Learning When to Wait: A Lightweight Binary Decision Framework for Container Migration in Edge Computing

Yasmeen Abu-Haltem[1], Sawsan AlShattnawi[2]
Computer Science Department, Princess Sumaya University for Technology, Amman, Jordan[1]
Computer Science Department, Yarmouk University, Irbid, Jordan[2]

***Abstract*—Container migration is one of the important techniques used to provide the continuity of services in Multi-access Edge Computing environments. Services are migrated from edge servers to follow the mobility of users among the available edge servers. Nevertheless, current strategies do not take into account the cost that a container migration causes in terms of downtime, bandwidth consumption, energy cost, and the possible occurrence of failures during the process. In this work, a lightweight pre-migration binary decision framework is introduced that works as a gatekeeper before any migration attempt. According to the actual state of the system, the framework decides at each epoch whether to migrate or not, classifying the epochs into Migrate and Wait categories. The training labels are obtained through a counterfactual approach using simulated data generated using the EdgeSimPy simulation tool. The Decision Tree classifier has been chosen as the main classifier due to its minimal computing complexity and interpretability. The effectiveness of the suggested framework is measured on the basis of comparison with three baseline policies, including always migrate, never migrate, and a latency-threshold rule, along with multiple machine learning models and deep learning models, and such metrics as accuracy, precision, recall, F1-score, AUC, and runtime performance metrics. It is shown that the suggested solution outperformed all three baseline policies along with the machine learning and deep learning models, proving that an intelligent decision-making stage is essential in Multi-access Edge Computing.**

Keywords*—Multi-access Edge Computing; container migration; machine learning; Decision Tree; mobility management; service continuity

I. Introduction

Multi-access Edge Computing (MEC), formerly known as Mobile Edge Computing, is becoming increasingly important as use cases such as augmented reality, self-driving cars, healthcare uses, and real-time video analytics require low latency. Computing services deployed at the edge shorten response time. Mobility, on the other hand, creates an important problem for edge computing. As the user moves around the city or country, the distance between them and the edge server providing the service will increase, resulting in higher latency. If the latency becomes too high, the service might become interrupted, which is unacceptable in time-critical scenarios.

The most well-known and used solution to this issue is what is referred to as container migration. In order to maintain low latency and service continuity, the service can follow the user by migrating a running container from one edge server to another. Optimizing container migration has been a great topic for current studies; researchers use advanced methods to find acceptable solutions for the issue. This is done by minimizing downtime, reducing checkpoint size, and choosing better destinations for migrating. Despite many studies on migration triggering methods, migration optimization, and destination determination techniques, most of them take for granted that once a trigger condition is satisfied, migration is always the most effective plan of action and needs to be performed. Not much effort has gone into finding out if migration is even necessary at all. This could potentially result in unnecessary migrations, wasting resources and degrading service performance.

This presumption ignores a crucial fact. Migration is not free, and it is not always the perfect solution. It adds service interruptions during checkpoint transfer, uses network bandwidth, costs energy to limited edge nodes, and carries the risk of failure if network conditions suffer in the middle of migration. Furthermore, a number of factors that cause migration, like slight CPU spikes or short-lasting network instability, may fade before the migration is even completed. Thereby, executing the entire process becomes ineffective and might negatively affect the quality of service (QoS).

In this study, it is argued that sometimes the best decision is not to migrate. A simple binary decision framework is proposed that serves as a gatekeeper before any attempt at migration. The framework provides an output to either proceed with migration, in other words, migrate, or not migrate and re-evaluate later, wait, based on the current system state. Factors affecting the system state include node resource pressure, container characteristics, and network quality. The proposed method is expected to minimize system overhead, prevent unsuccessful migrations, and maintain QoS when migration would be destructive by filtering unneeded and harmful migrations.

The main contributions of the study can be described as follows:

- A lightweight binary pre-migration decision framework is proposed as a modular gatekeeper layer, complementary to existing migration algorithms, that determines whether migration should be executed at all before any migration attempt.
- Creation of a balanced dataset containing 5,000 samples under various mobility settings and container configurations through EdgeSimPy.
- Application of a counterfactual labeling technique to label each decision point with Migrate or Wait labels.

- Evaluation and comparison of various machine learning and deep learning methods, alongside rule-based and unconditional migration baselines.
- Experimental findings show that lightweight decision tree models achieve an effective balance between prediction performance, interpretability, and efficiency, which makes them very well-suited for the edge computing environment.

This work is structured as follows: Section II presents a brief background of the topics mentioned, and relevant studies are discussed. The problem is formulated, and the research gap is discussed in Section III. The proposed decision framework is presented in Section IV. Section V presents the results. Section VI discusses the results. The study concludes in Section VII with a concise summary of the work and future work suggestions.

II. Background and Literature Review

MEC has become a crucial concept for helping and supporting latency-sensitive and data-intensive applications. This is achieved through moving storage and processing resources closer to end users. MEC allows processing at the edge, in contrast to traditional cloud computing, which processes data in centralized data centers. MEC is well-known and suitable for applications like augmented reality, autonomous cars, healthcare fields, and real-time video analytics because it works on greatly lowering communication latencies and enhances QoS [1] [2] [3].

However, because of user mobility and changing network conditions, MEC environments are considered to be very dynamic. The distance between a user and the serving edge node increases as the user travels across different geographic locations, which may result in higher latency and worse service performance [4]. Fig. 1 illustrates the user mobility and the impact of latency.

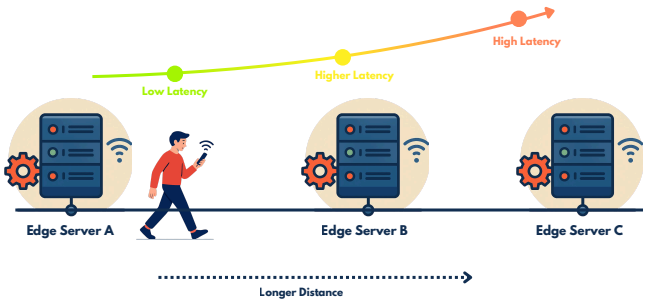

Fig. 1. Impact of user mobility on service latency in Multi-access Edge Computing environments.

Containerization has gained popularity in the world of effectively deploying applications in MEC environments. Applications can execute consistently across platforms due to containerization, which offers lightweight virtualization by packaging them with their needed dependencies. Containers are considered to be a good fit for edge nodes with limited resources because they require fewer resources, have quicker startup times, and are more scalable than traditional virtual machines [5]. Applications are often decomposed into microservices and deployed as containers in MEC systems, allowing for flexible and dynamic service management across different edge infrastructures.

One essential method for preserving service performance in MEC settings is container migration. To lower latency and preserve service reliability, the service container can be moved or migrated to a nearby node when a user moves away from the current edge server. Applications can dynamically adjust to user mobility as a result of this idea, which is frequently referred to as “follow-me" services, as defined in [6] and [7].

Despite all the benefits the container migration adds, it introduces some critical drawbacks. The container state must be transferred over the network as part of the migration process, which may add to the bandwidth usage, introduce additional computational overhead, and cause service outages. Additionally, unstable network conditions may cause migration to fail, which would further affect the system’s reliability. Container migration in edge computing has been widely studied, with the literature focusing on the most efficient way to perform the migration and the conditions that warrant the migration. This section reviews the literature to lay the foundation for the contribution of this study, highlighting the gap that this work addresses.

A. Container Migration Mechanisms

The migration of containers involves the migration of running applications across different physical hosts without stopping the processes. This migration process is mostly adopted in edge computing platforms to ensure continuity of service delivery in cases where the user leaves the current host of their application. Unlike restarting a container from the beginning, migration attempts to preserve the running state of the service, which is important for latency-sensitive and stateful applications.

The fundamental technology that facilitates this functionality on Linux-based operating systems includes CRIU (Checkpoint/Restore In User space). It involves the freezing of the running process, saving its complete status to the hard disk in the form of image files, moving the images to another machine, and restoring the process from where it stopped [8] [9]. CRIU can be used with container runtimes to support checkpoint/restore-based migration. One of the major discoveries in the literature is that the checkpoint phase is directly proportional to the size of memory allocated to the container; that is, a bigger container needs more time for the freezing and transferring phases and consequently causes longer downtime to the service [9].

Based on how the container state is transferred between the source and destination hosts, migration techniques are commonly classified into pre-copy [10], post-copy [11], and hybrid migration [12] approaches. During pre-copy container migration, the process of transferring pages from the origin server to the destination server takes place while the program is active on the origin server. Pages that are modified during the transfer are resent in later rounds. Once copying of most of the state has occurred, the container is momentarily paused, the dirty pages are copied, and then the container continues

its execution on the destination server. While such a method can minimize downtime, it may increase total transferred data when the application frequently modifies memory [10].

In post-copy migration, the container is first paused, and a minimal execution state is transferred to the destination host. The container will continue its execution in the target container, while missing pages will be retrieved from the source machine as required. This process results in fewer repeated page copies than in the case of pre-copy migration. The technique, however, is highly dependent on the source machine because the continuation of its execution is contingent on whether the source machine is available during the fetching process. In case the communication link breaks down, the service will be disrupted [11].

Hybrid migration includes elements of pre-copy and post-copy strategies. It involves copying a part of the container state before switching and moving the rest of the state after switching to the destination. The hybrid strategy tries to achieve a compromise between downtime, bandwidth, and failures [12].

Despite their differences in methodology and disadvantages, each of these migration approaches imposes a certain amount of overhead in terms of downtime, bandwidth consumption, power consumption, and risk of failure. Therefore, improving the migration mechanism alone is insufficient. It is equally important to determine whether migration should be performed in the first place, which motivates the framework proposed in this work.

B. Migration Decision Frameworks

Existing frameworks mainly address two questions: when to migrate and where to migrate. To address these questions, several strategies have been proposed in the literature, including threshold-based approaches, Markov Decision Process (MDP)-based frameworks, prediction-based methods, and resource-aware schemes. The question regarding optimal timing for migration has been extensively explored. Specifically, Wang et al. [6] solve the migration problem using a Markov Decision Process (MDP), in which the migration policy is developed by considering the mobility pattern of users and the distance between the user and the current edge node. Additionally, Shamsadini and Entezari-Maleki [13] utilize an MDP framework in their analysis of vehicular edge computing, and their migration decisions depend on path loss conditions resulting from the mobility of vehicles. On the other hand, in [14], migration occurs in response to node failures to ensure reliability and fault tolerance in the network. Latency minimization and service continuity are also considered. For instance, Tang et al. [15] develop a layer-aware migration approach to minimize the time required for initialization, computation, and migration. Further, Jin et al. [16] study container migration in industrial internet to reduce latency.

After making the decision to migrate, the destination edge server needs to be chosen. In [6], the destination edge is decided based on the policy obtained using the MDP model. In [17], an RNN-based trajectory prediction model is used to predict the future location of the vehicle and accordingly choose the destination edge server while taking into account the issues of service delay and load balancing. Network state and resource state information are used to select the destination in the mobile edge clouds to support real-time applications in [18]. In [15], container layer-sharing information from multiple users is used to select the destination edge server.

Different migration strategies have different optimization goals. For instance, some migration models prioritize reducing delays in service and keeping latency at a minimum [6] [17], while others place more emphasis on the fault tolerance aspect [14] [16]. Resource usage and migration costs are other critical issues tackled in traffic-aware and layer-aware systems [15] [18]. Despite the diversity of optimization objectives, these methods mainly seek to improve migration efficiency after the migration decision has already been made.

Despite the fact that these researchers use various techniques of optimization and prediction, they have something in common: the fact that migration must take place once the condition for that is met. As a result, prior research predominantly addresses the problem of when and where to migrate, but little attention has been paid to the necessity for the migration to take place at all. This issue drives the design of the proposed lightweight pre-migration decision scheme.

C. Migration Costs and Risks

Although container migration ensures service continuity, there are various costs and risks involved that should be taken into consideration prior to migration. One of the primary concerns is service downtime. Checkpointing involves pausing the container in order to save its state. The time taken by the container to save its state is directly dependent on the memory size of the container, as stated in [8]. For applications such as streaming videos and real-time analytics, which require near real-time responses, this disruption negatively impacts the user experience. If the migration process happens often, then these pauses tend to add up. Another important challenge is the accumulation of migration overhead caused by frequent migrations. Moving a container involves migrating the state of the memory of the container and the files. This consumes bandwidth that could otherwise serve user traffic. The authors of [18] noted that in edge networks with limited network bandwidth, this may lead to congestion. Migration operations also incur computational and energy costs. Checkpointing and data transmission use energy at both the source and destination edge nodes. The edge nodes generally run with limited energy resources, such as battery power, solar energy, or even electricity limitations. This is further discussed in [14]. Moreover, migration is not guaranteed to succeed. Network interruptions during migration can cause corruption of the container state. There may also be insufficient resources on the destination server during restoration. This may result in the failure of the entire system [14]. These challenges imply that enhancing the process of migration alone is not enough. It would be wise for an intelligent approach to not only decide how migration should take place, but also whether migration is desirable at all.

III. Research Gap and Problem Formulation

Despite the extensive research on the topic of container migration, the vast majority of proposed approaches concentrate on the problem of optimization of the migration process and the destination location for the container after migration.

Additionally, some works consider the problem of choosing the time of migration relying on threshold-based or model-driven triggers.

Nevertheless, all such approaches make the following assumption about the necessity of the migration once its triggering event occurs. The assumption does not take into account that the process of migration has a high cost and risk in terms of downtime, bandwidth consumption, energy consumption, and failures. And in some cases, the cost incurred could exceed the performance gain expected through migration. Moreover, network instability or transient resource congestion may be solved prior to the completion of the migration process, making such migration not only unnecessary but potentially harmful to QoS as well. Consequently, the problem of determining whether migration should be performed at all remains largely unexplored. Previous research on migration has focused primarily on answering the questions of when migration is needed and to which server migration must be made. The following question, however, remains largely unexplored:

Should migration be executed in the first place?: Therefore, there is a need to explore lightweight machine learning models for this pre-migration binary decision task. This research tries to fill this gap by presenting a machine learning decision model that can predict if migration is necessary according to the system state. Unlike prior work, this study introduces a lightweight pre-migration decision layer that filters unnecessary migrations before execution.

In this study, an MEC system is assumed where there are several edge servers, which provide cloud-based services to mobile clients via containerized application instances. As users move, the latency between the user and the serving edge node may increase, potentially degrading the QoS.

For each decision epoch t, the system will have a set of state variables reflecting the state of the system. Such state variables include current latency, post-migration latency, available bandwidth, container sizes, current CPU usage, and current memory usage, among others.

Given the above states, a decision needs to be made on whether migration should be started at epoch t or not. This problem can be viewed from the perspective of binary classification, where the target variable takes one of two values:

$$D(t) \in \{\text{Migrate}, \text{Wait}\}$$

where, Migrate indicates that migration should be executed immediately, whereas Wait indicates that migration should be postponed and the system should be re-evaluated at a later epoch.

The labeling of training samples is obtained using a counterfactual comparison strategy, described in detail in the following section.

Fig. 2 shows the position of the proposed framework in the MEC container migration pipeline. The framework acts as a pre-migration decision layer between the trigger condition and the migration executor, deciding whether to migrate or wait.

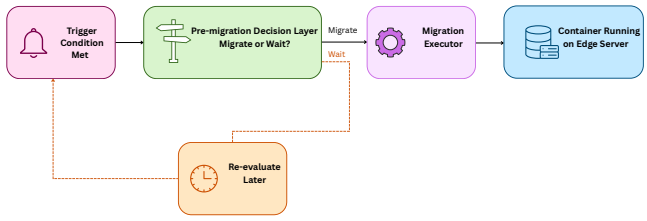

Fig. 2. Position of the proposed framework in the MEC migration pipeline.

IV. Methodology

This section describes the research methodology followed to make the classification problem for the container migration. The following subsections provide an explanation about each part of the experimental study, the dataset used, feature selection, models employed, and metrics used to evaluate the work.

A. Dataset

For this research, the utilized dataset was obtained via the use of EdgeSimPy [19], an open source Python simulator for edge computing. EdgeSimPy simulates a realistic MEC system that consists of 6 edge servers, 16 network switches arranged in a hexagonal mesh network, and 6 users operating containers in their devices. Fig. 3 illustrates the topology used in the dataset generation.

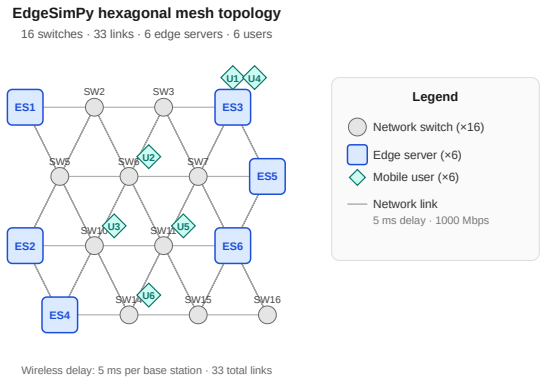

Fig. 3. EdgeSimPy simulation topology: a 4×4 hexagonal mesh of 16 network switches with 33 links, 6 edge servers, and 6 mobile users.

End-to-end latency is estimated within EdgeSimPy's routing engine by calculating the total wireless access latency at the base station (5 ms) and propagation time of the Dijkstra shortest path across the network (5 ms per hop). Thus, latency ranges from 5 ms to 25 ms. To obtain heterogeneous data, 30 simulation cases were conducted by varying two important variables in each simulation: container state size (8-200 MB) and mobility speed (5-35 seconds). Backbone link bandwidth was set to 1000 Mbps, consistent with real MEC infrastructure

TABLE I. DATASET FEATURES

Feature	Description	Range
latency_current_ms	End-to-end delay to current server	5–25 ms
latency_new_ms	Expected end-to-end delay after migration to candidate server	5–15 ms
latency_gain_ms	Difference in latency (migration improvement)	–5 to 20 ms
dist_current	Euclidean distance: user to current server	0–7.07
dist_new	Euclidean distance: user to candidate server	0–7.07
mobility	User movement direction: 0=stationary, 0.5=slow, 1.0=moving away	{0, 0.5, 1.0}
state_size_mb	Container state size in MB	8–200 MB
bandwidth_mbps	Available bandwidth on migration path in Mbps	Fixed at 1000 Mbps
dest_cpu_pct	CPU utilization on destination server	0–100%
dest_mem_pct	Memory utilization on destination server	0–50%
transfer_time_s	Estimated migration transfer time	0.07–1.64 s

specifications. The labeling scheme relied on a counterfactual comparison technique. In each situation, two simulations were conducted in the exact same environment: one where migration was never allowed, and another where migration was constantly conducted to the closest server at all times. At each decision time step, the end-to-end delay experienced in EdgeSimPy between the two policies was compared. If the delay value in the constantly migrating scenario was smaller than the one in the baseline scenario, then the time step was marked as Migrate; otherwise, it was Wait. As a point to note, the always migrate simulation algorithm employed for the generation of labels differs from the Always Migrate benchmark in Section VI, which makes predictions without regard to features or latencies, as opposed to the always migrate simulation algorithm, which generates latencies to generate labels. It is noted that although both feature generation and label derivation originate from the same simulation environment, the labels are derived from an independent comparison of latency outcomes under two distinct migration policies rather than directly from the input features, which attempts to mitigate the risk of closed-loop learning. The resultant data set has 5,000 samples, with each class having an equal number of samples (2,500 Migrate, 2,500 Wait), making the dataset a balanced dataset. Each sample consists of 11 features extracted from the simulation state at the time of the decision, summarized in Table I. Given that the hop delay for each link is constant at 5 ms, the latency will be an integer between 5 ms (co-located) and 25 ms (maximal hop distance in the network topology). Table II shows a sample of the simulation dataset generated for training the proposed machine learning model. Each row represents a system state, and the decision label assigned.

B. Decision Taking Model

A lightweight binary classifier is then constructed and trained on the labeled dataset generated. Input to the classifier will consist of system state features, and the output corresponds to one of two classes: Migrate or Wait. Thus, the classifier learns to determine whether migration should be executed immediately or postponed. Various machine learning and deep learning models have been used for analysis within this study, which include Logistic Regression, Decision Trees, Random Forests, Stacked Deep Neural Networks (DNN), and Stacked Long Short-Term Memory (LSTM) networks. This choice was made to allow the comparison of classical machine learning models and advanced deep learning models. Decision Trees were focused particularly in this work due to their suitability to MEC environments based on a number of advantages inherent to this algorithm. Firstly, decision trees do not require many computational resources for running and are therefore appropriate in resource-restricted edge systems. Secondly, decision trees make it possible to obtain understandable decision boundaries, which means that the process by which migration decisions are made can be easily explained. Finally, decision trees can detect nonlinear relations without scaling features.

1) Data preparation: The dataset of 5,000 records is divided into training, validation, and testing sets in a 70/10/20 split pattern, which results in 3,500 records being used for training, 500 records for validation, and 1,000 records being used for testing. The splitting ratio has been commonly used in machine learning literature in order to have enough data for training and still maintain an adequate test set. Since the synthetic dataset is balanced with 2,500 samples for each category, no form of resampling was performed. In addition, feature scaling did not take place since the key algorithm used in this project, which is the Decision Tree classifier, is insensitive to feature scaling.

2) Model architecture: The Decision Tree classifier was selected as the primary model because of its simplicity, interpretability, and low computational overhead. The model was trained using the maximum tree depth set to 3, 4, 5, 6, and 7, showing model performance across different depths, allowing the algorithm to have enough expressiveness to learn the non-linear dependencies among the features. All other hyperparameters were set to their default values. The model was trained using the training set containing 3,500 samples and tested on the test set containing 1,000 samples.

Along with the proposed Decision Tree approach, other alternative approaches were analyzed and compared with each other, including Logistic Regression, Random Forest, Stacked DNN, and Stacked LSTM. Also, two base methods were implemented: the always migrate, migration is made on every decision cycle, and the never migrate, migration is never performed. This helps evaluate the efficiency of the proposed decision framework. In addition, a latency-threshold rule was implemented as a realistic rule-based baseline, migrating whenever the expected latency gain exceeds a threshold tuned on the validation set.

C. Evaluation Metrics

To check how well the classification model performs on the task, several confusion matrix metrics were used. Several metrics were measured to give a clearer picture of the model's performance. Accuracy measurement was used as the primary metric, defined as the percentage of correctly classified cases compared to the total number of cases, giving an indication of how well the model classifies, as shown in Eq. (1):

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{1}$$

TABLE II. SAMPLE OF THE SIMULATED DATASET FOR TRAINING THE PROPOSED MODEL

Feature	Sample 1	Sample 2	Sample 3	Sample 4
Latency current (ms)	20	20	15	10
Latency new (ms)	10	10	10	5
Latency gain (ms)	10	10	5	5
Dist current	4.472	4.472	4	1.414
Dist new	2	1.414	1.414	0
Mobility	0.5	0.5	0.5	0.5
State size (MB)	18	10	25	12
Bandwidth (Mbps)	1000	1000	1000	1000
Dest CPU (%)	25	0	25	0
Dest mem (%)	50	0	50	0
Transfer time (s)	0.1475	0.0819	0.2048	0.0983
Label	Wait	Migrate	Wait	Migrate

In addition to accuracy, precision, recall, and F1-score were all measured for the model. Precision measures the number of truly positively classified cases to the number of cases classified as positive, as shown in Eq. (2):

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

Recall is defined as the ratio of correctly classified positives to all instances belonging to its own class, as shown in Eq. (3):

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3}$$

While the F1-score provides a balance between the two measures, precision and recall, as shown in Eq. (4):

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

The Area Under the ROC Curve (AUC) summarizes the trade-off between the True Positive Rate and False Positive Rate across all classification thresholds, serving as an aggregate measure of model performance, as shown in Eq. (5):

$$\text{AUC} \approx \sum_{i=1}^{n-1} \frac{(\text{FPR}_{i+1} - \text{FPR}_i)(\text{TPR}_i + \text{TPR}_{i+1})}{2} \tag{5}$$

V. RESULTS

This section presents a comprehensive analysis of the outcomes achieved from applying different machine learning and deep learning approaches. Results are presented in Fig. 4, Fig. 5, Table III, Table IV, Table V, and Table VI. Subsequently, a more detailed discussion of the outcomes is presented in the next section.

A. Comparison of Learning Models

Various machine learning and deep learning methods were trained on the training set and evaluated on the test dataset consisting of 1,000 cases. This experiment was aimed at identifying the best-fitting machine learning or deep learning method for the proposed pre-migration decision algorithm. Table III presents comparative results regarding the accuracy, precision, recall, F1-score, and AUC value for Logistic Regression, Decision Tree, Random Forest, Stacked DNN, and Stacked LSTM approaches.

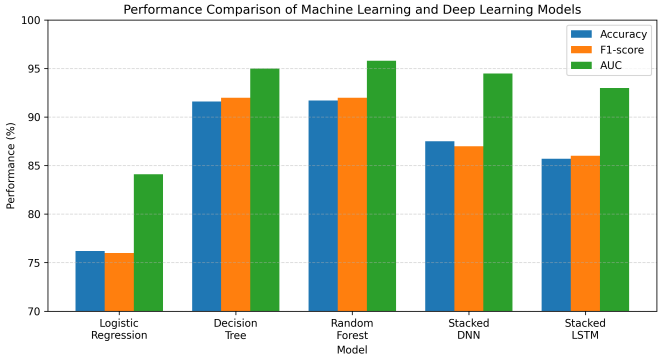

Fig. 4. Performance comparison of machine learning and deep learning models across accuracy, F1-score, and AUC.

As shown in Fig. 4, Logistic Regression achieved the lowest performance, achieving an accuracy of 76.20%, an F1-score of 76.11%, and an AUC of 84.13%. The poor performance indicates that linear models cannot be used to model the nonlinear interactions between the features related to migrations.

The deep learning models showed better results. The Stacked DNN was able to obtain an accuracy of 87.80% with an F1-score of 87.74% and an AUC of 94.38%, while the Stacked LSTM model yielded an accuracy of 85.70% with an F1-score of 85.66% and an AUC of 92.74%. Even though these models provided very promising predictions, they needed more computational capacity and engineering. The Decision Tree classifier achieved an accuracy of 91.60%, an F1-score of 91.58%, and an AUC of 94.86%, outperforming both deep learning models while maintaining low computational overhead and providing interpretable decision rules. Random Forest achieved very similar accuracy and a slightly higher AUC of 95.80%. However, Random Forest sacrifices interpretability and introduces additional computational complexity due to the ensemble structure. A latency-threshold rule was also included as a realistic rule-based baseline, achieving an accuracy of 60.40% and an F1-score of 56.87%, and is discussed in detail in Section VI-B.

Thus, the Decision Tree is the most optimal in terms of prediction accuracy, performance speed, and interpretability. This makes the model especially appropriate to be used as a pre-migration decision layer within MEC infrastructure.

To assess the statistical reliability of these results, a repeated

TABLE III. PERFORMANCE COMPARISON OF THE EVALUATED MACHINE LEARNING AND DEEP LEARNING MODELS

Model	Accuracy	Precision	Recall	F1-score	AUC
Logistic Regression	76.20%	76.61%	76.20%	76.11%	84.13%
Decision Tree	91.60%	91.92%	91.60%	91.58%	94.86%
Random Forest	91.80%	92.10%	91.80%	91.79%	95.80%
Stacked DNN	87.80%	88.51%	87.80%	87.74%	94.38%
Stacked LSTM	85.70%	86.08%	85.70%	85.66%	92.74%
Latency-Threshold Rule	60.40%	65.46%	60.40%	56.87%	60.40%

stratified k-fold cross-validation procedure was conducted (10 folds, 5 repeats, random seed 42) on the classical machine learning models, with results summarized in Table IV. The Decision Tree achieved a mean accuracy of 91.53% $\pm$ 1.25% and the Random Forest achieved 91.52% $\pm$ 1.29%, a difference of 0.01% with fully overlapping confidence intervals. This confirms that the performance gap between the two models is not statistically meaningful, and further supports the selection of the Decision Tree as the primary model on the basis of interpretability and computational efficiency rather than predictive performance alone.

To substantiate the claim that the Decision Tree is suitable for resource-constrained edge environments, runtime performance measurements were conducted and are summarized in Table V. The Decision Tree trained in 0.006 seconds and completed inference on 1,000 samples in 0.320 ms, with a serialized model size of 47.140 KB. By contrast, the Random Forest required 0.316 seconds to train, 22.785 ms for inference, which is approximately 71 times slower than the Decision Tree, and occupied 4,199.70 KB of memory, roughly 89 times larger. These measurements confirm that the Decision Tree is deployable on resource-constrained edge nodes, where memory and inference latency are critical constraints. All measurements were conducted on a standard CPU environment; absolute values would differ on dedicated edge hardware, but the relative advantages of the Decision Tree over the Random Forest remain consistent regardless of platform.

B. Effect of Decision Tree Depth

In order to analyze the effect of model complexity on the classification process, the Decision Tree classifier was tested for maximum depths varying between 3 and 7, in addition to the default model. Fig. 5 illustrates the impact of increasing the maximum tree depth on the Accuracy and AUC values of the Decision Tree classifier. The results attained from this process are listed in Table VI. As would be expected, the increase in the depth of the decision tree led to enhanced accuracy scores. A shallow tree having a depth of 3 gave accuracy and AUC values of 78.20% and 84.71%, respectively, suggesting inadequate representation power. With increases in tree depth, accuracy progressively increased up to 87.80% and 92.56% AUC at a depth of 7. The default Decision Tree classifier had superior performance levels, with accuracy and AUC being 91.60% and 94.86%, respectively. This suggests that the problem involves complex relationships requiring an expressive model. Nonetheless, shallower trees were also capable of performing effectively, demonstrating that lightweight models could be applied in edge environments.

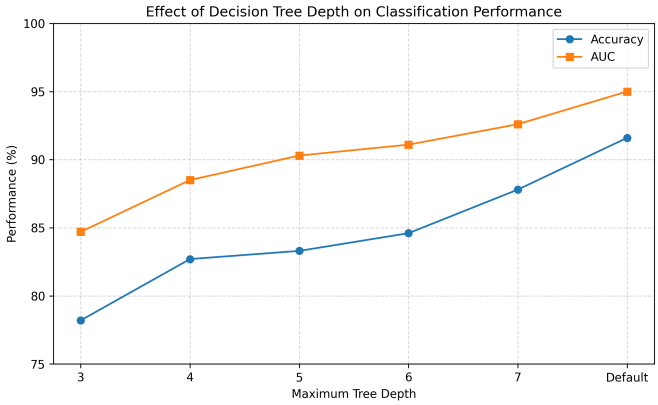

Fig. 5. Effect of decision tree depth on classification performance.

VI. DISCUSSION

A. Suitability of Decision Trees for MEC Environments

It can be seen from the results of the experiment that the Decision Tree classifier is an appropriate classifier to address the problem of pre-migration decision presented above. The Random Forest showed the highest AUC rate, achieving 95.80%, and also reached the accuracy of 91.80%. However, the advantage in comparison with the Decision Tree was insignificant. Meanwhile, the Decision Tree classifier achieved 91.60% of accuracy and 94.86% of AUC at significantly lower complexity of the model. When compared with other classifiers, which utilize more complicated architectures such as deep learning networks (Stacked DNN and Stacked LSTM), the Decision Tree is significantly more productive. It not only allows getting more accurate results but also does not need much computational power. Another advantage of the use of decision trees is their interpretability, meaning that the obtained rules could be easily interpreted. This feature is especially significant for MEC as edge nodes have relatively low computational capabilities [20]. Consequently, the Decision Tree classification algorithm provides an effective balance between accurate predictions, efficient computations, and interpretability, making it a suitable candidate for deployment as a pre-migration decision layer in edge computing infrastructures. Cross-validation results confirm that this performance advantage over the Random Forest is statistically consistent rather than an artefact of a specific train/test split, and runtime measurements provide direct evidence of the Decision Tree's computational suitability for resource-constrained edge deployment.

B. Comparison with Baseline Policies

Further analysis regarding the efficacy of the proposed framework can be performed through its comparison with three

TABLE IV. CROSS-VALIDATION RESULTS (10-FOLD, 5 REPEATS, RANDOM SEED 42) FOR CLASSICAL MACHINE LEARNING MODELS

Model	Accuracy	F1-score	AUC
Logistic Regression	77.96% ± 1.72%	77.85% ± 1.73%	85.34% ± 1.80%
Decision Tree	91.53% ± 1.25%	91.50% ± 1.26%	95.07% ± 1.24%
Random Forest	91.52% ± 1.29%	91.50% ± 1.30%	96.16% ± 0.94%

TABLE V. RUNTIME PERFORMANCE COMPARISON OF CLASSICAL MACHINE LEARNING MODELS, MEASURED ON A STANDARD CPU ENVIRONMENT. INFERENCE TIME IS REPORTED PER 1,000 TEST SAMPLES

Model	Training Time (s)	Inference Time (ms/1000 samples)	Model Size (KB)
Logistic Regression	0.350	0.224	0.790
Decision Tree	0.006	0.320	47.140
Random Forest	0.316	22.785	4199.700

TABLE VI. PERFORMANCE OF THE DECISION TREE CLASSIFIER FOR DIFFERENT MAXIMUM DEPTHS

Max Depth	Accuracy	Precision	Recall	F1-score	AUC
3	78.20%	79.63%	78.20%	77.93%	84.71%
4	82.70%	83.77%	82.70%	82.56%	88.45%
5	83.30%	83.57%	83.30%	83.87%	90.34%
6	84.60%	85.07%	84.60%	84.55%	91.08%
7	87.80%	88.00%	87.80%	87.78%	92.56%
Default	91.60%	91.92%	91.60%	91.58%	94.86%

baselines, two unconditional and one rule-based. The first approach migrates the data at each point of time, regardless of the system state. In contrast, the second policy never performs any migration actions. As demonstrated in Table VII, the considered simple baseline strategies provided equal performances in terms of accuracy and F1-score (both values being equal to 50.0% and 33.3%, respectively). Such an outcome seems quite predictable, since these approaches ignore the temporal dynamics of MEC systems and operate based on predetermined decisions. In comparison, the suggested Decision Tree model produced an accuracy rate of 91.60% and an F1-Score value of 91.58%, clearly outperforming the other two approaches. The obtained results have shown that random migration of containers or non-migration at all are ineffective approaches. To provide a more realistic point of comparison, a latency-threshold rule was also evaluated as an additional baseline. This rule migrates whenever the expected latency gain exceeds 0 ms — a simple and deployable heuristic that represents the kind of trigger-based policy commonly used in practice. Despite being more informed than the unconditional baselines, the latency-threshold rule achieved an accuracy of only 60.40% and an F1-score of 56.87%, with an MRR of 21.40% and a DRR of 24.02%. This result demonstrates that even a sensible single-feature rule is insufficient for this problem, and confirms that learning from the full system state across all 11 features is necessary to achieve reliable pre-migration decisions.

Beyond classification accuracy, the practical benefit of the proposed framework can be quantified directly in terms of migration overhead avoided. To this end, two additional metrics are introduced: Migration Reduction Ratio (MRR) and Downtime Reduction Ratio (DRR), both computed relative to the Always Migrate baseline. MRR captures the percentage decrease in the number of migrations triggered by the Decision Tree compared to a policy that migrates unconditionally, as shown in Eq. (6):

$$\mathrm{MRR} = \frac{M_{\mathrm{always}} - M_{\mathrm{DT}}}{M_{\mathrm{always}}} \times 100 \tag{6}$$

DRR captures the corresponding percentage decrease in cumulative transfer time, using the transfer_time_s feature described in Table I as a proxy for migration-induced downtime, as shown in Eq. (7):

$$\mathrm{DRR} = \frac{T_{\mathrm{always}} - T_{\mathrm{DT}}}{T_{\mathrm{always}}} \times 100 \tag{7}$$

where, M_{always} and T_{always} denote the number of migrations and total downtime under the Always Migrate policy, and M_{DT} and T_{DT} denote the corresponding values produced by the Decision Tree's predictions on the test set. Table VIII summarizes these results. Out of 1,000 test samples, the Always Migrate baseline triggers a migration at every decision epoch, resulting in 1,000 migrations and a cumulative downtime of 339.48 seconds. The Decision Tree, by contrast, predicts Migrate in only 544 cases, reducing the number of migrations by 45.60%. More notably, the cumulative downtime under the Decision Tree's policy falls to 98.79 seconds, a reduction of 70.90%. The disparity between the two reduction ratios indicates that the Decision Tree does not simply migrate less often at random; it disproportionately withholds migration in cases with higher transfer cost, while still permitting migration when the cost is low and the latency benefit justifies it.

The more crucial conclusion of this experiment is the fact that the obtained results provide evidence for the hypothesis of this study; namely, migration is not something mandatory after reaching the required trigger threshold. Thus, migration should be performed in a smart manner, taking into account the current state of the system and the balance between the advantages and disadvantages of migration.

C. Limitations of the Study

Despite the good performance of the suggested framework, there are still some limitations that need to be discussed. First, the dataset in the current study is simulated by means of the EdgeSimPy software package, instead of being collected from a

TABLE VII. COMPARISON OF THE PROPOSED FRAMEWORK AGAINST BASELINE MIGRATION POLICIES.

Policy	Accuracy	F1-score
Always Migrate	50.0%	33.3%
Never Migrate	50.0%	33.3%
Latency-Threshold Rule	60.4%	56.87%
Proposed Decision Tree	91.6%	91.58%

TABLE VIII. MIGRATION AND DOWNTIME STATISTICS FOR THE PROPOSED DECISION TREE RELATIVE TO BASELINE MIGRATION POLICIES.

Policy	Number of Migrations	Total Downtime (s)
Always Migrate	1000	339.48
Never Migrate	0	0.00
Latency-Threshold Rule	786	257.92
Proposed Decision Tree	544	98.79

live MEC implementation. While simulations provide a chance for controlled experiments with many possible mobility and container migration options, it is unlikely to completely reflect reality.

Additionally, even though the efficiency was calculated using the classification performance metrics, the influence of the current approach on the energy consumed, failure rate, and migration traffic was not studied. In addition, the dataset was generated from a single fixed topology of 6 edge servers, 16 network switches, and a backbone bandwidth of 1,000 Mbps. The generalisability of the reported performance figures to deployments with substantially different scale, heterogeneous network conditions, or varying bandwidth configurations remains to be investigated. Also, the proposed model was tested using offline experiments without integrating it into any existing container orchestration system, such as Kubernetes and KubeEdge. Hence, there is still a need for further testing for the efficiency of the proposed framework in practice.

VII. CONCLUSION

Container migration is one critical approach in ensuring service continuity in MEC. However, most previous research studies have concentrated on identifying the right time and location to perform container migration by taking the positive effect of the action for granted, even though there are numerous costs and risks associated with migration, such as service unavailability, network bandwidth usage, power consumption, and migration failures. Therefore, in order to solve the problem above, this study puts forward a pre-migration decision-making framework in the form of a binary classifier as a filter before migration. According to the present system status, the proposed approach will decide either to start the migration process or to wait until another period of time. A balanced dataset containing 5,000 samples was generated using EdgeSimPy, and labels were assigned through a counterfactual comparison strategy. Several machine learning and deep learning models were evaluated, including Logistic Regression, Decision Tree, Random Forest, Stacked DNN, and Stacked LSTM.

According to the experimental findings, Decision Tree had an accuracy of 91.60%, an F1-score of 91.58%, and AUC of 94.86%, which is better than the deep learning algorithms examined in terms of computation costs and interpretability of decisions. Moreover, the presented solution provided much better results than all evaluated baseline migration strategies, including the unconditional Always Migrate and Never Migrate policies and a realistic latency-threshold rule, confirming the necessity of intelligent migration decision-making within MEC. Cross-validation results further confirm that the Decision Tree's performance advantage is statistically consistent, and runtime measurements demonstrate its suitability for deployment on resource-constrained edge nodes.

In conclusion, it can be stated that migration cannot be treated as an automatic action based on certain triggers. Rather, the introduction of the decision-making component can ensure avoiding unnecessary migrations and make edge computing more efficient.

A. Future Work

There are various areas for future work. The proposed decision-making framework can be tested with practical trace datasets collected from live edge computing environments, and validated across additional simulation environments with varied topologies, network configurations, and bandwidth conditions, as well as deployed into practice within orchestration engines like Kubernetes and KubeEdge. Secondly, other objectives, including energy usage, migration probability of failure, and bandwidth utilization, can be considered as part of the decision process. Furthermore, reinforcement and online learning algorithms can be looked into to achieve adaptive migration policies. Additionally, runtime performance measurements on dedicated edge hardware platforms, such as Raspberry Pi or similar resource-constrained devices, would provide more precise evidence of the framework's suitability for edge deployment. Statistical significance analysis, including cross-validation, should also be extended to deep learning models such as the Stacked DNN and Stacked LSTM evaluated in this work. Finally, evaluating the framework on larger and more heterogeneous edge infrastructures would provide further insights into its scalability and applicability.

REFERENCES

[1] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE communications surveys & tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE internet of things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[3] S. Alshattnawi and A. M. AlSobeh, "A cloud-based iot smart water distribution framework utilising bip component: Jordan as a model," *International journal of cloud computing*, vol. 13, no. 1, pp. 25–41, 2024.

[4] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[5] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[6] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge computing based on markov decision process," *IEEE/ACM Transactions on Networking*, vol. 27, no. 3, pp. 1272–1288, 2019.

[7] T. Taleb and A. Ksentini, "Follow me cloud: interworking federated clouds and distributed mobile networks," *IEEE Network*, vol. 27, no. 5, pp. 12–19, 2013.

[8] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proceedings of the second ACM/IEEE symposium on edge computing*, 2017, pp. 1–13.

[9] A. Tošić, "Run-time application migration using checkpoint/restore in userspace," *Journal of Web Engineering*, vol. 23, no. 5, pp. 735–748, 2024.

[10] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, 2005, pp. 273–286.

[11] M. R. Hines, U. Deshpande, and K. Gopalan, "Post-copy live migration of virtual machines," *ACM SIGOPS operating systems review*, vol. 43, no. 3, pp. 14–26, 2009.

[12] S. Sahni and V. Varma, "A hybrid approach to live migration of virtual machines," in *2012 IEEE international conference on cloud computing in emerging markets (CCEM)*. IEEE, 2012, pp. 1–5.

[13] A. Shamsadini and R. Entezari-Maleki, "Time-aware mdp-based service migration in 5g mobile edge computing," in *2022 27th International Computer Conference, Computer Society of Iran (CSICC)*, 2022, pp. 1–5.

[14] S. Aleyadeh, A. Moubayed, P. Heidari, and A. Shami, "Optimal container migration/re-instantiation in hybrid computing environments," *IEEE Open Journal of the Communications Society*, vol. 3, pp. 15–30, 2022.

[15] Z. Tang, F. Mou, J. Lou, W. Jia, Y. Wu, and W. Zhao, "Multi-user layer-aware online container migration in edge-assisted vehicular networks," *IEEE/ACM Transactions on Networking*, vol. 32, no. 2, pp. 1807–1822, 2024.

[16] X. Jin, S. He, and Y. Chen, "Container migration for edge computing in industrial internet with joint latency reduction and reliability enhancement," *Scientific Reports*, vol. 14, no. 1, p. 25394, 2024.

[17] W. Zhang, J. Luo, L. Chen, and J. Liu, "A trajectory prediction-based and dependency-aware container migration for mobile edge computing," *IEEE Transactions on Services Computing*, vol. 16, no. 5, pp. 3168–3181, 2023.

[18] S. Maheshwari, S. Choudhury, I. Seskar, and D. Raychaudhuri, "Traffic-aware dynamic container migration for real-time support in mobile edge clouds," in *2018 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, 2018, pp. 1–6.

[19] P. S. Souza, T. Ferreto, and R. N. Calheiros, "Edgesimpy: Python-based modeling and simulation of edge computing resource management policies," *Future Generation Computer Systems*, vol. 148, pp. 446–459, 2023.

[20] S. Alshattnawi and M. Al-Marie, "Spider monkey optimization algorithm for load balancing in cloud computing environments." *Int. Arab J. Inf. Technol.*, vol. 18, no. 5, pp. 730–738, 2021.