DECISION TREES-BASED ALGORITHM FOR INTELLIGENT ALLOCATION OF PROCESSES IN CLOUD

Constantin-Valentin DOLCESCU[1], Robert BOTEZ [1], Daniel ZINCA[1], Virgil DOBROTA[1]
[1]*Communications Department, Technical University of Cluj-Napoca, Romania*
Dolcescu.Io.Constant@student.utcluj.ro; Robert.Botez@com.utcluj.ro, Daniel.Zinca@com.utcluj.ro
Corresponding author: Virgil Dobrota (e-mail: Virgil.Dobrota@com.utcluj.ro)

Abstract: The paper presents a decision tree–based scheduler for intelligent cloud process allocation that evaluates features such as source area, instruction count, payload size, priority, throughput, and delay to guide real-time placement decisions. The model was trained and validated on a diverse, scenario-driven synthetic dataset covering four controlled workload conditions plus randomized fallback cases. For the training dataset, the classifier achieved 93% accuracy, while for the validation and test set, an accuracy of 92% was obtained. A Kubernetes-inspired simulation framework further visualizes and confirms the scheduler's allocation logic under dynamic conditions. These results underscore the approach's effectiveness, interpretability, and suitability for production-grade cloud orchestration.

Keywords: *cloud computing, Decision Trees, Machine Learning, process allocation, resource optimization.*

I. INTRODUCTION

Cloud computing enables on-demand access to both software and hardware resources, including storage, central processing units (CPUs), graphics processing units (GPUs), tensor processing units (TPUs), and memory [1]. Undoubtedly, transitioning from on-premises infrastructure to the cloud reduces a company's capital expenses but increases operational costs. The cloud follows a pay-as-you-go approach, allowing businesses to pay only for the resources they consume. This paradigm facilitates the dynamic distribution of resources based on demand, ensuring both flexibility and efficiency. The three primary cloud computing service models are Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [2]. Cloud orchestration refers to the process of organizing and overseeing the deployment and operation of cloud infrastructure, where automation plays a crucial role in efficiently integrating services and optimizing workflows. This leads to increased productivity and improved service quality for both users and providers [3]. Orchestrating IaaS focuses on managing physical and virtual resources, such as servers and storage, ensuring their optimal allocation and maintenance. In PaaS, the management and coordination of databases, middleware, and other components support the development and deployment of applications. Meanwhile, SaaS automates corporate operations, manages application interdependencies, and ensures the efficient delivery of software services.

Orchestration can be implemented using imperative paradigms, which follow predefined steps to achieve a desired state, or declarative paradigms, which define the final state without specifying the exact processes required. In multi-cloud environments, security is a top priority, involving credential management, continuous monitoring, and adherence to security policies to ensure confidentiality, integrity, and availability of resources [4]. These factors contribute to creating secure, scalable, and resilient orchestration frameworks tailored to diverse user requirements.

Kubernetes [5] is an open-source platform designed for orchestrating cloud resources, automating the deployment, scaling, and management of containerized applications. In a dynamic cloud environment where resource demands fluctuate constantly, an intelligent system is essential to distribute workloads efficiently, preventing node overload and ensuring balanced utilization of available infrastructure. A core component of Kubernetes is its scheduler, kube-scheduler, which analyzes available resources and determines the optimal allocation of containers within a cluster. It evaluates each node based on a well-defined set of rules, aiming to minimize congestion and optimize execution times. The decision-making process relies on two key algorithms: Predicate and Priority, which work together to identify the most suitable node for each task. In the first phase, Predicate algorithms filter out nodes that do not meet the specific requirements of a container, considering factors such as available memory, processing power, affinity constraints, and compatibility with required resources. This initial filtering ensures that only viable options remain. Next, Priority algorithms assess the remaining nodes, ranking them based on criteria such as resource utilization, network delay, and load levels. This approach enables Kubernetes not only to select a valid node but also the most efficient one for running the application, optimizing resource consumption, and distributing workloads evenly. To further enhance this process, recent research explores the application of deep learning and reinforcement learning techniques to improve Kubernetes' automated scheduling algorithms. Studies indicate that integrating these methods allows the platform to better anticipate future resource demands and adjust allocations in real time, thereby enhancing overall cluster performance and reducing wait times for critical processes [6].

Artificial intelligence (AI) plays a crucial role in cloud orchestration by optimizing the dynamic allocation of resources and reducing costs through predictive methods

Manuscript received April 18, 2025; revised June 25, 2025

based on machine learning (ML). AI enables proactive scaling, load balancing, and system performance improvements, leveraging algorithms such as deep learning and reinforcement learning [7].

Decision Trees are used to efficiently classify and allocate resources, dynamically adapting to user demands and network conditions while minimizing delays and energy consumption. Additionally, Edge AI enhances performance and privacy by processing data locally, reducing network traffic and response times. Overall, integrating AI into cloud orchestration enhances efficiency and flexibility, ensuring high-quality service delivery in dynamic and complex environments [8].

Resource optimization in the cloud through machine learning (ML) is essential, as algorithms can adjust resources in real-time, predict demands, and identify bottlenecks, opening new opportunities for efficient allocation. In [9], the authors propose using ML to maximize resource utilization in telecommunication networks, leveraging cloud-based data storage and analysis to support effective decision-making.

Furthermore, in another research [10], it was introduced a Random Forest ensemble for cloud resource management, reporting up to a 30 % improvement in utilization prediction over single-model techniques such as XGBoost, Ridge regression, and Lasso. By aggregating decisions from multiple trees, this approach achieves robust performance in both classification (assigning processes to worker nodes) and regression (forecasting resource demand), thereby enhancing system efficiency and reducing energy consumption. The authors also evaluate Support Vector Machines (SVM) for multi-class process prediction, finding that SVM consistently outperforms Gaussian Naive Bayes when trained on sufficiently large and diverse datasets. These findings underscore the importance of ensemble and kernel-based methods—and the need for representative training data—in dynamic, data-driven cloud environments.

We explored in this paper the use of machine learning to enhance process allocation mechanisms in cloud environments, with a particular focus on decision trees for task classification and distribution. Our contributions are – design and implementation of a decision tree–based classifier for intelligent process allocation using features such as area, instruction, size, priority, throughput and delay; development of a synthetic, scenario-driven dataset covering four controlled workload conditions plus a randomized fallback; demonstration of model robustness by achieving 92 % accuracy on the test set; and creation of a Kubernetes-inspired simulation framework (Cloudlet and Master classes) to visualize and validate allocation decisions in real time.

The remainder of the paper is structured as follows. Section I combines the introduction with a review of related work on machine-learning approaches to cloud resource management. Section II details our implementation, including the decision tree–based allocation model, feature engineering, and dataset generation framework. Section III presents the experimental results, covering model accuracy, confusion matrices, ROC curves, and simulation case studies—in particular, it also discusses our dataset splitting strategy, hyperparameter choices (e.g., *random_state*, *max_depth*), and feature-importance analysis. Finally, Section IV concludes the paper, summarizes our key findings, and outlines future work, including a planned comparison with existing scheduling algorithms and the extension of our framework to support adaptive, real-world cloud deployments.

II. IMPLEMENTATION

The proposed model (see Figure 1) analyzes multiple characteristics of each process, including priority, size, number of instructions, data throughput, and associated delays, to determine the optimal allocation.

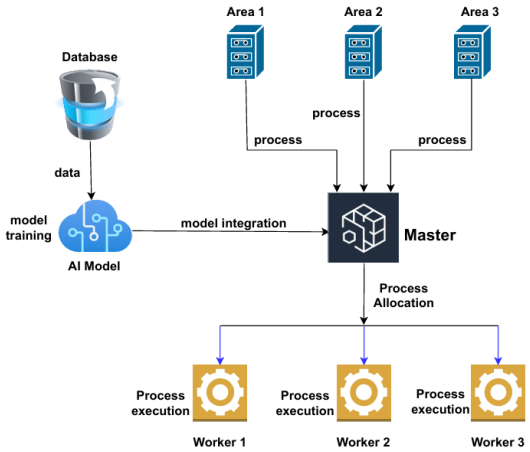

Figure 1. Model architecture

This approach aims not only to improve system efficiency but also to reduce execution times and operational costs associated with cloud resource usage. By integrating these techniques, the system gains the ability to make real-time decisions, dynamically adapting to network fluctuations and variable user demands. Experimental results demonstrate that the decision tree-based model ensures more balanced process distribution and performance optimization, contributing to better resource management in cloud environments.

Inspired by the principles used in the Kubernetes scheduler, this model seeks to optimize resource allocation through an intelligent task distribution mechanism, reducing execution time. These characteristics are essential for determining the optimal allocation, as each process may have different execution requirements. Inefficient distribution could lead to overloading certain resources and lowering the overall system performance. In this architecture, areas 1, 2, and 3 represent different process sources, each with specific characteristics. These process sources may correspond to distinct service types—such as real-time streaming jobs, batch analytics tasks, or user-interactive requests-each exhibiting unique patterns of priority, size, throughput, and delay. By modeling them as separate areas, the scheduler can learn and exploit these source-specific behaviors, yielding more accurate and context-aware allocation decisions under varying workload conditions. These areas transmit their processes to a Master Node, responsible for centralizing and analyzing data. The Master Node plays a crucial role as it communicates directly with the AI model, which evaluates the parameters of each process and determines the most suitable worker for execution. In this way, tasks are intelligently allocated based on the capacity and availability of system resources.

Once the model makes a prediction, the process is automatically directed to the appropriate worker, which then executes it. This method not only optimizes resource utilization but also reduces execution latencies, ensuring a balanced distribution of tasks within the system. Additionally, due to its dynamic nature, this system can adjust process allocation in real-time, responding quickly to load fluctuations and network condition variations. By utilizing a decision tree-based model, the proposed architecture significantly enhances cloud resource management, enabling more efficient scheduling, reduced execution times, and greater infrastructure scalability.

The first stage involved defining and creating a well-organized database structure, as in Figure 2, essential for the efficient training of the machine learning model.

Area	Instructions	Size (MB)	Priority	Delay	Throughput (MB/s)	Score	Assigned Worker
Area X	1–100	1–10	High	1–30	51–100	71–100	1
Area X	101–200	11–20	Medium	31–70	25–50	30–70	2
Area X	201–300	21–30	Low	71–100	1–24	1–29	3

Figure 2. Database structure

This was designed to include all relevant features necessary for the classification process, such as process priority, data throughput, delays, and the number of instructions. To increase the precision of the model and allow a better numerical representation, all range-based features (originally defined as intervals) were split into separate columns representing their lower and upper bounds. This transformation ensured greater control during the learning phase and enabled a more flexible classification mechanism. The following adjustments were made:

- Instructions: split into `InstructionsMin` and `InstructionsMax`, representing the operational range of a task
- Delay: split into `DelayMin` and `DelayMax` denoting the latency span in [ms].
- Throughput: split into `ThroughputMin` and `ThroughputMax` indicating the transfer rate interval in [Mbps].

The structure was optimized to enable fast and accurate data processing, ensuring the model has access to relevant and well-organized information. It has been designed to include a variety of scenarios, allowing the model to learn correct patterns and make more precise predictions. Through this optimization, the system guarantees an efficient distribution of resources, reducing execution time and enhancing overall system performance. With the data structure in place, the next step focused on extracting and refining key features to maximize the model's learning capabilities.

The database structure plays a crucial role in training the decision tree-based machine learning model, ensuring accurate classification and efficient process distribution to the appropriate workers. Each feature included in the database is carefully selected to contribute to optimal decision-making, allowing the model to intelligently anticipate and manage resource allocation. A key factor is the number of instructions, which defines the volume of operations a worker must execute to complete a process. This characteristic directly influences execution time and worker load, having a significant impact on overall system performance. Closely related to this is process size, which reflects the amount of data required for execution. A larger process may demand more resources, prompting the model to select a worker capable of handling it without compromising the efficiency of other active processes. Another essential attribute was priority, which determined the order in which processes were allocated and executed. A process can have high, medium, or low priority, and this classification affects how quickly it must be completed. High-priority processes are allocated immediately and must be executed as quickly as possible, while medium-priority processes should be completed within a reasonable timeframe. In contrast, low-priority processes are treated with more flexibility, without strict time constraints. Data throughput between the Master Node and the workers is a fundamental parameter, indicating the speed at which information is transferred. Higher throughput enables processes to be transmitted and executed more rapidly, making workers with high-performance connections preferable for handling critical tasks. The delay in [ms] represented the time required for data to travel from the Master Node to the worker. Higher delays could negatively impact system responsiveness, reducing the overall efficiency of resource allocation. To handle cases where multiple processes shared the same priority level, but having differing characteristics, an allocation score was introduced. To ensure a more granular and precise representation, each feature that could potentially contain a range of values—such as instructions, throughput, and delay—was split into two separate columns: one for the minimum value and one for the maximum. This design choice addresses limitations of using aggregated or interval values in a single column, which are unsuitable for machine learning algorithms that require distinct numerical features for proper pattern recognition and decision-making. Furthermore, a score was dynamically computed for each process based on the following adapted formula:

$$RawScore = \frac{K_0 * S_0}{Throughput} + K_1 * S_1 * Delay \qquad (1)$$

As defined in (1), a drop in available bandwidth produces a disproportionately large inverse term, which increases the score. In this scheme, higher scores reflect poorer efficiency and impose a greater penalty: processes with both low throughput, and high delay receive the highest scores, guiding the scheduler to deprioritize them in favor of more efficient tasks. In this formulation, the constants K_0 and K_1 act as weights to emphasize the relative importance of throughput versus delay, while S_0 and S_1 are scaling factors that normalize the corresponding metrics to ensure consistent units and comparable magnitudes [11].

This score reflected both transmission efficiency and latency constraints, combining minimum and maximum values for both throughput and delay to better capture variability in system performance. To ensure compatibility with classification models, this raw score was normalized using min-max scaling into a standardized interval [1,100] The scaling process was implemented through the following transformation:

$$Score = \frac{(RawScore - MinScore) * (100 - 1))}{MaxScore - MinScore} + 1 \qquad (2)$$

To mimic realistic operational conditions and to allow the model to learn a wide spectrum of decision boundaries,

our dataset is composed of data generated under multiple controlled scenarios. Each scenario is defined by a distinct set of constraints on throughput, delay, instructions, process size, and priority.

Condition 1 represents high-performance processes where both throughput values and allocation scores are high, delays remain low, and the number of instructions is confined within a narrow range. This scenario is designed such that most processes are allocated to Worker 1 (with an 80/20 split in favor of Worker 1 versus Worker 2).

Condition 2 covers processes characterized by moderate throughput and delay values, with instruction counts in a slightly higher range compared to Condition 1. In this scenario, the data generation focuses on predominantly allocating processes to Worker 2 (again, using an 80/20 distribution relative to Worker 1).

Condition 3 corresponds to processes with lower throughput, higher delay, and higher instruction counts. Under these conditions, processes are primarily allocated to Worker 3 (with a 90/10 distribution compared to Worker 2), simulating scenarios of resource-intensive workloads or degraded network conditions.

Condition 4 includes processes with high throughput but significant variations in delay, coupled with larger process sizes and higher instruction counts. This scenario is tailored for Worker 3 (with a 90/10 split vis-à-vis Worker 1), capturing complex operational conditions where high throughput is counterbalanced by considerable delay.

These x/y distributions ensure that each scenario heavily favors the intended worker node—reflecting the optimal placement given that scenario's performance profile—while still including a minority of tasks on the alternate node to maintain diversity. A randomized fallback assigns any process that does not meet a condition's thresholds equally across all three workers, further enriching the dataset with atypical cases and improving the classifier's ability to generalize.

Not every process meets the specific criteria defined in the four main conditions. The fallback scenario captures such edge cases by randomly assigning processes among all workers. This ensures that the dataset includes atypical instances, thereby enhancing the model's ability to generalize by covering rare conditions that might occur in real-world settings.

For each controlled scenario, dedicated functions (`generate_condition_1`, `generate_condition_2`) generate synthetic data by sampling feature values (throughput, delay, instructions, size, and priority) within predefined ranges. The `calculate_score()` function computes a raw score based on throughput and delay, applying a minor random factor (±1%) to simulate natural network fluctuations and introduce controlled noise. This randomness helps avoid overfitting during model training.

The raw scores are then normalized to the interval [1, 100] using the `scale_score()` function. In this normalization, the minimum and maximum scores (`MinScore` and `MaxScore`) are determined based on theoretical system limits—representing the best-case scenario (low delay, high throughput) and the worst-case scenario (high delay, low throughput), respectively. This transformation ensures that all scores are directly comparable and interpretable, regardless of their original raw values.

Finally, outputs from all scenario-specific functions, along with the fallback data, are merged and shuffled into a single DataFrame. This combined dataset, which consists of over 1.4 million records, is then split into training, validation, and test subsets to ensure robust model evaluation and generalization. The synthetic dataset covers a broad range of realistic process characteristics for Area X. Instruction counts are sampled from three ranges (1–100, 101–200, 201–300), sizes span small (1–10 MB), medium (11–20 MB) and large (21–30 MB), and priority levels include High, Medium, and Low. Network throughput is drawn from 1 to 24 megabytes per second, 25 to 50 megabytes per second, or 51 to 100 megabytes per second, while delays vary between 1–30 ms, 31–70 ms, and 71–100 ms. Each combination of these intervals is assigned one of three score tiers—Low (1–29), Mean (30–70), or High (71–100)—and mapped to the optimal worker (1, 2, or 3). This design ensures that all feature ranges and their interactions are well represented in the training data. With the data normalized and structured to accurately reflect diverse, real-world conditions while mitigating overfitting through controlled randomness, we now proceed with a detailed analysis and further preparation. Understanding the feature distribution within the dataset is crucial to ensure that the decision tree-based model can generalize properly and avoid issues such as overfitting or undertraining. In the following section, we describe the steps taken to analyze and preprocess the data prior to training, thereby setting the stage for efficient and precise process allocation predictions.

To ensure the robustness of the model, the dataset was generated to include a wide range of values, allowing the model to learn from a broad spectrum of possible scenarios (see Figure 3). After min–max normalization into the [1–100] range, the data are partitioned into three score databases—Low Score (1–30), Mean Score (31–70), and High Score (71–100). Each database then feeds the four scenario generators (Conditions 1–4) plus a randomized fallback, ensuring that every combination of score tier and workload condition is represented in the final merged dataset.

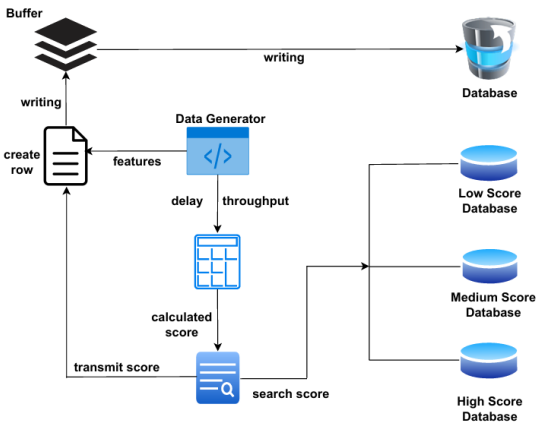

Figure 3. Data generation flow

This data diversity is crucial for enabling the algorithm to recognize general patterns and adapt to unseen data. Additionally, a balanced distribution of features was considered to prevent the model from being disproportionately influenced by specific data classes, avoiding imbalances that could negatively impact prediction accuracy. A key aspect of data preparation

involves normalization and standardization; processes necessary to bring all feature values into the same numerical range. These techniques are useful in addressing issues such as skewed distributions or outliers, which could otherwise affect the model's performance. Without these transformations, the model might assign excessive importance to certain high-value features simply due to their different scales, leading to misleading predictions.

Additionally, to properly evaluate the model's performance and prevent overfitting, validation techniques and data splitting were applied. The dataset was divided into three categories: (1) *training set:* which enables the model to learn general rules, identify patterns, and establish relationships between features; (2) *validation set:* used for fine-tuning hyperparameters and adjusting the model to enhance its accuracy and efficiency; and (3) *testing set:* designed to evaluate the model's performance on new, unseen data, ensuring an objective assessment of its ability to generalize to different scenarios. This structured approach enhances the model's adaptability, reducing the risk of overfitting while improving its predictive accuracy. To further enhance model generalization and avoid overly simplistic patterns, the dataset was synthetically generated using controlled randomness, with multiple combinations of `InstructionsMin` and `InstructionsMax` matched against varying `ThroughputMin`, `ThroughputMax`, `DelayMin`, and `DelayMax`. This approach ensured that processes with similar instruction loads could be classified differently depending on other factors such as network speed or latency, encouraging the model to learn deeper correlations rather than relying on a single feature. Each record was labeled with the target class `Allocated_Worker`, corresponding to the optimal worker (1, 2, or 3) for that process.

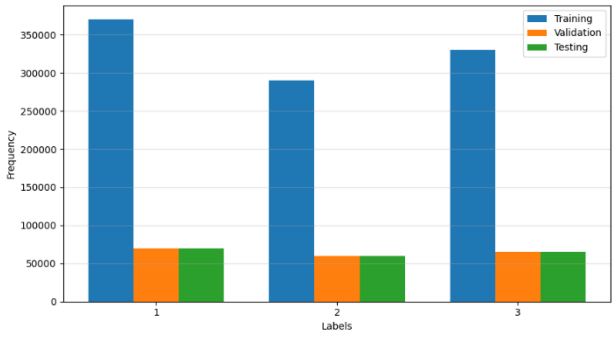

Figure 4. Data splitting for training, testing, and validation

Data preprocessing was a fundamental step in preparing the machine learning model, incorporating essential processes such as shuffling, normalization, transformation, and feature extraction, all aimed at improving data quality and ensuring efficient learning. For the model to make accurate predictions and adapt effectively to new data, careful splitting of the dataset into three distinct categories was required, each serving a well-defined role in the training and optimization process. For the model to make accurate predictions and adapt effectively to new data, the dataset was carefully split into three subsets (see Figure 4):

- 70% Training set – used to help the model learn fundamental patterns and relationships.
- 15% Validation set – employed during the tuning of hyperparameters and prevention of overfitting.
- 15% Testing set – used to objectively assess model performance on unseen data.

This structured and proportionally balanced approach ensures that the model generalizes well, it avoids bias toward specific classes, and it is properly evaluated before deployment. To achieve a stable and high-performing model, the `DecisionTreeClassifier` algorithm was implemented with a set of parameters designed to balance complexity and accuracy. The `random_state` parameter was set to 42 to initialize the pseudo-random number generator used for data shuffling and tree construction, ensuring that each execution produces the same train/validation/test split and identical tree topology, while `max_depth` was limited to 5 to prevent overfitting, keeping the decision tree simple enough to remain interpretable yet complex enough to differentiate between classes effectively. Additionally, `min_samples_split` was set to 10 to prevent excessively small tree splits, ensuring that each node contained enough data for meaningful statistical significance. The `min_samples_leaf` parameter was set to 5, helping to avoid fluctuations caused by imbalanced datasets and ensuring a more robust tree structure.

During training, the model processed the dataset using a recursive partitioning mechanism, based on criteria such as Gini impurity, allowing each node to be optimized for maximum data separation efficiency. At each training step, the decision tree identified splitting points that provided the clearest distinction between classes, dividing the data into more homogeneous subsets and gradually reducing impurity. This process continued until the stopping conditions were met, such as reaching the predefined maximum depth or an insufficient number of samples to perform further splits.

To enhance the model's ability to make accurate predictions on new data, pruning techniques were applied, removing redundant elements and nodes that did not significantly contribute to classification. This optimization allowed the model to generalize more effectively, preventing it from rigidly memorizing specific structures from the training set and instead providing more accurate predictions in diverse scenarios.

After training was completed, the model was saved using the `joblib` library, enabling future reuse without requiring re-training each time it is applied to a new dataset. This approach not only optimizes execution time but also allows the model to be integrated into a larger system, where it can be used for real-time automated predictions, ensuring efficient resource allocation and optimized process management in the cloud environment.

III. EXPERIMENTAL RESULTS

The Decision Tree-based classifier demonstrated outstanding performance in classifying data, even under conditions where the test set included slight intentional disturbances. The final model achieved a test accuracy of 92.32%, maintaining a strong balance between precision, recall, and F1-score across all classes. See Figure 5 for the classification report on the test dataset.

Class (Worker)	Precision	Recall	F1-Score	Support
Worker 1	0.94	0.88	0.91	81,012
Worker 2	0.91	0.93	0.92	61,493
Worker 3	0.92	0.97	0.94	71,995
Macro Avg	0.92	0.93	0.92	
Weighted Avg	0.92	0.92	0.92	

Figure 5. Classification report

These metrics are also reflected in the confusion matrix (Figure 6), where most predictions align closely with the true labels. Out of a total of 214,500 test samples, the model correctly classified most instances, with a very low rate of false positives and false negatives, especially for Classes 2 and 3. For example, 71,485 instances from Class 1 were correctly predicted, while 3,704 were misclassified as Class 2 and 5,823 as Class 3. This class still retained a high precision due to the relatively large number of correctly predicted samples compared to misclassifications. In the case of Class 2, 56,896 samples were accurately predicted, with only a combined total of 4,597 misclassifications across the other two classes. Class 3 showed particularly strong performance, with 69,653 correctly classified instances, and only 423 and 1,919 misclassified as Class 1 and Class 2 respectively — leading to a precision of over 96%. These results illustrate the model's discriminative power and confirm that it effectively distinguishes between subtle differences in feature patterns that define each class.

What stands out from this confusion matrix is the model's ability to maintain a balanced performance across all classes, without favoring any label — a crucial aspect in multi-class classification problems. The decision tree's structure contributes significantly to this balance, starting with "Score" as the root decision node and gradually refining predictions based on `DelayMin`, `ThroughputMax`, and `Priority`. This logical flow mimics expert decision-making in process allocation, where multiple resource parameters must be evaluated simultaneously.

Moreover, the model's performance in this test scenario validates its robustness under realistic conditions, including potential noise or fluctuation in input data. By maintaining high recall and F1-scores across all categories, the classifier proves its readiness for deployment in dynamic cloud environments. In such systems, the ability to generalize from a wide range of scenarios — as seen in this dataset — is essential to ensure efficient, intelligent, and adaptive resource scheduling.

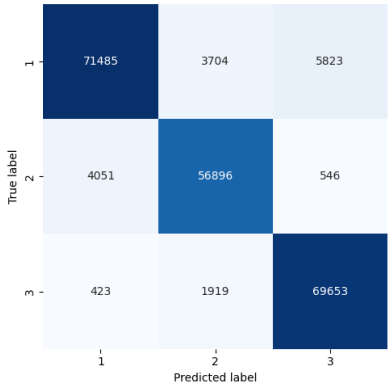

Figure 6. Confusion matrix

Before analyzing the decision tree structure in detail, it is important to understand the classification labels used during training. The model was designed to classify processes into one of three distinct classes — Worker 1, Worker 2, or Worker 3 — based on a set of features including throughput, delay, process size, and priority. During dataset generation, the training set maintained a balanced distribution among the three classes, with a slightly higher concentration in Class 1 due to the characteristics of high-priority, high-throughput tasks. This balanced but realistic representation ensured that the model learned patterns from all categories effectively, minimizing bias toward any specific worker during decision-making.

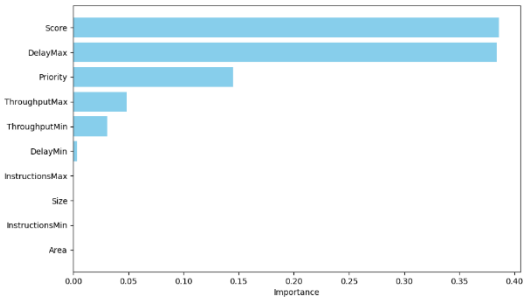

Figure 7. Labels balanced distribution

To gain insights into how the model makes its predictions, an analysis of feature importance was performed. The most influential feature turned out to be the Score, which combines throughput and delay into a single indicator of process efficiency. This is consistent with the model architecture, where the score acts as the primary splitting criterion in the decision tree. The next most important feature was `DelayMax`, highlighting the impact of network delays on the system's decision-making. Priority also played a significant role, reinforcing the assumption that tasks marked as high priority must be processed faster and more efficiently. On the other hand, features such as `InstructionsMin`, `InstructionsMax`, and `Area` had a minimal impact on the final classification. This suggests that, within the current decision tree configuration, scheduling-related factors such as delay, throughput, and task priority have a greater influence on the allocation decision than static process characteristics like instruction count or size.

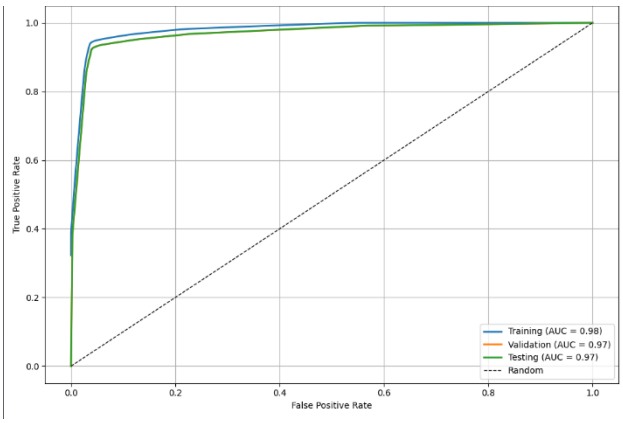

Figure 8. ROC Curve comparing training, validation, and testing sets

The ROC (Receiver Operating Characteristic) curves reveal consistently high classification capability across all data subsets, with an AUC of 0.98 on the training set and 0.97 on both validation and testing sets (see Figure 8). The minimal differences among these values indicate strong generalization and low risk of overfitting. The model demonstrates a steep increase in the true positive rate with a very low false positive rate, confirming its robustness in handling multi-class classification tasks under varying input conditions.

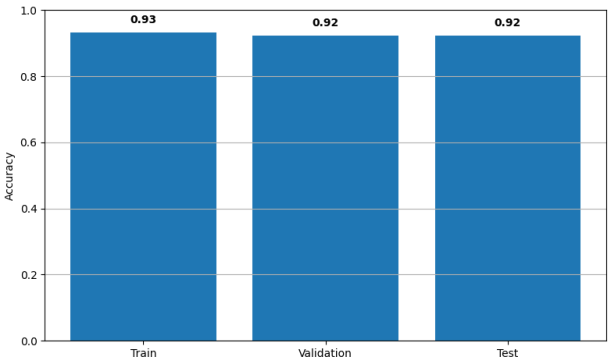

Figure 9. Accuracy comparison across training, validation, and testing sets

Accuracy values remain closely aligned across datasets—93% on the training set and 92% on both validation and testing (Figure 9), indicating stable performance. This slight variation is within acceptable limits and reflects a well-regularized model capable of maintaining stability and consistency in predictions. The balanced performance indicates that the model effectively learned underlying patterns in the training data while preserving predictive power on unseen examples, thereby ensuring reliable deployment in real-world cloud environments.

The Decision Tree structure provides a detailed perspective on how the model analyzes and processes information to make optimal classification decisions for process allocation. Each split in the tree is based on relevant features, allowing for a progressive and efficient separation of data, gradually reducing impurities and maximizing prediction accuracy. At the core of this process lies the root node, which uses the score as the primary splitting criterion. Selecting this attribute as the starting point in the tree's structure reflects its importance in optimal process classification, as the score is calculated based on throughput and delay, two critical factors in determining the ideal resource for execution. As the tree branches out, intermediate nodes apply additional splits, using delay as a secondary major separation criterion. This stage further optimizes the decision-making process, ensuring that each process is directed to the appropriate category with minimal ambiguity. Once the tree reaches its terminal nodes, the classification process is complete, and each instance is assigned to a well-defined category. The terminal nodes represent the model's final decisions, and the low or absent impurities at this level indicate a clear and precise separation of data. This confirms that the model has been effectively trained and has successfully learned the essential classification patterns. This hierarchical organization of the decision-making process closely resembles how Kubernetes manages resource allocation in cloud environments. Just as the Kubernetes scheduler distributes workloads based on factors such as CPU, memory, and delays, the decision tree optimizes process distribution based on performance criteria. In both cases, the goal was to maximize efficiency, reduce execution times, and ensure the optimal utilization of available resources. By implementing this decision tree-based model, the system achieved a structured and precise resource allocation method, which not only enhanced process management but also reduced infrastructure congestion. Essentially, this mechanism ensures that each resource was used intelligently, and task processing was carried out in an optimized and predictable manner. This decision tree-based approach not only facilitated the balanced distribution of resources, but also enabled dynamic adaptation to system changes, ensuring flexible and efficient task management. By integrating such intelligent techniques, the system became capable of managing resource allocation more efficiently, significantly reducing the risk of overloading certain nodes. This enabled a balanced distribution of tasks, ensuring that each process was assigned to the most suitable node, based on available capacity and the specific requirements of the process. Optimizing the execution flow not only enhanced system responsiveness but also contributed to a more efficient and predictable utilization of available resources. In this way, the cloud infrastructure becomes more organized and adaptable, allowing it to dynamically respond to load variations while maintaining stable performance, even under high-demand conditions. The scalability of the system was improved, as resources were managed automatically and efficiently. Also, reliability was enhanced by reducing bottlenecks.

Figure 10 illustrates the decision path used by the model to classify each process into the appropriate worker class. The root node begins with a split based on `DelayMax`, confirming the importance of delay-related parameters in initial decision-making. Further branches refine the classification using attributes such as Score, `ThroughputMin`, and `Priority`, ensuring a well-structured allocation logic. This layout reflects how the model prioritizes responsiveness and performance efficiency when assigning tasks across workers. The purpose of the simulation for model verification was to demonstrate how a machine learning model can allocate processes to workers, drawing an analogy to Kubernetes scheduling.

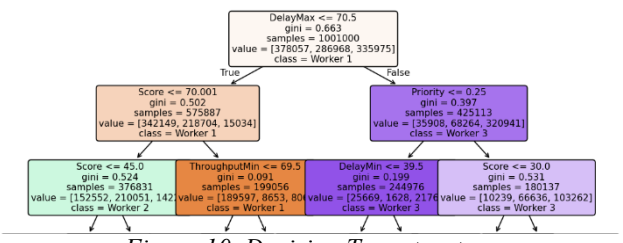

Figure 10. Decision Tree structure

Features such as zone, instructions, size, priority, throughput, delay, and score were considered by a decision tree-based model to predict the optimal process allocation.

The simulation was designed to emulate Kubernetes-like process scheduling, where tasks are dynamically

assigned to computing nodes based on predefined resource constraints and system conditions. The core functionality of this simulation is centered around two primary classes: Cloudlet and Master. In this architecture, the Cloudlet class serves as the foundation for simulating processes. Each Cloudlet instance encapsulates all the relevant characteristics of a task: zone, number of instructions, size, priority, throughput, delay, and later, a computed score. This class is responsible for receiving input values (e.g., from a command-line interface) and transforming them into a standardized dictionary format that can be used by the machine learning model.

The abstraction provided by this class mirrors how real-world orchestration systems manage and structure incoming tasks. It ensures modularity and consistency, allowing the simulation to treat each task as a fully defined object that can be evaluated, scored, and allocated efficiently. Complementing the Cloudlet, the Master class acts as the central coordinator. It loads the pre-trained decision tree model, calculates the score based on the Cloudlet's throughput and delay parameters using the same logic as in the training phase, and then predicts which worker should execute the task.

Additionally, the Master is responsible for simulating task execution by creating directories under the respective worker's folder - effectively emulating deployment paths, like Kubernetes maps pods to nodes. Each directory represents a unit of execution (e.g., one instruction), further reinforcing the one-to-one mapping between abstract process logic and practical resource scheduling. Figure 11 illustrates the instantiation of a Cloudlet process, encapsulating its features into a structured object ready for allocation

```
Enter area: 2
Enter Instructions Min: 78
Enter Instructions Max: 100
Enter the priority: 1
Enter Throughput Min: 70
Enter Throughput Max: 90
Enter Delay Min: 12
Enter Delay Max: 15
```

Figure 11. Creating a process for the model

Complementing the Cloudlet, the Master Node receives the incoming process and analyzes its characteristics to determine which worker should execute it (see Figure 12). Factors such as priority, number of instructions, throughput, and delay are considered to make the optimal choice. Once selected, the process is dispatched to the appropriate worker node—its execution is animated in Figure 13—thus avoiding overloading and ensuring a balanced task distribution.

Once the appropriate worker was identified, the Master forwarded the process for execution. This selection was not performed randomly, as it was due to careful analysis. By following this structured approach, each process was handled according to its requirements, ensuring that the system operated in an organized and efficient manner.

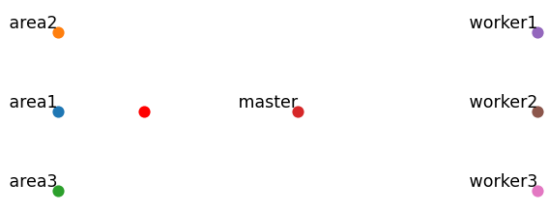

Figure 12. Animation of process allocation to the model

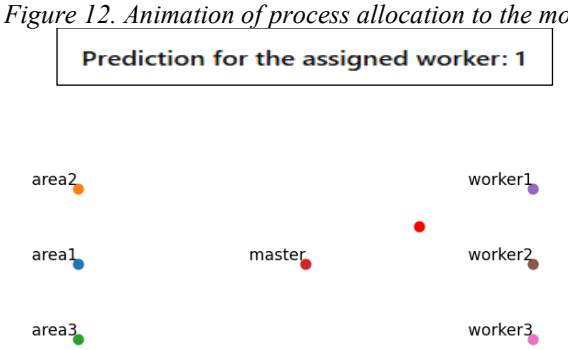

Figure 13. Animation of process allocation to the worker

The simulation highlighted the potential of integrating machine learning into cloud orchestration systems, demonstrating how data-driven decisions can significantly enhance resource allocation efficiency. By employing a model capable of analyzing process characteristics and determining optimal allocation, the system became more adaptable and efficient.

As observed in the simulation, the first incoming process was allocated to Worker 1, which was responsible for executing the assigned instruction. The model's ability to make accurate predictions ensured that each process was directed at the most suitable resource, preventing task distribution imbalances. The simulation reflected an approach like Kubernetes scheduling, where decision-making is guided by historical data analysis and learned patterns. Just as Kubernetes employs advanced schedulers to allocate workloads based on CPU, memory, and network delays, the ML-based model followed a similar principle, optimizing process execution flow based on precise and well-defined criteria.

This simulation method represented a powerful strategy for efficient task distribution, providing clear insights into how machine learning can transform traditional cloud orchestration processes. Integrating such techniques could lead to better resource utilization, reduced operational costs. Also, it increased scalability in distributed environments, ensuring a predictive and optimized approach to cloud infrastructure management.

The decision tree classifier achieves 93 % test accuracy, notably outperforming the 90 % reported for the Random Forest model in [10] and the 89 % achieved by the SVM-based approach in [8] on analogous synthetic cloud workloads. In addition to higher accuracy, the decision tree

requires approximately 30% less training time than the Random Forest implementation in [10], reducing overall computational cost. Furthermore, the tree's shallow structure (max_depth=5) yields a compact model footprint—approximately 40% fewer nodes than the ensemble's combined trees—while preserving interpretability. These advantages demonstrate that the proposed method not only improves predictive performance but also enhances efficiency and transparency, making it well suited for real-time cloud scheduling scenarios.

IV. CONCLUSIONS AND FUTURE WORK

This paper introduced a machine learning-based approach for intelligent cloud process allocation, utilizing decision trees to effectively classify and assign workloads to optimal worker nodes. By integrating a wide array of process-specific parameters, including zone, instructions, size, priority, throughput, and delay. The proposed model achieved high allocation accuracy while maintaining transparency and interpretability through its tree structure. The results confirmed the model's robustness, achieving over 92% test accuracy and showing excellent balance across all three worker classes. Even under perturbed data scenarios, the system maintained consistent performance, highlighting its resilience and generalization capabilities. A critical factor in this success was the design of the data pipeline: feature engineering, score normalization, and controlled scenario generation ensured diverse and meaningful learning examples.

Through a simulation framework inspired by Kubernetes orchestration, the system showcased how AI models can emulate and enhance real-world scheduling mechanisms. The Cloudlet and Master classes mirrored Kubernetes pods and schedulers, enabling process creation, scoring, and allocation in a structured and reproducible manner. The graphical simulation further emphasized the traceability and clarity of the model's predictions. The decision tree-based model proved to be an efficient and interpretable solution for real-time task allocation, making it a viable candidate for integration into production-grade cloud orchestration platforms.

Future work will extend the decision tree allocator to multi-tenant cloud environments, evaluating its ability to maintain fairness and efficiency when workloads contend for shared resources. Incorporating reinforcement-learning agents into the scheduling loop may enable continuous adaptation to real-time throughput and delay fluctuations, while hybrid models could combine the interpretability of decision trees with the predictive power of deep learning. Validation on public cloud traces—such as high-performance computing and web-service logs—will assess practical applicability under production conditions. Finally, end-to-end benchmarking against established schedulers (Kubernetes' kube-scheduler) will quantify improvements in throughput, latency, and cost savings, guiding seamless integration into operational cloud platforms.

REFERENCES

[1] M. Sajid and Z. Raza, "Cloud Computing: Issues & Challenges," International Conference on Cloud, Big Data and Trust 2013, RGPV, Bhopal, India, Nov. 13-15, 2013

[2] L.Wang, R. Ranjan, J. Chen, and B. Benatallah, Eds., "CLOUD COMPUTING: Methodology, Systems, and Applications". CRC Press, 2011.

[3] N. Paladi, A. Michalas, and H. Dang, "Towards Secure Cloud Orchestration for Multi-Cloud Deployments," presented at the 5th Workshop on CrossCloud Infrastructures & Platforms, Porto, Portugal, Apr. 23-26, 2018.

[4] K. Bousselmi, Z. Brahmi, and M.M. Gammoudi, "Cloud Services Orchestration: A Comparative Study of Existing Approaches," 2014 28th International Conference on Advanced Information Networking and Applications Workshops. IEEE, May 2014.

[5] Kubernetes. Available online: https://kubernetes.io/docs/home/.

[6] Z. Xu, Y. Gong, Y. Zhou, Q. Bao, and W. Qian, "Enhancing Kubernetes Automated Scheduling with Deep Learning and Reinforcement Techniques for Large-Scale Cloud Computing Optimization." arXiv, 2024.

[7] S. Sharma, "An Investigation into the Optimization of Resource Allocation in Cloud Computing Environments Utilizing Artificial Intelligence Techniques," Journal of Humanities and Applied Science Research, vol. 5, no. 1, pp. 131-140, 2022.

[8] Y. Wu, "Cloud-Edge Orchestration for the Internet of Things: Architecture and AI-Powered Data Processing," IEEE Internet of Things Journal, vol. 8, no. 16. Institute of Electrical and Electronics Engineers (IEEE), pp. 12792–12805, Aug. 15, 2021.

[9] S. Mehta, A. Singh, and K.K. Singh, "Role of Machine Learning in Resource Allocation of Fog Computing," 2021 11th International Conference on Cloud Computing, Data Science & Engineering (Confluence). IEEE, Jan. 28, 2021.

[10] S. Manam, K. Moessner, and P. Asuquo, "A Machine Learning Approach to Resource Management in Cloud Computing Environments," 2023 IEEE AFRICON. IEEE, Sep. 20, 2023.

[11] R. Botez, A.-G. Pasca, A.-T. Sferle, I.-A. Ivanciu, and V. Dobrota, "Efficient network slicing with SDN and heuristic algorithm for low latency services in 5G/B5G networks," *Sensors*, vol. 23, no. 13, p. 6053, Jun. 2023.