Cluster Computing (2026) 29:430
https://doi.org/10.1007/s10586-026-06233-3

Performance modeling in FaaS workloads at the edge using machine learning

Federica Filippini[1] · **Marco Savi**[1] · **Michele Ciavotta**[1]

Received: 30 June 2025 / Revised: 6 May 2026 / Accepted: 16 May 2026

Abstract

The Function as a Service (FaaS) paradigm has emerged as a compelling architectural model for both cloud and edge computing environments, enabling the execution of self-contained functions triggered by specific events while abstracting from developers infrastructure management complexities such as load balancing and auto-scaling. In FaaS-enabled clusters, particularly within resource-constrained edge environments, precise resource consumption estimation becomes critical to optimize resource utilization, minimize latency, prevent system overloads, and ensure scalability. This paper addresses performance modeling challenges in FaaS-enabled distributed and decentralized edge computing systems, operating at the granularity level of both nodes and individual functions. We propose a Machine Learning-based framework designed to predict key performance indicators, including CPU utilization, memory, and energy consumption, based on incoming workload patterns, while simultaneously forecasting potential system overload conditions. Moreover, our approach introduces a profiling methodology that characterizes serverless functions according to their resource consumption profiles, thereby enabling accurate prediction of node-level resource demands without requiring detailed knowledge of individual deployed functions. Experimental validation demonstrates that our predictive models achieve 97% accuracy in anticipating node overload scenarios, providing a robust foundation for proactive resource management in edge-based FaaS deployments. Moreover, while our best individual, function-based regression models predict node-level CPU, RAM and power consumption with a Mean Absolute Percentage Error below 9% on average, our experiments highlight the effectiveness of function profiling and cluster-based modeling. On one hand a novel multi-target regressor based on a permutation-invariant neural architecture is proved to generalize effectively across previously unseen workload compositions in the tested scenarios, with R^2 scores ranging from 0.94 to 0.98, i.e., aligned with the performance of specialized single-target regression models. On the other hand, cluster-level models generalize effectively to previously unseen functions within the same usage class, maintaining prediction errors within practically acceptable ranges and often in the single-digit percentage range.

Keywords Function as a service · Performance modeling · Machine learning · Edge computing

1 Introduction

The Function as a Service (FaaS) [1] paradigm is widely adopted in cloud computing, where it provides a highly scalable and event-driven framework to manage computational resources. This architectural model employs self-contained, stateless units of code (i.e., functions) that are activated by specific events or client requests. By delegating complex management decisions, including resource provisioning, load distribution, and dynamic scaling, to service providers, developers can focus exclusively on the application logic. This approach pushes the pay-per-execution billing model

✉ Federica Filippini
federica.filippini@unimib.it

Marco Savi
marco.savi@unimib.it

Michele Ciavotta
michele.ciavotta@unimib.it

[1] Department of Informatics, Systems and Communication, University of Milano-Bicocca, Viale Sarca, 336, 20126 Milan, Italy

Published online: 30 June 2026

to its limits as computational costs are incurred solely during function execution cycles [2].

In recent years, industrial and academic initiatives increasingly demonstrate the viability and advantages of deploying FaaS architectures beyond centralized cloud platforms. Contemporary solutions target private and hybrid cloud ecosystems [3], encompassing edge computing [4–6], decentralized edge computing [7] and Internet-of-Things infrastructures [8]. In this paper, we focus on *decentralized edge computing* architectures, featuring heterogeneous interconnected nodes across multiple geographical locations [7]. These distributed systems offer several significant advantages over traditional centralized approaches, including improved fault tolerance through distributed processing, enhanced scalability through load distribution, and reduced network congestion. In this context, the ability to predict the future resource state of edge nodes depending on the expected workload is a central requirement for the reliable management of decentralized FaaS platforms: unlike centralized cloud clusters, edge resources are usually limited and nodes are subject to strict Quality of Service (QoS) constraints (e.g., on energy consumption and latency) [9]. As a result, reactive mechanisms based on threshold violations are often insufficient to prevent service degradation, particularly in the presence of high workload volatility, since corrective actions may be triggered only after overload conditions have already occurred.

Accurate performance models that relate workload composition to node-level resource consumption (e.g., CPU, memory, and energy usage) enable the system to anticipate critical states and to support resource management decisions such as request routing, admission control, and load redistribution based on expected, rather than observed, conditions, reducing the probability of QoS violations [7, 10]. Although extensive benchmarking has characterized the resource consumption of serverless clusters (e.g., [5, 11]), methods that can generalize across different functions, nodes, and deployment configurations to provide accurate forecasts remain largely unavailable.

This paper builds upon our previous work [12], introducing a Machine Learning *(ML)-based framework* to predict critical performance metrics (e.g., CPU usage, memory usage, and energy consumption) for functions and computing nodes, with a particular emphasis on deployments in resource-constrained edge environments. Our models forecast system overload conditions based on incoming workload patterns at both the node and function granularity, which is essential to prevent service disruptions. Additionally, we developed a comprehensive profiling and classification system that categorizes functions according to their resource utilization profiles. This enables model training using a cluster-based methodology rather than individual function-specific approaches, significantly improving applicability by enabling node-level predictions without requiring detailed knowledge of individual deployed functions.

The key contributions of this work are:

- We propose a comprehensive framework to predict critical performance indicators (CPU utilization, memory consumption, and energy usage) in FaaS-enabled edge environments, enabling proactive detection of node overload conditions before they impact system performance.
- The framework integrates adaptive sampling strategies, model training and evaluation. This approach incorporates generalization techniques based on function profiling and clustering methodologies that classify serverless functions according to their resource consumption patterns, culminating in a runtime adaptive inference phase.
- We develop both function-level and cluster-level predictive models. In particular, we extend our previous work [12] by proposing a permutation-invariant neural architecture that generalizes effectively across previously unseen workload compositions and deployment scenarios.
- We demonstrate the practical effectiveness of our solution through comprehensive experimental evaluation in edge computing environments. This significantly expands the preliminary analysis conducted in [12], which already demonstrated a prediction accuracy up to 97% for overload detection, as follows:
  - We introduce a comparison among several regression techniques for function-based modeling, observing that our best-performing method predicts the node-level CPU, RAM and power consumption with a Mean Absolute Percentage error of 3.51%, 1.60% and 8.83% on average, respectively;
  - We discuss the effectiveness of permutation-invariant models for multi-target regression, which achieve a strong predictive accuracy with R^2 scores ranging from 0.94 to 0.98, i.e., aligned with the performance of specialized single-target regression models;
  - We validate the efficacy of our module for function profiling prediction, observing that it always manages to correctly assign previously-unseen functions to the cluster that more closely matches their resource consumption patterns.

The remainder of this paper is organized as follows: Section 2 reviews the state of the art in performance modeling for FaaS systems. Section 3 describes our motivational scenario, while Section 4 presents our proposed framework, covering the data collection process, the development of

the profiling and classification tool, and model generation. Finally, Sections 5 and 6 present experimental validation and conclude the paper.

2 Related work

The adoption of FaaS systems at the edge and in the edge-to-cloud continuum poses significant challenges [13], among which is the need to model the performance of applications in a highly volatile context [14–16]. For this reason, the development of analytical or ML-based models to predict the performance of applications running on edge and cloud systems is gaining momentum in the research landscape. However, recent works related to resource usage forecasting in the edge or cloud mostly consider general or microservice workflows [17–20], while those with a specific focus on serverless platforms and FaaS usually target the prediction of the incoming workload (e.g., in terms of number of future function invocations [21] or of concurrent invocations [22]), the average arrival/response time [23–25], mechanisms related to the containers warm and cold start [26], or the functions latency in the presence of co-location [16]. To the best of the authors' knowledge, considerations on the resource usage are mostly left to benchmarking suites [5, 11, 27, 28], often tailoring cloud serverless platforms. Only the authors of [29] focus on performance modeling by considering resource usage (i.e., CPU and RAM) in a similar way as we do. However, due to the fact that they focus on a cloud computing scenario, their work does not consider forecasting of the system overload status.

More in details, [17] proposes μP, a microservices development framework that enables the prediction of response times and other metrics related to the utilization of microservices *by design*, leveraging Layered Queuing Networks. The work in [18] focuses on both the short- and long-term prediction of resource usage in the cloud, through a self-adapting data-driven system that automatically selects the optimal ML-based prediction algorithm according to a preliminary load analysis, and leverages the usage forecasts to plan the resource provisioning and perform scaling actions. Differently, [19] proposes a framework to predict the performance degradation of microservices based on the type and amount of incoming requests, to be used for a proactive management of cloud-based applications; in particular, the framework supports the adoption of black-box ML models for the regression of general performance metrics.

Concerning the ability of predicting the incoming workload in a FaaS scenario, the authors of [21] aim at forecasting the future number of function invocations to support energy-efficient function scheduling at the edge. This is different from our approach, as we estimate whether a node is going to be in an overload state by directly looking at the resource consumption, and not at the number of function invocations. Indeed, different types of functions concurrently executed on the same node may lead to different resource consumption patterns, which entails that simply counting the number of replicas is not enough to accurately estimating the system status.

The work in [22] defines a metric, called *function capacity*, that indicates the maximal number of concurrent invocations that a function can serve without violating existing agreements; in addition, it proposes an estimation tool to estimate such a metric. The methodology adopted in [22] is similar to ours, but the performance modeling does not directly involve resources such as RAM and CPU. In addition, the work is tailored to cloud computing environments, without any focus on edge scenarios, and to the possibility of experiencing nodes overload states. A very similar scenario is analyzed in [16], which characterizes the $90th$ percentile of functions latency depending on the number of concurrently deployed functions and their saturation/cache status. This is fundamentally different from our approach, where saturation (i.e., overload conditions) is an output of the model: our framework indeed predicts resource consumption (particularly CPU and RAM) both at the functions level and for the node where functions are deployed, and learns to anticipate overload based on the rate of incoming requests.

With respect to the ability of modeling the requests arrival time and/or the system response time, authors in [23] exploit the Seasonal Auto Regressive Integrated Moving Average time series forecasting model to predict the arrival of new requests, which is used to guide the decisions of a prediction-based autoscaler. The authors of [24] focus on the trade-off between performance and cost, adopting the response time as the key performance metric. The end-to-end response time and the execution costs of serverless functions are also targeted by [25], which proposes an analytical model based on the characterization of delays and transition probabilities in a complex workflow, represented as a directed graph including cycles, self-loops, loops, and parallel paths. An analytical performance model is also developed in [26] under the assumption of Poisson request arrivals; based on the incoming workload, authors characterize system metrics as, e.g., the rate of cold and warm starts, the probability of rejections, the average response time, and the mean utilization of active resources. Although relevant to our context, in these works there is no focus on directly estimating computational resources consumption (e.g. CPU and RAM), which is pivotal to understand whether a node is in an overload state.

The work in [20] leverages a Long-Short-Term-Memory network to predict low-level performance metrics as, e.g.,

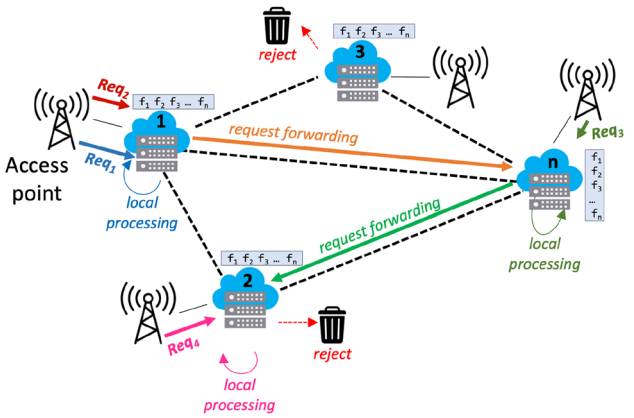


Fig. 1 Motivational scenario: a network of an edge computing node, each running an instance of the *DFaaS* platform [7]

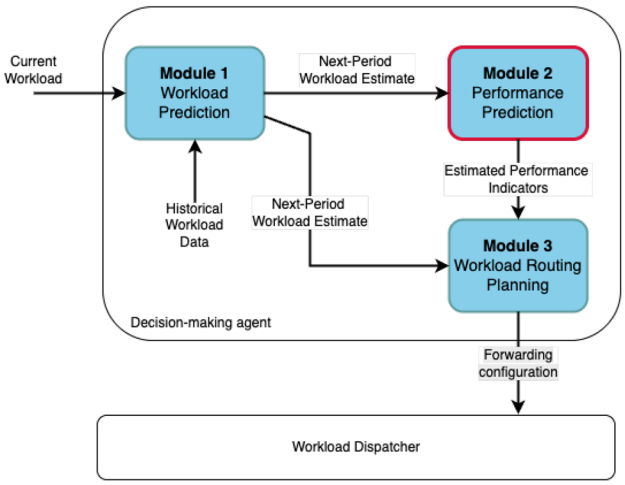


Fig. 2 Node-level agent architecture. Historical workload data is used to forecast next-period demand, estimate performance, and compute a routing strategy

the instructions per cycle based on recently observed measurements in cloud-native or microservice settings. This methodology builds upon the assumption that recent past behavior is informative of near-future utilization, which may not hold in FaaS deployments at the edge due to the bursty and event-driven nature of serverless workloads. For this reason, our framework explicitly relates the expected incoming load to resource consumption and latency, rather than extrapolating from recent utilization trends. Furthermore, [20] is not designed for FaaS environments and does not address function-level profiling, workload-driven prediction, or overload anticipation in decentralized edge deployments.

Finally, the authors of [29] propose a tool to profile the performance and resource utilization (i.e., CPU and RAM) of FaaS workloads using regression models, with the goal of enabling an accurate performance and cost prediction. Experiments were conducted in a cloud computing scenario, with focus on AWS Lambda and IBM Cloud functions. Our work takes inspiration from [29], but our focus is on performance modeling *at the edge*, where resources are constrained and nodes can become overloaded, with a consequent need of workload re-distribution.

3 Motivation

Figure 1 illustrates a representative scenario composed of telecommunication towers enhanced with local compute and storage resources–referred to as Edge Nodes (ENs). Each EN hosts an instance of the DFaaS (Decentralized Function-as-a-Service) platform, originally introduced in [7], and contributes to a fully decentralized coordination protocol. DFaaS implements the serverless execution paradigm in a peer-to-peer edge environment. In this model, nodes autonomously execute serverless functions in response to incoming client requests. When the predicted workload on a node is expected to exceed its available resources, the node dynamically redistributes excess requests to neighboring ENs. This cooperative offloading reduces congestion and improves system-wide efficiency.

A key architectural feature of DFaaS is the absence of centralized coordination. All nodes are functionally equivalent, operate independently, and make decisions based solely on local observations and interactions with peers. Consequently, there is no global controller orchestrating load distribution or topology maintenance.

In this context, load management is carried out at the node level through a decision-making agent comprising three integrated modules (see Figure 2). The first module predicts the upcoming workload [30]. The second module estimates resource consumption and the overall operational state of the node under the expected workload, providing essential context for informed decision-making. The third module derives a forwarding configuration that encodes an optimized routing strategy, leveraging algorithms such as heuristics [10] and Reinforcement Learning [31]. This forwarding configuration defines the proportion of incoming requests to be processed locally, offloaded, or rejected, based on the predicted state of the computing node. The configuration is then enforced by the Workload Dispatcher, which applies the strategy at runtime.

The framework we introduce in this paper–publicly available on GitHub[1]–serves as the second module in the decision-making agent. Its importance in enabling effective load management strategies becomes evident when contrasted with the simple *Static Strategy* proposed in [7], which employs fixed per-function thresholds without

[1] https://github.com/unimib-datAI/dfaas/tree/main/metrics_predictions

accounting for the node overall state. Consider, for example, a node hosting two functions, f_1 and f_2, with individual thresholds of 50 req/s and 60 req/s, respectively. Under the Static Strategy, the node is deemed capable of handling the load if f_1 and f_2 each receive requests up to their threshold rates. However, should f_1 receive 51 req/s while f_2 remains idle (0 req/s), the strategy would still flag the node as overloaded, despite available capacity.

This scenario exposes the inherent limitations of static, per-function thresholds, underscoring the need for precise, node-wide resource estimations to enable effective load distribution strategies.

4 Proposed framework

The main requirements that guided the definition of this framework are: (i) predicting performance metrics when new functions are dynamically deployed; (ii) ensuring scalability by making predictions for new functions without requiring model training for each possible configuration; and (iii) supporting incremental learning and model adaptation. As a consequence, the framework is not designed to be directly used by the clients generating traffic toward the serverless platform. Instead, its intended users are the platform administrators or infrastructure managers responsible for operating and maintaining the edge FaaS deployment. The predictive information produced by the framework is meant to support management-level decisions, such as resource provisioning, request routing, admission control, load redistribution, and energy-aware orchestration. Indeed, while many serverless platforms allow users to specify per-function resource limits (e.g., CPU and memory caps per instance), this form of resource provisioning does not in itself guarantee that overload cannot occur at the node level. These limits are enforced per container/function replica, but the underlying orchestrator can still schedule multiple such replicas on the same node, so that their aggregate demand under a burst of requests temporarily exceeds the node's physical capacity. Moreover, overload can also arise from pressure on shared kernel and network resources (e.g., file descriptors, connection tracking, buffers), which are not fully controlled by per-container limits. By providing forecasts of node-level resource utilization and potential overload conditions, the framework supplies actionable information that can be integrated into existing control loops, enabling more informed and proactive resource management strategies.

The resulting approach is embodied in a comprehensive ML pipeline, illustrated in Figure 3.

During the initial bootstrap phase, the **data collection** stage employs a *Samples Generator* that orchestrates controlled load testing experiments to profile functions deployed on the node. Through the node *Observability Platform*, it collects multi-dimensional performance metrics, including CPU usage, memory consumption, execution latency, and throughput measurements.

The **model training and selection** stage consists of two complementary learning processes. First, the *Function Profiler* applies unsupervised clustering to partition the function space into semantically coherent clusters based on resource consumption profiles. Second, the *Model Trainer* trains and evaluates a set of performance models, possibly leveraging the cluster information.

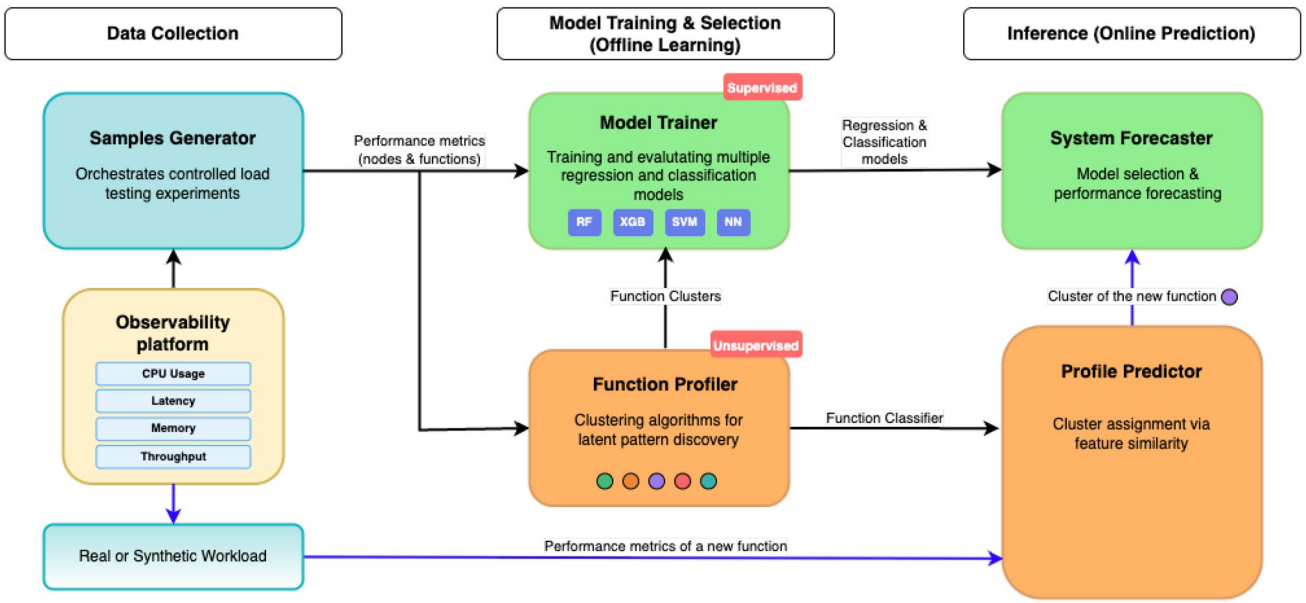


Fig. 3 Architecture of the proposed framework

During the **inference** stage, the *System Forecaster* predicts node performance for next-period workload estimates (see Section 3).

The next sections provide a detailed description of the framework components and how the performance models are generated.

4.1 Data collection

This section describes the data collection process used to gather training data for the performance prediction models. The process is managed by the *Samples Generator*, a component responsible for producing synthetic workload patterns that target deployed functions, and collecting system telemetry at both the function and node levels.

In the following subsections, we detail the feature space and provide a detailed breakdown of the data acquisition methodology.

4.1.1 Feature space definition

We define our feature vector to include both system resource utilization metrics and application-level performance indicators.

The node-level features comprise:

- *Per-node CPU utilization* (CPU_n): the percentage of used CPU on a node *n*, obtained as the sum of the percentages used by the individual cores (if, e.g., there are 4 CPU cores on *n*, CPU_n can vary between 0% and 400%).
- *Per-node memory utilization* (RAM_n): the value (in Bytes) of RAM used on *n* in the specified time interval, computed as the average difference between the node total memory and the available RAM. Analogously, we have also computed the *percentage* memory utilization ($RAM_n\%$) by dividing RAM_n by the total memory available on the node.
- *Per-node power consumption* (Π_n): Node energy consumption can be measured by external devices (e.g., wattmeters) or through software exporters. Although the first one provides higher accuracy, we followed the second approach as it allows to monitor also individual functions.

Function-level features capture application-specific behavioral patterns through:

- *Number of function replicas* (ρ_f), *and CPU* (cpu_f), *memory* (ram_f) *and power consumption* (π_f): To monitor the resource consumption of a function *f*, we first identify its replicas. Their number ρ_f is an important parameter to characterize the system response to workload changes. Moreover, cpu_f, ram_f, and π_f are computed as the sum of the CPU, RAM and power consumption, respectively, across all containers deployed to execute the function *f*.
- *Functions success rate and average latency* (s_f and l_f): These metrics express the percentage of requests that receive a positive response and quantify the average time it takes for a request to be served, respectively.

4.1.2 Sampling strategy

The *Samples Generator* implements a systematic experimental design methodology to replicate various traffic conditions. The system accepts user-defined parameters, including the set of functions to test, the maximum request rate per function, the scraping period, and the total experiment duration to ensure comprehensive coverage of the configuration space for ML model training.

Our sampling algorithm operates through three coordinated phases: (i) systematic configuration exploration using a uniform sweep approach across the parameter space, (ii) data collection and overload detection, and (iii) adaptive configuration space pruning to eliminate redundant sampling points. These phases are detailed in the following paragraphs and formalized in the complete sampling algorithm presented in Algorithm 1.

Configuration Space Exploration

To ensure accurate relative performance assessment, the *Samples Generator* first establishes baseline measurements for critical performance metrics under quiescent conditions: CPU utilization ($\overline{CPU}_n$), memory consumption ($\overline{RAM}_n$), and power consumption ($\overline{\Pi}_n$). These reference values serve as normalization factors during subsequent analysis phases, providing a consistent foundation for comparing configurations across different operational states. With these reference metrics established, the system then employs a systematic sweep approach to explore the configuration space, enabling the platform manager to specify the minimum and maximum workload they want to consider for profiling, and a step that is used to progressively increase the request rate. This uniform sampling methodology ensures comprehensive coverage of the operational range while maintaining consistent granularity across all explored configurations.

Finally, to ensure measurement independence between experimental trials, the system implements a systematic cool-down protocol following each test iteration. The cool-down duration $T_{cooldown}$ is determined by two convergence criteria: (i) CPU, memory, and power consumption must converge within 15% of baseline values, and (ii) function replica counts must return to minimal levels (fewer than 2

replicas per function). The recovery phase has a minimum duration of 10 seconds and continues until both criteria are met.

Data Collection and Overload Detection

During each configuration exploration step, the system collects comprehensive performance metrics through the node *Observability Platform*. A critical component of the collected dataset is the overload state label, which cannot be directly observed but must be derived from the raw performance metrics. We define the overload state Σ_n of node *n* using a multi-criteria assessment framework that processes the collected measurements. The system is classified as overloaded ($\Sigma_n = 1$) when any of the following empirically-validated conditions are satisfied:

- *Function Performance Degradation*: Average success rate across all deployed functions falls below 95%, formally expressed as $\frac{1}{|F_n|}\sum_{f\in F_n} s_f < 95\%$, where F_n represents the function set on node *n*.
- *CPU Saturation*: CPU utilization exceeds 80% ($CPU_n > 80\%$). This threshold ensures that sufficient capacity is preserved to adequately manage load spikes while maintaining responsiveness, and it aligns with Cloud Providers practice.
- *Memory Pressure*: Memory utilization surpasses 80% ($RAM_n > 80\%$). As above, the threshold is chosen to provide adequate buffering for peak demand and essential system operations, and it aligns with Cloud Providers practice[2].
- *Individual Function Failure*: At least one function is overloaded, that is it exhibits success rate below 90% ($s_f < 90\%$)[3]. This condition recognizes that individual failures significantly impact the overall Quality of Service.

Accurately identifying when a computing node is overloaded enables resource optimization and the maintenance of high system performance, thereby preventing potential bottlenecks and service degradation. This is particularly important in distributed FaaS contexts, where, as mentioned in Section 3, the system state significantly influences critical decisions, such as load balancing among nodes [10].

We note that the overload thresholds reported above are empirically defined based on common operational practices and preliminary observations. While these values provide a reasonable trade-off between responsiveness and stability, different threshold configurations may influence the labeling of borderline cases. In particular, lower thresholds would lead to earlier overload detection at the cost of increased false positives, whereas higher thresholds may delay detection but reduce sensitivity. A systematic sensitivity analysis is left for future work.

Adaptive Configuration Space Pruning

To optimize data collection, the sampling algorithm incorporates an intelligent pruning mechanism that leverages dominance relationships to reduce the configuration space exploration. When a configuration $c_{dom} = (\lambda_{f_1}, \lambda_{f_2}, \ldots, \lambda_{f_k})$ (where λ_{f_i} represents the request rate for function f_i) receives an overload label ($\Sigma_n = 1$), the algorithm identifies and eliminates dominated configurations from the remaining sampling queue.

A configuration $c = (\lambda'_{f_1}, \lambda'_{f_2}, \ldots, \lambda'_{f_k})$ is dominated by c_{dom} if $\lambda'_{f_i} \geq \lambda_{f_i}$ for all functions $i \in \{1, 2, \ldots, k\}$ with strict inequality for at least one function rate. This component-wise dominance relationship ensures that configurations with higher request rates are automatically classified as overloaded without explicit sampling.

Require: F, Λ_{max}, step δ
Ensure: Dataset $\mathcal{D} = \{(\mathbf{c}_i, \mathbf{y}_i, \Sigma_i)\}$
$\mathcal{C} \leftarrow \text{GenConfigSpace}(F, \Lambda_{max}, \delta)$
$\mathcal{D} \leftarrow \emptyset$; $\overline{CPU}, \overline{RAM}, \overline{\Pi} \leftarrow \text{Baseline}()$
while $\mathcal{C} \neq \emptyset$ **do**
  $\mathbf{c} \leftarrow \text{SelectNext}(\mathcal{C})$
  $\mathcal{C} \leftarrow \mathcal{C} \setminus \{\mathbf{c}\}$
  $\text{InjectLoad}(\mathbf{c})$
  $\mathbf{y} \leftarrow \text{CollectMetrics}()$
  $\Sigma \leftarrow \text{DetectOverload}(\mathbf{y}, F)$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{c}, \mathbf{y}, \Sigma)\}$
  if $\Sigma = 1$ **then** ▷ Overloaded
    $\mathcal{C}_{dom} \leftarrow \text{FindDominated}(\mathbf{c}, \mathcal{C})$
    $\mathcal{C} \leftarrow \mathcal{C} \setminus \mathcal{C}_{dom}$
    for $\mathbf{c}' \in \mathcal{C}_{dom}$ **do**
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{c}', \text{null}, 1)\}$
    end for
  end if
  $\text{CoolDown}(\overline{CPU}, \overline{RAM}, \overline{\Pi})$
end while
return $\mathcal{D}$

Algorithm 1 Sampling with Adaptive Pruning

4.2 Model training and selection

This section describes the model training and selection process that transforms collected performance data into predictive models. The process involves two main components: the *Function Profiler*, which applies unsupervised clustering to classify functions based on resource consumption

[2] https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Best_Practice_Recommended_Alarms_AWS_Services.html

[3] The threshold is chosen to ensure a high standard of reliability; while its value can be easily adjusted in different scenarios, we considered that the success rate of a functions workflow in AWS drops below 70% if the average s_f is 85%, and below 20% if it is 75% [32].

patterns, and the *Model Trainer*, which develops regression and classification models using either individual function features or features aggregated across functions within each cluster. The following subsections detail the function profiling methodology and model development pipeline.

4.2.1 Profiling and clustering

Function profiling enables our framework to learn over function classes rather than individual functions, achieving generalizability to unseen functions within known behavioral categories. This abstraction eliminates the combinatorial complexity of modeling every possible function: new functions require only classification into existing clusters via the *Profile Predictor*, enabling zero-shot performance prediction without retraining.

The *Function Profiler* constructs function representations by extracting statistical moments (i.e., mean, minimum, and maximum) across all observed metrics and load conditions. This triplet encoding captures both the central tendency and variability of resource consumption patterns, forming a compact yet expressive signature of function behavior that enables effective clustering and subsequent generalization.

To ensure numerical stability and comparability across heterogeneous metrics, all features undergo MinMax normalization to the [0, 1] range. The resulting feature vectors are then processed through Principal Component Analysis (PCA) to reduce dimensionality while preserving essential behavioral characteristics. Our framework allows to specify a threshold on the percentage of explained variance, thus retaining components that balance information preservation with computational efficiency.

The dimensionality-reduced feature vectors v' serve as input to an unsupervised clustering algorithm that partitions functions into behaviorally coherent classes. We employ the K-means algorithm[4], which iteratively assigns functions to clusters based on Euclidean distance to cluster centroids in the PCA-transformed feature space.

The optimal number of clusters k is initially estimated using the *elbow* method. However, this heuristic does not guarantee optimal performance for downstream prediction tasks. To address this limitation, our framework refines the cluster count by evaluating the performance of models trained on function classes derived from values in a neighborhood of k_e (typically $k_e \pm 1$). This model-driven validation ensures that the selected partitioning optimizes predictive accuracy while limiting computational overhead.

Table 1 Performance model classification: CB (Cluster-Based), FB (Function-Based), ST (Single-Target), MT (Multi-Target)

Input Features	Single-Target	Multi-Target
Cluster-Based	CB-ST	CB-MT
Function-Based	FB-ST	FB-MT

[4] We note that other implementations of the proposed framework are possible, e.g., PCA could be easily replaced by neural manifold fitting techniques and K-means by density-based clustering.

4.2.2 Modeling pipeline

The development of predictive performance models constitutes the second core phase of our methodology. Once sufficient performance data is collected, the *System Forecaster* component is responsible for training ML models to estimate node- and function-level performance metrics, forming the predictive engine of our framework.

As will be detailed later, the *System Forecaster* incorporates training strategies that develop a separate model for each target variable (**single-target**), but also a **multi-target** strategy that trains a model to predict all the performance indicators simultaneously. Even in the first case, it is relevant to observe that individual-target models can be subsequently combined into an integrated forecasting pipeline. Target variables correspond to the performance indicators introduced in Section 4.1.1, such as CPU_n, RAM_n, Π_n, cpu_f, ram_f, π_f, Σ_n, and l_f.

The features used for prediction include the function profiles, the incoming load to each function (**function-based**) or function class and the aggregated incoming load (**cluster-based**), as well as the type of node on which functions are deployed, characterized by its hardware specifications (see Section 5.1). The overall classification is shown in Table 1.

The modeling pipeline is shared between individual functions and function clusters, differing only in an initial aggregation step applied to the latter. Specifically, for each functions cluster g, the *System Forecaster* (i) computes aggregate metrics CPU_g, RAM_g, and Π_g by summing corresponding per-function values (e.g., cpu_f); (ii) sets $\Sigma_g = 1$ if at least one function in g is overloaded; and (iii) computes the average latency L_g from the l_f values of the class. Node-level metrics (e.g., CPU_n) remain unchanged.

Preprocessing steps are applied to all datasets prior to training. Outliers are removed to improve robustness, and the Synthetic Minority Over-sampling Technique (SMOTE) [33] is applied to mitigate imbalances in categorical target variables. All features are normalized via MinMax scaling, and the data are randomly split into training and test subsets.

To automate model selection and hyperparameter tuning, our framework integrates with open-source AutoML libraries, namely AutoGluon[5] and aMLLibrary[6]. AutoGluon supports a variety of state-of-the-art algorithms–including

[5] https://auto.gluon.ai/stable/index.html

[6] https://github.com/aMLLibrary/aMLLibrary

LightGBM (LGBM), CatBoost, XGBoost (XGB), and neural networks (NNs)–and combines bagging with *K*-fold cross-validation to enhance generalization [34]. In contrast, aMLLibrary–based on the `scikit-learn` ecosystem[7]–supports XGB and NNs, but also algorithms such as Decision Trees (DT), Random Forest (RF), Non-Negative Least Squares, and Support Vector Regression (SVR). It enables automatic hyperparameter optimization through grid search or Bayesian optimization via HyperOpt[8], as well as feature engineering and selection strategies [35].

Our *System Forecaster* is designed to consider, throughout the analysis, not only standard but also quantile regression [36], a statistical method that estimates the α-quantile of a dependent variable. Quantile regression is particularly relevant in performance prediction, as it allows for more robust interpretation in critical scenarios. For instance, by using the 0.95-quantile, we estimate a resource consumption threshold such that 95% of actual values are expected to fall below it. This is highly beneficial in cases where overestimating resource usage can help mitigate the risk of forwarding requests to overloaded nodes, thereby enhancing overall system efficiency. The quantile regression model uses the *quantile loss function*, denoted as $\mathcal{L}_Q$, to adjust model weights, unlike standard regression, which traditionally uses Mean Squared Error. In, e.g., 0.95-quantile regression, errors where observations exceed the forecast are penalized less than those below the forecast. This capability enables a more informed prediction of resource consumption under near-peak conditions, making it particularly advantageous for proactive capacity planning and to avoid node overload in edge computing environments.

Generalization Strategies

Function-based models suffer from a fundamental limitation: they cannot generalize to function combinations outside their training distribution. This restricts their utility in environments where novel workload compositions arise continuously. The combinatorial explosion of possible function mixtures makes it impractical to enumerate all configurations during training, necessitating models that can extrapolate to unseen combinations.

We address this challenge through two complementary approaches. First, cluster-based models capitalize on the observation that functions with similar execution profiles exhibit comparable resource consumption patterns. By learning representations at the cluster level rather than individual function level, these models achieve compositional generalization, that is, they can predict performance for arbitrary combinations of functions within known clusters, even when the specific mixture was never observed during training.

Second, we develop a permutation-invariant neural architecture based on Janossy pooling [37] (see Figure 5) that directly learns the mapping from function sets to performance metrics. This approach leverages the fundamental symmetry of the problem: node performance depends on the functions deployed, rather than their ordering.

Formally, we model node performance as $\varphi : \mathcal{F}^* \to \mathbb{R}^d$, where $\mathcal{F}^*$ denotes the set of all finite multisets over function space $\mathcal{F}$, and d is the dimensionality of performance metrics. The key insight is that φ must be permutation-invariant: $\varphi(\{f_1, f_2, \ldots, f_k\}) = \varphi(\{\pi(f_1), \pi(f_2), \ldots, \pi(f_k)\})$ for any permutation π, since node resource consumption depends only on the functions deployed, not their arrangement.

Our implementation employs a sequence-to-vector neural architecture where each function is represented by a tuple $(\lambda_{f_i}, \mathrm{e}_{f_i}, \mathrm{n})$: the input load λ_{f_i}, a function embedding e_{f_i} derived from execution profiles via PCA, and a node-type encoding n. The model processes these representations through Gated Recurrent Unit (GRU) layers followed by a Multilayer Perceptron (MLP). Permutation invariance emerges from the training dynamics: by randomly permuting input sequences at each step, the model learns to map all orderings of a given function set to identical outputs, effectively approximating the expectation over all permutations and ensuring invariance to input ordering.

4.3 Inference

The inference stage represents the operational phase where trained models provide real-time performance predictions for load management decisions. The *System Forecaster* supports two primary modeling approaches that can be configured based on system requirements and preferences.

When the deployed function set is fully known and characterized, the framework offers flexibility in model selection. The first approach utilizes cluster-based models: given a workload forecast $(\lambda_{f_1}, \lambda_{f_2}, \ldots, \lambda_{f_k})$ for the next prediction period, the system aggregates the load by cluster as $\Lambda_g = \sum_{f_i \in C_g} \lambda_{f_i}$, where C_g represents the set of functions assigned to cluster g. These cluster loads, along with node-specific features, serve as input to the trained cluster-based models to predict system-level metrics such as CPU_n, RAM_n, Π_n, and the overload state Σ_n. Alternatively, the system can be configured to use function-based models that operate directly on individual function profiles and their estimated workloads without cluster aggregation.

When new functions are deployed and, therefore, their execution profile is not yet known to the system, the

[7] https://scikit-learn.org/stable/

[8] https://hyperopt.github.io/hyperopt/

framework supports two complementary profiling modes. The first leverages the *Samples Generator* for rapid characterization through controlled load testing. The second adopts gradual profiling using actual user traffic, building the function profile incrementally.

Once sufficient performance data is collected for the new functions, the system can proceed with model-based inference using either of the two available approaches. If cluster-based models are selected, the *Profile Predictor* implements cluster assignment through feature similarity analysis. In particular, a feature vector v' characterizing the new function is computed through PCA following the same procedure applied during training (see Section 4.2.1), and used to assign the function to the cluster with the nearest centroid in the PCA-transformed feature space. Subsequently, the cluster-based models can generate predictions using the aggregated cluster loads.

Alternatively, if function-based models are selected, the system can directly employ permutation-invariant recurrent models (e.g., Janossy pooling networks as described in Section 4.2.2) that take individual function profiles as input, bypassing the need for cluster assignment and operating directly on the characterized function set.

5 Experimental analysis

This section details the setup (see Section 5.1) and results (see Section 5.2) of the experiments carried out to evaluate the accuracy of the trained models and the impact of the function clustering approach.

5.1 Experimental setup

The framework was tested using OpenFaaS[9] platform and Prometheus[10] as the metrics collection system. Node-Exporter[11], cAdvisor[12] and Scaphandre[13] exporters were integrated to collect performance metrics, both at the node and individual function level, gathering, in particular, the average CPU, RAM and power consumption over 10*s* time intervals. For what concerns power consumption, Scaphandre does not currently provide a native query to directly aggregate the energy usage of all replicas of a function based on the function name. Therefore, we implemented a custom solution to map system-level measurements to individual functions by: (1) identifying the Process Identifiers (PIDs) associated with each running function and its replicas, (2) querying Prometheus to gather the power consumption metrics exposed by Scaphandre at the process level, and (3) aggregating them to account for all active replicas when estimating per-function energy consumption.

Load is injected through Vegeta[14], a tool that can generate a configurable flow of HTTP requests targeting specific endpoints. Note that, despite the choice of exploiting OpenFaaS for our experiments, the framework we propose is agnostic to the specific platform, as long as they enable the integration with the aforementioned metrics exporters.

We selected 12 functions with diverse resource requirements from OpenFaaS's function repository[15] for our study, with no access to their source code. Five of these functions – `figlet`, `shasum`, `env`, `nmap`, and `curl` – were used for all experiments reported in the next sections; all the others were introduced when considering group-level performance in Sections 5.2.4–4.3. The main characteristics of these functions are summarized in Table 2. Moreover, for the initial set of experiments comparing different regression models we added to the five OpenFaaS functions a sixth one, `eat-memory`, specifically developed for this research, allowing us to investigate the cluster's behavior using a function with well-known internal characteristics. Specifically, `eat-memory` allocates 1 MB of memory and induces a 1-second delay.

Finally, we evaluated three distinct types of computing nodes, selected to align with common configurations used in edge network architectures [38]. Specifically, we instantiated three virtual machines (VMs) as follows: a *light* node with 2 CPUs and 8 GB of RAM, a *mid* node with 4 CPUs and 16 GB of RAM, and a *heavy* node with 6 CPUs and 24 GB of RAM. These configurations were designed to provide a comprehensive perspective on potential edge network setups, facilitating an assessment of architectural efficiency relative to the available computational resources. All VMs were hosted on a server running Linux Ubuntu 22.04, equipped with 8 CPUs and 32 GB of RAM. However, each VM was instantiated and tested in isolation to avoid interferences that could compromise the accuracy of the experimental results.

To characterize system behavior under different operational regimes, we varied the incoming request rate for each function up to a maximum of 200 req/s. We note that saturation points vary significantly depending on both the function type and the underlying node configuration. In particular, the maximum sustainable request rate is strongly influenced by node capacity: computationally intensive functions that can sustain rates close to 200 req/s on the *heavy*

[9] https://docs.openfaas.com/

[10] https://prometheus.io/

[11] https://github.com/prometheus/node_exporter

[12] https://github.com/google/cadvisor

[13] https://hubblo-org.github.io/scaphandre-documentation/index.html

[14] https://github.com/tsenart/vegeta

[15] https://github.com/openfaas/store-functions?tab=readme-ov-file

Table 2 Main characteristics of OpenFaaS store functions

Function name	Main purpose/operation	Typical workload type
figlet	Convert input text into large ASCII-art banners using the figlet CLI.	Text formatting, CPU-light
shasum	Compute a SHA checksum for the request body or input file via the shasum tool.	Cryptographic hash, CPU-bound
nmap	Run the nmap network scanner against a target host or range.	Network I/O + CPU-intensive
env	Echo or inspect environment variables/request context (debug-style helper).	Diagnostic, very lightweight
curl	Issue HTTP(S) requests to arbitrary URLs using curl and return the response.	Network I/O-bound
qrcode-go	Generate a QR code image from input text using Go libraries.	CPU-light + image generation
sentimentanalysis	Perform sentiment analysis (e.g., positive/negative) on input text.	CPU-bound ML/NLP, stateless
face-detect-pigo	Detect faces in input images using the Pigo face detection library.	CPU-intensive image processing
face-blur	Detect and blur faces in images to anonymize them.	CPU-intensive image processing
certinfo	Fetch and show TLS/SSL certificate information for a given host.	Network I/O + light CPU
markdown	Render Markdown input to HTML using Go middleware.	Text processing, CPU-light
openfaas-text-to-speech	Convert input text into speech audio using a TTS engine.	CPU-bound audio generation

node may reach saturation below 10 req/s on the *light* node, while lightweight functions may not saturate even at high request rates on more powerful nodes. As a result, no single request-rate threshold consistently corresponds to saturation across all function–node combinations. The selected upper bound of 200 req/s therefore represents a practical limit that ensures coverage of heterogeneous operational regimes (low-, medium-, and high-load conditions) across all configurations. Importantly, overload conditions are explicitly captured in the dataset and reflected in the node-state classification task, which achieves high predictive accuracy. We note that the generated workloads include both steady and progressively increasing request rates, as well as configurations that emulate burst-like conditions due to the uniform sweep combined with adaptive pruning.

Overall, the dataset comprises approximately 170,000 sampled configurations across all node types and workload combinations. Due to the pruning strategy described in Section 4.1.2, only approximately 10% correspond to overload conditions (as explained in Section 4.2.2, oversampling during pre-processing ensures class balance).

All models whose performance are discussed in the next sections were trained either on Linux 5.15.0–170-generic x86_64 machine with 20 cores and 100Gb RAM (for Section 5.2.2) or on a MacBook Pro M3 with 11 cores and 18Gb RAM (Sections 5.2.3–5.2.7).

5.2 Experimental results

This section presents the experimental results of our study. Specifically, we provide in Section 5.2.1 an empirical analysis of performance heterogeneity across functions, establishing the fundamental motivation for our work. Then, we report in Section 5.2.2 the comparison among several single-target function-based regression models trained with AutoGluon and aMLLibrary (see Section 4.2.2), considering several metrics: Mean Absolute Percentage Error (MAPE), Root Mean Squared Error (RMSE), R^2 score, Mean Absolute Error (MAE), and Mean Squared Error (MSE). Section 5.2.3 presents results from a function-based (permutation-invariant) multi-target architecture. Using the best-performing function-based architecture, namely LGBM, we investigate the impact of cluster cardinality k in Section 5.2.4, and, considering the optimal value $k = 4$, demonstrate that cluster-based models can accurately predict node-level metrics without requiring knowledge of individual function deployments. Moreover, we present in Section 5.2.5 a detailed evaluation of the single-target cluster-based LGBM Regressor models trained to predict both node-level and cluster-level metrics (e.g., the node-level CPU consumption CPU_n and the CPU consumption related to functions in the high-resource-usage cluster), and of the LGBM Classifier that evaluates the node overloaded state, whose accuracy is computed via precision, recall, and F1-score. Finally, Section 5.2.6 analyzes the cluster assignment accuracy of our Profile Predictor for unseen functions, and Section 5.2.7 discusses the generalization capability of the cluster-based performance modeling approach to unseen functions.

We acknowledge that all models considered in this study are based on ML techniques, as our objective is to exclusively adopt data-driven methods; accordingly, a comparison with non-data-driven benchmark approaches is left for future work.

5.2.1 Function performance characterization

This section provides a preliminary overview of the the resource consumption and latency patterns across the function suite described in Section 5.1

Figure 4 reveals distinct performance signatures across functions. Functions `eat-memory` and `nmap` exhibit similar resource consumption profiles across all metrics, while `figlet`, `shasum`, and `env` form another coherent group. The `curl` function demonstrates markedly different behavior patterns, except for average latency (Figure 4d).

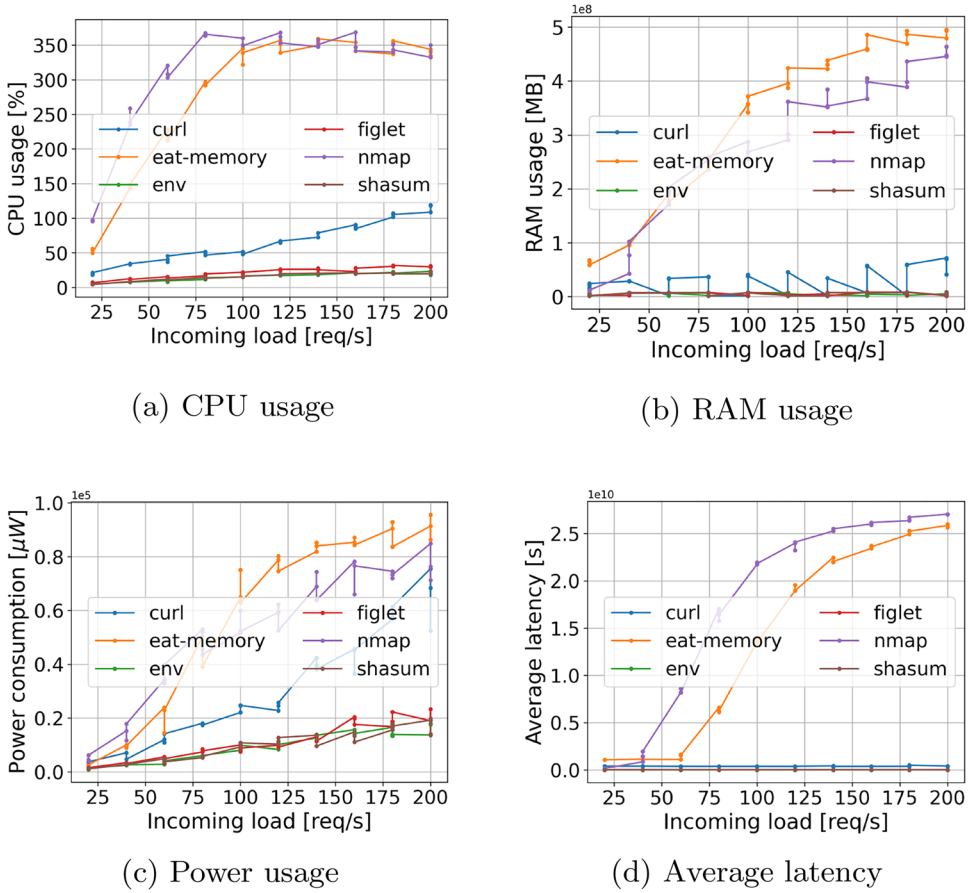


Fig. 4 Performance metrics of different functions

This performance heterogeneity, while suggesting natural clustering structure, proves insufficient for manual partitioning–the boundaries between behavioral groups are not trivially separable through visual inspection alone. This observation motivates the need for principled clustering algorithms that can discover latent performance patterns in high-dimensional metric space.

5.2.2 Function-based single-target regression models

This section evaluates function-based single-target models for performance prediction, conducting a comprehensive comparison of regression models to identify the best-performing approach.

We evaluate multiple regression algorithms using AutoGluon and aMLLibrary frameworks as described in Section 4.2.2. The complete list of methods and their hyperparameter configurations is provided in Table 3. Leveraging the feature engineering capabilities of aMLLibrary, we test DT, RF, SVR, XGB, Ridge Linear Regression, and Stepwise Regression in two configurations: a *base* setting using original features, and an *enhanced* setting incorporating interaction terms up to second degree. Feature selection employs XGBoost importance weights with 75% cumulative weight retention. All features undergo normalization and one-hot encoding for node type characterization. NNs are evaluated without explicit feature engineering, as they learn representations automatically.

Tables 4, 5 and 6 compare the prediction capabilities of the various models for the node-level CPU usage (CPU_n), the percentage of RAM usage ($RAM_n\%$), and the power consumption (Π_n) in terms of different metrics, which were computed both on the training and the test set[16]. The metrics in each table were computed considering the best-performing model trained for each method, automatically selected after hyperparameter tuning considering the MAPE computed on the validation set and repeating each experiment 5 times. Note that, due to how aMLLibrary is implemented, this is the only section where, while features are scaled and normalized during training, metrics are computed on the original unscaled data, which explains the significant difference in, e.g., the MSE when considering CPU_n, $RAM_n\%$

[16] The collected data were split considering 80% for training and 20% for test. Moreover, the training data were further split, leaving out the 20% for validation during the hyperparameter tuning process. Note that profiling experiments were repeated 3 times for each load configuration; the split was performed ensuring that all repetitions enter a single dataset.

Table 3 Regression models hyperparameters for tuning and hyperparameters selected for the best model (LightGBM Regressor)

Method	Hyperparameter	Values
Neural Network	num. of features	$\{[20, 10], [64, 64], [20, 10, 20]\}$
	dropout	0.2
	activation	ReLU
	optimizer	Adam
	learning rate	$\{0.01, 0.1, 0.05\}$
	loss	MSE
	batch size	$\{32, 64, 256\}$
	epochs	$\{5, 10, 100\}$
Decision Tree	max depth	$\{5, 10, 20\}$
	min samples split	0.01
	min samples leaf	0.01
Random Forest	num. of estimators	$\{5, 10, 20\}$
	max depth	$\{5, 10, 20\}$
	min samples split	0.01
	min samples leaf	0.01
Support Vector Regression	C	$\{0.001, 0.01, 0.1, 1\}$
	ε	0.05
	γ	$1e-7$
	kernel	linear
	degree	2
Lin. Regression (Ridge)	α	$\{0.01, 0.1, 0.5, 10\}$
Stepwise Regression (Draper&Smith)	probability to add	$\{0.05, 0.1, 0.25, 0.5, 0.75\}$
	probability to remove	$\{0.05, 0.1, 0.25, 0.5, 0.75\}$
	fit intercept	`true`
	max num. of iterations	$\{50, 100, 200\}$
XGBoost	min child weight	1
	γ	0
	num. of estimators	$\{20, 50, 100\}$
	learning rate	$\{0.01, 0.05, 0.001\}$
	max depth	$\{1, 5, 10\}$
	α	0
	λ	1
LightGBM Regressor	boosting type	Grad. Boosting Decision Tree
	max leaves	31
	max depth	no limit
	learning rate	0.1
	num. of estimators	100
	num. of samples for constructing bins	$2e5$
	min split gain	0
	min child weight	$1e-3$
	min child samples	20
	subsample ratio	1
	subsample frequency	0
	$L1$ regularization α	0.
	$L2$ regularization λ	0.

Table 4 CPU consumption (CPU_n) prediction results for different metrics and regression models

Method	Feature Augmentation and Selection	Metrics on Training					Metrics on Test				
		MAPE	*RMSE*	R^2	*MAE*	*MSE*	*MAPE*	*RMSE*	R^2	*MAE*	*MSE*
Neural Network	no	3.55%	10.26	0.98	7.11	105.26	3.53%	10.22	0.98	7.07	104.37
Decision Tree	no	7.62%	21.02	0.92	14.40	442.00	7.69%	21.09	0.92	14.49	444.87
Random Forest	no	8.62%	23.27	0.91	16.18	541.67	8.72%	23.55	0.91	16.36	554.73
Support Vector Regression	no	5.71%	17.18	0.95	11.48	295.04	5.66%	16.95	0.95	11.35	287.47
Linear Regression (Ridge)	no	6.26%	16.73	0.95	11.87	280.00	6.19%	16.49	0.95	11.71	272.02
Stepwise Regression (Draper&Smith)	no	6.26%	16.73	0.95	11.87	280.00	6.19%	16.49	0.95	11.71	272.02
XGBoost	no	2.64%	6.85	0.99	4.98	46.96	2.80%	7.41	0.99	5.34	54.98
Decision Tree	yes	7.53%	20.59	0.93	14.10	423.77	7.65%	20.84	0.93	14.34	434.24
Random Forest	yes	8.14%	22.67	0.91	15.50	514.09	8.21%	22.92	0.91	15.65	525.12
Support Vector Regression	yes	4.31%	12.88	0.97	8.74	166.02	4.31%	12.78	0.97	8.71	163.43
Linear Regression (Ridge)	yes	4.49%	12.77	0.97	8.90	163.03	4.50%	12.69	0.97	8.88	160.94
Stepwise Regression (Draper&Smith)	yes	3.80%	10.16	0.98	7.31	103.14	3.80%	10.15	0.98	7.32	102.93
XGBoost	yes	2.57%	6.45	0.99	4.74	41.61	2.74%	7.03	0.99	5.12	49.37
LightGBM Regressor	no	3.51%	9.62	0.98	6.86	92.59	3.57%	9.84	0.98	6.98	96.84

Table 5 RAM percentage consumption ($RAM_n\%$) prediction results for different metrics and regression models

Method	Feature Augmentation and Selection	Metrics on Training					Metrics on Test				
		MAPE	*RMSE*	R^2	*MAE*	*MSE*	*MAPE*	*RMSE*	R^2	*MAE*	*MSE*
Neural Network	no	2.28%	1.01	0.99	0.66	1.03	2.28%	1.02	0.99	0.65	1.04
Decision Tree	no	3.14%	1.54	0.97	0.91	2.39	3.12%	1.54	0.97	0.90	2.37
Random Forest	no	3.37%	1.63	0.97	0.98	2.66	3.35%	1.62	0.97	0.97	2.64
Support Vector Regression	no	6.48%	2.65	0.92	1.77	7.05	6.46%	2.66	0.92	1.77	7.06
Linear Regression (Ridge)	no	6.91%	2.58	0.93	1.86	6.64	6.90%	2.58	0.93	1.86	6.67
Stepwise Regression (Draper&Smith)	no	6.91%	2.58	0.93	1.86	6.64	6.90%	2.58	0.93	1.86	6.67
XGBoost	no	0.96%	0.50	1.0	0.28	0.25	0.98%	0.51	1.0	0.28	0.26
Decision Tree	yes	3.10%	1.54	0.97	0.89	2.36	3.09%	1.54	0.97	0.89	2.38
Random Forest	yes	3.28%	1.60	0.97	0.95	2.56	3.28%	1.60	0.97	0.94	2.55
Support Vector Regression	yes	5.99%	2.53	0.93	1.65	6.38	5.96%	2.52	0.93	1.65	6.34
Linear Regression (Ridge)	yes	6.44%	2.44	0.94	1.75	5.98	6.40%	2.44	0.94	1.75	5.94
Stepwise Regression (Draper&Smith)	yes	4.82%	1.98	0.96	1.33	3.92	4.76%	1.96	0.96	1.32	3.85
XGBoost	yes	0.96%	0.51	1.0	0.28	0.26	0.98%	0.52	1.0	0.28	0.27
LightGBM Regressor	no	1.60%	0.80	0.99	0.47	0.64	1.62%	0.81	0.99	0.47	0.66

and Π_n; in all the following sections, the reported metrics are computed over data scaled applying MinMax normalization as described in Section 4.2.1.

Based on these regression results, we identify LGBM [39] as the most efficient model. Its gradient boosting approach with decision tree ensembles delivers high prediction accuracy with reasonable computational overhead. Additionally, LGBM supports both standard and quantile regression modes, the latter being less readily available in alternatives like XGBoost.

The average training time of models presented in this section ranges between 1.6 hours for general ML regression models to almost 27 hours for NNs, further confirming the advantages of LGBM.

5.2.3 Function-based multi-target regression

We investigate the capability of Janossy-inspired neural networks (JNNs) to simultaneously predict multiple performance metrics at both node and function levels. This multi-target approach addresses the fundamental challenge of joint optimization across different granularities of system performance.

Our JNN implementation, detailed in Section 4.2.2 and illustrated in Figure 5, employs a hierarchical architecture designed to capture both sequential dependencies and cross-metric relationships. The network processes input sequences through an initial fully-connected layer with 64 neurons, followed by three stacked GRU layers with 80 hidden units each. The final prediction head consists of a

Table 6 Power consumption (Π_n) prediction results for different metrics and regression models

Method	Feature Augmentation and Selection	Metrics on Training					Metrics on Test				
		MAPE	*RMSE*	R^2	*MAE*	*MSE*	*MAPE*	*RMSE*	R^2	*MAE*	*MSE*
Neural Network	no	10.91%	1.36*e*5	0.95	1.01*e*5	1.84*e*10	11.11%	1.39*e*5	0.94	1.03*e*5	1.93*e*10
Decision Tree	no	13.97%	1.92*e*5	0.89	1.37*e*5	3.70*e*10	14.06%	1.95*e*5	0.89	1.38*e*5	3.80*e*10
Random Forest	no	14.64%	2.04*e*5	0.88	1.44*e*5	4.18*e*10	14.69%	2.07*e*5	0.87	1.45*e*5	4.30*e*10
Support Vector Regression	no	12.74%	1.68*e*5	0.92	1.23*e*5	2.83*e*10	12.80%	1.70*e*5	0.91	1.24*e*5	2.90*e*10
Linear Regression (Ridge)	no	13.30%	1.66*e*5	0.92	1.25*e*5	2.75*e*10	13.35%	1.68*e*5	0.92	1.26*e*5	2.83*e*10
Stepwise Regression (Draper&Smith)	no	13.30%	1.66*e*5	0.92	1.25*e*5	2.75*e*10	13.35%	1.68*e*5	0.92	1.26*e*5	2.83*e*10
XGBoost	no	6.75%	9.15*e*4	0.98	6.63*e*4	8.38*e*9	7.15%	9.98*e*4	0.97	7.13*e*4	9.96*e*9
Decision Tree	yes	13.64%	1.87*e*5	0.90	1.33*e*5	3.51*e*10	13.83%	1.91*e*5	0.89	1.35*e*5	3.65*e*10
Random Forest	yes	14.45%	2.03*e*5	0.88	1.42*e*5	4.12*e*10	14.57%	2.06*e*5	0.87	1.44*e*5	4.26*e*10
Support Vector Regression	yes	10.65%	1.52*e*5	0.93	1.10*e*5	2.30*e*10	10.75%	1.55*e*5	0.93	1.11*e*5	2.40*e*10
Linear Regression (Ridge)	yes	11.12%	1.50*e*5	0.93	1.12*e*5	2.26*e*10	11.23%	1.54*e*5	0.93	1.13*e*5	2.36*e*10
Stepwise Regression (Draper&Smith)	yes	10.16%	1.41*e*5	0.94	1.04*e*5	1.99*e*10	10.32%	1.45*e*5	0.94	1.06*e*5	2.10*e*10
XGBoost	yes	6.64%	8.74*e*4	0.98	6.40*e*4	7.64*e*9	7.08%	9.63*e*4	0.97	6.94*e*4	9.28*e*10
LightGBM Regressor	no	8.83%	1.23*e*5	0.96	8.77*e*4	1.51*e*10	8.91%	1.24*e*5	0.95	8.90*e*4	1.55*e*10

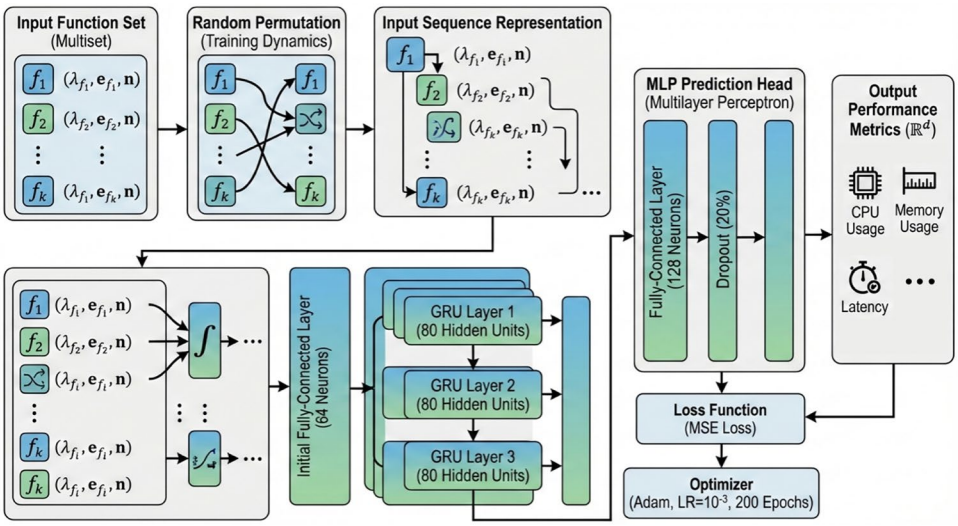


Fig. 5 Architecture of our JNN model

Table 7 Janossy-inspired neural network prediction results

Metric	MAPE	RMSE	R^2	MAE	MSE
CPU_n	5.93%	0.0249	0.98	0.0178	0.0006
RAM_n	5.42%	0.0322	0.94	0.0229	0.0010
Π_n	12.12%	0.0334	0.95	0.0243	0.0011

multi-layer perceptron with a 128-neuron fully-connected layer and 20% dropout regularization to prevent overfitting. We optimize the network using the Adam optimizer with a learning rate of 10^{-3} over 200 training epochs, minimizing the MSE loss across all target variables. All hyperparameters were selected through systematic manual tuning based on validation performance.

Table 7 presents the prediction performance across three key node-level metrics: CPU utilization (CPU_n), memory consumption (RAM_n), and power consumption (Π_n). The results demonstrate strong predictive accuracy, with R^2 scores ranging from 0.94 to 0.98 and MAPE values below 13% across all metrics.

Notably, CPU utilization prediction achieves the highest accuracy (MAPE = 5.93%, R^2 = 0.98), likely due to its more predictable temporal patterns compared to memory and power consumption. The slightly higher prediction error for power consumption (MAPE = 12.12%) reflects the inherent complexity and non-linear dependencies in power modeling.

These results closely match the performance of specialized single-target regression models reported in Table 8, validating the effectiveness of our multi-target approach. The JNN successfully captures the joint distribution of performance metrics without significant degradation compared to individual predictors, suggesting that the shared

Table 8 Comparison between cluster-based and function-based models for node-level metrics

	Metric	Model	MAPE (avg.)	MAPE (std.)	$P > R$	$P \leq R$
Classes	CPU_n	std regr.	1.01%	0.02%	-	-
		0.95-QR	1.01%	0.02%	94.45%	5.55%
		0.05-QR	2.07%	0.04%	5.97%	94.03%
	RAM_n	std regr.	4.15%	0.08%	-	-
		0.95-QR	7.97%	0.13%	93.97%	6.03%
		0.05-QR	7.63%	0.08%	5.28%	94.72%
Functions	CPU_n	std regr.	0.90%	0.02%	-	-
		0.95-QR	5.08%	0.15%	97.09%	2.91%
		0.05-QR	3.13%	0.05%	1.27%	98.73%
	RAM_n	std regr.	0.70%	0.02%	-	-
		0.95-QR	6.82%	0.43%	96.89%	3.11%
		0.05-QR	3.33%	0.10%	1.24%	98.76%

representations learned by the network effectively encode the underlying system dynamics. The average training time of our JNN model is of around 24 minutes.

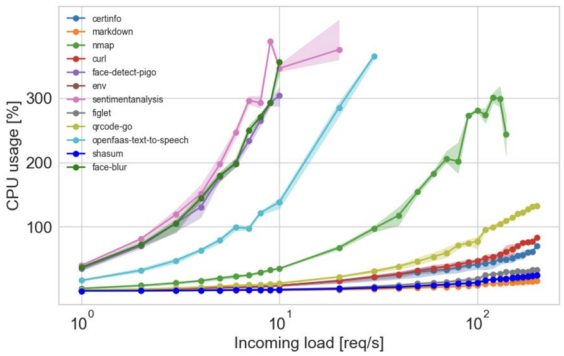
5.2.4 Cluster cardinality selection

This section describes the methodology adopted to determine the appropriate number of clusters k used to group functions based on their performance profiles.

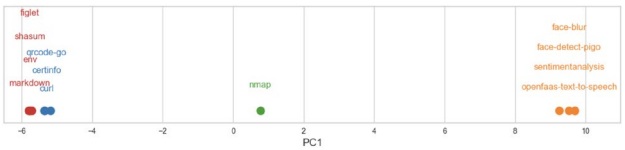
As described in Section 4.2.1, we first applied PCA to the profiling dataset collected for all the 12 functions listed in Table 2 across the three node types. The results showed that the first principal component alone explains more than 85% of the total variance in the dataset. Therefore, considering that using a single principal component enables a simple and easily interpretable representation of the performance space and since it already captures the vast majority of variability, we decided to retain only the first component for clustering.

To determine the appropriate cluster cardinality, we varied the number of clusters k between 1 and 10. Since the dataset contains 12 functions, exploring this range allows us to evaluate progressively finer partitions without trivially assigning each function to its own cluster. For each value of k, we applied the K-Means algorithm and computed the average distortion, defined as the mean Euclidean distance between each data point and the centroid of its assigned cluster, i.e., $d_k = \frac{1}{N}\sum_{i=1}^{N} \min_{c \in C_k} \|x_i - c\|_2$, where C_k is the set of centroids for a given k.

We observed values of d_k equal to [6.49, 0.97, 0.18, 0.08, 0.05, 0.03, 0.01, 0.007, 0.003] for increasing values of k: since the reduction between $k = 3$ and $k = 4$ remains significant while subsequent decreases become progressively marginal, we finally selected $k = 4$ as the optimal number of clusters in this context.



(a) CPU usage of the 12 functions



(b) Clusters

Fig. 6 CPU usage pattern of the 12 functions (chosen as a representative metric) and clusters obtained with $k = 4$

Figure 6 reports, as a representative metric, the CPU consumption pattern of the 12 functions executed on the three different nodes for increasing workload rates, together with the clusters obtained by following the procedure described above and choosing $k = 4$. We can observe that functions can be grouped in a *low-usage* (*lu*) cluster including `env`, `figlet`, `markdown` and `shasum`, a *medium-usage* (*mu*) cluster including `certinfo`, `curl` and `qrcode-go`, a *high-usage* (*hu*) cluster including `nmap`, and a *very-high-usage* (*vhu*) cluster including `face-blur`, `face-detect-pigo`, `openfaas-text-to-speech`, and `sentimentanalysis`. This is coherent with the fact that, as can be observed from Figure 6a, the functions in the *vhu* group could only be profiled for load rates up to 30*req/s* before even the *heavy* node reached an overloaded state.

Considering the optimal value $k = 4$, we compare in Table 8 the prediction capabilities of LGBM single-target regressors trained to estimate node-level performance (CPU_n and RAM_n) considering a per-function and per-cluster approach, evaluated in terms of MAPE and, for quantile regression models, percentage of observation above/below the predicted value. In terms of point prediction accuracy (i.e., for standard regression), the per-function models consistently achieve lower MAPE values than the cluster-based models. This improvement is particularly evident for RAM_n, where the MAPE decreases from 4.15% to 0.70%, but it is also noticeable for CPU_n (from 1.01% to 0.90%). This behavior aligns with the fact that using per-function incoming load as features provides a finer-grained representation of the workload composition, allowing the model to capture function-specific resource consumption patterns more precisely. In contrast, aggregating load at the cluster level inevitably introduces a degree of information loss, as heterogeneous behaviors within the same cluster are represented through a single aggregated feature.

However, analyzing the quantile regression models we can observe that, for the cluster-based approach, the empirical proportions of observations above/below the predicted quantiles ($P > R$ and $P \leq R$) are very close to the nominal confidence levels (e.g., around 94–95% for the 0.95 quantile and around 4–5% for the 0.05 quantile). In contrast, the per-function models exhibit more pronounced deviations from the expected coverage, particularly for CPU_n, where the 0.95-QR model overestimates the upper tail (97.09%) and the 0.05-QR model underestimates it (1.27% above prediction).

This suggests that, while per-function models are more accurate in terms of average error, cluster-based models provide better-calibrated uncertainty estimates. A plausible explanation is that aggregating functions into clusters smooths out function-specific irregularities and high-frequency variability in resource consumption. By reducing dimensionality and intra-class noise, the per-cluster representation may yield more stable conditional distributions, which in turn improves quantile calibration. Conversely, the higher granularity of the per-function feature space, while beneficial for point prediction, may lead to sharper but less well-calibrated conditional estimates, especially in the tails.

Overall, these results indicate a trade-off between fine-grained accuracy and robustness of uncertainty estimation. The per-function approach is preferable when minimizing average prediction error is the primary objective, whereas the cluster-based approach appears more suitable when reliable quantile estimates and well-calibrated prediction intervals are required.

5.2.5 Cluster-based regression and classification

This section reports a detailed evaluation of the prediction capabilities of cluster-based single-target LGBM regression and classification models. In particular, we considered both standard and quantile regression; the former is evaluated by reporting the values of MSE, RMSE, standard deviation, R^2 , while for quantile-based models we considered the quantile loss $\mathcal{L}_Q$ and the percentage of *predicted* values (*P*) that exceed the *observed* values (*R*) in the test set (or vice versa). All metrics are computed when predicting both node-level and cluster-level performance metrics (e.g., CPU_n and the CPU consumption related to functions in the very-high-usage – CPU_{vhu} –, high-usage – CPU_{hu} –, medium-usage – CPU_{mu} – and low-usage – CPU_{lu} – function clusters). In particular, each experiment is repeated 5 times; average metric values are reported in Tables 9 and 10 (in bold to improve readability) together with their standard deviation (within round brackets).

For what concerns classification performance for predicting the node overload state Σ_n, we evaluated the accuracy in terms of precision, recall, and F1-score.

For CPU-related metrics, both at node and cluster level, the standard regression models achieve consistently high accuracy. In particular, CPU_n is predicted with very low MAPE (1.01%) and high R^2 (0.97), confirming that node-level CPU utilization can be effectively captured through the aggregated cluster features. At the cluster level, prediction quality remains strong for CPU_{hu} and CPU_{lu}, while slightly lower R^2 values for CPU_{vhu} and CPU_{mu} reflect the higher variability typically associated with very-high and medium usage groups. Nevertheless, the relatively small standard deviations across the five repetitions indicate stable and robust model behavior.

When considering quantile regression for CPU, the empirical coverage probabilities are generally well aligned with the target quantiles, especially at node level (e.g., 94.45% for the 0.95-QR of CPU_n). For cluster-level metrics, predictions tend to be slightly conservative, particularly for the very-high-usage class (e.g., 87.10% for the 0.95-QR of CPU_{vhu}). This conservative behavior is consistent with the greater dispersion and burstiness of workloads in this group, which makes tail estimation more challenging. Still, the quantile loss $\mathcal{L}_Q$ remains low and stable, indicating that the models provide reliable uncertainty estimates even under higher variability.

RAM-related metrics confirm that memory consumption is inherently more difficult to predict. Compared to CPU, RAM exhibits lower R^2 values at both node and cluster level, especially for RAM_{mu} and RAM_{lu}, where R^2 drops to around 0.33. This degradation is likely due to the less direct and more heterogeneous relationship between

Table 9 Prediction results considering different metrics (standard regression). For each performance metric (e.g., MAPE), we report the average value over 5 experiments and its standard dev

	Metric	MAPE (std)	MSE (std)	RMSE (std)	stddev. (std)	$\mathbf{R^2}$ (std)
functionclass	CPU_{vhu}	**5.06**% (1.09%)	**0.0114** (0.0047)	**0.1051** (0.0205)	**0.2230** (0.0125)	**0.81** (0.077)
	CPU_{hu}	**1.21**% (0.08%)	**0.0009** (0.0001)	**0.0302** (0.0024)	**0.2398** (0.0028)	**0.98** (0.003)
	CPU_{mu}	**3.88**% (0.05%)	**0.0047** (0.0001)	**0.0687** (0.0010)	**0.1344** (0.0016)	**0.79** (0.005)
	CPU_{lu}	**3.44**% (0.06%)	**0.0039** (0.0001)	**0.0626** (0.0006)	**0.1578** (0.0010)	**0.86** (0.005)
node	CPU_n	**1.01**% (0.02%)	**0.0005** (0.0001)	**0.0213** (0.0015)	**0.1258** (0.0023)	**0.97** (0.004)
functionclass	RAM_{vhu}	**3.36**% (0.81%)	**0.0067** (0.0023)	**0.0805** (0.0148)	**0.1147** (0.0153)	**0.63** (0.099)
	RAM_{hu}	**3.96**% (0.29%)	**0.0091** (0.0012)	**0.0952** (0.0066)	**0.0969** (0.0071)	**0.52** (0.050)
	RAM_{mu}	**3.93**% (0.03%)	**0.0046** (0.0003)	**0.0679** (0.0021)	**0.0472** (0.0009)	**0.33** (0.019)
	RAM_{lu}	**8.95**% (0.11%)	**0.0206** (0.0006)	**0.1435** (0.0022)	**0.1017** (0.0014)	**0.33** (0.009)
node	RAM_n	**4.15**% (0.08%)	**0.0068** (0.0003)	**0.0823** (0.0020)	**0.1021** (0.0009)	**0.61** (0.015)
functionclass	L_{vhu}	**6.54**% (1.25%)	**0.0217** (0.0066)	**0.1458** (0.0241)	**0.3250** (0.0286)	**0.83** (0.050)
	L_{hu}	**1.59**% (0.25%)	**0.0032** (0.0013)	**0.0553** (0.0109)	**0.1143** (0.0071)	**0.83** (0.034)
	L_{mu}	**0.80**% (0.01%)	**0.0004** (0.0002)	**0.0182** (0.0051)	**0.0087** (0.0002)	**0.17** (0.100)
	L_{lu}	**1.86**% (0.02%)	**0.0009** (0.0001)	**0.0305** (0.0016)	**0.0304** (0.0006)	**0.49** (0.022)

Table 10 Prediction results considering different metrics (quantile regression). For each performance metric (e.g., MAPE), we report the average value over 5 experiments and its standard dev

	Metric	Model	MAPE (std)	$\mathcal{L}_{\mathbf{Q}}$ (std)	$P > R$	$P \leq R$
function class	CPU_{vhu}	0.95-QR	**7.26**% (0.92%)	**0.0062** (0.0007)	87.10%	12.90%
		0.05-QR	**10.84**% (0.88%)	**0.0137** (0.0101)	6.45%	93.55%
	CPU_{hu}	0.95-QR	**2.93**% (0.32%)	**0.0029** (0.0002)	90.86%	9.14%
		0.05-QR	**2.74**% (0.16%)	**0.0030** (0.0003)	6.58%	93.42%
	CPU_{mu}	0.95-QR	**3.88**% (0.05%)	**0.0047** (0.0001)	94.64%	5.36%
		0.05-QR	**4.76**% (0.10%)	**0.0036** (0.00004)	6.47%	93.53%
	CPU_{lu}	0.95-QR	**3.44**% (0.06%)	**0.0039** (0.0001)	94.10%	5.90%
		0.05-QR	**6.07**% (0.06%)	**0.0044** (0.0001)	6.53%	93.47%
node	CPU_n	0.95-QR	**1.01**% (0.02%)	**0.0005** (0.0001)	94.45%	5.55%
		0.05-QR	**2.07**% (0.04%)	**0.0016** (0.0001)	5.97%	94.03%
function class	RAM_{vhu}	0.95-QR	**7.95**% (2.59%)	**0.0057** (0.0022)	93.55%	6.45%
		0.05-QR	**2.95**% (0.53%)	**0.0020** (0.0003)	14.52%	85.48%
	RAM_{hu}	0.95-QR	**8.83**% (0.80%)	**0.0071** (0.0008)	93.78%	6.22%
		0.05-QR	**4.03**% (0.33%)	**0.0028** (0.0002)	8.96%	91.04%
	RAM_{mu}	0.95-QR	**10.72**% (0.31%)	**0.0079** (0.0003)	94.07%	5.93%
		0.05-QR	**4.80**% (0.13%)	**0.0029** (0.0001)	6.11%	93.89%
	RAM_{lu}	0.95-QR	**23.48**% (0.29%)	**0.0164** (0.0002)	93.15%	6.85%
		0.05-QR	**12.52**% (0.10%)	**0.0093** (0.0001)	5.74%	94.26%
node	RAM_n	0.95-QR	**7.97**% (0.13%)	**0.0050** (0.0001)	93.97%	6.03%
		0.05-QR	**7.63**% (0.08%)	**0.0054** (0.0001)	5.28%	94.72%
function class	L_{vhu}	0.95-QR	**14.78**% (3.58%)	**0.0105** (0.0036)	93.55%	6.45%
		0.05-QR	**5.74**% (1.12%)	**0.0046** (0.0013)	11.29%	88.71%
	L_{hu}	0.95-QR	**4.34**% (0.44%)	**0.0048** (0.0018)	94.88%	5.12%
		0.05-QR	**2.14**% (0.33%)	**0.0016** (0.0003)	5.67%	94.33%
	L_{mu}	0.95-QR	**1.55**% (0.04%)	**0.0015** (0.0003)	93.80%	6.20%
		0.05-QR	**0.69**% (0.02%)	**0.0004** (0.00002)	6.35%	93.65%
	L_{lu}	0.95-QR	**4.03**% (0.04%)	**0.0029** (0.0001)	94.02%	5.98%
		0.05-QR	**3.09**% (0.03%)	**0.0019** (0.00002)	6.72%	93.28%

incoming load and memory usage, which may depend on implementation details and internal buffering mechanisms. Nonetheless, MAPE values remain within acceptable ranges, and the low standard deviations across repetitions suggest consistent training behavior.

From the quantile perspective, RAM predictions at node level remain well calibrated, with empirical coverage close to the nominal quantile. At cluster level, the estimates are again slightly conservative, particularly for the low-usage group (RAM_{lu}), where larger MAPE and quantile errors reflect the higher relative impact of small absolute deviations. Interestingly, although R^2 degrades for RAM, the quantile loss $\mathcal{L}_Q$ stays reasonably close to CPU values, indicating that while point prediction accuracy is affected by variability, the conditional distribution learned by the model still captures the uncertainty structure effectively.

Overall, these results confirm that the cluster-based representation provides accurate and stable predictions for CPU metrics and reasonably robust performance for RAM and load-related metrics. Moreover, quantile regression models demonstrate good calibration properties, particularly at node level, supporting their suitability for resource-aware orchestration scenarios where reliable upper and lower bounds are as important as accurate point estimates.

Latency predictions exhibit a behavior that is partially similar to CPU, but with some distinctive characteristics. From Table 9, cluster-based standard regression achieves good accuracy overall, particularly for the high-usage (L_{hu}) and very-high-usage (L_{vhu}) classes, where R^2 reaches 0.83. This indicates that, in these regimes, latency scales in a relatively predictable manner with the aggregated workload of the corresponding function group. The medium-usage class (L_{mu}), despite showing a very low MAPE (0.80%), is associated with a much lower R^2 (0.17). This apparent discrepancy can be explained by the limited dynamic range of latency values in this cluster: when variability is intrinsically small, even modest absolute deviations may significantly affect the explained variance, while percentage errors remain small. For the low-usage group (L_{lu}), performance is intermediate, with moderate R^2 (0.49), reflecting a less deterministic relationship between load and response time in lightly loaded regimes.

Quantile regression results for latency further confirm the overall robustness of the approach. For most clusters, the empirical coverage of the 0.95 and 0.05 quantiles remains close to the nominal values, indicating good calibration of prediction intervals. Slight conservativeness is again observed in some cases (e.g., L_{vhu} for the 0.95-QR), which is consistent with the higher dispersion and occasional tail events typical of latency under heavy load. Importantly, the quantile loss $\mathcal{L}_Q$ remains low and stable across repetitions, suggesting that the model effectively captures the conditional distribution of latency, even in the presence of nonlinear effects and saturation phenomena.

Overall, latency prediction confirms the suitability of the cluster-based modeling approach: while variability and nonlinear queueing effects naturally make latency more complex than CPU, the models achieve stable performance, good calibration of uncertainty bounds, and satisfactory explanatory power in the most critical (high-load) regimes.

In terms of classification, as reported in Table 11, LGBM demonstrates high accuracy in distinguishing overloaded from non-overloaded conditions. Notably, it performs marginally better in detecting overloads, as reflected in the slightly higher recall for the overloaded class. This improved sensitivity to overloaded states reduces the risk of unaddressed performance issues, which is crucial to mitigate service disruptions. At the same time, the model balances this sensitivity with low false-positive rates, preventing unnecessary alarms in operational settings where maintaining system stability and availability is essential.

The average training time of all models discussed in this section is of around 2 seconds.

Table 11 Node state prediction results

Σ_n	Precision	Recall	$f1$-score	Number of instances
0 (not overloaded)	0.99	0.96	0.97	22654
1 (overloaded)	0.96	0.99	0.97	22716
Overall model accuracy: 97% over 45370 test instances				

5.2.6 Performance of the profile predictor

In this section, we discuss the results of experiments performed to validate the capability of our Profile Predictor module in correctly assigning an unseen function to one between the *low-usage*, *medium-usage* or *very-high-usage* class. In particular, we have followed a *leave-one-out* validation approach where: (i) the Function Profiler is trained to identify four clusters (i.e., the same *vhu*, *hu*, *mu*, and *lu* clusters discussed in Section 5.2.4) after having completely removed one function from the training data; (ii) that function is tentatively assigned to one of the classes by the Profile Predictor. Note that we have performed such an experiment by iteratively removing all functions except `nmap`, as this is the only function assigned to the *hu* class.

In general, we have observed that the Profile Predictor manages to assign to the correct class all left-out functions except one (`shasum`), achieving a 92% accuracy. We have reported two examples of function classification in Figure 7, considering `markdown` (i.e., an example of *low-usage* function), `curl` (for the *medium-usage* group) and `openfaas-text-to-speech` (*very-high-usage* function). In all cases, the assignment to a class is coherent with what was observed in Section 5.2.4.

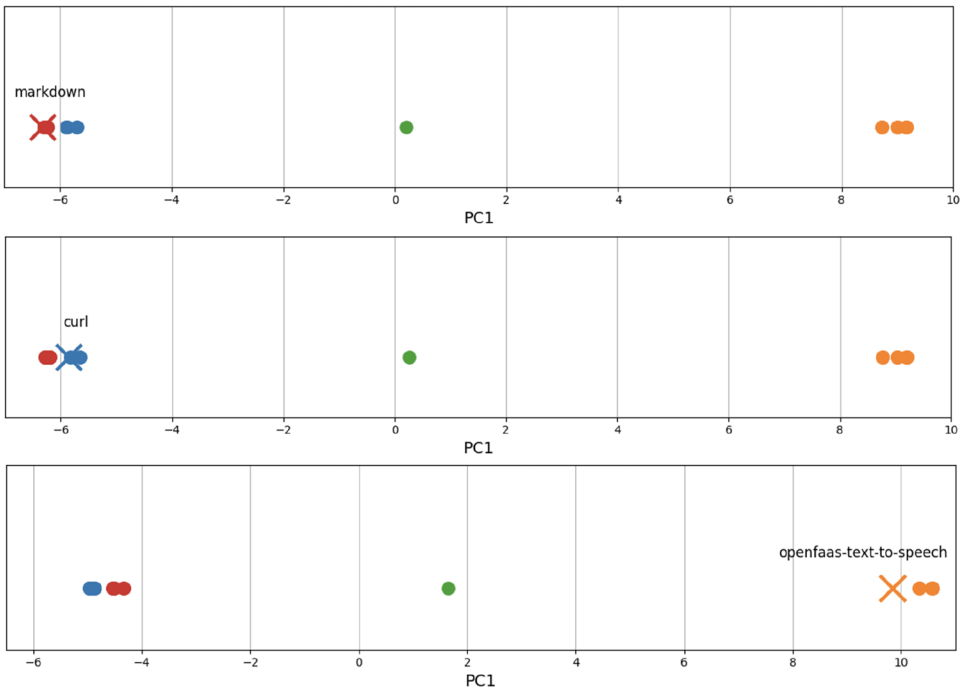


Fig. 7 Function classification into the *lu* (top), *mu* (center) or *vhu* (bottom) class using the Profile Predictor

5.2.7 Generalization properties of cluster-based regression

To evaluate the generalization capability of the proposed cluster-based performance modeling approach, we performed a leave-one-out validation at the function level within each cluster. In particular, for each function belonging to the *low-usage*, *medium-usage*, and *very-high-usage* groups (as mentioned in the previous section, the *high-usage* group contains only one function and is therefore excluded), we trained the corresponding group-level regression models after removing that function from the training set. The trained model was then used to predict the function-level metrics (CPU_f, RAM_f, and L_f) of the left-out function. Each experiment was repeated five times, and for each function we computed the average and standard deviation of MAPE and quantile loss $\mathcal{L}_Q$. Tables 12–14 report, for each group, the minimum, maximum, and average values of these statistics across all left-out functions.

The rationale behind this experiment is twofold. First, if the function profile and the resulting clustering capture the intrinsic resource-consumption behavior of functions, then a model trained on the remaining functions of the same group should be able to predict the performance of a previously unseen function in that group with acceptable accuracy. Second, combined with the results in Section 5.2.6, which show that the Profile Predictor correctly classifies unseen functions with 92% accuracy, this suggests a practical deployment strategy: when a new function is deployed, only a limited amount of profiling data is required to assign it to a usage class; subsequently, the pre-trained group model can be used to estimate its performance until sufficient data are collected to optionally train a function-specific model.

The results confirm that such a strategy achieves a reasonable level of accuracy. For the *low-usage* group (Table 12), standard regression yields an average MAPE of 7.76% for CPU_f (with values ranging from 3.69% to 12.33%) and 12.87% for RAM_f, while latency prediction remains particularly accurate, with an average MAPE of 4.27%. Quantile regression models exhibit moderate increases in MAPE, as expected, but maintain stable and relatively low quantile loss values, indicating consistent estimation of upper and lower bounds. Although memory quantiles show higher MAPE (e.g., 31.84% for the 0.95-QR on RAM_f), this behavior reflects the higher intrinsic variability of memory consumption rather than instability of the model, as also observed in previous sections.

For the *medium-usage* group (Table 13), generalization performance improves further. Average MAPE values are 7.25% for CPU_f, 5.05% for RAM_f, and only 1.47% for L_f under standard regression, with very small standard deviations across repetitions. Quantile models remain stable, with moderate $\mathcal{L}_Q$ values and controlled dispersion across left-out functions. These results indicate that medium-usage functions exhibit sufficiently homogeneous behavior within the cluster, enabling the group-level model to transfer effectively to unseen members.

The *very-high-usage* group (Table 14) presents the most challenging scenario, as expected due to the higher variability and burstiness of resource consumption. Here, the average MAPE for standard regression reaches 9.85% for CPU_f

Table 12 Minimum, maximum and average value of MAPE and $\mathcal{L}_Q$ when using low-usage models to predict function-level metrics

Metric	Model		MAPE (avg.)	MAPE (std.)	$\mathcal{L}_\mathbf{Q}$ (avg.)	$\mathcal{L}_\mathbf{Q}$ (std.)
CPU_f	std reg.	min	3.69%	0.03%	-	-
		max	12.33%	0.06%	-	-
		avg	7.76%	0.06%	-	-
	0.95-QR	min	5.33%	0.02%	0.0071	$4.68e-05$
		max	19.40%	0.08%	0.0524	0.0002
		avg	12.23%	0.06%	0.0235	0.0001
	0.05-QR	min	4.43%	0.02%	0.0038	$2.20e-05$
		max	12.46%	0.04%	0.0747	0.0004
		avg	7.96%	0.03%	0.0292	0.0002
RAM_f	std reg.	min	9.85%	0.04%	-	-
		max	18.72%	0.13%	-	-
		avg	12.87%	0.09%	-	-
	0.95-QR	min	29.53%	0.22%	0.0185	$2.52e-04$
		max	33.03%	0.50%	0.0668	0.0027
		avg	31.84%	0.38%	0.0348	0.0011
	0.05-QR	min	8.94%	0.01%	0.0083	$3.62e-05$
		max	23.82%	0.05%	0.0210	0.0005
		avg	14.15%	0.03%	0.0127	0.0002
L_f	std reg.	min	3.14%	0.01%	-	-
		max	12.33%	0.06%	-	-
		avg	4.27%	0.01%	-	-
	0.95-QR	min	2.72%	0.02%	0.0046	$1.61e-05$
		max	10.48%	0.04%	0.0144	0.0001
		avg	6.88%	0.04%	0.0081	0.0000
	0.05-QR	min	2.39%	0.01%	0.0033	$1.41e-05$
		max	5.48%	0.03%	0.0283	0.0002
		avg	3.61%	0.02%	0.0120	0.0001

, 5.36% for RAM_f, and 11.06% for L_f, with larger variability across functions (e.g., CPU MAPE up to 24.54%). Quantile models show correspondingly higher MAPE and $\mathcal{L}_Q$, especially for latency. Nevertheless, even in this most demanding regime, the errors remain within a range that is acceptable for capacity planning and resource provisioning decisions, particularly considering that no function-specific retraining is performed.

Overall, these results demonstrate that cluster-based regression models retain meaningful predictive power when applied to unseen functions within the same usage class, particularly considering that in practical capacity planning and resource management scenarios, prediction errors below 30% are generally considered acceptable [40], especially when models are used to support operational decisions rather than to provide exact point estimates. This allows us to conclude that function profiling and clustering effectively capture structural similarities in workload behavior, enabling performance estimation for newly deployed functions with a reasonable and practically useful level of accuracy.

We note that, while results indicate good generalization to unseen function compositions within known clusters, extrapolation to entirely new workload distributions or system configurations would likely require the adoption of more advanced strategies as, e.g., domain adaptation techniques; the investigation of such more complex scenarios is left for future work.

6 Conclusion

Accurate modeling of serverless function performance and computing node states is critical for optimizing operational efficiency, sustainability, and quality of service in edge-deployed FaaS infrastructures. Predicting resource consumption patterns enables proactive resource management and capacity planning strategies that are essential to maintain QoS at scale while avoiding overprovisioning and unnecessary energy consumption.

In this work, we presented a comprehensive framework integrating serverless function profiling, systematic

Table 13 Minimum, maximum and average value of MAPE and $\mathcal{L}_Q$ when using medium-usage models to predict function-level metrics

Metric	Model		MAPE (avg.)	MAPE (std.)	$\mathcal{L}_{\mathbf{Q}}$ (avg.)	$\mathcal{L}_{\mathbf{Q}}$ (std.)
CPU_f	std reg.	min	3.63%	0.03%	-	-
		max	12.11%	0.07%	-	-
		avg	7.25%	0.06%	-	-
	0.95-QR	min	7.39%	0.04%	0.0064	$5.13e-05$
		max	12.71%	0.11%	0.1100	0.0007
		avg	10.19%	0.08%	0.0413	0.0003
	0.05-QR	min	2.16%	0.01%	0.0027	$4.05e-05$
		max	16.29%	0.09%	0.0126	0.0003
		avg	7.48%	0.06%	0.0090	0.0001
RAM_f	std reg.	min	4.44%	0.02%	-	-
		max	6.19%	0.12%	-	-
		avg	5.05%	0.07%	-	-
	0.95-QR	min	6.91%	0.15%	0.0072	$7.63e-05$
		max	15.41%	0.21%	0.0310	0.0019
		avg	12.13%	0.17%	0.0155	0.0007
	0.05-QR	min	2.22%	0.01%	0.0017	$1.89e-05$
		max	6.86%	0.03%	0.0042	0.0001
		avg	4.02%	0.02%	0.0028	0.0001
L_f	std reg.	min	1.11%	0.02%	-	-
		max	1.66%	0.04%	-	-
		avg	1.47%	0.03%	-	-
	0.95-QR	min	2.00%	0.10%	0.0013	$5.22e-05$
		max	4.25%	0.17%	0.0153	0.0003
		avg	2.79%	0.14%	0.0066	0.0001
	0.05-QR	min	0.11%	0.00%	0.0002	$1.66e-07$
		max	1.87%	0.00%	0.0010	0.0001
		avg	0.73%	0.00%	0.0006	0.0000

performance metric collection, and machine learning–based forecasting at both function and node levels. Our approach combines (i) controlled workload generation to thoroughly cover the input space, (ii) dimensionality reduction and clustering to capture structural similarities among functions, and (iii) regression and classification models to predict resource consumption and node operational states.

Experimental validation demonstrates strong predictive performance, with models achieving 97% accuracy in node state prediction. This suggests that ML approaches can reliably capture serverless system dynamics, providing actionable insights for operational decision-making and enhancing the resilience of edge-deployed FaaS infrastructures. At the regression level, our best function-based models predict node-level CPU, RAM, and power consumption with an average MAPE below 9%, while quantile regression models provide well-calibrated uncertainty bounds; moreover, our experiments highlight the effectiveness of function profiling and cluster-based modeling. Our novel JNN-based multi-target regressor generalizes effectively across previously unseen workload compositions, with R^2 scores ranging from 0.94 to 0.98, i.e., aligned with the performance of specialized single-target regression models. Furthermore, by grouping functions according to their resource-consumption profiles, we obtain compact yet expressive representations that enable scalable performance prediction, with cluster-based regressors achieving accuracy levels comparable to function-specific models for CPU and latency while offering improved calibration properties for quantile predictions at the node level. Importantly, leave-one-function-out experiments demonstrate that cluster-level models retain meaningful predictive power when applied to previously unseen functions within the same usage class. Even in the most challenging scenarios, prediction errors remain within ranges that are generally considered acceptable in the performance modeling literature (i.e., below 30% for practical applications), with most results falling well below this threshold and frequently in the single-digit percentage range.

Overall, the proposed framework provides actionable insights for operational decision-making, enabling

Table 14 Minimum, maximum and average value of MAPE and $\mathcal{L}_Q$ when using very-high-usage models to predict function-level metrics

Metric	Model		MAPE (avg.)	MAPE (std.)	$\mathcal{L}_\mathbf{Q}$ (avg.)	$\mathcal{L}_\mathbf{Q}$ (std.)
CPU_f	std reg.	min	4.76%	0.32%	-	-
		max	24.54%	0.69%	-	-
		avg	9.85%	0.44%	-	-
	0.95-QR	min	10.32%	0.36%	0.0079	$2.47e-04$
		max	31.90%	0.86%	0.0188	0.0020
		avg	16.44%	0.62%	0.0118	0.0008
	0.05-QR	min	9.99%	0.66%	0.0155	$2.25e-03$
		max	18.54%	1.20%	0.1857	0.0119
		avg	13.38%	1.00%	0.0586	0.0053
RAM_f	std reg.	min	3.59%	0.22%	-	-
		max	7.94%	1.27%	-	-
		avg	5.36%	0.53%	-	-
	0.95-QR	min	7.71%	0.57%	0.0055	$2.99e-04$
		max	18.23%	2.12%	0.0263	0.0032
		avg	12.50%	1.10%	0.0142	0.0013
	0.05-QR	min	2.78%	0.05%	0.0045	$2.24e-05$
		max	8.84%	0.46%	0.0072	0.0041
		avg	4.65%	0.25%	0.0059	0.0020
L_f	std reg.	min	8.16%	0.46%	-	-
		max	13.06%	0.88%	-	-
		avg	11.06%	0.74%	-	-
	0.95-QR	min	9.93%	0.96%	0.0146	$5.21e-04$
		max	31.29%	5.60%	0.0345	0.0103
		avg	24.03%	3.31%	0.0203	0.0041
	0.05-QR	min	5.39%	0.08%	0.0083	$7.67e-05$
		max	22.95%	1.12%	0.0486	0.0046
		avg	10.89%	0.70%	0.0219	0.0030

predictive resource orchestration, improved resilience, and more sustainable operation of edge-deployed FaaS infrastructures

Future work should explore Bayesian optimization techniques to improve data collection efficiency by systematically identifying optimal load testing configurations. Additionally, more extensive validation encompassing complex multi-component applications and a more diverse set of functions, explicit inclusion of functions payload among the considered features, and direct comparisons with state-of-the-art approaches and non-ML baselines would strengthen the generalizability of our findings.

Acknowledgements This work was partially supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP E83C22004640001, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART")

Author contributions F.F, M.S. and M.C. contributed equally in writing and revising the main manuscript. F.F. also conducted the experimental campaign.

Funding Open access funding provided by Università degli Studi di Milano - Bicocca within the CRUI-CARE Agreement. This work was partially supported by the European Union - Next Generation EU under the Italian National Recovery and Resilience Plan (NRRP), Mission 4, Component 2, Investment 1.3, CUP E83C22004640001, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART")

Data availability The framework we introduce in this paper–together with the data used for the analysis–is publicly available on GitHub at https://github.com/unimib-datAI/dfaas/tree/main/metrics_predictions

Declarations

Competing interests The authors have no relevant financial or non-financial interests to disclose.

Ethical approval Not Applicable

References

1. Rajan, A.P.: A review on serverless architectures-function as a service (FaaS) in cloud computing. Telkomnika (Telecommunication Computing Electronics and Control) **18**(1), 530–537 (2020)
2. Adzic, G., Chatley, R.: Serverless computing: economic and architectural impact. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2017, pp. 884–889. Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3106237.3117767
3. Castro, P., Isahagian, V., Muthusamy, V., Slominski, A.: Hybrid serverless computing: Opportunities and challenges. In: Serverless Computing: Principles and Paradigms, 43–77. (2023)
4. Russo, G.R., Mannucci, T., Cardellini, V., Presti, F.L.: Serverledge: Decentralized function-as-a-service for the edge-cloud continuum. In: 2023 IEEE International Conference on Pervasive Computing and Communications (PerCom), 131–140 (2023). https://doi.org/10.1109/PERCOM56429.2023.10099372
5. Das, A., Patterson, S., Wittie, M.: Edgebench: Benchmarking edge computing platforms. In: IEEE UCC, 175–180. (2018)
6. De Palma, G., Giallorenzo, S., Mauro, J., Trentin, M., Zavattaro, G.: Funless: Functions-as-a-service for private edge cloud systems. In: 2024 IEEE International Conference on Web Services (ICWS), 961–967. IEEE (2024)
7. Ciavotta, M., Motterlini, D.: DFaaS: Decentralized Function-as-a-Service for Federated Edge Computing. In: IEEE CloudNet, 1–4. (2021)
8. Shafiei, H., Khonsari, A., Mousavi, P.: Serverless computing: a survey of opportunities, challenges, and applications. ACM Comput. Surv. **54**(11s), 1–32 (2022)
9. Ma, R., Zhan, Y.: Interless: Interference-Aware Deep Resource Prediction for Serverless Computing, 3783–3788. CCDC (2024)
10. Filippini, F., Calmi, N.: Analysis and Evaluation of Load Management Strategies in a Decentralized FaaS Environment: A Simulation-Based Framework. In: ACM SEATED Proceedings, 1–8. (2024)
11. Copik, M., Kwasniewski, G.: Sebs: a serverless benchmark suite for function-as-a-service computing. In: ACM Middleware Proceedings, 64–78. (2021)
12. Filippini, F., Cavenaghi, L., Calmi, N., Savi, M., Ciavotta, M.: ML-Based Performance Modeling in Edge FaaS Systems. In: Pahl, C., Janes, A., Cerny, T., Lenarduzzi, V., Esposito, M. (eds.) Service-Oriented and Cloud Computing, pp. 112–127. Springer, Cham (2025)
13. Calavaro, C., Cardellini, V., Lo Presti, F., Russo Russo, G.: Beyond cloud: Serverless functions in the compute continuum. SN Computer Science **6**(3), 194 (2025)
14. Eismann S, Costa DE, Liao L, Bezemer C-P, Shang W, van Hoorn A, Kounev S (2022) A case study on the stability of performance tests for serverless applications. J. Syst. Softw. 189:111294. https://doi.org/10.1016/j.jss.2022.111294
15. Khatri, D., Khatri, S.K., Mishra, D.: Performance validation of server-less computing using machine learning. In: 2024 IEEE International Conference on Blockchain and Distributed Systems Security (ICBDS), 1–6 (2024). IEEE
16. Liu, Q., Yang, Y., Du, D., Xia, Y., Zhang, P., Feng, J., Larus, J.R., Chen, H.: Harmonizing efficiency and practicability: Optimizing resource utilization in serverless computing with jiagu. In: 2024 USENIX Annual Technical Conference (USENIX ATC 24), pp. 1–17. USENIX Association, Santa Clara, CA (2024). https://www.usenix.org/conference/atc24/presentation/liu-qingyuan
17. Garbi, G., Incerto, E., Tribastone, M.: MP: A Development Framework for Predicting Performance of Microservices by Design, 178–188. IEEE CLOUD (2023)
18. Nawrocki, P., Osypanka, P., Posluszny, B.: Data-Driven Adaptive Prediction of Cloud Resource Usage. Journal of Grid Computing 21, (2023)
19. Grohmann, J., Straesser, M.: SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications. In: ACM/SPEC ICPE Proceedings, 165–176. (2021)
20. Masouros, D., Xydis, S., Soudris, D.: Rusty: Runtime Interference-Aware Predictive Monitoring for Modern Multi-Tenant Systems. IEEE Trans. Parallel Distrib. Syst. **32**(1), 184–198 (2021). https://doi.org/10.1109/TPDS.2020.3013948
21. Vahabi, S., Righetti, F., Vallati, C., Tonellotto, N.: The impact of prediction models on energy-aware resource management in faas platforms. IEEE Access **13**, 85711–85727 (2025). https://doi.org/10.1109/ACCESS.2025.3569068
22. Jindal, A., Chadha, M., Benedict, S., Gerndt, M.: Estimating the capacities of function-as-a-service functions. In: Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion, 1–8. (2021)
23. Jegannathan, A.P., Saha, R., Addya, S.K.: A Time Series Forecasting Approach to Minimize Cold Start Time in Cloud-Serverless Platform. IEEE BlackSeaCom, 325–330 (2022)
24. Lin, C., Mahmoudi, N., Fan, C., Khazaei, H.: Fine-grained performance and cost modeling and optimization for faas applications. IEEE Trans. Parallel Distrib. Syst. **34**(1), 180–194 (2022)
25. Kumari, A., Sahoo, B., Behera, R.K.: Workflow aware analytical model to predict performance and cost of serverless execution. Concurrency and Computation: Practice and Experience **35**(22), 7743 (2023)
26. Mahmoudi, N., Khazaei, H.: Performance Modeling of Serverless Computing Platforms. IEEE Transactions on Cloud Computing **10**(04), 2834–2847 (2022)
27. Maissen, P., Felber, P.: FaaSdom: a benchmark suite for serverless computing. In: ACM DEBS Proceedings, 73–84. (2020)
28. Somu, N., Daw, N.: PanOpticon: A Comprehensive Benchmarking Tool for Serverless Applications, 144–151. COMSNETS (2020)
29. Cordingly, R., Shu, W., Lloyd, W.J.: Predicting performance and cost of serverless computing functions with saaf. In: 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), pp. 640–649 (2020). IEEE
30. Tundo, A., Filippini, F., Regonesi, F., Ciavotta, M., Savi, M.: Decentralized edge workload forecasting with gossip learning. IEEE Transactions on Network and Service Management, 1–1 (2025) https://doi.org/10.1109/TNSM.2025.3570450
31. Petriglia, E., Filippini, F.: Comparing Actor-Critic and Neuroevolution Approaches for Traffic Offloading in FaaS-powered Edge Systems. In: ACM SEATED Proceedings, 17–24. (2024)
32. Ristov, S., Kimovski, D., Fahringer, T.: FaaScinating Resilience for Serverless Function Choreographies in Federated Clouds. IEEE Trans. Netw. Serv. Manage. **19**(3), 2440–2452 (2022)
33. Chawla, N.V., Bowyer, K.W., et al.: SMOTE: Synthetic Minority Over-sampling Technique. Journal of Artificial Intelligence Research **16**, 321–357 (2002)

34. Erickson, N., Mueller, J., Shirkov, A., Zhang, H., Larroy, P., Li, M., Smola, A.: AutoGluon-Tabular: Robust and Accurate AutoML for Structured Data, (2020). arXiv:2003.06505 arXiv preprint
35. Guindani, B., Lattuada, M., Ardagna, D.: AMLLibrary: An AutoML Approach for Performance Prediction. In: 37th International Conference on Modelling and Simulation (ECMS), 37, 241–247. ECMS, ??? (2023)
36. Waldmann, E.: Quantile regression: A short story on how and why. Stat. Model. **18**(3–4), 203–218 (2018)
37. Murphy, R.L., Srinivasan, B., Rao, V.A., Ribeiro, B.: Janossy Pooling: Learning Deep Permutation-Invariant Functions for Variable-Size Inputs. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, ??? (2019). https://openreview.net/forum?id=BJluy2RcFm
38. Premsankar, G., Di Francesco, M., Taleb, T.: Edge computing for the Internet of Things: A case study. IEEE Internet Things J. **5**(2), 1275–1284 (2018)
39. Ke, G., Meng, Q.: Lightgbm: A highly efficient gradient boosting decision tree. Advances in neural information processing systems, p. 30. (2017)
40. Sala R, Guindani B, Galimberti E, Filippini F, Sedghani H, Ardagna D, Risco S, Moltó G, Caballer M (2025) OSCAR-P and aMLLibrary: Profiling and predicting the performance of FaaS-based applications in computing continua. J. Syst. Softw. 221:112282. https://doi.org/10.1016/j.jss.2024.112282