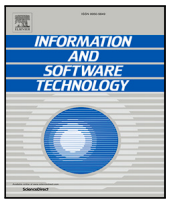
Information and Software Technology 186 (2025) 107819

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Integrating energy consumption in the development of serverless applications

Pablo Serrano-Gutierrez *, Inmaculada Ayala, Lidia Fuentes

Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Málaga, Spain
ITIS Software, Universidad de Málaga, Málaga, Spain

ARTICLE INFO

Keywords:
Energy consumption
Serverless
Sustainability
Self-adaptive
Kepler

ABSTRACT

Context: The increasing environmental impact of Information and Communication Technologies (ICTs), particularly the energy consumption associated with serverless applications, necessitates the development of methodologies to optimize energy efficiency. This study addresses the need for energy-aware design and runtime adaptation in serverless architectures.
Objective: To develop and validate a methodology that integrates energy monitoring into the development and runtime management of serverless applications, thereby enabling significant reductions in energy consumption while maintaining functionality.
Methods: A new version of FUSPAQ, a framework for the optimization of serverless applications, was developed. This version incorporates tools like Kepler for real-time energy monitoring and employs an energy-aware orchestration mechanism to dynamically select energy-efficient function configurations. Validation was conducted through a facial recognition case study and benchmark experiments, comparing energy consumption across different scenarios with and without the proposed adaptations.
Results: The enhanced FUSPAQ framework successfully integrated energy consumption metrics into the decision-making process for function selection and runtime adaptation. Benchmark tests confirmed the scalability of the solution, with energy-efficient outcomes even in complex applications.
Conclusion: The study highlights the potential of integrating energy-aware practices in serverless applications, presenting a scalable and practical approach to reducing their environmental footprint. By leveraging tools like Kepler and frameworks like FUSPAQ, developers can achieve significant energy savings without compromising application performance. This work contributes to the advancement of Green Software Engineering by emphasizing runtime energy adaptation in Function-as-a-Service (FaaS) architectures.

1. Introduction

Carbon dioxide emissions have increased by more than 60% since 1990, accelerating the greenhouse effect. In the coming years, the Information and Communication Technology sector is expected to account for 14% due to a proliferation of everyday devices connected to the Internet (e.g., mobile phones, watches, electric meters, etc.) [1]. Although software systems do not consume energy directly, they affect the energy consumption of the hardware. As a result, a new discipline, Green Software Engineering, has recently emerged [2]. This new discipline aims to ensure that software efficiently uses hardware resources to minimize environmental impact. Therefore, current methodologies and software development technologies should be reviewed to produce sustainable software from an energy point of view (Energy-aware Software Engineering [3]). The role of a software engineer is crucial because it involves understanding how each design and implementation decision impacts the system's energy consumption. Therefore, when running an application, it is important to consider not only non-functional requirements such as execution time and latency but also to prioritize energy consumption as a critical requirement.

In recent years, driven by the popularity of cloud environments, the serverless paradigm, sometimes known as Function as a Service (FaaS), has taken centre stage. In these systems, microservices-based applications can be scaled simply without a fixed infrastructure and with reduced administrative costs. In addition, providers usually offer a pay-per-use model, which is more economically beneficial as costs are adapted to the needs of each moment. However, due to the high amount of microservices composing an application, it is also necessary to consider their behaviour from an energy point of view, studying how the

* Correspondence to: ETSI Informática, Blvr. Louis Pasteur, 35, 29010 Málaga, Spain.
E-mail address: pserranogu@uma.es (P. Serrano-Gutierrez).

https://doi.org/10.1016/j.infsof.2025.107819
Received 15 January 2025; Received in revised form 27 April 2025; Accepted 13 June 2025
Available online 26 June 2025
0950-5849/© 2025 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

global energy footprint can be reduced. Recently, several works have been proposed to address energy consumption in serverless contexts. These works mainly focus on modifying the scheduling of orchestrators used by serverless frameworks (e.g. Kubernetes), forcing them to place or offload some function replicas in edge-based environments [4] or postponing the deployment of not-critical functions during the off-peak time [5]. These approaches require that developers interested in reducing the energy footprint of their applications must install extended versions of these orchestration and serverless frameworks. But, sometimes this is impossible because serverless frameworks are typically shared by all applications within an organization, and some clients may not prioritize energy savings.

We argue that there are many opportunities for energy savings at the serverless application level [6,7], which have not been adequately explored by previous works. A Software Architect for serverless applications can promote the use of more energy-efficient functions whenever possible. Suppose that an application needs the exact location of a person. Still, several functions differ in accuracy and energy consumption (e.g., GPS, WiFi triangulation or a combination of both methods). Those that are most accurate typically consume a lot of energy, but not all applications require a high level of accuracy at all times. The differences in terms of energy consumption between alternative implementation options are sometimes notable, specially in multimedia and communication related functions. Thus, a possible environmentally friendly design choice could be to select implementations providing the same functionality that result in the lowest energy consumption.

To address this issue, we need to compare the resource consumption of several functions. However, it is not enough to compare only easily measurable parameters, such as CPU time or execution duration. Functions that use fewer CPU resources may still lead to high energy consumption. For instance, these functions might have more intensive memory usage or may engage the CPU and communications more heavily in certain situations, leading to greater energy consumption during idle times. Therefore, it is important to use tools that can accurately measure the actual energy consumption of a serverless function within a specific infrastructure.

It is possible to use both hardware and software tools to measure the energy consumption of a software system. Hardware tools tend to be more precise, but they often cannot measure the energy usage of a specific software process. This limitation arises because multiple processes typically run on the same system, making it challenging to obtain reliable measurements unless one process is isolated from the others. Over the years, the development of software tools to measure software energy consumption has been an active area of research. These energy measurement tools mainly use machine learning algorithms to estimate energy consumption based on known parameters such as CPU usage, memory usage, and other factors at both the application level and the process or function level.

Recently, a new group of tools has emerged that leverages hardware components to calculate the energy consumed by software. Most of these tools use the RAPL [8] hardware component found in Intel or compatible processors developed since 2012. Software energy consumption largely depend on hardware, making this approach an effective way to seamlessly integrate processor characteristics into a software energy measurement tool. In addition, the generated values can be exported via monitoring software like Prometheus [9], enabling other applications to easily capture and analyse them. Typically, these tools offer various metrics that range from energy consumption at the host level to consumption at the process level.

The main goal of our work is to reduce the energy consumption of serverless applications by means of selecting the most energy-efficient function at runtime. We enhance FUSPAQ [10], a solution for the optimal deployment of serverless applications dynamically at runtime, implemented on top of OpenFaaS [11]. Our starting point was the model generation and optimization engine, and we have extended it to use energy measurements and self-adapt the system's behaviour based on them. Initially, we used Scaphandre for monitoring, but now we have redesigned it to work with Kepler, a software specifically designed for Kubernetes. The architecture has also been restructured with components adjusted to a MAPE-K loop architecture to validate it as a self-adaptive system. The contribution of this article is twofold. Firstly, we present a methodology to measure the energy consumed by a serverless application at the function level. We have integrated FUSPAQ with energy measurement software tools and developed a method to make its information available for a self-adaptation process. Our second contribution is an application-level function orchestrator that uses energy measurements to adapt an application composed of serverless functions, considering the evolution of the application's energy consumption. We specifically extended FUSPAQ [10] for using autonomously the energy measurements provided by the first contribution, and also we re-implemented part of this framework to be more energy efficient itself. Our approach (i.e., FUSPAQ) uses an application model based on BPMN 2.0 and feature models (FMs) [12] to define the alternative functions that can compose a serverless application at runtime. It employs a Satisfiability Modulo Theories (SMT) solver to generate the optimal configurations.

In this paper, we focus on incorporating energy metrics into the FUSPAQ process and demonstrate how this information can enable the system to self-adapt a serverless application, making it more environmentally friendly. We have validated the new version of FUSPAQ by means of two illustrative examples, and also have performed a benchmark-based validation addressing different possible scenarios. The results of these experiments show that our approach can save energy from 20% to more than 50% of the application's energy consumption, excluding the energy consumed by FUSPAQ which is also calculated.

The paper is organized as follows: in Section 2, we present some related work, in Section 3, we introduce the technologies and tools used, OpenFaaS and energy monitoring tools; Section 4 provides a brief introduction to FUSPAQ. Section 5 introduces our approach; Section 6 presents illustrative examples of a facial recognition system and a metheorological application, exposing the results to the experiments that we have carried out; Section 7 discusses some threats to the validity of the presented results; and Section 8 presents some conclusions to this work.

Table 1
Energy saving strategies in FaaS.

Energy saving strategy	Approach
Integration with the edge	[6,18–21]
Horizontal scaling	[20,22]
Vertical scaling	[7,23–25]
Energy-aware scheduling	[6,7,18–21,23,25]
Energy saving architectures	[26]

2. Related work

In contrast to cloud environments or microservices architectures, energy efficiency in FaaS architectures is still a challenge. Solutions on energy saving for container-based applications like container consolidation, dynamic resource allocation, green orchestration, service aggregation or container migration cannot be directly applied to FaaS or are less effective due to the dynamic and ephemeral nature of this technology [13,14]. In addition, in the FaaS model, users and development teams do not have control over the infrastructure, which limits the options to save energy or other computational resources [5,15,16]. There are not many works that actually address the reduction of energy consumption in FaaS, some of them are just evaluations [17]. In this section, we analyse how the optimization of energy consumption is approached in Serverless. Table 1 classifies them according to the strategy adopted.

Integrating FaaS with edge computing can significantly reduce energy consumption by bringing functions closer to data sources and final users. The FaaS model facilitates function placement on different locations, such as the edge, but FaaS platforms should consider edge device constraints. The integration with the edge to save energy consumption has been considered in [6,18–21]. The work presented in [18] studies the impact of cold-start techniques on response time and resource consumption (i.e., energy consumption) in serverless applications. The authors propose to extend serverless platforms to the edge using a two-layer model to reduce latency and resource consumption. FADE [6] is an application decomposition framework into serverless functions that consider the edge for deployment. FADE has a model for energy consumption prediction of functions that permits the prediction of the energy consumption of a specific device. Then, the approach selects the node with the predicted lowest energy consumption. Similarly, FoRLess [19] uses a Deep-Reinforcement-Learning approach to select the node to deploy a function that optimizes latency and energy consumption. This approach considers the Fog as a target for the deployment. EERM [20] includes two mechanisms to optimize energy consumption. The approach distributes function execution on different edge devices and can turn on and off the nodes available on the system. The authors demonstrate a 62.5% of energy saving compared to other baselines in a simulated scenario. The work presented in [21] extends the Serverledge platform with an energy-aware policy that can offload functions to the edge or select the most appropriate variants using a trade-off between energy consumption and computation accuracy. The key component of their solution is an Energy Manager that monitors current energy consumption and availability using RAPL. The limitation of this approach is that RAPL accounts for the whole node, while Kepler (the tool used by our approach) accounts for specific functions. Another common aspect is that the approach can change the deployed function at runtime when the battery level of the edge device is lower than a threshold. Experiments show that the approach can extend edge devices' life between 40% and 76%.

The *horizontal scaling* of nodes minimizes idle resources and avoids over-provisioning, which are common causes of energy waste in cloud environments [27]. Several approaches have adopted this strategy in the context of FaaS [20,22]. The work proposed by F. Righetti et al. [22] proposes to extend the OpenWhisk by orchestrating and consolidating service resources for function invocations while maintaining QoS. The authors validate the approach with real function call traces and show an energy saving of 46% compared to non-customized OpenWhisk.

Vertical scaling in FaaS environments reduces energy consumption by optimizing resource allocation for individual functions [7,23–25]. Examples of optimized resources are CPU frequency, memory allocation or bandwidth. Energy Efficient Scheduler [7] uses Dynamic Voltage and Frequency Scaling (DVFS), a power management technique used in modern processors to optimize performance while minimizing power consumption. DVFS adjusts the voltage and clock frequency of the processor dynamically depending on the workload demand at any given moment. Energy Efficient Scheduler selects the most energy-efficient node that makes it possible to meet the performance requirements. EcoFaaS [23] is an energy management framework that uses a similar approach, but it defines pools of processors working in nodes with different CPU frequencies. The framework assigns the serverless function to the one that allows the function to meet a service-level objective defined by the user using the minimum possible CPU frequency. Compared to state-of-the-art systems, EcoFaaS reduces the total energy consumption of serverless clusters by 42% while simultaneously decreasing the tail latency by 34.8%. RAEF [24] is a function-level runtime system that manages the resource allocation of functions. After a careful analysis of energy consumption in Serverless, the authors coined energy fungibility, which considers that a configuration of different parameters of nodes can achieve similar performance, reducing energy consumption. Experiments demonstrate that RAEF can reduce the energy consumption of the same function by 21.2% compared to state-of-the-art techniques while guaranteeing the service level agreement of the functions' 99th percentile latency. Enex [25] is an online scheduler designed to minimize energy consumption while meeting deadlines. Enex spreads computational loads over time, maintaining a constant processing frequency between deadlines. This mechanism reduces peak energy consumption compared to other execution strategies. In addition, it can reduce CPU frequency via DFVS at runtime to reduce energy consumption. The authors compare the energy consumption and latency of Enex to that of Fixed Scheduler and AdaptAvg. Enex achieves a lower energy consumption in all experiments with a minor increase in latency.

Energy-aware scheduling in FaaS involves optimizing the execution of functions in specific nodes to minimize energy consumption. This technique is present to some extent in most approaches and is behind function allocation in the edge. The work presented in [28] focuses solely on this optimization technique. It proposes an extension of the Kubernetes Scheduler that uses machine learning to predict the energy consumption of pods. The authors evaluate the approach in OpenFaaS, obtaining an overall energy reduction of 8%.

Regarding architectural solution to save energy, MicroFaas [26] is an alternate architecture for energy efficiency. This architecture is based on a cluster of single-board computers that act as single-tenant worker nodes with highly reproducible and virtualisation-free execution environments. The authors demonstrate with a prototype that they can increase a factor 5.6 in the energy efficiency of FaaS functions.

The presented self-adaptation approaches for energy-saving in FaaS have a focus on the node in common. The general idea is to select the node where the function consumes less energy. To do so, you can predict the function energy consumption or customize the node to consume the least energy possible while meeting other application requirements. The work presented here focuses on the functions and the applications that they made up. In a few words, we control the function energy consumption at runtime and replace it with another if its consumption is very high. So, it can be easily combined with self-adaptive approaches or even architectural ones.

In conclusion, we did not identify any other works that addressed the orchestration of functions in a FaaS application at runtime by selecting those that consumed the least resources. The most similar work to the one presented here is [21]. As mentioned, this solution can change functions at runtime, replacing them with less energy-consuming variants using a trade-off with other qualities. They replace these when the edge device battery is lower than a threshold. So, the trigger of the replacement is the node's state, not the function's behaviour. In the context of carbo-aware services, approaches [29,30] follow a similar approach. Most solutions focus on optimizing resource allocation and energy-aware container placement, but this decision is taken with information about function behaviour before the deployment. They do not control the energy function at runtime or update the data about its energy consumption for future decisions. The work presented here exploits the knowledge about the energy consumption of serverless functions to dynamically select or replace with the most energy-efficient option for performing a specific task using the information at runtime.

3. Background

3.1. OpenFaaS

OpenFaaS is a serverless framework that provides a FaaS service. It is often used in cloud computing but can also be deployed in other environments like the Edge. In a FaaS system, applications use serverless functions managed by the framework, which is responsible for their deployment and management. These environments, therefore,

allow us to build applications based on microservices in a simple way, providing high scalability and low management costs.

OpenFaaS allows functions to be defined using multiple languages, which are then deployed in Docker containers and orchestrated by Kubernetes. The framework provides an environment in which serverless functions can be defined and subsequently invoked by applications. In addition, it monitors the functions during their execution, sending data to Prometheus continuously to scale the functions if necessary, generating replicas of these and distributing calls between them.

When choosing the framework, our main requirement was that it be free to use. Although OpenFaaS has a paid version, the basic features are included in a free version. The source code of the free version is available and can be configured to suit the needs of a specific application. This feature makes OpenFaaS a suitable environment for testing using our infrastructure. In the context of energy consumption measurement, as the work presented here, this feature is essential because energy data is provided by a hardware sensor that we could not access if we use a private infrastructure. Furthermore, OpenFaaS is an energy-efficient framework [17], making it interesting in this context.

We also evaluate the advantages of this framework over other widely used free frameworks, such as OpenWhisk, Knative and Kubeless [31]. Its Docker-friendly design facilitates deployment in multiple environments, whether in Kubernetes or Docker Swarm, reducing vendor lock-in and improving mobility between platforms [32]. OpenFaaS also stands out for its simplicity and ease of configuration, allowing for a fast learning curve compared to more complex options such as Knative and OpenWhisk [33,34].

OpenFaaS allows for fine-grained scaling control thanks to its integration with Prometheus for metrics, allowing for performance and resource tuning in a more accessible way than in Knative or OpenWhisk, which often require advanced configurations in Kubernetes [34,35]. Furthermore, the OpenFaaS community is active and well documented, making it easy to adopt and support, outperforming Kubeless and providing a more accessible experience than Knative and OpenWhisk, which are geared towards complex enterprise environments [36].

The OpenFaaS architecture is made up of several services that are deployed using Kubernetes, which is the orchestrator that is also used to deploy serverless functions in the infrastructure. On the one hand, the Gateway component provides an interface between the applications and the framework, responsible for processing requests for serverless functions that are deployed in containers in the infrastructure. To do this, it uses the services provided by Kubernetes. The Gateway's operation is not limited to redirecting requests to the container associated with the function, but rather it maintains control of the replicas that have been deployed of each function and routes the requests to them. On the other hand, it has an element that controls the scaling of functions. This is the AlertManager component, which processes data about requests to serverless functions and decides when to create new replicas or destroy existing ones. This data it uses comes from the constant monitoring that Kubernetes performs on the containers and, therefore, on the deployed functions. To do this, Prometheus is used, a system that collects metrics and has a powerful query language that allows large amounts of data to be processed in a simple way.

3.2. Energy monitoring tools

In order to self-adapt a serverless application with the goal of reducing its energy footprint, we need to use a software energy measurement tool. We have tried two of these tools: Scaphandre [37] and Kepler [38]. Both of them use the Intel Running Average Power Limit (RAPL), a well-known tool that provides energy measurements. Initially, we used Scaphandre

3.2.1. RAPL

Most of energy measurement tools rely on RAPL (Running Average Power Limit), a feature designed to monitor and control the power consumption of the CPU and other components on the processor package. RAPL can be found in modern computer processors, particularly in Intel and AMD processors with x86 architecture. The main functionality of RAPL is the continuous measurement of the power consumption of various components, including the CPU cores, integrated GPUs, and memory controller. In addition, RAPL provides mechanisms to control the power consumption of these components by setting power limits. These power limits determine the maximum amount of power each component can consume under normal operation, and they can be adjusted dynamically based on system requirements and thermal conditions. As RAPL reports accumulated energy consumption, it can be used by software tools to obtain the energy consumption of the host, applications or even processes running in a machine. Because of that, it is the method used by most of the software energy measurement tools.

3.2.2. Scaphandre

Scaphandre is a measurement open-source energy consumption tool founded by Benoit Petit in October 2020. Scaphandre is implemented in Rust and works with bare metal server providing a kernel with RAPL support. Scaphandre monitoring agent provides power metrics at the host, socket and process level. It also can be integrated in Kubernetes, in which case containers running on a node are considered processes to measure. The raw metric collected by Scaphandre includes the PID of the container and if the application is deployed in multiple containers is common to add the pod name and namespace each container belongs to. Scaphandre is usually used with Prometheus monitoring tool, which displays the power metrics collected by Scaphandre. Although, Scaphandre looks as a good alternative tool to monitoring energy consumption of software processes, it has many limitations that we will discuss later.

3.2.3. Kepler

Kepler is an open-source energy monitoring tool developed by IBM as part of the Cloud Native Computing Foundation (CNCF). It can collect real-time power consumption and export this information to other applications. It can be used to monitor different elements of the microservice architecture, like the host or even the containers. This is very interesting for our purposes because every function that can be executed in a serverless environment is associated with a container. Furthermore, unlike other tools of this type, Kepler is specifically focused on the Kubernetes environment, which fits perfectly with our system, since OpenFaaS uses it as the orchestrator of the serverless functions.

Using RAPL data to calculate the energy of the different executed processes is not easy because it only measures total energy consumption related to some of a computer's components. That is why Kepler needs to collect data from CPU, like the number of instructions executed or the number of cycles used, to estimate the consumption of specific processes. To do this, Kepler uses eBPF, a technology that allows you to run custom programs within the Linux kernel safely and efficiently, without having to modify the kernel itself or reboot the system. To perform the measurements, it uses kprobes, a debugging mechanism that enables to dynamically collect debugging or performance information while the system is running. Kprobes are integrated with eBPF allowing to execute code when they are activated, so they can be used to capture detailed data of the software that is being executed. Kepler combine these data with the information provided by RAPL to calculate the energy consumed by processes. In addition, Kepler has the ability to make power consumption estimates in case there is no energy measurement hardware available, although for our system we will only work with direct real-time system power measurements.

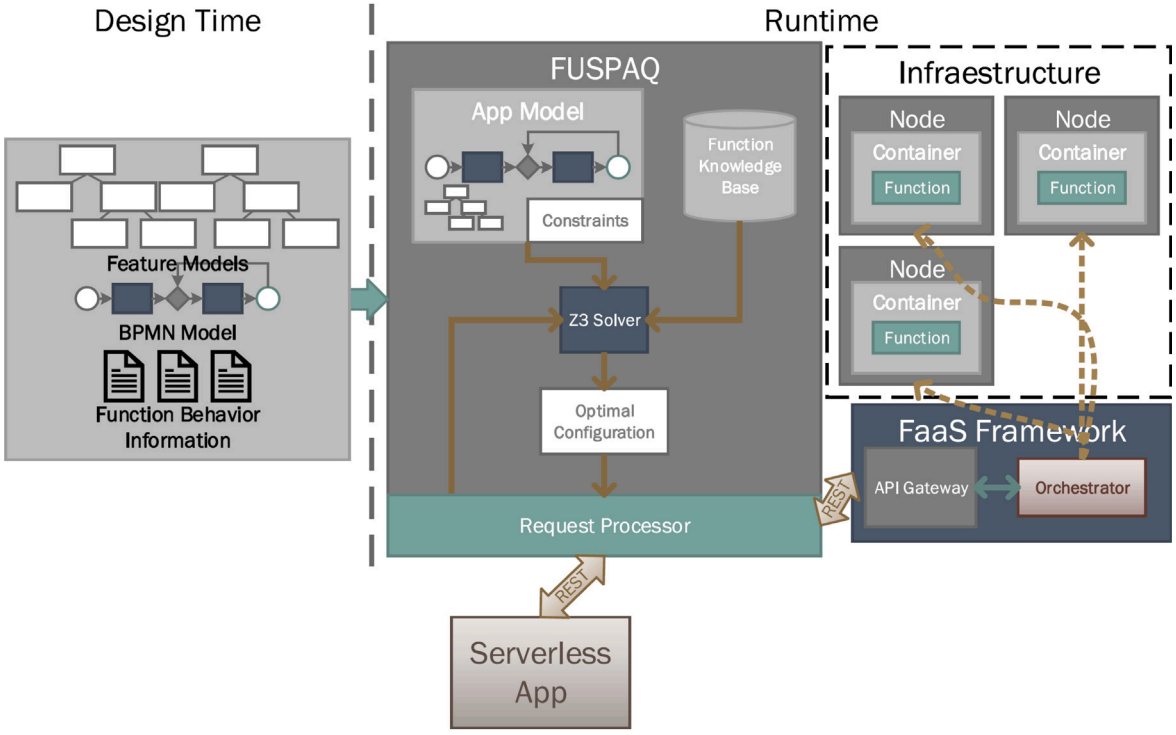


Fig. 1. The FUSPAQ approach.

4. FUSPAQ

As we stated in the introduction, the work presented here is based on FUSPAQ [10], a solution for optimizing serverless functions that uses models at runtime. This solution is based on a preliminary work [39] that proposed a Software Product Line (SPL) approach to select the best serverless functions to deploy, considering any service parameter quality. In contrast, FUSPAQ follows a runtime approach based on Dynamic SPL (DSPL) [40] that exploits alternate implementations and configurations of the activities that make up the serverless app.

Our approach comprises two phases: design and runtime (see Fig. 1). The first step at design time is to model the serverless application using a BPMN 2.0 model. BPMN models comprise activities and gateways representing the application's decision points. This model should be transformed to be used at runtime by FUSPAQ. So, we automatically generate a feature model of the app from this BPMN model. The activities are transformed in the features of the feature model, and the relationships between activities are transformed into relationships between features. More details of the process are given in [39].

The optimization of the serverless app occurs at runtime when the application is deployed or when the application requests a change in some quality of service value. This optimization process exploits alternate implementations for activities in the form of serverless functions and configuration options. So, FUSPAQ needs information about the serverless functions that can implement the activities that comprise the app. Developers introduce this information in the *Function Knowledge Base* that generates feature models with the alternate implementations of each activity. These models consist of a feature representing the activity, with children for each implemented alternative. At this point, we have several models that are combined using Software Product Lines refactoring techniques [41] to generate the *App Model*.

As we stated, the *Function Knowledge Base* includes information about the functions that can perform activities, but it has additional information for the optimization process. First, it contains information about the quality of service parameters of each of the functions, that is, execution times, latencies, etc. In addition, for each function, the possible configurations under which they can be executed are also included since they constitute an additional source of variability. For example, if a function performs a specific image processing, it is not the same if it works with one resolution or another. It is essential to remember that the optimization process does not choose functions solely based on their energy consumption. Our system performs an optimization process on the application model in which we have information about each function's energy consumption and other aspects such as execution time or user experience. In this way, the system does not choose functions solely based on their energy consumption, which would effectively always result in selecting the lowest-consumption versions, neglecting other important aspects such as quality of experience. Instead, our system integrates all these variables into the model and optimizes it based on the specified parameters (i.e., QoS values). The optimization process uses the generated feature models and parameters stored in the Function Knowledge Base (the K of the MAPE-K loop). The Z3 solver uses all this information and QoS constraints to generate the system's optimal configuration. This process makes a tradeoff between all the optimization variables, as presented in [10].

FUSPAQ is the intermediate between the *Serverless App* and the *FaaS Framework* at runtime. The component in charge of the communication between these three is the *Request Processor* that uses REST. REST makes the natural integration of FUSPAQ with FaaS platforms possible because they are commonly used to call serverless functions. FUSPAQ receives two kinds of requests: the regular call of a serverless function, which is forwarded to the FaaS framework, or the adaptation of the application to meet a QoS. In this last scenario, the *Z3 solver* uses the *App model* and the information contained in the *Function Knowledge Base* to generate an optimal configuration of the *Serverless App* that meets the QoS requested. Then, the *Request Processor* sends the request to the FaaS platform that instantiates the functions deployed in containers in the nodes that are considered suitable.

5. Energy-driven self-adaptation for FUSPAQ

The previous version of FUSPAQ optimizes serverless apps when requested. In this contribution, we present a new version of FUSPAQ[1] (see Fig. 2) that can analyse the state of the serverless app and autonomously configure it at runtime. Specifically, we focus on self-adaptation to optimize energy consumption. Here, the challenge is twofold: firstly, we need to measure the energy consumption of serverless functions, and secondly, we have to devise an approach to use effectively this energy measurement for the self-adaptation of the app. So, we have evolved FUSPAQ with new components: the *Energy Consumption Monitor* and the *Energy Consumption Analyser*. With these two components, we complete the classical MAPE-k loop of the autonomic computing approaches [42]. Our planner is the *Z3 Solver*, the executor is *OpenFaaS*, and the knowledge is composed of the *Function Knowledge Base* and the *App Model*.

5.1. Energy consumption monitoring

To make possible the work of the *Energy Consumption Monitor*, we have integrated FUSPAQ with OpenFaaS, a serverless framework, and Kepler, a tool to measure energy in containerized applications.

In a previous work [43], we opted for Scaphandre because it introduces less overhead than other similar tools that can work at the process level, like PowerAPI [44]. Finally, we decided to use Kepler [38], a more recent and powerful tool that is better suited to our system and provides more accurate results.

The reasons for discarding Scaphandre are as follows. When studying the system's overall energy consumption, we observed that Scaphandre underestimates the functions' consumption since their aggregate energy consumption is lower than the total. It must be said that the node consumption will always be lower than the aggregate of the serverless functions since it executes other operating system processes and the OpenFaaS core itself, as well as Prometheus. However, the values obtained by Scaphandre for the containers are too low compared to those of other tools such as Kepler. We have also verified it through a comparative study of the work of Scaphandre and Kepler for multimedia serverless applications, as a part of a master thesis [45]. For this reason, we decided to work with Kepler. One of its advantages is that it works with energy values, which we use to compare functions. In addition, the RAPL registers store accumulated energy, thus avoiding unnecessary conversions. Furthermore, the fact that it is specifically designed to work with Kubernetes translates, in our experience, into more accurate results for the reasons mentioned above.

OpenFaaS deploys Docker[2] containers for each function and orchestrates them using Kubernetes.[3] Therefore, to determine the consumption of the serverless functions, we must know the consumption of the associated containers. Kepler can obtain measurements per container, allowing us to get the consumption of serverless functions. Each replica has an associated container corresponding to a running process on the node.

Kepler provides a Prometheus[4] exporter, so we can use this tool to store and monitor energy measurements. Prometheus is a powerful software used extensively for containerized applications and is also used by OpenFaaS. We use the same instance of Prometheus deployed by OpenFaas to avoid the consumption of unnecessary energy. We configure Kepler to serve the data to it. To do this, we custom-installed OpenFaaS, setting Kepler as an additional data source.

The *Energy Consumption Monitor* procedure is as follows. Firstly, it obtains the identifier of a container associated with the function of interest to us by querying Kubernetes. This is necessary since, although Kepler allows us to search for functions by name, there may be different containers, active or not, associated with the same function. This procedure must be repeated for each container associated with each function replica. Adding the energy consumption of all the replicas, the *Energy Consumption Monitor* obtains the consumption of the function. More details of this procedure are given in [43]. To make the work of FUSPAQ more efficient, *Energy Consumption Monitor* is not continuously monitoring the functions, it only takes the energy consumption after each call to a function by the serverless application.

We aim to obtain stable energy values that allow us to know in real-time if one function consumes more energy than another. However, the energy data we obtain from the monitoring are variable. Obviously, if we carry out measurements for a long time, like hours or days, we can obtain more precise values, but this would not help us to compare consumption in real-time. Therefore, the data provided by Kepler cannot be used directly for energy consumption measurement in the context of self-adaptation. So, we opted to use the mean value of the measurements. Since power consumption can vary over time, the way to achieve this is by calculating the moving average. This data is obtained using energy values returned by *kepler_container_joules_total*. We calculate the total energy increase in that period and divide it by the number of function executions during that interval (this information is provided by OpenFaaS via Prometheus), estimating the individual average consumption of a function. In reality, the consumption value we obtain is not exactly the consumption of the function since we would have to subtract what the container has consumed. However, our interest is not to calculate the precise data of the energy consumed by every function, that is, the consumption of the code executed by the function, but rather to compare the differences in system consumption produced by the execution of the functions, so we will directly use this data that we have obtained about consumption.

Since we want to obtain lines that are as little variable as possible, we thought about calculating trend lines. One method often used in financial analysis to calculate stock quote trends is the double-moving average method. Applying this method, we can improve the variability of the energy measurements compared to the average data using the same amount of time. Then, if we use a second moving average to the average data of 5 min, we consume 10 min, but we improve the data obtained for the 10 min moving average. In our experiments with Scaphandre, we reduced the variability of energy measurements to close to 26% using this method with 5 min intervals. More details of these experiments can be found in [43].

The variability of the energy values obtained by Kepler is lower than that obtained by transforming the power values provided by Scaphandre. By calculating the moving average in 1 min periods for the same set of functions used in the mentioned tests, we get variations of less than 15%. In addition, we significantly reduce the time required for data smoothing compared to that mentioned for Scaphandre.

As explained above, we can further smooth out the energy curve by calculating a second average over the estimated energy values, which will help us to avoid overlapping consumption measurements for different functions without significant differences in their consumption. To do this, we calculate the accumulated energy after some time and divide it by the number of executions in that period. In our experiments, we have used a period of 1 min, obtaining very stable curves with low variation as mentioned above (see Fig. 3). At this point, we obtain the energy consumed by each of the functions at each instant, with which we would be able to compare them with each other.

5.2. Energy Consumption Analyser

The *Energy Consumption Analyser* uses the information about the energy consumption of the functions stored in the *Function Knowledge Base* and Event Condition Action (ECA) policies to request to the *Z3 solver* a new configuration of the *Serverless App*. The main challenge of the

[1] Source code available at https://github.com/pserranouma/fuspaq/tree/v3.

[2] https://www.docker.com/.

[3] https://kubernetes.io/.

[4] https://prometheus.io/.

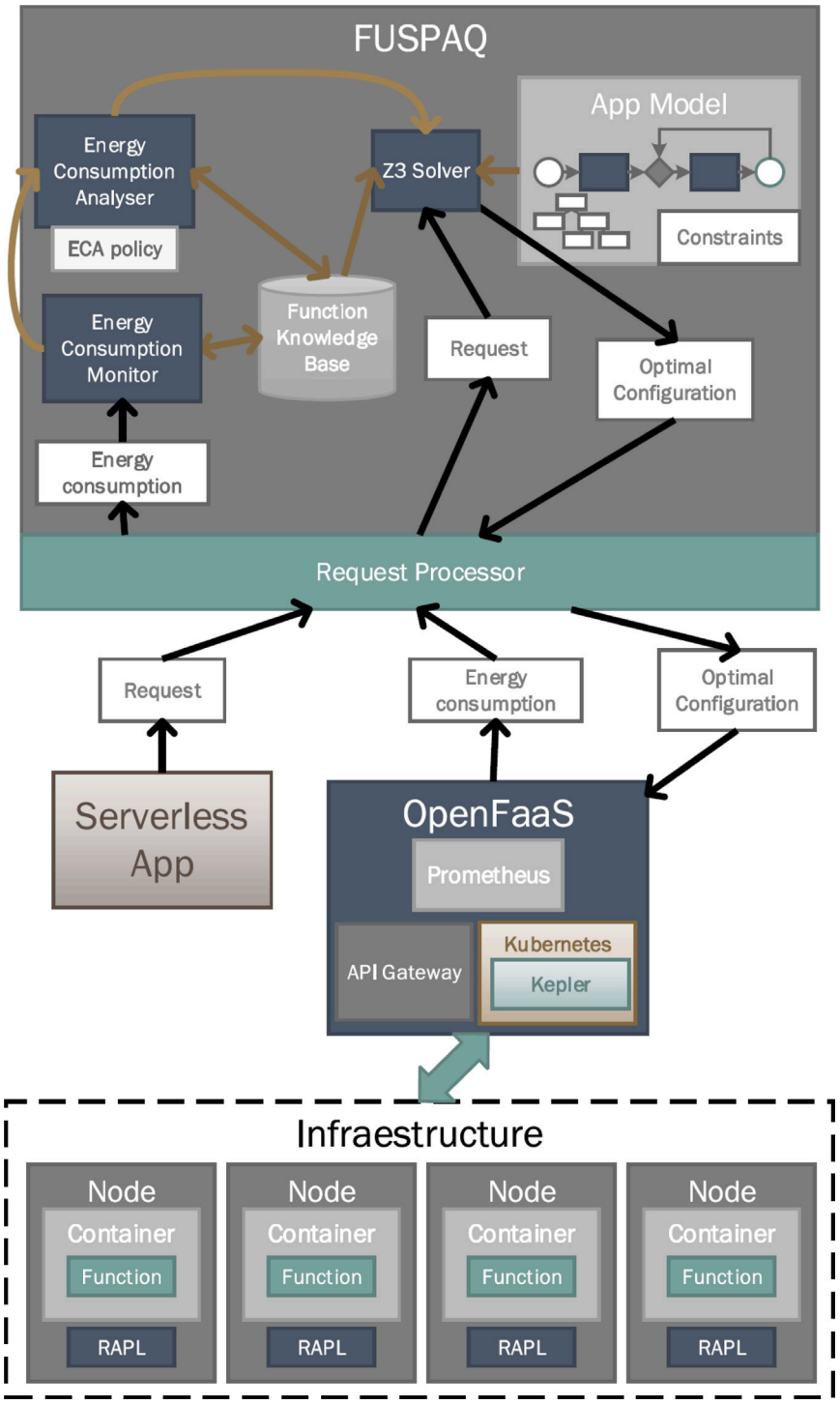


Fig. 2. FUSPAQ for self-adaptation driven by energy.

analyser component is to detect when it is necessary to self-adapt the system. As we stated, after several experiments, we observed significant variability in the energy measurements. These fluctuations can cause unnecessary application adaptations. Therefore, it is necessary to act on the data itself and on how the self-adaptation process is triggered. To face this challenge, we process the data provided by Kepler and use a particular type of ECA policy to trigger the self-adaptation process.

One of the challenges of using averages for self-adaptation is that the data will not be sufficiently representative until we have enough samples. Therefore, we consider a setup time during which we do not perform system reconfiguration operations. This setup time limits the reaction time of FUSPAQ to changes but prevents continuous, unnecessary changes from occurring. If you want to optimize other quality of the system like execution times, this procedure could be unacceptable. Still, ensuring that the system adapts better is more convenient when the objective is to save energy. Therefore, a setup time of 1 min seems quite acceptable to us. After that setup time, the *Energy Consumption Analyser* can start working. At runtime, the *Energy Consumption Monitor* notifies the *Energy Consumption Analyser* that there is new information about the energy consumption of any function. In this way, this component only works when it is necessary. In any case, to avoid the initial setup time, the repository may contain information about the values

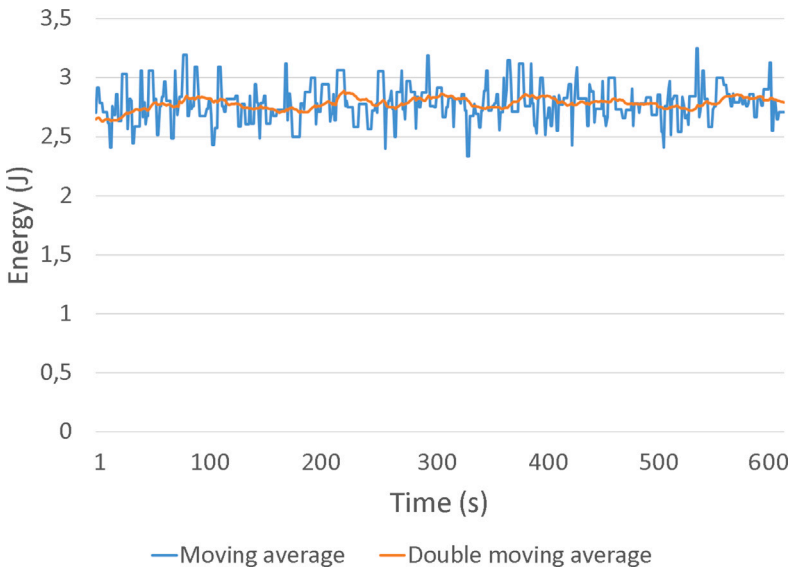


Fig. 3. Experiments on moving averages for energy data obtained from monitoring.

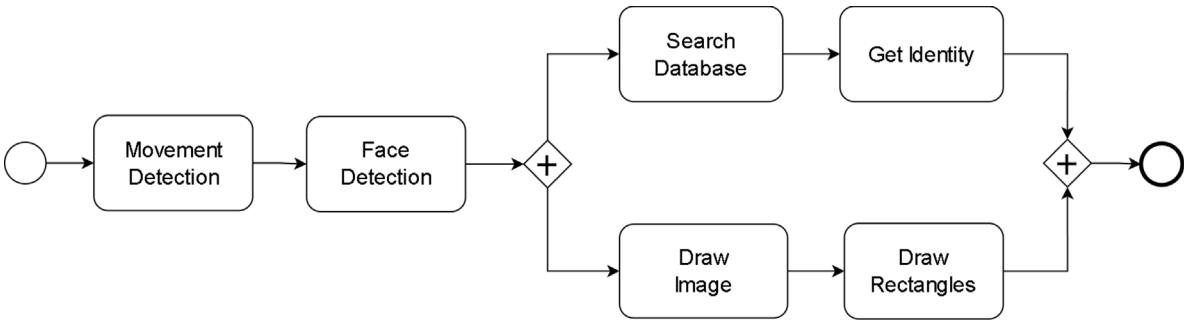


Fig. 4. BPMN 2.0 model of the face recognition application.

measured in previous executions or tests of the functions to compare the consumption of the different implementations.

The *Energy Consumption Analyser* uses ECA policies with thresholds. The component launches the reconfiguration process when the energy consumption of a function is higher or lower than the threshold. The issue is the behaviour of a function is so unpredictable that it is tough to set the threshold for self-adaptation at design time. One simplification we have made is using percentages instead of absolute values. However, even choosing a percentage poses challenges because the variability in energy consumption can differ from one function to another, as we have check in our previous experiments [43]. So, we decided to compute this threshold at runtime, considering the information on the energy consumption of the functions. To calculate these variable thresholds, we use the same periods as for moving averages; we calculate the average, the maximum and minimum values, and we find the maximum percentage of variation between the average value and said extremes. We consider the greatest of these two results and set the threshold, adding a certain margin. Every time we update the energy values associated with a function, we will also update the considered threshold for that function.

6. Validation

To test the feasibility of our approach, we use as illustrative examples, an application for face detection and a second one for obtaining and processing meteorological data. In addition, we have conducted a benchmark analysis of FUSPAQ's scalability regarding energy consumption, savings, and the time needed for the optimization process. The goal of these experiments is to show the benefits in terms of the reduction of energy consumption. In addition, we test the energy consumption of self-adaptive FUSPAQ.

6.1. Illustrative example 1

The application for face detection (see Fig. 4) analyses images taken by a camera and looks for significant changes. If they occur, it considers that a movement has occurred and will go to a face detection stage. In this way, the system only performs detection tasks when it is needed, and it can save computational resources. After detecting faces, the system will have obtained a series of rectangular areas in which it has detected a face. The next step is to extract patterns from each area and then search a database for the most probable identities. Finally, the recognition results will be shown on the screen. In parallel, the image is drawn, and rectangles are drawn on each detected area to highlight faces in the image.

Following the procedure described in Section 4, we have implemented this application using serverless functions for each task. The general idea is to have alternate implementations with different configuration options to exploit them to self-adapt the application. All this variability is modelled in several variability models (see Fig. 5) that FUSPAQ refactors and joints with the BPMN model to create the DSPL that uses the Z3 solver to generate the optimal configurations. We can see that the operations that are carried out successively in the workflow appear grouped by tasks. In this way, *Detection* is a task composed of the operations *Movement Detection* and *Face Detection*, *Processing* is composed of the subtasks *Identify* and *Draw* , and, in turn,

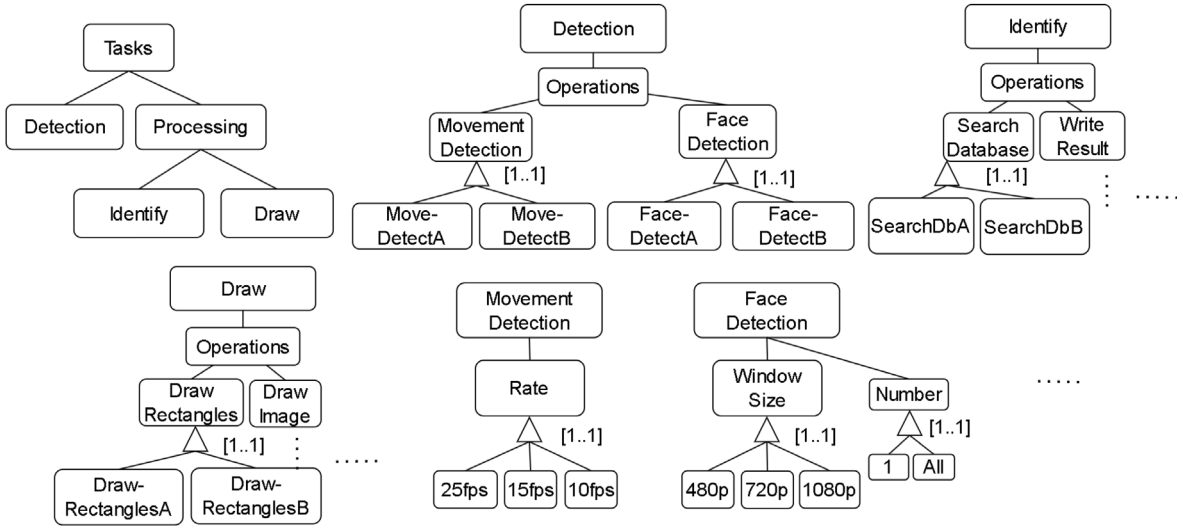


Fig. 5. Variability models for the face recognition application.

the *Identify* task performs the operations *Search Database*, where we have the registered patterns of the faces, and *Get Identity*, which extracts data from the detected face and compares it with the stored data. For these tasks, we use Python's *Torch* library. For its part, the task *Draw* is made up of the operations *Draw Rectangles* and *Draw Image*. Each of these relationships is reflected in the corresponding feature model. Finally, the rest of the feature models refer to possible configurations of some functions. For example, motion detection can be done using different frames per second. This will increase the energy needed, but may also decrease the quality of the user experience.

Our experiments focus on the energy consumption of the face detection task. We have used two different implementations that are available in the OpenFaaS store for this purpose. The first implementation of the face recognition function uses the OpenCV library, which is widely known and used for image processing. The second uses Pigo, a Go-based library specialized in functionality related to facial recognition. We have studied the power consumption of these functions and computed the average and the double average (see Fig. 3). Our experiments show that the Pigo version is more energy-efficient than the OpenCV version (see Fig. 6). The two functions do not have to behave the same in all aspects. For example, their success rate in certain conditions may differ. Therefore, each function may have associated energy consumption and other quality of service parameters, which we consider when selecting the best function to meet the system requirements. However, to verify the operation of our case study, we focus on energy consumption. We store data for all features used in our function knowledge base, including the consumption of the two alternative implementations of the face detection function. We have previously carried out these measurements on the same machine and under the same conditions. These values are only initially necessary so that our system can compare and use both functions to optimize the serverless application. As mentioned, other factors related to the quality of service also intervene in this process, so the optimization result is not reduced to choosing the functions with the lowest energy consumption. To properly compare those stored values, given that consumption is variable over time, we decided to store the average values over a sufficiently long period, in this case, one hour.

6.2. Illustrative example 2

In this second case, we have chosen a more complex and real application based on collecting and processing meteorological data using machine learning techniques. The application, whose BPMN diagram is shown in Fig. 7, consists of two distinct parts. On the one hand, it displays photos that users upload to a database. The system obtains the coordinates of the user's location (*GetLocation*) and searches for locations close to it within a radius entered by the user (*GetNearLocations*). After this, it obtains a list of the images users have uploaded in those areas (*GetImages*). Finally, it selects several random pictures and shows them to the user (*RenderImages*). On the other hand, the application collects data to make a prediction and display it alongside a map of the area. The application can obtain the data in two different ways. One is based on an external service that uses data from sensors that users enter into the system (*GetSensorData*), and another in which we obtain the data (*GetModelData*) and process it using different machine learning techniques (*ProcessModel*). Finally, we render a map of the area and overlay the prediction made (*RenderMap*).

For this example, we have increased the number of available implementations with different energy consumption. Specifically, for the data processing function, we have three alternatives: one that uses linear regression, another that uses random forests, and the last one uses neural networks. We have also considered alternatives for the map rendering function. We have used two techniques: the first one using Python's *Folium* API and the second one generating the map from a static image. In Fig. 8, we can observe the energy consumption of the different implementations of both *ProcessModel* and *RenderMap*.

When measuring the energy of the *ProcessModel* function, our experiments show that the implementation that consumes the most energy is the one that uses the neural network, which has an average consumption of around 0.73 J. The one that uses random forest consumes about 0.51 J, and the one that performs linear regression, around 0.43 J, as can be seen in Fig. 13(a). Obviously, the accuracy of the results will not be the same, nor will the time spent, which could influence the user experience. Therefore, it is possible to adjust this parameter in the *Function Knowledge Base* so that it is taken into account when our system carries out the optimization process. If we consider a scale of 1 to 10 for the user experience, we could assign, for example, a value of 6 for the implementation based on linear regression, 8 for the neural network, and 9 for the random forest. By telling FUSPAQ to optimize not only by trying to minimize the energy consumed but also by trying to maximize the user experience, it will carry out a multi-objective optimization process taking both parameters into account. Regarding the *RenderMap* function, the differences in energy consumption are greater, with the version that generates the map from a static image being the most efficient (see Fig. 13(b)). However, it is important to keep in mind that downloading it also generates energy consumption in an external server that is not quantifiable in our system, so the actual total energy savings would not be so high.

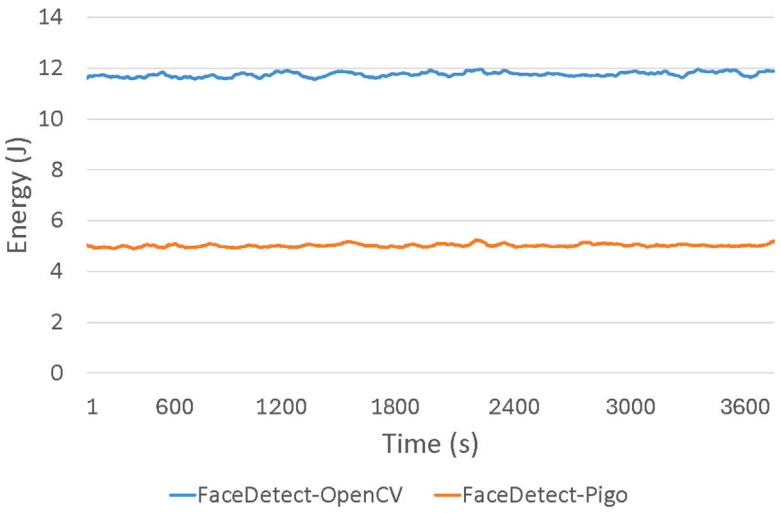


Fig. 6. Evolution of the energy consumption per execution of the face detection function using OpenCV and Pigo versions.

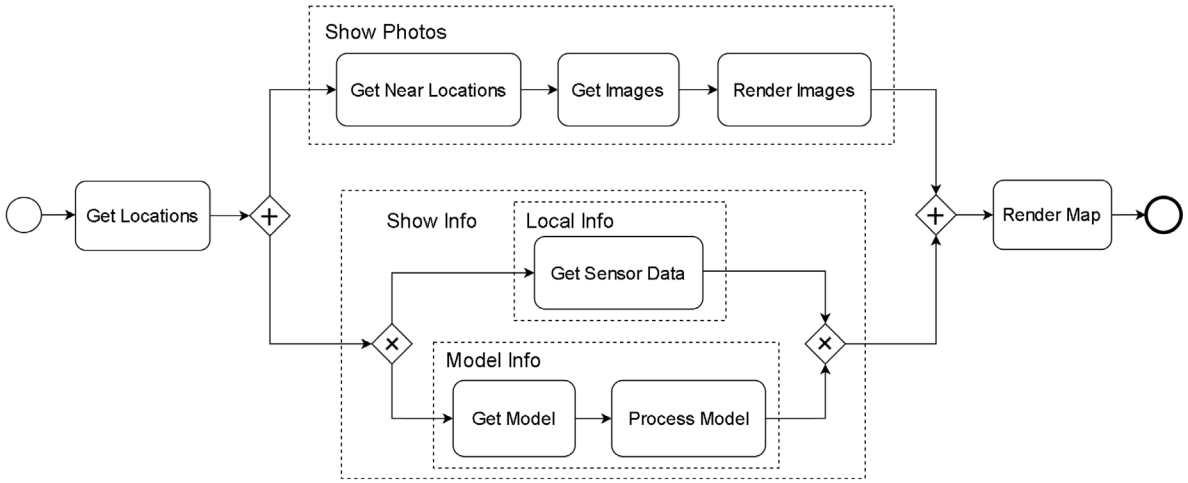


Fig. 7. BPMN 2.0 model of the meteorological application.

Model processing functions make predictions based on previously trained models. However, we have also measured the power consumption of these training processes. In this case, the differences between functions are very significant. The implementation of neural networks has the highest consumption (around 11.5 J). Random forest consumption was around 3 J, and with linear regression, the measured consumption was 0.5 J. Therefore, if our system performed this type of processing in real time, the savings would be even greater by choosing simpler models. However, since models are typically trained initially rather than continuously, we have conducted the following experiments, considering only the prediction process.

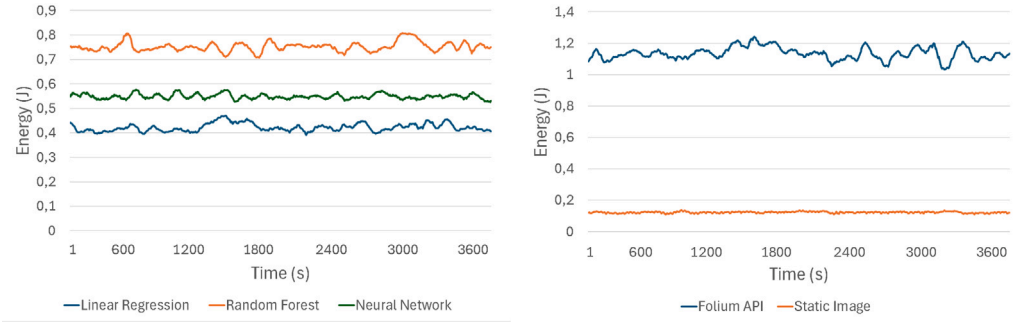
6.3. Experiments with the illustrative examples

The experiments presented here aim to verify the self-adaptation capabilities of FUSPAQ. We compare the energy consumption of the applications in three scenarios. The first one is designed to compare the energy consumed by the optimal and non-optimal configuration, that is, without performing self-adaptation. The second one is with self-adaptation and initial information about the energy consumption of the functions, and the third one is with self-adaptation and without the initial information. We have measured the energy consumption in these three scenarios for 5000 executions of the application and compared them. Because the execution times of the different implementations considered could be different, to compare the results we have performed a certain number of executions instead of running the application for the same period of time. We have chosen 5000, since it results in a sufficiently high execution period of more than 1 h. Also, to recreate a more realistic environment, we have used 10 concurrent requests. To carry out the experiments, we used Kepler v0.7.11, OpenFaaS v0.27.12, and FUSPAQ, which is programmed using Python 3 and C. PCs with different specifications have been used as hardware: Intel i5-7400, 3.00 GHz, 24 GiB of RAM, and Intel Core Ultra 7 155H, 3.80 GHz, 32 GiB of RAM.

6.3.1. Experiment 1

As we stated in Section 4, the first version of FUSPAQ selected an optimal configuration for the Serverless Application at deployment or under request, taking into account the information contained in the app model and some quality information. For the first illustrative example, as the Pigo version of the face detection function is the more energy efficient according to our stored data, this is the one selected by FUSPAQ. In any case, to have a baseline, we also measured the energy consumption of the OpenCV version. Comparing both results, the energy consumption of the application using the Pigo version of the face detection function is clearly lower, about 45.3%.

Considering the second illustrative example, the differences are more significant for the model processing function. The energy savings of the application when choosing the neural network version of *ProcessModel* instead of the random forest version is approximately 8.1%,



(a) Energy Consumption of ProcessModel implementations

(b) Energy Consumption of RenderMap implementations

Fig. 8. Evolution of the energy consumption per execution of model processing and map rendering functions.

and with linear regression it is approximately 9.5%. If we also consider alternatives for the second function mentioned, i.e., *RenderMap*, when the most energy efficient version is selected, energy savings are of 26.8% for neural network and 28.8% for linear regression. As mentioned above, considering quality of service parameters, such as user experience, is common, which means that FUSPAQ does not always choose the most energy-efficient versions. Therefore, these are maximum savings values and do not necessarily coincide with the actual savings values during the application execution in a production environment. However, this does not mean FUSPAQ would never choose the linear regression version. Since our system is configurable in real time, it would be possible, for example, to reduce user experience requirements at times when energy is more expensive.

6.3.2. Experiment 2

In the second experiment, we deploy the same applications with self-adaptive FUSPAQ. For the first application, at deployment, the optimal configuration is to use the Pigo version. However, this function worsens its behaviour from an energy point of view after a while. When the system detects this situation, it will launch a reconfiguration process. As the measured values exceed those of the alternative implementation, FUSPAQ generates a new configuration that, in this case, chooses the OpenCV implementation for the operation of face detection. We can verify that the system behaves as expected. As shown in Fig. 9, after a stable start, the selected function for performing face detection increases its energy consumption, causing the energy consumed by the application to increase significantly. When this exceeds a certain threshold, the alternative implementation is chosen when reconfiguring, which causes the abrupt drop that can be seen in the figure. To measure the impact of this reconfiguration from an energy point of view, we have also simulated a situation in which the system does not use Self-Adaptive FUSPAQ, consuming the high energy level reached by the function whose behaviour has changed. The result is also shown in Fig. 9.

A function's energy behaviour worsening at runtime is possible due to various factors. For example, it may be caused by problems in accessing files, databases, or network access. These situations can occur when several applications simultaneously use these elements or when the number of replicas of the same function increases, and they compete for the same resources. It is even possible that some functions behave worse if executed on less appropriate hardware, such as those that use the GPU intensively and are forced to use the CPU if they do not have sufficient GPU capacity. In our case, to simulate this variation in the function's behaviour in our experiment, we have reproduced a situation in which it was necessary to repeat network access operations frequently. Energy consumption measurements for 5000 executions are 74 kJ in case of self-adaptation and 79 kJ in case self-adaptation had not been carried out, approximately 6.8% more.

For our second application, we have generated new versions of *ProcessModel* that perform both the prediction and pre-training processes if the server where the model data is stored is unavailable. As we mentioned at the beginning of this section, training is expensive and is not usually done in real time. In this way, we can force an apparent worsening of the function's energy performance. It so happens that, although the neural network consumes less in its prediction than the random forest, its training is considerably more energy-intensive, as detailed in Section 6.2. In Fig. 10, we can observe that at the beginning energy consumption is low because our system is executing the application using the neural network version of *ProcessModel* function, but a certain point (when we disconnect the server with the trained models), there is a notable worsening of that function that was being executed (the implementation of the neural network). However, since the random forest performs better during training, it will be the one selected by the system when it is reconfigured, reducing the energy consumed, which is higher than when trained models where used, but much lower than before reconfiguration.

6.3.3. Experiment 3

Our third experiment focuses on the work of FUSPAQ when it does not have initial values on energy consumption. Initial measurements are convenient because the system can generate an optimal initial configuration, but since the system has self-adaptive capabilities, it is unnecessary. Therefore, you can run the system starting from zero values and wait a while for it to store the values measured during that time. As we consider a setup time, as soon as it finishes, the functions will take the values measured so far, which will already be stable. Each operation with different associated functions will update its data each new setup time since a value of 0 in energy will cause each implementation to be chosen successively after the respective setup times. This approach is also advantageous over isolated measurements since the values are initialized considering the specific infrastructure conditions on which the application is deployed. The drawback is that a global setup time is required equal to the maximum number of alternatives times the setup time. If the system chooses an expensive function in terms of energy consumption, it returns to the best one after a new setup time. However, since it is enough to do it once, it is better than having to perform measurements for each implementation one by one separately. In Fig. 11, we can check the results of an execution without previously stored energy values.

As we can observe in Fig. 11, the system had opted for the first stored implementation of the face detection operation because it has no values to compare with, and which, in this case, is the least energy efficient, the OpenCV version, so the system is consuming around 15 J per execution of the application. This selection is not a problem since after the setup time, the system chooses the most appropriate alternative in terms of energy consumption, i.e., Pigo, decreasing energy consumption immediately. After a new setup interval for the face

Fig. 9. Energy consumption of the Face Recognition Application with and without self-adaptation.

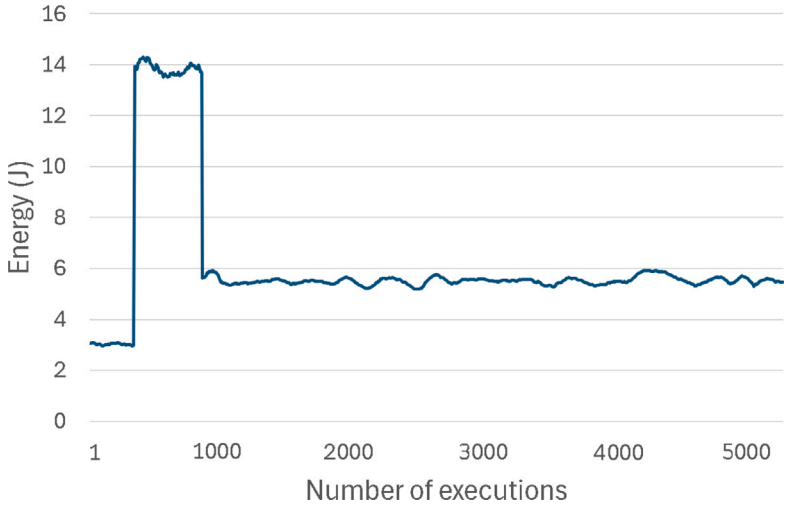


Fig. 10. Energy consumption of the Meteorological Application with self-adaptation.

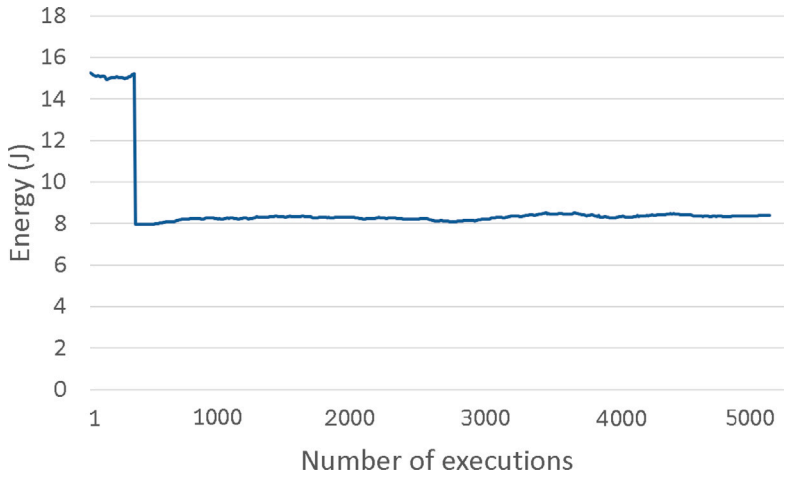


Fig. 11. Evolution of the Face Recognition Application energy consumption, when there is not initial information at disposal.

detection task, the system assigns the measured energy value to this selected Pigo implementation, so from that moment, none of the energy values associated in the repository with the functions that perform that operation remains at zero. However, as it is lower than that previously obtained for OpenCV, the reconfiguration process is not launched, and there is no change in the selected implementation, so the

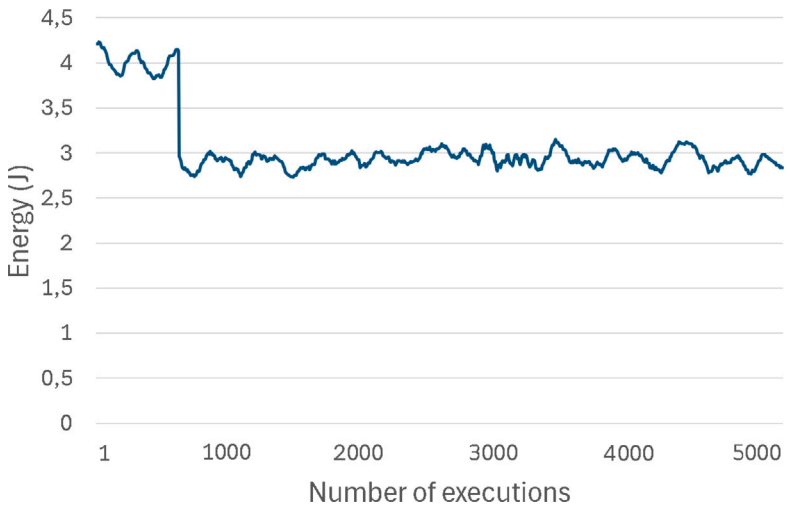


Fig. 12. Evolution of the Meteorological Application energy consumption, when there is not initial information at disposal.

measured energy values remain stable (around 14.7 J per execution of the application).

For the second illustrative example, Fig. 12 shows a situation in which the random forest implementation is initially chosen for model processing and the *Folium* implementation is chosen for rendering the map, generating higher energy consumption than other options. Once the setup time has elapsed, the reconfiguration process starts, and FUSPAQ selects lower-powered implementations for these functionalities, the one that uses neural networks and the one based on a static image. Our approach considers user experience in this experiment, so instead of selecting the linear regression function (the one with the lowest energy consumption), it selects the neural network function. Regarding the *RenderMap* function, FUSPAQ selects the version with the lowest energy consumption, so it does not change after the reconfiguration process.

6.3.4. Comparing energy footprint in two machines

One of the challenges we face when working with services that can be deployed on different nodes is the heterogeneity of the hardware that makes up our infrastructure. To study the implications of this on our system, we repeated the consumption measurements for the applications we developed for the illustrative examples, using different hardware. We used the second of the hardware configurations described at the beginning of this section. The results show slight differences compared to those obtained with the other hardware. The values obtained, detailed by function, can be seen in Table 2. We can observe that the measured consumption is somewhat lower when using the second hardware configuration, varying by about 2% or 3%. However, this does not affect the operation of our system, as we explain below. Infrastructure heterogeneity can be considered another source of variability in energy measurements. Using the same hardware, the energy consumption of functions usually fluctuates by a certain percentage. This percentage likely increases when we have more hardware options to deploy functions. The advantage of our system in this regard is that it does not use fixed values to initiate the reconfiguration process, but, as explained in Section 5, it takes into account the variability in the measurements for each function when calculating the thresholds. Therefore, the system will continue to function similarly but with wider margins.

6.4. Benchmark-based validation

To study the scalability of our solution and the potential benefits, we have developed a FaaS system generator to generate benchmarks for the analysis of energy consumption. This generator can set up applications composed of a specific number of operations. In addition, we can establish the number of functions that could perform those operations. Finally, we can configure the functions to consume different amounts of energy.

Table 2
Mean energy consumption (in Joules) and mean duration (in Seconds) per execution of the implemented functions for both illustrative examples, measured with different hardware.

App	Functions	Mean energy (J)		Mean duration (s)	
		HW1	HW2	HW1	HW2
App 1	MoveDetect	0.713	0.695	0.742	0.735
	FaceDetect-OpenCV	11.984	11.833	0.460	0.453
	FaceDetect-Pigo	5.219	5.038	0.931	0.907
	SearchDB	0.369	0.361	0.033	0.032
	GetIdentity	1.235	1.208	0.407	0.395
	DrawRect	0.405	0.392	0.830	0.823
	DrawImage	0.257	0.250	0.363	0.348
App 2	GetLocation	0.671	0.695	0.082	0.089
	GetNearLocations	0.391	0.387	0.034	0.034
	GetImages	0.372	0.367	0.034	0.032
	RenderImages	0.420	0.412	1.060	1.054
	GetSensorData	0.271	0.270	0.081	0.080
	GetModel	0.280	0.272	0.085	0.084
	ProcessModel-LR	0.431	0.426	0.044	0.041
	ProcessModel-RF	0.737	0.729	0.070	0.065
	ProcessModel-NN	0.511	0.498	0.047	0.045
	RenderMap-Folium	1.028	1.006	0.068	0.063
	RenderMap-Static	0.139	0.147	0.604	0.610

We have generated four applications for our experiments with 3, 10, 100, and 1000 operations. Each application has a different number of functions that can perform those operations: 1, 10, 100, and 1000. These variants of the FaaS application allow us to study the scalability of the FUSPAQ optimization process. To study the potential benefits in terms of energy consumption, we have two versions of these applications: one (*ST1*) in which it is not possible to save energy using alternative functions, and another (*ST2*) in which the energy consumption of the alternative functions ranges from 50% to 100% of a specific base consumption. We have performed 100 executions for each variant, obtaining the average consumption and the standard deviation of each of them. Results of these experiments regarding the time to select a function and the energy consumption are shown in Tables 3–5.

Firstly, we have analysed the scalability of FUSPAQ in terms of energy consumption and time for the optimization process, i.e., the time to select the most appropriate function to operate. In our experimental setup, the number of operations that made the applications and the number of alternative functions vary in the ranges mentioned below.

Table 3
Energy consumption (in Joules) and duration (in Seconds) of the FUSPAQ optimization process for benchmark applications.

Operations	Functions	Energy (J)		Duration (s)	
		Mean	Std.Dev.	Mean	Std.Dev.
3	1	1.16	0.58	0.01	0.008
	10	0.73	0.18	0.04	0.011
	100	3.49	0.14	0.24	0.01
	1000	26.56	0.08	2.16	0.007
10	1	2.53	1.05	0.02	0.009
	10	2.41	0.31	0.09	0.011
	100	8.16	0.07	0.75	0.008
	1000	86.75	0.18	7.27	0.015
100	1	3.66	0.43	0.1	0.013
	10	7.02	0.12	0.77	0.013
	100	81.68	0.14	6.94	0.007
	1000	915.05	0.1	76.72	0.029
1000	1	6.76	0.75	1.02	0.017
	10	39.66	0.28	7.82	0.023
	100	863.23	0.27	75.41	0.021
	1000	9143	0.32	770.3	0.059

As we can see in the results shown in Table 3, energy consumption and time increase with the number of operations and the number of alternative functions. So, the experiment with the lowest energy consumption is the one with 3 operations and one function, and the one with the highest energy consumption and duration is the one with 1000 operations and 1000 functions. The results are pretty stable, with a standard deviation of less than 1 J and 0.06 s for all experiments. For realistic scenarios, i.e., operations between 1 and 10 and functions between 1 and 100, the energy consumption of the optimization process is similar to the consumption of a regular function (around 10 J, see Section 6.3). However, for massive scenarios, i.e., 100 and 1000 operations, the optimization process can have an energy consumption 1000 times higher (i.e., 9143 J) than the consumption of a regular function. Analysing the time for these experiments, they are lower than 10 s but with 1000 operations and 100 and 1000 functions. The massive scenarios are uncommon, and we have used them to analyse the scalability of the FUSPAQ optimization process. Graphics in Fig. 13 show the increase in energy consumption 13(a) and time 13(b) when the number of functions increases for scenarios with 3, 10, 100 and 100 operations. As we can see in the graphics, the scalability in terms of energy consumption and time worsens when the number of operations increases. For example, for the experiment with three operations, when we increase 1000 times the number of functions, the energy consumption is increased by 26, and the time is doubled. However, in the scenario with 1000 operations, when we increase 1000 times the number of functions, the energy consumption is multiplied by 1357 times, and the duration is multiplied by 770. Even in the worst scenario, energy consumption and time grow linearly. In any case, FaaS applications with 1000 operations are unrealistic.

Energy saving: Regarding the potential benefits in terms of energy savings, we have seen that having functions with a variety of energy consumption contributes to energy savings in every scenario (see Table 4). To measure the benefit, we have calculated the percentage difference between *ST1* and *ST2*, and the result is between 20% and 30% for one alternative function and greater than 50% for the rest of the scenarios. It is interesting to note that the values of percentage difference are very similar when the number of functions is 10, 100 and 1000. Therefore, having around ten alternative functions is enough to significantly reduce energy consumption, and we do not incur the overhead introduced when the number of functions is 100 and 1000 (see Table 3).

Table 4
Energy consumption for experiments *ST1* and *ST2* in relation to the number of operations and the number of alternative functions.

Operation	Functions	Mean (J)		Std.		Percentage
		ST1	ST2	ST1	ST2	difference
3	1	70.46	50.99	0.90	5.39	32%
	10	67.35	33.55	0.84	1.36	67%
	100	68.78	31.66	0.80	0.2	74%
	1000	71.46	34.82	0.67	0.24	69%
10	1	231.20	168.51	1.37	0.9	31%
	10	232.12	128.73	1.48	3.28	57%
	100	241.15	120.88	1.63	0.36	66%
	1000	231.07	126.74	1.62	0.39	58%
100	1	2274	1777	5.19	31.14	25%
	10	2390	1352	5.20	9.40	55%
	100	2279	1273	3.35	1.10	57%
	1000	2305	1271	3.08	0.90	58%
1000	1	23 019	17 809	13.91	24.03	26%
	10	23 301	13 794	10.14	7.84	51%
	100	23 169	12 757	8.64	1.22	58%
	1000	23 647	12 689	6.36	0.94	60%

Table 5
Benefit in energy saving of ST1 compared to ST2 (in Joules) considering optimization cost.

Operations	Functions	ST1	ST2	Optimization	Benefit
3	1	70.46	50.99	1.16	18.31
	10	67.35	33.55	0.73	33.07
	100	68.78	31.66	3.49	33.63
	1000	71.46	34.82	26.56	10.08
10	1	231.20	168.51	2.53	60.16
	10	232.12	128.73	2.41	100.98
	100	241.15	120.88	8.16	112.11
	1000	231.07	126.74	86.75	17.58
100	1	2274	1777	3.66	493.54
	10	2390	1352	7.02	1030
	100	2279	1273	81.68	924
	1000	2305	1271	915.05	118.98
1000	1	23 019	17 809	6.76	5203
	10	23 301	13 794	39.66	9467
	100	23 169	12 757	863.23	9575
	1000	23 647	12 689	9143	1814

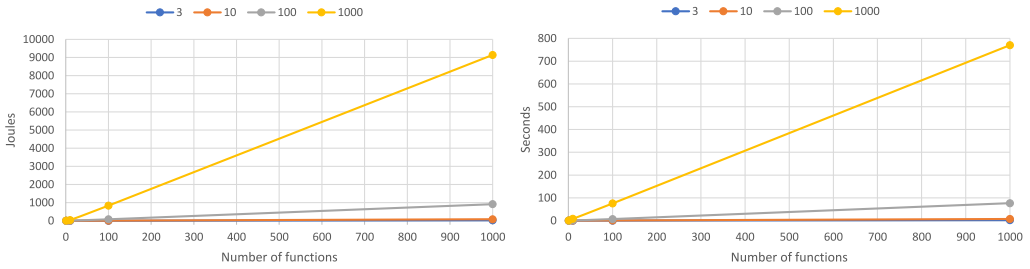
Another interesting issue is the cost-effectiveness of FUSPAQ in terms of energy consumption.

FUSPAQ Energy consumption: We have compared the energy consumption of ST1 and ST2 experiments and added to this last one the cost of the optimization process (see Table 5). We can see that there is energy saving in each scenario. So, using FUSPAQ is worth it, even for massive scenarios.

6.5. Energy consumption of our solution

To verify that our solution does not introduce high energy consumption, we have measured the power consumed by our self-adaptive system during the experiments. Since Kepler only performs measurements on containers or complete nodes, it is impossible to use it directly to measure the energy consumed by a specific application. So, we have estimated the energy consumption of FUSPAQ measuring node energy consumption during the execution of the application, using and without using our system. The difference obtained, therefore, corresponds to the consumption of the self-adaptive system.

We have implemented the FUSPAQ components in Python because it facilitates the development and modification of our prototypes. However, as a high-level interpreted language, Python's energy consumption is higher than other languages. We also collect and store monitoring data, which represents additional consumption. So, we have



(a) Energy Consumption (b) Duration

Fig. 13. Energy consumption and time for FUSPAQ optimization process.

developed alternative implementations of some parts of FUSPAQ in C, in order to estimate what the energy consumption of our solution could be in a real system. Specifically, we have implemented the ones that consume more energy in total: the *Request Processor* and the *Energy Consumption Monitor*. This is due to the frequency with which these components are used, as they are executed on each call to a function, while the configuration process is only executed when needed.

In addition, different strategies are carried out to minimize FUSPAQ consumption. On the one hand, its monitoring frequency is reduced. This is possible while maintaining the same intervals used for the averages since Prometheus continues to collect the values provided by Kepler with the same frequency as before, we only change the frequency with which our software processes them. On the other hand, instead of monitoring every function when it is called, only the functions that have an alternative in the function database are monitored.

After applying these two solutions (the use of C for the two more demanding components and the optimization of the monitoring process), we reduced 40 times the consumption of FUSPAQ compared to the initial pure Python version. Thus, the consumption of our system is approximately 1 kJ during the tests of 5000 executions mentioned above. This consumption is lower than the energy savings obtained in all the tests shown above. In any case, we must also take into account that these are experiments in which only one function had different implementations and that the differences between their consumptions were minor, so in other cases with more significant differences and a greater number of functions that can be optimized, the final energy savings will be more substantial, as we will check below.

We have also carried out a study of the consumption when executing the benchmark application. We have obtained separately the consumption of the application and the configuration processes. As we can see in Fig. 13(a), the consumption of the modelling and optimization processes is increasing and depends on both the number of operations performed by the application and the number of alternative functions that can perform each of these operations. It can be observed that, while this energy consumption is proportional to the number of operations performed, the increase in consumption due to the increase in the number of alternative functions is much smaller.

Suppose we compare the consumption of the configuration process with the consumption of the application in experiments *ST1* and *ST2*. We see that, although with low values of the number of operations and functions, the energy consumed by the configuration process is much lower, as these increase, the energy levels begin to rise. At this point, it is essential to consider that the number of operations and functions considered are much higher than those of a real case. Usually, we can have two or three alternative implementations of a function, maybe up to 10. In addition, we must consider that the configuration processes are carried out infrequently compared to the number of executions of the application. For example, if it were performed once for every 100 executions of the application, the energy that the configuration process would contribute to the total would be an average of 10 J per execution of the application, considering 100 alternative functions per operation, for the worst case that we have studied, that is, an application that performs 1000 operations. In this case, the consumption of the application was 12 KJ, so the consumption of the configuration processes is negligible.

7. Threats to validity

This section briefly discusses the internal validity, construct validity and external validity of our study [46]. The internal validity intends to explore whether the results obtained are influenced or not by other factors. Threats to construct validity concern the completeness of our study, as well as any potential bias. Finally, external validity analyses whether we can generalize the experiment results.

A critical threat to the internal validity of our study is the accuracy of the results provided. In our case, we have discussed the challenges of getting accurate results for the energy consumption of containers. We aim to have results that compare different implementations for the same task and to use this information to adapt the deployment. The second experiment demonstrates that this is possible. Additionally, our experimental setting facilitates the reproduction of our results.

One of the threats to the construct validity of this work is the lack of comparison with other approaches. As we can see in Section 2, other approaches do not focus on self-adaptation using energy in the context of FaaS. So, this comparison is not possible, and our experiments focus on illustrating that our extension of FUSPAQ actually works. Another threat is the stochastic factors present in the experiments. In our case, we do not control when the container process is executed or the other processes running in the container node. As we have discussed in Section 5 and in [43], this randomness and other factors make moving averages necessary to make the information provided by Kepler useable for self-adaptation.

Regarding the external validity of the results, we have presented two illustrative examples of the use of FUSPAQ and a benchmark application to study the behaviour with more complex applications. We have demonstrated that the approach works, but our system does not consider the node in which the functions are deployed. This could produce more significant variability in the measurements, especially if the environments are not homogeneous. However, it is a feature we are already working on.

8. Conclusions

In this article, we have presented FUSPAQ, a solution for the self-adaptation of FaaS applications considering energy consumption. FUSPAQ exploits the existence of different implementations of the same operation to reduce energy consumption. We have analysed the behaviour of the energy measurements obtained using Kepler. This software uses a hardware sensor (RAPL), and we have studied how to adapt it to the needs of a self-adaptive system. To test the operation of the system and its capabilities, we have carried out a case study on a facial recognition application. As stated in the results section, the implemented system can select the best function implementation from an energy point of view. Still, it can self-adapt, reducing the application's

energy consumption even if the consumption of implementations varies over time.

Our approach may present some limitations due to the characteristics of FaaS systems. These applications are often deployed on private infrastructures we cannot access. This is an advantage from the point of view of administration and maintenance costs but limits the use that can be done. Access to energy data through the RAPL interface is limited to internal use from the machine itself, so it is currently not possible to access said data in a private Cloud through Kepler or similar. Therefore, to use the system presented, it would be necessary to use estimation tools. Another limitation of our system is that it does not consider the node in which the functions are deployed. This could produce more significant variability in the measurements, especially if the environments are not homogeneous. Using information about our infrastructure can also be beneficial if it has nodes powered by green energy, allowing us to prioritize the selection of these nodes for the deployment of the most costly functions in terms of consumption. However, these are features we are already working on.

As a line of future work, we plan to study the energy behaviour of each replica that is running and its dependence on the infrastructure to make more optimal decisions. There is a strong relationship between sustainability and reduction of energy consumption. It is commonplace that energy consumption significantly impacts environmental sustainability by influencing greenhouse gas emissions and resource depletion. Conversely, renewable or green energy consumption improves ecological sustainability by reducing emissions and promoting cleaner energy practices. So, another line for future work is to integrate information about the carbon footprint of the energy used by functions in a given moment to make more sustainable decisions. This approach is especially advantageous when the FaaS function can be executed in edge nodes powered by a renewable energy source.

CRediT authorship contribution statement

Pablo Serrano-Gutierrez: Writing – original draft, Validation, Software, Methodology, Investigation. **Inmaculada Ayala:** Writing – review & editing, Supervision, Methodology. **Lidia Fuentes:** Writing – review & editing, Supervision, Methodology, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Lidia Fuentes reports financial support was provided by European Union. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Work supported by the *TASOVA PLUS research network* (RED2022-134337-T), and the projects *IRIS* PID2021-122812OB-I00 (FEDER funds), *DAEMON* H2020-101017109.

Data availability

Sources are linked in the article.

References

[1] Lotfi Belkhir, Ahmed Elmeligi, Assessing ICT global emissions footprint: Trends to 2040 & recommendations, J. Clean. Prod. (ISSN: 0959-6526) 177 (2018) 448–463, http://dx.doi.org/10.1016/j.jclepro.2017.12.239, URL https://www.sciencedirect.com/science/article/pii/S095965261733233X.

[2] Rajesh Ranjan Mohanty, Building a sustainable future with green software engineering, SAP (2023) URL https://engineering.leanix.net/blog/sustainable-green-software-engineering/.

[3] Alcides Fonseca, Rick Kazman, Patricia Lago, A manifesto for energy-aware software, IEEE Softw. 36 (6) (2019) 79–82, http://dx.doi.org/10.1109/MS.2019.2924498.

[4] Mohammad Sadegh Aslanpour, Adel N. Toosi, Muhammad Aamir Cheema, Raj Gaire, Energy-aware resource scheduling for serverless edge computing, in: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2022, pp. 190–199, http://dx.doi.org/10.1109/CCGrid54584.2022.00028.

[5] Panos Patros, Josef Spillner, Alessandro V. Papadopoulos, Blesson Varghese, Omer Rana, Schahram Dustdar, Toward sustainable serverless computing, IEEE Internet Comput. 25 (6) (2021) 42–50, http://dx.doi.org/10.1109/MIC.2021.3093105.

[6] Achilleas Tzenetopoulos, Charalampos Marantos, Giannos Gavrielides, Sotirios Xydis, Dimitrios Soudris, FADE: Faas-inspired application decomposition and energy-aware function placement on the edge, in: Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems, SCOPES '21, Association for Computing Machinery, New York, NY, USA, ISBN: 9781450391665, 2021, pp. 7–10, http://dx.doi.org/10.1145/3493229.3493306.

[7] Michail Tsenos, Aristotelis Peri, Vana Kalogeraki, Energy efficient scheduling for serverless systems, in: 2023 IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS, 2023, pp. 27–36, http://dx.doi.org/10.1109/ACSOS58161.2023.00020.

[8] Intel, RAPL, 2012, URL.

[9] Navin Sabharwal, Piyush Pandey, Getting started with prometheus and alert manager, in: Monitoring Microservices and Containerized Applications: Deployment, Configuration, and Best Practices for Prometheus and Alert Manager, A Press, Berkeley, CA, ISBN: 978-1-4842-6216-0, 2020, pp. 43–83, http://dx.doi.org/10.1007/978-1-4842-6216-0_3.

[10] Pablo Serrano-Gutierrez, Inmaculada Ayala, Lidia Fuentes, FUSPAQ: A function selection platform to adjust QoS in a faas application, in: Service-Oriented Computing – ICSOC 2022 Workshops, Springer Nature Switzerland, Cham, 2023, pp. 249–260.

[11] Dac-Nhuong Le, Souvik Pal, Prasant Kumar Pattnaik, OpenFaaS, in: Cloud Computing Solutions, John Wiley & Sons, Ltd, ISBN: 9781119682318, 2022, pp. 287–303, http://dx.doi.org/10.1002/9781119682318.ch17.

[12] Kwanwoo Lee, Kyo C. Kang, Jaejoon Lee, Concepts and guidelines of feature modeling for product line software engineering, in: Cristina Gacek (Ed.), Software Reuse: Methods, Techniques, and Tools, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN: 978-3-540-46020-6, 2002, pp. 62–77.

[13] Marco Gaglianese, Jacopo Soldani, Stefano Forti, Antonio Brogi, Green orchestration of cloud-edge applications: State of the art and open challenges, in: 2023 IEEE International Conference on Service-Oriented System Engineering, SOSE, 2023, pp. 250–261, http://dx.doi.org/10.1109/SOSE58276.2023.00036.

[14] Avita Katal, Susheela Dahiya, Tanupriya Choudhury, Energy efficiency in cloud computing data centers: a survey on software technologies, Clust. Comput. (ISSN: 1573-7543) 26 (3) (2023) 1845–1875, http://dx.doi.org/10.1007/s10586-022-03713-0.

[15] Alexander Poth, Niklas Schubert, Andreas Riel, Sustainability efficiency challenges of modern IT architectures – a quality model for serverless energy footprint, in: Murat Yilmaz, Jörg Niemann, Paul Clarke, Richard Messnarz (Eds.), Systems, Software and Services Process Improvement, Springer International Publishing, Cham, ISBN: 978-3-030-56441-4, 2020, pp. 289–301.

[16] Prateek Sharma, Challenges and opportunities in sustainable serverless computing, SIGENERGY Energy Inf. Rev. 3 (3) (2023) 53–58, http://dx.doi.org/10.1145/3630614.3630624.

[17] Abdulaziz Alhindi, Karim Djemame, Fatemeh Banaie Heravan, On the power consumption of serverless functions: An evaluation of OpenFaaS, in: IEEE/ACM 15th International Conference on Utility and Cloud Computing, UCC, IEEE, ISBN: 9781665460873, 2022, pp. 366–371.

[18] Rafael Moreno-Vozmediano, Eduardo Huedo, Rubén S. Montero, Ignacio M. Llorente, Latency and resource consumption analysis for serverless edge analytics, J. Cloud Comput. (ISSN: 2192-113X) 12 (1) (2023) 108, http://dx.doi.org/10.1186/s13677-023-00485-9.

[19] Cherif Latreche, Nikos Parlavantzas, Hector A. Duran-Limon, FoRLess: A Deep Reinforcement Learning-based approach for FaaS Placement in Fog, in: UCC 2024 - 17th IEEE/ACM International Conference on Utility and Cloud Computing, Sharjah, United Arab Emirates, 2024, pp. 1–9.

[20] Shahrokh Vahabi, Francesca Righetti, Carlo Vallati, Nicola Tonellotto, Energy-efficient resource management for real-time applications in faas edge computing platforms, in: Proceedings of the IEEE/ACM 16th International Conference on Utility and Cloud Computing, UCC '23, Association for Computing Machinery,

New York, NY, USA, ISBN: 9798400702341, 2024, http://dx.doi.org/10.1145/3603166.3632240.

[21] Cecilia Calavaro, Gabriele Russo Russo, Martina Salvati, Valeria Cardellini, Francesco Lo Presti, Towards energy-aware execution and offloading of serverless functions, in: Proceedings of the 4th Workshop on Flexible Resource and Application Management on the Edge, FRAME '24, Association for Computing Machinery, New York, NY, USA, ISBN: 9798400706417, 2024, pp. 23–30, http://dx.doi.org/10.1145/3659994.3660313.

[22] Francesca Righetti, Nicola Tonellotto, Nicola Barsanti, Carlo Vallati, Energy-efficient orchestration strategies for function-as-a-service platforms, in: 2024 IEEE International Conference on Pervasive Computing and Communications Workshops and Other Affiliated Events (PerCom Workshops), 2024, pp. 290–295, http://dx.doi.org/10.1109/PerComWorkshops59983.2024.10502855.

[23] Jovan Stojkovic, Nikoleta Iliakopoulou, Tianyin Xu, Hubertus Franke, Josep Torrellas, EcoFaaS: Rethinking the design of serverless environments for energy efficiency, in: Proceedings of the 51st Annual International Symposium on Computer Architecture, ISCA'24, 2024.

[24] Xuechao Jia, Laiping Zhao, RAEF: Energy-efficient resource allocation through energy fungibility in serverless, in: IEEE 27th International Conference on Parallel and Distributed Systems, ICPADS, IEEE, ISBN: 9781665408783, 2021, pp. 434–441.

[25] Seyed Hamed Rastegar, Hossein Shafiei, A. Khonsari, Enex: An energy-aware execution scheduler for serverless computing, IEEE Trans. Ind. Informatics (2024) http://dx.doi.org/10.1109/TII.2023.3290985.

[26] Anthony Byrne, Yanni Pang, Allen Zou, Shripad Nadgowda, Ayse K. Coskun, MicroFaaS: Energy-efficient serverless on bare-metal single-board computers, in: Design, Automation & Test in Europe Conference & Exhibition, DATE, 2022, pp. 754–759, http://dx.doi.org/10.23919/DATE54114.2022.9774688.

[27] Dinh-Dai Vu, Minh-Ngoc Tran, Younghan Kim, Predictive hybrid autoscaling for containerized applications, IEEE Access 10 (2022) 109768–109778, http://dx.doi.org/10.1109/ACCESS.2022.3214985.

[28] Raulian Chiorescu, Karim Djemame, Scheduling energy-aware multi-function serverless workloads in OpenFaaS, in: Maurizio Naldi, Karim Djemame, Jörn Altmann, José Ángel Bañares (Eds.), Economics of Grids, Clouds, Systems, and Services, Springer Nature Switzerland, Cham, ISBN: 978-3-031-81226-2, 2025, pp. 137–149.

[29] Roberto Amadini, Simone Gazza, Jacopo Soldani, Monica Vitali, Antonio Brogi, Stefano Forti, Saverio Giallorenzo, Pierluigi Plebani, Francisco Ponce, Gianluigi Zavattaro, Pick a flavour: Towards sustainable deployment of cloud-edge applications, in: Juliana Bowles, Harald Sø ndergaard (Eds.), Logic-Based Program Synthesis and Transformation, Springer Nature Switzerland, Cham, ISBN: 978-3-031-71294-4, 2024, pp. 117–127.

[30] Stefano Forti, Jacopo Soldani, Antonio Brogi, Carbon-aware software services, in: Claus Pahl, Andrea Janes, Tomas Cerny, Valentina Lenarduzzi, Matteo Esposito (Eds.), Service-Oriented and Cloud Computing, Springer Nature Switzerland, Cham, ISBN: 978-3-031-84617-5, 2025, pp. 65–80.

[31] Andrei Palade, Aqeel Kazmi, Siobhán Clarke, An evaluation of open source serverless computing frameworks support at the edge, in: 2019 IEEE World Congress on Services, SERVICES, 2642-939X, 2019, pp. 206–211, http://dx.doi.org/10.1109/SERVICES.2019.00057.

[32] S.M. Ali Kazmi, Ammar Nadeem, Serverless computing: Comparative analysis of faas and paas in cloud computing, Int. J. Adv. Comput. Sci. Appl. 11 (9) (2020) 617–626.

[33] Abhijit Mandal, Javier Parra, Alberto Pérez, Exploring the simplicity of OpenFaaS: A framework for serverless simplicity, J. Cloud Comput. 8 (1) (2019) 3–12.

[34] Gregory McGrath, Jason Short, Investigating the impact of knative on serverless architectures, ACM Trans. Serverless Comput. 3 (1) (2020) 1–13.

[35] Karim Djemame, Matthew Parker, Daniel Datsev, Open-source serverless architectures: an evaluation of apache OpenWhisk, in: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing, UCC, 2020, pp. 329–335, http://dx.doi.org/10.1109/UCC48980.2020.00052.

[36] Gojko Adzic, Robert Chatley, Serverless computing: economic and architectural impact, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 884–889.

[37] B. Petit, Scaphandre, 2020, URL Available: https://hubblo-org.

[38] Marcelo Amaral, Huamin Chen, Tatsuhiro Chiba, Rina Nakazawa, Sunyanan Choochotkaew, Eun Kyung Lee, Tamar Eilam, Kepler: A framework to calculate the energy consumption of containerized applications, in: 2023 IEEE 16th International Conference on Cloud Computing, CLOUD, 2023, pp. 69–71, http://dx.doi.org/10.1109/CLOUD60044.2023.00017.

[39] Pablo Serrano-Gutierrez, Inmaculada Ayala, Lidia Fuentes, Applying QoS in FaaS Applications: A Software Product Line Approach, in: Joint Post-Proceedings of the Third and Fourth International Conference on Microservices (Microservices 2020/2022), Vol. 111, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, ISBN: 978-3-95977-306-5, 2023, pp. 9:1–9:15, http://dx.doi.org/10.4230/OASIcs.Microservices.2020-2022.9.

[40] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, Klaus Schmid, Dynamic software product lines, Computer 41 (4) (2008) 93–95, http://dx.doi.org/10.1109/MC.2008.123.

[41] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, Carlos Lucena, Refactoring product lines, in: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE '06, Association for Computing Machinery, New York, NY, USA, ISBN: 1595932372, 2006, pp. 201–210, http://dx.doi.org/10.1145/1173706.1173737.

[42] J.O. Kephart, D.M. Chess, The vision of autonomic computing, Computer 36 (1) (2003) 41–50, http://dx.doi.org/10.1109/MC.2003.1160055.

[43] Pablo Serrano-Gutierrez, Inmaculada Ayala, Using energy consumption for self-adaptation in faas, in: Achilleas Achilleos, Lidia Fuentes, George Angelos Papadopoulos (Eds.), Reuse and Software Quality, Springer Nature Switzerland, Cham, ISBN: 978-3-031-66459-5, 2024, pp. 123–134.

[44] Mathilde Jay, Vladimir Ostapenco, Laurent Lefevre, Denis Trystram, Anne-Cécile Orgerie, Benjamin Fichel, An experimental comparison of software-based power meters: focus on cpu and GPU, in: 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid), 2023, pp. 106–118, http://dx.doi.org/10.1109/CCGrid57682.2023.00020.

[45] Álvaro Bermúdez Gámez, Study of the energy performance of serverless multimedia applications, 2024.

[46] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén, et al., Experimentation in Software Engineering, Vol. 236, Springer, 2012.