FaaS@Edge: Bringing Function-as-a-Service to Voluntary Computing at the Edge

Catarina Gonçalves[1], José Simão[1,2], and Luís Veiga[1⋆]

[1] INESC-ID, Instituto Superior Téncico, Universidade de Lisboa
[2] Instituto Superior de Engenharia de Lisboa (ISEL / FIT)

Abstract. Function-as-a-Service (FaaS) is an emerging cloud computing model ideal for processing vast amounts of data generated by the Internet of Things. However, existing FaaS approaches struggle to leverage resources efficiently on distributed edge devices. Our work presents FaaS@Edge, a solution that employs volunteered resources from edge nodes, discovered through the IPFS network, to deploy functions using the Apache OpenWhisk framework and enhancing the system's scalability and efficiency. This approach supports various language runtimes, ensuring near-universal deployability on edge devices. Our evaluation demonstrates that FaaS@Edge introduces a latency overhead for function submission but achieves similar invocation times compared to a local OpenWhisk deployment. FaaS@Edge maintains high request success rates, with overall request success rates around 98% for both submission and invocation. These results confirm that FaaS@Edge provides a viable and efficient model for FaaS deployment in edge computing environments, facilitating low latency and efficient resource utilization.

Keywords: Function-as-a-Service, Edge Computing, Cloud Computing, Volunteer Computing, Peer-to-Peer Data Networks

1 Introduction

Function-as-a-Service (FaaS) is an emerging paradigm aimed to simplify Cloud Computing and overcome its drawbacks by providing a simple interface to deploy event-driven applications that execute the function code, without the responsibility of provisioning, scaling, or managing the underlying infrastructure [12, 18]. In the FaaS model, the management effort is detached from the responsibilities of the consumer, since the cloud provider transparently handles the lifecycle, execution, and scaling of the application. This model was originally proposed for the cloud but has since been explored for deployments in geographically distributed systems [6]. With the expansion of the Internet of Things, the cloud has become an insufficient solution to respond to the growing amounts of data transmitted and the variety of Internet of Things applications that require low latency and location-aware deployments, as stated by CISCO [20]. This led to the

⋆ Work partially developed while as a Visiting Researcher with the Hybrid Cloud Computing Group at IBM Research Europe – Zurich.

introduction of the Edge Computing paradigm, designed to reduce the overload of information sent to the cloud through the Internet, by bringing the resources and computing power closer to the end user and processing the data at the edge of the network, e.g., recently for AI workloads [13, 10].

The intersection between Function-as-a-Service and Edge Computing presents a captivating area of research and innovation since the growing demand for low latency, real-time applications urges the need to explore the integration of FaaS in Edge Computing devices. At the same time, this integration also needs to address its inherent challenges, such as managing distributed architectures, optimizing resource allocation, and ensuring compatibility with the heterogeneous characteristics of edge devices.

Most cloud service platforms still rely on centralized architectures and services that are neither designed to operate on resource-constrained environments, nor on the heterogeneous devices that characterize edge systems. Solutions to bring Function-as-a-Service deployments to the edge of the network have been explored [8], [16] but few have managed to realize efficient resource provisioning and allocation [5], by leveraging volunteered resources in a completely distributed and decentralized manner [3].

Our contribution consists of a FaaS@Edge system that uses volunteer resources from multiple users, that are announced and discovered through the IPFS network, to submit and invoke user functions on their volunteered edge devices using the Apache OpenWhisk framework.

The rest of the paper is structured as follows: Section 2 describes FaaS@Edge's architecture and algorithms, alongside the implementation details of our solution. Section 3 presents the evaluation of our prototype. Section 4 presents an analysis of the related work in Cloud Computing including Function-as-a-Service, Edge Computing, and Peer-to-Peer Content, Storage and Distribution. Finally, Section 5 wraps up the paper with our closing remarks.

2 Architecture

FaaS@Edge represents a distributed and decentralized middleware framework designed to facilitate Function-as-a-Service (FaaS) deployments across a network of volunteer edge computing devices. This framework aims to minimize execution latency, optimize resource utilization, and enhance the distribution and availability of content within edge environments. The architecture of FaaS@Edge necessitates that participating nodes be equipped with specific components, as illustrated in Figure 1:

- FaaS@Edge's middleware running as daemon;
- An initialized IPFS Kubo node;
- The IPFS daemon;
- An OpenWhisk stack running as a Java process (if the node is supplying its resources to execute function requests).

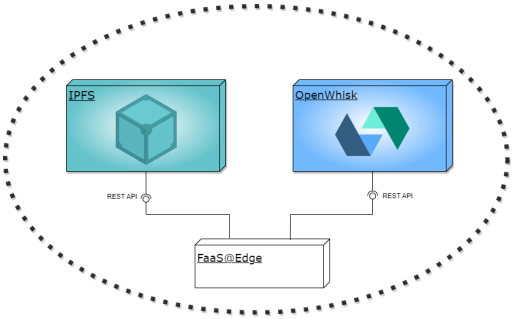

Fig. 1. FaaS@Edge participant node's complete components.

Distributed Architecture The architecture of FaaS@Edge is built upon the IPFS peer-to-peer framework, leveraging a Kademlia-based Distributed Hash Table (DHT) for its operations. This DHT is instrumental in associating content identifiers with the node identifiers and IP addresses that host the content, enabling efficient lookup, routing, and retrieval mechanisms that are crucial for large-scale content distribution, aided by built-in caching features. With every FaaS@Edge node accessible via IPFS and uniquely identified by a PeerID - a SHA-256 multihash of the node's public key - the system facilitates a distributed and decentralized approach to resource discovery. Nodes can broadcast resource availability, namely memory, on IPFS, encapsulating the memory offered in files tagged with specific strings and content identifiers (CIDs), which other nodes can then discover to deploy their functions effectively.

The system has two roles, suppliers that offer resources and clients that look for the best place to run their functions. Each node can have either one or both roles. The nodes that are running the OpenWhisk component are described as the suppliers and are the ones volunteering their resources to the system, indicating the maximum memory amount they are willing to offer in the start command. However, all nodes can send function deployment requests. During the initialization of a node, the CIDs of all the possible offer values (ranging from 128MB to 512MB, in power of 2 sizes) are calculated with IPFS' `only-hash` option in the `add` command and stored to be used by the supply and discovery algorithms.

The following data structures are used in the resource scheduling algorithms:

- **Offer** contains the *resources* a supplier node is offering and is published in IPFS as a text file with the string faas-edge-MEM, where MEM is the memory being offered (only one offer file is published per memory amount, the rest is incremented/decremented in the map presented next);
- **Supplier's Active Offers Map** keeps a record of the *number of offers* made of each *resource value*;
- **Available Offer** contains the *resources* and *supplier IP address* of an offer discovered in IPFS.

Algorithm 1: Supplier's resource supplying algorithm.

```
Data: suppActiveOffersMap, suppOfferPlan
Function SupplyResources(freeRes, maxRes):
    usedRes ← ResourcesInUse(freeRes, maxRes)
    removeSupplierOffers()
    offerCount, offerSize ← suppOfferPlan.CalculateOffers()
    foreach offerCount, offerSize do
        newOffer ← CreateOffer(offerSize)
        suppActiveOffersMap.Add(newOffer)
```

Algorithm 2: Supplier's create offer algorithm.

```
Data: suppActiveOffersMap, IPFSClient
Result: newOffer
Function CreateOffer(offerRes):
    if suppActiveOffersMap[offerRes.Value] < 1 then
        offerStr ← GetResourcesString(offerRes)
        ok ← IPFSClient.Add(offerStr)
        if ok = false then
            return Error("Unable to create offer")
    /* Only add to IPFS if there are no active offers of that memory value,
       otherwise, just create the new offer to add to the map.              */
    newOffer ← NewOffer(offerRes)
    return newOffer
```

Algorithm 1 is executed by a supplier node upon its initial integration into the system or whenever its resource availability fluctuates, either due to the allocation of a function or by freeing resources following a deployment failure. The node starts by calculating the amount of resources currently in use, given the maximum value of resources it is willing to provide and the current value of free resources it has. Then, all active offers are removed in order to calculate the offers that match the current resource availability. This is achieved by making the IPFS client to remove the offer file's *pin* per each size of active offer, the remaining offers are simply decremented in the supplier's active offers map. Given the distributed nature of IPFS and its caching capabilities, there is no direct way to delete a file, only to *unpin* it from storage and let the garbage collector reclaim it. Meanwhile, requests routed to now unavailable resources are replayed; randomness in clients sorting offers promotes selecting others available (Alg. 3).

After this, the algorithm will calculate the number and size of offers to be made, according to the respective offering plan. For each of these, it will then use Algorithm 2 to publish the file in IPFS and create a new offer object that is added to the supplier's active offers map. Adding the offer files to IPFS during each resource availability update can serve as an offer refresh and help to ensure liveness. Algorithm 2 starts by checking if the node already has active offers of that value in its offers map, or if it needs to make the offer available in IPFS. To do so, the node retrieves the string representative of that offer value and calls the IPFS client to add a file with the string to its distributed file system. If the publishing operation was successful or there was no need to publish, because at least one offer of that memory value was already being made in IPFS, the

Algorithm 3: Algorithm to schedule function in supplier node.

```
Function Schedule(fnConfig):
    resNeeded ← fnConfig.Resources
    availOffers ← DiscoverResources(resNeeded)
    availOffers ← RandomOrder(availOffers)
    foreach offer in availOffers do
        fnStatus ← SubmitFunction(fnConfig, offer, self.IP)
        if fnStatus = ok then
            deployedFn ← DeployedFn(fnConfig, offer.SuppIP)
            functionsMap.Add(deployedFn)
            return fnStatus
    return Error("Unable to schedule function")
```

function can finally create a new offer object containing the resources being offered.

When the supplier node has available resources to supply, it can follow several options on how to arrange different combinations of resource offers. These offering plans will achieve different results when it comes to effective resource utilization, fragmentation, and resource allocation. The different offering plan options are the following:

- **Balanced** - Provides the same number of offers for each size, without exceeding its maximum resource capacity.
- **Overbook** - Generates all the possible resource combinations that it can offer, thus overbooking its available resources. This approach favors resource utilization and avoids fragmentation since there are offers of all sizes. Free resources will be a result of the different supply and demand in the system.
- **Balanced Ranges** - Equivalent to the Balanced option except the offer sizes are limited within one of the following ranges: Small (128MB), Medium (256MB), or Large (512MB).
- **Overbook Ranges** - Equivalent to the Overbook option except the offer sizes are limited within one of the ranges Small, Medium, and Large presented above.
- **Random Balanced** - Each supplier node randomly chooses the offering plan between the Balanced and the three Balanced Ranges plans.
- **Random Overbook** - Each supplier node randomly chooses the offering plan between the Overbook and the three Overbook Ranges plans.

The resource discovery method occurs when a client node receives a function submission request with specified resource limits. In this process the node determines the minimum necessary memory size and obtains the relevant CID. It then uses IPFS's to find providers and identify up to 20 potential suppliers by their IPFS addresses, creating Available Offer objects for each, detailing the resources and supplier node's IP address.

Algorithm 3 is called when a user's submission request is received through the CLI application that interacts with FaaS@Edge's daemon, containing the function's configuration (source code's CID, function's name, function's runtime

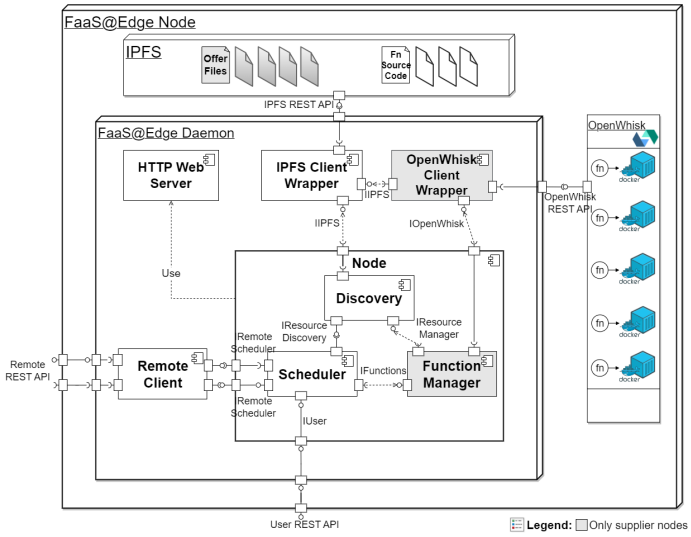

Fig. 2. FaaS@Edge's components and interfaces.

type, and resources needed). It takes the resources needed for the deployment and uses the previously described action to find a set of available offers. Then, this set of offers is sorted in random order, to contribute to spread the load and thus avoid overloading any supplier nodes. Finally, it will iterate over the sorted offers, and send a SubmitFunction message, through the node's remote client, containing the function's configuration, the offer to be used, and the node's own IP address.

When the supplier node's receives a function submission message from a client node it starts by signaling the use of the resources provided in that offer, triggering the update of the supplied offers to adjust to the decrease in the node's available resources. Then it calls the OpenWhisk component, passing the function's configuration so that it can retrieve the source code file from IPFS using its CID, and insert/create the function in OpenWhisk. If the creation is successful, the node stores a new local function object in a map, where it keeps the functions of each client node, to be able to invoke them when requested, and informs the client node of the successful deployment. In case of failure, the supplier's resources are released, and an error message is returned to the client node that requested the deployment.

Implementation Details Figure 2 provides an overview of FaaS@Edge's node software components, interfaces, and their relationships. Note that only the supplier nodes need to include the OpenWhisk Client Wrapper and Function Manager components. The main components are:

Node - Super component that drives the initialization of all other components, receiving the configuration parameters from the user through the CLI tool,

and passing them to its internal components. The node makes its scheduling services available to the other nodes and to the user through interfaces exposed via REST API;

Scheduler - Responsible for the function deployments and subsequent invocations, exposes interfaces to the user, to inject requests, and to remote nodes, allowing them to send message requests to deploy/invoke functions in this node. Interacts with the Discovery component to find available resources for a deployment;

Discovery - Implements the resource discovery algorithms to find resources offered by other provider nodes and to oversee the supplier node's resources and offers, according to the offering plan. Interacts with the IPFS Client Wrapper to add offer files to IPFS, query the DHT to find providers, and get the CID of each offer value;

Function Manager - Manages function deployments in the local node's OpenWhisk platform via OpenWhisk Client Wrapper. Provides an interface used by the Scheduler to submit and invoke functions requested by other nodes and interacts with the Discovery component to validate the use of the node's resources for deployments and release them in case an error is received from OpenWhisk;

IPFS Client Wrapper - Wraps the Go client library for the HTTP RPC API exposed by IPFS' daemon in order to provide a simplified interface that isolates the use of IPFS at our middleware's level from IPFS' core API that provides direct access to the core commands;

OpenWhisk Client Wrapper - Wraps the Go client library for the OpenWhisk API to access the running OpenWhisk services, isolating our middleware's function management from OpenWhisk's API details. Exposes an interface to be used by the Function Manager component to insert, invoke, and delete functions and enforces the system limit for how much memory a function can allocate, defined during the function's insertion in OpenWhisk;

HTTP Web Server - Serves REST API endpoints and redirects requests to the respective FaaS@Edge components. The server is started by the Node component once the user issues a *start* command.

We selected OpenWhisk and IPFS due to their widespread usage. Our architecture could be adapted to other: open-source FaaS frameworks (e.g. OpenFaaS, Knative); P2P structured overlays, e.g., Chord, CAN; P2P storage, e.g., Freenet.

Finally, FaaS@Edge provides a CLI tool to consume the REST API, similar to OpenWhisk's CLI tool, that allows users to perform the following operations: `Start` - Start running a new FaaS@Edge node; `Exit` - Shut down the instance node; `Submit` - Submit a user function in FaaS@Edge.; `Invoke` - Invoke a function previously submitted in FaaS@Edge.

3 Evaluation

To evaluate the FaaS@Edge prototype, we used a configuration consisting of a cluster deployment setup, illustrative of an edge deployment, of 1 to 15 virtual

machines with 2 vCPUs and 2048MB of RAM and a remote client node on a geographically distant machine with 2 vCPUs and 4096MB of RAM. In order to test our system accordingly, we used FaaS workload functions that we developed, using the Go language, to be supported by our prototype and use some typical FaaS scenarios [18, 21]:

- **Content Hashing** - Receives data contents as a function parameter and generates the SHA256 hash of that content. The resulting hash is returned to the user if requested.
- **Database Query** - The user can request the initialization of an in-memory database that stores information regarding a library's books in JSON format. Then, the user can query the database for any specific book by passing its International Standard Book Number (ISBN) as a parameter.
- **Image Transformation** - Receives a public image URL which is used to get the image data using HTTP. Then, performs a transformation to flip the image vertically and returns the image data in base64 format.

Our study compared FaaS@Edge's overhead and performance with local OpenWhisk on edge devices, analyzing function latency, bandwidth, CPU, memory usage, and success rates. We examined how different offering plans, particularly the Balanced plan, affected these metrics across six deployment setups.

- Local deployment of OpenWhisk on a single node instance;
- One client node and one supplier node on the same machine;
- One client node and one supplier node on remotely distant machines;
- Five nodes with two supplier nodes and three client nodes on the same machine;
- Ten nodes with five supplier nodes and four client nodes on the same machine, and a client node on a remote machine;
- Fifteen nodes with eight supplier nodes and six client nodes on the same machine, and a client node on a remote machine.

The remainder of this section presents the results of our evaluation.

Function Latency Figure 3 presents the distribution of the submission latency times for each of the deployments mentioned previously, measured since the client nodes sent the submission requests until an answer was received, excluding the time it took the supplier node available to create the function in OpenWhisk. The values observed are situated between the interval of 0.02s and 0.1s, and the lower latency values belong to the 2 nodes and 5 nodes deployments, and higher values correspond to the 15 nodes deployment.

The function memory values specified in a submission request have an important role in our algorithms to select the available provider, contrary to the function types that have no influence, but the results returned relatively close values of overhead time, which indicates a leveled distribution of the different sizes of resources as a result of our offering strategy.

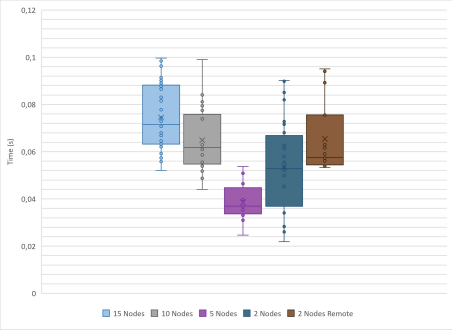

Fig. 3. Submission latency times per nodes (Box plot).

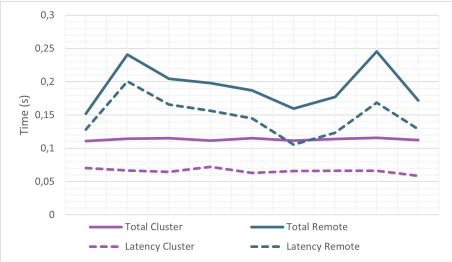

Fig. 4. Submission times comparison between client node in cluster machine and remote machine.

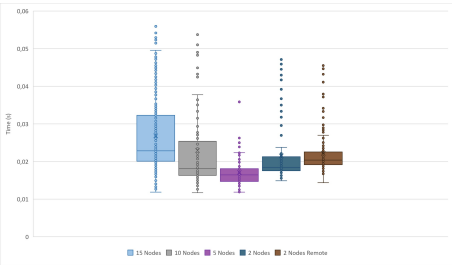

Fig. 5. Invocation latency times per nodes (Box plot).

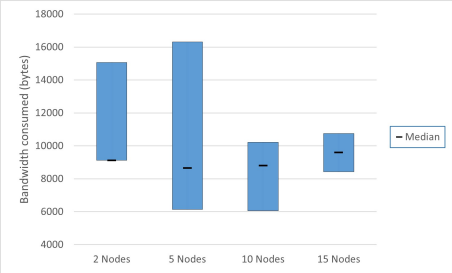

Fig. 6. Bandwidth consumed per nodes.

Figure 4 provides a comparison between the submission times obtained by client nodes located in a cluster machine, where the supplier nodes are also running, and the client node in a remote machine. A remote client node spends ≈70% more time during resource discovery and/or exchanging of messages to fulfill the submission request.

Figure 5 presents the distribution of the latency times obtained for each of the deployments, again, excluding the time it takes for the function to execute in OpenWhisk. The results fitted all within an interval of 0.04s, as predicted since the invocation request has no additional overhead from resource discovery or scheduling algorithms, already handled during the function's submission.

These results only consider *warm start* invocations, where there is already a running container, as a way to normalize their averages. The image transformation function (that is more CPU demanding) revealed a significantly higher total invocation time than the rest, which was spent in OpenWhisk. The function memory allocation values do not cause significant implications on the total and latency invocation times.

Contrary to what we witnessed with the submission times, the remote client took only 2.9% more invocation total time, indicating that the physical distance

between nodes can have an impact on IPFS' lookup protocol during the resource discovery but does not impose a lot of added time on the execution of invocation requests (maintaining an acceptable network throughput).The results show that using FaaS@Edge is slower than local deployment but still offers practical benefits. Submissions through FaaS@Edge take about 90.9% longer than local deployment, while invocations are 25.5% slower. Despite this, the slight delay in FaaS@Edge can be deemed acceptable, particularly as it allows less powerful edge nodes to leverage the FaaS model without relying on cloud providers. The minimal performance loss is especially favorable considering users are likely to make more invocation than submission requests in FaaS@Edge.

Bandwidth consumed per node Figure 6 presents the overall bandwidth consumed by the supplier node instances in the different deployments during test executions with each fulfilling an arbitrary number of requests. Notice that the amplitude of bandwidth values decreases with the increase of nodes in the deployment and the median values are all situated between 8659B and 9604B (showing that increased scale improves load balancing to a significant extent). Bandwidth consumption over time typically suffered 2-3 increases of transmitted bandwidth by intervals of ≈3000B, with the exception of the image transformation function which also revealed an increase in the received data due to an HTTP request performed to retrieve the image data. The Bandwidth consumed per node did not show any direct relation to the number of requests a supplier node executed, thus we can simply conclude that the average consumption during the program's execution in an edge device is admissible and does not hinder the node's performance.

CPU and memory usage per node The CPU and Memory used per node metrics were retrieved periodically over time on both supplier and client nodes, during each test execution. The average CPU usage observed in supplier nodes and client nodes for each deployment gradually decreased from 5.61% to 2.31% as the number of nodes in the deployments increased, indicating an efficient utilization of the extra resources and good load balancing between the supplier nodes. The usage in client nodes is also significantly lower (averaging between 0.30%-0.80% CPU) than in supplier nodes seeing as the latter are the ones satisfying the requests and running the OpenWhisk platform thus using more processing power.

Figure 7 and Figure 8 show that functions with 128MB of memory consumed the most CPU and memory resources, which we attribute to increased memory-disk swapping, which degraded performance. Despite this, the overall resource consumption for all memory allocations was deemed reasonable, ensuring that edge devices remain efficient for other functionalities while partaking in the FaaS@Edge network.

Request Success Rate The request success rate measures how many user requests to submit and invoke a function the FaaS@Edge system was able to suc-

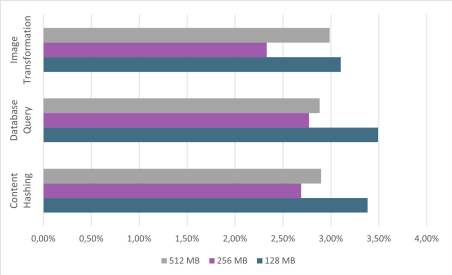

Fig. 7. CPU usage per node and memory value for each function type.

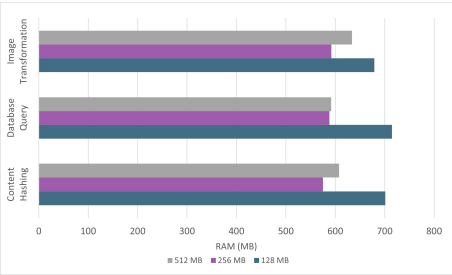

Fig. 8. Memory used per node and memory value for each function type.

	Request Success Rate	
Function Type	**Submission**	**Invocation**
Content Hashing	99.49%	100.00%
Database Query	95.16%	94.98%
Image Transformation	100.00%	100.00%
Function Memory		
128 MB	95.24%	97.28%
256 MB	99.49%	98.73%
512 MB	100.00%	100.00%
Total Requests	98.76%	98.69%

Table 1. Request Success Rate by Function Type and Memory

cessfully fulfill, which directly translates into the resource discovery and scheduling algorithms' effectiveness and, in turn, the user's satisfaction. Table 3 shows varying success rates for different FaaS functions and memory sizes during tests. Image transformation had the highest success rate, while database queries saw more failures, often due to supplier node crashes, not resource issues. The offering plan had minimal effect on bandwidth, CPU, or memory, suggesting stable resource availability and demand-supply balance, without network scaling or churn impacting resource allocation.

4 Related Work

Function-as-a-Service, first presented by Amazon, in the form of Lambda functions, allows the consumer to run their function code automatically, at a more fine-grained level, when a request occurs, i.e., an event is triggered, without having to provision virtual machine instances or monitor and upgrade the system. A widely used open source example of this technology is Apache OpenWhisk.

WOW [11] is a prototype for a WebAssembly runtime environment, as a lightweight alternative to traditional container runtimes, designed mainly for serverless computing at the edge. It introduces the components to support the WebAssembly runtime, similar to Docker's container runtime support, using the Apache OpenWhisk framework.

Our work is also related to volunteer computing approaches, where users offer their unused resources to build a large computational infrastructure. A well-known example is SETI@home [2], a volunteer computing project that uses Internet-connected computers to analyze radio signals in search of extraterrestrial intelligence. It uses the BOINC [1] software platform for volunteer computing, but the system is only designed for this specific set of applications, although there are other extensions of BOINC for cycle-sharing applications, such as the nuBOINC system [19]. Caravela [17] is a completely decentralized Edge Cloud system that utilizes volunteered user resources where users can deploy their applications using Docker containers. Peers in Caravela can act as suppliers, publishing offers to supply their resources, buyers, searching for resource offers in order to deploy a container, or traders, registering and mediating the offers made within their resource region.

As computational progress evolves rapidly on a global scale with the emergence of increasingly more powerful processors, cloud storages have been more sought after to handle these data management functions. However, the typical characteristics of centralized management and single-entity infrastructure providers which are linked to cloud storages may pose several privacy and security concerns and threaten data accessibility and availability [9]. Peer-to-Peer Data Networks aim to overcome these issues by creating overlay networks where peers can autonomously share their resources with each other. While other data-sharing and content distribution approaches like Content Delivery Networks [14], that addressed the lack of dynamic management of Web content, focus on fulfilling the customer's (often a company) requirements for performance and Quality-of-Service, Peer-to-Peer Data Networks' main goal is to efficiently locate and transfer files across peers (often final users) [15].

IPFS [7] is a decentralized file system merging DHTs, block exchanges, and version control to create a peer-to-peer network. It uses a Kademlia-based DHT for peer discovery and content location, with data stored in content-addressed chunks forming a Merkle DAG for retrieval. The BitSwap protocol manages data distribution, where peers exchange lists of content identifiers for the chunks they want or offer. Support for publish-subscribe notifications was also added [4].

The previous systems address some of the aspects that we tackle in our work, but none achieves the implementation of all aspects. Apache OpenWhisk is a framework for FaaS deployments, but it was not intentionally designed to maintain performance in an edge environment and does not feature content distribution. WOW targets wasm that could run at browsers at the edge, but it focuses mostly on reducing cold starts and does not address decentralization at the edge specifically. Caravela uses a peer-to-peer network with similar capabilities as IPFS and introduces the execution of long-running container applications,

however, it is not designed for FaaS deployments. SETI@home uses large-scale volunteer computing, but still relies on a centralized server. IPFS focuses on content storage and distribution, which is highly important in peer-to-peer edge environments but involves no computation execution by itself.

5 Conclusion

In this study, we introduced FaaS@Edge, a novel decentralized framework designed to implement Function-as-a-Service (FaaS) within edge computing environments by utilizing the resources of edge nodes to deploy user functions via Apache OpenWhisk. Our observations revealed that while the middleware introduces a significant bootstrapping overhead (nearly doubling the latency time for function submission compared to a local OpenWhisk deployment), the invocation times remained comparably low, which is advantageous considering the typically higher frequency of invocations relative to submissions. Moreover, the system's bandwidth consumption, CPU, and memory usage were found to be within acceptable ranges for edge devices, and FaaS@Edge demonstrated a high success rate in function deployment and execution.

Regarding future work, our objectives include enhancing FaaS@Edge's functionality by developing a mechanism that allows users to deploy functions using methods beyond source code files and custom runtimes, such as Docker containers or binary-compatible executables.

Acknowledgements: This work was supported by national funds through FCT, Fundação para a Ciência e a Tecnologia, under project UIDB/50021/2020 (DOI:10.54499/UIDB/50021/2020). This work was supported by: "DL 60/2018, de 3-08 - Aquisição necessária para a atividade de I&D do INESC-ID, no âmbito do projeto SmartRetail (C6632206063-00466847)". This work was supported by the CloudStars project, funded by the European Union's Horizon research and innovation program under grant agreement number 101086248.

References

1. D. P. Anderson. Boinc: a platform for volunteer computing. *Journal of Grid Computing*, 18(1):99–122, 2020.
2. D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.
3. A. Antelmi, G. D'Ambrosio, A. Petta, L. Serra, and C. Spagnuolo. A volunteer computing architecture for computational workflows on decentralized web. *IEEE Access*, 10:98993–99010, 2022.
4. J. Antunes, D. Dias, and L. Veiga. Pulsarcast: Scalable, reliable pub-sub over P2P nets. In Z. Yan, G. Tyson, and D. Koutsonikolas, editors, *IFIP Networking Conference, IFIP Networking 2021, Espoo and Helsinki, Finland, June 21-24, 2021*, pages 1–6. IEEE, 2021.
5. O. Ascigil, A. G. Tasiopoulos, T. K. Phan, V. Sourlas, I. Psaras, and G. Pavlou. Resource provisioning and allocation in function-as-a-service edge-clouds. *IEEE Transactions on Services Computing*, 15(4):2410–2424, 2021.

6. I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. Serverless computing: Current trends and open problems. In *Research advances in cloud computing*, pages 1–20. Springer, 2017.
7. L. Balduf, M. Korczyński, O. Ascigil, N. V. Keizer, G. Pavlou, B. Scheuermann, and M. Król. The cloud strikes back: Investigating the decentralization of ipfs. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, IMC '23, page 391–405, New York, NY, USA, 2023. Association for Computing Machinery.
8. L. Baresi and D. F. Mendonça. Towards a serverless platform for edge computing. In *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 1–10. IEEE, 2019.
9. E. Daniel and F. Tschorsch. Ipfs and friends: A qualitative comparison of next generation peer-to-peer data networks. *IEEE Communications Surveys & Tutorials*, 24(1):31–52, 2022.
10. F. Freitag, L. Wei, C.-H. Liu, M. Selimi, and L. Veiga. Server-side adaptive federated learning over wireless mesh network. In *International Conference on Information Technology & Systems*, pages 289–298. Springer, 2023.
11. P. Gackstatter, P. A. Frangoudis, and S. Dustdar. Pushing serverless to the edge with webassembly runtimes. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 140–149. IEEE, 2022.
12. E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.
13. A. Mathur, D. J. Beutel, P. P. B. de Gusmao, J. Fernandez-Marques, T. Topal, X. Qiu, T. Parcollet, Y. Gao, and N. D. Lane. On-device federated learning with flower. *arXiv preprint arXiv:2104.03042*, 2021.
14. G. Pallis and A. Vakali. Insight and perspectives for content delivery networks. *Communications of the ACM*, 49(1):101–106, 2006.
15. A.-M. K. Pathan, R. Buyya, et al. A taxonomy and survey of content delivery networks. *Grid computing and distributed systems laboratory, University of Melbourne, Technical Report*, 4(2007):70, 2007.
16. T. Pfandzelter and D. Bermbach. tinyfaas: A lightweight faas platform for edge environments. In *2020 IEEE International Conference on Fog Computing (ICFC)*, pages 17–24. IEEE, 2020.
17. A. Pires, J. Simão, and L. Veiga. Distributed and decentralized orchestration of containers on edge clouds. *J. Grid Comput.*, 19(3):36, 2021.
18. P. Raith, S. Nastic, and S. Dustdar. Serverless edge computing—where we are and what lies ahead. *IEEE Internet Computing*, 27(3):50–64, 2023.
19. J. N. Silva, L. Veiga, and P. Ferreira. nuboinc: Boinc extensions for community cycle sharing. In *2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 248–253. IEEE, 2008.
20. C. Systems. Fog computing and the internet of things: extend the cloud to where the things are. White paper, 2016.
21. V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller. Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, UCC'19, page 273–283, New York, NY, USA, 2019. Association for Computing Machinery.