A Decentralized Framework with Dynamic and Event-Driven Container Orchestration at the Edge

Umut Can Özyar
Computer Engineering Department
Bogazici University
Istanbul, Turkey
umut.ozyar@boun.edu.tr

Arda Yurdakul
Computer Engineering Department
Bogazici University
Istanbul, Turkey
yurdakul@boun.edu.tr

***Abstract*—Virtualization provides an abstraction layer for the Internet of Things technology to tackle the heterogeneity of the edge networks. Deploying virtualized applications on different architectures requires autonomous scaling and load balancing while ensuring their authenticity. A decentralized end-to-end solution is necessary for applications with variable workloads to co-exist on a heterogeneous environment with multiple edge devices. Hence, this study lays down the fundamentals of a framework for dynamic and event-driven orchestration towards a fully decentralized edge. It provides a blockchain-based delivery platform for containerized applications registered with their resource requirements through a registry on a distributed file system. The decentralized 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. An event-driven architecture is built over a lightweight messaging protocol, MQTT, capitalizing on the asynchronous and distributed nature of the publish/subscribe pattern to achieve a truly distributed system.**

***Index Terms*—Edge computing, resource-constrained devices, orchestration, containers, decentralized applications**

I. Introduction

The term "Internet of Things (IoT)" describes a sophisticated system of heterogeneous devices, dynamic environments, and complex sub-systems [1]. The quality of IoT services delivered to the end-users is a major concern of the application developers. Edge computing has been proposed to improve user experience by moving services, processing, and actionable insights closer to the user. However, edge devices are also heterogeneous in terms of processing power, memory amount, and operating system. Hence, virtualization schemes, such as containerization, widely used in computer systems have recently stepped into the domain of edge computing [2]. As containers can be scaled according to the available resources of the host system, application developers can present an IoT solution in a container that should ideally execute on almost every edge device [3].

Even though the usage of containers is a breath-taking solution for IoT application developers, it bears its own problems that have to be resolved for a seamless user experience. Firstly, as edge devices are implemented with different types of hardware, container scaling should be done at the edge autonomously. Since each application can have a unique workload during runtime, dynamic load balancing has to be taken into account to maximize resource utilization while scaling the containers. Secondly, the authenticity of the container should be ensured prior to its deployment or upgrade, because malicious containers can overwhelm the host edge device which may be running on scarce resources. Finally, irresponsive or latent services at the edge should be avoided to improve user experience through autonomous deployment of containers to neighboring devices.

In this study, we propose a decentralized framework for autonomous deployment and scaling of containerized applications on resource-constrained edge devices. The secure delivery of containerized applications is enabled by a smart contract that holds container registries on the blockchain. The container images are stored in a decentralized file system, namely, IPFS [4]. The use of blockchain and IPFS provides the ubiquity of the applications for a seamless user experience. Once the user registers to a service or employs an IoT device, the container deployment is done autonomously based on available on-device resources. An event-driven decentralized resource manager is designed for this purpose. It analyzes running services and forecasts future requirements. Based on its findings, it scales containers running on the device. The decision about the new container depends on where the edge device is deployed. If there exist multiple devices at the edge, their resource managers talk, and the device with maximum abundant resources employs the container. Otherwise, only on-device resources are used in giving the decision. Our study is unique in the sense that it provides an end-to-end solution between the application release and the user experience.

The rest of the paper is organized as follows: The next section presents related studies in the literature. Section III introduces the proposed framework while explaining design choices and incorporated technologies. Section IV analyzes the behavior of the framework under different experimental setups. The final section concludes the work.

II. Related Works

In the recent literature, there exist several studies on edge container orchestration frameworks. An overview is presented in Table I.

As mentioned in the previous section, the heterogeneous nature of IoT end devices requires scaling. Hence, autonomous

scaling (C1) of the containers has to be done at the edge for a seamless user experience. If supported by the system, the options are horizontal (H) or vertical (V). Horizontal scaling refers to changing the number of containers of an application to meet the varying loads. Performance on a single-CPU single-threaded edge device may be degraded in horizontally-scaled containers because having multiple instances of the same application will have to execute sequentially. Thus, [5]–[10] explore horizontal scaling across all available devices in their networks. Vertical scaling is the adaptation of the resources of an existing container. Containerization engines such as Docker provide all necessary tools for vertical scaling, making it inherently more straightforward to configure than horizontal scaling. Scaling type is tightly correlated with software heterogeneity (C3), such as accommodation of different kinds of applications like microservices, batch jobs, and streaming applications on the same device [5], [6], [8], [11]–[14]. Since our target is the utilization of software heterogeneity on resource-constrained edge devices, vertical scaling is preferred.

Decision-taking for autonomous scaling can be either reactive or predictive. Reactive methods only consider the current state. Predictive methods (C2) consider the past and future states of the system. In [10] predictive methods such as machine learning are proposed for horizontal scaling. When vertical scaling is considered, predictive solutions provide a smoother user experience and better decision performance than reactive ones. Our framework relies on statistical models for time-series forecasting as a reliable method that doesn't depend on large amounts of training data. Auto-scaling decisions are based on optimization metrics (C6). The two main categories are application (A) and system (S) metrics. Application metrics are driven by the application requirements such as response time and error rate [7]–[10]. System metrics, such as CPU and memory, are harnessed from the host. Application metrics fail to offer a viable solution when it has to execute on different types of hardware. Hence, our framework utilizes system metrics so that the applications can run on all types of hardware. Frameworks of [3], [5], [6], [10], [12], [15] focus on the resource-constrained devices (C4) as we do, while the rest are deployed on more powerful devices on the edge. This makes on-device orchestration (C5) a challenging issue since it also consumes system resources [3], [6], [7], [9], [10], [13], [15]. In this study, on-device orchestration is also adopted as it supports decentralization.

Edge computing paradigm inherently supports the decentralization of every IoT service [13]. However, the studies in the literature suffer from centralized application registries throughout the whole delivery process. A centralized registry gives the authority to set download rates, storage limits, or pricing decisions to a single party, granting them unfair advantages as well as introducing a single point of failure. Traditional decentralized registry solutions may suffer from some security issues such as access control, immutability, and authenticity of applications. To cope with this limitation, our decentralized delivery process leverages blockchain technology and a tamper-proof distributed file system to provide proof of ownership on the smart contracts (C8). Decentralized communication between edge devices is also essential. Event-driven communication (C7) using the publish/subscribe pattern is adopted in many studies [3], [5], [6], [9], [14] as its resilience and robustness make it a popular choice on the edge for building scalable platforms. It is also strategic for decentralized load balancing (C9) across a cluster of edge devices autonomously [7], [13], [15] to distribute the load on other edge devices. Since these works assume trustless setups, a consensus mechanism is implemented. Since we consider secured resource-constrained devices and blockchain is used for secure application delivery, a two-step broadcast followed by an approval-by-silence mechanism is proposed in this work.

TABLE I
OVERVIEW OF EDGE ORCHESTRATION FRAMEWORKS.

	Criteria[1]								
Authors	***C1***	***C2***	***C3***	***C4***	***C5***	***C6***	***C7***	***C8***	***C9***
[3]	✗	✗	✗	✓	✓	✗	✓	✗	✗
[5]	H	✗	✓	✓	✗	✗	✓	✗	✗
[6]	H	✗	✓	✓	✓	S	✓	✗	✗
[7]	H	✗	✗	✗	✓	A	✗	✗	✓
[8]	H/V	✗	✓	✗	✗	S/A	✗	✗	✗
[9]	H	✗	✗	✗	✓	A	✓	✗	✗
[10]	H/V	✓	✗	✓	✓	S/A	✗	✗	✗
[11]	✗	✗	✓	✗	✗	S	✗	✗	✗
[12]	✗	✗	✓	✓	✗	✗	✗	✗	✗
[13]	✗	✗	✓	✗	✓	✗	✗	✗	✓
[14]	✗	✗	✓	✗	✗	✗	✓	✗	✗
[15]	✗	✗	✗	✓	✓	✗	✗	✗	✓
This work	V	✓	✓	✓	✓	S	✓	✓	✓

[1] Description of Abbreviations: C1: Auto-scaling, C2: Predictive Scaling, C3: Software Heterogeneity, C4: Resource Constrained Hardware, C5: On-Device Orchestration, C6: Optimization Metrics, C7: Event-driven Communication, C8: Decentralized Application Delivery, C9: Decentralized Load Balancing, H: Horizontal, V: Vertical, A: Application, S: System

III. FRAMEWORK ARCHITECTURE

The proposed framework manages the lifecycle of containerized applications, including design, delivery, execution, and optimization steps. It provides data storage, resource management, monitoring, application registry and delivery solutions, including service upgrades and software updates. The overview of the framework's architecture on the edge is illustrated in Figure 1. The framework on the edge is designed as a distributed set of components to support decentralized orchestration. It consists of four components deployed on the edge device. *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.

The four components of the framework rely on an array of tools and technologies to form the full framework as shown in Figure 1. *Docker* provides an engine with virtualization

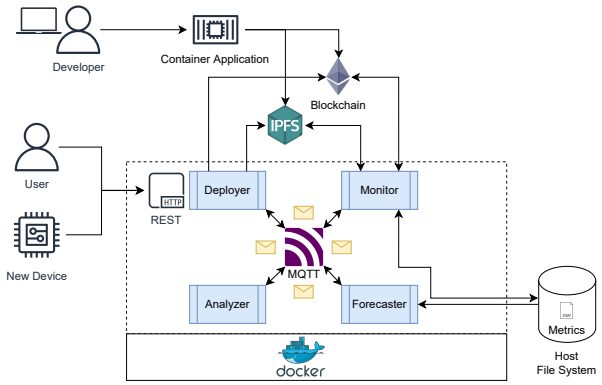

Fig. 1. Overview of the framework's architecture on the edge.

TABLE II
SUPPORTED ORCHESTRATION ACTIONS AND MQTT TOPICS

Action	Topic
Deployment Request	Deploy
Deployment Analysis Request	Analyze
Deployment Optimization Request	Analyze
Forecast Request	Forecast
Forecast Response	Forecast
Deployment Accept	Deploy
Deployment Cancel	Deploy
Deployment Update	Deploy
Monitoring Result	Monitor

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. *IPFS-Backed Docker Registry (IPDR)* is a Docker registry proxy that utilizes IPFS for decentralized image storage [16]. *Blockchain* hosts smart contracts that connect the application delivery process with the edge framework. *Host File System* is a mounted volume through Docker to grant short-term storage for application metrics on the host device. The framework's components communicate with the publish/subscribe pattern. *Message Queueing Telemetry Transport (MQTT) Broker* is a prominent lightweight protocol for event-driven architectures on the edge with low resource and power consumption [17]. Here, components publish messages on specific topics to pass events and data for subscribed components to pick up. Messages contains the `action` field for identification of a message's purpose. Subscribed components act upon the received *action* and payload of a message. Table II presents a list of supported actions and their target topics.

A. Application Delivery

The application delivery process provides a pipeline for developers to publish and deliver applications to their clients. The pipeline shown in Figure 2 is built over a decentralized application registry. Versioning, updates, and upgrades are managed on top of IPFS using IPDR as a decentralized and agile solution. IPDR offers the advantage of downloading layers of a Docker image individually with the use of less bandwidth and local storage. In this work, an interface for this decentralized Docker registry is developed as a decentralized application with a smart contract using the data structures and functions in Listing 1. Developers start the *Release* process of a container application. The application release consists of either *Publish*, or subsequent *Update* processes recurring for each service upgrade and software update. First, the released or updated Docker image must be stored on IPDR, which returns an IPFS hash. Then, the application metadata stored on the blockchain is manipulated through the smart contract. The metadata consists of the image hash acquired from IPFS and an optional set of resource limits such as `requestedLimit` and `baseLimit` that can be configured by the developer. Data stored on IPFS cannot be traced to its owner unless proof of ownership is deliberately included; nevertheless, the smart contract allows developers to prove the application's authenticity. The delivery process ensures that applications are tamper-proof and only the owner can deliver updates and upgrades.

```
1  /* Image ownership */
2  mapping(address => Image) images;
3  /* Image metadata */
4  struct Image {
5      string imageHash;
6      string imageName;
7      uint baseLimitMemory;
8      uint requestLimitMemory;
9      uint baseLimitCPU;
10     uint requestLimitCPU;
11 }
12 /* Get image metadata */
13 function get(address imageOwner,
14         string memory imageName)
15         public view returns (Image memory);
16 /* Set image metadata */
17 function set(string memory imageHash,
18                 string memory imageName,
19                 uint baseLimitMemory,
20                 uint requestLimitMemory,
21                 uint baseLimitCPU,
22                 uint requestLimitCPU
23                 ) public;
```

Listing 1. Core data structure and functions for application delivery.

B. Application Orchestration

The framework components shown in Figure 1 asynchronously pass messages between each other to provide dynamic orchestration shown in Figure 2. The messaging architecture is inspired by the *service providers* and *session messages* described in [18]. The orchestration flow is a step-by-step process executed on the edge. 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 [19].

1) Deployment Admission: 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.

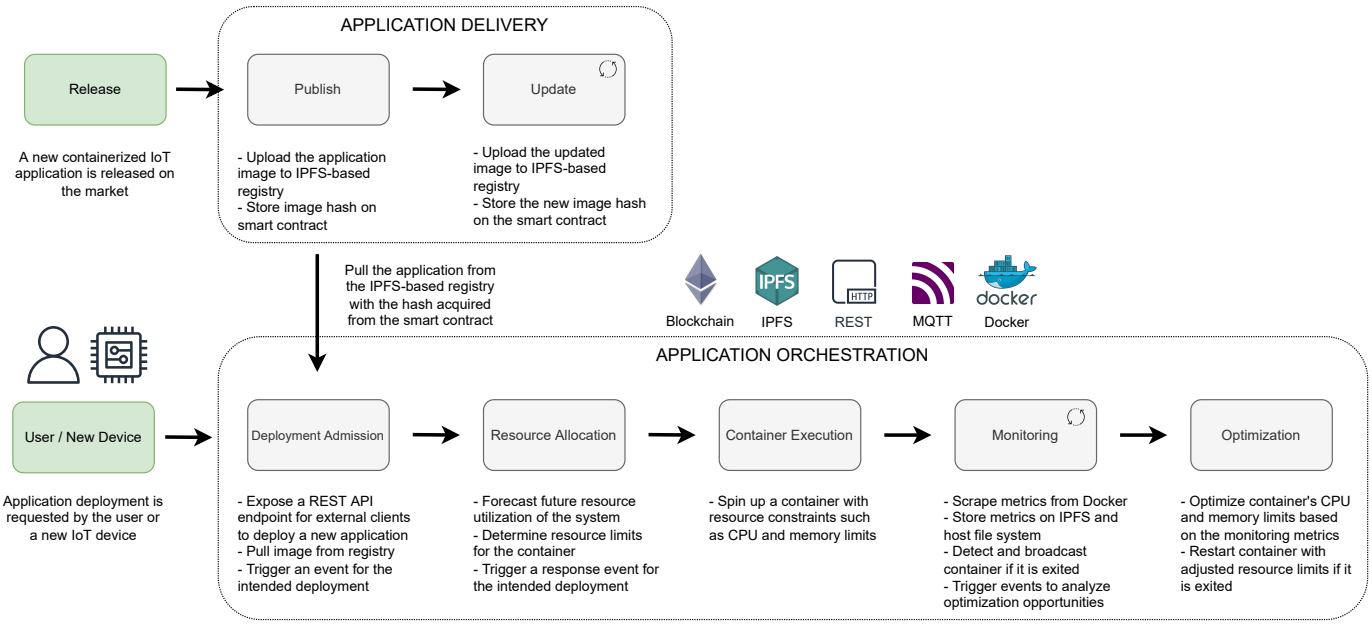

Fig. 2. Application delivery and orchestration workflows of the framework.

Then, the *Deployer* acquires the image hash and resource limits stored on the blockchain via the smart contract. After pulling the corresponding application image from the registry, a *deployment analysis request* is published with the resource limits on the *analysis* topic.

2) Resource Allocation: Each container is deployed with a set of CPU and memory limits specified in the smart contract. These limits are used in scaling the containers based on the status of other active deployments and vendor-defined resource limits. The *Analyzer* is subscribed to two different topics: *analyze* and *forecast*. It listens to *deployment analysis requests* and publishes a corresponding *deployment analysis response* on the *analyze* topic as shown in Table II. The analysis determines the feasibility of a new deployment with its resource requirements.

The analysis takes the future availability of the resources into account rather than their current values to ensure the long-lasting stability of the system. Therefore, a time-series forecasting analysis is conducted by the *Forecaster*. It is subscribed to the *forecast* topic and acts on a *forecast request* messages published by the *Analyzer*. The Autoregressive Integrated Moving Average (ARIMA) model is used to interpret the trends in time-series data and predict the future values with statistical analysis [20], [21]. The time-series data consist of the metrics collected and stored locally by the *Monitor* as described in Section III-B4. *Auto-Regression* and *Integrated* models of ARIMA are configured together to capture non-stationary patterns seen in each analyzed metric and forecast their values. The ARIMA model used in our implementation is with an order of (5,1,0), which can capture the short time trends in available metrics data. This configuration sets up a lag order of 5 to smooth the time-series data and a degree of differencing of 1. The use case scenarios depend on the environment where the edge unit is deployed. This work focuses on daily patterns which can be observable in smart homes or offices. In order to take this temporal characteristic into account, data points are aggregated hourly.

The prediction results for each container are its predicted CPU and memory utilization based on its lifecycle. These results, $P_{r,c}^{util}$, are published on the *forecast* topics so that *Analyzer* will be able to compute the feasibility of deploying a new container. To achieve this, it has to compute system availability, $P_{r,S}^{avail}$. Existing containers' resources are always prioritized over a new deployment. Hence, their resources, $\mathbf{L}_{r,c}^{current}$, are not lowered. Therefore, $\mathbf{P}_{r,c}^{util}$ is updated as $\mathbf{P}_{r,c}^{util}[t] := max(\mathbf{L}_{r,c}^{current}, \mathbf{P}_{r,c}^{util}[t])$ for each time point t. Predicted availability of each resource in the host $\mathbf{P}_{r,S}^{avail} := S_r^{total}$ is updated as $\mathbf{P}_{r,S}^{avail} -= max_t(\mathbf{P}_{r,c}^{util}[t])$ for each C_c and R_r. The symbol list is given in Table III.

The analysis request includes $L_{r,c}^{target}$ which can be one of the resource limit values: $L_{r,c}^{request}$, or $L_{r,c}^{base}$. In their absence, a default value is used. The *Analyzer* approves a deployment request if $L_{r,c}^{target} \leq P_{r,S}^{avail}$, meaning that the system will have enough resources to accommodate the incoming application deployment. Otherwise, the request is rejected.

3) Container Execution: Deployment analysis responses received by the *Deployer* can be twofold, corresponding to either a *deployment accept* or a *deployment cancel* as shown in Table II. 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.

4) Monitoring: Resource utilization metrics are acquired from Docker API by calling the relevant endpoints and *cgroup* stats in Linux systems by the *Monitor*. These metrics are published as a *Monitoring Result* on the *monitor* topic and

TABLE III
TABLE OF SYMBOLS.

Symbol	Description
R_r	Resource $r \in \{cpu, mem\}$
C_c	Container c
S	System
S_r^{total}	Total resources R_r of the system S
S_r^{avail}	Availability of resource R_r of the system S
$C_{r,c}^{util}$	Utilization of resource R_r by container C_c
$P_{r,S}^{avail}$	Predicted availability of resource R_r of the system S
$P_{r,c}^{util}$	Predicted utilization of resource R_r by container C_c
$P_c^{throttle}$	Predicted CPU throttling percentage of container C_c
$L_{r,c}^{current}$	Current limit definition of resource R_r for container C_c
$L_{r,c}^{target}$	Target limit definition of resource R_r for container C_c
$L_{r,c}^{request}$	Request limit definition of resource R_r for container C_c
$L_{r,c}^{base}$	Base limit definition of resource R_r for container C_c
$L^{throttle}$	Limit CPU throttling percentage
L_r^{buffer}	Buffer ratio for scaling of resource R_r
$L_{r,i}^{scale}$	Limit scaling amount of R_r for $i \in \{up, down\}$

also stored locally. After the retention time configured in the framework is reached, the metrics are uploaded to IPFS, and their hashes are stored on the blockchain via a smart contract as long-term storage. *Monitor* also identifies containers that have stopped prematurely. New *deployment request* messages are published based on the retry counts. The *Monitor* also initiates the optimization process for active containers.

5) Optimization: In this step, the framework dynamically adjusts the resource limits of active containers. The aim is to downscale underutilized containers and upscale containers approaching the limits. *Analyzer* determines whether the 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* interacts with the Docker engine if there is a need for optimization.

Optimization analysis starts with the calculation of $P_{r,S}^{avail}$, identical to the process explained in Section III-B2. Then, $L_{r,c}^{target}$ is derived from $L_{r,c}^{current}$ for CPU and memory, respectively as explained in the following paragraphs. After every approved optimization response, $P_{r,S}^{avail}$ is updated with $\mathbf{P}_{R,S}^{avail} := \mathbf{P}_{R,S}^{avail} - \Delta\mathbf{L}^{R,c}$ where $\Delta\mathbf{L}^{r,c} := \mathbf{L}_{r,c}^{target} - \mathbf{L}_{r,c}^{current}$.

Optimization for memory limit sets $L_{mem,c}^{target}$, which is bounded by system-defined minimum and maximum resource values, L_{mem}^{min} and L_{mem}^{max}. The upper limit ensures that a single container cannot allocate a very high share of system memory. The lower limit protects the container from scaling down indefinitely. The amount for scaling up and down, $L_{mem,up}^{scale}$ and $L_{mem,down}^{scale}$, are constants that can be configured in the framework. However, since $C_{mem,c}^{util}$ exceeding $L_{mem,c}^{current}$ is a risk that can kill the container, a margin is defined between $P_{mem,c}^{util}$ and $L_{mem,c}^{target}$ while scaling up. If $P_{mem,c}^{util}$ is less than $C_{mem,c}^{util}$, the *Analyzer* computes $\mathbf{L}_{mem,c}^{target}$ as $\mathbf{L}_{mem,c}^{current} - \mathbf{L}_{mem,down}^{scale}$ to scale down the container. After the $L_{mem,c}^{target}$ is set, in case of downscaling, the new limit is once again compared with $max(\mathbf{P}_{mem,c}^{util})$ to make sure that the container is not scaled below the allocated margin. Otherwise, $L_{mem,c}^{current}$ is returned, and memory is not scaled down any further. For scaling up, a similar approach is adopted.

Optimization for CPU limit sets $L_{cpu,c}^{target}$. If $max(P_{cpu,c}^{util})$ is greater than $L_{cpu,c}^{current}$, container is upscaled with $\mathbf{L}_{cpu,c}^{target} := \mathbf{L}_{cpu,c}^{current} + \mathbf{L}_{cpu,up}^{scale}$. In the opposite case, container is downscaled with $\mathbf{L}_{cpu,c}^{target} := \mathbf{L}_{cpu,c}^{current} - \mathbf{L}_{cpu,down}^{scale}$. Setting a limit on the CPU causes throttling in Docker [22]. As high amount of $C_c^{throttle}$ severely degrades the application performance, two extra precautions are taken while downscaling the containers that have already been throttling. Firstly, $P_c^{throttle}$ is compared against $L^{throttle}$ which is a constant defined in our framework. If $max_t(P_c^{throttle}[t])$ is greater than $L^{throttle}$, an *adjustedScale* amount where $adjustedScale := (\mathbf{L}_{cpu,up}^{scale} \times max_t(P_c^{throttle}[t]))/100$ is used instead of $\mathbf{L}_{cpu,up}^{scale}$ for a minor scale up. The second control is applied before returning $L_{cpu,c}^{target}$, which can cause throttling after a scale-down. In order to prevent sudden throttling, $\mathbf{L}_{cpu,c}^{target}$ is recalculated as $\mathbf{L}_{cpu,c}^{target} := max(P_{cpu,c}^{util}) \times \mathbf{L}_{cpu}^{buffer}$, which is set to 110% in our framework. Buffer ensures the application to have enough room for unexpected load spikes until the next optimization without causing unnecessary throttling. After this adjustment, it is possible that $L_{cpu,c}^{target}$ can be above $L_{cpu,c}^{current}$. In that case, any scaling operations are overturned by returning $L_{r,c}^{current}$ as the throttling was already below $L^{throttle}$.

C. Cluster Deployment

The proposed framework can be deployed on multiple edge devices as a cluster in the same network. This strategy supports not only load balancing across the connected hosts but also decentralized orchestration of deployments. The setup depends on MQTT bridges where brokers can automatically broadcast configured messages in the network. Each device's IP address is provided to each other broker to set up the bridges. However, it is also possible to set up an MQTT device discovery mechanism to automatically find other brokers on the network [23]. Shared topics, *monitor* and *deploy*, are configured and prefixed with the *cluster* keyword so that each component can differentiate internally or externally generated messages.

Deployer on each device is subscribed to both *monitor* and *cluster/monitor* topics. Key/value pairs of IP addresses and S_r^{avail} are generated by the *Deployer* for each device and updated with each *monitoring result* message. *Deployer* uses these pairs to track the available resources of all devices in the network. It is also subscribed to the *deploy* and *cluster/deploy* topics where it 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 its key/value pairs. Then, only the device that decides itself as the best candidate continues with the deployment by publishing a *deployment analysis request*. Cluster deployment follows the same steps as the regular deployment workflow. If S_r^{avail} of multiple devices are the same, each device marks the device with the smallest IP address for deployment.

IV. Experiments and Results

All experiments are carried out on Raspberry Pi 4 Model B with a 64-bit quad-core Cortex-A72 processing unit, 8GB LPDDR4-3200 SDRAM, and Broadcom BCM2711. A slice with 1 CPU and 1GB memory with disabled swap usage is created to emulate a resource-constrained device. Docker daemon's *cgroup* parent is assigned to this slice to limit the maximum allowed resource usage. CPU limits and utilization are given in CPU units. 1 CPU is equal to a single CPU core, and 50% of this core is represented as 500mCPU. The framework also exists on the same hardware. The CPU utilization is negligible for most components except for the *Forecaster* and IPFS. The *Forecaster* utilizes all available CPUs during forecasting to produce results as quickly as possible. IPFS utilizes 50-250 mCPU on average. It can be reduced by switching to a minimal setup, disabling unnecessary features, and limiting the peers. The memory footprint of the entire framework is currently 400MB, which can be reduced significantly by pruning and optimizing the libraries.

In order to create a controlled environment for the tests, containerized applications are prepared for each resource type. Five workload patterns proposed in [24] for container-based cloud applications are adapted to this work to generate workload patterns that represent different IoT application types:

1) Slowly rising/falling workload pattern
2) Drastically changing workload pattern
3) On-off workload pattern
4) Gently shaking workload pattern
5) Real-world workload pattern

The IoT applications can be classified as data-dominant and CPU-dominant. Hence, ten containerized applications are generated. Labels for each workload pattern are referenced with the class name. For example, a slowly rising/falling pattern for the data-dominant workload is identified as `Memory 1`. The same pattern for CPU-dominant workload is represented by `CPU 1`. The experiments are carried out on each group separately. This approach does not exclude the case where a CPU-dominant application coexists with a data-dominant application on the same device as different algorithms execute for memory optimization and CPU optimization. The maximum memory amounts in these workloads are 95MB, 95MB, 95MB, 80MB, and 95MB, respectively, as listed above. Similarly, 150m, 150m, 150m, 120m, and 140m represent the maximum CPU utilization. The experiments are carried out on each group separately. This approach does not exclude the case where a CPU-dominant application coexists with a data-dominant application on the same device as different algorithms execute for memory optimization and CPU optimization. The experimental results are presented in Figure 3 where solid and dashed lines show workloads and resource limits, respectively.

A. Orchestration of Multiple Containers on a Single Device

These experiments are carried on for a single device that has to host multiple containers with different limit settings and deployment scenarios. The first two scenarios consider application deployments in an extremely short period. This usually happens when the device is installed for the first time. The third scenario simulate deploying new applications on an already running device. The fourth scenario handles the case where the host is extremely low on resources

1) Sequential Deployment Requests with Ample Resource Limits: In this scenario, the application limits specified are above the expected peak memory usage of the workloads. For each data-dominant workload, $L^{request}_{mem,c} = 150MB$ and $L^{base}_{mem,c} = 100MB$. As shown in Figure 3(a.1-5), all five workloads start simultaneously. The memory 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 make utilization predictions for the upcoming optimization interval.

In CPU-dominant workloads, $L^{request}_{cpu,c} = 300m$ and $L^{base}_{r,c} = 100m$. Recall that if neither $L^{request}_{cpu,c}$ nor $L^{base}_{cpu,c}$ is accepted, the deployment is rejected because of the limited resource of the system. Note that total request makes 1500mcore which is more than the allocated CPU on the emulation hardware. Thus, the first deployment of the fourth workload with $L^{request}_{cpu,c}$ is rejected and deployed with $L^{base}_{cpu,c}$ in a second attempt by the framework, as shown in Figure 3(b.1-4). With the first four workloads running on the system, neither $L^{request}_{cpu,c}$ nor $L^{base}_{cpu,c}$ for the fifth workload can be satisfied (Figure 3(b.5)). The limits of the successfully deployed workloads are initially lowered drastically with a constant $L^{scale}_{cpu,down}$, which is later adjusted based on $P^{util}_{cpu,c}$ and $P^{throttle}_{c}$ resulting in more minor decrements.

2) Sequential Deployment Requests with Drastically Low Resource Specifications: This scenario assumes that all limits are given much lower than the actual requirements. In this experiment, we set $L^{request}_{mem,c} = 15MB$ and $L^{base}_{mem,c} = 10MB$. All five memory workloads fail to start for the first couple of tries, as shown in Figure 3(c1-5). After the initial two deployment attempts with no prior knowledge of the application, the framework begins assigning by rapidly increasing $L^{target}_{mem,c}$ based on $L^{base}_{mem,c}$ incremented by $L^{scale}_{mem,c}$ by several retry counts. Then, after the first four tries, Memory 1 and Memory 2 can be deployed as their resource usage is comparatively lower than the rest in the first one-third of their period. Other workloads keep getting killed as they immediately try to allocate more memory than allowed by $L^{current}_{mem,c}$. These burst restarts cause their $L^{target}_{mem,c}$ to increase so rapidly that this limit surpasses $C^{util}_{mem,c}$. However, once the containers are deployed, their limits are lowered by $L^{scale}_{down,mem}$ until a tighter fit based on $P^{util}_{mem,S}$ is found.

The deployment behavior of the CPU workloads vastly differs from the memory workloads. This is due to the absence of an error similar to *out of memory*. The limits are set as $L^{request}_{cpu,c} = 100m$ and $L^{base}_{cpu,c} = 50m$. All five deployments are accepted simultaneously due to their misconfigured $L^{request}_{cpu,c}$ values. These workloads typically exhibit $C^{util}_{r,c}$ higher than $L^{current}_{cpu,c}$. Consequently, they are fully throttled and containers exhibit delays in their workload which is shown with light red

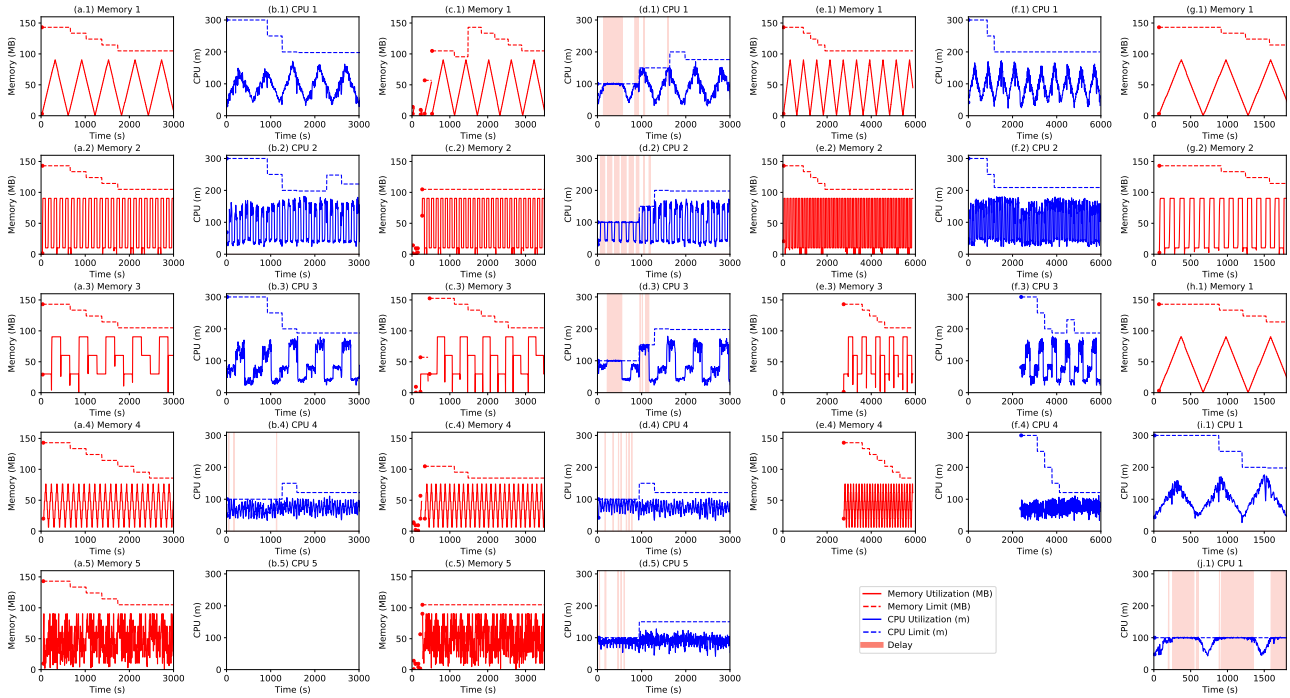

Fig. 3. Workload CPU/memory utilization/limit and delays observed in each experiment. Sequential deployments with ample resource specification: Memory (a), CPU (b). All workloads with drastically low resource specification: Memory (c), CPU (d). Interleaved deployment of workload pairs: Memory (e), CPU (f). Extreme resource constraints on the system: Memory (g,h), CPU (i,j).

vertical regions in Figure 3(d.1-5). The framework optimizes $L_{cpu,c}^{current}$ after a couple of optimization cycles, eventually matching the limits similar to those in Figure 3(b.1-5). Note that the delay is removed.

The deployment behavior of the CPU workloads vastly differs from the memory workloads, as shown in Figure 3(c.1-5). This is tied to the absence of an error similar to *out of memory*. Therefore, all five deployments are deployed with their misconfigured $L_{cpu,c}^{request}$ values. These workloads typically exhibit a $C_{r,c}^{util}$ higher than this $L_{cpu,c}^{current}$. Consequently, they are all immediately throttled. Fully throttled containers exhibit delays in their workload which is demonstrated in Figure 3(d.1-5) for workloads *CPU 1-5*. The framework optimizes $L_{cpu,c}^{current}$ responsible for the throttling after a couple of optimization cycles, eventually matching the limits similar to those in Figure 3(b.1-5).

3) Interleaved Deployment of Workload Pairs: This experiment studies the effects of introducing a new container to a stabilized system. For this purpose, after the first two workloads are optimized, deployment requests from the last two workloads will be received.

As shown in Figure 3(e.1-4) and Figure 3(f.1-4), 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 optimization of the newly introduced applications starts once enough data is collected for predictions to be performed.

4) Extreme Resource Constraints on the System: Some hosts can be running on extremely low resources. Pre-existing deployments can impose very tight constraints on the new requests. To simulate this environment for data-dominated applications, we set $S_{mem}^{avail} = 400MB$ and studied with the first three memory workloads. The first two workloads are deployed with their $L_{mem,c}^{request}$ (Figure 3(g.1-2)). The third workload cannot be deployed as there is not enough memory for the third workload's $L_{mem,c}^{request}$ or $L_{mem,c}^{base}$. Another experiment is run with a similar configuration with $S_{mem}^{avail} = 200MB$. As shown in Figure 3(h.1), only the first deployment is accepted as any subsequent deployments would require more memory than S_{mem}^{total}.

Similar results are obtained from the same experiments for CPU images. We set $S_{cpu}^{avail} = 350m$ and deployment requests for the first three workloads are sent. Only the first one is accepted with $L_{cpu,c}^{request}$ as shown in Figure 3(i.1). However, S_{cpu}^{avail} is not enough to deploy the remaining two workloads. These workloads are tried twice with $L_{cpu,c}^{request}$ and $L_{cpu,c}^{base}$ before they are rejected. Then, the experiment is rerun with even tighter resources where $S_{cpu}^{avail} = 100m$. A similar deployment pattern can be seen in in Figure 3(j.1). The only difference is that the first workload is deployed with $L_{cpu,c}^{base}$. This container exhibits delays at regular intervals;

nevertheless, it cannot be scaled up due to lack of S_{cpu}^{avail}.

B. Experiments with Cluster Deployment

This experiment is devised to observe the load balancing capabilities of the framework across multiple devices. Three edge devices with identical resources, S_r^{avail}, are used for this purpose. Device IP addresses increase from device 1 to 3, where *device 1* is the device with the smallest IP address. The first device receives sequential deployment requests from the first four memory workload patterns, i.e. Memory 1-4.

Firstly the MQTT broker on each device starts, and the framework is deployed. Then *device 1* receives Memory 1 workload. It deploys it on itself since its IP is the smallest. Following this deployment, all three devices synchronize their resource availability key/value pairs with *monitoring_result* messages published by *device 1*. With the second request, each device identifies *device 2* as the processor. The same steps are followed for the remaining two applications. The third application is deployed on *device 3* which evens up the resource availability of all three devices. The final application is deployed on *device 1* which has the smallest IP address. It should be noted that the cluster deployment does not introduce latency overhead during deployments. 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.

V. Conclusion

In this paper, we have presented an end-to-end solution for a seamless user experience between application developers and end-users on the IoT edge. The exploitation of virtualization and decentralization technologies transformed heterogeneous and resource-constrained edge gateways into vendor-agnostic hosts for IoT applications. Our framework lowers the edge computing footprint by sharing hardware and maximizing resource utilization with autonomous scaling and load balancing in the presence of multiple devices. The event-driven resource manager establishes a fair and reliable platform for all involved parties based on analysis of running services and forecasts of future requirements. Concerns about the authenticity of deliveries such as releases and upgrades are ensured through smart contracts and distributed files system. Specific aspects of the framework can be effortlessly extended due to its modular architecture of message queues and distributed components.

Acknowledgments

This research is partially supported by Boğaziçi University Scientific Research Projects No: 17A01P7 and Textkernel B.V.

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," *Future generation computer systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] Z. Tao, Q. Xia, Z. Hao, C. Li, L. Ma, S. Yi, and Q. Li, "A survey of virtual machine management in edge computing," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1482–1499, 2019.

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

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

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

[6] S. Muralidharan, G. Song, and H. Ko, "Monitoring and managing iot applications in smart cities using kubernetes," *Cloud Computing*, vol. 11, 2019.

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

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

[9] O. Ajayi, 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, Oct. 2021.

[10] T. Subramanya and R. Riggio, "Centralized and federated learning for predictive vnf autoscaling in multi-domain 5g networks and beyond," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 63–78, 2021.

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

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

[13] A. Pires, J. Simão, and L. Veiga, "Distributed and Decentralized Orchestration of Containers on Edge Clouds," *Journal of Grid Computing*, vol. 19, no. 3, p. 36, Jul. 2021.

[14] L. Cui, 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, Feb. 2021.

[15] C. Cicconetti, 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, Jun. 2021.

[16] "IPDR: InterPlanetary Docker Registry," accessed in February 2022. [Online]. Available: https://github.com/ipdr/ipdr

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

[18] G. Nalin, "Orchestration of smart objects with MQTT for the Internet of Things," accessed in February 2022. [Online]. Available: http://tesi.cab.unipd.it/44964/

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

[20] "Time-series | Stata," accessed in February 2022. [Online]. Available: https://www.stata.com/features/time-series/

[21] "Arima — statsmodels," accessed in February 2022. [Online]. Available: https://www.statsmodels.org/stable/generated/statsmodels.tsa.arima.model.ARIMA.html

[22] "Runtime options with memory, cpus, and gpus," accessed in February 2022. [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/

[23] D. Rende, "Dag Rende: Find the MQTT broker without an IP address," accessed in February 2022. [Online]. Available: http://dagrende.blogspot.com/2017/02/find-mqtt-broker-without-hard-coded-ip.html

[24] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *The Computer Journal*, vol. 62, no. 2, pp. 174–197, 2019.