Energy-Efficient Function Invocation Scheduling for Edge FaaS Platforms

Francesca Righetti[1], Biagio Cornacchia[1], Gabriele Russo Russo[2],
Nicola Tonellotto[1], Valeria Cardellini[2], Carlo Vallati[1]
[1]*Dep. of Information Engineering, University of Pisa, Pisa, Italy. E-mail: name.surname@unipi.it*
[2]*DICII, Tor Vergata University of Rome, Rome, Italy. E-mail: surname@ing.uniroma2.it*

2025 IEEE International Conference on Smart Computing (SMARTCOMP) | 979-8-3315-8646-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/SMARTCOMP65954.2025.00057

***Abstract*—Function-as-a-Service (FaaS) is a serverless computing model that enables applications to be composed of self-contained functions triggered by events. The edge-cloud continuum extends this paradigm by allowing function deployment closer to users and IoT devices, reducing communication latency. However, large-scale FaaS deployment at the edge demands energy-efficient resource management to ensure cost reduction and sustainability. To address this challenge, we present E^2FIS: *Energy-Efficient Function Invocation Scheduling*, a framework that optimizes resource management and minimizes energy consumption in edge FaaS platforms. E^2FIS formulates function scheduling as a Mixed Integer Linear Programming (MILP) problem, minimizing energy consumption by consolidating workloads while ensuring execution deadlines are met. Through simulations with real-world traces and experiments on the Serverledge FaaS platform, E^2FIS demonstrates to outperform the Earliest Deadline First (EDF) baseline, reducing energy consumption up to 92% while maintaining timely function execution.**

***Index Terms*—Function-as-a-Service, Edge-Cloud Continuum, Energy Efficiency, Scheduling.**

I. Introduction

Function-as-a-Service (FaaS) [1] is a computing paradigm that enables the deployment of applications composed of self-contained functions triggered by events. FaaS platforms handles the management and scaling of functions, abstracting the complexities of the underlying infrastructure. This abstraction makes FaaS a cost-efficient model in terms of resource usage and optimization. By abstracting infrastructure management, FaaS simplifies development and accelerates time-to-market.

Recently, cloud computing is shifting towards the edge-cloud continuum [2], integrating edge environments to enable real-time processing and low-latency applications. Edge devices provide computing and storage resources near users and IoT devices, creating a distributed infrastructure. In this model, applications, or functions in FaaS, are deployed across the edge-cloud continuum based on their latency requirements.

The growing demand for FaaS services and the large-scale deployment of FaaS platforms across the edge-cloud continuum place a strong emphasis on energy efficiency in resource management to reduce operational costs and minimize environmental impact [3]. This requires balancing the Quality of Service (QoS) compliance [4] with minimizing energy consumption to ensure efficient and sustainable operations.

Existing research on FaaS platforms has primarily focused on energy-aware scheduling for infrastructures with nodes powered by renewable energy, with the aim of improving service availability by prioritizing nodes with stable energy sources. In [3], nodes are grouped based on their power budget, prioritizing those with greater energy stability to improve service availability. Similarly, faasHouse [5] leverages computation offloading to balance energy availability across battery-powered nodes, enhancing operational efficiency in Kubernetes-based environments. In [6], approximate computing is used to balance energy efficiency and computational accuracy, extending the lifespan of battery-powered edge devices. Among the few works that do not focus on energy-constrained devices, EcoFaaS [7] aims to improve energy efficiency at the node level, by profiling functions and dynamically scaling CPU frequency. Conversely, in [8], which targets cloud environments, energy consumption reduction is achieved through load balancing across function chains to distribute workload and optimize system performance. However, these works overlook node and function heterogeneity, a critical factor in the edge-cloud continuum, where nodes have heterogeneous computational capacities and functions have diverse execution deadlines and require different amount of resources.

In this paper, we present E^2FIS: *Energy-Efficient Function Invocation Scheduling*, a framework for optimizing resource management in edge FaaS platforms, with a focus on energy efficiency and QoS compliance, guaranteeing that functions meet their deadlines. Unlike previous approaches, E^2FIS prioritizes workload consolidation, scheduling functions on energy-efficient nodes and deactivating idle ones, achieving system-wide energy savings. Additionally, it explicitly considers node and function heterogeneity, making it well-suited for the edge-cloud continuum. We formulate function scheduling as a Mixed Integer Linear Programming (MILP) problem to minimize energy consumption by consolidating FaaS workload, assigning functions to energy-efficient nodes. Idle nodes are switched off to reduce energy waste. MILP was chosen for its ability to precisely model the problem and produce optimal solutions under constrained resources and stringent QoS requirements, as required in edge FaaS platforms. To address scalability limitations [9], we adopted a time-limited heuristic: when the optimization problem is solved within a certain time, the optimal solution is used. Otherwise, the best suboptimal solution available within that limit is employed.

We evaluate E^2FIS through simulations using Microsoft Azure traces [10] and through real experiments on the Serverledge FaaS platform [11], comparing its performance

2693-8340/25/$31.00 ©2025 IEEE
DOI 10.1109/SMARTCOMP65954.2025.00057

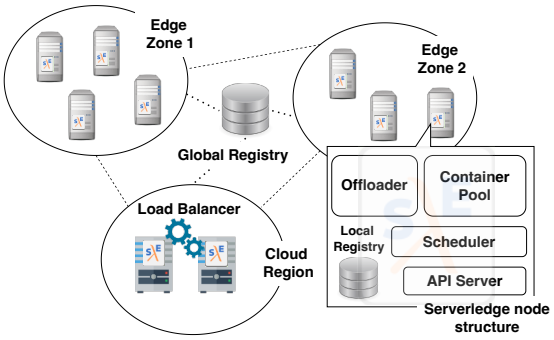


Fig. 1: Serverledge framework

against the Earliest Deadline First (EDF) scheduling [12]. Results show that E^2FIS reduces energy consumption up to 92%, ensuring that functions execute within their deadlines.

The rest of the paper is organized as follows: Section II provides background on edge computing and FaaS. Section III details the MILP formulation behind E^2FIS, which is then evaluated through simulations (Section IV), and real experiments (Section V). Finally, Section VI draws the conclusions.

II. Technical Background

FaaS has evolved from cloud computing, allowing applications to be executed as a composition of self-contained functions triggered by events, in a serverless fashion. The FaaS platform manages the execution and scalability of functions transparently for developers.

To support geographically pervasive applications, and those with stringent QoS requirements, especially in terms of latency, FaaS has extended to the edge-cloud continuum [5], [13], integrating edge nodes near users and IoT devices.

Despite the variety of FaaS platforms for cloud and edge-cloud environments [11], [14]–[16], they share a common architecture. Functions are triggered by events (e.g., IoT alarms, database changes, sensor readings) and are executed on worker nodes. Function invocations may be dispatched by a centralized gateway/load-balancer or through a distributed scheduling approach, in which worker nodes manage execution and dispatch of functions. In both cases, the decision on which worker node a function should be executed is made according to a certain policy, which takes into account the requirements of the function and the current status of the platform. Functions run in containerized environments like Docker [17] for isolation and resource management.

Among open-source FaaS platforms, Serverledge [11] is one of the few designed for the edge-cloud environment. Its architecture, shown in Fig. 1, organizes nodes into edge zones (serving regions near end users) and cloud regions (data centers), to enable localized function scheduling and minimize delays. Node memberships and registered functions, are managed by a global registry to ensure scalability and fault tolerance. The global registry is built on etcd [18], which is a highly available, distributed, and consistent key-value store designed to provide reliable data management for distributed systems. Additionally, regions or zones, particularly cloud

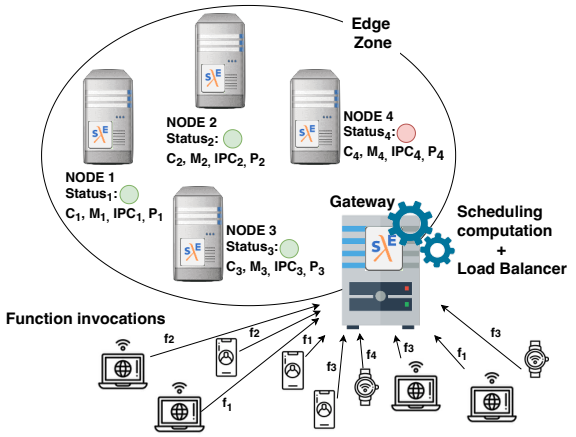


Fig. 2: System architecture

regions, may include a Load Balancer to evenly distribute requests among the nodes within the region.

Serverledge enables function execution on distributed nodes deployed at edge locations or cloud data centers. Each worker node handles invocation requests using a local container pool, ensuring function isolation. By maintaining a warm pool of containers, it reduces initialization overhead for frequently invoked functions. When edge nodes are overloaded, Serverledge supports horizontal offloading (between edge nodes) and vertical offloading (from edge to cloud), enhancing scalability and flexibility during peak demand.

A Serverledge node, as shown in Fig. 1, is composed by:

- API Server: provides HTTP endpoints for creating, invoking, and managing functions.
- Local Registry: caches the global registry, storing information about nearby nodes, registered functions, and monitoring local resources and neighboring nodes status.
- Scheduler: based on the workload, it assigns requests to available nodes, allocating warm containers, initializing new ones, or offloading requests, prioritizing low-latency responses and resource balancing.
- Container Pool: manages function execution within containers, reusing warm containers to reduce initialization overhead and dynamically adjusting to workload changes.
- Offloader: transfers invocation requests to other nodes when local execution is not feasible. Requests are routed on the basis of local registry data.

By integrating containerized execution, dynamic scheduling, and offloading, Serverledge ensures an efficient and scalable FaaS execution environment in the edge-cloud continuum [13].

III. System Model and Problem Formulation

This section presents the system architecture we considered to define E^2FIS, describes the system model, and presents the problem formulation for E^2FIS.

A. System Architecture and Model

The system architecture for E^2FIS is shown in Fig. 2. Function invocations are directed to the closest edge zone,

where a gateway executes E^2FIS. Time is divided into discrete epochs, with the optimization problem at the core of E^2FIS (formulated in Section III-B) solved before each epoch begins. The solution determines function assignments to worker nodes and, hence, the minimum number of nodes that must be active to handle the FaaS workload. Any idle worker node is powered off to minimize energy consumption.

Each edge node i is characterized by: (i) $status_i$, indicating if the node is active or powered off, (ii) C_i, available computational capacity, i.e., CPU, (iii) M_i, available memory, (iv) IPC_i, instructions per cycle, which is the average number of instructions executed per clock cycle, and (v) P_i, power consumption.

The functions registered in the FaaS platform are identified by the set $F = \{f_1, f_2, ..., f_M\}$. Each function f_j is characterized by: (i) m_j, the amount of memory required for its execution, (ii) w_j, the number of instructions it requires to execute, i.e., workload, and (iii) d_j, deadline, the maximum time that can elapse between function invocation and completion.

To ensure functions meet their deadlines, E^2FIS assumes the worst-case load to compute the function allocation to nodes. Specifically, for each epoch, an estimate of the maximum number of concurrent invocations of a function within a given time slot, denoted as n_j, is provided as input. This approach aligns with [19], where batches of requests are scheduled for serverless workflows in the cloud-continuum. A forecasting model could estimate n_j, but load prediction is beyond the scope of this paper. We assume n_j is known in advance, potentially derived using an off-the-shelf forecaster, which could be seamlessly integrated into E^2FIS without modifications. The evaluation of E^2FIS's robustness to inaccuracies in the forecasting model is left as future work.

B. Problem Formulation

At the core of E^2FIS is an optimization problem, formulated as a MILP, that determines the optimal functions placement across worker nodes. It prioritizes execution on the most energy-efficient nodes while ensuring functions meet their deadlines, allowing idle nodes to be powered off. To define this optimization problem, we consider an edge zone with N heterogeneous edge nodes, each with different computing capacity, memory, and power consumption. The problem is formulated as follows:

$$\min \quad \sum_{i=0}^{N} y_i E_i \tag{1.1}$$

$$\text{subject to} \quad \sum_{j=0}^{M} n_{ij} m_j \leq M_i y_i, \quad \forall i \tag{1.2}$$

$$\sum_{j=0}^{M} c_{ij} \leq C_i y_i, \qquad \forall i \tag{1.3}$$

$$\frac{n_{ij} w_j}{c_{ij} IPC_i} \leq d_j, \quad \forall i, \forall j \tag{1.4}$$

$$\sum_{i=0}^{N} n_{ij} = n_j, \quad \forall j \tag{1.5}$$

The model outputs: (i) node status for the next epoch, given by the binary decision variable y_i; (ii) expected maximum number of concurrent invocations of function f_j assigned to node i for the next epoch, given by the decision variable n_{ij}; (iii) overall CPU allocated to function f_j on node i, given by the decision variable c_{ij}.

The objective function in Eq. 1.1 determines the set of active edge nodes needed to execute functions within their deadlines, while minimizing the total energy consumption of the system. Here, E_i is the energy consumption required to keep node i active for the next epoch.

The constraint in Eq. 1.2 ensures that the total memory required for executing function f_j on node i does not exceed its memory capacity, where n_{ij} is the maximum number of concurrent requests to be allocated for function f_j on node i.

The constraint in Eq. 1.3 ensures that the total CPU allocated for all invocations of function f_j on node i does not exceed its total CPU available.

Eq. 1.4 ensures that, for all nodes, the allocated resources for each function are sufficient to guarantee execution within its deadline, d_j. Specifically, the execution time τ is defined as:

$$\tau = \frac{w_j}{c'_{ij} \cdot IPC_i}, \qquad c'_{ij} = \frac{c_{ij}}{n_{ij}}$$

where c'_{ij} is the computational capacity allocated for function f_j on node i, w_j is the function workload, and IPC_i is the number of instructions per clock cycle of node i.

Finally, the constraint in Eq. 1.5 ensures that the total number of invocations of function f_j assigned to each node i, i.e., n_{ij} matches the maximum number of concurrent invocations expected in the edge zone for each function f_j, i.e., n_j.

Without loss of generality, we define each node's computational capacity C_i (GHz), available memory M_i (GB), and instantaneous power consumption P_i (Watt). Additionally, we model the energy consumption of each node, E_i, assuming that an active node continuously consumes its nominal power P_i. It is worth noting that alternative energy consumption models can be seamlessly integrated into the problem formulation.

IV. Simulation Analysis

We conducted both simulations and real-world experiments to evaluate the performance of E^2FIS. Simulations offer insights into its large-scale behavior, allowing us to explore a broad range of parameters and configurations that would be challenging to assess in a real testbed. Conversely, real-world experiments, though conducted on a smaller scale, demonstrate the practical feasibility and effectiveness of the proposed framework. In this Section, we focus on the simulation-based evaluation of E^2FIS, analyzing its performance across various parameters and configurations. We first present the simulation setup (Section IV-A) and then discuss the results (Section IV-B).

A. Simulation Setup

We implemented the optimization problem at the core of E^2FIS in Python, utilizing OR-Tools [20], an open-source

combinatorial optimization suite developed by Google, to solve it. Specifically, we employed the Constraint Programming SATisfiability (CP-SAT) solver, which combines Constraint Programming (CP) and Boolean Satisfiability (SAT) techniques, encoding constraints as Boolean formulas to efficiently solve MILP-based problems. For ease of deployment, we integrated the solver with a Flask-based server[1] exposing HTTP APIs and encapsulated it in a *Docker* container. This setup enables seamless execution across simulations and real experiments without requiring any change in the code.

The simulator developed to evaluate E^2FIS replicates the behavior of a system with many edge nodes, each characterized by specific computational capacity, memory, and power consumption. The node parameters we used are taken from realistic values [21]–[23], and are summarized in Table I. We simulated an edge zone with 20 heterogeneous nodes, distributed according to the node type percentages in Table I. The FaaS platform hosted 40 registered functions, which is a realistic configuration in such environments [11]. We considered epochs of varying durations: 15, 30, and 60 minutes.

The simulator divides time into epochs and solves the MILP problem, before the start of each epoch, to determine the optimal functions placement. This requires up-to-date resource and function data for each epoch, allowing the CP-SAT solver to decide which nodes should remain active and the allocation of functions for the next epoch. To ensure practical execution times, we set a 400-second limit for the solver. If an optimal allocation is not found within this time, the solver returns a *feasible*, but potentially suboptimal, solution. In cases where system resources are insufficient to meet all function QoS requirements, the solver fails, resulting in the activation of all nodes and function assignment based on the EDF approach.

To realistically simulate function invocations, we used real traces from the *Microsoft Azure* dataset [10], which contains serverless workload data collected over two weeks in 2019. From this dataset, we extracted the number of function invocations and their deadlines. The latter, were computed as $d_j = \mu_j + 2\sigma_j$, where μ_j is the mean execution time of f_j, and σ_j is its standard deviation.

We define the evaluation scenarios by classifying function invocation load as **Low Intensity (LI)** or **High Intensity (HI)** and function deadlines as **Short Deadline (SD)** or **Long Deadline (LD)**. In LI scenarios, functions receive at most a few hundred invocations per 1-second time interval, whereas in HI scenarios, the invocation count reaches thousands. This directly impacts the maximum number of concurrent invocations, as a higher number of function invocations leads to greater concurrency demands within the given time interval. LD scenarios feature deadlines ranging from several to tens of seconds, whereas SD scenarios have stricter deadlines in the millisecond range. Our four evaluation scenarios, ordered from the most demanding (highest workload and strictest deadlines) to the least demanding are: *SD-HI*, *LD-HI*, *SD-LI*, *LD-LI*.

As discussed in Section III, predicting the maximum number of concurrent invocations for each function in the next epoch is beyond the scope of this work. Therefore, in our experiments, we assume an ideal predictor that accurately determines this value at the start of each epoch.

[1]https://palletsprojects.com/projects/flask

TABLE I: Characteristics of the heterogeneous (edge) worker nodes considered in the simulations.

Node ID	CPU (GHz, cores)	Memory (GB)	IPC	Power (W)	% node used in simulation
A	1.8, 4	8	0.5	4	20% of 20
B	2.4, 4	8	0.5	6	20% of 20
C	3.2, 6	16	1.0	150	15% of 20
D	2.4, 12	16	1.5	200	10% of 20
E	3.3, 8	32	1.0	230	15% of 20
F	3.1, 16	32	2.0	300	10% of 20
G	2.6, 16	64	3.0	350	10% of 20

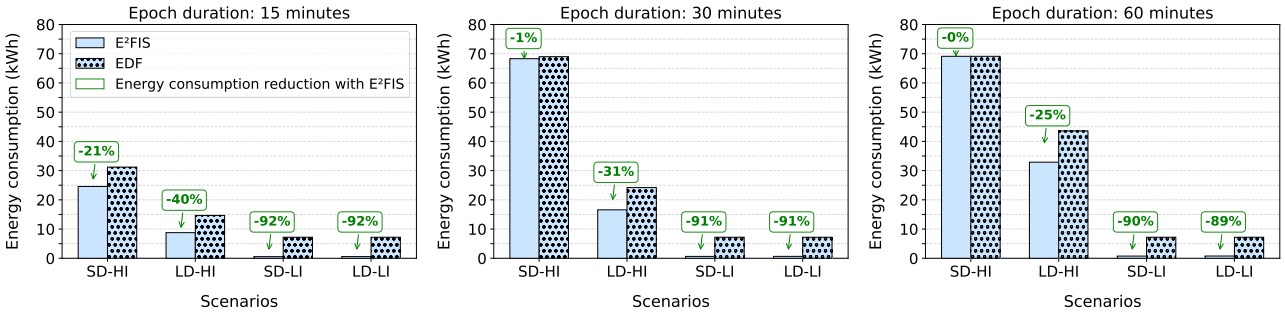
To evaluate E^2FIS, we use *Earliest Deadline First (EDF)* as a baseline comparison. EDF is a widely used scheduling algorithm for real-time systems [12] that prioritizes function execution based on deadline proximity, assigning higher priority to functions with the earliest deadlines. In our EDF implementation, nodes are first sorted by computational and memory capacity, favoring those with higher resources. Functions are then ordered by deadline, prioritizing those with the shortest completion times. The scheduler assigns high-priority functions to high-capacity nodes, aiming to optimize resource usage and ensure deadline adherence.

To assess the effectiveness of E^2FIS, we define the following evaluation metrics: (i) *energy consumption*: the total energy, in KWh, consumed by active nodes over the evaluation period (one full day). It is calculated as the cumulative sum of energy consumed by all active nodes, based on their power consumption; (ii) *computational capacity utilization*: the percentage of the total available computational capacity utilized by active nodes in each epoch; (iii) *system memory utilization*: the percentage of the total available memory utilized by the active nodes in each epoch; (iv) *number of active nodes*: the number of nodes that remain active in each epoch. Additionally, to gain deeper insights into the system's behavior, we analyze the solver's performance, evaluating the percentage of solutions classified as *feasible*, or *unfeasible*.

The results presented in the graphs are derived from simulations where E^2FIS and EDF schedule function invocations over a full day. Metrics are computed per epoch, and to ensure statistical sound results, we report their average values along with the 95% confidence interval. An exception is energy consumption, which is measured as the total daily energy usage of the edge zone. In this case, the reported value represents the cumulative energy consumed throughout the day by the nodes that remained active in each epoch.

B. Simulation Results

Fig. 3 presents the total energy consumption for E^2FIS and EDF over a full day of simulation with 40 functions registered on the FaaS platform. Experiments were also conducted with different numbers of functions, but for the sake of brevity, these results are omitted because they exhibit similar trends.



(a) (b) (c)

Fig. 3: Total energy consumption for E^2FIS and EDF across all considered scenarios and epoch durations

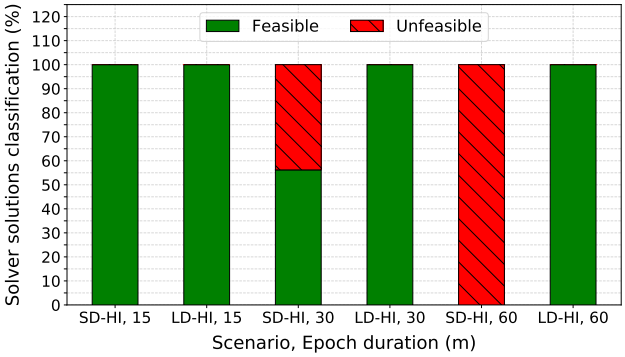


Fig. 4: E^2FIS optimization problem: solutions classification

The three graphs in (Fig. 3) display energy consumption values when considering the epoch duration of 15, 30, and 60 minutes, respectively.

Let us start analyzing these results considering Fig. 3a, i.e., when epochs are 15 minutes long. As shown in the graph, E^2FIS consistently achieves *lower* energy consumption compared to EDF, with savings ranging from 21% to 92%, depending on the scenario. The lowest energy saving occurs in the SD-HI scenario, which is characterized by tight deadlines and high invocation intensity. However, when deadlines are more relaxed (LD-HI) or invocation intensity decreases (SD-LI, LD-LI), energy saving increases significantly. This is because, in less congested scenarios, there is greater flexibility in function allocation, allowing E^2FIS to keep active only the most energy-efficient nodes. These nodes are often less powerful in terms of computational resources, whereas EDF prioritizes activating the most powerful nodes first to ensure functions meet their deadlines. This results in unnecessary energy waste, as high-performance nodes consume more power.

If we analyze energy consumption over longer epochs, such as 30 and 60 minutes (Figs. 3b and 3c), we observe a trend where E^2FIS energy savings gradually decrease. This reduction is almost negligible in low-intensity scenarios but becomes more significant when the number of function invocations is high. This behavior is due to the reduced precision in function allocation with longer epochs. Larger time windows increase the likelihood of high peaks in the number of concurrent function invocations, making it more challenging to efficiently consolidate workload and power off idle nodes, ultimately limiting energy savings. Moreover, it is worth highlighting the behavior of the SD-HI scenario (both in Figs. 3b and 3c) that requires separate consideration. In this case, unlike previous cases, E^2FIS does not achieve energy savings, as longer epochs not only reduce energy efficiency but also impact the feasibility of allocation. The high system load prevents the solver from finding a feasible solution that meets function deadlines. As a result, the system falls back to the EDF approach for function allocation. Since E^2FIS and EDF exhibit the same energy consumption, we observe that E^2FIS behaves like EDF in this scenario, since both scheduling policies activate all nodes, leading to comparable energy usage.

To further analyze E^2FIS's behavior, Fig. 4 illustrates the percentage of solutions of the optimization problem classified as feasible (green) or unfeasible (red with bars). These results align with the previous analysis, confirming that feasibility decreases as system load and epoch duration increase. Specifically, in the SD-HI scenario, where the system experiences high load, a significant portion of solutions (45%) is unfeasible for epochs of 30 minutes, while for 60 minutes epochs, no feasible solution is found. For readability, we excluded lower-load scenarios (SD-LI and LD-LI) from the graph, as with lower function invocation intensity, E^2FIS consistently finds feasible solutions, resulting in all green bars.

Another key aspect in comparing E^2FIS and EDF is the average number of active nodes per epoch, as shown in Fig. 5, across all scenarios and epoch durations. In most cases, E^2FIS consistently activates more nodes than EDF, despite achieving greater energy efficiency (Fig. 3). This behavior is directly linked to the optimization strategy at the core of E^2FIS, which prioritizes function allocation to the most energy-efficient nodes. These nodes typically have lower computational capacity, requiring more nodes to be activated to handle the system's workload. However, even with a higher number of active nodes, overall energy consumption remains lower or, at worst, comparable to that of EDF. The cases in which both E^2FIS and EDF activate almost all nodes, i.e., 20, occur when a significant portion (or all) of the allocations cannot be determined, as the optimization problem fails to find a QoS compliant solution.

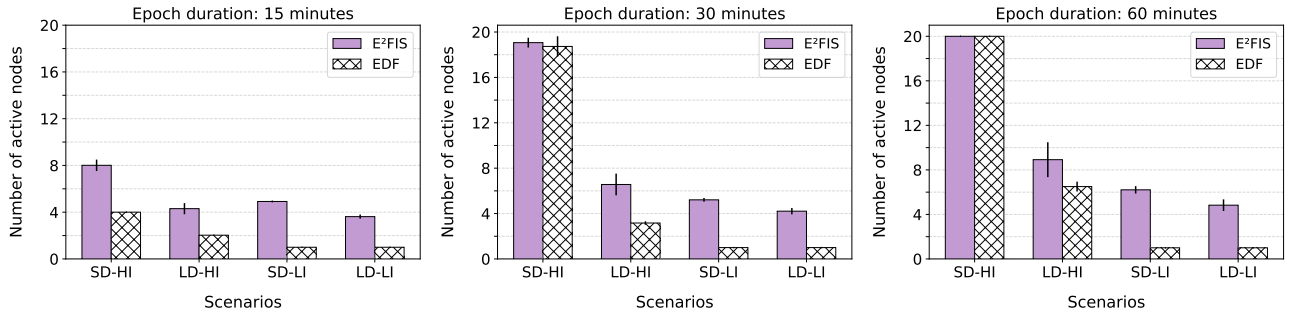


Fig. 5: Average number of active nodes per epoch for E^2FIS and EDF

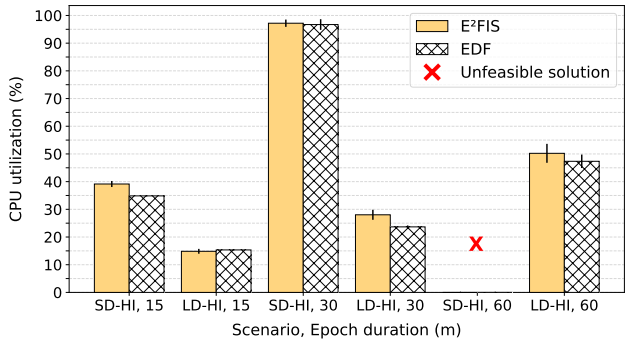


Fig. 6: CPU utilization estimation. E^2FIS and EDF

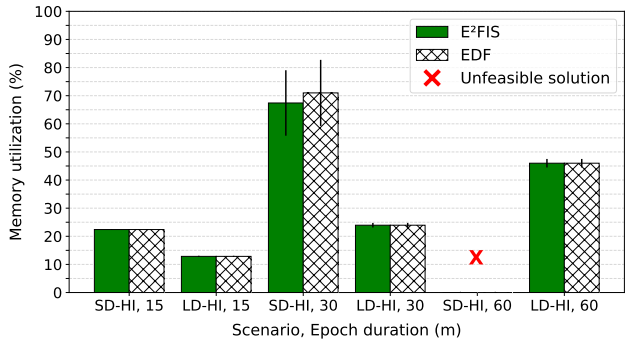


Fig. 7: Memory utilization estimation. E^2FIS and EDF

To conclude the simulation analysis, we report per-epoch estimation of CPU (Fig. 6) and memory utilization (Fig. 7) for all active nodes. It is important to note that, in the simulator, the functions are not actually executed. Therefore, resource utilization is estimated based on each function's workload, memory and CPU requirements, and deadline. For brevity, we report only the two most loaded scenarios, as the same trend applies to the others. Additionally, since this estimation relies on the function allocation policy, it cannot be computed when all the solutions are unfeasible. This is why a red "X' marker appears in the graph. Regarding CPU utilization, E^2FIS shows utilization percentages comparable to or higher than EDF. This aligns with the approach of E^2FIS, which consolidates the workload on energy-efficient nodes, leading to higher resource utilization and minimizing computational waste. In contrast, EDF prioritizes more powerful nodes, which often remain underutilized due to its allocation strategy.

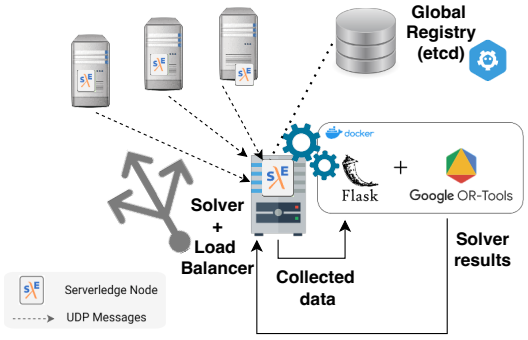


Fig. 8: E^2FIS integration with Serverledge

Fig. 7 presents memory utilization estimates, included for completeness, to show that scheduling policies have minimal impact on memory usage. Unlike computational capacity, which affects execution time, memory usage remains constant. Each function allocates a fixed memory footprint, leading to identical utilization under both E^2FIS and EDF.

V. Experimental Analysis

To evaluate the effectiveness and practicality of E^2FIS, we carried out real-world experiments on the Serverledge FaaS platform, complementing simulation insights under realistic conditions. We first describe the experimental setup, including E^2FIS integration [24], hardware configuration, and workload characteristics, followed by a discussion of the results.

A. Experimental Setup

E^2FIS has been integrated into Serverledge by developing a *solver module* and introducing a *Load Balancer* which acts as a reverse proxy to enforce the solver's scheduling policy for the assignment of functions.

The solver module, deployable on any Serverledge node (referred to as the *Solver Node* or simply the *Solver*), communicates with other nodes using Serverledge's edge monitoring mechanisms, as shown in Fig. 8. Through packet exchanges, edge nodes transmit to the Solver Node the necessary information to solve the E^2FIS optimization problem, including function details and node-specific data. At the start of each epoch, the Solver processes these data, determines the function allocation, and stores them in the global registry.

TABLE II: Characteristics of the heterogeneous (edge) worker nodes considered in the real deployment

Role	Node ID	CPU Specs (GHz, cores)	Memory (GB)	IPC	Power (W)
CLOUD	0.1	2.2, 4	8	N/A	N/A
SOLVER + LOAD BALANCER	0.2	2.2, 2	2	N/A	N/A
EDGE	1	2.7, 2	2	2.45	6
EDGE	2	2.8, 4	8	2.20	130
EDGE	3	2.8, 16	32	2.45	250
EDGE	4	2.7, 4	4	2.45	100
EDGE	5	2.8, 4	4	2.15	100

TABLE III: Details of functions used in the experiments

Name	Number of instructions	Memory (GB)	Deadline (s)	n_j-High workload	n_j-Medium workload
Isprime 1	5.86×10^7	0.128	0.08	10	4
Isprime 2	5.86×10^7	0.128	0.08	10	4
Jsonschema	4.84×10^8	0.128	0.13	3	3
Linear search	1.38×10^9	0.128	0.37	1	1
K-means clustering	6.46×10^9	0.256	0.41	5	5
Integer factorization	1.57×10^{10}	0.128	3.82	6	3
Caesar cipher	2.29×10^{10}	0.128	5.68	4	3
Count inversions	5.22×10^{10}	0.128	12.71	3	3
Bubblesort 1	7.81×10^{10}	0.128	19.1	5	5
Bubblesort 2	4.91×10^{11}	0.256	119	3	3

The Load Balancer incorporates an observer of the etcd global registry to monitor any updates in real time. Upon detecting a new allocation, the observer triggers the Load Balancer, which uses the Serverledge API to create the necessary containers on assigned nodes for the next epoch. If the Solver fails to provide an allocation (unfeasible solution), the Load Balancer resorts to an EDF-like policy.

For the experiments, we consider an edge zone with 7 nodes equipped with amd64-architecture CPUs, as detailed in Table II. Node 0.1 hosts the etcd global registry, and Node 0.2 runs the Solver and Load Balancer, both always active and thus excluded from power consumption calculations. The remaining five worker nodes handle the execution of functions.

To estimate nodes' IPC, we executed the functions listed in Table III, and profiled their execution using the *perf tool* [25], which provides IPC and instruction count metrics. The average IPC per node was computed by averaging IPC values across all the considered functions. Similarly, the instructions count for each function (Table III), was derived from the profiling data collected by executing perf on each node.

In our experiments, we considered 15 and 30 minute epochs and evaluated two workload types, defined by different values

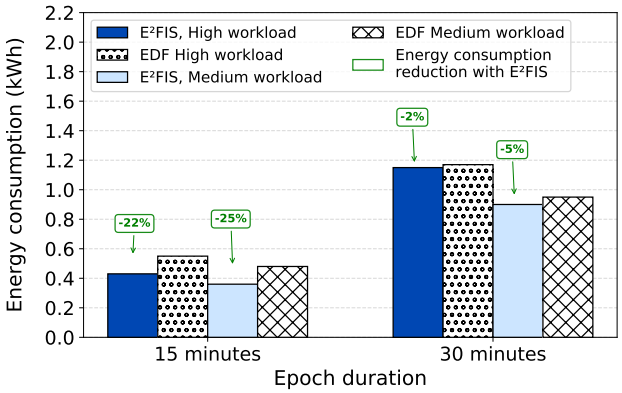


Fig. 9: Total energy consumption. Real experiments

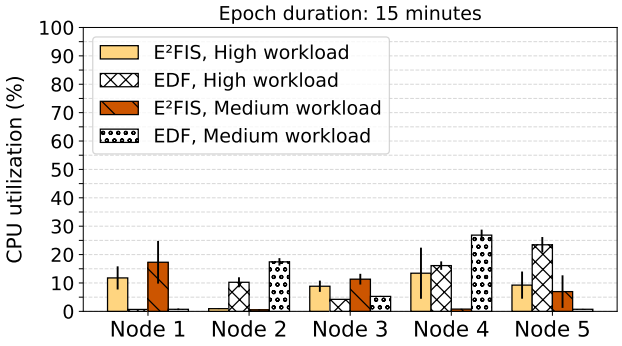


Fig. 10: CPU utilization. Real experiments

of the maximum number of concurrent function invocations per 1-second time slot, n_j: *high* and *medium workload*, as shown in Table III. For instance, the *Isprime 1* function has a maximum of 10 concurrent invocations per second (4) under the high (medium) workload scenario. We tested each scenario over five experiments to ensure statistically sound results.

B. Experimental Results

Fig. 9 shows the energy consumption of E^2FIS and EDF over a single epoch. The results obtained from real-world experiments confirm the findings from the simulations, demonstrating that E^2FIS achieves a significant reduction in energy consumption compared to EDF. This effect is particularly evident for 15 minute epochs, where energy consumption is reduced in a range from 22% to 25%. However, for longer epoch durations, the reduction becomes less pronounced (ranges from 2% to 5%), which aligns with the observations from the simulation analysis. Specifically, longer epochs decrease scheduling granularity, making it more challenging to consolidate workloads effectively and power off idle nodes.

Beyond energy consumption, real-world experiments allowed us to obtain precise insights into CPU and memory utilization (Fig. 11) on a per-node level, enabling a more granular analysis compared to simulations. Both graphs focus on a 15 minute epoch duration. Results for 30 minute epochs are omitted for brevity, as they exhibit a similar trend.

From Fig. 10, we observe significantly different node utilization patterns between the two scheduling policies. E^2FIS

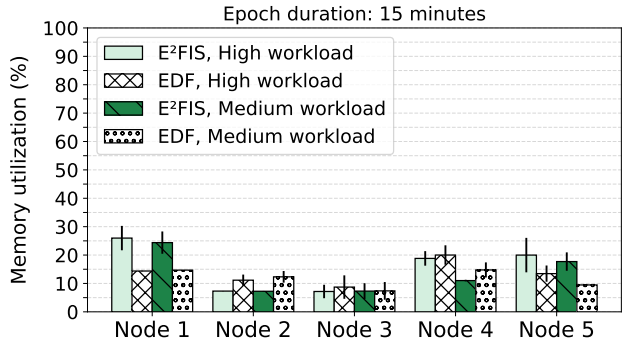


Fig. 11: Memory utilization. Real experiments

predominantly relies on Node 1 in both high and medium workload scenarios. This behavior aligns with the core principle of E^2FIS, as Node 1 has the lowest power consumption, making it the preferred choice for energy-efficient scheduling. Conversely, EDF prioritizes nodes with higher CPU frequencies, particularly Nodes 2, 4, and 5, which offer greater computational capacity but are less energy-efficient.

This difference in scheduling behavior is also reflected in Fig. 11, where memory utilization follows the same pattern as CPU usage. Nodes that experience higher CPU utilization also exhibit higher memory consumption, further confirming that the scheduling strategy directly impacts resource utilization.

VI. Conclusions

In this paper, we proposed E^2FIS (Energy-Efficient Function Invocation Scheduling), a novel framework for optimizing function scheduling in edge FaaS platforms. By modeling the problem as a MILP, E^2FIS reduces energy consumption while meeting function deadlines, prioritizing execution on energy-efficient nodes and powering off idle ones. We evaluated E^2FIS through simulations and real-world experiments. Simulations allowed us to analyze the system performance under various workload, and the results showed that E^2FIS outperforms Earliest Deadline First (EDF), achieving up to 92% energy consumption reduction under low load, and maintaining savings even in high-load scenarios by consolidating execution on energy-efficient nodes. Real-world experiments on Serverledge confirmed its energy-saving potential and effectiveness in dynamic and realistic operational conditions.

For future work, we will explore distributed scheduling to enhance scalability and reduce computation time in large-scale edge deployments.

Acknowledgment

This work was supported by the Italian Ministry of University and Research (MUR) in the framework of the (i) FoReLab project ("Departments of Excellence" program), (ii) PNRR National Centre for HPC, Big Data, and Quantum Computing (Spoke 1, CUP: I53C22000690001).

References

[1] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: the next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, p. 76–84, Apr. 2021.

[2] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *Internet of Things*, vol. 3, pp. 134–155, 2018.

[3] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and R. Gaire, "Energy-aware resource scheduling for serverless edge computing," in *22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 190–199.

[4] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, p. 1063–1075.

[5] M. S. Aslanpour, A. N. Toosi, M. A. Cheema, and M. B. Chhetri, "Faashouse: Sustainable Serverless Edge Computing Through Energy-Aware Resource Scheduling," *IEEE Transactions on Services Computing*, vol. 17, no. 4, pp. 1533–1547, 2024.

[6] C. Calavaro, G. Russo Russo, M. Salvati, V. Cardellini, and F. Lo Presti, "Towards energy-aware execution and offloading of serverless functions," in *4th Workshop on Flexible Resource and Application Management on the Edge*. ACM, 2024, p. 23–30.

[7] J. Stojkovic, N. Iliakopoulou, T. Xu, H. Franke, and J. Torrellas, "Ecofaas: Rethinking the design of serverless environments for energy efficiency," in *ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 471–486.

[8] S. H. Rastegar, H. Shafiei, and A. Khonsari, "Enex: An energy-aware execution scheduler for serverless computing," *IEEE Transactions on Industrial Informatics*, vol. 20, no. 2, pp. 2342–2353, 2024.

[9] B. Ahat, A. C. Baktır, N. Aras, I. K. Altınel, and A. Özgövde, "Optimal server and service deployment for multi-tier edge cloud computing," *Computer Networks*, vol. 198, p. 108385, 2021. [Online]. Available: https://doi.org/10.1016/j.comnet.2021.108385

[10] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC '20)*, 2020, pp. 205–218.

[11] G. Russo Russo, T. Mannucci, V. Cardellini, and F. Lo Presti, "Serverledge: Decentralized function-as-a-service for the edge-cloud continuum," in *2023 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, 2023, pp. 131–140.

[12] F. Zhang and A. Burns, "Schedulability analysis for real-time systems with EDF scheduling," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1250–1258, 2009.

[13] G. Russo Russo, V. Cardellini, and F. Lo Presti, "A framework for offloading and migration of serverless functions in the edge–cloud continuum," *Pervasive and Mobile Computing*, vol. 100, 2024.

[14] "AWS Lambda: Serverless compute service," accessed: 2025-02-28. [Online]. Available: https://aws.amazon.com/lambda/

[15] "OpenFaaS: Serverless functions made simple," accessed: 2025-02-28. [Online]. Available: https://www.openfaas.com/

[16] "Apache OpenWhisk: A serverless cloud platform," accessed: 2025-02-28. [Online]. Available: https://openwhisk.apache.org/

[17] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[18] "etcd," accessed: 2025-02-28. [Online]. Available: https://etcd.io

[19] R. Farahani, N. Mehran, S. Ristov, and R. Prodan, "Heftless: A bi-objective serverless workflow batch orchestration on the computing continuum," in *IEEE International Conference on Cluster Computing (CLUSTER)*, 2024, pp. 286–296.

[20] L. Perron and F. Didier, "CP-SAT Solver," Google. [Online]. Available: https://developers.google.com/optimization/cp/cp_solver/

[21] "Raspberry Pi 4 Model B," https://www.raspberrypi.com/products/raspberry-pi-4-model-b/.

[22] "Industrial High Performance Xeon E Edge Server," https://www.onlogic.com/store/mc850-54/, 2024.

[23] "Raspberry Pi 5," https://www.raspberrypi.com/products/raspberry-pi-5/.

[24] "E^2FIS implementation," accessed: 2025-02-28. [Online]. Available: https://github.com/Serverledge-UNIPI/PNRR-Spoke1_HPC/tree/dev

[25] PerfWiki, "Perf: linux profiling with performance counters," https://perf.wiki.kernel.org/index.php/Main_Page, 2024.