Adaptive and Intelligent Task Offloading in Heterogeneous Computing Environments: A Statistical and Machine Learning Powered Framework

Sharad Shahi

***Abstract*—The increasing computational requirements due to the growing number of IoT devices require enhanced task offloading strategies in hierarchical IoT-Edge-Cloud computing frameworks. Conventional offloading approaches, typically based on static guidelines and set thresholds, struggle to manage the dynamic and unpredictable characteristics of contemporary workloads, resulting in poor performance, resource conflict, and a decline in Quality of Services (QoS). This paper presents the Statistical and Machine Learning Powered Offloading Algorithm (SML-OA), an innovative framework aimed at dynamically and efficiently distributing computational tasks by developing a self-aware, adaptive, and resilient decision-making system. The SML-OA is built upon a composite offloading score, a statistical tiered framework utilising the interquartile range of the real-time task distribution; a sophisticated anomaly detection method that employs higher order statistical moments to identify volatility and distributional drift; and a smart fallback engine built on gradient boosting to capture complex system behaviours. Through the combination of strong statistical analysis and predictive machine learning, the SML-OA offers a comprehensive solution that addresses both the inherent characteristics of specific tasks and the statistical nature, as well as the current status of the entire distributed system, enabling robust and context-sensitive offloading. Experimental evaluation using synthetic IoT workloads demonstrates the efficacy of the proposed framework to maintain QoS stability and adapt effectively to varying workload, illustrating its potential for deployment in real-world, latency-sensitive, distributed computing scenarios.**

***Index Terms*—Internet of Things (IoT), Edge Computing, Cloud Computing, Osmotic Computing, Task Offloading, Machine Learning, Statistics, Quality of Services (QoS).**

I. Introduction

The modern computing environment is marked by an extraordinary increase in data production, driven by the expected rise of over 50 billion connected Internet of Things (IoT) devices in the near future [1]–[5]. This influx of data, originating from a wide variety of sensors, actuators, and intelligent wearables, generates computational requirements that greatly surpass the abilities of standalone, resource-limited devices [6]. To address this fundamental challenge, a tiered computing framework that effortlessly integrates IoT, Edge, and Cloud resources into a cohesive continuum has become the standard architecture for future applications.

Sharad Shahi is with the School of Computer Engineering, Kalinga Institute of Industrial Technology (KIIT) Deemed to be University, Bhubaneswar, 751024, Odisha, India.

At the top of this hierarchy, the Cloud layer provides an extensive, virtually infinite reservoir of centralised computing, storage, and networking resources [7]–[9]. It serves as the perfect location for handling extensive, computation-heavy, and frequently latency-tolerant activities like intricate analytics and archiving data for the long term [10]. Nonetheless, the physical distance of cloud data centers from the data source creates considerable network latency [11], [12]. This built-in lag makes the pure cloud model inappropriate for an expanding range of real-time, delay-sensitive applications, such as augmented reality, online gaming, and real-time health monitoring [13], [14].

Edge computing has become an essential architectural element to close this latency gap. The Edge tier offers low-latency processing capabilities by implementing a distributed layer of more powerful computing nodes, like micro-datacenters (μDCs), gateways, or base stations, situated near end users and data sources [8], [15]. This closeness is its key benefit, making it the perfect choice for time-sensitive tasks and decreasing the amount of data that needs to be sent to the central cloud. This architectural development has established a federated IoT-Edge-Cloud ecosystem where computational operations can, theoretically, be handled at the most suitable site to satisfy strict Quality of Services (QoS) demands [14].

The osmotic computing paradigm elegantly conceptualises the dynamic administration of services and the smooth transfer of data within this varied continuum [6]. The osmotic computing model offers the overarching rationale for the dynamic migration of services across IoT, Edge, and Cloud tiers. Section III provides an in-depth account of the proposed osmotic model, the five-tier ecosystem, and the functionality and roles of the Osmotic Manager (OM). The main challenge in the osmotic computing framework is creating smart task offloading methods that can serve as this adaptive "membrane". Although the offloading decision is essential, current approaches frequently lack the necessary dynamism and intelligence, relying on rigid, predetermined guidelines and set limits that do not adjust to the extremely dynamic and uncertain characteristics of actual workloads and the ever-changing conditions of system resources. This inflexibility is the main factor contributing to performance decline in distributed systems, causing considerable operational challenges, including congested edge nodes, increased queuing delays, inefficient resource use, and ultimately, an inability to satisfy

the strict QoS requirements of latency-sensitive applications. In areas like real-time health monitoring, augmented reality networks, and smart city applications, where prompt responses are essential, the effects of inadequate offloading can lead to a diminished user experience or even catastrophic system failure [5]–[7], [10]. The failure of static systems to adapt to bursty traffic, changes in workload complexity, or resource contention significantly restricts their efficiency and scalability.

A. Contribution

This paper contends that a crucial transition is needed from the existing reactive, rule-oriented offloading approaches to a proactive, data-centric model. An efficient offloading system should take into account not just the characteristics of a single task but also grasp the statistical features of the overall workload and the current status of the distributed infrastructure. To address this requirement, we present the Statistical and Machine Learning Powered Offloading Algorithm (SML-OA), an innovative and all-encompassing framework intended to function as the intelligent nucleus of an offloading management system which employs sophisticated statistical techniques to attain a level of system self-awareness and adaptability. Furthermore, it includes a machine learning fallback model built on gradient boosting, which becomes active under system stress to provide informed corrective actions when statistical inference alone becomes unreliable.

This study enhances the understanding by delivering a formal and critical assessment of the latest offloading techniques by methodically examining their progression and recognising a significant research gap resulting from the absence of statistical and predictive insights in existing methods. Additionally, the paper outlines the comprehensive design of the SML-OA framework, elaborating on its four novel and complementary elements, i.e., the adaptive Composite Offloading Score (COS), the dynamic Interquartile Range (IQR) based statistical tiering, the sophisticated anomaly detection and management system, and the machine learning powered fallback engine. The study also suggests a thorough experimental approach to confirm the effectiveness of SML-OA compared to current algorithms in diverse, realistic, and demanding workload situations. To prove the efficacy of the proposed algorithm, several synthetic test scenarios are used for testing and evaluation, and, as far as we are aware, the proposed algorithm is not directly related to any recent research.

B. Paper Organization

The subsequent sections of this paper are structured in the manner outlined as follows. Section II provides a thorough evaluation of current offloading techniques, Section III presents the suggested osmotic model and elaborates on the five-tier osmotic ecosystem along with a comprehensive explanation of the OM. Section IV describes the framework for statistical tiering and managing anomalies, and outlines the fallback engine driven by machine learning. Section V shows the mathematical working and performance evaluation of the proposed algorithms, and ultimately, Section VI wraps up the paper.

II. A Critical Review of Offloading Strategies

Task offloading research has progressed from simple threshold-based rules to more sophisticated multi-factor decision systems. Despite this evolution, a key limitation remains. Most existing approaches still make decisions for each task independently, without considering global system state, cumulative workload patterns, or statistical trends in task arrivals. This limitation motivates the development of the proposed SML-OA framework, which aims to introduce system-aware and learning based intelligence rather than relying on static and reactive logic.

A. Mobile-Edge Computing (MEC) Offloading Approaches

In MEC environments, several studies have investigated efficient ways to distribute computation between edge and cloud servers. Survey research has categorised these approaches based on optimisation goals, such as minimising delay, energy consumption, service cost, and load imbalance [26]–[28]. Other works classify strategies according to system architecture, such as device-to-edge or edge-to-cloud execution paths [29], and by decision methodology, including heuristics, game-theoretic models, and machine learning based approaches [30], [31]. Additional taxonomies consider resource scheduling, user mobility, and task partitioning granularity [32]. These surveys provide a structured view of MEC research; however, they also show that many solutions still rely on static optimisation assumptions and do not continuously adapt to changing workloads.

Several practical schemes have also been proposed to enhance task offloading under specific conditions. Barbera et al. [20] recommend processing latency-critical tasks at edge nodes, while forwarding delay-tolerant tasks to the cloud. Liu et al. [30] suggest that applications with large data transfer requirements but low computation intensity suit the edge, and compute-intensive tasks are better handled in the cloud. Tong et al. [34] present a hierarchical edge structure combined with heuristic decision making to reduce delay, while Sun et al. [23] propose mobility-aware offloading for dense MEC networks. Chen et al. [24] study multi-user computation offloading, and Xiao and Krunz [25] demonstrate that cooperation among edge servers can reduce delay and energy usage. While these MEC strategies are effective in certain situations, conventional greedy and game-theoretic approaches continue to face challenges when system conditions fluctuate often [16]–[19]. Their dependence on set rules and rigid assumptions restricts their adaptability, and consequently, they encounter challenges in adjusting to diverse environments or varying workloads, emphasising the necessity for adaptive and learning based offloading systems that can respond to real-time system activities.

B. Osmotic Computing and Distributed Cloud-Edge Environments

In osmotic environments, early offloading techniques depend on predefined criteria to decide the execution tier of tasks. B. Neha et al. [33] used two fixed thresholds, τ_1 and

τ_2, to categorise tasks and assign them strictly to IoT, Edge, and Cloud tiers. The ELBO algorithm enhances flexibility by comparing local execution time and device energy against defined thresholds; tasks that surpass these thresholds are transferred to the Edge, which conducts analogous validations [34]. While more flexible than static thresholds, ELBO continues to assess tasks separately and overlooks current system situations. Similarly, the ELTO algorithm allocates assignments according to priority, directing urgent tasks to the Edge and less urgent ones to the Cloud [34]. While the concept acknowledges QoS requirements, the binary categorisation simplifies complex real-world tasks while neglecting IoT tier as a potential processing choice, reducing adaptability. Fuzzy logic systems manage various inputs like task size, virtual machine (VM) usage, and delay tolerance, making these techniques more effective in representing uncertainty but dependent on fuzzy rules and membership functions defined by experts. After deployment, these rules do not adjust to the new workload patterns, and hence, decision boundaries stay constant despite changes in task characteristics.

C. Research Gap

Although extensive studies in both MEC and osmotic environments have provided strong heuristics and optimisation strategies, existing algorithms remain primarily reactive, in which decision-making is often based on established static guidelines, particular predefined system assumptions, or parameters that are selected manually. Proactive, workload-sensitive methods that can learn from real-time system performance and dynamically adjust decision logic in real-time are notably absent. Existing techniques fail to create a comprehensive statistical grasp of the entire task stream, resulting in reduced effectiveness when managing unpredictable workloads or changing demand patterns. For situations where numerous tasks exhibit similar characteristics and belong to the same category, resource imbalance is created where one tier might become overwhelmed while another remains underutilised. Since the tasks are managed individually, the system does not comprehend queue accumulation, overall resource usage, or elevating system stress. Furthermore, essential tasks vary in scale and intricacy, while routine tasks can require significant computation, yet most of the frameworks do not provide a robust fallback mechanism for "system-in-stress" conditions. This study addresses these limitations by suggesting a framework aimed at establishing real-time workload awareness and utilising predictive learning to inform offloading choices and introduce an intelligent fallback mechanism when the system is in a stressed operational state. The goal is to shift from fixed, task-specific heuristics to a self-adjusting dynamic system that enables ongoing learning and overall optimisation. A concise overview of the offloading algorithms is presented in Table I, highlighting their core mechanism and limitations, which motivated the development of our proposed SML-OA.

Method	Core Mechanism	Limitation
Task Mapping [33]	Fixed thresholds assign tasks to IoT/Edge/Cloud tiers.	Static rules; no queue/load awareness; fails in dynamic traffic.
ELBO [34]	Sequential energy and latency checks for offloading.	Local task-based decisions; ignore global resource state.
ELTO [34]	Critical tasks to Edge; normal tasks to Cloud.	Coarse binary classification; no IoT execution; limited flexibility.
Fuzzy Logic [5], [7]	Multi-parameter fuzzy inference and rule-based mapping.	Expert-defined rules; no learning or real-time adaptation.
MEC Heuristics / Greedy [16]–[19]	Heuristic and greedy delay/energy optimisation.	Scenario specific; limited generalisation across conditions.
Hierarchical / Mobility Aware MEC [22]–[25]	Placement based on mobility and edge hierarchy.	Architecture-bound; no statistical workload learning.
Data / Compute Aware MEC [20], [21]	Data-heavy tasks to Edge; compute-heavy to Cloud.	Simplistic rules; ignore dynamic system load.

TABLE I: Summary and Comparison of Foundational Offloading Strategies in MEC and Osmotic Computing

III. Proposed Osmotic Model

This section details the osmotic ecosystem we have used for the SML-OA algorithm. The osmotic model draws inspiration from the osmosis process and establishes a specific architecture consisting of five logical tiers, along with specialised control plane entities known as osmotic managers. These OMs coordinate the migration and positioning of tasks throughout the continuum using both local and global decision-making processes. In this framework, fundamental system components are aligned with the osmosis analogy. "Solute" corresponds to stable tasks or resource attributes that remain fixed during execution, "solvent" refers to workload units such as tasks or microservices whose execution location can be changed, while "solution" represents the aggregate state formed by current tasks, system resources, and placement policies. The "semi-permeable membrane" is realised through a software-defined control mechanism operated by OM instances, and these membranes determine task mobility based on predefined polices, system conditions, and QoS constraints. In our proposed ecosystem, two types of osmotic membranes have been used; one membrane separates the IoT and Edge tiers, while the other lies between Edge and Cloud. Each membrane is instantiated as an OM that makes autonomous local placement decisions while participating in inter-agent coordination. The five-tier architecture layout includes two osmotic control layers integrated as logical intermediaries between physical tiers, following the conventions from prior osmotic computing studies [33], [34].

A. Osmotic Ecosystem

The ecosystem consists of five logical tiers, as shown in Figure 1, which are organised as follows.

1) Tier 1 (IoT): This is the foundational layer, which comprises resource-constrained industrial devices and sensors responsible for data generation and initial computation.
2) Tier 2 (Osmotic Layer between IoT and Edge): A control tier colocated with edge gateways that governs migration between IoT devices and edge nodes.

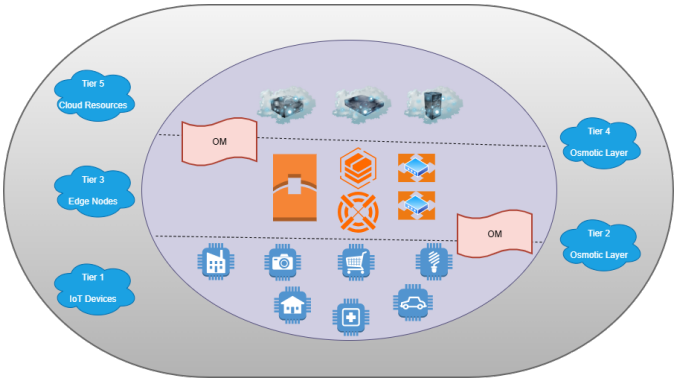

Fig. 1: Osmotic computing ecosystem showing IoT devices, osmotic layers, edge nodes, and cloud resources.

3) Tier 3 (Edge): This layer acts as local gateways and micro-datacenters that provide low-latency computation and data caching close to data sources.
4) Tier 4 (Osmotic Layer between Edge and Cloud): A regional control tier that facilitates migration between edge infrastructures and centralised cloud systems.
5) Tier 5 (Cloud): This represents the uppermost layer of centralized hyperscale data centers, which provide high-capacity computing and enable long-term data storage and analysis.

Osmotic membranes function as control plane tiers placed between execution tiers. The local decision area manages offloading requests from the nearby tier, while the coordination domain interacts with adjacent membranes, overarching policy organisations to discuss positioning, load allocation, and resource access. This framework facilitates the execution of the two OM deployment models, wherein each member agent is situated between execution levels, carries out localised task allocation and distribution.

B. Osmotic Manager

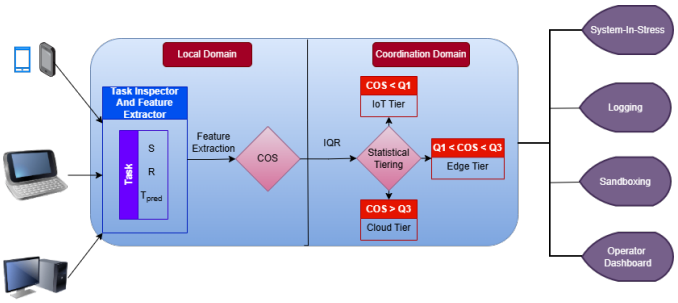

Fig. 2: Local and coordination domain architecture of OM.

OM instances execute the control logic necessary for real-time offloading, tier forecasting, and coordination, where every OM functions autonomously while upholding connections for cooperative decision making and data analysis. Each OM contains two domains, namely, a local domain, which handles internal functioning and resource management, and a coordination domain, which communicates with OM at other tiers and monitors the broader system state. An abstract representation of these components and their interactions is shown in Figure 2. An OM handles incoming workload descriptors, examines metadata, and gathers features needed for prediction and decision phases, while also observing local resource status, such as CPU, memory, queue depth, energy levels, bandwidth, and latency. Leveraging these inputs, OM computes the COS and examines statistical properties, specifically quartiles and IQR, as well as higher-order statistics such as skewness and kurtosis to determine adaptive migration thresholds. The forecasting engine in the local domain calculates the execution time per tier T_{pred} using a linear regression or adjusted baseline models, then the feature extraction engine generates feature vectors and utilises trained models designated as the fall-back mechanism for "system-in-stress" conditions to generate placement probabilities and confidence scores. The OM also performs anomaly detection by identifying irregularities in COS distributions, leveraging both its local and coordination domains during cross-tier validation, separating questionable workloads via sandboxing when required, and logging events for subsequent analysis. Final placement tasks are carried out after overseeing acknowledgements, rate management, retries, system status, and task division when applicable, with the process supported by an integrated operator dashboard that offers insight into system performance, model reliability, and instantaneous resource consumption, as well as facilitating mechanisms for manual interventions like COS reweighting, throttling, or escalation. Every task is assessed for privacy, regulatory and policy compliance, and those that do not meet the requirements are either kept in authorised areas or transferred to a secure sandbox environment.

IV. Proposed Adaptive Osmotic Offloading Framework

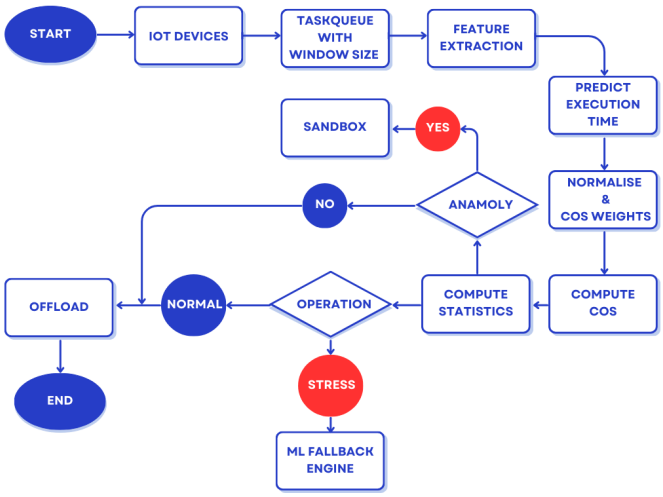

Fig. 3: End-to-end workflow of the SML-OA Algorithm.

In this section, the proposed solution for adaptive and dynamic task offloading in an osmotic environment is presented, starting with the foundational prerequisite on which the entire framework hinges, which is the formulation of the COS. Building on this scoring mechanism, the statistical

tiering process is described along with an integrated anomaly detection module, which together enhances the reliability of decision-making under normal operating conditions. However, since the real-world environments rarely behave in a predictable fashion and often experience uncertain and irregular surges of incoming data that can overwhelm a specific tier, a machine learning powered fallback mechanism is used to deal with such "system-in-stress" conditions. The end-to-end operational pipeline is shown in Figure 3, illustrating how tasks flow through COS computation, anomaly detection, and tier decisions.

A. Formulation of a Unified Composite Offloading Score (COS)

To go beyond the constraints of single-parameter or inflexible rule-based systems, a comprehensive metric is needed to capture the complex "cost" or "demand" of a computational task. This discussion explains the formulation of COS, a consolidated, single-value measure that compiles various essential task characteristics like task size, required resources, and execution time, and it also serves as the core element of analysis for the subsequent statistical and machine learning models, offering a consistent gauge of task difficulty. The COS is based on three main task characteristics, as mentioned earlier, guided by parameters widely utilised in the current literature. Before using them in the formula, every attribute needs to be clearly defined and normalised to a uniform scale to prevent any one attribute with a significant numerical range from unduly affecting the overall score. Table II provides a summary of all primary characteristics along with their associated symbols, units, and functions in the COS formulation.

a) Task Size (S): This is the clearest indicator of a task's requirement for network and storage resources, which is a key factor in initial offloading models, and it is officially described as the volume of the task's data payload measured in bytes. A larger task size means a higher demand on network bandwidth for transmission and increased storage requirements at the destination node.

b) Resource Requirements (R): This attribute represents the computational demand of the task. Unlike simpler models that use task size as a proxy for overall resource needs, this is a composite value reflecting both CPU and memory requirements. The resource requirement is formulated as a weighted sum, as shown in Equation (1).

$$R = \alpha \cdot \mathrm{CPU}_{\mathrm{cycles}} + \beta \cdot \mathrm{Memory}_{\mathrm{req}} \tag{1}$$

Where α and β are system-level parameters that can be tuned to reflect the relative scarcity or importance of CPU versus memory resources in a specific deployment environment. Determining the optimal parametric values is a complex problem that exceeds the scope of this study.

c) Predicted Execution Time (T_{pred}): Instead of treating execution time as a fixed task attribute, a tier dependent predicted execution time T_{pred} is computed for every candidate resource tier IoT, Edge, or Cloud using a lightweight pre-trained regression model, such as linear regression, which is trained on historical task traces to estimate time-to-completion on a standardized baseline processor. To produce tier-specific estimates, the OM uses a separate tier-calibrated regression model for each tier, which makes T_{pred} dynamic and adaptive, ensuring that both task characteristics and current tier performance are reflected and thereby enabling realistic, context-aware offloading decisions.
Mathematically:

$$T_{\mathrm{pred}} = f_{\theta,\mathrm{tier}}(X_T) \tag{2}$$

Where:

- X_T are task features such as input size, CPU speed, or similar characteristics.
- $f_{\theta,\mathrm{tier}}$ is the pre-trained tier-specific regression model.

The OM periodically fine-tunes $f_{\theta,\mathrm{tier}}$ using observed task run times to incorporate runtime variability and hardware heterogeneity.

d) Normalisation: Before these heterogeneous metrics can be aggregated, they are first normalised using Min–Max scaling. For an attribute X, the normalised value X_{norm} is computed as mentioned in Equation (3).

$$X_{\mathrm{norm}} = \frac{X - X_{\mathrm{min}}}{X_{\mathrm{max}} - X_{\mathrm{min}}} \tag{3}$$

An essential aspect of the normalization process in our system is that the values for X_{min} and X_{max} are not fixed, predetermined constants; rather, they are recalibrated dynamically within a sliding time frame that includes the latest tasks, which typically corresponds to all tasks currently present in the offloading queue. This flexible method guarantees that the normalisation context stays adaptable and reacts to changes in the workload characteristics. Utilising this method, the system can autonomously adapt to changes in task scale or resource needs, preserving an ideal operational equilibrium as task demands shift over time.

1) A Weighted Aggregation Model for the COS: Once the primary attributes are collected and normalised, they are combined into the COS through a linear weighted sum, which provides a clear and efficient approach to generate a unified score that captures the total demand of a task. The COS formula is defined in Equation (4).

$$\mathrm{COS} = w_S \cdot S_{\mathrm{norm}} + w_R \cdot R_{\mathrm{norm}} + w_T \cdot T_{\mathrm{pred\_norm}} \tag{4}$$

Where w_S, w_R, and w_T are the weights assigned to the normalised task size, resource requirements, and predicted execution time, respectively. These weights must satisfy the constraint mentioned in Equation (5).

$$w_S + w_R + w_T = 1 \tag{5}$$

A larger COS value indicates that the task imposes a higher computational demand and therefore serves as the backbone of our statistical framework; however, its true intelligence lies not in the formula itself but in the adaptive weighting approach. Indeed, while a basic, fixed weighting system would merely replicate the rigidity of earlier models, an adaptive mechanism is essential because it establishes a critical feedback loop between the system's operating state and its evaluation of task cost.

a) Policy Based Dynamic Weighting: A sophisticated method that includes weights adjusted in real time to reflect system conditions, whereby the system constantly monitors operational metrics and adjusts the significance of COS elements as needed. For instance, when metrics show heightened network congestion or increased latency between the Edge and Cloud layers, the framework can temporarily elevate the value of w_S so that tasks with significant data transfer needs incur a greater cost. Similarly, if the Edge tier shows elevated CPU usage, the system might raise w_R to deter the running of resource-demanding tasks at that tier. With this policy-driven approach, COS becomes attuned not only to the unique traits of each task but also to the wider context in which it operates, so that the weights w_S, w_R, and w_T are seen as variable entities instead of static constants.

b) Reinforcement Learning Based Weighting: A more sophisticated method includes acquiring the weighting strategy via Reinforcement Learning (RL). In this approach, an RL agent gradually discovers weight settings that optimise a long-term performance goal like increased throughput or better QoS throughout the system, thereby changing the COS from a manually adjusted metric into a learned framework that adapts alongside the workload and operating conditions. A thorough examination of RL-based weight optimisation, encompassing agent and reward design, and training methods is will be addressed in a dedicated future study.

TABLE II: Symbols, Units, and Descriptions for the Components of COS

Symbol	Unit	Function in Formula
S (Task Size)	Bytes	Raw input representing network/storage load.
R (Resource Requirements)	Composite units	Raw input representing computational load.
T_{pred} (Predicted Execution Time)	Seconds	Raw input representing temporal load.
S_{norm} (Normalised Task Size)	Dimensionless	Component of COS representing size.
R_{norm} (Normalised Resource Requirements)	Dimensionless	Component of COS representing resource requirements.
$T_{\text{pred\_norm}}$ (Normalised Predicted Execution Time)	Dimensionless	Component of COS representing time.
w_S (Weight for Task Size)	Dimensionless	Determines the influence of task size on the final score.
w_R (Weight for Resource Requirements)	Dimensionless	Determines the influence of resource requirements on the final score.
w_T (Weight for Predicted Execution Time)	Dimensionless	Determines the influence of execution time on the final score.
COS (Composite Offloading Score)	Dimensionless	The final, unified metric used for statistical analysis and tier allocation.

B. Statistical Tiering

Effective task offloading in a distributed environment demands not only precise scoring systems but also strong tiering strategies capable of adjusting to different and evolving workloads. Statistical approaches offer a systematic means to categorise tasks into IoT, Edge, and Cloud tiers by examining their combined traits and distribution characteristics, and by utilising statistical metrics like quartiles and variability, the system can set flexible thresholds that differentiate light, moderate, and heavy tasks without depending on fixed or manually adjusted parameters. Moreover, robustness strategies based on distributional analysis assist in recognising outliers, extreme instances, and unbalanced workloads, guaranteeing that resource distribution stays even during unusual or sudden spikes in demand and thereby helping the system maintain essential QoS requirements. The integration of statistical tiering and robustness improves fairness and efficiency, establishing

Algorithm 1 Statistical Tiering Algorithm with Anomaly Handling

```
Initialize TaskQueue with sliding window size W
Initialize SystemMonitor for CPU, memory, and latency
Load pre-trained tier-specific ML_Model
Initialize COS weights (w_S, w_R, w_T)  ▷ subject to Eq. (5)
function OnNewTaskArrival(task)
    T_pred ← PredictExecutionTime(task)  ▷ Eq. (2)
    Normalise (S, R, T_pred) within TaskQueue  ▷ Eq. (3)
    COS ← ComputeCOS(task, COS weights)  ▷ Eq. (4)
    TaskQueue.enqueue(task)
    Stats ← UpdateQueueStatistics(TaskQueue)  ▷ Q1, Q3, IQR, Skewness, Kurtosis
end function
function ProcessTaskQueue
    while TaskQueue is not empty do
        task ← TaskQueue.peek()
        is_outlier ← CheckIfOutlier(task.COS, Stats)
        is_volatile ← (Stats.Kurtosis > K_THRESHOLD)
        if is_outlier and is_volatile then
            DispatchToCloudSandbox(TaskQueue.dequeue())
            continue
        end if
        if IsDistributionAnomalous(Stats.Skewness, Stats.Kurtosis) then
            TriggerAlert()
            COS weights ← AdaptWeights(COS weights, Stats)
        end if
        if task.COS < Stats.Q1 then
            StatDecision ← IoT
        else if task.COS > Stats.Q3 then
            StatDecision ← Cloud
        else
            StatDecision ← Edge
        end if
        DispatchTask(TaskQueue.dequeue(), StatDecision)
    end while
end function
```

a strong basis for the following decision-making process. A critical complement to these statistical mechanisms is visual transparency. The OM is envisioned as supporting a real-time monitoring dashboard that continuously visualises the statistical state of the workload through intuitive plots, including box plots, skewness trend lines, and kurtosis indicators, thereby improving explainability and enabling human operators to monitor system behaviour, understand changes in task distributions, and react to anomalies through customisable alerts or triggers. This conceptual mapping is reinforced in Figure 4, where the operator dashboard visualises these statistical indicators in real time.

Following the creation of a standard measure for task complexity via COS, the subsequent aim is to develop a tier allocation system that adjusts to variations in workload features. The proposed Algorithm 1 presents a statistical method that uses the IQR of the COS distribution to establish tier limits, and in contrast to previous techniques that relied on fixed, arbitrary thresholds, the IQR-based approach determines its thresholds explicitly from current system data, allowing for smarter and context-sensitive task allocation. The IQR, representing the dispersion of the central 50% of a dataset, serves as a natural foundation for dividing tasks based on their relative difficulty. After calculating and collecting the COS values, they are arranged, and the first quartile (Q1) and third quartile (Q3) are obtained. These quartiles act as flexible benchmarks that direct tier selection. Tier allocation functions are based on three principles. Tasks with COS values beneath Q1 signify the least complex quarter of the workload, making these lightweight tasks ideal for IoT tier processing, saving higher-level resources for more intensive activities. Tasks with COS values falling between Q1 and Q3 comprise the central segment of the distribution and typically indicate standard or moderately complex workloads. These are directed to the Edge tier, designed to manage common tasks needing low-latency performance. Tasks with a COS greater than Q3 are categorised in the highest complexity group, and these are sent to the Cloud tier, where sufficient computing power can handle larger workloads without interfering with latency-sensitive tasks at the edge.

A significant advantage of this method is its capacity to automatically adjust to changes in workload composition and adapt to the uncertain real-world environment. As Q1 and Q3 are constantly recalculated, any alteration in the COS distribution, like a sudden rise in complex tasks, leads to an immediate modification of the tier boundaries, enabling the system to sustain an even distribution of tasks among tiers and avoid congestion at the Edge tier, even during varying workloads. In practical implementations, these quartile thresholds are displayed using a real-time box plot of COS values on the system dashboard managed by the coordination domain of the OM, enabling the human operator to monitor the system in real time. The graph allows operators to easily see how the concept of a standard task changes over time and how tasks are allocated across IoT, Edge, and Cloud tiers.

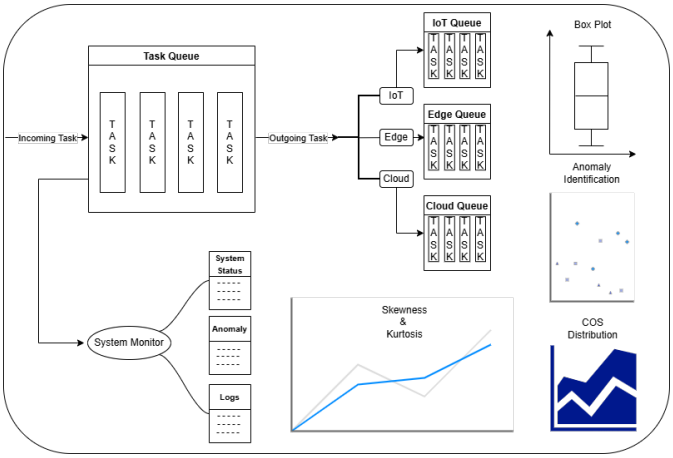

Fig. 4: A high-level abstracted view of the operator dashboard illustrating task flow, queue states, anomaly indicators, skewness and kurtosis trends, real-time offloading decisions, etc.

C. Visualisation of System State using Box Plots

The IQR-based allocation technique has a distinct benefit as its behaviour can be easily demonstrated using a box plot. A box plot effectively summarises a distribution by displaying its minimum, first quartile, median, third quartile, and maximum values, allowing for clear observation of the distribution of COS scores at any given time and enabling effective monitoring of the offloading system's decision logic in real time. The relationship between the allocation guidelines and the formation of the box plot is straightforward. The central box extends from Q1 to Q3, directly aligning with the range of COS values designated for the Edge tier, while the median line within this box represents the usual complexity level of the present workload. The lower whisker stretches from Q1 to the smallest non-outlier COS value and indicates tasks directed to the IoT tier, whereas the upper whisker extends from Q3 to the maximum non-outlier value, representing the group of tasks assigned to the Cloud tier. This depiction facilitates the creation of an intuitive monitoring interface. By examining the median's position, the box's width, and the whiskers' length, operators can swiftly determine if workloads are transitioning towards greater or lesser complexity. Extra visual cues for skewness and kurtosis can aid in the early identification of asymmetry or fluctuations in the workload distribution, and if these measures exceed established limits, the system can issue alerts to prompt human intervention before the onset of performance problems.

D. Anomaly Aware Workload Management

While the IQR-based approach successfully regulates the majority of task distribution, a robust offloading system must also identify and properly handle tasks that significantly differ from typical behaviour, since if they are not managed properly, some extreme situations may skew resource utilisation or adversely affect service quality. To tackle this, the SML-OA framework includes higher-order statistics, particularly skewness and kurtosis, facilitating a more sophisticated two-stage anomaly detection process as shown in Algorithm 1. Initially,

the OM consistently analyses the ongoing workload stream. Skewness provides an understanding of the COS distribution's symmetry. A prolonged negative skew signifies a higher influx of highly demanding tasks that could burden the Cloud tier, while a positive skew indicates an abundance of lightweight tasks. Kurtosis measures the extent to which the distribution has heavy tails, reflecting the probability of generating extreme values, and an increased kurtosis indicates a workload that is more unpredictable and susceptible to outliers, leading the system to take a more conservative approach.

These statistical metrics provide the framework for a deeper insight into workload behaviour beyond simple volume, enabling a more context-sensitive strategy for managing anomalies. The initial phase of detection applies the standard IQR rule, i.e., any COS value beyond the range $Q1 - 1.5 \cdot IQR$ to $Q3 + 1.5 \cdot IQR$ is preliminarily identified as an outlier. Nonetheless, the importance of this flag relies on the wider distribution. In situations with low kurtosis and a stable workload, such an outlier might present minimal operational risk, whereas when kurtosis is elevated and the situation is unstable, the same deviation gains significantly greater significance. Consequently, an anomaly is classified as high risk only if both criteria are satisfied, i.e, it must be identified by the IQR rule and take place during a timeframe in which system kurtosis surpasses a specified volatility limit. This dual condition enables SML-OA to adjust its sensitivity dynamically according to the predictability of the overall workload. Anomalies identified in this process are displayed on the dashboard by overlaying flagged tasks onto the COS box plot as red markers and providing real-time updates of skewness and kurtosis indicators, and the system can generate alerts when several high-risk anomalies occur in quick succession, allowing operators to respond swiftly to situations that may indicate developing performance problems or suspicious behaviour.

After detecting anomalies, the system activates a dual-level response strategy that differentiates isolated outliers from wider workload inconsistencies. For individual anomalies, any task identified as a high-risk outlier by the hybrid detection method is promptly removed from the main processing queue and, instead of following the usual offloading route, is sent to a protected sandbox space in the Cloud tier. This isolation process stops a single flawed or resource-heavy task from interfering with regular operations and also enables specific logging and diagnostic analysis. Conversely, when the system detects persistent anomalies in the COS distribution, like extended negative skewness or increased kurtosis, it views these signs as evidence of a significant change in the workload patterns. In these instances, the framework implements a coordinated approach across the entire system, which may involve flexibly modifying COS weights to deter certain task types, alerting human operators about irregularities, or, if needed, applying temporary admission control by limiting the arrival of new tasks until the workload stabilises. Every response is logged and displayed on the monitoring dashboard. Sandbox tasks are displayed in a specific visualisation panel, and system-level modifications like COS reweighting or rate limiting are noted alongside real-time trends of skewness and kurtosis. This blend of automatic protections and clear reporting guarantees dependable functioning even in unstable, hostile, or unpredictable workload environments.

E. Machine Learning as a Context-Aware Fallback Decision Engine

Algorithm 2 Machine Learning Powered Fallback Algorithm Under System Stress

```
Load pre-trained ML_Model (GBM or RL-based)
Initialise SystemMonitor to track CPU, memory, and latency
Define action space:
        0 → Edge, 1 → Cloud, 2 → StopOffloading, 3 → ReweightCOS
function MLFallback(task, Stats, COS_weights)
    SystemState ← SystemMonitor.GetCurrentState()
    If (SystemState.utilization_high or SystemState.latency_high) then
        StressFlag ← True
    Else
        StressFlag ← False
    EndIf
    If StressFlag = False then
        return NoFallback                    ▷ Call Algorithm 1
    EndIf
    FeatureVector ← ConstructFeatureVector(task, Stats, SystemState)
    Probs ← ML_Model.predict_proba(FeatureVector)
    MLDecision ← ArgMax(Probs)
    If MLDecision = 0 then
        return Edge
    ElseIf MLDecision = 1 then
        return Cloud
    ElseIf MLDecision = 2 then
        TriggerAdmissionHold()
        return StopOffloading
    ElseIf MLDecision = 3 then
        COS_weights ← AdaptWeights(COS_weights, Stats)
        return ReweightCOS
    EndIf
end function
```

The statistical tiering framework provides the primary decision mechanism within the SML-OA architecture, and by relying on COS values and IQR-based thresholds, it is able to make stable, interpretable, and adaptive allocation decisions during routine operation. However, these statistical rules alone may become inadequate when the system approaches resource saturation, such as when the Edge tier task queue is full, or its CPU utilisation is too high, or the IoT tier exhibits abnormal fluctuations in workload patterns. In such cases, the OM must prevent degradation in performance and QoS. To ensure reliable performance under these stressed conditions, the SML-OA incorporates a machine learning (ML) module that acts as

a context-aware fallback mechanism under "system-in-stress" conditions. When the system monitor, embedded within the local and coordination domains of each OM, detects instability such as elevated CPU utilisation, network congestion between different tiers, or excessive queue length, the OM triggers the "system-in-stress" state, activates the ML fallback mechanism using Algorithm 2, and sends an alert to the operator through the dashboard. The ML fallback mechanism employs a Gradient Boosting Model (GBM), which is chosen for its strong performance on structured data, resistance to outliers, and ability to provide insights through feature importance metrics. The GBM is trained on comprehensive historical data that captures both task-level attributes and global ecosystem conditions, leveraging a rich feature set comprising task-specific properties such as task size, resource requirements, and execution time, as well as real-time system state metrics such as CPU and memory utilisation at the Edge and Cloud tiers and inter-tier network latencies. Together, these features enable the model to proactively identify the most suitable decisions, including selecting appropriate placement location, stop offloading to edge nodes when necessary, etc., ensuring that the QoS is maintained or improved and preventing any degradation.

During offloading, the ML module is invoked only when the system monitor detects stress conditions. The OM passes the constructed feature vector for the incoming task to the GBM, which produces a probability distribution over action space mentioned in Algorithm 2. The task is then rerouted according to the ML prediction, overriding the statistical decision temporarily until the normal condition is established again. In this manner, the ML model "powers" the statistical offloading by providing a safety valve during conditions where statistical rules alone may lead to suboptimal performance. An operational walkthrough of the fallback mechanism has been demonstrated in section V-G using a practical use case. Through this integration, the system maintains the stability and interpretability of statistical allocation while leveraging machine learning for resilience and adaptability during periods of stress, resulting in the proposed hybrid offloading strategy that remains efficient under normal workloads yet is capable of sophisticated, context-sensitive decisions when operational conditions become unpredictable.

F. An Evolutionary Path to Autonomous Control with Reinforcement Learning

While effective, the supervised approach requires a comprehensive, pre-generated dataset and is limited by the scenarios it has seen during training. A more advanced and fully autonomous system can be built using RL, which allows the decision-making agent to learn and adapt continuously from live, online experience without needing a pre-labelled dataset. The problem is formulated as a Markov Decision Process (MDP):

- Agent: The Osmotic Manager.
- State Space (S): The state is characterised by the identical feature vector employed in the fallback model, giving the agent a thorough understanding of the existing environment. The characteristics pertinent to decision making, including CPU usage, memory consumption, and network delay, are presented in Table III.

TABLE III: Task and System Features

Feature Name	Category	Description
COS	Task	Composite Offloading Score for the current task.
Task_Size	Task	Raw size of the task payload.
Resource_Req	Task	Raw composite CPU and memory requirement.
Predicted_Exec_Time	Task	Raw predicted execution time on a baseline system.
COS_Q1	Queue	The 25th percentile of COS values in the current task queue.
COS_Q3	Queue	The 75th percentile of COS values in the current task queue.
COS_Median	Queue	The 50th percentile (median) of COS values in the queue.
COS_Skewness	Queue	Skewness of the COS distribution in the queue.
COS_Kurtosis	Queue	Kurtosis of the COS distribution in the queue.
Edge_CPU_Utilization	System	Current average CPU utilisation across all Edge nodes.
Edge_Memory_Utilization	System	Current average memory utilisation across all Edge nodes.
Cloud_CPU_Utilization	System	Current average CPU utilisation across relevant Cloud resources.
Network_Latency_IoT_Edge	System	Measured network round-trip time between IoT and Edge.
Network_Latency_Edge_Cloud	System	Measured network round-trip time between Edge and Cloud.

- Action Space (A): For each incoming task, the agent selects one of three discrete allocation actions, namely assignment to the IoT tier, the Edge tier, or the Cloud tier.
- Reward Function (R): The reward function is the most critical element, crafted to incentivise the desired global system behaviour. A potential multi-objective reward function can be defined as shown in Equation (6).

$$R_t = w_{\text{throughput}} \cdot \text{Throughput}_t \quad - w_{\text{latency}} \cdot \text{AvgLatency}_t \\ - w_{\text{penalty}} \cdot \text{FailedTasks}_t \tag{6}$$

Here, $Throughput$ is the number of tasks successfully completed, $AvgLatency$ is their average completion time, and $FailedTasks$ is the number of tasks that missed their deadlines. The weights are hyperparameters that tune the trade-off between maximising performance and minimising failures.

The agent's goal is to learn an optimal policy, $\pi(S) \to A$, that maps states to actions in a way that maximises the cumulative discounted reward over time. A Deep Q-Network (DQN) algorithm is a suitable choice for this problem. The DQN uses a neural network to approximate the Q-function, $Q(s,a)$, which represents the expected future reward of taking action a in state s. Through iterative interactions with the environment and a combination of exploration and exploitation, the agent improves its decision-making policy and converges toward an optimal offloading strategy.

This RL-based formulation has the potential to revolutionise task offloading in multi-tier computing architectures by enabling adaptive, intelligent, and context-aware decisions in real time. Its autonomy and capacity to generalise beyond predefined scenarios make it especially promising for dynamic and heterogeneous environments. A further dedicated study focusing on the design, implementation, and evaluation of such an RL-driven framework is planned in the near future.

V. Simulation Setup and Illustration

The proposed algorithm is simulated using a Python environment to illustrate its working principle and adaptive behaviour when interacting with dynamic input data. Since traces of a real-world IoT-Edge-Cloud federated environment dataset are complicated to collect, a synthetic dataset was created to test multiple scenarios, including static, bursty, heavy-tail, or skewed, to reproduce results across multiple independent runs. This section further presents a detailed demonstration of the statistical decision-making over sliding windows to illustrate the working principles of the proposed algorithm using synthetic test cases.

A. Generation of Synthetic Training Data

Since traces of a real-world IoT-Edge-Cloud federated environment dataset are complicated to collect, a 1000-synthetic dataset was created to mimic the generation of workloads from IoT devices for the sole purpose of training a tier-specific predictive model for the execution time of the new incoming tasks. Each task is attributed with three characteristics as discussed earlier in section IV-A. Each attribute is generated individually and then compiled together to define a task, uniquely identified by `TASK_ID`. A dataframe was created to collect all the individually generated tasks, which were later exported as a CSV file and served as a well-organised foundation for predictive modelling.

a) Task Size (S): Each task is assigned a data size in the range 0.001–500 GB, drawn from a lognormal distribution with $\mu = 0.5$ and $\sigma = 1.2$, which means most values are small, some values are moderate, while a few values are very large, creating a long right tail. Lognormal distribution is used instead of the commonly used normal distribution, because it is commonly used to model quantities that grow abruptly and real-world values are often skewed and not symmetric, so a normal distribution would be unrealistic. This heavy-tailed behaviour aligns with real IoT deployments, where small sensor packets coexist with occasional large multimedia or batch-processing uploads. The lognormal profile ensures realistic variability and skewness in the generated sizes.

b) Resource Requirement (R): It is modelled in terms of CPU demand, representing the computational requirement of each task expressed in compute units, and is drawn from a gamma distribution with $shape = 2.0$ and $scale = 15$. The resulting values are clipped between 1 and 100 to mimic realistic CPU utilisation. This distribution is commonly used to model bursty and heterogeneous compute loads like the ones arising from real-time IoT operations, such as event processing, inference workloads, and anomaly detection tasks.

c) Execution Time (T_{true}): Since this is our target variable for the predictive model, if we randomly generate it unrelated to either task size or resource requirement, the model will learn patterns in pure noise, resulting in catastrophic performance. To represent the prediction module in Equation 2 and ensure that the learning model captures an interpretable and stable mapping, the execution time for each task is defined by the controlled linear model shown in Equation 7. A linear formulation is chosen because execution time increases with task size, resource demand, and some inherent overhead.

$$T_{\text{true}} = aS + bR + c + \varepsilon, \tag{7}$$

Where S is the task size, R is the resource requirement, and a and b are constants that reflect their respective contributions to execution time, while the constant c is added for some fixed latency overhead that is present at the IoT tier, with values $a = 0.02$, $b = 0.3$, and $c = 5$. $\varepsilon \sim \mathcal{N}(0,1)$ represents small random gaussian noise, added to mimic unpredictable fluctuations in real-world execution times due to OS jitter, network delays, etc. This formulation embeds a known dependency structure between size, resource demand, and execution latency, enabling accurate benchmarking of the regression model's learning performance. To maintain consistency with edge-computing latency constraints, the generated execution times are clipped to the interval $[0, 50]$ ms, reflecting practical operational limits.

B. Training of the Execution Time Prediction Model

As the core principle of our proposed algorithm is dynamic adaptation, we integrate a machine learning module at all tiers to predict the execution time rather than assuming it. This eliminates the rigidity of previous algorithms and enables the system to adapt to real-world uncertainty by using historical data for predictive modelling. Scikit-learn was used for modelling the prediction module, and a linear regression model was chosen for its low computational overhead and its alignment with the controlled linear relationship defined during dataset generation. Because our algorithm adapts at runtime, prediction must be fast, and linear regression offers minimal computational cost, making it suitable for real-time or near-real-time prediction across all tiers of the system.

The proposed supervised regression model was trained using the previously generated synthetic dataset, utilising task size and resource requirements as predictors for execution time. The learning pipeline begins by selecting the two primary features that govern execution behaviour and subsequently divides the dataset using an 80/20 train–test split to ensure evaluation on unseen samples. Because the magnitudes of the two features differ significantly, StandardScaler normalisation is applied to both features, and the model is then trained on the scaled training set. This scaling ensures numerical stability and prevents larger-valued features from disproportionately influencing the regression coefficients.

C. Performance Evaluation

TABLE IV: Performance metrics evaluating the effectiveness of the linear regression model. The metrics include the R^2 score, MAE, and RMSE.

Metric	Value
R^2 Score	0.9691
Mean Absolute Error (MAE)	0.8003
Root Mean Squared Error (RMSE)	1.0367

The performance assessment of the linear regression model in predicting execution time of the IoT workload is shown

Fig. 5: Scatter plot showing the predicted versus actual values for the linear regression model. The dashed line represents the perfect prediction, while the scatter points represent the model's prediction.

through visual and quantitative analyses. The scatter plot in Figure 5 illustrates the predicted values against the actual values of the model, with the dashed line indicating perfect prediction. The model's outstanding predictive accuracy and capacity for successful data generalisation are highlighted by a notable degree of clustering along this line, demonstrating how the linear regression model can uncover the defined linear relationships between the input parameters and execution time. To provide a comprehensive assessment, key performance metrics have been summarised in Table IV. These include the R^2 score, which reached an impressive value of 0.9691, the MAE of 0.8003, and the RMSE of 1.0367, which highlight the small discrepancy between the actual and predicted values, indicating the accuracy and reliability of the model. These findings demonstrate the model's high degree of accuracy in predicting execution time with minimal errors, which makes it a crucial tool in the run-time federated environment.

D. Illustration

TABLE V: Synthetic Task Dataset Used for Algorithm Illustration

Task ID	Task Size (GB)	Resource Requirement (Units)	T_{pred} (ms)
1	17.0252	17.0605	10.4598
2	0.8913	36.6970	24.0098
3	1.5339	22.2945	11.6967
4	3.0483	19.1720	10.7894
5	21.6837	41.9427	15.4186
6	7.7460	9.0773	7.8465
7	1.5378	54.4410	21.3972
8	3.3846	13.9925	9.4363
9	2.3504	6.6397	6.2965
10	2.0961	15.0989	8.9807
11	15.9590	30.4829	12.4455
12	0.9477	9.5654	7.8444
13	33.8485	14.4299	19.2160
14	13.2670	27.3183	12.4419
15	18.1817	12.4009	9.0768

To illustrate how Algorithm 1 interacts with real data, we construct a controlled simulation using the trained linear regression model and a small synthetic workload. For demonstration, we generate a separate set of 15 synthetic IoT tasks, shown in Table V, using the same statistical distributions as the training dataset, i.e, lognormal for task size and gamma for resource requirements. For each $task_i$, its execution time T_i is obtained by applying the trained model to its features to represent the tier-specific prediction module in Equation (8).

$$T_i = \hat{f}_{\text{LR}}(S_i, R_i), \tag{8}$$

where S_i is the task size, R_i is the resource requirements, and $\hat{f}_{\text{LR}}(\cdot)$ denotes the learned regression function. These 15 tasks form the initial input task queue.

1) COS Weights, Data Structures for Modelling, and Stress-Condition Handling: For illustration purposes, we have used fixed weights for size, resource demand, and predicted time for the COS calculation as mentioned in Equation (9), subject to the constraint mentioned in Equation (5), and mathematically validated in Equation (10).

$$w_S = 0.4315, \quad w_R = 0.4163, \quad w_T = 0.1522, \tag{9}$$

$$w_S + w_R + w_T = 1.0000. \tag{10}$$

To simulate the operational behaviour of a hierarchical IoT-Edge-Cloud continuum, three logical queues are defined that represent the task buffering capability at each tier. These queues form the core data structures used to mimic resource availability, congestion, and stress conditions.

- IoT Task Queue: It initially stores all 15 tasks synthetically generated to mimic the IoT workload. It represents the task arrival buffer prior to tier-wise allocation.
- Edge Queue (Capacity $C_E = 5$): A constrained buffer modelling the Edge tier. The limited capacity reflects the resource-restricted nature of the edge nodes.
- Cloud Queue (Capacity $C_C = 10$): Although more scalable than the edge, it is intentionally modelled with a finite limit to enable controlled evaluation of "system-in-stress" scenario behaviour under concurrent load conditions.

In the real deployment, the "system-in-stress" conditions are intended to be handled through the ML-based fallback mechanism described in Algorithm 2. However, for simplicity of demonstration in this illustration scenario, the same stress-handling principle is abstracted and expressed through queue overflow, reproduced using an explicit if-else control structure shown in Algorithm 3. If the statistical model assigns a task to the Edge tier while the edge queue is full, the system flags a stress event and automatically redirects the task to the Cloud tier. This constitutes the first-level fallback response. If the Cloud queue is also found to be at full capacity, the system identifies a compounded stress event. In such cases, offloading is halted for the specific time window, and the decision is recorded as a "StopOffloading" state, the system enters a cooldown state until stable conditions are measured again. This hierarchical redirection preserves QoS under saturation and ensures deterministic stress-handling behaviour. This explicit formulation is used only as a pedagogical representation of how stress events are detected through queue overflow. In the

real system, the same logic is instead governed by the ML-powered fallback mechanism in Algorithm 2, which incorporates predictive decision making and context-aware offloading. This discussion only presents the conceptual and structural foundations of the statistical offloading; a thorough operational walkthrough of Algorithm 2, including the practical behaviour of the fallback strategy under stress, step-by-step task flow, redirection flows, etc., is explained using a practical use case in Section V-G.

Algorithm 3 Illustrative Stress-Handling Logic Using Explicit Conditional Checks

Input: Predicted tier decision *target_tier*
Input: Queue lengths

$$|edge\_queue| \leftarrow \text{len(edge\_queue)}$$

$$|cloud\_queue| \leftarrow \text{len(cloud\_queue)}$$

Parameters: C_E (Edge capacity), C_C (Cloud capacity)
▷ Check for Edge overflow
if *target_tier* = "Edge" **and** $|edge\_queue| \geq C_E$ **then**
    $stress\_flag \leftarrow$ True
    $stress\_action \leftarrow$ "EdgeFull_OffloadToCloud"
    $target\_tier \leftarrow$ "Cloud"
end if
▷ Check for Cloud overflow after fallback
if *target_tier* = "Cloud" **and** $|cloud\_queue| \geq C_C$ **then**
    $stress\_flag \leftarrow$ True
    $stress\_action \leftarrow$ "EdgeCloudFull_StopOffloading"
    $final\_tier \leftarrow$ "StopOffloading"
else
    $final\_tier \leftarrow target\_tier$
end if
return *final_tier*

E. Mathematical Demonstration

As mentioned in Algorithm 1, a sliding window is used to calculate the statistics of incoming tasks for statistical offloading, to dynamically adapt to the variability of the real-world data. In our simulation, the algorithm operates on a sliding window of size $W = 5$. For $N = 15$ tasks from Table V, we obtain three disjoint windows as shown in Equation (11).

$$\begin{aligned} W_1 &= \{1,2,3,4,5\}, \\ W_2 &= \{6,7,8,9,10\}, \\ W_3 &= \{11,12,13,14,15\}. \end{aligned} \tag{11}$$

For a generic window W_k, we denote the set of tasks as shown in Equation (12).

$$W_k = \{i \mid i = (k-1)W+1, \ldots, kW\}, \quad k \in \{1,2,3,....N\}. \tag{12}$$

where W denotes the window size, k represents the window index, and i corresponds to the task indices included in each window. The formulation ensures that every window W_k contains exactly W consecutive tasks, beginning from the index $(k-1)W+1$ and ending at kW. This guarantees that each window consists of a fixed number of sequential tasks, enabling a structured partitioning of the entire task set into disjoint segments. Within each window, Algorithm 1 executes the following sequence of operations:

1) Dynamic normalization of S_i, R_i, T_i is performed using Equation (3).
2) COS scores are computed using Equation (4).
3) The statistical characteristics of the COS distribution are derived, including quartiles, interquartile range, skewness, and kurtosis.
4) Anomaly detection is conducted based on outlier behaviour and kurtosis measures.
5) IQR-based tier assignment to IoT, Edge, or Cloud.

The subsequent discussion provides a detailed mathematical walkthrough for Window 1, which contains Tasks 1-5, and illustrates how the normalisation adapts when the window slides to Tasks 6-10. For the first window W_1, we have

$$W_1 = \{1,2,3,4,5\}$$

corresponding to Tasks 1-5 from Table V, and the raw feature values for Window 1 are shown in Table VI.

TABLE VI: Raw feature values for Window 1 (Tasks 1-5)

Task ID	S_i (GB)	R_i (units)	T_i (ms)
1	17.0252	17.0605	10.4598
2	0.8913	36.6970	24.0098
3	1.5339	22.2945	11.6967
4	3.0483	19.1720	10.7894
5	21.6837	41.9427	15.4186

For Window 1, the feature-wise minima, maxima, and ranges are

$$\begin{aligned} S^{(1)}_{\min} &= 0.8913, \quad S^{(1)}_{\max} = 21.6837, \quad \Delta S^{(1)} = 20.7924, \\ R^{(1)}_{\min} &= 17.0605, \quad R^{(1)}_{\max} = 41.9427, \quad \Delta R^{(1)} = 24.8822, \\ T^{(1)}_{\min} &= 10.4598, \quad T^{(1)}_{\max} = 24.0098, \quad \Delta T^{(1)} = 13.5500. \end{aligned} \tag{13}$$

1) Dynamic Normalisation: For each window W, the algorithm applies normalisation to the three features. For Window 1, the normalised values S'_i, R'_i and T'_i are computed using Equation (14), and for all $i \in W_1$, the ranges $\Delta S^{(1)}, \Delta R^{(1)}, \Delta T^{(1)}$ are taken from Equation (13).

$$S'^{(1)}_i = \frac{S^{(1)}_i - S^{(1)}_{\min}}{\Delta S^{(1)}}, \quad R'^{(1)}_i = \frac{R^{(1)}_i - R^{(1)}_{\min}}{\Delta R^{(1)}}, \quad T'^{(1)}_i = \frac{T^{(1)}_i - T^{(1)}_{\min}}{\Delta T^{(1)}} \tag{14}$$

For subsequent windows W_k, the same normalisation procedure is applied, but the ranges $\Delta S^{(k)}, \Delta R^{(k)}, \Delta T^{(k)}$ are recomputed for each window based on the feature values observed within that window. Consequently, the normalised values S'_i, R'_i, T'_i adapt to the local statistics of W_k. Because these ranges are updated at every window, the normalisation dynamically adjusts to the variability of the real-world workload and the changing task-load characteristics across windows. As an example, the normalised size of Task 1 is shown in Equation (15).

$$S'^{(1)}_1 = \frac{17.0252 - 0.8913}{20.7924} \approx 0.7760 \tag{15}$$

When the window slides to $W_2 = \{6,7,8,9,10\}$, the normalisation is recomputed from the statistics of the new subset only. For Window 2, the feature-wise extrema are

$$S_{\min}^{(2)} = 1.5378, \quad S_{\max}^{(2)} = 7.7460, \quad \Delta S^{(2)} = 6.2082,$$
$$R_{\min}^{(2)} = 6.6397, \quad R_{\max}^{(2)} = 54.4410, \quad \Delta R^{(2)} = 47.8013,$$
$$T_{\min}^{(2)} = 6.2965, \quad T_{\max}^{(2)} = 21.3972, \quad \Delta T^{(2)} = 15.1007.$$

Table VII compares the extrema values of the two consecutive windows. The dramatic change in ranges, particularly for task size and resource requirement, highlights the adaptive nature of the normalisation. This dynamic normalisation and re-estimation of statistics at every window boundary is what enables the SML-OA to remain sensitive to shifts in workload patterns across time, while preserving the interpretable behaviour defined by the COS and IQR-based tiering rules.

TABLE VII: Comparison of Feature Ranges Between Window 1 and Window 2

Feature	Window 1			Window 2		
	Min	Max	Range	Min	Max	Range
S (GB)	0.8913	21.6837	20.7924	1.5378	7.7460	6.2082
R (units)	17.0605	41.9427	24.8822	6.6397	54.4410	47.8013
T (ms)	10.4598	24.0098	13.5500	6.2965	21.3972	15.1007

2) COS Calculation: For each $task_i$ in Window 1, the COS is calculated using Equation (16).

$$\mathrm{COS}_i^{(1)} = w_S S_i'^{(1)} + w_R R_i'^{(1)} + w_T T_i'^{(1)} \tag{16}$$

As an example, the COS calculation for Task 3 is shown in Equation (17).

$$\mathrm{COS}_3^{(1)} = 0.4315 \cdot 0.0309 + 0.4163 \cdot 0.2104 + 0.1522 \cdot 0.0913 \approx 0.1148$$

Using Equation (14) and (16), all normalised values in Window 1 were calculated, followed by the COS computation, and the results are summarised in Table VIII.

TABLE VIII: Normalised features and COS values for Window 1

Task ID	S_i'	R_i'	T_i'	COS_i
1	0.7760	0.0000	0.0000	0.3348
2	0.0000	0.7892	1.0000	0.4807
3	0.0309	0.2104	0.0913	0.1148
4	0.1037	0.0849	0.0243	0.0838
5	1.0000	1.0000	0.3660	0.9035

3) Statistical Metrics for the Window: The COS values in W_1 are mentioned below:

$$\{\mathrm{COS}_i\}_{i=1}^5 = \{0.3348,\ 0.4807,\ 0.1148,\ 0.0838,\ 0.9035\}$$

Sorting these values in ascending order gives

$$\{0.0838,\ 0.1148,\ 0.3348,\ 0.4807,\ 0.9035\}$$

The empirical first and third quartiles, along with the IQR for Window 1, are

$$Q_1^{(1)} = \frac{0.0838 + 0.1148}{2} = 0.0993,$$
$$Q_3^{(1)} = \frac{0.4807 + 0.9035}{2} = 0.6921,$$
$$IRQ^{(1)} = Q_3^{(1)} - Q_1^{(1)} = 0.5928.$$

The lower and upper IQR fences used for outlier detection are

$$\mathrm{LF}^{(1)} = Q_1^{(1)} - 1.5\,\mathrm{IQR}^{(1)} = 0.0993 - 1.5 \times 0.5928 = -0.7899$$
$$\mathrm{UF}^{(1)} = Q_3^{(1)} + 1.5\,\mathrm{IQR}^{(1)} = 0.6921 + 1.5 \times 0.5928 = 1.5813$$

4) Anomaly Detection: For $task_i \in W_1$, Algorithm 1 declares an anomaly if both an outlier condition and a volatility condition hold as shown in Equation (17).

$$\left[\left(\mathrm{COS}_i^{(1)} > \mathrm{UF}^{(1)}\right) \wedge \left(\mathrm{Kurt}^{(1)} > K_{\text{threshold}}\right)\right] \tag{17}$$

where $K_{\text{threshold}}$ is the volatility threshold and set to 3.5 in our simulation.

All COS values in Window 1 lie in the interval $[\mathrm{LF}^{(1)}, \mathrm{UF}^{(1)}]$, hence no task is flagged as an outlier by the IQR rule. The shape statistics of the COS distribution are computed using the standardised central moments, as shown in Equations (18)-(21).

$$\mu = \frac{1}{5}\sum_{i=1}^{5} \mathrm{COS}_i. \tag{18}$$

$$\sigma^2 = \frac{1}{5}\sum_{i=1}^{5} (\mathrm{COS}_i - \mu)^2. \tag{19}$$

$$\text{Skewness} = \frac{1}{5}\sum_{i=1}^{5} \frac{(\mathrm{COS}_i - \mu)^3}{\sigma^3}. \tag{20}$$

$$\text{Kurtosis} = \frac{1}{5}\sum_{i=1}^{5} \frac{(\mathrm{COS}_i - \mu)^4}{\sigma^4}. \tag{21}$$

For Window 1, these evaluate numerically to

$$\mu \approx 0.3835, \quad \sigma \approx 0.2981.$$
$$\text{Skewness}^{(1)} \approx 0.72, \quad \text{Kurtosis}^{(1)} \approx 2.19.$$

Since the Kurtosis is below the volatility threshold

$$K_{\text{threshold}} = 3.5,$$

the distribution in Window 1 is not considered volatile, and no task satisfies the joint anomaly condition mentioned in Equation (17).

5) IQR-Based Tier Assignment: Ignoring stress for the moment, Algorithm 1 applies an IQR-based tiering rule to the COS values in each window as shown in Equation (22). For any task_i that is flagged as an anomaly, it is isolated and sent to a "sandbox" environment in the Cloud tier, and its statistical tier is given by $\mathrm{Tier}_{\text{Aglo 1}}^{(i)} = \text{Cloud}$.

$$\mathrm{Tier}_{\text{Aglo 1}}^{(i)} = \begin{cases} \text{IoT}, & \text{if } \mathrm{COS}_i < Q_1^{(i)}, \\ \text{Cloud}, & \text{if } \mathrm{COS}_i > Q_3^{(i)}, \\ \text{Edge}, & \text{otherwise.} \end{cases} \tag{22}$$

If the system monitors alerts "system-in-stress", here modelled using queue overflow, this statistical tiering is overridden. Let L_E and L_C denote the current lengths of the edge and cloud

queues, respectively, and assume that the system is stressed. The final tier after stress handling is shown in Equation (23).

$$\text{Tier}_{\text{final}}^{(i)} = \begin{cases} \text{StopOffloading}, & L_E \geq C_E \wedge L_C \geq C_C \\ \text{Cloud}, & \text{Tier}_{\text{Aglo 1}}^{(i)} = \text{Edge}, L_E \geq C_E \\ \text{Tier}_{\text{Aglo 1}}^{(i)}, & \text{otherwise} \end{cases} \tag{23}$$

Using $Q_1^{(1)} = 0.0993$ and $Q_3^{(1)} = 0.6921$, the COS values in Table VIII lead to the assignments summarised in Table IX. For this first window, queue capacities are not yet saturated, so no stress fallback is triggered.

TABLE IX: Tier decisions for Window 1 based on COS and IQR rules

Task ID	COS_i	Assigned Tier
1	0.3348	Edge
2	0.4807	Cloud
3	0.1148	IoT
4	0.0838	IoT
5	0.9035	Cloud

F. Operator Dashboard

Since our system is equipped with an operator dashboard that supports real-time monitoring and decision-making, it displays multiple visual analytics. To demonstrate examples of visuals that could be shown, we generated them based on the 15 illustration tasks. These visualisations help operators interpret statistics in real-time, enabling them to analyse COS distributions, validate tiering decisions, examine statistical shape characteristics, and identify potential anomalies. With these insights, operators can make informed decisions such as adjusting COS weightings or temporarily halting offloads for cooldown.

1) COS Distribution Visualization: Figure 6 presents the box plot of COS values across all 15 tasks. The visualisation highlights the central tendency, dispersion, and potential extreme values of the scheduling metric. The operator can see this box plot, check for the presence of outliers and be ready for an anomaly. The median and IQR reflect the balance between edge-leaning tasks and cloud-leaning tasks, while the whiskers indicate the range of statistically acceptable COS values.

2) COS Scatter Plot: Figure 7 shows the COS value of every task and the tier they have been assigned to through colour coding. The scatter distribution demonstrates how the IQR-based tiering logic, augmented by system stress and anomaly rules, assigns tasks differently based on COS magnitude. From the scatter plot, a clear separation is visible between tasks with high COS values that are offloaded to the Cloud tier and the tasks with low to moderate COS values that are distributed across the IoT-Edge tier. This clustering highlights the significance of COS in determining tier assignment. For the last task, i.e. `TASK_ID` $= 15$, we observe that it was assigned to the Cloud tier despite having a moderate COS value. This behaviour is explained by the activation of the "system-in-stress" mechanism when the Edge capacity becomes saturated. This example also demonstrates the value of visual interpretability for analysing system status in real time.

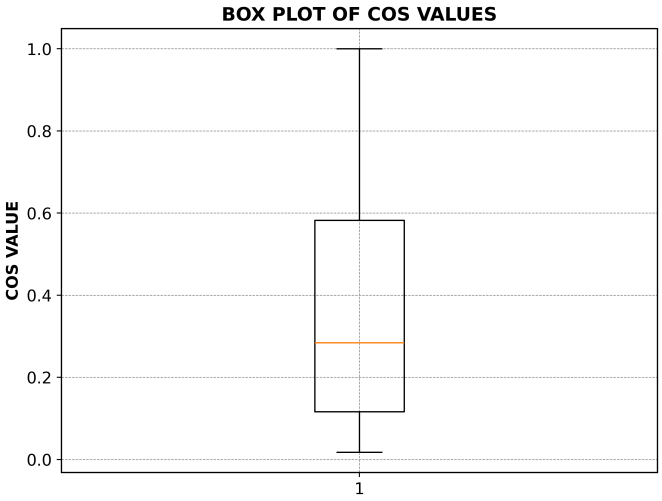

Fig. 6: Box plot of COS for all tasks.

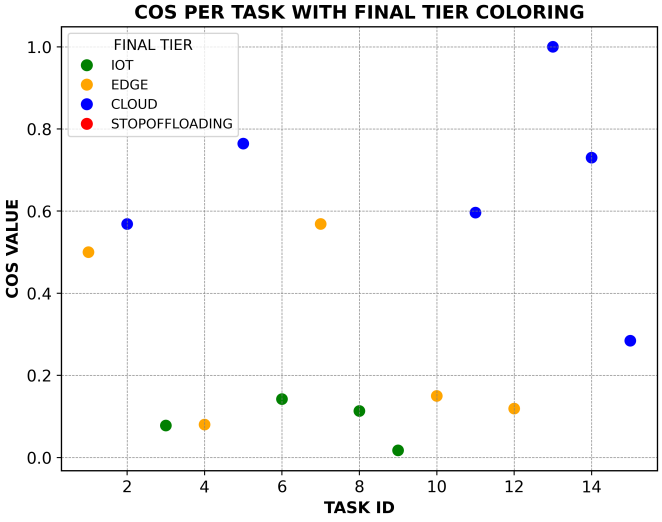

Fig. 7: COS values per task with colour-coded final tier assignments.

3) Skewness and Kurtosis Trends: Figure 8 presents the skewness and kurtosis trends computed incrementally as more tasks enter the queue. This visualisation is essential for assessing distributional stability, because algorithmic anomaly detection depends on tail heaviness, i.e. kurtosis, and asymmetry, i.e. skewness. Large spikes in kurtosis for `TASK_ID` 12 correspond to volatile windows that may trigger anomaly handling behaviour. When viewed together with Figure 6, the operator can more easily detect the presence of an anomaly and take the necessary actions.

4) Tier-Wise Task Distribution: Figure 9 depicts the number of tasks assigned to the IoT, Edge, and Cloud tiers. This distribution changes in real time, enabling the operator to analyse the workload on each tier. It can be used to identify which tier processes the highest number of tasks and which tier is underutilised, allowing the operator to manually offload tasks in order to boost throughput and improve QoS.

5) Anomaly Identification: Figure 10 visualises anomalous tasks detected using the joint condition of IQR outliers and high kurtosis described in Equation (17). Although no anoma-

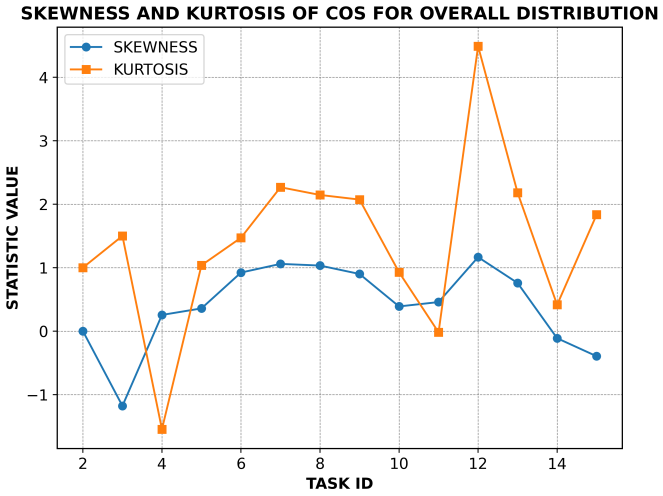

Fig. 8: Skewness and kurtosis evolution of COS values across task arrivals.

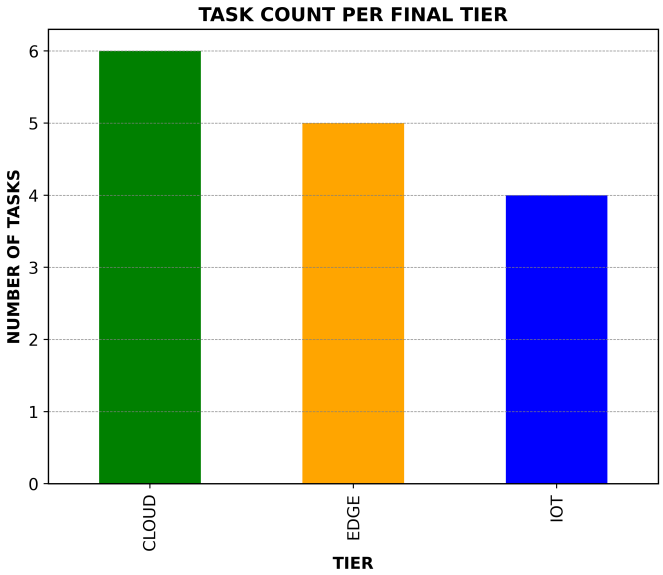

Fig. 9: Task count per final assigned tier.

lies were present in our synthetic dataset, the dashboard still highlights potential detections clearly. This helps operators monitor extreme cases or volatility spikes and take the necessary action under critical conditions.

G. Operational Walkthrough of Algorithm 2 with a Practical Use Case

To illustrate how Algorithm 2 operates during real deployment, consider a heterogeneous workload containing periodic sensor readings, mixed edge-processing tasks, and one computationally intensive job. The example highlights how the ML fallback mechanism intervenes only when statistical tiering from Algorithm 1 becomes unreliable due to stress, saturation, or QoS threats, under "system-in-stress".

a) Initial Stable Regime: At system start, the platform operates under low utilisation conditions. The COS follow a well-behaved, lightly skewed distribution, and the SystemMonitor reports no abnormal CPU or latency readings. Under

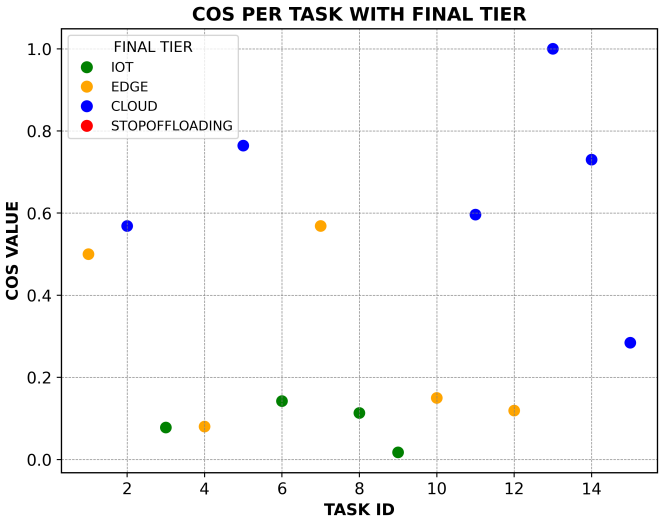

Fig. 10: Identified anomalies in COS distribution using IQR and kurtosis-based detection.

these conditions, Algorithm 2 returns *NoFallback*, delegating the decision entirely to Algorithm 1.

b) System Under Stress and ML Fallback Activation: As the workload progresses, a wave of moderate COS tasks arrives. Initially, they are placed on the Edge tier via Algorithm 1. However, once edge utilisation exceeds a certain threshold, say approximately 90%, the SystemMonitor indicates resource saturation, and StressFlag is set to true, triggering the ML fallback mechanism. Utilising the real-time feature vector containing COS, statistics, queue depth, latency, and resource utilisation, the ML model outputs a probability distribution across the action space

$$\{0 : \text{Edge}, 1 : \text{Cloud}, 2 : \text{StopOffloading}, 3 : \text{ReweightCOS}\}.$$

As a result, the statistical decision is superseded, and later tasks are directed according to the final decision of Algorithm 2. This conduct avoids latency breaches and preserves overall system efficiency. QoS degradation and service delay are prevented, even though statistical IQR-based logic alone would have continued assigning them to Edge.

VI. Conclusion

This study introduced the Statistical and Machine Learning Powered Offloading Algorithm (SML-OA), a cohesive system for smart task allocation among IoT, Edge, and Cloud resources. The suggested design incorporates a composite offloading score for feature-aware task characterisation, a dynamic statistical tiering mechanism based on windowed IQR analysis, a hybrid anomaly detector leveraging distributional shape statistics, and a machine learning powered fallback module that activates under stress or QoS risk. Collectively, these components allow the framework to adaptively react to changing workload features, transcending the inflexibility of traditional threshold-dependent or rule-based offloading methods. The simulation with synthetic IoT workloads showed that the framework reliably separates routine activity from

high-impact outliers, remains stable as data distributions shift, and makes sound decisions even when statistics alone fall short. Future research will focus on integrating reinforcement learning for long-term policy optimisation, exploring more flexible and non-linear COS formulations, and validating the system within a real-world IoT-Edge-cloud federated environment.

REFERENCES

[1] B. Rababah, T. Alam, and R. Eskicioglu, "The next generation internet of things architecture towards distributed intelligence: Reviews, applications, and research challenges," *Journal of Telecommunication, Electronic and Computer Engineering (JTEC)*, vol. 12, no. 2, pp. 11–19, 2020.

[2] C. Perera, A. Zaslavsky, P. Christen, and D. Georgakopoulos, "Sensing as a service model for smart cities supported by internet of things," *Transactions on emerging telecommunications technologies*, vol. 25, no. 1, pp. 81–93, 2014.

[3] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM computer communication Review*, vol. 44, no. 5, pp. 27–32, 2014.

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

[5] J. Almutairi and M. Aldossary, "A novel approach for iot tasks offloading in edge-cloud environments," *Journal of cloud computing*, vol. 10, no. 1, p. 28, 2021.

[6] B. Neha, S. K. Panda, P. K. Sahu, K. S. Sahoo, and A. H. Gandomi, "A systematic review on osmotic computing," *ACM Transactions on Internet of Things*, vol. 3, no. 2, pp. 1–30, 2022.

[7] V. Nguyen, T. T. Khanh, T. D. Nguyen, C. S. Hong, and E.-N. Huh, "Flexible computation offloading in a fuzzy-based mobile edge orchestrator for iot applications," *Journal of Cloud Computing*, vol. 9, no. 1, p. 66, 2020.

[8] I. A. Elgendy, W.-Z. Zhang, C.-Y. Liu, and C.-H. Hsu, "An efficient and secured framework for mobile cloud computing," *IEEE Transactions on Cloud Computing*, vol. 9, no. 1, pp. 79–87, 2018.

[9] H. Tyagi and R. Kumar, "Cloud computing for iot," in *Internet of Things (IoT) Concepts and Applications*. Springer, 2020, pp. 25–41.

[10] A. Morshed, P. P. Jayaraman, T. Sellis, D. Georgakopoulos, M. Villari, and R. Ranjan, "Deep osmosis: Holistic distributed deep learning in osmotic computing," *IEEE Cloud Computing*, vol. 4, no. 6, pp. 22–32, 2017.

[11] K. Bilal, O. Khalid, A. Erbad, and S. U. Khan, "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers," *Computer Networks*, vol. 130, pp. 94–120, 2018.

[12] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[13] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of systems architecture*, vol. 98, pp. 289–330, 2019.

[14] Y. Sahni, J. Cao, S. Zhang, and L. Yang, "Edge mesh: A new paradigm to enable distributed intelligence in internet of things," *IEEE access*, vol. 5, pp. 16 441–16 458, 2017.

[15] P. Cong, J. Zhou, L. Li, K. Cao, T. Wei, and K. Li, "A survey of hierarchical energy optimization for mobile edge computing: A perspective from end devices to the cloud," *ACM Computing Surveys (CSUR)*, vol. 53, no. 2, pp. 1–44, 2020.

[16] T. G. Rodrigues, K. Suto, H. Nishiyama, and N. Kato, "Hybrid method for minimizing service delay in edge cloud computing through vm migration and transmission power control," *IEEE Transactions on Computers*, vol. 66, no. 5, pp. 810–819, 2016.

[17] Q. Yuan, H. Zhou, J. Li, Z. Liu, F. Yang, and X. S. Shen, "Toward efficient content delivery for automated driving services: An edge computing solution," *IEEE Network*, vol. 32, no. 1, pp. 80–86, 2018.

[18] L. Tang and S. He, "Multi-user computation offloading in mobile edge computing: A behavioral perspective," *IEEE Network*, vol. 32, no. 1, pp. 48–53, 2018.

[19] H. Guo and J. Liu, "Collaborative computation offloading for multiaccess edge computing over fiber–wireless networks," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 5, pp. 4514–4526, 2018.

[20] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *2013 Proceedings Ieee Infocom*. IEEE, 2013, pp. 1285–1293.

[21] Y. Liu, M. J. Lee, and Y. Zheng, "Adaptive multi-resource allocation for cloudlet-based mobile cloud computing system," *IEEE Transactions on mobile computing*, vol. 15, no. 10, pp. 2398–2410, 2015.

[22] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[23] Y. Sun, S. Zhou, and J. Xu, "Emm: Energy-aware mobility management for mobile edge computing in ultra dense networks," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2637–2646, 2017.

[24] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM transactions on networking*, vol. 24, no. 5, pp. 2795–2808, 2015.

[25] Y. Xiao and M. Krunz, "Qoe and power efficiency tradeoff for fog computing networks with fog node cooperation," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[26] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE communications surveys & tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[27] B. Yang, X. Cao, J. Bassey, X. Li, and L. Qian, "Computation offloading in multi-access edge computing: A multi-task learning approach," *IEEE transactions on mobile computing*, vol. 20, no. 9, pp. 2745–2762, 2020.

[28] B. Wang, C. Wang, W. Huang, Y. Song, and X. Qin, "A survey and taxonomy on task offloading for edge-cloud computing," *IEEE Access*, vol. 8, pp. 186 080–186 101, 2020.

[29] H. Lin, S. Zeadally, Z. Chen, H. Labiod, and L. Wang, "A survey on computation offloading modeling for edge computing," *Journal of Network and Computer Applications*, vol. 169, p. 102781, 2020.

[30] C. Jiang, X. Cheng, H. Gao, X. Zhou, and J. Wan, "Toward computation offloading in edge computing: A survey," *IEEE Access*, vol. 7, pp. 131 543–131 558, 2019.

[31] A. Shakarami, A. Shahidinejad, and M. Ghobaei-Arani, "A review on the computation offloading approaches in mobile edge computing: A g ame-theoretic perspective," *Software: Practice and Experience*, vol. 50, no. 9, pp. 1719–1759, 2020.

[32] J. Wang, J. Pan, F. Esposito, P. Calyam, Z. Yang, and P. Mohapatra, "Edge cloud offloading algorithms: Issues, methods, and perspectives," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–23, 2019.

[33] B. Neha, S. K. Panda, and P. K. Sahu, "An efficient task mapping algorithm for osmotic computing-based ecosystem," *International Journal of Information Technology*, vol. 13, no. 4, pp. 1303–1308, 2021.

[34] B. Neha, S. K. Panda, P. K. Sahu, and D. Taniar, "Energy and latency-balanced osmotic-offloading algorithm for healthcare systems," *Internet of Things*, vol. 26, p. 101176, 2024.