Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 256 (2025) 602–609

Procedia Computer Science

www.elsevier.com/locate/procedia

CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN – International Conference on Project MANagement / HCist – International Conference on Health and Social Care Information Systems and Technologies 2024

A Serverless Approach for Resource-constrained Smart Locker Networks

M. Seleiro[a], J. Simão[a, b,*], N. Datia[a,d], R. Rebelo[a,c], M. Pato[a,d,e], A. Serrador[a], P. Sampaio[a]

[a]*Lisbon School of Engineering (ISEL) Politécnico de Lisboa, Lisbon, Portugal*
[b]*INESC-ID Lisboa, Lisboa, Portugal*
[c]*CTT - Correios de Portugal S.A*
[d]*NOVA LINCS, NOVA School of Science and Technology, Monte da Caparica, Portugal*
[e]*LASIGE, FCUL, Universidade de Lisboa, Lisboa, Portugal*

Abstract

Online retail sales continue to grow, presenting couriers companies with a chance to enhance their market presence. Couriers must keep up with this trend in package processing and delivery. A modern solution is the use of smart lockers. These are equipped with a minimum amount of computing resources, while communicating efficiently with cloud supported services. To address these constraints, developers must consider the system architecture, deployment strategies, and data processing optimisation of the lockers (edge) and the main services (cloud), working alongside stakeholders to acquire the most appropriate hardware for the project. In this paper, we explore the use of a serverless architecture for smart locker. Using stateless functions and workflows, the locker can quickly and effectively react to events from the cloud and its own frontend, while supporting a lightweight serverless framework that provides an easy way to deploy and maintain the continuous software services/functions execution. These serverless frameworks have low resource requirements, allowing developers to utilize less resourceful hardware for the benefit of stakeholders. Additionally, a work-in-progress workflow prototype for the locker functions will also be presented, tested in a resource-constrained environment using *faasd* as the supporting framework, and compared against a more traditional service-oriented approach – Spring services. Our results show that serverless performed considerably well when compared to service-oriented solutions, with comparable results in terms of both memory usage and response latency.

Peer-review under responsibility of the scientific committee of the CENTERIS - International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement / HCist - International Conference on Health and Social Care Information Systems and Technologies

* Corresponding author. Tel.: +0-000-000-0000 ; fax: +0-000-000-0000 .
E-mail address: jose.simao@isel.pt

Peer-review under responsibility of the scientific committee of the CENTERIS - International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement / HCist - International Conference on Health and Social Care Information Systems and Technologies
10.1016/j.procs.2025.02.157

Keywords: Smart lockers; Serverless architecture; Edge computing; IoT

1. Introduction

During the COVID-19 pandemic, online retailers increased their sales and maintained their growth in the post-lockdown period [1]. This opened a window of opportunity for couriers to strengthen their business and increase their market activity. However, to achieve sustainable growth, they need to be efficient, more environmentally friendly and keep their services attractive to customers [2]. One solution that has proved successful is the use of lockers, distributed in many convenient locations. These lockers address both delivery efficiency, ensuring that a scheduled order is always delivered successfully, and customer convenience and satisfaction, since customers can collect their online orders at a convenient moment. On the one hand, maintaining a large number of lockers purchased at different times poses some problems, such as the difference in hardware capabilities, due to the hardware replacement life-cycle, and also due to hardware shortages and costs. On the other hand, there is the need to have a common software base for some common features, such as authentication, across different hardware versions and hardware solutions. These previous needs are driving a change in software architecture, away from a monolithic approach to a more microservices-centric one, forming a centrally coordinated IT infrastructure [3], with containerised services for portability. This trend is proved to be challenging as it requires the use of orchestration runtimes for the service containers, and hardware has memory limitation constraints, amongst others. Most of the currently available frameworks do not take these hardware constraints into account in an edge/fog setup.

Recently, researchers started analysing the possibility of using serverless approach in the world of edge computing. The FaaS model, coupled with Edge/Fog computing approaches, proves beneficial for IoT applications by deploying personalised functions onto IoT devices for localised and efficient responses to events [4]. IoT services, in this context, evolve from merely collecting raw data to producing actionable results, reducing latency, and enhancing security by isolation of execution units and/or runtimes [4].

The main contribution of this paper can be summarized as follows: (a) exploration and evaluation of a novel serverless approach for edge computing in the context of smart locker networks; (b) comparison analysis between serverless approach and a traditional service-oriented architecture, highlighting the advantages of the serverless paradigm in terms of code development, deployment, maintenance, and resource utilization on edge devices. While this research is ongoing, our initial findings based on small-scale functional demos demonstrate the potential effectiveness of serverless computing in edge environments for distributed locker systems. The remainder of this paper is structured as follows: Section 2 presents an analysis of literature review about the serverless edge computing paradigm, and describes several serverless platforms that support development in edge without cloud connection; Section 3 establishes the problems that need to be considered and tackled in order to develop a serverless framework in smart locker networks; Section 4 provides a high-level overview of the prototype architecture for the locker; Section 5 presents and discusses the experimental results; and Section 6 reflects and outlines the future direction of this research.

2. Background

The discussion of serverless edge computing as a real use case, started to rise in 2017 [5], as the scenario where code could/should be executed outside the cloud data centre started to be considered. One of the main possibilities discussed, was to include serverless support in IoT devices and associated computation at the edge. With this motivation, efforts were made to extend the serverless paradigm to the world of edge computing.

Baresi, L. and Mendonça, D.F. [6] in 2019, picked up on this discussion and reflected on the possibility of a serverless platform for edge computing. The authors began by identifying possible characteristics of a serverless edge platform through reference scenarios. These include (i) *Low-Latency Computation Offload* for computation-intensive tasks; (ii) *Inter-Platform Collaboration* between multiple edge nodes running FaaS platforms; (iii) *Latency*

Optimisation when creating function workflows; and (iv) *Opportunistic Data Analysis* and processing at the edge, preventing large volumes of data to be sent to distant cloud servers. The authors identify key requirements and challenges for the development of these characteristics, proposing solutions for their resolution. This analysis served as a base for the authors to develop a prototype for serverless edge platforms, taking all the characteristics into account, built on top of an adopted version of the open-source project Apache OpenWhisk [7], a FaaS platform. The authors performed two experiments, using Linux machines with 16 GB of RAM, and measured the memory footprint of all components in the prototype. The total memory footprint was measured to be about 2.2 GB in idle, which the authors reflect that still needs further optimisation. On a positive note, the authors say that overall, the obtained results corroborate the benefits of the serverless architecture in: (i) drastically reducing the burden of infrastructure management, (ii) allowing more functionality to be deployed on fog nodes with limited resources, and (iii) fulfilling the requirements of different application scenarios and heterogeneous deployments of edge nodes. The authors hope that this prototype will motivate future research into the development of serverless at the edge.

Similarly, in 2021 Aslanpour et al. [8] identified, through an extensive literature review of serverless and edge computing, the essential components of such an infrastructure, discussing how it is placed in the cloud and edge. This infrastructure is split into three layers of components – computation, execution and coordination. In the computation layer, we have storage, networking, and functions computing; the execution layer includes the execution engine and monitoring; and finally, the coordination layer is where the controller is located, together with schedulers and queues. In typical cloud serverless, the serverless infrastructure is housed in a single cloud data centre, handling all functions. However, in edge computing, each edge node has its own individual infrastructure, only executing functions related to whichever devices it controls. The authors describe how the serverless edge computing paradigm offers a wide range of opportunities for edge computing. These include the ability to handle greater amounts of workload and data by scaling functions up, save resources such as RAM and battery in low traffic scenarios by being able to scale functions down to zero, and a serverless, stateless life-cycle that is more compatible with the event-driven scenario typically seen alongside the edge. Conversely, the authors also highlighted several open issues that still need to be addressed, such as function cold starts which add a delay to response times, reliability and fault tolerance issues, and a lack of simulation tools to test the developed functions. Evidently, the implementation of a serverless approach in edge computing has its merits. Serverless seamlessly tackles resource constraint issues commonly seen in IoT and edge, and its model of stateless functions with variable scalability allows for plenty of flexibility in architectural design and resource management. However, before we can accept serverless as a sustainable edge solution, we must first tackle the many open issues serverless edge is facing. To work through these issues, many tools and frameworks have been developed to support serverless functions in edge environments, promoting serverless edge and advancing the research in the area.

These developments can be divided in two categories, cloud-hosted and self-hosted. A cloud-hosted framework example is AWS Greengrass [9], a framework which can be used in conjunction with the serverless platform AWS Lambda [10], to achieve edge integration. However, while cloud-hosted solutions show consistent performance results, connection to the cloud is still required, which creates extra latency in operation and may lead developers into choosing more traditional edge solutions, e.g. Spring services, over serverless. Given that smart lockers require quick reactions to new events and interactions, self-hosted solutions are more fitting.

2.1. Self-hosted Serverless frameworks

OpenFaaS [11] is a framework that makes it easy for developers to create serverless functions. The main component of OpenFaaS is its API gateway, which is where all the other components connect to. Access to the functions deployed on the orchestration engine, i.e. Kubernetes, is provided through the gateway, and from there, tools such as Prometheus can collect metrics on the function usage and performance, and autoscaling can be enabled based on traffic by interacting with Kubernetes. To increase support for edge solutions, OpenFaaS developed a variant of itself called *faasd*, which removes the cost and complexity of Kubernetes. Running on a single host, *faasd* uses *containerd* and *Container Networking Interface* (CNI) alongside the same core components of OpenFaaS, turning the edge device it is running on into a lightweight server for hosting OpenFaaS functions, only losing out on the autoscaling feature.

Knative [12] is a platform-agnostic solution for running serverless functions on top of Kubernetes. It is composed of 2 main components: serving and eventing. Knative serving defines a set of objects as Kubernetes Custom Resource

Definitions (CRDs) which will be used to define and control the serverless workload. Serving can enable rapid deployment and automatic scaling of containers based on demand. Knative eventing is a collection of APIs that enable the use of event-driven architecture in the serverless infrastructure. The APIs are used to create route events from producers to consumers, using HTTP requests for communication. The main components of Knative are compatible with edge and using Knative on top of a lightweight solution of Kubernetes, such as k3s, allows for the development of edge solutions in low-resource devices. Knative serving can scale functions to 0, granting a low-resource footprint, and Knative eventing makes the detection of events from IoT devices integrate seamlessly with the serverless functions running on the edge.

Kappa [13] is a serverless deployment concept for IoT, developed on top of the distributed IoT framework Calvin [14] due to being fully distributed and not requiring the cloud for its execution, and tasks can be executed anywhere, including edge nodes. Calvin applications are based on the actor model, implemented as re-usable units and written in Python, and the dataflow model, expressed using the purely declarative *CalvinScript* language. Kappa is divided in 3 layers, these being the frontend, backend and IoT layer. In the frontend, the user can create, communicate and delete kappas. The backend contains the Calvin runtime, which interprets the user script from the frontend and compiles a new script with the current actor workflow for the Calvin runtime. Finally, the IoT layer is where all the kappa actors are running. These actors, named kappas, have 4 categories: (i) input/output kappas; (ii) input only kappas; (iii) output only kappas; (iv) no input/output kappas. The categories are such so that the different events of an IoT system are executed by the most fitting kappa. For example, an event that are triggered by timers to execute some individual periodically task should be implemented in a no input/output kappa.

LaSS [15] is a platform for managing Latency Sensitive Serverless computations in edge, built on top of Apache OpenWhisk. The authors present a principled approach for allocating edge resources to latency-sensitive serverless functions, using this approach to design and implement an algorithm to dynamically scale the resource allocation of each function up or down based on time-varying workload demands. When the system is not under load, LaSS dynamically allocates the necessary resources to each function based on its observed workload, and when it is under heavy load, LaSS uses a weighted fair-share resource allocation approach that guarantees a minimum fair share of the edge cluster resources to each function in proportion to its weight. The authors show that the models being LaSS can react quickly to load fluctuations, and that, when stopping containers, their resource deflation policies [16] can improve resource utilisation by 6% compared to the typical termination policy.

3. Problem Statement

As stated previously, there are some problems that need to be addressed when developing a management system for distributed lockers, many of these being IoT and edge computing inherent. The problems related to different hardware versions and solutions can be solved with the use of containerisation, which is used by many distributed and serverless frameworks. However, since the software base for essential locker functions needs to be executed locally in its entirety, the hardware resources, which are extremely limited, become even more problematic to manage, and with many frameworks not assuming these hardware constraints, the options become quite limited in choosing the supporting framework. This selection of a framework is an important step of development, as it affects all the stakeholders of the project. Thus, we must ideally choose a framework which enables developers to adopt simple processes for development, deployment and maintenance of the system; ensure that it performs quickly and efficiently to keep the customer satisfaction; and use a low amount of hardware resources, so that hardware for the system can be acquired at a lower cost for the benefit of investors. Additionally, being able to acquire hardware at lower costs also allows scaling the locker network at a faster rate.

Besides the framework, it is also important to define the locker and its workflow during operation. The locker is composed by a set of several differently sized doors, each door being able to contain one (1) parcel at most. It has a touchscreen, which the mail courier and customer use to interact with the locker to deliver and retrieve parcels respectively. To separate mail courier from customer interactions, a set of three (3) keys corresponding to each parcel are generated during its creation, encrypted and sent to the locker. These are the delivery key, retrieval key, and emergency key. The delivery and emergency keys are sent in plain to the mail courier so they can deliver the parcels to the lockers and remove any parcels in case of a mistake, and the retrieval key is used by the customer to retrieve their package. Finally, to open any door, the locker must first verify if the key communicated from the screen matches

any encrypted key from the local database. If a match is found, then the locker orders the opening of the door the parcel is associated with.

Another key information that needs to be stored and updated is the parcel status, as both the locker and the back-end need to keep track of the parcel status in order to ensure continuity of service and reusability of doors. As such, we defined the parcel status to be one of the following: *arriving*, *in storage* and *delivered*. When the delivery key is used, if the operation is successful, then the parcel will change from the *arriving* status to *in storage*, and after the retrieval key is used, the parcel will go from *in storage* to *delivered* status. The emergency key is an exception, since the use case for this key is only when the mailman accidentally inserted the incorrect parcel into a specific door and needs to change it, so the parcel status will continue to be *in storage*. After every status change, the locker notifies the back-end so that both systems are synchronized.

Thus, the aim of this work is to develop a prototype, utilising a serverless approach to software development, that fulfils the locker requirements and functionalities. The prototype should (i) Communicate with the back-end system to receive the data it requires for service; (ii) Order the opening of doors if the codes inserted in the touchscreen are correct; (iii) Keep the back-end updated with parcel status; (iv) Have all the software base be executed in a lightweight serverless framework; (v) Be comparable or better to a traditional service-oriented approach in terms of performance. In order to fulfil the final requirement, we will measure response latency and memory occupation of the business logic when supported by a serverless framework, and compare these values against the same logic, but using a service-oriented approach. Other values such as battery usage won't be measured because, as a part of this case study, the locker will always have access to a power source and a running 4G terminal for back-end communication.

4. Serverless solution prototype

The workflow events for the serverless prototype can be seen in Fig. 1. Here, the information sent by the back-end, defined as *new parcel event*, it is received by the database access functions, which support all CRUD operations to the local database. The screen controller handles the touchscreen display and the *parcel request event*, created by the touchscreen interaction. The lock controller sends commands to open and close the door locks. Finally, the main controller function is where the locker's operations logic is executed, such the *lock open/close event* and where updates to the back-end database are sent to.

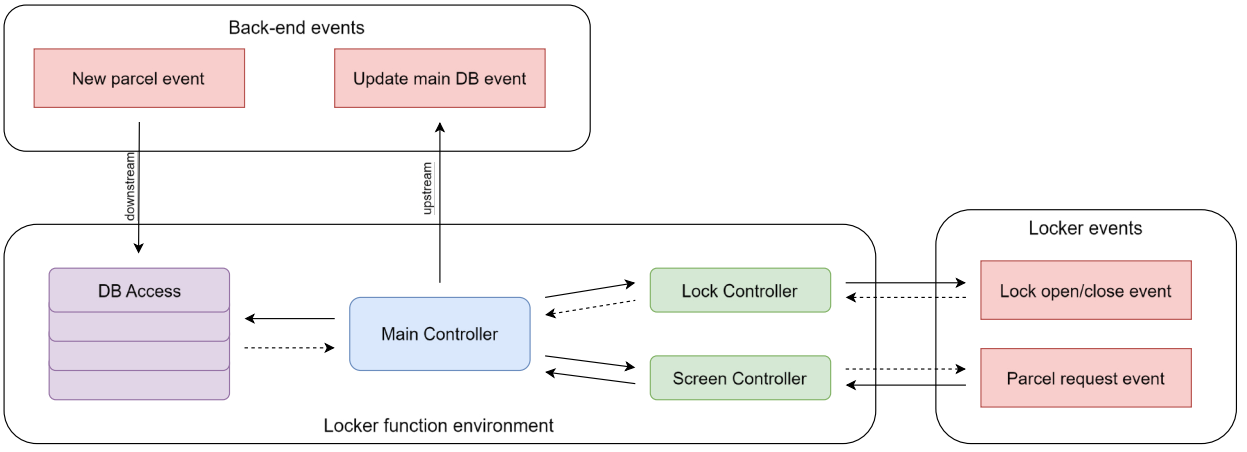

Fig. 1. Locker function environment and workflow events.

There are two main entry points to this workflow, the *new parcel* event and the *parcel request event*. Because the downstream, data does not need any extra processing at the edge, nor will it trigger any events by itself, the event gets redirected to the database handling functions to store the incoming data. Meanwhile, touchscreen interactions, will execute the main locker business logic in the main controller. By making use of the data received from the back-end,

the main controller will decide whether to open a door to the locker, or to display information on screen to convey that the key inserted is incorrect or invalid.

To support this workflow, and its execution in the test environment, we chose to use *faasd*, the lightweight, edge-oriented, OpenFaaS variant. This decision was based on analysis and comparisons performed on several serverless platforms and frameworks for edge, where OpenFaaS and *faasd* showed great results compared to similar frameworks [17],[18]. With *faasd*, one can remotely deploy the functions to the Raspberry Pi through a gateway hosted on the device, that can be also used to manage and track the functions execution. These functions are assigned to a URL path through the gateway, and all HTTP requests are made to the same path. Since *faasd* supports a large amount of programming languages, Java was selected since it is commonly used in distributed applications development.

5. Performance evaluation

To measure the performance of this prototype, a set of performance metrics were analysed, namely memory usage and response time, and compared against a more traditional service-oriented approach, SpringBoot Java services. The memory usage, consists of how much memory is still considered to be free after each individual service deployment, including the hosting frameworks. This is key to measure, as the available memory is the primary constrain in edge devices. Response time is the sum of the times it takes for the request to reach the server, the request to be handled by the service/function, and the time for the response to be delivered back to the client. Although not as important as system memory, response time is relevant due to the route of the HTTP requests on each framework – SpringBoot is hosted on a system port, while faasd works with an additional indirection of a gateway. The test environment hardware is a Raspberry Pi 3 Model B, with a 1.2GHz 64-bit quad-core ARM processor and 1GB of RAM, running Debian 11 bullseye, which will host all the functions/services. The Raspberry Pi 3 was chosen as our test environment as it is the edge device with the biggest hardware limitations in-use on smart lockers in Portugal.

To obtain these metrics, a Windows Machine was used with a six-core Ryzen 3600 and 16GB of RAM, connected to the same local network as the Raspberry Pi to reduce latency. Through an SSH connection to the Raspberry Pi, the free memory was directly measured from the system data after each individual deployment of a function/service stabilizes. The HTTP requests are generated through Apache JMeter, calculating the arithmetic average response time after 10 requests. To access performance of both approaches, the database insertion logic is used as the main testing code for response latency taken by functions/services.

The functions will be executed inside individual containers, hosted by faasd, while the SpringBoot Java services will be deployed using a multi-service launcher technique [19], which allows for the execution of multiple spring services inside one JVM. Both are designed to accept two HTTP requests, get service status and post user to database.

5.1. Results and discussion

Fig. 2, shows eight deployments impact into memory usage. These results were measured after each service stabilized. Note, the first deployment, the *faasd* process, which hosts the function containers, occupies approximately 60000 kB less memory than a JVM + SpringBoot service. As more services are deployed, the gap between memory usage of *faasd* and Spring starts decreasing, with Spring eventually surpassing *faasd* when 7 or more services are deployed. This is due the multi-service deployment technique using only one JVM, saving a considerable amount of memory per extra service after the first one. In relation to response latency, we can observe in Table 1, that both architectural approaches suffer from cold starts, and the first call to either approach had a much higher response time than the following calls.

In the case of the Spring service, both the *get health status* request and the *post user* request were affected by cold starts on the first call to each, whereas in *faasd*, only the first call to the function itself had a delayed response. This is due to the requests in *faasd* having to be implemented on the same endpoint, as mentioned previously. After allowing the services to stabilize, one can run the JMeter tests to measure the average response latency. There, we measured that the SpringBoot services replied faster to HTTP requests when compared to *faasd*, by about 28 ms. However, the highest value was from the SpringBoot services at 153 ms, while *faasd* takes 119 ms.

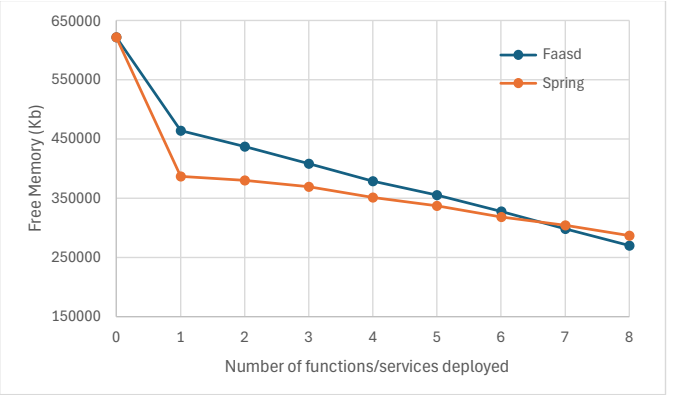

Fig. 2. Deployment effects on RAM (higher is better).

Table 1. Response latency to HTTP requests with 7 services deployed.

Architectural approach	First call (ms)	Cold start	Average (ms)	Highest (ms)
OpenFaaS/faasd functions	1990	Yes	79	119
SpringBoot services	1682	Yes	51	153

These experimental results, show that *faasd* performs considerably well in edge when compared to Spring services. This difference can be attributed to the architectural approach of this framework, which consumes a lower number of resources, as well as its native support of the ARM architecture. Alongside that, the containerization strategy of *faasd* allowed for a more optimized usage of system memory when compared to Spring when using 6 or less services. This however, increased the response latency of *faasd* functions, with the requests having to be redirected to the function container, unlike Spring which is executed directly in a JVM.

In terms of software development and maintenance, the function deployment model is much more effective than the Spring multi-launcher technique. Using *faasd*, we can make function updates seamlessly to ensure the minimum service downtime without affecting the rest of the system. However, with Spring, since it uses a launcher service to start all other Spring services inside the same JVM, updating one service involves taking down all other services. Similarly, to work around not having an autoscaling feature, with *faasd* one can deploy multiple similar functions and set up a reverse proxy for load balancing, which is impossible to do on the Spring multi-service launcher.

6. Conclusions

This paper presents a serverless workflow for smart locker networks, following the advent of the serverless edge computing paradigm. The workflow is designed to handle both back-end and front-end communication seamlessly and enables the locker to synchronize itself with the rest of the network. A smart locker system prototype based on the workflow was developed and proposed, built on top of the *faasd* framework. The experimental results show that *faasd* is a more suitable alternative to Spring multi-launcher services for stakeholders, utilizing resources such the memory usage, more efficiently, responding to HTTP requests within comparable values, and having a more flexible deployment solution. These experimental results show the potential of this framework when compared to traditional development techniques, as well as the merits of further investigation in serverless edge/IoT. Future work will explore the deployment of pilot prototype into a physical locker, measuring power consumption and analyze the long-term maintenance and cost implications of this serverless system on a distributed smart locker network.

Acknowledgements

This work is supported by CTT protocol through project “Reformulação do Sistema de Lockers CTT”, ref. CTT_188829/2024, by the LASIGE Research Unit, ref. UIDB/00408/2020 (https://doi.org/10.54499/UIDB/00408/2020) and

ref. UIDP/00408/2020 (https://doi.org/10.54499/UIDP/00408/2020, the NOVA LINCS, ref. UIDB/04516/2020 (https://doi.org/10.54499/UIDB/04516/2020) and the INESC-ID (https://doi.org/10.54499/UIDB/50021/2020) with financial support from FCT, through national funds.

References

[1] L. Szász, C. Bálint, O. Csíki, B. Z. Nagy, B. G. Rácz, D. Csala and L. C. Harris (2022) “The impact of COVID-19 on the evolution of online retail: The pandemic as a window of opportunity”, *Journal of Retailing and Consumer Services*, Elsevier, vol. 69, 103089

[2] J. Pilawa, L. Witell, A. Valtakoski and P. Kristensson (2022) "Service innovativeness in retailing: Increasing the relative attractiveness during the COVID-19 pandemic", *Journal of Retailing and Consumer Services*, Elsevier, vol. 67, 102962

[3] P. G. Keen (1991) “Shaping the future: Business design through information technology”, *Harvard Business School Press*, Harvard

[4] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov and R. Prodan (2017) "A serverless real-time data analytics platform for edge computing", *IEEE Internet Computing*, IEEE, vol. 21, n.º 4, 64–71

[5] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski and P. Suter (2017) "Serverless computing: Current trends and open problems", *Research advances in cloud computing*, Springer, 1–20

[6] L. Baresi and D. F. Mendonça (2019) "Towards a serverless platform for edge computing", *2019 IEEE International Conference on Fog Computing (ICFC)*, IEEE, 1–10

[7] Apache OpenWhisk. https://openwhisk.apache.org accessed June 2024

[8] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire and S. Dustdar (2021) "Serverless edge computing: vision and challenges", *Proceedings of the 2021 Australasian computer science week multiconference*, 1–10

[9] AWS Lambda. https://aws.amazon.com/lambda/ accessed June 2024

[10] AWS Greengrass. https://aws.amazon.com/greengrass/ accessed June 2024

[11] OpenFaaS. https://www.openfaas.com/ accessed June 2024

[12] Knative. https://knative.dev/docs/ accessed June 2024

[13] P. Persson and O. Angelsmark (2017) "Kappa: serverless iot deployment", *Proceedings of the 2nd International Workshop on Serverless Computing*, 16–21

[14] P. Persson and O. Angelsmark (2015) "Calvin - merging cloud and iot", *Procedia Computer Science*, Elsevier, vol. 52, 210–217

[15] B. Wang, A. Ali-Eldin and P. Shenoy (2021) "Lass: Running latency sensitive serverless computations at the edge", *Proceedings of the 30th international symposium on high-performance parallel and distributed computing*, 239–251

[16] P. Sharma, A. Ali-Eldin and P. Shenoy (2019) "Resource deflation: A new approach for transient resource reclamation", *Proceedings of the Fourteenth EuroSys Conference 2019*, 1–17

[17] Javed, Hamza and Toosi, Adel N and Aslanpour, Mohammad S (2022) “Serverless platforms on the edge: a performance analysis”, *New Frontiers in Cloud Computing and Internet of Things*, Springer, 165–184

[18] Palade, Andrei and Kazmi, Aqeel and Clarke, Siobhán (2019) “An evaluation of open source serverless computing frameworks support at the edge”, *2019 IEEE World Congress on Services*, IEEE, vol. 2642, 206–211

[19] Springboot multi service launcher, https://github.com/rameez4ever/springboot-demo/tree/master/springboot-multi-service-launcher, accessed June 2024