A Unifying, Lightweight Platform for Microservice and Serverless Deployments

Saverio Giallorenzo[1,2] · Claudio Guidi[3] · Luca Tagliavini[1]

[1] Alma Mater Studiorum - Università di Bologna, Bologna, Italy
[2] INRIA, Sophia-Antipolis, France [3] italianaSoftware s.r.l., Imola, Italy

✉ luca.tagliavini5@studio.unibo.it

1 Introduction

The two complementary software architectural styles of microservices [5] and serverless functions (FaaS) [8] represent the state-of-the-art in cloud architectures. In both styles, developers break down the components of a cloud application into small software units. A microservice implements a cohesive set of operations, while a function handles a single operation. Both microservices and functions can be efficiently replicated/scaled. In the case of microservices, existing methods [10, 6, 11, 3, 4, 2, 1] determine a threshold, such as performance or traffic, which when exceeded, leads to the replication or deduplication of microservice instances. Since, in microservices, a process is constantly available to handle new requests, this style achieves small per-invocation costs when the traffic is steady (given an allocation of resources proportional to the traffic). However, when there are no inbound requests, microservices are "idle", wasting money and resources. Upscaling and downscaling policies address issues related to traffic fluctuation, but they complicate the deployment logic of the application. FaaS scales based on the number of inbound requests [7] since each new call triggers the allocation of an instance of the specific function on a cloud node. Hence, the one-request-one-allocation approach of FaaS makes scaling implicit and automatically determined by the amount of inbound traffic—so that, when there are no requests, no functions are running, and no resources are wasted for their execution. Unfortunately, current cost models oppose steady-traffic configurations, exacting a disproportionate price w.r.t. their microservice counterparts.

It is difficult to predict what kind of traffic a given architecture will endure—and it is likely to sustain different traffic shapes during its lifecycle phases, if not at different times of the same day. Nevertheless, developers need to commit early on which style to use when they start developing their system (and adapting a system developed in either style to the other can be a costly decision).

In this abstract, we present JFN, a platform that supports the deployment of microservices as serverless functions (with minimal adjustments, e.g., we ask the developer to indicate which of the operations of the microservice the function exposes) as well as the execution of functions as microservices under certain conditions (e.g., when receiving a steady stream of requests within a given timeframe). To tackle the objective of JFN, we leverage the linguistic constructs of the service-oriented language Jolie. In Section 2, we provide the essential details on the architecture of JFN to support the hybrid execution of microservices and serverless functions. In Section 3, we overview related work close to JFN and consider future evolutions of the platform. JFN is an open-source project available at `https://github.com/lucat1/jfn`.

2 The Architecture of JFN

JFN implements a lightweight serverless runtime, supporting both the function and microservice execution modalities. The main components of the JFN architecture are four.

The *Gateway* receives operation calls and redirects them to the appropriate *Executor*, which will consume the request either executing the associated function or passing it to the running microservice. The Gateway, interacting with the Provisioner, also acts as a load balancer among the Executors.

The *Provisioner* takes care of scaling the number of Executors and determines the routing for each function call, performing the actual distribution of the load. This component also handles the FaaS-microservice adaptation logic; it monitors the number of calls per-function and per-executor, necessary to determine when a function should be converted to a microservice. Once adapted to the microservice execution modality, the Provisioner gives priority to a microservice instance.

JFN considers two kinds of *Executors*: a Runner and a Singleton. The Runner uses the `embed` feature of the Jolie language to perform a FaaS-like, one-shot run of a microservice, invoking the operation selected by the developer at deployment time. On the contrary, a Singleton works as a wrapper for the aforementioned microservice, loading and executing it in continuous execution mode. To help illustrate the components of the architecture and explain the difference between the two runners, we depict a schema of JFN's architecture in Figure 1. There, the Runners r_1 and r_2 contain multiple running functions (the thread-like icons within the Runner's shapes). Contrarily, j_1 and k_1 are Singleton microservice-mode instances of the related functions.

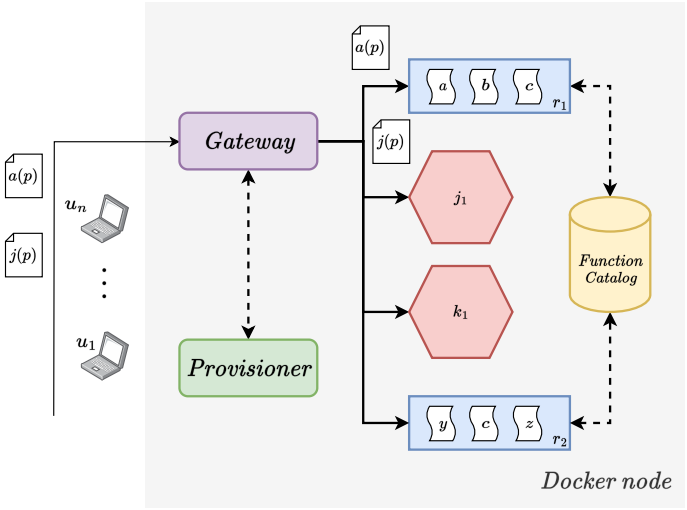

Figure 1: JFN architecture and invocation flow of two function calls: one computed as a function $a(p)$, one redirected to its microservice $j(p)$.

The last main component is the *Function Catalog*, which stores the code for functions and allows developers to deploy new ones on and remove existing ones from the platform.

As mentioned, the distinguishing feature of JFN is its ability to dynamically convert the execution modality of a function into a microservice, which offers better cost/performance when dealing with steady loads. This flexibility is supported by the fact that each function in JFN is a Jolie service, so it can be directly executed, with some minor adjustments. Notably, in section 2, we find the gray area encompassing all the platform's components and running functions tagged as "Docker node". This annotation showcases a possible installation of the JFN architecture, which uses Docker to containerise the architecture and simplify the deployment of the platform.

Invocation protocols. To conclude our overview of JFN, we briefly present the invocation protocols followed by the platform. In Figure 1, we show all the components of the JFN architecture, and exemplify their interaction focusing on the execution of two functions. In the figure, we represent Runners as blue boxes, functions as red hexagons, users as laptops, and invocation requests as sheets.

The first function is a, whose invocation is requested with parameter p and represented in the

figure as $a(p)$. The invocation is executed by the Runner r_1 in function mode. The second request, $j(p)$, is handed to the dedicated j_1 Singleton service. A function may be executed by any Runner, but, with sufficient load, it can also have multiple dedicated Executors. This whole process is transparent to the users, who see both $a(p)$ and $j(p)$ as two invocations to the Gateway.

When a function invocation request is sent to the Gateway, it asks the Provisioner which Executor should execute it. Then, the request is relayed to the designated Runner or Singleton. While a Singleton will immediately perform the computation, a Runner may need to fetch the function's code from the Catalog, then *embed* it—using the namesake linguistic construct/runtime provided by the Jolie language—and finally execute it. After the execution of a function on a Runner, the function is unloaded, while its code is temporarily cached for possible subsequent invocations.

Within a given timeframe (e.g., every minute), the Provisioner evaluates whether to allocate one or more Singletons for a function, based on the number of calls received in the past timeframe. At startup, a Singleton fetches the function's code from the Catalog and embeds it permanently, configuring a `redirection` so that the embedded function service can be reached.

3 Discussion and Conclusion

As far as we know, there is only one proposal similar to JFN, by Li et al. [9]. Specifically, the platform by Li et al. solves the problem of reconciling serverless- and microservice-style executions by running a given microservice in "serverless mode", executing it when new requests arrive until it fulfilled those invocations. We see the approach by Li et al. as complementary to ours. Indeed, JFN takes a function-first stance (allowing developers to deploy microservices as functions if needed) and then automates the microservice-style deployment according to the experienced traffic shape. Performance-wise, we hypothesize that the platform by Li et al. could suffer from inefficiencies e.g., due to the unnecessary allocation of resources for a whole microservice (which usually encompasses multiple operations, each with its code, dependencies, and potential connections to other resources such as databases and dependent microservices; resulting in possible chains of allocations).

Regarding future work, we envision working on distributing the JFN platform across multiple clusters, which would enhance its scalability and robustness, so that, by leveraging the capabilities of distributed systems, we can ensure efficient resource usage and handle increased workloads effectively.

Another area is the improvement of scaling policies. These policies play a crucial role in dynamically adapting the system to varying demands. We aim to enhance the existing policies and develop new ones that not only consider factors such as request frequency and resource availability but also take into account the characteristics of the functions/microservices themselves. This includes determining the optimal conditions under which a function should be executed in service mode, considering additional factors like data dependencies and performance requirements.

While our main focus regards the definition and implementation of the JFN architecture, we also plan to provide developer support when adapting an existing microservice into a function. We deem the choice to use Jolie on point. Indeed, the language provides explicit, idiomatic ways to express intra-session coordination within a microservice's code (e.g., accessing the `global` state of the microservice). This linguistic support expedites the implementation of programming aids such as linters and code analysers able to indicate when it is feasible to deploy a microservice as a stateless function and provide hints about which elements of a microservice the user needs to change to make sure that the deployed function would preserve the same behaviour when switching between the function and microservice execution modalities.

References

[1] Lorenzo Bacchiani, Mario Bravetti, Maurizio Gabbrielli, Saverio Giallorenzo, Gianluigi Zavattaro, and Stefano Pio Zingaro. Proactive-reactive global scaling, with analytics. In Javier Troya, Brahim Medjahed, Mario Piattini, Lina Yao, Pablo Fernández, and Antonio Ruiz-Cortés, editors, *Service-Oriented Computing - 20th International Conference, ICSOC 2022, Seville, Spain, November 29 - December 2, 2022, Proceedings*, volume 13740 of *Lecture Notes in Computer Science*, pages 237–254. Springer, 2022.

[2] Lorenzo Bacchiani, Mario Bravetti, Saverio Giallorenzo, et al. Microservice dynamic architecture-level deployment orchestration. In *COORDINATION 2021*, LNCS. Springer, 2021.

[3] Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, et al. Optimal and automated deployment for microservices. In *FASE 2019*, pages 351–368. Springer, 2019.

[4] Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, et al. A formal approach to microservice architecture deployment. In *Microservices, Science and Engineering*, pages 183–208. Springer, 2020.

[5] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017.

[6] Alim Ul Gias, Giuliano Casale, and Murray Woodside. Atom: Model-driven autoscaling for microservices. In *2019 IEEE ICDCS*, pages 1994–2004. IEEE, 2019.

[7] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, et al. Serverless computing: One step forward, two steps back. In *CIDR 2019*. www.cidrdb.org, 2019.

[8] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[9] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408. IEEE, 2020.

[10] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[11] Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti. Hierarchical scaling of microservices in kubernetes. In *ACSOS*, pages 28–37. IEEE, 2020.