GAIKube: Generative AI-Based Proactive Kubernetes Container Orchestration Framework for Heterogeneous Edge Computing

Babar Ali, Muhammed Golec, Subramaniam Subramanian Murugesan, Huaming Wu, *Senior Member, IEEE*, Sukhpal Singh Gill, Felix Cuadrado, and Steve Uhlig

***Abstract*—Containerized edge computing emerged as a preferred platform for latency-sensitive applications requiring informed and efficient decision-making accounting for the end user and edge service providers' interests simultaneously. Edge decision engines exploit pipelined knowledge streams to enhance performance and often fall short by employing inferior resource predictors subjected to limited available training data. These shortcomings flow through the pipelines and adversely impact other modules, including schedulers leading to such decisions costing delays, user-experienced accuracy, Service Level Agreements (SLA) violations, and server faults. To address limited data, substandard CPU usage predictions, and container orchestration considering delay accuracy and SLA violations, we propose a threefold GAIKube framework offering Generative AI (GAI)-enabled proactive container orchestration for a heterogeneous edge computing paradigm. Addressing data limitation, GAIKube employs DoppelGANger (DGAN) to augment time series CPU usage data for a computationally heterogeneous edge cluster. In the second place, GAIKube leverages Google TimesFM for its long horizon predictions, 4.84 Root Mean Squared Error (RMSE) and 3.10 Mean Absolute Error (MAE) against veterans Long Short-Term Memory (LSTM) and Bidirectional LSTM (Bi-LSTM) on concatenated DGAN and original dataset. Considering TimesFM quality predictions utilizing the DGAN extended dataset, GAIKube pipelines CPU usage predictions of edge servers to a proposed dynamic container orchestrator. GAIKube orchestrator produces container scheduling, migration, dynamic vertical scaling, and hosted application model-switching to balance contrasting SLA violations, cost, and accuracy objectives avoiding server faults. Google Kubernetes Engine (GKE) based real testbed experiments show that the GAIKube orchestrator offers 3.43% SLA violations and 3.80% user-experienced accuracy loss with zero server faults at 1.46 CPU cores expense in comparison to industry-standard model-switching, GKE pod scaling, and GKE *optimized* scheduler.**

***Index Terms*—Edge computing, generative AI, container migration, vertical scaling, Kubernetes, service level agreement.**

Received 2 July 2024; revised 30 October 2024; accepted 24 November 2024. Date of publication 2 December 2024; date of current version 9 April 2025. B. Ali is supported by the Ph.D. Scholarship at the Queen Mary University of London. M. Golec is supported by the Ministry of Education of the Turkish Republic. H. Wu is supported by the National Natural Science Foundation of China (No. 62071327). F. Cuadrado has been supported by the HE ACES project (No. 101093126). The associate editor coordinating the review of this article and approving it for publication was J. Kang. *(Corresponding author: Huaming Wu.)*

Babar Ali, Muhammed Golec, Subramaniam Subramanian Murugesan, Sukhpal Singh Gill, and Steve Uhlig are with the School of Electronic Engineering and Computer Science, Queen Mary University of London, E1 4NS London, U.K. (e-mail: b.ali@qmul.ac.uk; m.golec@qmul.ac.uk; s.subramanianmurugesan@qmul.ac.uk; s.s.gill@qmul.ac.uk; steve.uhlig@qmul.ac.uk).

Huaming Wu is with the Center for Applied Mathematics, Tianjin University, Tianjin 300072, China (e-mail: whming@tju.edu.cn).

Felix Cuadrado is with the School of Telecommunications Engineering, Universidad Politécnica de Madrid, 28040 Madrid, Spain (e-mail: felix.cuadrado@upm.es).

Digital Object Identifier 10.1109/TCCN.2024.3508771

I. Introduction

EXPLOSIVE growth of the Internet of Things (IoT) devices is a key enabler for producing a diverse range of services and applications. Machine Learning (ML) and Deep Learning (DL) applications are constantly improving end users' experiences generating notably accurate inferences in the fields of spanning healthcare, transportation, surveillance, and computer vision [1]. End users are mainly interested in quick and accurate responses. To address latency challenges, edge computing infiltrated the computing continuum hosting ML/DL applications in closer proximity to data, enticing users to offload compute-intensive tasks to edge. It extends cloud service closer to users, which is mainly a distributed, heterogeneous and resource-constrained computing paradigm [2].

A. Opportunities and Challenges

The inevitable confluence of edge and containerization brings tremendous opportunities in the computing landscape enhancing reliability, latency, ease of application management, lesser overhead, and bandwidth conservation [3], [4]. However, it introduces critical management challenges requiring efficient solutions to manage service provider cost, user-experienced accuracy, latency, and edge server health. The computational heterogeneity of edge servers (CPU, memory), Kubernetes containers (CPU, memory), and multiple accessible ML/DL model versions add up to existing challenges [5]. Model-Switching [6] offers to change ML/DL model versions responding to dynamic load and high accuracy demands. However, static resource provisioning incurs either cost or SLA violations [7]. Thus, Model-Switching assisted vertical container scaling at edge computing offers a multitude of benefits with greater responsibility on scheduler. For example, a decision for all the containers to employ the best accuracy model in the lowest possible container cores extremely overloading a few edge servers can increase accuracy and conserve cost but it can damage edge servers and increase Service Level Agreement (SLA) violations [8]. Therefore, an efficient and

dynamically adaptive orchestrator for containers at the edge is required.

It is financially infeasible to deploy powerful servers at the edge [9]. Sub-optimal and reactive solutions can cause resource contention and potential downtimes ultimately incurring monetary cost and wasted computations [9]. To avoid system faults, the orchestration unit must be equipped with a server usage predictor to assist in producing proactive and dynamic decisions [10]. Moreover, high-precision predictors are inherently data-hungry requiring diverse training data while data limitation can result in average models [11]. Generative Adversarial Networks (GAN) attempted to address this limitation and offer to produce synthetic data employing state-of-the-art ML models under the hood [12], [13], [14]. DoppelGANger (DGAN) [15] and TimeGAN [16] are prominent GANs characterizing time series data. Furthermore, with the shorter prediction sequences limitation of TimeGAN, this work employs DGAN for its salient features [17]. The next section presents the motivation for Generative Artificial Intelligence (GAI) based container orchestration.

B. Motivation

Firstly, GAI can generate vast amounts of high-quality data while significantly cutting down on data collection time and cost [18]. Original data can be extended with unseen and realistic data encompassing a wide range of workload patterns leading to the production of accurate and robust models [19]. Secondly, researchers are producing multiple variants of ML/DL applications differing in accuracy and computation resource demands [20]. Fig. 1a represents the SLA violation percentage comparison of Yolo5 application nano, small, and medium versions hosted in 0.5, 1, and 2 cores containers. SLA violations for each model are reduced with the increment of provisioned cores. It can be concluded that under-provisioning increases SLA violations and over-provisioning incurs costs [21]. Moreover, Fig. 1b shows SLA violations against container cores with dynamic model-switching subjected to violation rate. Two cores container offering the lowest SLA violations with the highest accuracy. Half core suffers from under-provisioning while one core changed models resulting in improved accuracy and a higher violation rate. Thus, model-switching requires dynamic scaling for accuracy, cost, and SLA violations. Finally, GKE offers container placement in *balanced* and *optimized* modes. Prior one distributes containers evenly among server nodes, while *optimized* mode opts to overload servers. The balanced mode can lead to higher cost with more active nodes while optimized one can suffer from server damage where putting load more than 75-80% impacts machine performance [22], [23]. Thus, there is a need for an efficient and proactive container placement in *optimized* mode to avoid performance degradation.

C. Contributions

Considering the challenges of user-experienced accuracy, latency, service provider cost, SLA violations, edge server health, and data limitation, in this paper we propose the GAIKube container orchestration framework offering threefold contributions. It utilizes GAN to produce CPU usage data for heterogeneous edge servers. Leveraging the extended dataset, GAIKube employs a decoder-only time series predictor for proactive scheduling. Finally, pipelining predictions into a dynamic and adaptive scheduler responsible for heterogeneous edge server cluster management. The following are key contributions of this work.

- GAIKube employs DGAN [15] to produce time series CPU usage data of heterogeneous 2, 4, and 6 cores edge servers utilizing Bitbrains [24] dataset historical records
- We adopted Google TimesFM [25] offering minimal RMSE and MAE errors against LSTM and Bi-LSTM for six timestep ahead CPU usage forecasts of heterogeneous edge nodes on the extended Bitbrain dataset
- GAIKube proposes a proactive scheduler to dynamically modify container resources, switch YOLO models, and migrate containers exploiting pipelined predictions and SLA violation rates. These decisions account for accuracy, latency, cost, and edge server health
- Experimental results in the GKE testbed show GAIKube outperforms the default GKE *optimized* scheduler [26], dynamic pod scaler [27] and model-switching [6] offering 3.43% SLA violations, 3.80% accuracy loss, and zero server faults at 1.46 CPU cores expense.

The rest of the paper is organized as follows. Section II highlights existing research works conducted in this paradigm followed by critical analysis. The accuracy, cost, and server utilization problem is formulated in Section III. GAIKube framework is detailed in Section IV. Performance evaluation and results are discussed in Section V. Section VI concludes this work and highlights future directions.

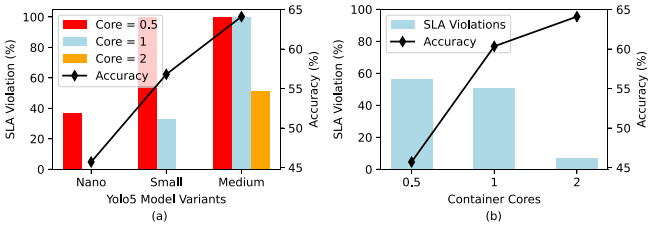


Fig. 1. (a) Latency SLA violations for various cores and model versions against model accuracy with 700ms SLA threshold. (b) Container cores against SLA violations for Model Switching.

II. Related Work

Tuli et al. [28] proposed a PreGAN model for proactive container migrations in heterogeneous edge computing. The generator employed faulted edge server classification to produce modified scheduling decisions. The discriminator selects between the original and the generator decisions targeting various quality parameters. PreGAN improved energy and SLA violations, However, this work does not offer dynamic scaling and model-switching for user-experienced accuracy. In another work, Ray et al. [29] proposed a proactive VM migration technique to avoid server faults. The authors modeled the profit and migration cost problem using Integer Linear Programming (ILP) to redistribute VMs from faulty to healthy nodes. This

TABLE I
COMPARISON OF GAIKUBE WITH EXISTING WORKS

Work	Edge/ Cloud	GAI	Proactive	Heterogeneity	Method	Objective	Model Switching	Vertical Scaling	Real Testbed	Performance Parameters				
										CPU Util.	Cost	Accuracy	SLA Violations	Migrations
[28]	Edge	✓	✓	✓	GRU+GAN	Fault Avoidance			✓	✓	✓		✓	✓
[6]	Cloud		✓		Heuristic	Accuracy, Cost	✓		✓			✓		
[7]	Cloud		✓		LSTM + ILP	Delay, Cost, Accuracy	✓		✓		✓	✓	✓	
[29]	Cloud		✓	✓	ILP	Profit, Migration Cost					✓			✓
[32]	Cloud	✓		✓	GAN + PPO	Task waiting time								
[27]	Cloud/Edge				GKE VPA	Cost, Workload		✓	✓		✓			
[26]	Cloud/Edge			✓	GKE Optimized Scheduler	Cost, Utilization		✓	✓	✓	✓			
[30]	Cloud		✓		LSTM+Heuristic	CPU cost		✓	✓		✓			
[31]	Cloud		✓	✓	Bi-LSTM+Heuristic	Fault Avoidance			✓			✓	✓	
GAIKube	Edge	✓	✓	✓	DGAN+TimesFM+ GAIKube Orchestrator	Accuracy, Utilization SLA violatons	✓	✓	✓	✓	✓	✓	✓	✓

Columns GAI, Proactive, and Heterogeneity show the employment of Generative AI, resource predictor, and computational heterogeneity (CPU, memory). Column Real Testbed classifies simulation and real-world experiment platforms

work outperformed its counterparts however, they did not consider IoT applications and dynamic scaling.

Zhang et al. [6] proposed a model-switching technique to modify the hosted DL application version in response to dynamic workload and accuracy requirements employing profiled statistics. It improved user-experienced accuracy in comparison to individual DL models. However, this work is evaluated in homogeneous settings without offering dynamic scaling. Salmani et al. [7] proposed the idea of selecting a set of model variants employing model-switching to meet accuracy, delay, and cost objectives. It utilized ILP to identify the model variants and container replicas to respond to dynamic workload. Kubernetes experiments show reduced cost and improved accuracy. However, this work overlooked heterogeneity and employed the default Kubernetes scheduler for container placements. Wang et al. [30] proposed a proactive heuristic Vertical Pod Autoscaler (VPA) employing LSTM predictions to conserve CPU cores. Containers are rescaled when the predicted usage falls outside the lower and upper bound thresholds. However, the predictive performance can be enhanced with the latest predictive models and extended datasets. In another work, Tran et al. [31] employed Bi-LSTM on the Bitbrains dataset for CPU prediction to avoid server faults. Bi-LSTM predictions are pipelined to a container migration framework that proactively migrates containers to predicted healthy servers. However, this work can be analyzed in heterogeneous settings with dynamic scaling.

GARLSched [32] algorithm accelerates the Proximal Policy Optimization (PPO) scheduler learning process employing a GAN expert. Discriminator classification performance is enhanced by distinguishing GAN expert and PPO actions. Simulation results show GARLSched improved task waiting time in scalable settings however, this work lacks real testbed evaluation. GKE VPA [27] and GKE *optimized* scheduler [26] are the industry-standard production scalers and container schedulers of the Google platform. VPA dynamically updates container CPU and memory resources employing historical records and monitored performance responding dynamic workload and costs. In parallel, the GKE *optimized* scheduler aggressively maximizes server utilization by placing containers to the maximum limit. This approach improved utilization and minimized cost at the possible expense of server faults.

A. Critical Analysis

Table I presents a detailed comparison of existing research conducted in this paradigm. Works [28], [32] offered GAN solutions for scheduling. PreGAN aligns closely with our work however, it does not entertain model-switching and dynamic scaling. GARLSched worked on improving task waiting time in the simulated settings and overlooked significant orchestration parameters. Most of the works employed prediction algorithms in computationally heterogeneous clusters. Objectives include fault avoidance, cost, accuracy, utilization, etc. where only [6], [7] offered dynamic model-switching. However, both of these works not only overlooked vertical scaling but also employed the default Kubernetes orchestrator. Vertical scaling is offered by [26], [27], [30] but these works address specific objectives without considering multiple ones together. The cost metric represents the cost considered in terms of CPU cores, energy, or nodes consolidated by the considered works. It can be seen that none of the works have considered a diverse range of parameters and objectives together accounting for both the end users and edge service provider interests simultaneously.

Given the heterogeneous and resource-constrained edge environment, there is a need for a framework capable of responding to dynamic workload, SLA violations, and user-experienced accuracy avoiding edge server faults. Furthermore, it should cope with the limited available training data to generate quality resource predictions. Therefore in this work, we are proposing GAIKube which offers threefold contributions. It employs DGAN to augment CPU usage data for heterogeneous edge cluster nodes. Utilizing the extended dataset GAIKube produces quality CPU usage predictions using TimesFM. Finally, it exploits pipelined predictions to produce container scheduling, migration, YOLO application model-switching, and dynamic vertical scaling decisions.

III. PROBLEM FORMULATION

This section formally describes the problem of accuracy, cost, and utilization of heterogeneous edge server nodes. We consider a computationally heterogeneous edge cluster with nodes $N = \{n_1(\theta,\eta), n_2(\theta,\eta), \ldots, n_K(\theta,\eta)\}$ offering distinct CPU (θ) and memory (η) resources. Containers can request dynamic CPU (θ^r) and memory (η^r) resources from edge nodes to host ML/DL applications. Let $P =$

$\{p_1(\theta^r, \eta^r), p_2(\theta^r, \eta^r), \ldots, p_J(\theta^r, \eta^r)\}$ be this set of pods representing distinct and heterogeneous resource requests. At each point in time i, every node has a certain utilization of its resources and a set of all the nodes utilization be $U = \{u_1(\theta, \eta), u_2(\theta, \eta), \ldots, u_K(\theta, \eta)\}$. CPU and memory utilization of a node k at time i come from the total requested to the maximum allocatable resources [24].

$$u_k(i, \theta, \eta) = \left(\frac{\sum n_k(\theta^r)}{n_k(\theta^{\max})}, \frac{\sum n_k(\eta^r)}{n_k(\eta^{\max})} \right). \tag{1}$$

DL applications can have multiple model versions differing in accuracy. Let $m \in \{1, 2, \ldots, M\}$ be the set of model versions and each version has unique associated accuracy a_m. At any time interval i, there can be only one active model in a given container p by:

$$\sum_{m=1}^{M} x_{m,p}(i) = 1, \tag{2}$$

where $x_m \in \{0, 1\}$. The accuracy is subjected to the active version for a container p and the accuracy of the system at time i be:

$$A(i) = \sum_{p \in P} \sum_{m=1}^{M} a_m x_{m,p}(i). \tag{3}$$

Active model in any container can be switched to another version subjected to system state and requirements and it can be stated as:

$$s(i) = \begin{cases} 1 & \text{if } x_{m,p}(i-1) \neq x_{m,p}(I), \\ 0, & \text{otherwise}. \end{cases} \tag{4}$$

While all the model switches in interval i in total containers P can be:

$$S_P(i) = \sum_{p \in P} \left| x_{m,p}(i) - x_{m,p}(i-1) \right|. \tag{5}$$

To measure the overall average accuracy of the system, user traffic served by each model is required. Each container p entertains a fraction user traffic λ_p of total traffic λ. Since the model may be switched at intervals on each container, this leads to the possibility that traffic λ_p is further split between model versions $\lambda_{m,p}$. Thus, the average accuracy aa_p of container p over all the intervals I comes from the traffic entertained by each model to its accuracy [6].

$$\text{aa}_p = \sum_{i \in I} \sum_{m=1}^{M} \left(\frac{\lambda_{m,p}(i)}{\lambda_p} \right) a_m x_{m,p}(i). \tag{6}$$

Now the average or Mean Accuracy (MA) of the system over all the containers, models, and the traffic of each model can be estimated using Eq. (6):

$$\text{MA} = \sum_{p \in P} \left(\frac{\lambda_p}{\lambda} aa_p \right). \tag{7}$$

As stated earlier each container requests for resources from edge server nodes. Thus, the total system cost C at i comes from the number of CPUs requested by containers [7].

$$C(i) = \sum_{p=1}^{P} p(\eta^r). \tag{8}$$

Each application model responds to user queries and returns the response. Considering this scenario of varying model accuracy and their requirement for dynamic CPU, this work estimates the SLA Violations (SLAV) to improve end-user experience. SLA is violated when a request processing time T_e exceeds the threshold SLA_{th}. SLA Violation of a given model in container p is calculated by:

$$\text{SLAV}_{m,p} = \begin{cases} 1 \text{ if } T_e > SLA_{th}, \\ 0 \text{ otherwise}. \end{cases} \tag{9}$$

The SLAV Rate (SVR) of a given container for interval i can be estimated using the number of violations to the traffic observed in this interval.

$$\text{SVR}_{m,p}(i) = \frac{SLAV_{m,p}(i)}{\lambda_{m,p}(i)}. \tag{10}$$

At the end of time interval $i-1$, metrics are carried forward to interval i including the cost of the system, current utilization of edge server nodes, violation rate observed for each of the containers, and accuracy of active models. Considering these, the objective is to maximize accuracy A, maximize edge server CPU utilization U, and minimize the violation rate VR in each interval i given by:

$$\max: \quad MA + U - SVR \tag{11}$$

$$\text{s.t.:} \quad T_e \leq SLA_{th}, \tag{11a}$$

$$\theta^r \leq \max(N(\theta)), \tag{11b}$$

$$\theta^r \leq \theta_{th}, \tag{11c}$$

$$A(i) \leq \sum_{p \in P} \max_{m \in M} \left(a_m x_{m,p}(i) \right), \tag{11d}$$

$$\sum_{p \in P} p(\theta_r^r, \eta_r^r) \leq N(\theta, \eta), \quad \forall k. \tag{11e}$$

The first constraint ensures SLA violation does not occur while the second constraint states that a container can never request CPU than the maximum available CPU in the cluster. The third constraint limits the container requested CPU from the defined CPU threshold θ_{th}. The accuracy can never exceed the maximum achievable. Lastly, the resources used by all the containers in a node should stay below the node capacity.

IV. The GAIKube Framework

This section describes the system architecture, machine learning employed models, and techniques used at each stage of this framework.

A. System Architecture

Fig. 2 shows the system architecture highlighting the DGAN generator, TimesFM predictor, GAIKube scheduler, GKE heterogeneous edge datacenter, IoT users, and the framework flow. GAIKube framework entertains computationally heterogeneous edge servers and containers hosting a variety of YOLO images shown in the data center module of Fig. 2. Beginning with decoupling historical CPU usage data into individual series followed by DGAN training. The trained models generate new records followed by custom metadata

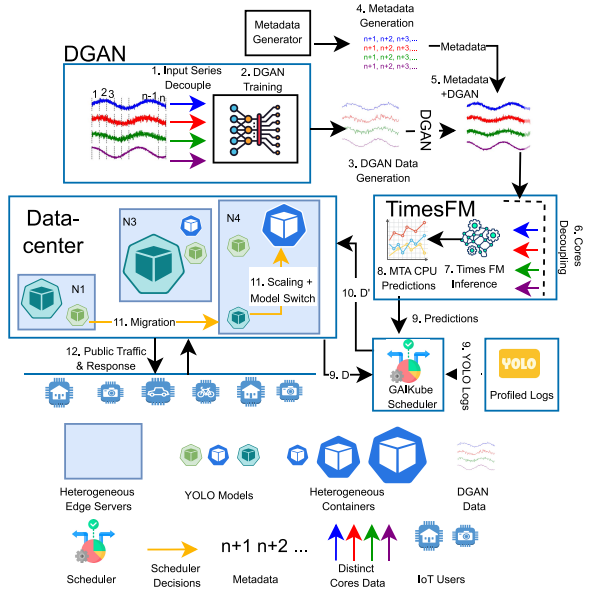


Fig. 2. GAIKube System Architecture representing DGAN, TimesFM Predictor, Scheduler, GKE Heterogeneous Testbed.

(MDT) creation. Both the DGAN data and MDT are concatenated for individual series and appended to the original dataset in step 5 of Fig. 2. These extended records are pipelined to the TimesFM predictor to generate Multi Timestep Ahead (MTA) predictions for heterogeneous edge servers. TimesFM differentiates the input dataset on unique cores. Moreover, TimesFM is tuned with the desired prediction length followed by MTA predictions pipelined to the GAIKube scheduler in step 9. MTA predictions are conducted in an auto-regressive mode where the predicted timesteps are appended to the training data to extract forecast CPU usage for the next step.

The scheduler is responsible for producing proactive migration, scaling, and model-switching decisions leveraging TimeFM predictions and YOLO profiled dataset. In addition, it leverages the edge data center state including container placements, selected YOLO models, provisioned container cores, observed SVR, and edge server utilization. The scheduler produces the latest decision D' followed by its implementation in the edge data center. Possible migration, model-switching, and scaling are shown in the data center in Fig. 2. Finally, public traffic is directed to the IP and port exposed by containers.

B. Datasets

This section describes the Bitbrains and custom YOLO datasets employed in this research work. GAIKube leverages heterogeneous 2, 4, and 6 core machine records of Bitbrains for edge servers CPU prediction. The YOLO profiled dataset enriches the scheduler offering processing time of distinct versions hosted in 0.5, 1, and 2 cores containers.

1) Bitbrains: Bitbrains [24] is an open-source dataset comprising CPU, memory, disk, and network features of distributed data center-hosted financial applications of banks, credit card operators, and insurance companies. It is commonly employed for edge/ fog settings [33] subjected to dynamic

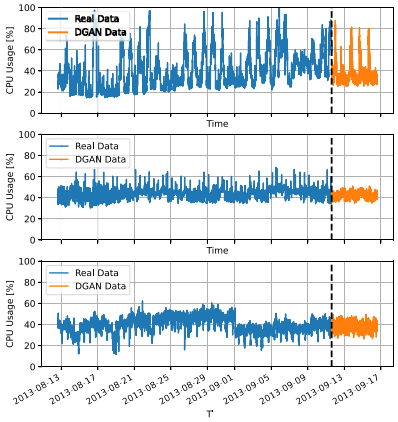


Fig. 3. Real and Doppler GAN CPU usage data of 2, 4 and 6 core VMs.

TABLE II
SYMBOLS AND DEFINITIONS

Symbol	Definition
N	Set of heterogeneous edge nodes
P	Set of containers and/ or pods
U	Set of CPU and memory utilization of N
I	Total time intervals
$i\ \varepsilon\ I$	Time interval i from I
$C(i)$	Cost in terms of container cores in i
D'	New scheduling decision
M	Yolo5 model versions
$Y(i)$	Migration count in i
$S(i)$	Total model switches in i
θ	CPU of edge node
θ^r	CPU requested by pod p
η	Memory/ RAM of edge node
η^r	Memory/ RAM requested by pod p
λ	Total user traffic in I
λ_p	Total traffic of pod p in I
$\lambda_{m,p}$	Total traffic of pod p and model m in I
μ	CPU predictions
ω	Prediction window length
θ_{th}	CPU usage threshold for edge nodes
SLA_{th}	SLA threshold of Yolo5 execution time
a_m	Accuracy of model version m
$A(i)$	Accuracy of active model version in pod at i
T_e	Execution time

workload characteristics. Accounting for the requirement of highly dynamic workloads of heterogeneous servers, we analyzed 1250 VMs from *fastStorage* class and selected machines 983, 980, and 943 of 2, 4, and 6 cores, respectively. These selected machines showed regular patterns of CPU usage percentage over a month as shown in Fig. 3. 2 core VM exhibits regular spikes, while 4 and 6 core VMs show balanced and consistent usage patterns. The rest of the machines are either underutilized or exhibit non-uniform spikes. Considering CPU as the key factor in Google Cloud platform cost calculation [34], we considered *CPU Usage [%]* in this work. Initial data analysis showed records are logged at irregular intervals with missing values at certain timestamps for each core. We addressed these limitations by *resampling* and *forwardfill* at a 5-minute interval. These processed records of 2, 4, and 6 core VMs are concatenated, sorted, and ingested to the TimesFM.

2) Yolo Profiling: We employed You Look Only Once 5 (Yolo5) [35] DL application, which performs detection and classification on images and live video streams, identifies objects by drawing bounding boxes, and shows confidence using probability. Yolo5 has multiple versions including

Algorithm 1 DGAN Data Generation

1: **Input**: Historical Bitbrains Data ρ
2: **Output**: Combined New and Historical Records ρ
3: **procedure** DATA GENERATION
4: ρ = load data
5: $\rho = \text{clean}(\rho)$
6: $f[1, 2, \ldots, F] = \rho[\theta], \rho[\theta], \rho[\theta], \ldots \rho[\theta]$
7: **for** $j = 1$, $j{+}{+}$, while $j \leq F$ **do**
8: $f = f[j]$
9: $\phi = train(f, H)$
10: $f^* = \phi(f)$
11: mdt = datetime(begin, end)
12: $f^* = f^*[mdt]$
13: $f = f + f^*$
14: $\rho^*[f] = f$
15: **end for**
16: $\rho = \rho^*[1] + \rho^*[2] + , \cdots , \rho^*[F]$
17: $return\ \rho$
18: **end procedure**

Yolo5n (nano), Yolo5s (small), Yolo5m (medium), etc. differing in accuracy and computational resources requirements. Considering the constrained resources at the edge and the timely response requirement of end users, we adopted nano, small, and medium versions for conducting inferences in less than a second. Given that CPU cores substantially influence task processing time, we profiled a custom dataset of the YOLO model version and hosted container CPU cores. A Docker[1] image for each nano, small, and medium version is created and deployed in 0.5, 1, and 2 cores containers hosted in the GKE London region. These containers can receive public traffic on port 5000 using Flask,[2] detect and label the provided image employing the model version, and return the results to the user. Finally, the received response is stored on the client side. The response includes the detected image, processing time, propagation time, model version, container core, etc. These profiled metrics are utilized by the scheduler for decision-making shown in step 9 of Fig. 2. Details related to the working of Kubernetes are given in our previous work [4].

C. AI/ML Models

This section details DGAN and TimesFM models utilized in the GAIKube framework.

1) DoppelGANger CPU Generator: DGAN [15] is a novel time series data generation model aiding the traditional generator and discriminator with per-feature scaling to mitigate mode collapsing in GANs. It incorporates multi-step RNN predictions capturing temporal sequences to produce long-term data. However, DGAN fails to generalize multiple series simultaneously. Despite MDT generation claims, it was unable to capture time series MDT patterns and produced MDT at irregular timestamps. Finally, DGAN is sensitive to the frequency and number of future samples. Experimental analysis with 72, 144, 288, and 2016 future samples showed that DGAN collapses with prolonged future sample generation. On the contrary, short-term future samples resulted in high fluctuations failing to grasp the trend.

Considering these challenges, we split the Bitbrains dataset into individual 2, 4, and 6 core series shown in Algorithm 1 step 6 and Fig. 2 step 1. GAIKube trains a separate model for each server core shown in Algorithm 1 step 9, where H is the DGAN hyperparameter given in Table III. Moreover, we already had decoupled the date-time MDT. Leveraging future sample analysis, we tuned DGAN to generate one-day data f^* comprising 288 CPU usage samples for each core at a 5-minute interval. To make freshly generated logs consistent with the Bitbrains, GAIKube generates MDT for the number of samples per day and the number of days followed by logs concatenation represented in steps 11-13 of Algorithm 1. Finally, all the core logs are combined to produce an extended single dataset ρ in Algorithm 1 step 15 and Fig. 2 step 5. Fig. 3 exhibits original and DGAN generated data of employed 2, 4, and 6 cores heterogeneous servers from Bitbrians. It can be seen that the DGAN data follows similar trends for each core.

TABLE III
DGAN HYPERPARAMETERS

Model	Hyperparameters
DGAN	max_sequence_len: 288, feature_noise_dim: 32, num_records:5, attribute_num_layers: 3, use_attribute_discriminator: true, normalization: 1, apply_feature_scaling: true, attribute_loss_coef: 10, generator_learning_rate: 0.0001, discriminator_learning_rate: 0.0001, batch_size: 64, attribute_discriminator_learning_rate: 0.0001

2) TimesFM CPU Predictions: Leveraging tokens, positional encoding, and self-attention in the decoder-only mode, Times Foundation Model (TimesFM) [25] addresses RNN limitations of mishandling long-term past temporal dependencies. It equates tokens as the input time series data patch and implements positional encoding to capture temporal dependencies followed by self-attention enabling the model to learn the relation among different patches. GAIKube pipelines Bitbrains extended dataset from DGAN for heterogeneous VMs. To better capture the trends, TimesFM's internal architecture enables each core to be processed as a unique series. Addressing the limited visibility into the future given the uncertain and dynamic edge computing paradigm, GAIKube exploits TimesFM long-horizon forecasts producing half-hour or six timesteps ahead CPU usage predictions for each server.

TimesFM prediction procedure is shown in Algorithm 2. TimesFM requires historical data ρ, prediction length ω, and unique VM cores. As stated earlier, we utilize the DGAN returned dataset. After loading and splitting the dataset into 80% and 20% training and testing sets, we calculate the number of iterations for auto-regressive model predictions in step 6. Predictions μ are made using hyperparameters H given in Table VI. TimesFM employs previous outputs in an auto-regressive mode requiring predicted timesteps to be incorporated into the training data to generate predictions for the next iteration as shown in step 10. Finally, TimesFM CPU usage predictions of edge servers are ingested into the scheduler for informed decision-making as shown in Fig. 2 step 9.

D. Scheduler

GAIKube scheduler is responsible for container migration, scaling, and YOLO model-switching decisions to achieve contrasting accuracy, utilization, and SVR objectives given

[1] https://www.docker.com/
[2] https://flask.palletsprojects.com/en/3.0.x/

Algorithm 2 TimesFM Prediction

Input: Historical Data ρ, Prediction Window ω, Unique VM Cores vm_core
Output: Predictions μ
procedure TIMESFM
  ρ = load data
  trainset, testset = train_test_split()
  $iter = (|testset|/\omega \times vm\_core)$
  $\mu^* = []$
  for $j = 1$, $j{+}{+}$, while $j \leq iter$ **do**
    $\mu = timesfm.predict(trainset, H)$
    $trainset = trainset + trainset[\mu]$
    $\mu^*[j] = \mu$
  end for
  $\mu = \mu^*[1] + \mu^*[2], \ldots$
  $return\ \mu$
end procedure

Algorithm 3 GAIKube Scheduler

Input: Logs Path, SLA Threshold SLA_{th}, CPU Threshold θ_{th}, Previous Decision D
Output: New Scheduling Decision D'
procedure PLACEMENT
  for $i = 1$, $i{+}{+}$, while $i \leq I$ **do**
    μ = Timesfm(ρ)
    $\Theta = logs[i-1]$
    V = SLA-Violations(Θ)
    if O = $\exists\, \mu \geq \theta_{th}$ **then**
      $V', D' = Overload(O, V)$
    else
      $V', D' = Normal - Load(V)$
    end if
    $D' = Model - Switching(V')$
    Calculate $C(i)$, $Y(i)$
    return D', $C(i)$, $Y(i)$
  end for
end procedure

Algorithm 4 Overload Handler

1: **procedure** OVERLOAD(O, V)
2: **for** $o = 1$, $o{+}{+}$, while $o \leq |O|$ **do**
3: P[o] = $\sum_{p \in o} p$
4: **end for**
5: $\hat{P} = descend\_sort(P)$
6: $\hat{N}_{free} = descend\_sort(\hat{N})$
7: **for** $o = 1$, $o{+}{+}$, while $o \leq |O|$ **do**
8: **for** $p = 1$, $p{+}{+}$, while $p \leq |\hat{P}_o|$ **do**
9: $placed = False$
10: **while** $|\hat{N}_{free}| > 1$ **do**
11: $h = \hat{N}_{free}.pop()$
12: $\theta = p(\theta)$
13: **while** $\theta >= min(p(\theta^r))$ **do**
14: **if** $((h(\theta) + \theta)/h_{alloc}) * 100 \leq \theta_{th}$ **then**
15: $D'[p] = (h_{id}, \theta, mi)$
16: $V[p][pass] = True$
17: $placed = True$
18: *break*
19: **else**
20: $\theta = \theta/2$
21: **end if**
22: **end while**
23: **if** $placed = True$ **then**
24: *break*
25: **end if**
26: **end while**
27: **if** $o_{util} \leq \theta_{th}$ **then**
28: *break*
29: **end if**
30: **end for**
31: **end for**
32: **return** V, D'
33: **end procedure**

in Eq. (11) for underlined heterogeneous edge clusters. This scheduler is classified into sub-categories to respond to dynamic workloads leveraging Times FM predictions and observed latency metrics. Algorithm 3 presents a scheduling procedure responsible for managing overload, normal load, and SVR using Algorithms 4, 5 and 6, respectively. Moreover, containers hosted in forecasted overloading servers are proactively migrated to mitigate server faults. GAIKube scheduler extracts MTA TimesFM predictions μ at each interval. In addition, it reads public traffic YOLO statistics of the previous iteration for each container in the system to calculate SVR. Violations are calculated from execution time T_e of user requests as each model version exhibits different processing times for distinct container cores. Algorithm 3 step 8 leverages predictions to detect overloading nodes. In case of a non-empty response, Algorithm 4 is invoked with overloaded nodes indices *O*. There are possible migrations or container vertical downscaling from predicted overload servers *O*. Otherwise, the predicted load is in control where GAIKube strives for accuracy hosting improved accuracy YOLO model using Algorithm 5. Algorithm 6 entertains the non-altered containers and switches hosted YOLO versions to either maximize accuracy or reduce SLA violations. System cost in terms of containers CPU cores, migration count, model switches count, and the rest of the metrics are updated in step 13. New decision D' with possible migrations, vertical scalings, and model-switches is implemented in the GKE heterogeneous edge server as shown in steps 10-11 of Fig. 2.

1) Predicted Overload: Algorithm 4 presents GAIKube procedure responding to predicted overload situations. Fig. 3 presents the CPU usage of employed core VMs and it is visible that 2 core has regular CPU usage exceeding the threshold of 80%. The overload handler identifies and sorts containers $\hat{P}$ on each of the predicted overloaded nodes in descending order of their requested resources as shown in Algorithm 4 step 5. In search of a new destination for these containers, edge cluster nodes $\hat{N}$ are shortlisted offering at least 0.5 cores of CPU followed by descending order sorting. $\hat{P}$ and $\hat{N}$ given in scheduler section are subsets of original *P* and *N*, respectively.

Exploiting the sorted available edge nodes with predicted normal load $\hat{N}$, GAIKube attempts to migrate each container without CPU downscale in the first place if the new node stays under the 80% threshold as shown in steps 7-17 Algorithm 4. If the server usage condition fails in step 13, GAIKube cuts down the container CPU and repeats the search. Container CPU cutdown stops at 0.5 cores considering the base case for hosting the YOLO application. In a successful search, subjected container location, YOLO image *mi*, and CPU core are modified given in Algorithm 4 step 15. If the requested CPU resources on given overloaded nodes fall under the threshold confirmed in step 22 Algorithm 4, GAIKube leaves the rest of the containers on this node and moves to the next overloaded node in the list. Finally, updated decision D' and SLA violation list *V* are returned for further processing.

2) Predicted Normal Load: GAIKube Algorithm 3 invokes *Normal Load* Algorithm 5 when MTA predicted CPU usage of heterogeneous edge servers is under the threshold for all edge servers. In such times, GAIKube strives for higher accuracy with possible upscaling. Differing from overload, Algorithm 5 selects containers $\hat{P}$ hosting less accuracy YOLO versions and sorts them in ascending order of CPU cores. Moreover, edge servers are sorted in the descending order of remaining

Algorithm 5 Normal Load Handler

procedure NORMAL-LOAD(V)
  $\hat{P} = ascend\_sort(\hat{P})$
  $\hat{N}_{free} = descend\_sort(\hat{N})$
  for $p = 1$, $p{+}{+}$, while $p \leq |\hat{P}|$ **do**
    placed = False
    $pod = \hat{P}.pop()$
    while $|\hat{N}_{free}| > 1$ **do**
      $h = \hat{N}_{free}.pop()$
      $\theta = \max(p(\theta^r))$
      while $\theta > pod(\theta^r)$ **do**
        if $((h(\theta) + \theta)/h_{alloc}) * 100 \leq \theta_{th}$ **then**
          $D'[p] = (h_{id}, \theta, mi)$
          $V[p][pass] = True$
          placed = True
          break
        else
          $\theta = \theta/2$
        end if
      end while
      if *placed* $=$ *True* **then**
        break
      end if
    end while
  end for
  return V, D'
end procedure

Algorithm 6 Model-Switching

procedure MODEL-SWITCH(V')
  for $p = 1$, $p{+}{+}$, while $p \leq |P|$ **do**
    $row = V[p]$
    if $row[p][pass] = True$ **then**
      continue
    end if
    $\_curr\_vio = row[violation]$
    if $\_curr\_vio > 0.1$ **then**
      Model version down
    else
      Model version up
    end if
    $\hat{N}_{free} = sort(\hat{N})$
    placed = True
    while $|H_{free}| > 1$ *and placed* $=$ *False* **do**
      $h = \hat{N}_{free}.pop()$
      $\theta = p(\theta)$
      if $((h(\theta) + \theta)/h_{alloc}) * 100 \leq \theta_{th}$ **then**
        $D'[p] = (h_{id}, \theta, mi)$
        placed = True
        break
      end if
    end while
  end for
  return D'
end procedure

resources. For each container, GAIKube begins the search with the highest CPU core and model version based on the YOLO profiled dataset to select a new server capable of hosting this container without violating the CPU usage threshold checked in step 11 Algorithm 5. In success, container location, CPU cores, and YOLO image are modified along with locally managed cluster metrics. However, the failure leads to container CPU cutdown shown in step 17 followed by a repeating cycle on the same server node. This cycle breaks when CPU cutdown reaches the currently provisioned cores with the threat of possible scale-down instead of up. Algorithm 5 returns decision D' and modified SLA violation list V of containers to Algorithm 3.

3) Model Switching: In either of the load situations explained before, the model-switching Algorithm 6 is called at each iteration given in step 12 of Algorithm 3 with the modified violation list V'. This modified list has the SVR of unaffected containers in the system so far where Algorithm 6 modifies their YOLO models. GAIKube downgrades the YOLO model for the container offering a 10% violation rate for public traffic entertained in the previous iteration otherwise the model is upgraded in steps 7-10 of Algorithm 6. In either case, a new container has to be started with the updated YOLO image and edge server node. Algorithm 6 search edge server for this container in steps 13-19 with 80% threshold condition.

4) Complexity Analysis: Beginning with DGAN being responsible for data augmentation exploiting limited available historical records by creating synthetic samples. Among data preprocessing, DGAN training, new sample generation, and concatenation, training is the dominant factor with the complexity of $\mathcal{O}(M \cdot R \cdot T)$ where *M*, *R*, and *T* represent unique servers (each core has an individual model), number of employed records, and training steps, respectively. TimesFM predictor is the second significant component offering autoregressive inferences with the computational complexity $\mathcal{O}(I \cdot R)$ where *I* is the predictions iteration count shown in step 6 of Algorithm 2. Finally, the scheduler component poses more complexity for its sub-modules to handle dynamic workload and accuracy objectives producing container migration, dynamic scaling, and model-switching decisions. Scheduler Algorithm 3 calls either Algorithm 4 or 5 with a set of predicted overloaded nodes *O* where each of these sub-modules deals with modified pods list $\hat{P}$ and nodes list $\hat{N}$. The model-switching algorithm is mandatory with an additional complexity of pods *P* and modified nodes $\hat{N}$. Thus, the scheduler yields an overall complexity of $\mathcal{O}(O \cdot \hat{P} \cdot \hat{N} + P \cdot \hat{N})$.

Combining the complexities of DGAN, TimesFM predictor, and the GAIKube scheduler, the worst-case complexity of this framework can be calculated by $\mathcal{O}(M \cdot R \cdot T + I \cdot R + O \cdot \hat{P} \cdot \hat{N} + P \cdot \hat{N})$. It is important to note that the actual runtime may vary due to implementation details, parallel processing capabilities, network latency, and the specific algorithms employed for model training, prediction, and decision-making.

V. Performance Evaluation

This section details the GKE experimental testbed, evaluation metrics, and workload module. In addition, we analyzed the performance of compared predictors and schedulers separately to signify their contributions in terms of respective evaluation metrics.

A. Experimental Setup

GAIKube evaluates the performance of the proposed work and its counterparts in the GKE-based real testbed Europe-West2-b London regional cluster. It is a three heterogeneous nodes cluster of 2, 4, and 6 cores VMs with 50 GB of fixed storage for each VM. Due to budget limitations, one node from each core category is considered. Furthermore, we employed two 0.5 cores, one of 1 core and one of 2 cores containers in the experimentation. Table IV represents GKE cluster configurations considered in this work. After cluster creation, these machines are warmed up by pulling each Docker image in every node/ VM as image pulling poses significant delays that

TABLE IV
SERVER NODES AND CONTAINERS CONFIGURATIONS

Features / Machines	vCPU Cores	Memory (GB)	Disk Size (GB)	Quantity	IP Type	Model Version
Node-1	2	4	50	1	Private	
Node-2	4	8	50	1	Private	
Node-3	6	8	50	1	Private	
Container-1	0.5[1]	1	Dynamic	1	LoadBalancer	Nano[2]
Container-2	1[1]	1	Dynamic	2	LoadBalancer	Small[2]
Container-3	2[1]	1	Dynamic	1	LoadBalancer	Medium [2]

[1]Container-CPU*: 0.5, 1, 2. [2]Model-Version*: Nano, Small, Medium.

TABLE V
99TH PERCENTILE PROCESSING TIME (MS) COMPARISON

Core / Version	Nano	Small	Medium
Half	902.03	2015.69	6662.79
One	499.33	895.28	2349.16
Two	368.76	569.76	925.28

can substantially influence experimentation. Beginning with the deployment of all the containers in the *optimized* mode (explained in Section I-B), the scheduler is activated from iteration 1 as it requires metrics for decision-making.

Table V presents the 99th percentile (P99) of processing time for each container core and model version extracted from the YOLO profiled dataset. Considering 700ms SLA as shown in the Motivation section, nano can meet SLA for one and two-core containers, small is better suitable for two-core, and medium is highly likely to violate SLA for all the cores given these results. Considering these insights 0.5, 1, and 2 core containers begin with nano, small, and medium versions, respectively while the container CPU cores and YOLO model versions are dynamically modified.

Each of the experiments is set to run for 43 iterations of 5-minute intervals from the testing set of Bitbrains, making it 3 hours and 40-minute trace. Due to time and resource limitations, we rescaled Bitbrains at 5-minute to 1-minute intervals. After each scheduling decision, there is a 1-minute of public traffic to each container generated by the workload module excluding container recreation overhead. At the end of an interval, the scheduler extracts various metrics, TimesFM predictions, and SVR to produce a new decision for the next iteration.

B. Evaluation Metrics

This section describes the evaluation metrics of both the prediction models and the edge cluster, used in this work to evaluate the performance of compared techniques.

1) Perdition Models Metrics: We have evaluated the performance of GAIKube using time series metrics such as Root Mean Square Error (RMSE) and Mean Absolute Error (MAE), and their details are given in previous works [36].

2) Edge Cluster Metrics: The following metrics are used to evaluate the performance of the GAIKube framework:

- SLA Violation Percentage (VP) [33]: SLA violation is defined as the workload request exceeding the 700ms threshold given by Eq. (9). For a given container over the total time intervals of I, VP comes from:

$$VP = \frac{\sum_{i=1}^{I} T_e > SLA_{th}}{|I|} \times 100\%. \tag{12}$$

- Average Migrations [7]: Migration is the case when a container location changes in comparison to the previous interval location. Average migrations over the total time interval I comes from:

$$AM = \frac{\sum_{i=1}^{I} Y(i)}{|I|}, \tag{13}$$

where $Y(i)$ shows the migration count in interval i.
- Mean Accuracy (MA [%]): Mean accuracy of Yolo models can be calculated by Eq. (7).
- Average Cost [7]: Cost comes from the CPU cores requested by containers given in Eq. (8) and the mean cost is the sum of cost over the total time intervals.

$$AC = \frac{\sum_{i=i}^{I} C(i)}{|I|}. \tag{14}$$

- CPU Utilization Percentage (CUP): CPU utilization [37] of a server node can be calculated using Eq. (1):

$$CUP = \frac{\sum n_k(\theta^r)}{n_k(\theta^{\max})} \times 100\%. \tag{15}$$

C. Baselines

This section describes all the baselines considered in the GAIKube framework. We evaluated time series prediction models for CPU usage, followed by schedulers including default Kubernetes configured in *optimized* mode and the combination of Pod level VPA along with Model-Switching.

1) Prediction Baselines: This section presents the baselines compared to the TimesFM to highlight the justification of its employment. We aimed to leverage long-term predictions for proactive decision-making to improve cluster nodes' health and avoid over-usage. For LSTM and Bi-LSTM, we added time series features of minute, hour, day, month, year, etc. into the Bitbrains dataset followed by normalization to [0, 1]. For the fair comparison with TimesFM 6 timesteps ahead predictions, we created data sequences for LSTM and Bi-LSTM, comprising the last two hours of data as input and target CPU Usage [%] of the next 18 values making 6 timesteps for each core. As stated in Section IV-C2 that TimesFM entertains each core as an individual series, we followed the same approach for both LSTM and Bi-LSTM by creating a separate model for each core. Table VI shows the hyperparameters of TimesFM and compared models.

- LSTM [30], [38]: LSTM is an RNN model commonly used for time series predictions based on its feature of holding memory for past intervals. It uses input, forget and output gates for traversing input data and weights to minimize the error by learning the data patterns.
- Bi-LSTM [31], [39]: It is an extended version of LSTM and it creates two LSTM models i.e., forward and backward to update model weights using backpropagation aiming to minimize error.

TABLE VI
MODEL HYPERPARAMETERS

Model	Hyperparameters
TimesFM	input=(timestamp, vm_core, cpu_usage), horizon=6, context_len=288, num_layers=20, input_patch_len=32, output_patch_len=128, frequency='5min', model_dims=128
LSTM	hidden_layers=2, layers=lstm, neurons=100, optimizer=Adam, lr=0.0001, loss='mse', epochs=100, batch=128, input=(72, 7), hidden_activate='relu', output=18, output_activate='linear'
Bi-LSTM	hidden_layers=2, layers=bi_lstm, neurons=100, optimizer=Adam, lr=0.0001, loss='mse', epochs=100, batch=128, input=(72, 7), hidden_activate='relu', output=18, output_activate='linear'

2) Scheduler Baselines: As GAIKube offers model-switching, dynamic scaling, and container scheduling simultaneously leveraging TimesFM pipelined predictions. Thus, the counterparts are required to be equipped with similar features for fair comparison. We selected two counterparts comprised of industry-standard composite features.

- The first scheme called Single model VPA GKE (SVG) offers no model-switching employing industry-standard GKE Vertical Pod Autoscaling (VPA) [27] and the GKE *optimized* mode scheduler [26].
- Model-switching [6], GKE VPA [27] and GKE *optimized* scheduler [26] combines together to create MVG scheme. Unlike SVG, MVG dynamically updates the YOLO model.

The scheduling and vertical container scaling decisions are the responsibility of GKE VPA and GKE scheduler for both SVG and MVG. GKE produces these decisions by exploiting its internal decision engine, monitoring, and historical records.

D. Workloads

Workload is the real-world traffic directed to Yolo applications serving public users. In each experiment interval *i*, public traffic exploiting FLASK API is directed for one minute to all the active pods/ containers in a sequential manner followed by storing the logs. These insights are employed for container resource management in the next iterations subjected to the SLA violation rate.

E. Experimental Results

This section compares the performance of prediction models and the edge cluster schedulers.

1) Prediction Models Comparison: Fig. 4 shows the prediction results of baselines LSTM and Bi-LSTM against TimesFM. This figure presents a comparison of compared models on concatenated Bitbrains (left) and DGAN (right) data split by a vertical line. Each of these models is employed to predict CPU Usage [%] for the next half an hour as stated earlier. It can be seen that all three compared models captured the trend with LSTM a bit shy to detect peaks for 2 core machines. However, TimesFM and Bi-LSTM have better performance in detecting trends and peaks. Moreover, the auto-regressive mode enables TimesFM to handle the Bitbrains to DGAN data transition very well where LSTM and Bi-LSTM can be seen of the beat for 4 cores at a few times.

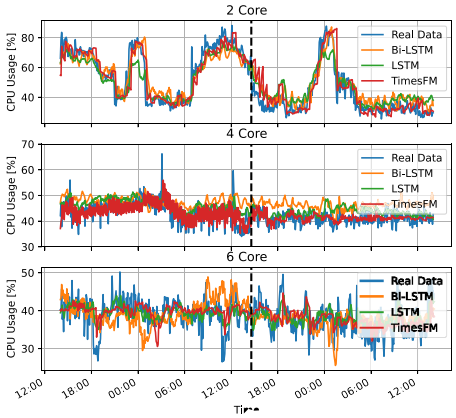


Fig. 4. TimesFM, Bi-LSTM, LSTM CPU predictions on testing & DGAN data of 48 hours for 2, 4 and 6 Core VMs. Bitbrains Testing data is at the left while DGAN is at the right of the vertical line.

TABLE VII
PROPOSED AND BASELINES MODELS COMPARISON

Model	Test Data RMSE	Test Data MAE	Test + DGAN RMSE	Test + DGAN MAE	Avg. Inf. Time (ms)
TimesFM	4.301	2.824	4.847	3.102	644.808
Bi-LSTM	5.617	4.526	7.108	5.535	47.483
LSTM	4.448	3.528	5.366	4.289	49.311

Two cores server has CPU usage between 20-85% with regular spikes exceeding the 80% threshold. 4 and 6 cores server show stable usage. Prior ranges between 30-70% while later has CPU usage between 30-50%. TimesFM outperforms both LSTM and Bi-LSTM for 4 core VM where it can be seen that both counterparts get off the pattern but later cover the loss. The auto-regressive mode data feeding enables TimesFM to predict more accurate results due to fresh data. Finally, 6 core VM has the most stable usage among the other 2 VMs. Similar results are shown by each model where TimesFM is staying in the middle, missing a few short peeks but these are not critical in comparison to 2 core where CPU usage hits 80% critical point. LSTM stays closer to TimesFM and Bi-LSTM attempts to detect peaks at few points. Table VII presents RMSE, MAE error scores, and the average prediction or inference time for a single prediction. We are presenting RMSE and MAE on the Bitbrains testing data and concatenated Bitbrains and DGAN data. TimesFM has the lowest RMSE and MAE, closely followed by LSTM while Bi-LSTM has a poor score for both metrics. We have similar results for Bitbrains and DGAN data RMSE and MAE where TimesFM has the lead followed by LSTM and Bi-LSTM validating Fig. 4 results. Furthermore, TimesFM has the worst inference time of 645ms which is more than 10 times the baselines for reasons. Firstly, both the LSTM and Bi-LSTM are provisioned well-processed data while TimesFM requires non-normalized data and a few hyperparameters performing the processing itself. Secondly, baselines are once trained while TimesFM requires auto-regressive data. These reasons are responsible for highly accurate predictions of TimesFM at the cost of inference time.

Analyzing the RMSE and MAE errors of TimesFM on the concatenated dataset, two conclusions can be drawn.

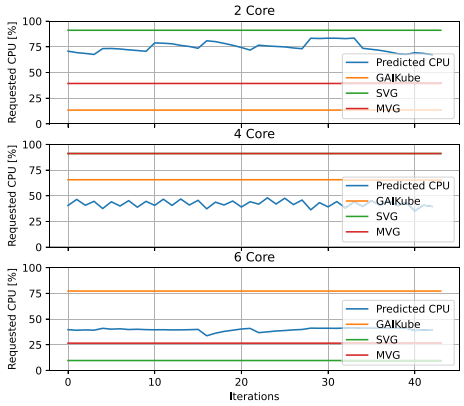


Fig. 5. Requested CPU Percentage of 2, 4 and 6 core VMs.

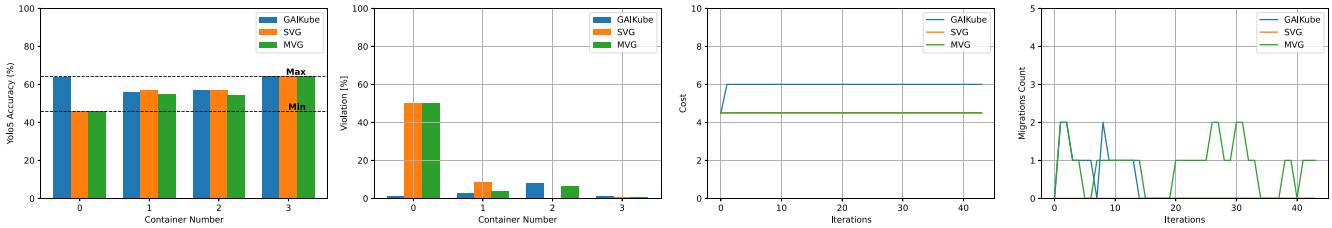
Firstly, DGAN can produce quality time series synthetic data to address the data limitations with significantly low errors. Secondly, auto-regressive logs appending enables the Transformer family TimesFM to produce significant accurate predictions. Thus, we employed TimesFM in the GAIKube framework.

2) Cluster Metrics Comparison: This section presents the comparison of GAIKube and the baselines mentioned in Section V-C2 in terms of metrics defined in Section V-B.

Fig. 5 presents the CPU utilization of each of the 2, 4, and 6 cores GKE edge servers. These are the results of 43 timesteps approximately 3 hours 40 minutes from the Bitbrains dataset. Exploiting the TimesFM predictions, GAIKube improves utilization and successfully avoids faults by not letting any server violate the CPU 80% threshold. It emptied 2 cores server and maximally utilized the 4 and 6 cores servers. Dynamic upscaling of containers in response to higher accuracy exploiting P99 latency given in Table V with the CPU threshold condition eliminates 2-core VM from the race. On the contrary, both SVG and MVG utilize the same GKE-optimized scheduler and it can be seen that 2 and 4-core VMs are over-utilized by each of them, while the 6-core VMs are under-loaded. GKE optimized mode strives to maximize utilization without any limitation on resource usage raising faults and damage probability by a higher magnitude. Results analysis shows that GAIKube efficiently utilizes available resources while avoiding over-utilization at the heterogeneous cluster.

Fig. 6(a) shows the comparison of average accuracy given in Eq. (6) achieved by each container. Max shows the maximum achievable accuracy of Yolo5 offered by the medium version while Min is the lowest bound by the nano version. Containers 0, 1, 2, and 3 are initialized with nano, small, small, and medium versions, respectively based on P99 latency comparison in Table V. Model-switching capability subjected to SVR and higher accuracy hunger, results in the best performance for GAIKube. Despite starting with 0.5 core, container 0 upscales in the initial phase and stays at this CPU core to maximize accuracy for GAIKube while SVG and MVG fail to upscale. Both of these employ the same GKE scaling mechanism. Despite enabled vertical scaling, GKE VPA fails to upscale for two reasons. Firstly, it can scale utilizing monitored CPU and memory resources only. Secondly, GKE VPA is offered for the *Deployment* level, not the individual *Pod* level. These limitations left container 0 with a 0.5 core hosting nano model for each SVG and MVG. Yolo5 small version offers 56.8% of accuracy and SVG achieves this at containers 1 and 2 as it never changes the model. While GAIKube and MVG model switching resulted in slightly lesser accuracy to SVG. Finally, all of GAIKube, SVG, and MVG achieve the highest accuracy at container 3. SVG never changes the model, while MVG and GAIKube have the same model-switching logic. With this logic, the model is only switched to low accuracy if there is at least a 10% violation rate in the last interval, which never happened for GAIKube and MVG. Given the highest achievable accuracy of 64.1% for Yolo5 medium, GAIKube acquires 60.21%, SVG 55.85%, and MVG 54.7% mean accuracy at cluster level as shown in Table VIII.

SLA violation percentage for each container is shown in Fig. 6(b) given by Eq. (12). GAIKube offers the lowest VP for containers 0 and 1 while, there are 8.23% and 1.14% violations for containers 2 and 3, respectively. In conjunction with container 0 accuracy in Fig. 6(a), GAIKube has the lowest VP while the failed scaling of SVG and MVG results in almost 50% VP. GAIKube has the lowest violations for container 1 because of model switching followed by MVG for the same reason. SVG hosts the same small version for both containers 1 and 2 and it has 8.74% violations for container 1 and 0% for container 2. As the SLA violations are subjected to execution time T_e of user requests, some unforeseen background CPU usage can lead to such cases. As expected container 3 has the lowest violations for all the compared techniques. Further, Table VIII presents cluster-wide VP where GAIKube has 3.43%, SVG has 14.77% and MVG has 15.30% VP. GAIKube and MVG are offering higher SLA violations for container 2 subjected to switching logic which shows the requirement for a fine-tuned logic. Cost is presented in Fig. 6(c) for each time interval. As stated earlier container 0 is upscaled in the initial phase and it stays at this scale throughout the lifetime which resulted in higher cost for GAIKube in comparison to SVG and MVG. The experiment is initialized with 0.5, 1, 1, and 2 cores for containers 0, 1, 2, and 3, respectively. It sums up the cost to 4.5 cores. This cost is constant for SVG and MVG for its vertical scaling limitation. However, GAIKube scales containers 0, 1, 2, and 3 to 2, 1, 1, and 2 cores respectively offering the highest cost. Table VIII presents the AC of each technique. Finally, Fig. 6(d) shows the container migrations in each iteration. SVG has zero migrations as it never scales and does not change the model version. There are frequent migrations for MVG for its model switching and the GKE-optimized mode scheduling. On the contrary, there are fewer migrations for GAIKube but these are significant ones. As GAIKube moves from nano to small and medium versions in all containers. Thus, there is a possibility of container migration subjected to SLA VP of 10% in the last iteration. However, the available resources in the 4 and 6 core machines are not enough to host more containers while 2 core is predicted to be overloaded, thus there is no migration after the 13th interval for GAIKube. AM presented in Table VIII shows that SVG conducted no migrations and scaling while



(a) YOLO mean accuracy of containers (b) SLA VP of containers (c) CPU Cores Cost (d) Migration Count

Fig. 6. YOLO mean accuracy, SLA VP, CPU cores cost, and migration count comparison of GAIKube, SVG, and MVG.

TABLE VIII
VIOLATION PERCENTAGE, MEAN ACCURACY, AVERAGE MIGRATIONS AND AVERAGE COST COMPARISON

Method \ Metric	VP [%]	MA [%]	AM	AC
GAIKube	3.43	60.21	0.34	5.96
SVG	14.77	55.85	0.0	4.5
MVG	15.30	54.70	0.84	4.5

there is an average of 0.34 migrations for GAIKube and 0.84 migrations for MVG.

VI. CONCLUSION AND FUTURE WORK

We presented GAIKube to address the challenges of data limitation, poor time series predictions, and beyond the safe threshold CPU usage by industry-standard GKE scheduler. GAIKube employs DGAN to generate new CPU usage data for heterogeneous edge servers. Google TimesFM exploits this data to produce MTA predictions for informed decision-making. RMSE and MAE errors for concatenated Bitbrians and DGAN data demonstrate the quality of DGAN synthetic data and the significance of TimesFM predictions. Finally, GAIKube proposed a proactive and efficient Kubernetes container orchestrator to maximize resource utilization, reduce SLA VP, improve user-experienced accuracy, and avoid server faults for the hosted Yolo5 DL application. CPU usage-oriented proactive container management and SLA violation-oriented reactive model switching enable GAIKube to achieve contrasting accuracy, SLA, and cost objectives. GAIKube offers a reduced 3.43% SLA violations and 3.89% accuracy drop at 1.46 CPU core expense. The industry-standard GKE SVG scheduler offers 14.77% SLA violations and 8.25% MA loss while MVG has 15.30% and 9.4% SLA violations and MA loss, respectively, where both schemes failed to avoid server faults. There are a few possible directions to explore in the future. Firstly, TimesFM performed the best however, the higher inference time can be a bottleneck that should be reduced. Secondly, GAIKube addressed computationally heterogeneity for CPU servers leaving room to explore a mix of CPU and GPU heterogeneous edge clusters. Finally, we employed the YOLO application and these IoT applications can be increased to produce an extensive framework.

SOFTWARE AVAILABILITY

GAIKube framework code is publicly available for the researchers at https://github.com/BabarAli93/GAIKube.

REFERENCES

[1] X. Shao, G. Hasegawa, M. Dong, Z. Liu, H. Masui, and Y. Ji, "An online orchestration mechanism for general-purpose edge computing," *IEEE Trans. Services Comput.*, vol. 16, no. 2, pp. 927–940, Mar./Apr. 2023.

[2] F. Zantalis, G. Koulouras, S. Karabetsos, and D. Kandris, "A review of machine learning and IoT in smart transportation," *Future Internet*, vol. 11, no. 4, p. 94, 2019.

[3] L. M. Al Qassem, T. Stouraitis, E. Damiani, and I. M. Elfadel, "Containerized microservices: A survey of resource management frameworks," *IEEE Trans. Netw. Service Manag.*, vol. 21, no. 4, pp. 3775–3796, Aug. 2024.

[4] B. Ali, M. Golec, S. S. Gill, H. Wu, F. Cuadrado, and S. Uhlig, "EdgeBus: Co-simulation based resource management for heterogeneous mobile edge computing environments," *Internet Things*, vol. 28, Dec. 2024, Art. no. 101368.

[5] S. S. Gill et al., "Edge AI: A taxonomy, systematic review and future directions," *Clust. Comput.*, vol. 28, no. 1, pp. 1–53, 2025.

[6] J. Zhang, S. Elnikety, S. Zarar, A. Gupta, and S. Garg, "Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems," in *Proc. 12th USENIX Workshop Hot Topics Cloud Comput. (HotCloud)*, 2020, pp. 1–8.

[7] M. Salmani et al., "Reconciling high accuracy, cost-efficiency, and low latency of inference serving systems," in *Proc. 3rd Workshop Mach. Learn. Syst.*, 2023, pp. 78–86.

[8] G. T. Francis, A. Souri, and N. Inanç, "A hybrid intrusion detection approach based on message queuing telemetry transport (MQTT) protocol in Industrial Internet of Things," *Trans. Emerg. Telecommun. Technol.*, vol. 35, no. 9, 2024, Art. no. e5030.

[9] S. Tuli, G. Casale, and N. R. Jennings, "PreGAN+: Semi-supervised fault prediction and preemptive migration in dynamic mobile edge environments," *IEEE Trans. Mobile Comput.*, vol. 23, no. 6, pp. 6881–6895, Jun. 2024.

[10] T. Zonta, C. A. Da Costa, R. da Rosa Righi, M. J. de Lima, E. S. da Trindade, and G. P. Li, "Predictive maintenance in the industry 4.0: A systematic literature review," *Comput. Ind. Eng.*, vol. 150, Dec. 2020, Art. no. 106889.

[11] A. Halevy, P. Norvig, and F. Pereira, "The unreasonable effectiveness of data," *IEEE Intell. Syst.*, vol. 24, no. 2, pp. 8–12, Mar./Apr. 2009.

[12] I. Goodfellow et al., "Generative adversarial nets," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 27, 2014, pp. 1–9.

[13] J. Gui, Z. Sun, Y. Wen, D. Tao, and J. Ye, "A review on generative adversarial networks: Algorithms, theory, and applications," *IEEE Trans. Knowl. Data Eng.*, vol. 35, no. 4, pp. 3313–3332, Apr. 2023.

[14] E. Brophy, Z. Wang, Q. She, and T. Ward, "Generative adversarial networks in time series: A systematic literature review," *ACM Comput. Surveys*, vol. 55, no. 10, pp. 1–31, 2023.

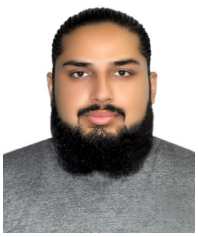
[15] Z. Lin, A. Jain, C. Wang, G. Fanti, and V. Sekar, "Using GANs for sharing networked time series data: Challenges, initial promise, and open questions," in *Proc. ACM Internet Meas. Conf.*, 2020, pp. 464–483.

[16] J. Yoon, D. Jarrett, and M. Van der Schaar, "Time-series generative adversarial networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 1–11.

[17] E. Ayanoglu, K. Davaslioglu, and Y. E. Sagduyu, "Machine learning in NextG networks via generative adversarial networks," *IEEE Trans. Cogn. Commun. Netw.*, vol. 8, no. 2, pp. 480–501, Jun. 2022.

[18] M. Allen, U. Naeem, and S. S. Gill, "Q-Module-Bot: A generative AI-based question and answer bot for module teaching support," *IEEE Trans. Educ.*, vol. 67, no. 5, pp. 793–802, Oct. 2024.

[19] C. Liang et al., "Generative AI-driven semantic communication networks: Architecture, technologies and applications," *IEEE Trans. Cogn. Commun. Netw.*, early access, Jul. 29, 2024, doi: 10.1109/TCCN.2024.3435524.
[20] M. Xu et al., "Unleashing the power of edge-cloud generative AI in mobile networks: A survey of AIGC services," *IEEE Commun. Surveys Tuts.*, vol. 26, no. 2, pp. 1127–1170, 2nd Quart., 2024.
[21] M. Golec et al., "CAPTAIN: A testbed for co-simulation of scalable serverless computing environments for AIoT enabled predictive maintenance in industry 4.0," *IEEE Internet Things J.*, early access, Oct. 30, 2024, doi: 10.1109/JIOT.2024.3488283.
[22] S. Ghafouri, S. Abdipoor, and J. Doyle, "Smart-Kube: Energy-aware and fair kubernetes job scheduler using deep reinforcement learning," in *Proc. IEEE 8th Int. Conf. Smart Cloud (SmartCloud)*, 2023, pp. 154–163.
[23] S. Tuli et al., "HUNTER: AI based holistic resource management for sustainable cloud computing," *J. Syst. Softw.*, vol. 184, Feb. 2022, Art. no. 111124.
[24] S. Shen, V. Van Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *Proc. 15th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2015, pp. 465–474.
[25] A. Das, W. Kong, R. Sen, and Y. Zhou, "A decoder-only foundation model for time-series forecasting," in *Proc. 41st Int. Conf. Mach. Learn.*, 2024, pp. 1–21.
[26] "Optimized GKE scheduling." 2024. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler
[27] "Vertical pod autoscaling." 2024. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler
[28] S. Tuli, G. Casale, and N. R. Jennings, "PreGAN: Preemptive migration prediction network for proactive fault-tolerant edge computing," in *Proc. IEEE Conf. Comput. Commun.*, 2022, pp. 670–679.
[29] B. K. Ray, A. Saha, S. Khatua, and S. Roy, "Proactive fault-tolerance technique to enhance reliability of cloud service in cloud federation environment," *IEEE Trans. Cloud Comput.*, vol. 10, no. 2, pp. 957–971, Apr.–Jun. 2022.
[30] T. Wang, S. Ferlin, and M. Chiesa, "Predicting CPU usage for proactive autoscaling," in *Proc. 1st Workshop Mach. Learn. Syst.*, 2021, pp. 31–38.
[31] M.-N. Tran, X. T. Vu, and Y. Kim, "Proactive Stateful fault-tolerant system for kubernetes containerized services," *IEEE Access*, vol. 10, pp. 102181–102194, 2022.
[32] J. Li, X. Zhang, J. Wei, Z. Ji, and Z. Wei, "GARLSched: Generative adversarial deep reinforcement learning task scheduling optimization for large-scale high performance computing systems," *Future Gener. Comput. Syst.*, vol. 135, pp. 259–269, Oct. 2022.
[33] S. Tuli, S. R. Poojara, S. N. Srirama, G. Casale, and N. R. Jennings, "COSCO: Container orchestration using co-simulation and gradient based optimization for fog computing environments," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 1, pp. 101–116, Jan. 2022.
[34] "Pricing compute engine: Virtual machines (VMS) | Google cloud," 2024. [Online]. Available: https://cloud.google.com/compute/all-pricing
[35] G. Jocher, "YOLOv5 by Ultralytics." 2020. [Online]. Available: https://github.com/ultralytics/yolov5
[36] S. Velu, S. S. Gill, S. S. Murugesan, H. Wu, and X. Li, "CloudAIBus: A testbed for AI based cloud computing environments," *Clust. Comput.*, vol. 27, pp. 11953–11981, Jun. 2024.
[37] K. Mason, M. Duggan, E. Barrett, J. Duggan, and E. Howley, "Predicting host CPU utilization in the cloud using evolutionary neural networks," *Future Gener. Comput. Syst.*, vol. 86, pp. 162–173, Sep. 2018.
[38] L. Nashold and R. Krishnan, "Using LSTM and SARIMA models to forecast cluster CPU usage," 2020, *arXiv:2007.08092*.
[39] F. Ullah, M. Bilal, and S.-K. Yoon, "Intelligent time-series forecasting framework for non-linear dynamic workload and resource prediction in cloud," *Comput. Netw.*, vol. 225, Apr. 2023, Art. no. 109653.



Babar Ali is currently pursuing the Ph.D. degree with the School of Electronic Engineering and Computer Science, Queen Mary University of London. He has published his research findings in journals such as *Internet of Things* (Elsevier) and *International Journal of Network Management* (Wiley). His research interests include cloud computing, IoT, edge computing, and wireless sensor networks.

Muhammed Golec received the M.Sc. (Distinction) degree in computer science through the Ministry of Education Scholarship from the Queen Mary University of London, where he is currently pursuing the Ph.D. degree. His research interests include cloud computing, serverless computing, AI, and security and privacy.

Subramaniam Subramanian Murugesan received the master's degree in big data science from the Queen Mary University of London, where he is currently pursuing the Ph.D. degree in electronic engineering. He has published his research findings in journals such as IEEE JOURNAL OF BIOMEDICAL AND HEALTH INFORMATICS and *Cluster Computing* (Springer). His research focuses on AI/ML/DL applications, cloud & IoT, software engineering, and edge AI technologies.

Huaming Wu (Senior Member, IEEE) received the B.E. and M.S. degrees in electrical engineering from the Harbin Institute of Technology, China, in 2009 and 2011, respectively, and the Ph.D. (Highest Honor) degree in computer science from Freie Universität Berlin, Germany, in 2015. He is currently a Professor with the Center for Applied Mathematics, Tianjin University, China. His research interests include mobile cloud computing, edge computing, Internet of Things, deep learning, complex networks, and DNA storage.

Sukhpal Singh Gill is an Assistant Professor of cloud computing with the School of Electronic Engineering and Computer Science, Queen Mary University of London, U.K. His research interests include cloud computing, edge computing, IoT, and energy efficiency. He is serving as an Editor-in-Chief for *International Journal of Applied Evolutionary Computation* (IGI Global) and an Area Editor for *Cluster Computing Journal* (Springer), also serving as an Associate Editor for IEEE INTERNET OF THINGS JOURNAL, *Internet of Things* (Elsevier), Wiley SPE, *Transactions on Emerging Telecommunications Technologies* (Wiley), and *IET Networks Journals*.

Felix Cuadrado received the Ph.D. degree in telecommunications engineering from the Universidad Politécnica de Madrid, Spain, in 2009, where he is an Associate Professor with the School of Telecommunications Engineering. He is also the Visiting Reader with the Queen Mary University of London. He has numerous publications in top-tier journals and conferences, including the IEEE TRANSACTIONS ON SERVICES COMPUTING, the IEEE TRANSACTIONS ON CLOUD COMPUTING, *Journal of Systems and Software*, *Future Generation Computer Systems* (Elsevier), IEEE ICDCS, and *Scientific Reports* (Nature). He is a Fellow of the Alan Turing Institute.

Steve Uhlig received the Ph.D. degree in applied sciences from the University of Louvain, Belgium, in 2004. Prior to joining Queen Mary, he was a Senior Research Scientist with Technische Universität Berlin/Deutsche Telekom Laboratories, Berlin, Germany. He has been the Professor of Networks and the Head with the Networks Research Group, Queen Mary University of London, since January 2012. From 2012 to 2016, he was a Guest Professor with the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. His current research interests include Internet measurements, software-defined networking, and content delivery.