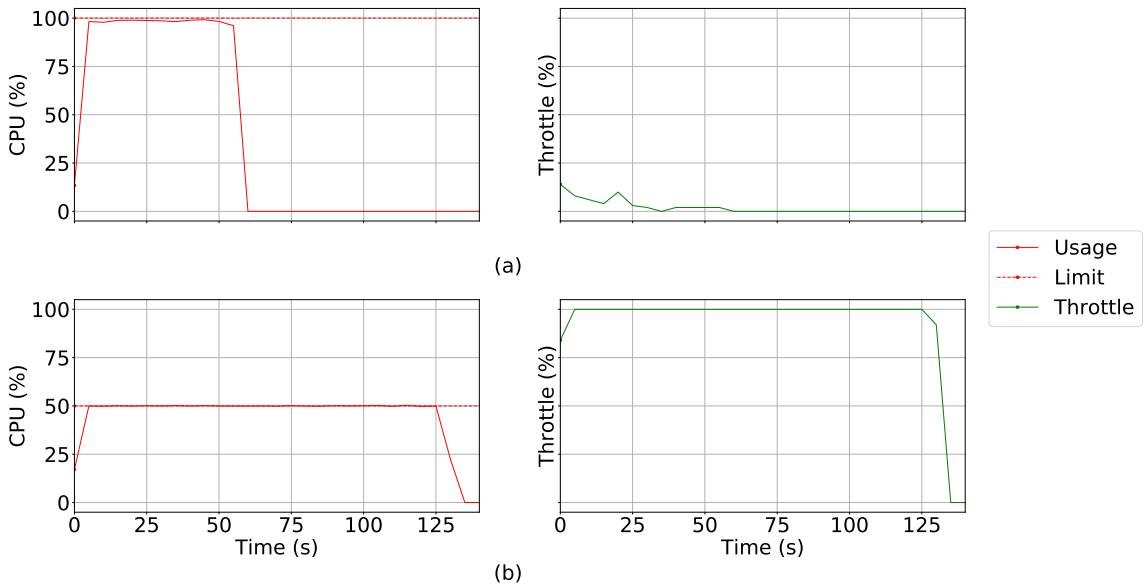

Figure 4.3. Sequentially Executed 100 Sorting Operations.
(a) Unlimited CPU utilization with a single core.
(b) CPU utilization limited to 50% of a single core.

4.2. Workload Patterns

Five different workload patterns are created to replicate workloads of various kinds of IoT tasks:

- Slowly rising/falling workload pattern
- Drastically changing workload pattern

- On-off workload pattern
- Gently shaking workload pattern
- Real-world workload pattern

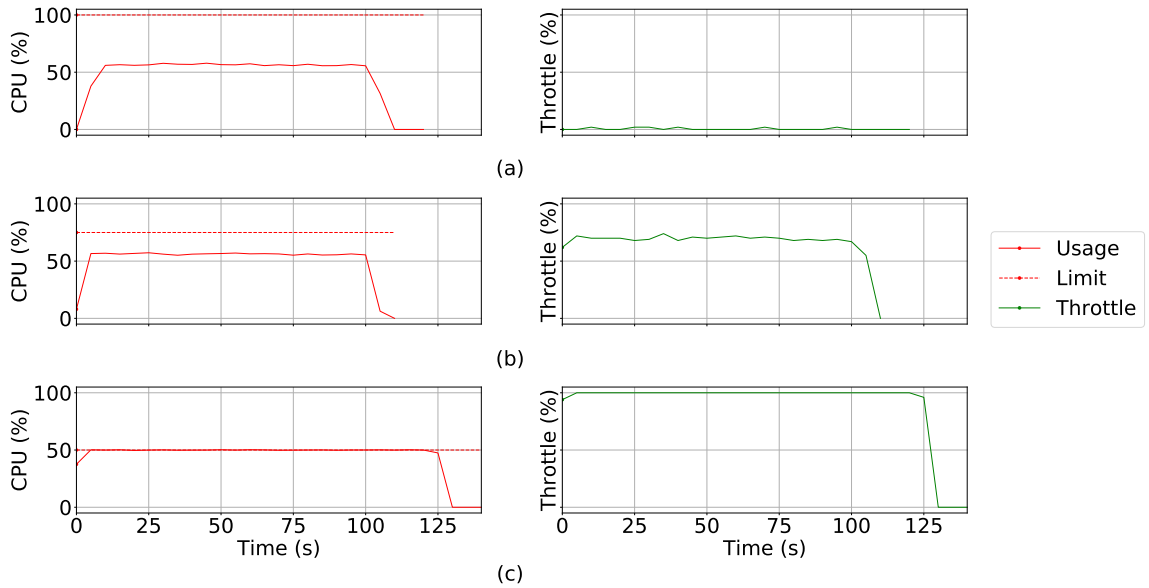

Figure 4.4. Sequences of Unit Tasks Executed With Various CPU Limits.
(a) Unlimited CPU utilization with a single core.
(b) CPU utilization limited to 75% of a single core.
(c) CPU utilization limited to 50% of a single core.

These workload patterns were proposed by [81] to categorize IoT applications based on the number of service requests they have made over time. The same principle is followed in this experiment by measuring the memory and CPU usage statistics instead. These system-level metrics offer the chance to uniformly assess all IoT tasks without limiting the IoT applications to microservices which is the case for any system relying on the application level metrics like service requests. The resource demands of all workloads are adapted to portray different IoT applications capable of running in a constrained environment.

Each workload pattern runs in a separate container. A respective image is prepared for each workload pattern for each resource individually, amounting to a total of ten container applications. The base and request limit specifications can be found in the Table 4.1. Both memory and CPU limits are defined with resource units adopted by Kubernetes [82]. Memory usage and limits are expressed with fixed point integers for simplicity and are in the power of 10 according to International Electrotechnical Commission (IEC) standards [83]. These units are represented with symbols such as KB and MB. CPU limits and utilization are expressed as CPU units, where 1 CPU is equal to a CPU core. Therefore, fractions of a CPU can be assigned to an application with the *milli* suffix, m. In this representation, 50% of a single CPU core is represented as 500m. A system with two cores is represented with 2000m, following the same standards.

Table 4.1. Workload Resource and Limit Specifications.

Workload	**Application Name**	**Resource**	**Base Limit**	**Request Limit**
Slowly rising/falling	Memory 1	Memory	100MB	150MB
Drastically changing	Memory 2	Memory	100MB	150MB
On-off	Memory 3	Memory	100MB	150MB
Gently shaking	Memory 4	Memory	100MB	150MB
Real-world	Memory 5	Memory	100MB	150MB
Slowly rising/falling	CPU 1	CPU	100m	300m
Drastically changing	CPU 2	CPU	100m	300m
On-off	CPU 3	CPU	100m	300m
Gently shaking	CPU 4	CPU	100m	300m
Real-world	CPU 5	CPU	100m	300m

4.2.1. Slowly Rising/Falling Workload Pattern

This workload represents the resource usage of an IoT application that is either increasing or decreasing slowly. The pattern has a period of 600 seconds as shown in

Figure 4.5. The memory workload starts from a minimum value of 0MB and increases uniformly by 3MB every 10 seconds for the first half of the period up to 90MB. When the vertex is reached, the resource usage goes on a decline with the same ratio until the full period is complete. The CPU workload, on the other hand, follows the same pattern with minimal increments or decrements between 50m and 150m CPU utilization. This pattern replicates tasks where there is a continuous yet changing usage of resources at a slow pace. The task can be following slowly changing environmental conditions especially those following a cyclic pattern.

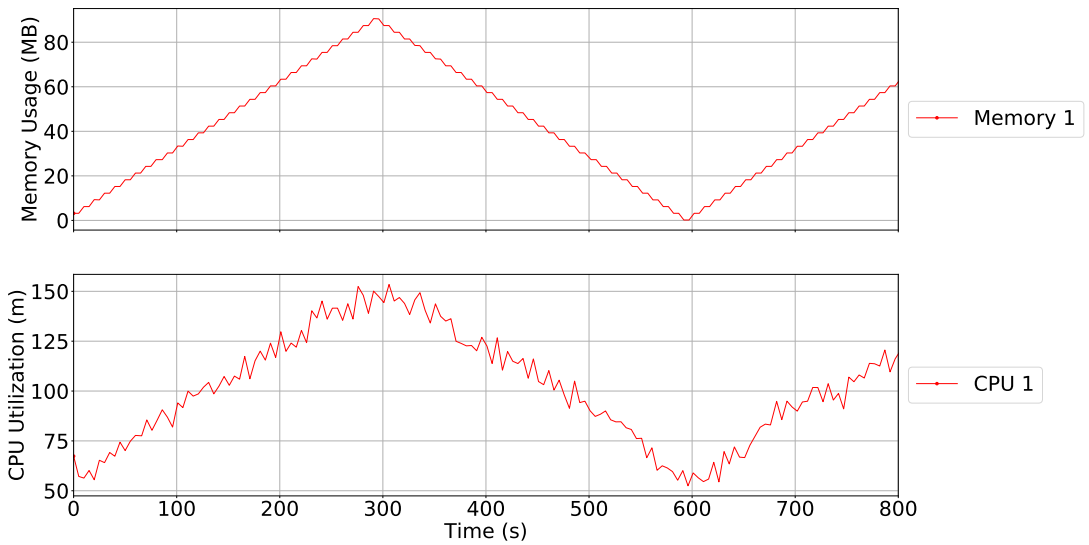

Figure 4.5. Slowly Rising/Falling CPU and Memory Workloads.

4.2.2. Drastically Changing Workload Pattern

This workload pattern follows the extremes. Yet, it stays constant once one of the extremes is reached. The resource usage switches between predefined min and max values every 300 seconds corresponding to half of its period as shown in Figure 4.6. The utilization of the CPU workload is either 50m or 150m the memory can have the values of 10MB or 90MB, similar to the first workload presented. A drastically changing pattern represents tasks executed in mini batches. The behavior changes instantly once the new mini batch is started. A real world example can be a system

waiting for a trigger which can switch the behavior in an instant like a routine job waiting for a signal to process a set of data.

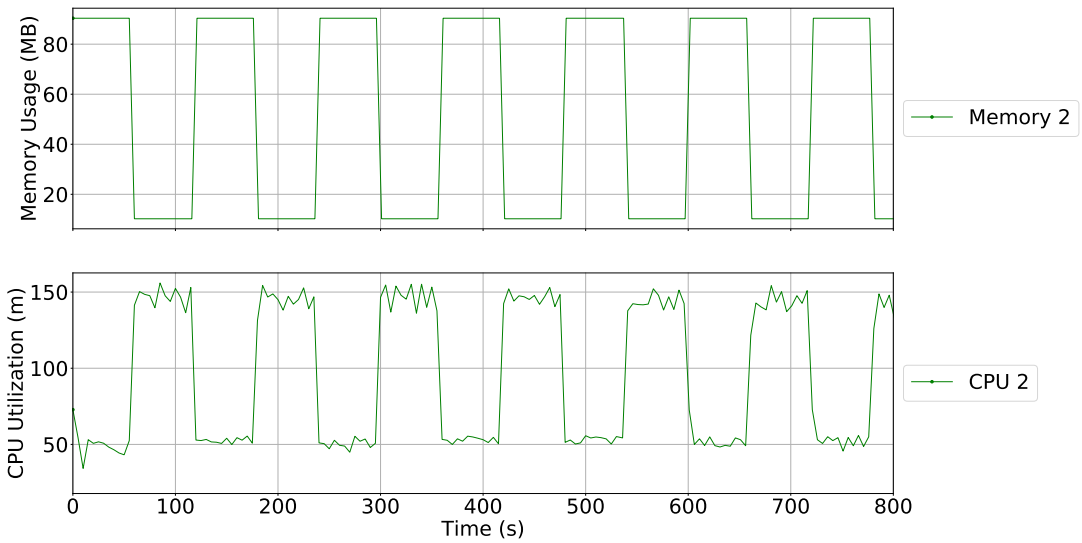

Figure 4.6. Drastically Changing CPU and Memory Workloads.

4.2.3. On-Off Workload Pattern

This is the first pattern with more than two characteristics in a single period. This pattern represents a workload that handles a set number of tasks with different resource requirements within a single period as illustrated in Figure 4.7. In one of these phases resource utilization is significantly lower than the rest. The possible CPU workloads are 50m, 100m or 150m whereas the memory workload can be 30MB, 60MB or 90MB. A device that sleeps briefly and turns back on or some batch requests of varying sizes are being processed is an example of such a pattern.

4.2.4. Gently Shaking Workload Pattern

This workload has a more conservative pattern compared to the first three. The amplitude is smaller, resulting in a more consistent workload changing around a fixed value as demonstrated in Figure 4.8. The CPU utilization changes around a center

value of 100m and the memory usage changes around 45MB. It also has the same increasing/decreasing characteristic of the first workload pattern but in a more confined boundary changing more rapidly. This pattern results in the CPU workload to differentiate from the center by ±50m and the memory workload by ±35MB. This pattern is more constrained than the previous patterns and can be observed on routine IoT tasks such as monitoring where the behavior stays relatively the same during the lifetime of the application.

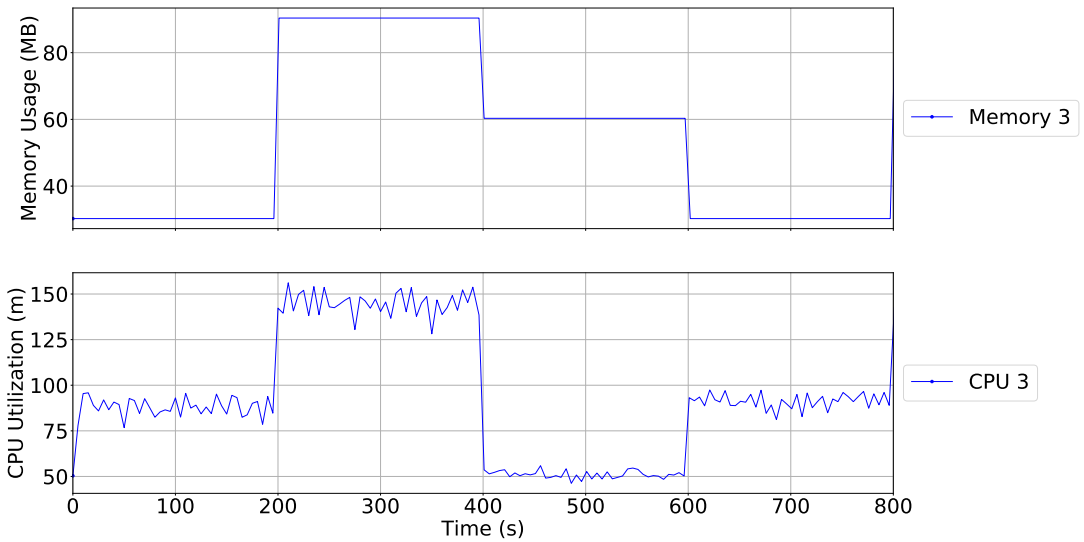

Figure 4.7. On-Off CPU and Memory Workloads.

4.2.5. Real-World Workload Pattern

This pattern represents tasks with a more erratic pace. The resource usage fluctuates between a predefined range in an irregular pattern as shown in Figure 4.9. These rangers are (50m, 150m) for CPU utilization and (10MB, 90MB) for memory usage. The data points are generated randomly between these values to achieve an irregular pattern. This pattern is used to represent previously uncategorized tasks. An example could be an extremely dynamic environment for this pattern.

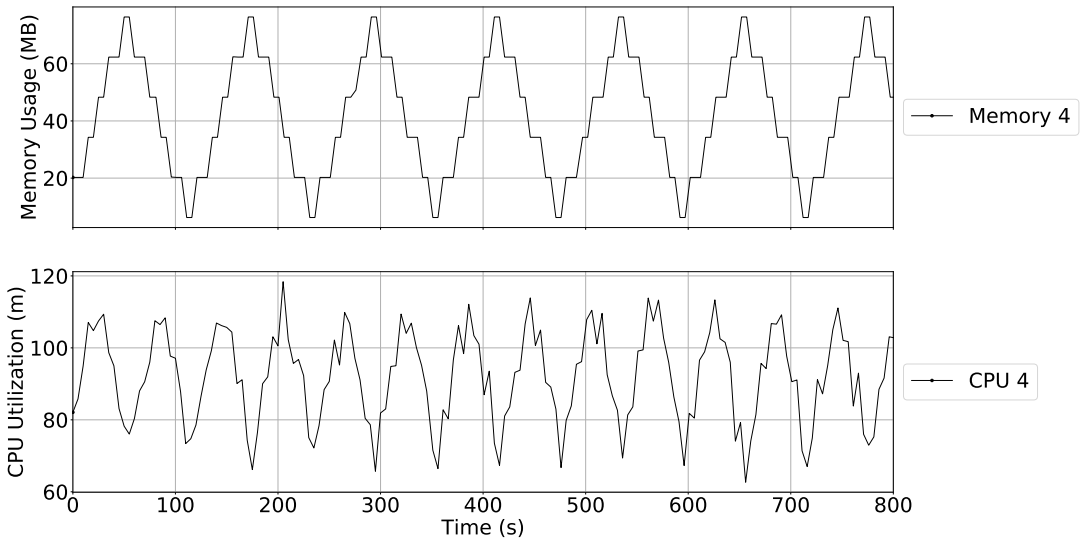

Figure 4.8. Gently Shaking CPU and Memory Workloads.

In addition to these patterns with generated workloads, a real-time video streaming application is separately deployed on the framework to demonstrate the behavior with an actual IoT application. The experiments carried out with this real-world example can be found in Section 4.4.

4.3. Experiments on the Co-Location of Workloads

The overall limits of the system are determined by the physical limitations of the system, the number of applications running on the host, and their resource consumption. An edge environment where multiple IoT devices and applications coexist will be replicated with the following experiments, and the orchestration framework's scheduling and resource allocation capabilities will be observed. In each experiment, a subset of containerized applications listed in Table 4.1 are deployed on the same device to demonstrate four different scenarios of co-located workloads.

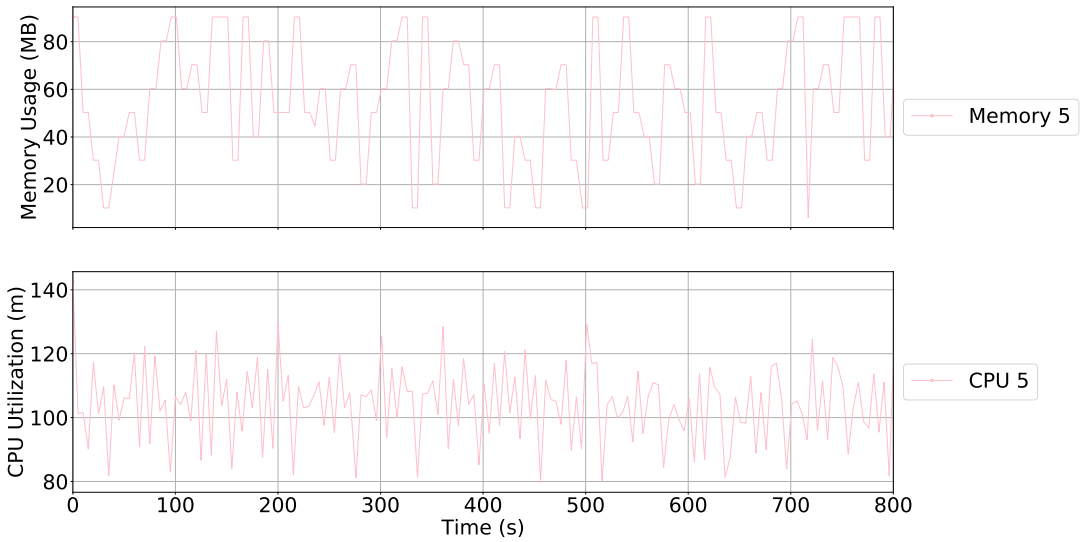

Figure 4.9. Real-world CPU and Memory Workloads.

4.3.1. All Workloads with Ample Resource Specifications

In the first experiment, all five workloads will be started simultaneously with their resource limit specification set generously. In the memory workload, both $L^{base}_{mem,j}$ and $L^{request}_{mem,j}$ are set higher than their actual usage for both workloads. $L^{request}_{i,j}$ amounts to around 50% more than the maximum memory usage, and $L^{base}_{mem,j}$ only 20% more. For the CPU utilization, $L^{request}_{cpu,j}$ is given as double the utilization rate expected to be seen from the container. $L^{base}_{i,j}$ is set to half of the actual value to observe. The subsequent requests after the $L^{request}_{cpu,j}$ cannot be assigned because of the limited resource of the system. As indicated by Figure 4.10, all five memory workloads have started simultaneously with more than enough resources necessary for them to complete their actual work. Suppose the limits set by the developer can replicate the customer's experience this closely. In that case, the resource limits converge to the actual usage in a few iterations of optimization steps every 5 minutes. The initial delay before the first optimization allows the framework to collect enough metrics to understand the characteristic of the workload and make utilization predictions, $P^{util}_{i,j}(t)$, for the upcoming optimization interval.

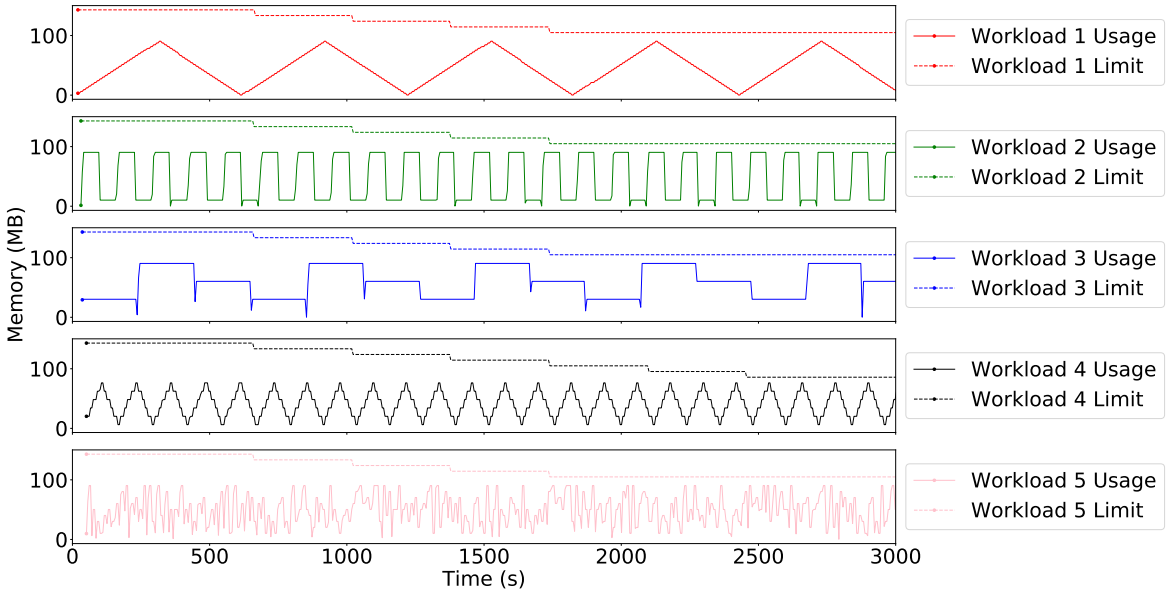

Figure 4.10. Co-Located Memory Workloads with Ample Resource Specifications.

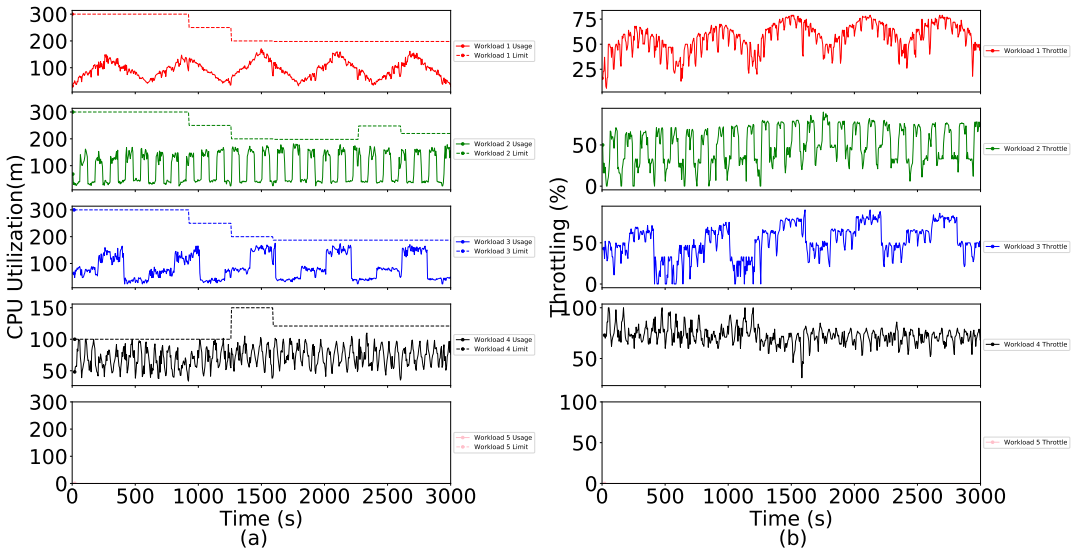

Figure 4.11. Co-Located CPU Workloads with Ample Resource Specifications.

(a) CPU utilization of each container.

(b) Throttling percentage of each container.

Figure 4.11 draws a scenario for the CPU utilization where the combined value of $L_{cpu,j}^{request}$ exceeds the system's resources, S_{cpu}^{total}. Thus, the first deployment of the 4th workload with $L_{cpu,j}^{request}$ is rejected and only deployed afterwards with $L_{cpu,j}^{base}$ in a second attempt by the framework. With the first four workloads running on the system, neither $L_{cpu,j}^{request}$ nor $L_{cpu,j}^{base}$ for the fifth workload can be satisfied. The deployment of the fifth application is rejected. The limits of the successfully deployed workloads are initially lowered drastically with a constant $L_{down,j}^{scale}$ value, which is later adjusted based on $P_{cpu,j}^{util}$ and $P_{j}^{throttle}$ resulting in more minor decrements. It should be noted that CPU workloads are generated using a sequence of CPU tasks as previously discussed in Section 4.1. The throttling percentages seen in Figure 4.11 are based on the individual CPU tasks as previously shown in Figure 4.4. After the fourth optimization attempt, all limits converge to an upper limit of 10% above the $P_{cpu,j}^{util}$. If the fifth workload were introduced to the system now, the framework would accommodate it as well.

4.3.2. All Workloads with Drastically Low Resource Specifications

The application developer does not have a complete picture of the actual environment the application is supposed to run in. They set the necessary $L_{mem,j}^{base}$, and $L_{mem,j}^{request}$ to satisfy the SLAs. If the system cannot honor these, the deployment is expected to be rejected. The application could potentially be deployed to an edge system with resources vastly different from the test setups. Also, the customer could be running multiple applications competing for resources.

The following experiment will create a scenario where these specifications are not just imperfectly tuned but drastically wrong to observe the framework's behavior in less than ideal situations. The simultaneous deployment of all five workloads will be continued, but their resource limit specifications are set to 10% of the actual maximum resource usage. Approved memory workload deployments are expected to recover after a few failed restarts, and CPU workloads are expected to reach the same limits as the previous experiment.

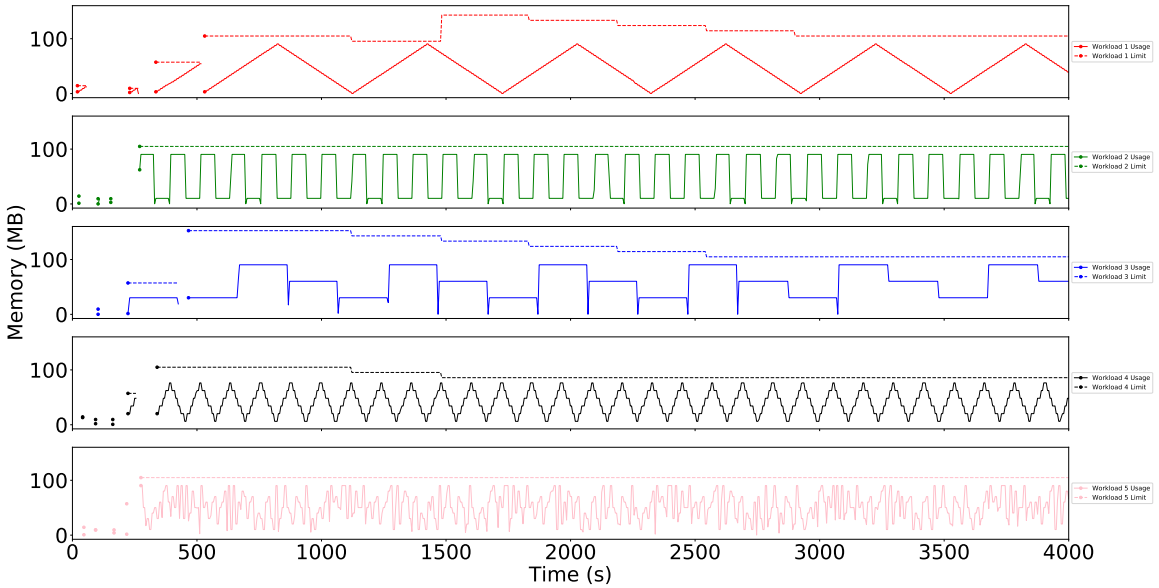

Figure 4.12. Co-Located Memory Workloads with Drastically Low Resource Specifications.

With only 10% of the actual resource available, all five memory workloads failed to start for the first couple of tries as shown in Figure 4.12. After the initial two deployment attempts with no prior knowledge of the application, the framework begins assigning rapidly increasing $L^{target}_{mem,j}$ values based on $L^{base}_{mem,j}$ incremented by $L^{scale}_{mem,j}$ by several retry counts, previously discussed in Section 3.4.4.2. Then, it can be seen that after the first four tries, the first and second workloads can be deployed as their resource usage is comparatively lower than the rest in the first one third of their period. The remaining three workloads keep getting killed as they immediately try to allocate more memory than allowed by $L^{current}_{mem,j}$. These frequent restarts cause their $L^{target}_{mem,j}$ to increase rapidly, allowing more memory allocation within the workload. However, the final $L^{target}_{mem,j}$ value reached by these bursts of up scaling does not represent the container's actual $C^{util}_{mem,j}$. Once the containers are deployed, their limits are lowered by $L^{scale}_{mem,j}$ allowing the system to find a tighter fit based on $P^{util}_{mem,S}$.

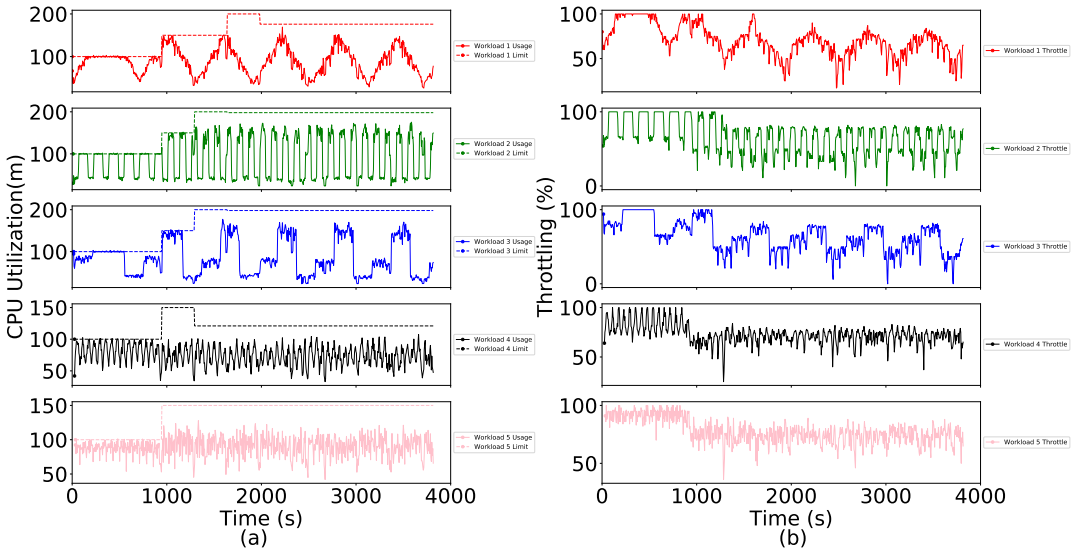

Figure 4.13. Co-Located CPU Workloads with Drastically Low Resource Specifications.
(a) CPU utilization of each container.
(b) Throttling percentage of each container.

The deployment behavior of the CPU workloads vastly differs from the memory workloads as shown in Figure 4.13. This is tied to the absence of an error similar to *OOMKilled*. Therefore, all five deployments are accepted simultaneously with their misconfigured $L_{cpu,j}^{request}$ values. These workloads typically exhibit a $C_{i,j}^{util}$ higher than this $L_{cpu,j}^{current}$. Consequently, they are all immediately throttled. The clearest indicator in Figure 4.13 is the second phase of the fourth workload, where the $C_{i,j}^{util}$ would typically amount to 600m, 60% of the CPU time. As shown in Figure 4.7, given enough CPU time, this workload moves onto the third phase at the 400 seconds mark. However, in this case, the same milestone is achieved at the 600 seconds mark, indicating that the application is heavily throttled. The framework optimized these $L_{cpu,j}^{current}$ responsible for the throttling after a couple of optimization cycles, eventually matching the limits similar to those in Figure 4.11.

4.3.3. Separate Deployment of Workload Pairs

In the first two experiments, deployments of applications were all requested simultaneously. And given enough optimization loops, they always converge to a limit bounding the resource usage of the applications. However, in an IoT edge system, both devices and applications are suspect to change. In this experiment, workloads 1 and 2 will be started simultaneously. Once no further optimization is necessary for the first two workloads, the second pair, workload 3 and 4, will be introduced to a stabilized system, and their effects will be observed.

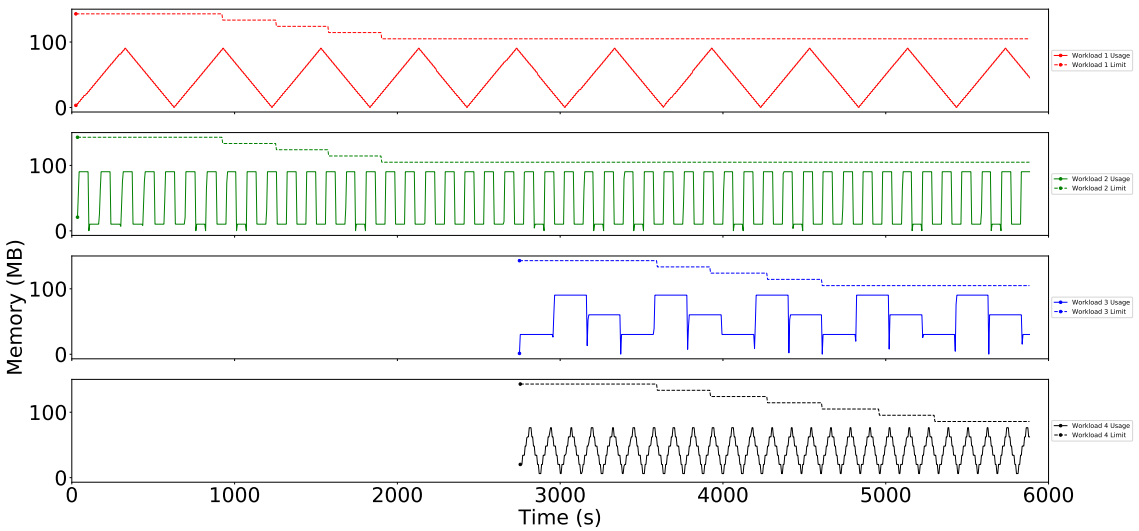

Figure 4.14. Separate Deployment of Memory Workload Pairs.

As shown in Figure 4.14 and Figure 4.15, the outcome is the same for CPU and memory workloads. The optimization steps adapt to the new workloads without interfering with the past deployments. The frameworks' optimization loops are independent of the arrival of the new deployment. Once enough data is collected for predictions to be performed on the newly introduced applications, optimization cycles will also start to include those. When they arrive at the system, the framework can reach the same limits for the application.

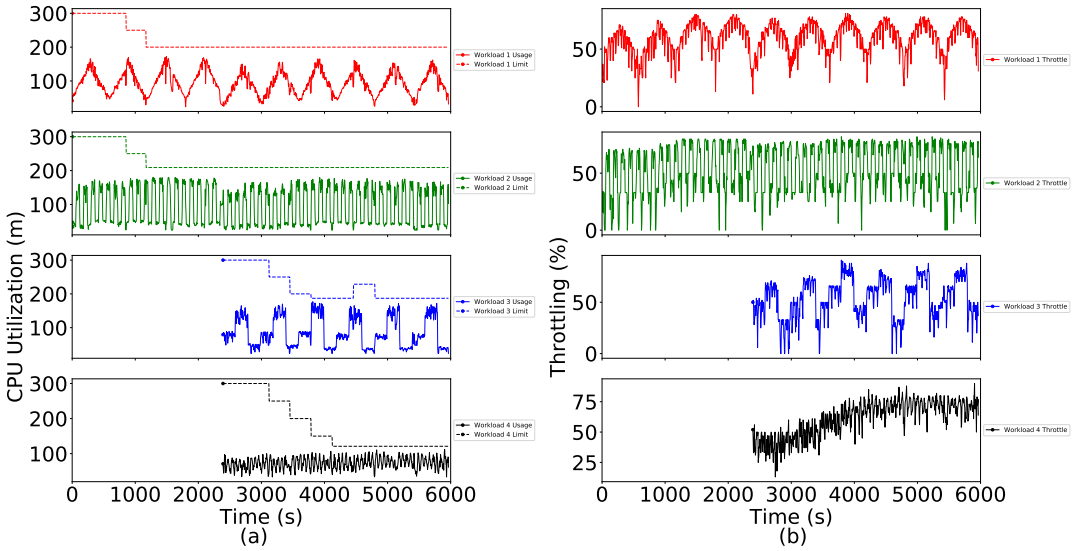

Figure 4.15. Separate Deployment of CPU Workload Pairs.
(a) CPU utilization of each container.
(b) Throttling percentage of each container.

4.3.4. Extreme Resource Constraints on the System

Most applications deployments were successfully accepted in all previous experiments as the system was not previously utilized. The final set of experiments studies the case where there are tighter resource constraints on the system due to pre-existing deployments. These deployments are simulated by a sixth workload which can constantly utilize significantly more resources than the rest. Workloads from 1 to 3 will be deployed on this further limited set up to replicate the system overloaded by applications.

A sixth workload with a memory limit, $L_{mem,j}^{current}$, of 600MB is initially started to limit S_{mem}^{avail} to 400MB as shown in the first graph of Figure 4.16. Then, deployment requests for the first three memory workloads are sent. First two workloads were able to be deployed with their $L_{mem,j}^{request}$. After these two deployments, the third workload cannot be deployed as there is not enough memory for $L_{mem,j}^{request}$ and $L_{mem,j}^{request}$ specifications of the third workload.

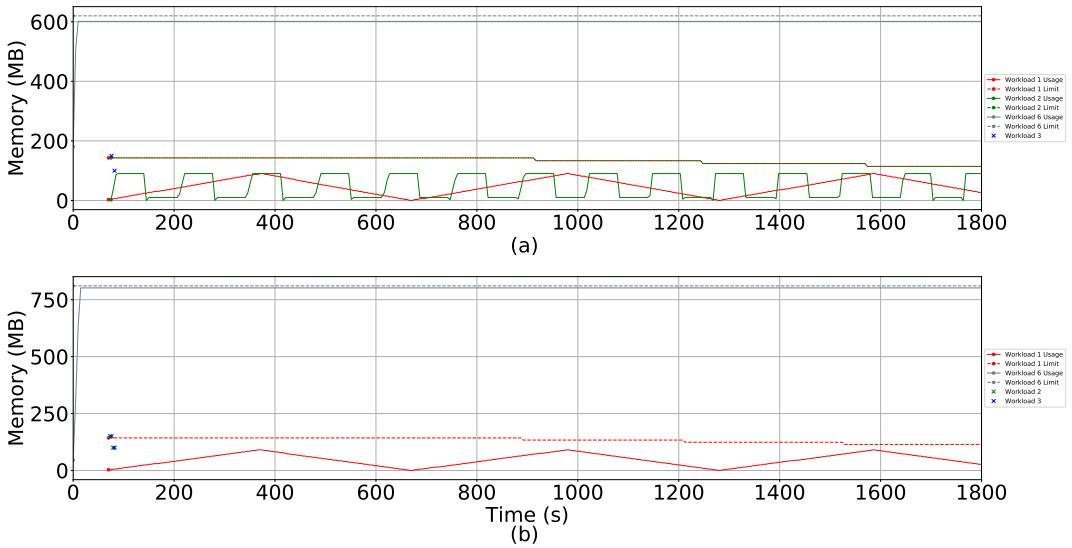

Figure 4.16. Memory Workloads Deployed Under Extreme Resource Constraints.
(a) Deployments on a System with 400MB Memory Availability.
(b) Deployments on a System with 200MB Memory Availability.

Then, another experiment is run with a similar configuration, but the memory limit of the extra workload, $L_{mem,j}^{request}$, is increased to 800MB to limit S_{mem}^{avail} to 200MB. Then deployment requests for the first three memory workloads are sent. As shown in the second graph of Figure 4.16. Once the first deployment is accepted, any subsequent deployments would require more memory than S_{mem}^{total}. These workloads are tried twice with $L_{mem,j}^{request}$ and $L_{mem,j}^{base}$ before rejected.

As shown in the first CPU graph of Figure 4.17(a), the extra workload is allocating 650m of the total available CPU, S_{cpu}^{avail}, of 1000m which limits S_{cpu}^{avail} to 350m. When deployment requests for the first three deployments are sent the first one is accepted. Since the first deployment is never rejected, $L_{cpu,j}^{current}$ is set to 300m based on $L_{cpu,j}^{request}$. However, the system now lacks the resources, S_{cpu}^{avail} to deploy the remaining two workloads. These workloads are tried twice with $L_{cpu,j}^{request}$ and $L_{cpu,j}^{base}$ before rejected.

Then, the experiment is run again with even tighter resources by giving the extra workload 900m CPU utilization to limit S_{cpu}^{avail} to 100m. A similar deployment pattern

can be seen in the second CPU graph in Figure 4.17(b). The only difference is that the first workload is deployed with $L_{cpu,j}^{base}$ as there is not enough S_{cpu}^{avail} for $L_{cpu,j}^{request}$.

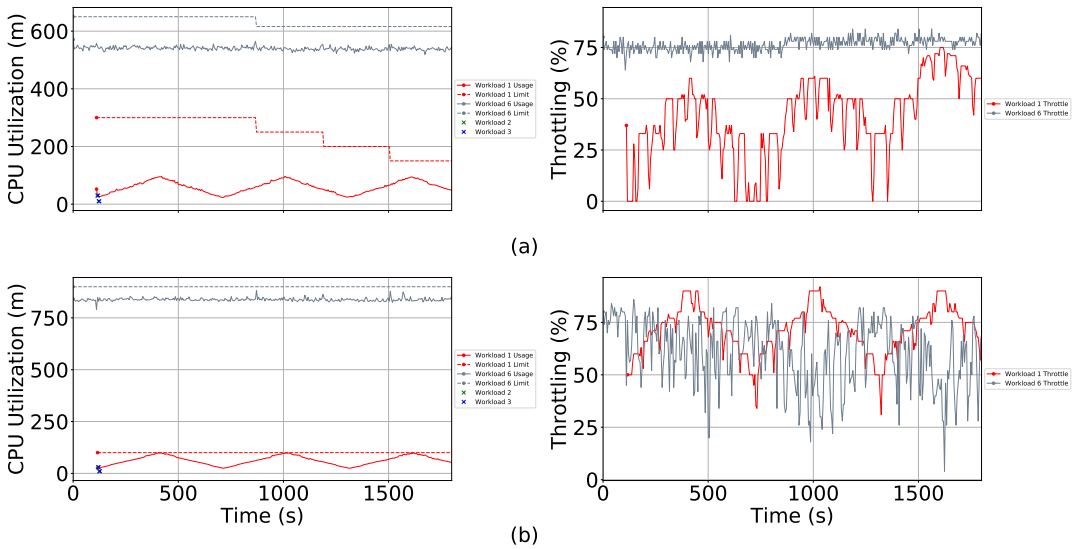

Figure 4.17. CPU Workloads Deployed Under Tighter Resource Constraints.
(a) Deployments on a System with 350m CPU Availability.
(b) Deployments on a System with 100m CPU Availability.

4.4. Experiment with a Real-time Video Streaming Application

Throughout this work, the heterogeneity of IoT hardware and software was emphasized. The workload patterns are a logical method to categorize heterogeneous IoT applications. Nevertheless, the framework's behavior should also be observed with an actual IoT application. A real-time streaming application is selected for this experiment. This application transmits the video data generated by a multimedia device such as a camera through the local network with Real-Time Messaging Protocol (RTMP). The full application pipeline is illustrated in Figure 4.18.

The RTMP server is a containerized application normally distributed through DockerHub [84]. The image of the same application can be delivered and deployed through our framework. This application's purpose is to share the video footage from

Open Broadcaster Software (OBS), which can capture camera, video, or other multimedia footage with multiple clients in the same network. After setting $L_{mem,c}^{request}$ and $L_{cpu,c}^{request}$ as 100MB and 100m respectively, an application request is sent to the framework.

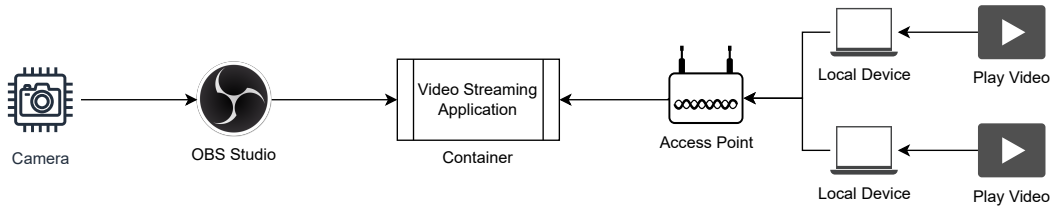

Figure 4.18. Real-time Video Streaming Application Pipeline.[1]

After going through the analysis steps, the application is deployed with $L_{r,c}^{request}$ where r in $\{cpu, mem\}$. The initial observation is the low resource consumption of the application as shown in Figure 4.19. This indicates that an RTMP server is a suitable deployment for resource-constrained edge gateways in a real-world scenario. The application continues to exhibit low and constant resource utilization. After the optimization steps, limits for both resources are reduced close to L_{mem}^{min} and L_{cpu}^{min}. Although up to 25% CPU throttling is observed after the CPU utilization optimization, no extra frame drops were observed on the video playback.

The primary optimization metric should typically be network usage as a video streaming application. Although transmitted bytes metric is being collected by the *Monitor* as shown in Section 3.4.4.1, the optimization workflow does not make use of network metrics yet. In the future, the *Analyzer* and *Forecaster* components of the framework can be configured to include the remaining fields in the optimization analysis steps to have more accurate predictions for a broader scope of applications.

4.5. Experiments with Cluster Deployment

The cluster deployment experiment is carried out with three resource-constrained devices with identical resources. This experiment is the realization of the high-level

[1]Licenses for the icons are as follows: OBS Logo, Hugh "Jim" Bailey, Public domain

design presented in Figure 3.19. As previously discussed in Section 3.5, there are some extra configuration steps in a cluster deployment. First, the IP address of each device is noted. Then, on each device, the MQTT configuration file is updated with two bridge connections towards the other two devices. In the same file, the bridged topics, *deploy* and *monitor*, are also defined. These configurations can be found in Appendix C for each device. After these configurations, the MQTT broker on each device is started, and our framework is deployed.

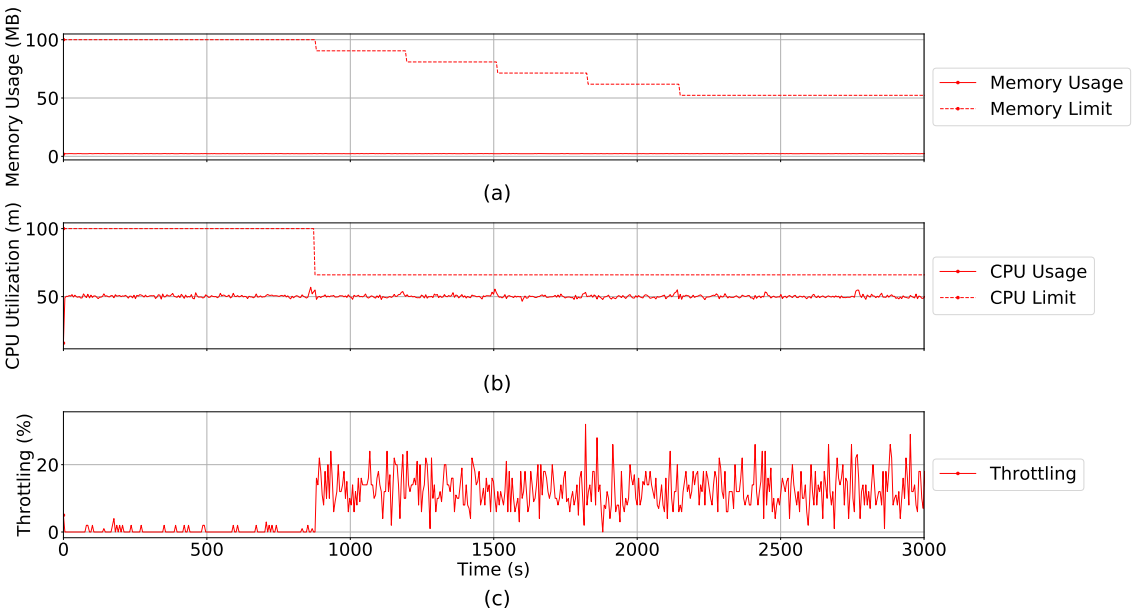

Figure 4.19. Real-time Streaming Application Deployment.
(a) Memory usage of the container.
(b) CPU utilization of the container.
(c) Throttling percentage of the container.

The experiment consists of sending deployment requests on a single device, *device A*, and observing the load balancing capabilities of the framework across all three devices, as shown in Figure 4.20. The expected behavior is for the monitoring and deployment messages to be shared across the network. Then based on these messages, each device should decide on the ideal candidate device to accept the deployment on its own in a decentralized manner. Therefore, deployment messages for the first four memory workload patterns are sent sequentially after each accepted deployment.

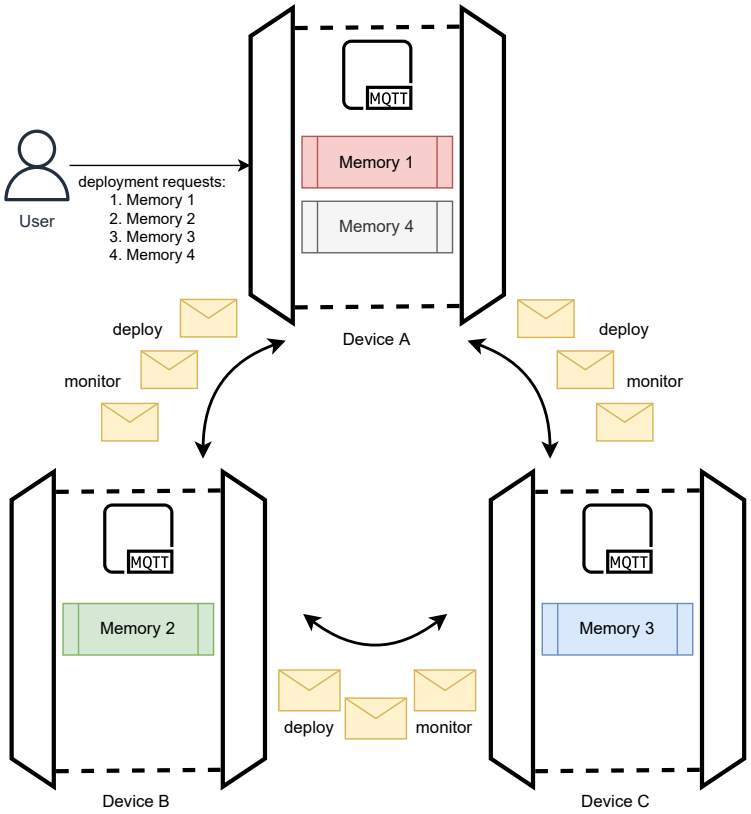

Figure 4.20. Cluster Deployment Setup with Three Devices.

These four individual deployments are load-balanced across the three connected devices based on their resource availability. It should be noted that if the available resources, S_r^{avail}, of multiple devices are the same, the device that has determined itself as the best candidate proceeds with the deployment. In this experiment, the value of IP addresses increases from device A to C, where *device A* is the device with the smallest IP address. Since in the beginning, S_r^{avail} is the same for all devices, the first application is deployed on *device A*. After this deployment, *monitoring_result* messages published by *device A* on *cluster/monitor* topic start reporting its reduced resource availability. *Devices B and C* update the resource availability hash maps that they keep locally with this decreased value. With each message, all three devices synchronize their resource availability information.

Following the first deployment, a deployment request for the application with the second workload pattern is sent to *device A*. If the framework had been deployed on a single device, *device A* would have continued with the deployment. In the cluster deployment, this request is shared over *cluster/deploy* topic. Then, each device checks its resource availability hash maps and identify *device B* as the processor of the second deployment request. After the deployment analysis is conducted on *device B*, the application is deployed. If the request fails to pass the analysis on *device B*, the *Deployer* of *device B* picks up the rejected deployment message. A new deployment message can be broadcasted with the next $L_{i,j}^{target}$ value on the *cluster/deploy* topic as illustrated in Figure 3.12.

The same steps are followed for the remaining two applications. The final distribution of deployed applications on the devices can be seen in Figure 4.20 after all deployments are completed. Without the cluster deployment, all four applications would have been deployed on a single device. An experiment for this case was carried out previously in Section 4.3.1. It should be noted that the cluster setup does not introduce latency overhead during deployments. If the application is deployed on the same device that has received the initial deployment request, the workflow is identical to the single device setup. Only a single extra event is exchanged over the *cluster/deploy* topic if another device satisfies the deployment request while the rest of the workflow stays the same.

5. CONCLUSION AND FUTURE WORK

In this thesis, a study is made on the decentralization of computing and orchestration edge IoT systems. The challenges introduced by the heterogeneity of the edge layer are investigated. Previous research shows that the computing capabilities of the edge devices can be utilized to offload some IoT tasks previously done on the cloud. Although resource-constrained, edge gateway devices are capable of running such tasks. Typically, these devices are centralized and tightly coupled with their applications. OS-level virtualization opens up the possibility of a single gateway device hosting multiple applications. A framework is designed to govern the co-location of container applications on resource-constrained edge gateway devices. The framework's application delivery and orchestration capabilities are presented with the choices made to achieve decentralization across all components.

A smart contract is written for the application delivery steps where developers can publish container applications. The same contract can be used to share applications' resource specifications with the framework running on the edge to establish SLAs. The Ethereum blockchain and a decentralized image registry are used through a distributed files system, IPFS, to achieve a fully decentralized workflow. The smart contract also provides support for updating the application and the resource specifications. With the separation of concerns in mind, four different containerized components are implemented to orchestrate the deployed applications. An event-based communication workflow is introduced for communication within the distributed components of the framework. This enables the individual components of the framework to coordinate the orchestration tasks asynchronously using the publish/subscribe pattern. These components are responsible for monitoring, container availability, resource allocation, and scaling. Another smart contract is written to manage the archival process of long-term metrics and use IPFS to back up past knowledge. The dynamic scaling operations are evaluated within the framework with time series forecasting and analysis.

A set of containerized applications representing various workload patterns of IoT tasks is created to analyze the framework's applicability. These applications are designed in pairs where each application in a pair depicts either CPU utilization or memory usage. The resource allocation decisions are also based on these two system-level metrics. The framework's capabilities are tested on a Raspberry Pi 4 Model B. A systemd slice is created to limit further the resource available on the device to a single CPU and 1 GB of memory. Then, these applications are deployed in various combinations and times, each representing different use cases. The scheduling and orchestration capabilities of the framework are observed based on the deployment and resource optimization decisions. In all iterations, the framework was able to adjust their resource allocation and ensure the co-location of deployed applications while honoring the resource specifications defined by the publisher. The framework retries failed deployments with revised resource constraints and adapts the limits based on the forecasting results of active containers' resource utilization.

The framework can be extended by improving the configuration steps and orchestration workflow in cluster deployment where the load is distributed across multiple gateways in the same edge network. Service discovery would be the initial improvement in this setup to automatically discover each gateway device on the network. Once service discovery is set in place, migration opportunities can be investigated. Live migration methods like Checkpoint-Restore in Userspace (CRIU) can boost container availability under tight resource constraints [85]. A migration analysis step can be introduced to mark applications for migration to a new host with higher resource availability. Such migration methodologies ensure that the state of the container is saved, and once relocated to a new host, its execution can resume from the saved state with minimal disruption.

REFERENCES

1. Zubair, N., N. A, K. Hebbar and Y. Simmhan, "Characterizing IoT Data and its Quality for Use", *arXiv Computing Research Repository [CoRR]*, 2019.

2. Sarkar, S. and S. Misra, "Theoretical Modelling of Fog Computing: A Green Computing Paradigm to Support IoT Applications", *IET Networks*, Vol. 5, No. 2, pp. 23–29, 2016.

3. Jalali, F., K. Hinton, R. Ayre, T. Alpcan and R. S. Tucker, "Fog Computing May Help to Save Energy in Cloud Computing", *IEEE Journal on Selected Areas in Communications*, Vol. 34, No. 5, pp. 1728–1739, 2016.

4. Muralidharan, S., G. Song and H. Ko, "Monitoring and Managing IoT Applications in Smart Cities Using Kubernetes", *Cloud Computing*, Vol. 11, 2019.

5. Rusek, M., G. Dwornicki and A. Orłowski, "A Decentralized System for Load Balancing of Containerized Microservices in the Cloud", J. Świątek and J. M. Tomczak (Editors), *Advances in Systems Science*, Advances in Intelligent Systems and Computing, pp. 142–152, Springer International Publishing, Cham, 2017.

6. Shafagh, H., L. Burkhalter, A. Hithnawi and S. Duquennoy, "Towards Blockchain-based Auditable Storage and Sharing of IoT Data", *Proceedings of the 2017 on Cloud Computing Security Workshop*, CCSW '17, pp. 45–50, Association for Computing Machinery, New York, NY, USA, 2017.

7. Kato, S., E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda and T. Hamada, "An Open Approach to Autonomous Vehicles", *IEEE Micro*, Vol. 35, No. 6, pp. 60–68, 2015.

8. Casalicchio, E. 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, p. e5668, 2020.

9. Santana, C., B. Alencar and C. Prazeres, "Microservices: A Mapping Study for Internet of Things Solutions", *IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pp. 1–4, 2018.

10. Nalin, G., "Orchestration of Smart Objects with MQTT for the Internet of Things", `http://tesi.cab.unipd.it/44964/`, accessed in February 2022.

11. Buzachis, A., A. Galletta, L. Carnevale, A. Celesti, M. Fazio and M. Villari, "Towards Osmotic Computing: Analyzing Overlay Network Solutions to Optimize the Deployment of Container-Based Microservices in Fog, Edge and IoT Environments", *IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pp. 1–10, 2018.

12. Celesti, A., L. Carnevale, A. Galletta, M. Fazio and M. Villari, "A Watchdog Service Making Container-Based Micro-services Reliable in IoT Clouds", *IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pp. 372–378, 2017.

13. Dolui, K. and C. Kiraly, "Towards Multi-Container Deployment on IoT Gateways", *IEEE Global Communications Conference (GLOBECOM)*, pp. 1–7, 2018.

14. Lee, B. and J.-H. Lee, "Blockchain-based Secure Firmware Update for Embedded Devices in an Internet of Things Environment", *The Journal of Supercomputing*, Vol. 73, No. 3, pp. 1152–1167, 2017.

15. Westerlund, M., J. Wickström and G. Pulkkis, "Version Control Using Distributed Ledger Technologies for Internet of Things Device Software Updates", *Cloud Computing*, p. 53, 2019.

16. Liu, J., K. Luo, Z. Zhou and X. Chen, "ERP: Edge Resource Pooling for Data Stream Mobile Computing", *IEEE Internet of Things Journal*, Vol. 6, No. 3, pp.

4355–4368, 2019.

17. Karwowski, W., M. Rusek, G. Dwornicki and A. Orłowski, "Swarm Based System for Management of Containerized Microservices in a Cloud Consisting of Heterogeneous Servers", L. Borzemski, J. Świątek and Z. Wilimowska (Editors), *Information Systems Architecture and Technology: Proceedings of 38th International Conference on Information Systems Architecture and Technology – ISAT 2017*, Advances in Intelligent Systems and Computing, pp. 262–271, Springer International Publishing, Cham, 2018.

18. Benet, J., "IPFS - Content Addressed, Versioned, P2P File System", *arXiv Computing Research Repository [CoRR]*, 2014.

19. Dias, D. and J. Benet, "Distributed Web Applications with IPFS, Tutorial", A. Bozzon, P. Cudre-Maroux and C. Pautasso (Editors), *Web Engineering*, Lecture Notes in Computer Science, pp. 616–619, Springer International Publishing, Cham, 2016.

20. "Take a Look at Pubsub on IPFS", `https://blog.ipfs.io/25-pubsub/`, accessed in February 2022.

21. Chen, Y. and T. Kunz, "Performance Evaluation of IoT Protocols Under a Constrained Wireless Access Network", *International Conference on Selected Topics in Mobile Wireless Networking (MoWNeT)*, pp. 1–7, 2016.

22. Gündoğan, C., P. Kietzmann, M. Lenders, H. Petersen, T. C. Schmidt and M. Wählisch, "NDN, CoAP, and MQTT: A Comparative Measurement Study in the IoT", *Proceedings of the 5th ACM Conference on Information-Centric Networking*, ICN '18, pp. 159–171, Association for Computing Machinery, New York, NY, USA, 2018.

23. Colombo, P. and E. Ferrari, "Access Control Enforcement within MQTT-based

Internet of Things Ecosystems", *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies*, SACMAT '18, pp. 223–234, Association for Computing Machinery, New York, NY, USA, 2018.

24. Larmo, A., A. Ratilainen and J. Saarinen, "Impact of CoAP and MQTT on NB-IoT System Performance", *Sensors*, Vol. 19, No. 1, p. 7, 2019.

25. Sareen, P., P. Kumar and T. D. Singh, "Resource Allocation Strategies in Cloud Computing", *International Journal of Computer Science & Communication Networks*, Vol. 5, No. 6, pp. 358–365, 2015.

26. Xiong, X., K. Zheng, L. Lei and L. Hou, "Resource Allocation Based on Deep Reinforcement Learning in IoT Edge Computing", *IEEE Journal on Selected Areas in Communications*, Vol. 38, No. 6, pp. 1133–1146, 2020.

27. Liu, X., J. Yu, J. Wang and Y. Gao, "Resource Allocation With Edge Computing in IoT Networks via Machine Learning", *IEEE Internet of Things Journal*, Vol. 7, No. 4, pp. 3415–3426, 2020.

28. Pešić, S., M. Radovanović and M. Ivanović, "An MQTT-based Resource Management Framework for Edge Computing Systems", *International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*, pp. 1–7, 2020.

29. Zhao, M. and K. Zhou, "Selective Offloading by Exploiting ARIMA-BP for Energy Optimization in Mobile Edge Computing Networks", *Algorithms*, Vol. 12, No. 2, p. 48, 2019.

30. Ghodsi, A., M. Zaharia, B. Hindman, A. Konwinski, S. Shenker and I. Stoica, "Dominant Resource Fairness: Fair Allocation of Multiple Resource Types", p. 14.

31. Yigitoglu, E., M. Mohamed, L. Liu and H. Ludwig, "Foggy: A Framework for Continuous Automated IoT Application Deployment in Fog Computing", *IEEE International Conference on AI Mobile Services (AIMS)*, pp. 38–45, 2017.

32. Peinl, R., F. Holzschuher and F. Pfitzer, "Docker Cluster Management for the Cloud - Survey Results and Own Solution", *Journal of Grid Computing*, Vol. 14, No. 2, pp. 265–282, 2016.

33. Burger, A., C. Cichiwskyj, S. Schmeißer and G. Schiele, "The Elastic Internet of Things - A Platform for Self-integrating and Self-adaptive IoT-systems with Support for Embedded Adaptive Hardware", *Future Generation Computer Systems*, Vol. 113, pp. 607–619, 2020.

34. Imran, S. Ahmad and D. H. Kim, "A Task Orchestration Approach for Efficient Mountain Fire Detection Based on Microservice and Predictive Analysis in IoT Environment", *Journal of Intelligent & Fuzzy Systems*, Vol. 40, No. 3, pp. 5681–5696, 2021.

35. Zhong, Z., M. Xu, M. A. Rodriguez, C. Xu and R. Buyya, "Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions", *ACM Computing Surveys (CSUR)*, 2021.

36. Bashir, H., S. Lee and K. H. Kim, "Resource Allocation Through Logistic Regression and Multicriteria Decision Making Method in IoT Fog Computing", *Transactions on Emerging Telecommunications Technologies*, Vol. 33, No. 2, p. e3824.

37. Ajila, S. A. and A. A. Bankole, "Cloud Client Prediction Models Using Machine Learning Techniques", *IEEE 37th Annual Computer Software and Applications Conference*, pp. 134–142, 2013.

38. Zhong, Z., J. He, M. A. Rodriguez, S. Erfani, R. Kotagiri and R. Buyya, "Heterogeneous Task Co-location in Containerized Cloud Computing Environments", *IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 79–88, 2020.

39. Jiang, C., J. Wan and H. Abbas, "An Edge Computing Node Deployment Method

Based on Improved K-Means Clustering Algorithm for Smart Manufacturing", *IEEE Systems Journal*, Vol. 15, No. 2, pp. 2230–2240, 2021.

40. Rossi, F., M. Nardelli and V. Cardellini, "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning", *IEEE 12th International Conference on Cloud Computing (CLOUD)*, pp. 329–338, 2019.

41. Chien, W.-C., H.-Y. Weng and C.-F. Lai, "Q-learning Based Collaborative Cache Allocation in Mobile Edge Computing", *Future Generation Computer Systems*, Vol. 102, pp. 603–610, 2020.

42. Edsinger, D., *Auto-scaling Cloud Infrastructure with Reinforcement Learning: A Comparison between Multiple RL Algorithms to Auto-scale Resources in Cloud Infrastructure*, Master's Thesis, 2018.

43. Liu, Y., C. Yang, L. Jiang, S. Xie and Y. Zhang, "Intelligent Edge Computing for IoT-Based Energy Management in Smart Cities", *IEEE Network*, Vol. 33, No. 2, pp. 111–117, 2019.

44. Liu, X., J. Yu, Z. Feng and Y. Gao, "Multi-agent Reinforcement Learning for Resource Allocation in IoT Networks with Edge Computing", *China Communications*, Vol. 17, No. 9, pp. 220–236, 2020.

45. Zhao, X. and C. Wu, "Large-scale Machine Learning Cluster Scheduling via Multi-agent Graph Reinforcement Learning", *IEEE Transactions on Network and Service Management*, 2021.

46. Adel Serhani, M., H. T. El-Kassabi, K. Shuaib, A. N. Navaz, B. Benatallah and A. Beheshti, "Self-adapting Cloud Services Orchestration for Fulfilling Intensive Sensory Data-driven IoT Workflows", *Future Generation Computer Systems*, Vol. 108, pp. 583–597, 2020.

47. Li, C., H. Sun, H. Tang and Y. Luo, "Adaptive Resource Allocation Based on the

Billing Granularity in Edge-cloud Architecture", *Computer Communications*, Vol. 145, pp. 29–42, 2019.

48. Tan, C. N. L., C. Klein and E. Elmroth, "Location-aware Load Prediction in Edge Data Centers", *Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 25–31, 2017.

49. Yassine, A., S. Singh, M. S. Hossain and G. Muhammad, "IoT Big Data Analytics for Smart Homes with Fog and Cloud Computing", *Future Generation Computer Systems*, Vol. 91, pp. 563–573, 2019.

50. Nawrocki, P., M. Grzywacz and B. Sniezynski, "Adaptive Resource Planning for Cloud-based Services Using Machine Learning", *Journal of Parallel and Distributed Computing*, Vol. 152, pp. 88–97, 2021.

51. Paul, U., A. P. Subramanian, M. M. Buddhikot and S. R. Das, "Understanding Traffic Dynamics in Cellular Data Networks", *Proceedings IEEE INFOCOM*, pp. 882–890, 2011.

52. Vogel, P., B. A. Neumann Saavedra and D. C. Mattfeld, "A Hybrid Metaheuristic to Solve the Resource Allocation Problem in Bike Sharing Systems", M. J. Blesa, C. Blum and S. Voß (Editors), *Hybrid Metaheuristics*, Lecture Notes in Computer Science, pp. 16–29, Springer International Publishing, Cham, 2014.

53. Tom, R. J., S. Sankaranarayanan and J. J. P. C. Rodrigues, "Smart Energy Management and Demand Reduction by Consumers and Utilities in an IoT-Fog-Based Power Distribution System", *IEEE Internet of Things Journal*, Vol. 6, No. 5, pp. 7386–7394, 2019.

54. Nikravesh, A., D. R. Choffnes, E. Katz-Bassett, Z. M. Mao and M. Welsh, "Mobile Network Performance from User Devices: A Longitudinal, Multidimensional Analysis", D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C.

Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, M. Faloutsos and A. Kuzmanovic (Editors), *Passive and Active Measurement*, Vol. 8362, pp. 12–22, Springer International Publishing, Cham, 2014.

55. Yang, S., Y. Ren, J. Zhang, J. Guan and B. Li, "KubeHICE: Performance-aware Container Orchestration on Heterogeneous-ISA Architectures in Cloud-Edge Platforms", *IEEE International Conference on Parallel Distributed Processing with Applications, Big Data Cloud Computing, Sustainable Computing Communications, Social Computing Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pp. 81–91, 2021.

56. Baresi, L., D. F. Mendonça and G. Quattrocchi, "PAPS: A Framework for Decentralized Self-management at the Edge", S. Yangui, I. Bouassida Rodriguez, K. Drira and Z. Tari (Editors), *Service-Oriented Computing*, pp. 508–522, Springer International Publishing, Cham, 2019.

57. Ajayi, O., J. Rafferty, J. Santos, M. Garcia-Constantino and Z. Cui, "BECA: A Blockchain-Based Edge Computing Architecture for Internet of Things Systems", *IoT*, Vol. 2, pp. 610–632, 2021.

58. Struhár, V., S. S. Craciunas, M. Ashjaei, M. Behnam and A. V. Papadopoulos, "REACT: Enabling Real-Time Container Orchestration", *26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pp. 1–8, 2021.

59. Goethals, T., F. De Turck and B. Volckaert, "FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices", pp. 174–189, 2020.

60. Xiong, Y., Y. Sun, L. Xing and Y. Huang, "Extend Cloud to Edge with KubeEdge", *IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 373–377, 2018.

61. Cicconetti, C., M. Conti and A. Passarella, "A Decentralized Framework for Serverless Edge Computing in the Internet of Things", *IEEE Transactions on Network and Service Management*, Vol. 18, No. 2, pp. 2166–2180, 2021.

62. Pires, A., J. Simão and L. Veiga, "Distributed and Decentralized Orchestration of Containers on Edge Clouds", *Journal of Grid Computing*, Vol. 19, No. 3, p. 36, 2021.

63. Cui, L., Z. Chen, S. Yang, Z. Ming, Q. Li, Y. Zhou, S. Chen and Q. Lu, "A Blockchain-Based Containerized Edge Computing Platform for the Internet of Vehicles", *IEEE Internet of Things Journal*, Vol. 8, No. 4, pp. 2395–2408, 2021.

64. Rouhani, S. and R. Deters, "Security, Performance, and Applications of Smart Contracts: A Systematic Survey", *IEEE Access*, Vol. 7, pp. 50759–50779, 2019.

65. Koziolek, H., S. Grüner and J. Rückert, "A Comparison of MQTT Brokers for Distributed IoT Edge Computing", A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya and O. Zimmermann (Editors), *Software Architecture*, Lecture Notes in Computer Science, pp. 352–368, Springer International Publishing, Cham, 2020.

66. Bellavista, P. and A. Zanni, "Towards Better Scalability for IoT-cloud Interactions via Combined Exploitation of MQTT and CoAP", *IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a Better Tomorrow (RTSI)*, pp. 1–6, 2016.

67. Sommer, P., F. Schellroth, M. Fischer and J. Schlechtendahl, "Message-oriented Middleware for Industrial Production Systems", *IEEE 14th International Conference on Automation Science and Engineering (CASE)*, pp. 1217–1223, 2018.

68. "Develop with Docker Engine API", `https://docs.docker.com/engine/api/`, accessed in February 2022.

69. "Echo Dot 3rd Gen Smart Speaker Teardown", `https://www.briandorey.com`

/post/echo-dot-3rd-gen-smart-speaker-teardown, accessed in February 2022.

70. "Runtime Options with Memory, CPUs, and GPUs", https://docs.docker.com/config/containers/resource_constraints/, accessed in February 2022.

71. "Docker Hub Container Image Library", https://hub.docker.com/, accessed in February 2022.

72. "IPDR: InterPlanetary Docker Registry", https://github.com/ipdr/ipdr, accessed in February 2022.

73. Arcaini, P., E. Riccobene and P. Scandurra, "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation", *IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pp. 13–23, 2015.

74. "Time-series | Stata", https://www.stata.com/features/time-series/, accessed in February 2022.

75. "ARIMA — Statsmodels", https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMA.html, accessed in February 2022.

76. Chiluk, D., "CPU Throttling - Unthrottled: Fixing CPU Limits in the Cloud", https://engineering.indeedblog.com/blog/2019/12/unthrottled-fixing-cpu-limits-in-the-cloud/, accessed in February 2022.

77. Rende, D., "Dag Rende: Find the MQTT Broker without an IP Address", http://dagrende.blogspot.com/2017/02/find-mqtt-broker-without-hard-coded-ip.html, accessed in February 2022.

78. "Raspberry Pi 4 Model B Specifications", https://www.raspberrypi.com/products/raspberry-pi-4-model-b/, accessed in February 2022.

79. "systemd.slice(5) - Linux Manual Page", https://man7.org/linux/man-pages/

man5/systemd.slice.5.html, accessed in February 2022.

80. "Control Groups, Part 4: On Accounting", https://lwn.net/Articles/606004/, accessed in February 2022.

81. Taherizadeh, S. and V. Stankovski, "Dynamic Multi-level Auto-scaling Rules for Containerized Applications", *The Computer Journal*, Vol. 62, No. 2, pp. 174–197, 2019.

82. "Resource Management for Pods and Containers", https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/, accessed in February 2022.

83. "Byte", https://en.wikipedia.org/w/index.php?title=Byteoldid=1082283626, accessed in February 2022.

84. "tiangolo/nginx-rtmp - Docker Image Docker Hub", https://hub.docker.com/r/tiangolo/nginx-rtmp/, accessed in February 2022.

85. "Live Migration - CRIU", https://criu.org/Live_migration, accessed in February 2022.

APPENDIX A: SETTING UP DOCKER RESOURCE CONSTRAINTS

A normal Docker setup runs with no resource constraints on the parent system. Although, the Desktop version offers resource customization settings there is no such setting for the server version of Docker. On the Linux kernel, slice units can be used to manage resources of processes. As long as the Docker daemon is configured to run on a slice of its own, all containers running will satisfy the constraints set on the same slice limit. A new slice is created:

- /etc/systemd/system/docker_limit.slice

In the slice configuration, it's possible to set limits for CPU and memory. The CPUQuota is set to use a single CPU. The MemoryLimit is set to 1GB. MemorySwap is also disabled. After the slice unit is created, Docker's cgroup-parent should be assigned to this newly created slice. The cgroup can be set inside "/etc/docker/daemon.json". By restarting the daemon it can be observed with the *docker info* command that slice is active and Docker daemon is restricted by the configuration set.

APPENDIX B: EXAMPLE ORCHESTRATION MESSAGES

```
{
  "trace_id": "1",
  "action": "deployment_request",
  "image_name": "application"
}
```

Figure B.1. An Example of Deployment Request.

```
{
    "trace_id": "1",
    "action": "deployment_analysis_request",
    "image_name": "application",
    "resources": {
      "cpu": 5,
      "memory": 300000000
    },
    "forecast_fields": ["cpu_usage, memory_usage"]
}
```

Figure B.2. An Example of Deployment or Optimization Analysis Request.

```
{
    "trace_id": "1",
    "action": "forecast_request",
    "image_name": "application",
    "resources": {
      "cpu": 5,
      "memory": 300000000
    },
    "forecast_fields": ["cpu_usage", "memory_usage"],
    "images": {
      "application_1": {
        "containers": {
          "9d15...44dc": {}
        }
      },
      "application_2": {
        "containers": {
          "be23...5bee": {}
        }
      },
      "application_3": {
        "containers": {
          "3950...02a9": {}
        }
      }
    }
}
```


Figure B.3. An Example of Forecast Request.

```
{
    "trace_id": "1",
    "action": "forecast_response",
    "image_name": "application",
    "resources": {
      "cpu": 5,
      "memory": 300000000
    },
    "forecast_fields": ["cpu_usage", "memory_usage"],
    "images": {
      "application_1": {
        "containers": {
          "9d15...44dc": {
            "prediction": {
              "cpu_usage": [0.0, 0.0, 0.0, 0.0, 0.0],
              "memory_usage": [1839104.0, 1839104.0,
                  1839104.0, 1839104.0, 1839104.0]
            }
          }
        }
      },
      "application_2": {
        "containers": {
          "be23...5bee": {
            "prediction": {
```


Figure B.4. An Example of Forecast Response.

```
            "cpu_usage": [6.071608496023066,
              6.0519568969621504, 5.966500648610864,
              5.674217082010118, 5.633116742073893],
            "memory_usage": [622592.0, 622592.0, 622592.0,
              622592.0, 622592.0]
          }
        }
      }
    },
    "application_3": {
      "containers": {
        "3950...02a9": {
          "prediction": {
            "cpu_usage": [5.384377813287831,
              6.991252506916891, 9.493952140427064,
              10.357942298607727, 11.192738901089502],
            "memory_usage": [5562666.274354553,
              5564592.8728406355, 5562953.402692312,
              5562986.773036226, 5563218.888662458]
          }
        }
      }
    }
}
```


Figure B.4. An Example of Forecast Response. (cont.)

```
{
    "trace_id": "1",
    "image_name": "application",
    "action": "deploy_ok",
    "message": "Enough future resources",
    "resources": {
      "cpu": 5,
      "memory": 300000000
    }
}
```


Figure B.5. An Example of Deployment Accept.


```
{
    "trace_id": "2",
    "image_name": "application",
    "resources": {
     "cpu": 30,
     "memory": 150000000
    },
    "action": "cancel",
    "message": "Not enough resources"
}
```


Figure B.6. An Example of Deployment Reject.

```
{
    "trace_id": "3",
    "action": "update",
    "image_name": "application_1",
    "container_id": "3950...02a9",
    "resources": {
     "cpu": 6.0,
     "memory": 10000000
    }
}
```

Figure B.7. An Example of Deployment Update.

```
{
    "application_2": {
     "containers": {
       "54aa...1f96": {
         "stats": {
           "cpu_usage": 8429,
           "memory_usage": 5292032,
           "block_read": 0,
           "block_write": 0,
           "network_read": 62639,
           "network_write": 690,
           "read_time": "2022-01-01 10:06:44",
           "throttle": 82
        }}
    }}
}
```

Figure B.8. An Example of Monitor Output.

APPENDIX C: MQTT BRIDGE CONFIGURATIONS

```
# device A address 192.168.1.1

connection bridge-A-to-B
address 192.168.1.2:1883

topic monitor out 0 fw/ fw/cluster/
topic deploy out 0 fw/ fw/cluster/

connection bridge-A-to-C
address 192.168.1.3:1883

topic monitor out 0 fw/ fw/cluster/
topic deploy out 0 fw/ fw/cluster/
```

Figure C.1. Device A Mosquitto Configuration.

```
# device B address 192.168.1.2

connection bridge-B-to-A
address 192.168.1.1:1883

topic monitor out 0 fw/ fw/cluster/
topic deploy out 0 fw/ fw/cluster/

connection bridge-B-to-C
address 192.168.1.3:1883

topic monitor out 0 fw/ fw/cluster/
topic deploy out 0 fw/ fw/cluster/
```

Figure C.2. Device B Mosquitto Configuration.

```
# device C address 192.168.1.3

connection bridge-C-to-A
address 192.168.1.1:1883

topic monitor out 0 fw/ fw/cluster/
topic deploy out 0 fw/ fw/cluster/

connection bridge-C-to-B
address 192.168.1.2:1883

topic monitor out 0 fw/ fw/cluster/
topic deploy out 0 fw/ fw/cluster/
```

Figure C.3. Device C Mosquitto Configuration.