IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 8 August 2023, accepted 12 September 2023, date of publication 9 October 2023, date of current version 16 October 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3322936

RESEARCH ARTICLE

FaaS-Oriented Node.js Applications in an RPC Approach Using the Node2FaaS Framework

LEONARDO REBOUÇAS DE CARVALHO AND ALETÉIA PATRICIA FAVACHO DE ARAÚJO
Department of Computer Science, University of Brasília, Brasília 70910-900, Brazil
Corresponding author: Leonardo Rebouças de Carvalho (leouesb@gmail.com)

ABSTRACT The Function-as-a-Service (FaaS) service model has aroused great interest since its introduction in the context of cloud computing. Although FaaS can be used to perform isolated tasks, it is in the composition of applications that this service model can promote significant performance improvements. This work presents an evolution in the Node2FaaS framework, whose objective is to assist in the conversion of originally monolithic node.js applications to work with FaaS in the Remote Procedure Call (RPC) communication mechanism. The new implementations in the Node2FaaS framework allowed new experiments to be conducted. Those showed significant gains in runtime for applications with CPU-bound, memory-bound, I/O-bound characteristics, especially for a Bioinformatics application that aims to align genetic sequences.

INDEX TERMS Code transformer, FaaS, function-as-a-service, node.js, Node2FaaS, serverless.

I. INTRODUCTION

With the aim of significantly reducing costs and time for providing infrastructure, cloud computing has gained an important place in today's society. Before this goal became a reality, a considerable investment in datacenters was essential to support the processing and storage necessary for the start and continuity of projects that demanded computational resources. Service models that encapsulate both the infrastructure and the development platform itself have gained ground. These models give the developer an interface for including source code, written in the most common programming languages. One of the models promotes event-triggered stateless execution with high parallelism. This service model is called Function-as-a-Service (FaaS) [1], but it can also be referred to as a serverless model, since the user has the impression that there is no server to worry about.

Despite the benefits that the FaaS offers, it is necessary to adjust the development processes to suit this new model. In addition, certain computational problems may suffer from an increase in execution time when replacing their technology stacks with a FaaS-oriented approach, since this technology increases the total number of layers of the original solution. Moreover, each provider may have a particular way of offering interaction with their services, and the developer will need to become familiar with these interfaces. In this context, ''FaaSification'' is the process of converting a code structure into a format which is executable on a Function-as-a-Service platform [2]. Some proposals are available to perform FaaSification, such as: Zappa [3], PyWren [4], Lithops [5], ToLambda [6], DaF [7], M2FaaS [8], however none of them has the characteristics that Node2FaaS does, in particular the adoption of a multiclod orchestrator, among other characteristics, as will be presented in Section V.

This work presents advances in the Node2FaaS framework that allowed its use on a real application and a deeper evaluation of the benefits of the framework approach, including a cost analysis.

In the preliminary version [9], a RPC approach was introduced with the aim of incorporating the benefits of FaaS, such as elasticity and underlying infrastructure managed by the provider and cost-efficiency. By transferring responsibility for scaling as well as infrastructure management to the provider, developers can stay focused on addressing the software's core purpose. Furthermore, as charging is based on activations, the cost is adjusted to real demand. Even in the initial version, use cases were developed to

The associate editor coordinating the review of this manuscript and approving it for publication was Somchart Fugkeaw.

stress certain aspects, such as CPU, memory and I/O. In the subsequent version [10], the concept of multi-cloud was introduced through the adoption of the Terraform [11] cloud orchestrator. In this work, the Node2FaaS framework was used to convert an application for genetic sequence alignment. Both applications were then submitted to batteries of tests with different levels of concurrency and the results of this experiment complemented the results obtained in the experiments carried out previously, plus the respective cost analysis of each experiment.

There are a few initiatives similar to Node2FaaS such as Lambada [2] and Podilizer [12]. Therefore, the authors of these projects launched a portal to publicize FaaSification [2] initiatives and ensure a better comparison between the different approaches and implementations. This portal is available at http://www.faasification.com and is open to receive suggestions and new projects. The idea is to bring together the initiatives and encourage the development of this area.

This article is divided into 7 sections, the first being introductory. Section II provides the fundamental conceptual basis for this work, while Section III presents the proposal for this paper. In Section IV, the results of the experiments carried out are presented. Section V addresses the related works, Section VI presents a discussion on the topic and, in Section VII, final considerations are presented and future work is discussed.

II. BACKGROUND

In the traditional model of computing, resources such as processor, memory, disk and connection to the network are physically handled [13]. The maintenance and scale processes in this type of approach, besides being expensive, are laborious and slow. As computing has evolved, and a considerable amount of computing power has become available on just one machine, there has been a need to improve the internal management of these resources. In this scenario, resource virtualization was the solution found and this created the opportunity to commercialize the excess idle computing capacity existing in the datacenters and so cloud computing emerged.

A. CLOUD COMPUTING

In the early days of cloud computing, National Institute of Standards and Technology (NIST) defined only three service delivery models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [14]. These three models are able to define any type of computational resource available for hire on the Internet, be it the most basic such as a disk, CPU, network, memory or any other resource of the IaaS model. It might also be a development platform configured and managed by the provider, as in PaaS model services, or fully functional software, available for use by the end user without any effort to configure the infrastructure on the part of the customer, such as SaaS models.

The market for cloud providers has become more and more competitive over time. Currently, there is a profusion of service offerings at different providers. Many of them offer similar products but using different approaches. In this scenario, a multicloud approach has grown within organizations, in which resources offered by different providers make up the technological support framework of these organizations. This approach has been called multicloud [15].

Multicloud [15] can be defined as a level above cloud computing, since its resources are dynamically provisioned at different providers and can also be found under the denominations of sky computing [16], intercloud [17], and cross-cloud [18]. Although each of these concepts has its own particularities, in this paper they will be treated as synonymous with multicloud.

In order to manage the huge pool of resources that this concept can reach, and considering the different forms of consumption, the need arose to develop a mechanism capable of systematically managing these computational resources, such as xAFCL [19], as well as enabling the quick and efficient execution of provisioning of environments across multiple services from different providers. This phenomenon causes difficulties in migrating from one provider to another, and even between different services from the same provider. In this context, considering the need to manage environments made up of resources spread over several providers, and given that each provider has a particular way of offering its services, a specific segment of Infrastructure-as-a-Code (IaC) tools has grown, the multicloud orchestrators.

B. MULTICLOUD ORCHESTRATORS

Cloud orchestrators can be defined as platforms on which the developer creates an infrastructure descriptor that the orchestrator follows to implement autonomously, without any interaction with the user [20]. In other words, cloud orchestrators are deployment technologies that can automatically deploy software components based on deployment models (including their orchestration).This concept ensures that an updated and tested infrastructure descriptor is available for developers to begin the infrastructure deployment process, simply providing the orchestrator with the infrastructure descriptive code.

Orchestrators that can be found in the literature have certain characteristics, such as: year of appearance, license, strategy used, providers with which it has integration and the project URL presented in Table 1. In the Purpose column of Table 1, the solutions were classified as ''Crossplane'' when their projects indicated the intention of using them for proof of concept and/or academic purposes. On the other hand, solutions whose objective was considered more comprehensive, including potential for use in real environments, were classified as ''General''.

To act as a multicloud cloud orchestrator together with Node2FaaS, among those orchestrators mentioned, those with the greatest compatibility with Node2FaaS were

TABLE 1. Cloud orchestrators.

Orchestrator	Purpose	Year	License	Architecture	Providers
Roboconf [21]	Crossplane	2013	Open-source	Messaging queue-based	AWS, Azure, OpenStack, VMware, and CloudStack
Trans-Cloud [22]	Crossplane	2018	Open-source	TOSCA [23]	Agnostic
Live Cloud [24]	Crossplane	2012	Undefined	Event-driven	Tsinghua University (Private Cloud)
SALSA [25]	Crossplane	2014	Open-source	Master/agents with messaging queue	Openstack, flexian and strauslab
GRyCAP IM [26]	Crossplane	2013	Free Software	Monitoring-based	AWS, GCP, Azure, FogBow, OpenStack EGI Federated Cloud and Open Nebula
BioNimbuZ [27]	Crossplane	2017	Open source	Interactive	AWS, Azure and GCP0
The Celar Project [28]	Crossplane	2015	Eclipse Public License	TOSCA [23]	Agnostic
OpenTOSCA [29], [30]	Crossplane	2013	Open-source	TOSCA [23]	Agnostic
Cloudify [31]	General	2012	Proprietary	TOSCA [23]	Agnostic
Heat [32]	General	2013	Open-source	TOSCA [23]	Agnostic
Apache ARIA [33]	General	2018	Open-source	TOSCA [23]	Agnostic
CloudFormation [34]	General	2011	Proprietary	Templates-based	AWS
Terraform [11]	General	2014	Proprietary	Execution plan-based	More than 200
xOpera Orchestrator [35]	General	2019	Open-source	TOSCA [23]	Agnostic
Cloud Assembly [36]	General	2018	Proprietary	Iterative	VMware cloud services

TABLE 2. Results of tests carried out between Cloudify and Terraform [37].

		Aspect	Orchestrator	
			Terraform	Cloudify
Provider	AWS	**Managed Resources**	17	20
		Provisioning Time (min.)	3.3	4.3
		Unprovisioning Time (min.)	1.5	3.5
	GCP	**Managed Resources**	10	7
		Provisioning Time (min.)	4.6	4.3
		Unprovisioning Time (min.)	1.9	3.5
	Azure	**Managed Resources**	19	19
		Provisioning Time (min.)	6.2	5.1
		Unprovisioning Time (min.)	15.3	9.6

submitted to an extensive assessment in [37]. Considering the general characteristics, only Cloudify [31] and Terraform [11] would be eligible for adoption by the framework. After several batteries of tests (results in Table 2), the most suitable tool to meet the needs of Node2FaaS proved to be Terraform. As can be seen in Table 2, Terraform obtained better results in average provisioning and deprovisioning time in all providers, except GCP, where the average provisioning process took less time with Cloudify, but a difference of only 0.3 seconds. The complete analysis of this result can be seen in the article *Performance Comparison of Terraform and Cloudify as Multicloud Orchestrators* [37].

C. FUNCTION-AS-A-SERVICE

The traditional infrastructure management model needs to invest heavily in installation and configuration tasks, thus enabling an environment to receive source code and execute it properly. To rationalize this effort, cloud computing defined the service model PaaS [14], whose delivery consists of a platform completely configured and ready to receive the source code. However, this type of service still exposes some details of the infrastructure to the developer who, eventually, still needs to guarantee, among other aspects, the elasticity of the environment. In addition, it is necessary to maintain the provisioned environment during the development period, and this increases the final cost of the project. In this context, providers started offering a service model that delivers a completely encapsulated platform for software development and charges only for the processing actually performed by the platform. This type of service, known as FaaS [38] has been very interested recently, since its approach meets the architectural model that has become very widespread, the microservices [39] paradigm.

In FaaS [38] the customers contract the execution of a predefined function, load the code they want to execute, and can receive an address to access the service or simply check the expected result, for example, in database. Applications that use this type of cloud service have been called ''serverless'' applications, since there is no guarantee that at the time of the request there will be a server provisioned to meet the request and therefore these requests are handled by the provider in order to guarantee the existence of instances able to meet the request, including promoting auto-scaling in case of excessive demand.

Figure 1 shows the schema of Function-as-a-Service. This schema shows how the interaction between the developer and the FaaS service model takes place, whether it is actually loading code, and can use the Software Development Kits (SDKs) of the provider or simply interacts with the provider's console or CLI. The client can configure triggers for invocation of the service or make it on demand by requesting an API, for example. In addition, Figure 1 also

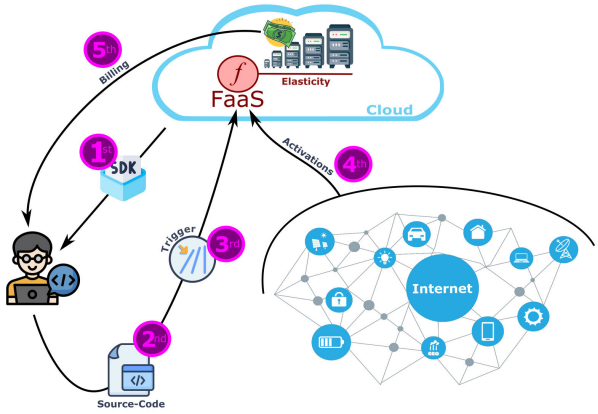


FIGURE 1. FaaS schema.

shows the elasticity of the service and its on-demand payment model (pay-as-you-go).

On the other hand, making use of the FaaS service model can be complicated, as it requires the developer to know the interface of consumption that each provider offers. In addition, it is necessary that the developer build the applications considering the use of this model, or invest considerable time adjusting applications already developed to work with FaaS.

Thus, to use the FaaS approach over an existing monolith application it is necessary that the developer knows the details of the APIs of each provider, as well as how to segment the functions of the application and convert them into calls appropriate to the structure of the service. This process can become tedious, and discourage developers from adopting a FaaS-based approach. With this, many professionals can lose the benefits that a microservice cloud-based architecture can offer, such as: high availability, resilience, cost reduction, among others. Furthermore, the programming language chosen to consume this computing paradigm can be decisive for the success or failure of this journey.

In this context, one can see that the JavaScript [40] language has gained a lot of importance for the success of web applications. Currently, this language has become a standard for including interactivity on Internet pages, and several frameworks have used it to increase the behavior of applications on the client side [41]. Thus, Node.js [42] emerged as an alternative for executing JavaScript code on the server side, and consequently its market as expanded. The wide adoption of JavaScript has offered Node.js a favorable scenario for its adoption, since the learning curve can be greatly mitigated by taking advantage of JavaScript knowledge. Even so, the constant evolution of software development models, as well as the adoption of new computing paradigms, such as the cloud, requires that the improvement processes of programming languages maintain a constant flow.

Since FaaS services do not always keep active instances able to process a request, it is common that in the first request or after some time without activations, a FaaS service presents a longer response time, this effect is known as "coldstart" [43]. As the focus of this work is not to deepen the understanding of this phenomenon, it was disregarded in the experiments through heating requests to the services before the batteries of tests.

Another feature that can impact the performance of FaaS functions is known as the "Spawn Effect" [44]. In this case, only a certain level of concurrency is maintained by the service, while subsequent requests have to wait for a slot to be released for processing. This characteristic can lead to degradation in the execution time, as well as in the triggering of timeouts culminating in failures. In this work, the experiments used concurrent requests, that is, a certain number of requests for the service were opened in sequence, making the processing occur according to the simultaneity supported by the provider.

Table 3 presents the characteristics of the main FaaS providers, such as supported programming languages, supported platforms, virtualization system used by the provider to deliver the service, time limits, memory and temporary storage, billing method, ways of activating the service and the middleware that supports the operation of the service. It is important to note that this is a very diversified market and providers offer similar solutions, but with many differences, which is why it is important to have mechanisms for onboarding these services in a less costly way and avoid vendor lock-in, that is characterized by expensive and time-consuming migration of application and data to alternative providers [45].

One of the important characteristics of FaaS services is to offer a runtime ready for execution, and this task involves delivering environments containing all the libraries that the application will need. Providers deliver some pre-installed libraries, as can be seen in Table 4, which shows the pre-installed libraries of Node.js, since this is the object language of this work. However, it is common for real applications to require libraries that are not part of this list. For these situations, some providers allow the user to declare a list of libraries that must be downloaded from the central repository, while others only allow these libraries to be part of the published package along with the function. In any case, it is essential to have a tool like Node2FaaS to manage and abstract (as far as possible) this complexity.

Considering the need for improvements in the software processing model in Node.js, this work brings improvements to the Node2FaaS framework. This is a tool for automatic and fruitful conversion of monolithic applications, written in Node.js, into FaaS-oriented applications, and will be described in more detail in the next section.

III. Node2FaaS FRAMEWORK

The Node2FaaS framework [9], [10] processes the original source code of a Node.js application offered as an input for its execution, and converts it to an application whose functions are executed in a FaaS service model. The internal code of the functions is converted into deploys created automatically at

TABLE 3. Characteristics of the main FaaS services. T = Time in minutes, M = Memory in gigabytes, and S = Temporary storage in gigabytes.

Service	Languages	Platforms	Virtualization	Limits			Billing method	Triggering	Middleware
				T	M	S			
AWS Lambda	Node.js, Python, Java, C#, Go, PowerShell, and Ruby	Linux and MacOS	Xen	15	3	0.5	Price per invocation and runtime	AWS events, HTTP APIs, and AWS API Gateway	AWS Elastic Container Service (ECS)
Google Cloud Functions	Node.js, Python, Go, .NET, Java, PHP, and Ruby	Linux	KVM	9	2	0.5	Price per invocation and runtime, or consumption-based pricing	Google Cloud events, HTTP APIs, and Pub/Sub	Knative
Azure Functions	C#, F#, Java, JavaScript, PowerShell, Python, and TypeScript	Windows, Linux, and MacOS	Hyper-V	10	1.5	1	Price per invocation and runtime, or consumption-based pricing	Azure events, HTTP APIs, and webhooks	Azure Functions Runtime
IBM Cloud Functions	Node.js, Python, Swift, Java, PHP, Ruby, Go, and .NET	Linux and MacOS	KVM	10	2	1	Price per invocation and runtime, or consumption-based pricing	IBM Cloud events, HTTP APIs, and webhooks	Apache OpenWhisk
Alibaba Function Compute	Node.js, Python, Java, PHP, C#, .NET Core, Ruby, and Go	Linux	Xen	10	3	0.5	Price per invocation and runtime, or consumption-based pricing	Alibaba Cloud events, HTTP APIs, and MQTT	Alibaba Cloud Container Service for Kubernetes (ACK)
Oracle Functions	Java, Node.js, Python, Go, Ruby, and .NET	Linux and Windows	KVM	10	2	1	Price per invocation and runtime, or consumption-based pricing	Oracle events, HTTP APIs, and webhooks.	Fn Project

TABLE 4. Node.js libraries pre-installed on FaaS providers.

Service	Pré-installed Node.js libraries
AWS Lambda	AWS SDK for JavaScript AWS X-Ray SDK Node.js runtime interface client (RIC) for AWS Lambda
Google Cloud Functions	Firebase SDK for Cloud Functions Google Cloud client libraries Google Cloud Firestore client
Azure Functions	Azure SDK Azure Functions core tools Azure Storage SDK
IBM Cloud Functions	BM Cloud SDK IBM Cloud Object Storage SDK OpenWhisk SDK
Alibaba Function Compute	Alibaba Cloud SDK Alibaba Cloud Function Compute SDK Aliyun OSS SDK
Oracle Functions	Oracle Cloud Infrastructure SDK Oracle NoSQL Database SDK Oracle Functions SDK

the provider. Calls to the API of the FaaS, which correspond to the original code of the functions, are replaced by their original definitions. The product of this processing is a new application, based on the original, whose effective execution of its functions does not occur in the environment in which the application is being executed, but in an external FaaS. Therefore, the framework has the following characteristics:

- **Ease of operation:** the installation process for the framework configures all the tooling necessary for its use. The user must configure only the information of the providers and Node2FaaS also assists in this task, collecting the credentials and creating the respective configuration files. This ease of adoption of framework aims to shorten the distance between users and the best results that the proposed model can deliver;
- **Automatic conversion:** the conversion process is completely automatic and does not require any interaction from the user. Once the target application is pointed out, Node2FaaS simply fulfils its mission, and in the end delivers a version of the original application, whose functions, if they are adherent, are executed in FaaS. The framework promotes an offloading for developers interested in FaaS consumption;
- **Function analysis:** for some types of algorithms there is no advantage in being executed in FaaS. Node2FaaS performs an analysis of the internal source code of each function, in order to define the best approach, either transferring the execution to an FaaS provider or maintaining the local execution;
- **Optimized execution:** the execution of the converted application must be the same or faster than the original. To guarantee this, the framework mixes different forms of treatment of the functions in order to obtain the best result in each one, optimizing the execution of the application in a generalized way;
- **Flexibility:** eventually the developer can deal with business requirements that require a change in the default behavior of the framework, from avoiding publishing some function in FaaS to guaranteeing its

publication. This flexible feature of Node2FaaS adds a larger set of applications to the list of candidates to use the framework;
- **Multiple providers:** considering that the concept of Sky computing is a rapidly growing approach, Node2FaaS allows publishing to multiple providers through the use of an integrated multicloud orchestrator;
- **Effectively usable:** Node2FaaS consists of a technological solution that can be effectively used in Node.js application development processes, serving both cross-plane and professional purposes;
- **Automatic dependency management:** the conversion process embeds the necessary dependencies in the package that will be published to the provider;
- **Open source tool:** the use license allows usage and modification of the software.

A. FaaS ADHERENCE ANALYZER

The processing of functions in FaaS inevitably impacts on the application's execution time, since a network must be crossed to request execution and obtain the result, be it a local network, as in services within the same zone within a provider, or much more distant, necessitating the crossing of the Internet. For this burden to be offset, the function to be performed must be sufficiently expensive to the point that it is more advantageous to transfer the execution to the provider and enjoy intense parallelism than to maintain the execution locally and impact the execution time due to the eventual queuing that can occur when there is a high number of competing requests.

In this work, a phase of checking the adherence of the functions to the FaaS paradigm is included in the conversion process. For this, two checks are carried out on the code:

- **Function source code size:** the framework checks if the function's source code size exceeds a threshold, before promoting it to FaaS;
- **Parameters usage inside loops:** if the framework notes the use of parameters within loops, there is a possibility they receive a high value and with that a high load will occur in the execution. In this case, the framework will decide to forward this processing to FaaS, since the function load will be affected by the value informed and potentially its execution time will be increased.

Both implementations could be verified in the framework's GitHub.[1] Although simple, this verification phase promotes an important filtering in the conversion process, avoiding very simple functions, whose execution time is low, their execution is impacted by the need to cross the Internet to obtain the result. However, this phase can be improved to verify other aspects of the function such as: complexity of the algorithm, behavior of the execution time in simulations using hypothetical values for the parameters, among other aspects. In addition, it would be appropriate to equalize the analysis process to the limitations of the destination provider for the execution time, size of the request, amount of memory required, etc.

[1] github.com/node2faas/framework/tree/0.2.14/lib/analyzer.js

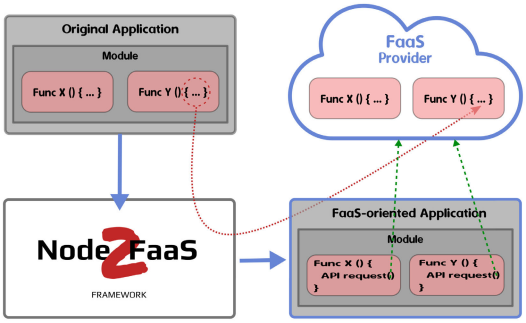


FIGURE 2. Node2FaaS architecture [10].

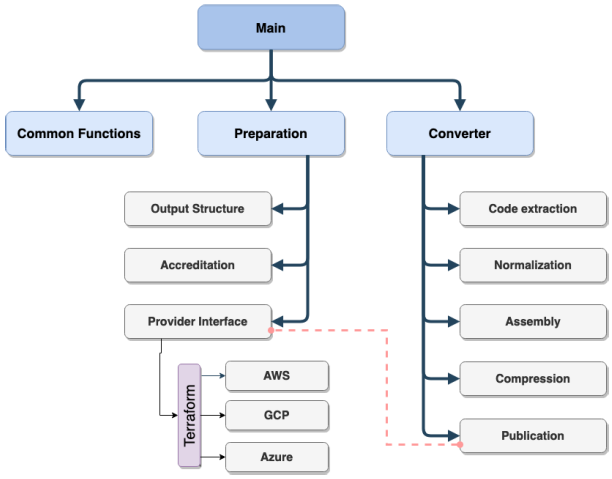


FIGURE 3. Node2FaaS composition [10].

B. Node2FaaS ARCHITECTURE

The solution architecture proposed by the Node2FaaS framework is based on the Remote Procedure Call (Remote Process Call, RPC) paradigm, which is a useful paradigm for providing communication over a network between programs written in high-level languages. Figure 2 presents the solution proposed by the framework in which it is possible to observe that in each module of an application there may be functions. Within each function there is code that performs some useful operation for the software. Once submitted to Node2FaaS, this code will be published in a cloud provider and, in its place, in the converted application, a URI call will appear pointing to the REST API provided by the provider as a result of publication. In short, the original code of the functions is transferred to the FaaS in the cloud and then consumed through requests using the HTTP protocol.

To carry out its mission, Node2FaaS is internally segmented into modules that fulfil specific tasks and integrate with each other. Thus, working in a coordinated manner, these modules receive inputs and return results that enable the processing of applications, and the generation of a new application using the proposed approach. Figure 3 shows the

internal structure of the framework, and it is possible to verify the existence of a main module, named index, whose role consists of coordinating the other modules. In addition to the main module, Node2FaaS is composed of the following modules:

- **Functions**: concentrate a set of utilities that are in common use among the other modules;
- **Preparation**: ensures that the requirements for the proper execution of the framework are met, and relies on the following sub-modules:
  - **Output structure**: responsible for creating the target application directory;
  - **Accreditation**: responsible for obtaining and storing credential information of providers;
  - **Interface with provider**: responsible for communicating with the providers' services and abstracting their complexities to the other modules. It has an internal segmentation to deal with the specifics of each supported provider.
- **Converter**: coordinates the conversion process and has the help of the following sub-modules:
  - **Code extraction**: responsible for extracting the internal source code of the functions;
  - **Normalization**: responsible for making the source code executable in the service of FaaS;
  - **Assembly**: responsible for assembling the new definition of functions after publication in the provider;
  - **Compression**: some services require that the code be compressed before publication, this module is responsible for performing this task;
  - **Publication**: responsible for requesting the publication from the interface module with the provider and handling their return.

C. Node2FaaS INTEGRATION WITH CLOUD

Once Terraform was elected as an orchestrator that would bridge the gap between Node2FaaS and the providers, the construction of the artifacts was carried out in order to build IaCs to define the architectural model (blueprints) of Terraform integration with each provider. Considering the Gartner magic quadrant [46], which indicates the cloud market leaders, the following were selected: AWS, GCP and Azure.

The Blueprint snippet 1 shows the AWS Lambda definition for Terraform. Other resource definitions such as API gateway, function source code deployment, and permissions can be verified on GitHub,[2] as well as the full blueprint definition for GCP and Azure.

Analyzing the blueprints for both providers, it is noticeable that the definition made for CGP is the one that needs the least explicit amount of terraform resources. On the other hand, the definition made for Azure is considerably greater than that made for AWS and, consequently, for GCP as well. This shows the existence of different degrees of complexity in the configuration of these resources and even in an eventual process like this involving human interaction, it would be highly prone to errors.

[2]github.com/node2faas/framework/tree/0.2.14/terraform/modules

```
28 resource "aws_lambda_function" "function" {
29   filename         = var.sourcecode_zip_path
30   function_name    = "node2faas-${var.name}"
31   description      = "Function ${var.name} automaticly created by node2faas"
32   role             = "${aws_iam_role.iam_for_lambda.arn}"
```

Blueprint snippet 1. Terraform source code for AWS Lambda.

Despite the need to build blueprints for each provider and thereby add a new layer to the Node2FaaS solution, this decision is justified by the fact that, if an integration were built using each provider's native API, it would be necessary to unveil the specifics of each one. In addition, this work could be highly unstable, since version changes to the APIs could destroy the entire integration. However, although it may still be so, the difference is that now there is an orchestrator supported by the community. Therefore, if this occurs, it is expected that a new version of the orchestrator will be released soon, correcting the problem.

D. Node2FaaS WORKFLOW

Once available in the local environment (after installation), the Node2FaaS flow is started from the execution of the ''node2faas'' application. If the framework was installed via Node Package Manager (NPM), this application will be registered in the path of the machine and can be run directly from the command *"node2faas –target [path to the application to be converted]"*. Otherwise, it will be necessary to access the directory where the framework was downloaded, grant permission to execute it, only then to run Node2FaaS. If the user does not know the options of the tool, he can enter the parameter - *help* (or -*h*). Therefore, the Node2FaaS framework offers the following functionality through its CLI:

- **help:** display of tool options;
- **clean:** removes local provider-related information as well as credentials. Thus, it will be necessary to fill this information again in a new execution of the framework;
- **destroy:** access the provider and promote the destruction of the functions that have been created;
- **verbose:** displays in detail the step-by-step processing of the framework;
- **provider:** allows the current default provider to be changed;
- **region:** allows the default region for the current provider to be changed.

Given the above, having an active account with a provider to perform the function is essential, since initially the framework seeks credentials to access the cloud. If the credentials file is not found, the application prompts the user to provide credential information for accessing cloud services. After that, the system will create the credentials file and will no longer request its completion for future executions, unless the user provides a parameter stating his intention.

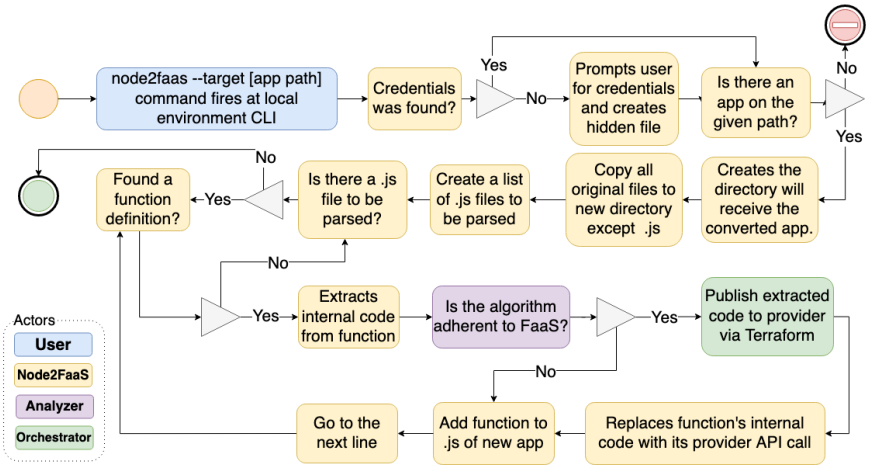


FIGURE 4. Node2FaaS application conversion workflow [10].

Once the credential is obtained and an application for conversion is offered by Node2FaaS, it is submitted to a conversion process that will analyze the application code looking for function definitions to perform the conversion, as shown in Figure 4. During the process, if a file include command (*include*) is found, then the target file is also searched for candidate for conversion, and this process is repeated recursively, until no file for inclusion is found.

This way, when the application finds a function, it checks if it is eligible for submission to the cloud. If so, preparation is made in the function code in order to normalize it to the functioning of the respective cloud, and then the code is delivered to the orchestrator. The orchestrator, in turn, provides access to the cloud to create a new FaaS. After receiving confirmation of the FaaS creation, the application obtains the associated URI to access the service, and creates the request within the definition of the original function. In this way, the function call remains unchanged and its operation on the cloud platform is done in a totally transparent manner.

In the end, Node2FaaS will have generated all the files that should make up the initial application, but with the original code of the functions replaced by HTTP requests to the FaaS of the cloud provider. The converted application maintains the same signature as the original functions, allowing its use to remain unchanged for the processes requesting the functions. This prevents the need to make adjustments to the application. The entire source code of the Node2FaaS framework and its hot page[3] link is available on GitHub[4] as well as in Zenodo FAIR repository.[5]

A preliminary version of the Node2FaaS framework was presented at [9] containing its main functionalities, such as the conversion process of node.js applications and automatic publication of functions in the AWS provider using the API of the provider itself. A multicloud approach was introduced to Node2FaaS through the incorporation of the Terraform cloud orchestrator and integration with FaaS solutions from AWS, Google and Azure [10]. Although the framework was able to prove that its RPC-oriented approach was able to considerably reduce the execution time of certain applications, these were specific to the experiments with the framework. In order to expand Node2FaaS's scope of action so that it could be applied to real problems, some improvements in the framework were necessary, such as:

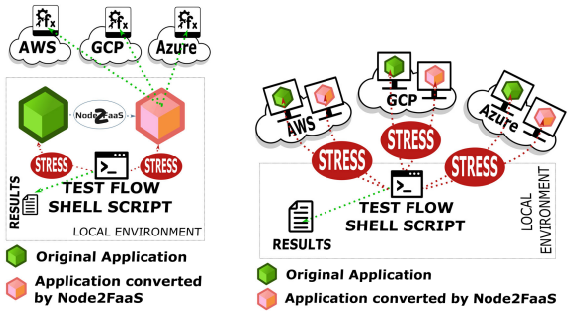
- AWS CloudWatch automatic integration;
- Update runtime from nodejs10.x (deprecated) to nodejs14.x;
- FaaS zip file generated with dependencies (node_modules);
- Replacing package Request (deprecated) by Axios;
- Automatically add Axios in generated app package.json;
- Parameters passed via POST;
- New annotation feature to force or skip publication on FaaS.

Considering the possibilities from the new functionalities of Node2FaaS, an experiment was elaborated with a real use case in which an application performs alignments of genetic sequences in parallel, with different levels of concurrency. The results of this real experiment complement the results already obtained previously, and both will be detailed in Section IV, as well as their respective cost analyses, which corroborate the approach proposed by Node2FaaS, both for specific applications for the tests and for real applications.

IV. RESULTS

In order to explore the potential of a Node.js application converted to FaaS, using the Node2FaaS framework, some experiments were conducted. The purpose of these experiments is to evaluate the behavior of an application whose processing is being forwarded to a cloud provider,

[3] node2faas.faasification.com
[4] github.com/node2faas/framework
[5] doi.org/10.5281/zenodo.7668232



(a) 1st and 2nd experiments. (b) 3rd experiment.

FIGURE 5. Architectures of the experiments.

as defined by the solution proposed by Node2FaaS and to ascertain if the framework is converting the applications properly.

A. METHODOLOGY

For the experiments, four functions were developed and used with one real application as test cases. The test cases are described below:

- **Math:** performs a simple sum operation;
- **CPU overload:** explores the server's processing power, stressing the CPU;
- **RAM memory overload:** explores the server's processing capacity, stressing RAM memory resources, through successive vector-filled loops;
- **Stress on disks:** exploits the server's processing capacity, stressing the input and output channel, through successive loops with creating and removing files;
- **Genetic sequence alignment:** this is a very common use case in Bioinformatics, for this work we used the BioSeq[6] library that implements the Smith Waterman [47] algorithm. This library was encapsulated by a node.js[7] application that acts as a REST API. Upon receiving an alignment request and two strings as parameter, the application triggers BioSeq to perform the alignment and then generate the respective result.

Once these test cases were encapsulated in a single application, three experiments were conducted. In all experiments, the application containing the test cases was submitted to the Node2FaaS framework in order to generate a new application using the Node2FaaS approach. Then, the initial application was run in a local environment and went through testing batteries. Likewise, the converted application also went through testing batteries and had its runtime performance measured in each round.

Figure 5a shows the architecture used in the 1st and 2nd experiments. This figure illustrates how the experiments were conducted. A shell script was used to execute the flow of the experiments and to command the various cycles of the test cases (stress), as well as to collect and store the data obtained for further analysis. In addition, it may be seen in the figure that while in the execution of the original application everything happens in the local environment, in the converted application there is an integration with the cloud provider to execute the operations in their respective FaaS. Figure 4b shows the architecture of the 3rd experiment in which two virtual machines were provisioned at each provider. The original sequence alignment application was installed on one of them while the application resulting from the conversion process of the original application by the Node2FaaS framework was installed on the other machine. As in the 1st and 2nd experiments, a shell script conducted the execution of the workloads referring to the concurrence tests, stressing the applications and collecting the results.

In the 1st experiment, the Node2FaaS conversion process did not have a cloud orchestrator and therefore only the AWS provider could be evaluated, as it was the only integration available. In the 2nd experiment, with the assistance of Terraform, it was possible to evaluate the behavior of the Node2FaaS approach in other providers, such as GCP and Azure, in addition to AWS. Furthermore, the method of application of concurrence between experiments has also been changed. While in the 1st experiment the concurrence varied continuously starting at 0 and ending at 100, in the 2nd experiment the values were fixed at 10, 50 and 120 concurrent requests. In the 3rd experiment, the genetic sequence alignments were requested in parallel in blocks of 1, 2, 4, 8, 16, 32, 64, 128, 256 and 512 simultaneous requests as can be seen in Table 5 which shows in detail the parameters used in each test case. In the 3rd experiment on-demand t3a.xlarge EC2 instances were used in AWS, in GCP on-demand e2-standard-4 compute engine instances were used, and in Azure on-demand standard_D4s_v3 virtual machines. These machine flavors in both providers have the equivalent amount of resources, such as memory and vCPU. Both machines were provisioned in near regions in order to avoid too much latency interference.

The test cases of the 1st and 2nd experiments were repeated 10 times, and the test case of the 3rd experiment was repeated 30 times to allow an average to be calculated. After carrying out the experiments, the resulting data were collected and consolidated so that they could be analyzed. An analysis of these results will be presented in the next sections.

Another interesting way to analyze the results is to evaluate the costs related to the test cases. Considering that the 3rd experiment adopted an experimental approach closer to real applications, a comparative analysis of the costs determined at each level of concurrence was carried out for the scenarios with and without the FaaS approach proposed by Node2FaaS in both providers. In the Table 6 are defined the parameters that are used to calculate the cost of the experiments, expressed in Equations 1 and 2 for the experiment without FaaS and with FaaS, respectively.

$$E_{withoutfaas} = T * C_{vm} \quad (1)$$

$$E_{withoutfaas} = (T * C_{vm}) + (T * C_{faas}) + (S * C_{request}) \quad (2)$$

[6]https://www.npmjs.com/package/bioseq
[7]https://github.com/node2faas/alignment

TABLE 5. Parameters of the experiments.

Exp.	Environments	Workloads	Concurrency	Single execution duration
1st [9]	Local and AWS	Math, CPU, Memory and I/O	0-100	≈ 1s
2nd [10]	Local, AWS, GCP and Azure	CPU, Memory and I/O	10 / 50 / 120	1s / 5s / 10s
3rd	Local, AWS, GCP and Azure	Genetic sequence alignment	1 / 2 / 4 / 8 / 16 / 32 / 64 / 128 / 256 / 512	N/A

TABLE 6. Cost analysis variables.

Variable	Explanation
T	Time spent in experiment
S	Concurrence level
C_{vm}	Virtual machine cost
C_{faas}	FaaS execution time cost
$C_{request}$	FaaS request cost
$E_{withoutfaas}$	Experiment without FaaS cost
$E_{withfaas}$	Experiment with FaaS cost

In Table 6 T represents the time spent on each experiment, which is understood from the beginning of the execution of the genetic sequence alignment to the receipt of the response by the client. S represents the different concurrency levels of the 3rd experiment, from 1 to 512. C_{vm} represents the amount charged by the provider for the use of the virtual machine used in the experiment for a certain time. C_{faas} represents the amount charged by the provider for a period of time of execution of its FaaS function. This value varies according to the amount of memory available for the function, in the case of the 3rd experiment all FaaS functions were created with 128MB of memory allocated. $C_{request}$ requires the fixed amount charged per request to the provider's FaaS service. $E_{withoutfaas}$ and $E_{withfaas}$ consist of the experiment cost calculations without FaaS and with FaaS, respectively, whose actual calculations are described in Equations 1 and 2. Table 7 presents the costs of provisioning services for virtual machines used in the 3rd experiment at each provider, as well as the respective values for using FaaS services configured for 128MB of RAM memory. In Table 7 use is expressed in minutes and the MacBook cost was estimated based on a five years of obsolescence in a cost of $1499. It is worth mentioning that the values expressed in Table 7 do not consider any free tiers offered by providers. The results of this cost analysis will be analyzed in Section IV-D. The scripts used in the experiments, as well as the raw data and tabulation files are available at:

- 1st Experiment - https://doi.org/10.5281/zenodo.7668258;
- 2nd Experiment - https://doi.org/10.5281/zenodo.7668266; and
- 3rd Experiment - https://doi.org/10.5281/zenodo.7668286.

B. ANALYSIS OF THE 1ST EXPERIMENT RESULTS

The results of the math workload in the 1st experiment test case are shown in Figure 6a. In this figure, it is possible to note that for most requests, the application performing local processing obtained better results when considering the execution time. While the average request execution time for the application without FaaS was 0.46 seconds, the application average with FaaS was 1.45 seconds, as can be seen in Table 8. This represents a difference of 215%. Thus, it is clear that for simple algorithms, the adoption of FaaS through the Node2FaaS approach, does not represent a gain in the execution time, since processing the overhead generated by the passage through the network is not offset by the FaaS execution time.

The results of the test case for overloading the CPU are shown in Figure 5b. It is possible to observe that the converted application maintained stability in the execution time, varying between 1.12 seconds and 2.61 seconds, while the other application presented degradation, starting at 0.30 seconds and ending at 2.87 seconds, as shown in Table 8. This demonstrates that even in relation to the consumption of CPU, after a certain point, an application using FaaS presents higher runtime times than the application running locally.

The performance degradation that occurs in the application locally is due to concurrence, because the more simultaneous requests arrive for treatment by the server, the greater the amount of resources that the machine needs to allocate for this treatment. These resources tend to run out, since a local instance is a limited machine, and with that the server can even stop responding, removing an application from operation. This behavior can be seen in Figure 5b at around 82 simultaneous requests, in which there is a sudden jump in the execution time in comparison with the application without FaaS. It is likely that at that point the server reached the limit of its processing capacity and started to queue requests, increasing the time for handling processes. This increase lasted until around request 93, when the queuing overhead started to interfere less in the execution time.

A similar behavior can be observed for application with FaaS at around 57 simultaneous requests, that is, well before the same occurrence in the application without FaaS. This anticipation is explained by the fact that the instance provided by the provider has less processing capacity than an instance destined for the execution of the application without FaaS and, therefore, saturated earlier. Despite this, using the services of FaaS in the Node2FaaS way, the provider guarantees automatic elasticity, the service can potentially need a much higher amount of concurrence compared to the approach without FaaS, as new entrants are automatically available to meet this increase in processing and ensure its execution.

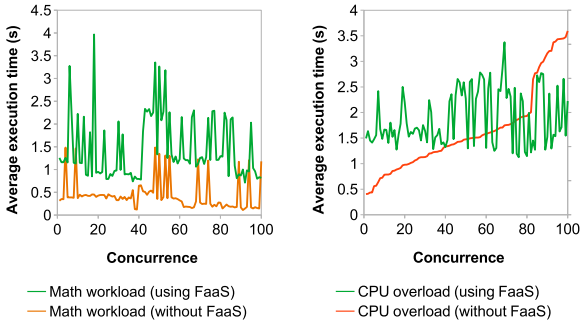
The test case with memory overload, shown in Figure 5c, obtained a result similar to the overhead test of CPU. While the variation in the execution time of the application using

TABLE 7. Instances and FaaS costs in dollar ($).

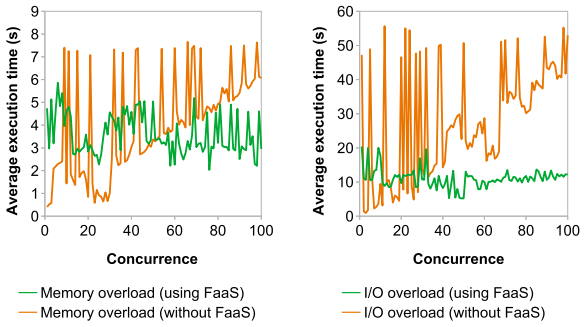
Exp.	Provider	Instance Type	Instance use cost	FaaS use cost	FaaS request cost
1st	AWS	t2.micro	0.00019	0.000125	0.00000019
2nd	Local	MacBook Pro	0.00057	-	-
2nd	AWS	MacBook Pro	-	0.000125	0.00000019
2nd	GCP	MacBook Pro	-	0.000138	0.00000039
2nd	Azure	MacBook Pro	-	0.000120	0.00000019
3rd	AWS	t3a.xlarge	0.00554	0.000125	0.00000019
3rd	GCP	e2-standard-4	0.00258	0.000138	0.00000039
3rd	Azure	standard_D4s_v3	0.00390	0.000120	0.00000019

TABLE 8. 1st experiment execution times (without the Terraform as cloud orchestrator).

Application profile	Test case	Better time (s)	Worst time (s)	Average time (s)
No FaaS	Math	0.71	3.96	0.43
With FaaS	Math	0.10	1.48	1.45
No FaaS	CPU	0.30	2.87	1.24
With FaaS	CPU	1.12	2.61	1.72
No FaaS	Memory	0.40	7.65	3.99
With FaaS	Memory	2.02	5.86	3.59
No FaaS	I/O	0.93	55.64	27.54
With FaaS	I/O	5.19	20.43	11.04



(a) Math workload results. (b) CPU workload results.



(c) Memory workload results. (d) I/O workload results.

FIGURE 6. Results of the 1st experiment (without the Terraform as cloud orchestrator).

FaaS remained stable, the curve of the application without FaaS pointed upwards. However, the crossing of the curves occurred faster compared to the test with CPU. This shows that high memory consumption degrades the execution time more significantly than the consumption of CPU, in this case.

In the test case with I/O overload, shown in Figure 5d, the same scenario of the tests involving CPU and memory is verified. In the first requests, the application without FaaS alternates between times above and below those registered by the application with FaaS. But as of the thirty-third simultaneous request, the execution time of the application without FaaS starts registering values continuously higher than the results of the application with FaaS, an demonstrates a gradual upward trend. Table 8 shows that the worst results observed with FaaS and without FaaS were 20.43 seconds and 55.64 seconds, respectively. This shows that, in the I/O overload test, there was a difference of 172% in the worst case. This can be explained by the fact that there is a greater parallelism in processing using FaaS, while local execution is penalized by queuing in high concurrence situations.

Table 8 shows the consolidated comparison of the results obtained in each type of test. Thus, it is possible to observe that the tests with simple load and CPU obtained lower average execution times on the server without FaaS. On the other hand, execution with memory overload and I/O were, on average, faster on servers using FaaS. This is explained by the fact that the provider of FaaS is delivering automatic elasticity for the consumption of resources, while on the server without FaaS the task of managing this concurrence is on the server itself, and this causes queuing, which blocks processing and delays execution.

C. ANALYSIS OF THE 2ND EXPERIMENT RESULTS

Table 9 shows the consolidation of the data obtained from the execution of the test cases in the 2nd Experiment. These data reveal information about the average execution time of the workload in each provider, as well as the accounting of unsuccessful executions and the reliability rate calculated in each workload. It was also possible to calculate the shortest and longest execution times in order to calculate the amplitude of the waiting time for the execution of the workloads.

The results showed a marked difference in the time of execution of the local tests in relation to the times registered by the applications adopting the RPC (FaaS-based) Node2FaaS approach. As can be seen in Figure 7, which shows the consolidated average times for the CPU test cases, while in local execution the average was around 283 seconds, FaaS services fared much better, reaching 9.8 seconds, in the case of AWS. Even the provider with the worst result, Azure, still improved upon the result from local processing.

Figure 7 shows an even more discrepant result between local execution times and Node2FaaS approach execution

TABLE 9. Consolidated results of the 2nd experiment (with the Terraform as cloud orchestrator).

Bound	Provider	Reliability	Average time (s)	Minimal time (s)	Maximal time (s)	Await time (s)
CPU	Local	100.00%	280.1	1.1	1321.8	1320.7
CPU	AWS	98.59%	9.8	0.000	22.6	22.6
CPU	GCP	98.50%	16.0	2.0	55.1	53.1
CPU	Azure	37.26%	58.7	6.3	120.1	113.8
Memory	Local	100.00%	388.5	0.5	2386.0	2385.5
Memory	AWS	65.78%	2.4	1.1	6.5	5.4
Memory	GCP	58.43%	6.7	1.8	26.4	24.6
Memory	Azure	30.15%	69.9	9.2	114.7	105.5
I/O	Local	100.00%	415.1	1.2	3801.2	3799.9
I/O	AWS	98.50%	2.0	0.4	4.8	4.3
I/O	GCP	98.57%	4.9	0.6	30.5	29.8
I/O	Azure	0.00%	N/A	N/A	N/A	N/A

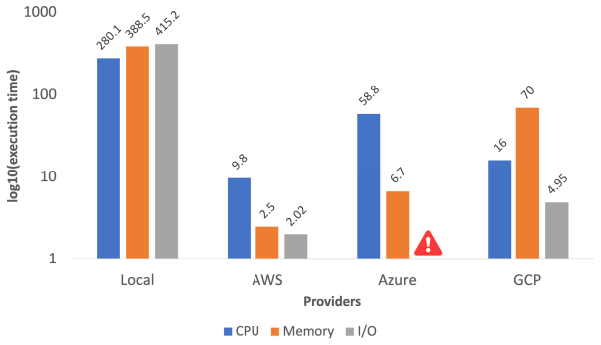


FIGURE 7. Results of the 2nd experiment (with the Terraform as cloud orchestrator).

times, in this case, for tests with high memory consumption. AWS and GCP recorded average times of 2.5 and 6.7 seconds, respectively. The Azure provider recorded a higher result, with an average time of 64.04 seconds, but still much lower than the 288 seconds recorded as the average test processing on the local machine.

The test case that aimed to stress disk activity produced an intriguing result. While the AWS and GCP providers recorded averages of 2 and 4.9 seconds of execution time and local tests resulted in an average of 415 seconds, as can be seen in Figure 7, the Azure provider was unable to end any test cases. Considering that the test cases varied between 10, 50 and 120 simultaneous requests, and that for a single request the Azure provider was able to successfully fulfil the request, it is possible to infer that unfortunately the Azure FaaS cannot satisfactorily handle situations involving concurrently recording and deleting files.

Other occurrences of failures during the test runs were observed in AWS and GCP. However, unlike Azure which simply drops the connection, they return an error message that allows some treatment of the problem, such as:

- **AWS:** {"message":"Internal server error"}
- **AWS:** {"errno" : "ENOTFOUND", "code" : "ENOTFOUND", "syscall" : "getaddrinfo", "hostname": "5xyg1589i5.execute-api.us-east-1.amazonaws.com"}
- **AWS:** {"errno" : "ETIMEDOUT", "code" : "ETIMED-OUT", "syscall" : "connect", "address" : "99.84.27.18", "port" : 443}

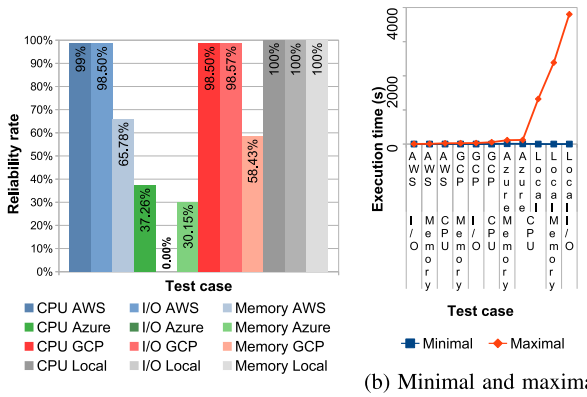
TABLE 10. Reduction in execution times in 2nd experiment (with the Terraform as cloud orchestrator).

Workload	GCP	Azure	AWS
CPU	92.12%	68.55%	94.76%
Memory	97.54%	74.08%	**99.01%**
I/O	92.48%	0.00%	96.89%

- **GCP:** Error: Server Error. The server encountered an error and could not complete your request. Please try again in 30 seconds.
- **GCP:** {"errno" : "ENOTFOUND", "code" : "ENOTFOUND", "syscall" : "getaddrinfo", "hostname" : "us-central1-node2faas-248113. cloudfunctions.net"}

Table 10 shows the reduction percentages obtained in each provider and, it is possible to observe that for the memory test, a gain of 99% was obtained using the AWS provider processing. As mentioned earlier, from the comparison between the number of executions for each workload and the number of successful executions, as shown in Table 9, it was possible to calculate the respective reliability rates. Figure 8a shows these rates graphically. In this figure it is possible to see the valley that is created when the bars refer to the Azure provider indicating a noticeably lower reliability rate than the others. However, for the memory-related workload a drop in reliability is also noted in both other providers. Only local execution maintained its reliability rate fixed at 100% for all workloads.

The results of the reliability rates of the application whose execution took place in a local environment, that is, without the adoption of the Node2FaaS approach using FaaS are excellent, the application completed 100% of the tasks sent to them. However, as can be seen in Figure 7b, the waiting time for this execution to take place can be quite high. Figure 7b shows two curves, with the minimum and maximum execution times for each workload. It shows that the curves related to FaaS providers are very close together, almost overlapping, while for the local environment there is a huge gap between them. This trade-off related to the waiting time for executing a process versus the possibility of failure is an aspect that needs to be considered when adopting FaaS.



(a) Reliability rates.

(b) Minimal and maximal execution times.

FIGURE 8. Qualitative analysis of results in the 2nd experiment (with the Terraform as cloud orchestrator).

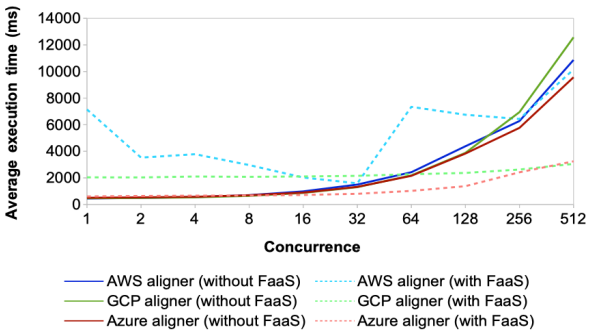


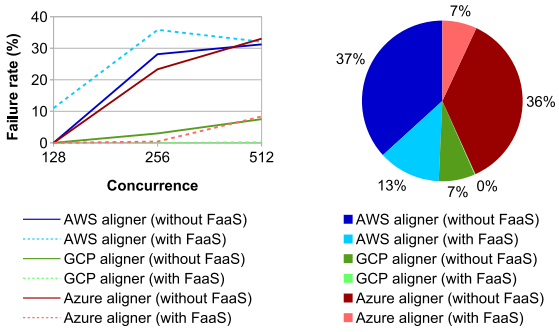
FIGURE 9. Execution time results in the 3rd experiment (with the Terraform as cloud orchestrator and real application).

D. ANALYSIS OF THE 3RD EXPERIMENT RESULTS

Figures 9 and 10a present the results obtained in the 3rd Experiment. The results obtained from applications running on AWS are represented in bluish colors, while the results from GCP are represented in green, and from Azure in red. The solid lines represent the original applications, that is, without the use of FaaS in the RPC model. The dashed lines represent the applications that were converted by Node2FaaS to use the respective providers' FaaS services.

In Figure 9 each line represents the average execution time of the alignment set of genetic sequences executed in parallel according to the level of concurrence, from 1 to 512. In Figure 10a only the levels of concurrence that presented failures are represented, that is, from 128 simultaneous alignment requests up to 512.

Analyzing Figure 9, it is possible to observe that both applications without FaaS present the same behavior in which they present low execution times in the face of low concurrence and as concurrence increases, the execution time increases proportionally. On the other hand, applications converted to FaaS by the Node2FaaS framework behave differently. Those running on GCP and Azure have higher runtimes than those recorded by applications without FaaS in the face of low concurrence, however, this runtime changes little as the level of concurrence increases. This is due to



(a) Failure rates per concurrence.

(b) Failure rate distribution.

FIGURE 10. Failure rates in the 3rd experiment (with the Terraform as cloud orchestrator and real application).

the elastic characteristic of FaaS services in which more resources are added as demand increases. The application converted to FaaS executed by AWS presented a different behavior from its equivalent, presenting execution times much higher than the other services, as well as an oscillation of the average execution times as the competition increases. This is due to the scaling strategy adopted by the provider at the time of testing.

In Figure 10a it is possible to identify the applications running on AWS as the ones with the highest failure rates, both in the application without FaaS and mainly in the application with FaaS. This result differs considerably from the results obtained in the first two experiments where the maximum levels of concurrence were 100 and 128. This demonstrates that the AWS provider presented less satisfactory results in terms of reliability from 128 simultaneous requests. The other providers also recorded a certain failure rate starting at concurrence level 128, but below the rates recorded by AWS.

Figure 9b shows the distribution of failures regardless of the level of concurrence. In this analysis, it is possible to see that AWS and Azure appear practically tied in number of failures, giving GCP, in this experiment, the highest reliability rate.

E. COST ANALYSIS

The analysis of the average execution times shows (Sections IV-B, IV-C, and IV-D) that applications using FaaS through the Node2FaaS approach present superior results as the level of concurrence rises. However, it is necessary to analyze the associated costs, since the charging models are quite different. Thus, Figures 11, 12, and 13 presents a comparative analysis of the costs calculated according to Section IV-A.

In the 1st Experiment, as can be seen in Figure 11, the total cost of the simplest use case, which is a mathematical calculator, was predominantly composed of the approach using FaaS, which accounted for 85% of the cost. This occurred due to the simplicity of the function, which, when migrated to FaaS, only received the network overhead, which was reflected in the cost. Similarly, use cases that stress CPU and memory also registered a higher cost for the FaaS approach, since the developed functions were not complex

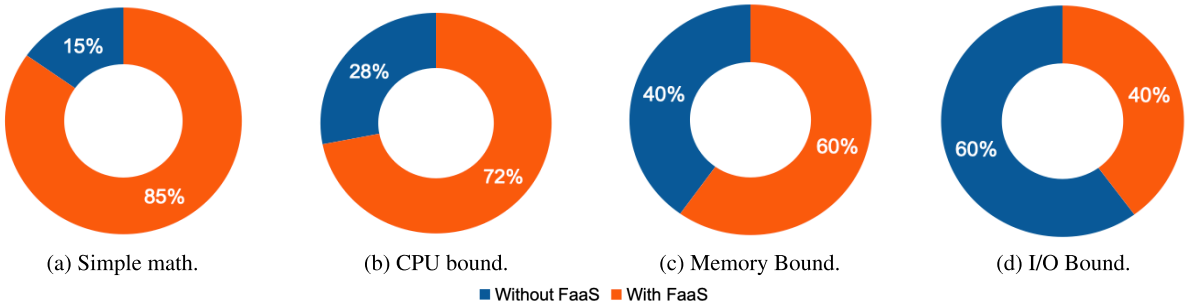


FIGURE 11. Costs distribution in the 1st experiment (without the Terraform as cloud orchestrator).

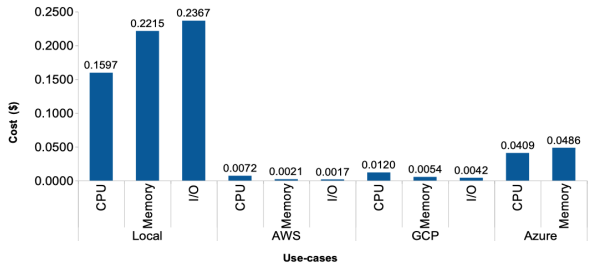


FIGURE 12. Costs in the 2nd experiment (with the Terraform as cloud orchestrator).

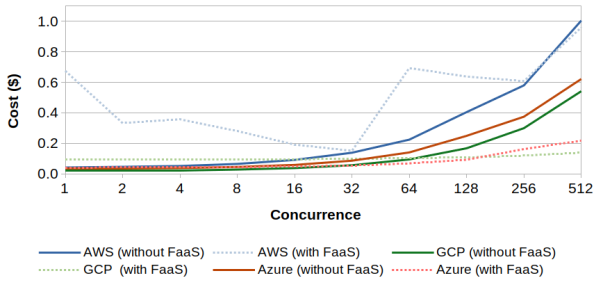


FIGURE 13. Costs in the 3rd experiment (with the Terraform as cloud orchestrator and real application).

enough to retain processing in the local environment to the point that it is worth migrating to FaaS. On the other hand, the use case that stresses I/O achieved financial success by migrating to FaaS, as only 40% of the cost of this use case referred to the approach with FaaS, while the other 60% was due to the approach without FaaS. This is explained by the characteristic of the use case, which, when creating and reading a file, triggers blocks in the processing that burden the execution time.

Figure 12 presents the costs registered in the 2nd Experiment. It is possible to notice a drastic drop in the costs of the use cases executed exclusively in the local environment and comparison with the use cases that used a FaaS approach after the conversion made by Node2FaaS. This was due to the greater degree of complexity of the functions in the 2nd experiment compared to the 1st, so that the execution time calculated locally was so much greater than those using the FaaS approach that it suppressed the additional cost of executing the functions remotely, in FaaS. In this analysis, the Azure I/O use case could not be calculated due to the lack of execution times that make up the calculation, since in this scenario the Azure provider reported error in all executions.

Figure 13 presents the costs registered in the 3rd Experiment. It is possible to observe that except for the application converted to FaaS running on AWS, the cost curves show an upward trend. This is an expected result, since the base of the calculation is the time in use of the virtual machines in which the applications are installed. Although applications converted to FaaS also have a cost of operating time on the virtual machines on which they are installed, and in addition to FaaS costs, given the decrease in the total application execution time, a reduction in the intensity of the curve growth of the applications converted to FaaS from GCP and Azure was noticed. This behavior was not observed in the application converted to FaaS running on AWS due to its high failure rate.

V. RELATED WORKS

The work [2] brings an approach for converting applications written in Python to perform deployments in multiple providers. The application built by Spillner, called Lambada, processes an application in Python and converts it to the appropriate code to be instantiated in the cloud.

Spillner and Dorodko [12] apply the same approach adopted in Lambada, but for applications developed in Java. In this work, the authors question the economic feasibility of running a Java application entirely using FaaS, and whether it could be possible to automate the application conversion process.

Zappa [3] is a command line tool that converts Flask to run on AWS Lambda. It is a framework that packages and deploys Python applications compatible with WSGI in an AWS Lambda function and in the AWS API gateway.

PyWren [4] exposes a Python continuous primitive map over AWS Lambda. Although AWS Lambda was designed to run microservices controlled by events at scale, dynamically extracting S3 code (storage service from AWS), it makes each call to AWS Lambda perform a different function. PyWren serializes a Python function, capturing all relevant

TABLE 11. Comparative of FaaS-oriented solutions. (a) Languages: P = Python; J = Java; and N = Node.js. (b) Applicability: G = General and C = Crossplane.

(a) **Languages:** P = Python; J = Java; and N = Node.js.

(b) **Applicability:** G = General and C = Crossplane.

Characteristic	Lambada [2]	Podilizer [12]	Zappa [3]	PyWren [4]	Lithops [5]	ToLambda [6]	DAF [7]	M2FaaS [8]	Node2FaaS
Languages	P	J	P	P	P	J/N	N	N	**N**
Applicability	C	C	G	G	G	C	C	G	**G**
Multicloud	✓	-	-	-	✓	-	-	✓	✓
All main providers support (AWS, GCP and Azure)	-	-	-	-	✓	-	-	-	✓
Multicloud orchestrator	-	-	-	-	✓	-	-	-	✓
FaaS adherence analysis	-	-	-	-	-	-	-	-	✓
Low complexity App evaluation	✓	✓	-	-	-	✓	✓	-	✓
CPU bound App evaluation	-	-	-	✓	-	-	-	✓	✓
Memory bound App evaluation	-	-	-	✓	-	-	-	✓	✓
I/O bound App evaluation	-	-	-	-	-	-	-	✓	✓
Real App evaluation	-	✓	-	✓	-	-	-	✓	✓

information, as well as most modules that are not present at the server's runtime.

Lithops [5] is a Python multi-cloud serverless computing framework. It allows unmodified local python code to be run at massive scale in the main serverless computing platforms. Lithops delivers the user's code into the cloud without requiring knowledge of how it is deployed and run. Moreover, its multicloud-agnostic architecture ensures portability across cloud providers, overcoming vendor lock-in. Lithops provides great value for data-intensive applications like Big Data analytics and embarrassingly parallel jobs.

ToLambda [6] is a generic automation and transformation tool. It takes a zip file with Java monolith application source code and generates Node.js AWS Lambda functions. These methods can be deployed into AWS manually or through SAM Cloud Formation templates. Decomposition and dependency search is done completely automatically. The code generated is self-sufficient and contains all the referenced methods.

Dependency-Aware FaaSifier (DAF) [7] is a code transformer that works to resolve both code and package dependencies of each faasified method of a Node.js monolith from annotations made directly in the code. Afterwards the annotation DAF automatically builds equivalent serverless functions and replaces the monolith method code with an API call.

M2FaaS [8], is a FaaSifier for monolithic Node.js applications that automatizes FaaSification of arbitrary code blocks using simple annotation constructs that do not change the semantics of the monolith. The authors claim that the work is the first FaaSifier that introduces FaaSification to extract a code block as a serverless function considering not only all package and code dependencies, but also the data dependency.

Table 11 presents a comparative of the solutions mentioned in which it is possible to see that although an adherence analysis phase is fundamental to characterize if the essence of the function fits the FaaS paradigm, none of them has a FaaS adherence analyzer, unlike what happens with the Node2FaaS framework. Furthermore, the mentioned works do not integrate a multicloud orchestrator and do not guarantee the publication of functions in the three main FaaS providers. On the other hand, the Node2FaaS framework uses Terraform and defined blueprints to integrate with AWS, GCP and Azure. The set of applications that the Node2FaaS framework was evaluated on ranges from simple functions to applications for real use, while the other works have a lower set of evaluations.

VI. DISCUSSION

The bibliographic survey carried out on the use of FaaS with the purpose of making the processing of applications executable under the RPC paradigm showed that this is an area that is still little explored. There are some works that propose an approach similar to this work, however for other programming languages, like Python and Java, such as Lambada [2] and Podilizer [12]. There are platforms, such as Serverless [48] and Vercel (formerly known as Zeit) [49], which promote the management of code *deployments* in FaaS services, in this case acting as an abstraction layer above the providers. Claudia.js [50], Zappa [3] and PyWren [4] are part of a class of tools that handle deployments in FaaS services without the commitment of transforming code already written in something executable over FaaS, but it only helps in the development of applications whose approach is natively oriented to FaaS.

In addition, the work [2], for example, was designed to use Lambada to convert applications with a unique function. In production environment applications, it is common to compose multiple functions for the execution of an application. The Node2FaaS framework, on the other hand, allows a

better use in real scenarios, not being focused experimental environments.

The analysis of the results allows us to infer that, in general, for few requests, the response time of the application running without FaaS tends to be better. For applications that require a lot of CPU resources, if there is little concurrence, applications without FaaS show better results, however, from a threshold, which in experiments was around 80 simultaneous requests, both approaches perform similarly, as can be seen in Figure 5b.

For applications with high memory consumption and I/O, the benefits of using FaaS are evident, even from low levels of concurrence. This is because in monolithic applications the consumption of this type of resource causes many blockages in processing. On the other hand, in applications oriented to FaaS, there is a high parallelism in the consumption of these resources, reducing the blocking effect. Thus, applications with this type of characteristic are strong candidates for adopting the FaaS model, using the approach proposed by Node2FaaS.

The results of the tests presented (without and with the orchestrator) clearly demonstrate that the approach proposed by the Node2faaS framework provides a significant reduction in the execution time of resource-intensive applications such as CPU, memory and disk activity (I/O). The necessary adjustments to make the internal functions of a traditional Node.js application executable in FaaS services can be a tedious process and is highly error-prone if performed manually by a developer. On the other hand, the use of the Node2FaaS framework delivers the uniformity and consistency that an application needs for this type of operation and also provides great agility for the developer due to the automations that the framework performs. In addition, the use of an orchestrator supported by the community gives this process greater flexibility, since there is no need to transfer the processing from one provider to another, just activate Node2FaaS again pointing to the provider and then this movement will be left to the framework and its coupled orchestrator. Everything is very simple and fast.

With the adoption of Terraform, it was possible to execute a broader experiment using the three main providers in the market today: AWS, GCP and Azure. The results showed gains of up to 92% for applications with intensive use of CPU, 96% for applications with intensive use of reading and writing on disk, and up to 99% for applications with intensive use of memory.

Considering the characteristics of the framework: ease of operation, automatic conversion, function analysis, optimized execution, flexibility, multiple providers, effectively usable, and open source tool mentioned in Section III, the ease of use is observable since with only one command in the CLI of the framework it is possible, automatically and selectively (thanks to the adhesion analyzer and the possibility of defining a specific behavior for certain functions), to generate an application whose architecture is RPC-oriented and can be used to solve real-world problems in an optimized way using FaaS in multiple providers. In addition, once the code of the tool is published and the access being open, it is possible to perform changes in its implementation.

VII. CONCLUSION

The Node2FaaS framework proved efficient in the task of converting monolithic Node.js applications to work with FaaS. The experiments showed that after the conversion made by Node2FaaS there were significant gains in the execution time of applications with intensive use of CPU, memory and disk usage. The Node2FaaS, Terraform and FaaS triad consists of a platform for amplifying the performance of applications that demand intensive use of computational resources.

The Node2FaaS framework delivers to the applications that are submitted to it the advantages of being verified as to the adherence of their functions to the FaaS paradigm. In addition, the entire automated deployment process introduces uniformity to the process making it less prone to human error.

However, even though the framework has a layer to analyze and decide on sending functions to FaaS services, this layer can be improved to make a deeper semantic analysis in the source code so that the decision is made using metrics related to aspects of application behavior. In future works would be possible to improve this, including features such as: a list of not adherent functions check, implementations according to node.js version, and an event machine learning process to make the decision.

In addition, although the framework is prepared to work with leading providers, its cloud orchestrator, Terraform, supports other providers. Therefore, in future works this support can be extended in order to offer a greater range of options to users of the framework.

REFERENCES

[1] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2017, pp. 405–410.
[2] J. Spillner, "Transformation of Python applications into function-as-a-service deployments," 2017, *arXiv:1705.08169*.
[3] R. Jones, "Zappa—Serverless Python," Tech. Rep., 2020. [Online]. Available: https://github.com/zappa/Zappa
[4] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. Symp. Cloud Comput.*, Sep. 2017, pp. 445–451.
[5] *What is Lithops?* T. L. Team, 2023. [Online]. Available: https://lithops-cloud.github.io/docs/
[6] A. Kaplunovich, "ToLambda-automatic path to serverless architectures," in *Proc. IEEE/ACM 3rd Int. Workshop Refactoring (IWoR)*, May 2019, pp. 1–8.
[7] S. Ristov, S. Pedratscher, J. Wallnoefer, and T. Fahringer, "DAF: Dependency-aware FaaSifier for Node.Js monolithic applications," *IEEE Softw.*, vol. 38, no. 1, pp. 48–53, Jan. 2021.
[8] S. Pedratscher, S. Ristov, and T. Fahringer, "M2FaaS: Transparent and fault tolerant FaaSification of Node.Js monolith code blocks," *Future Gener. Comput. Syst.*, vol. 135, pp. 57–71, Oct. 2022.
[9] L. R. de Carvalho and A. F. de Araújo, "Framework Node2FaaS: Automatic NodeJS application converter for function as a service," in *Proc. 9th Int. Conf. Cloud Comput. Services Sci.*, 2019, pp. 271–278.
[10] L. R. de Carvalho and A. F. de Araujo, "Remote procedure call approach using the Node2FaaS framework with terraform for function as a service," in *Proc. 10th Int. Conf. Cloud Comput. Services Sci.*, 2020, pp. 312–319.

[11] Y. Brikman, *Terraform: Up and Running: Writing Infrastructure as Code*. Sebastopol, CA, USA: O'Reilly Media, 2017.
[12] J. Spillner and S. Dorodko, "Java code analysis and transformation into AWS lambda functions," 2017, *arXiv:1702.05510*.
[13] A. E. Youssef, "Exploring cloud computing services and applications," *J. Emerg. Trends Comput. Inf. Sci.*, vol. 3, no. 6, pp. 838–847, 2012.
[14] P. Mell and T. Grance, "The NIST definition of cloud computing," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. 2011, Sep. 2011.
[15] K. Kritikos and D. Plexousakis, "Multi-cloud application design through cloud service composition," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 686–693.
[16] A. Monteiro, J. S. Pinto, C. Teixeira, and T. Batista, "Sky computing," in *Proc. 6th Iberian Conf. Inf. Syst. Technol. (CISTI)*, Jun. 2011, pp. 1–4.
[17] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: Taxonomy and survey," *Softw., Pract. Exper.*, vol. 44, no. 3, pp. 369–390, Mar. 2014.
[18] Y. Elkhatib, "Defining cross-cloud systems," 2016, *arXiv:1602.02698*.
[19] S. Ristov, S. Pedratscher, and T. Fahringer, "XAFCL: Run scalable function choreographies across multiple FaaS systems," in *Proc. IEEE World Congr. Services*, Jul. 2022, p. 32.
[20] J. Kovács and P. Kacsuk, "Occopus: A multi-cloud orchestrator to deploy and manage complex scientific infrastructures," *J. Grid Comput.*, vol. 16, no. 1, pp. 19–37, Mar. 2018.
[21] L. M. Pham, A. Tchana, D. Donsez, N. de Palma, V. Zurczak, and P.-Y. Gibello, "Roboconf: A hybrid cloud orchestrator to deploy complex applications," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 41–48.
[22] J. Carrasco, F. Durán, and E. Pimentel, "Trans-cloud: CAMP/TOSCA-based bidimensional cross-cloud," *Comput. Standards Interface*, vol. 58, pp. 167–179, May 2018.
[23] *Topology and Orchestration Specification for Cloud Applications Version 1.0*, Oasis, Burlington, MA, USA, 2019. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca
[24] X. Wang, Z. Liu, Y. Qi, and J. Li, "LiveCloud: A lucid orchestrator for cloud datacenters," in *Proc. 4th IEEE Int. Conf. Cloud Comput. Technol. Sci.*, Dec. 2012, pp. 341–348.
[25] D.-H. Le, H.-L. Truong, G. Copil, S. Nastic, and S. Dustdar, "SALSA: A framework for dynamic configuration of cloud services," in *Proc. IEEE 6th Int. Conf. Cloud Comput. Technol. Sci.*, Dec. 2014, pp. 146–153.
[26] M. Caballer, D. Segrelles, G. Moltó, and I. Blanquer, "A platform to deploy customized scientific virtual infrastructures on the cloud," *Concurrency Comput., Pract. Exper.*, vol. 27, no. 16, pp. 4318–4329, Nov. 2015.
[27] M. E. Walter, M. Holanda, G. Vergara, M. Rosa, A. Araújo, and B. Moura, "BioNimbuZ: A federated cloud platform for bioinformatics applications," *Int. J. Data Mining Bioinf.*, vol. 18, no. 2, p. 144, 2017.
[28] N. Loulloudes, "The celar project," EU CELAR Project, Tech. Rep., 2019. [Online]. Available: https://github.com/CELAR/c-Eclipse
[29] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA—A runtime for TOSCA-based cloud applications," in *Service-Oriented Computing*. Cham, Switzerland: Springer, 2013, pp. 692–695.
[30] *Open Tosca*, Open Tosca, Univ. Stuttgart, Stuttgart, Germany, 2019. [Online]. Available: https://www.opentosca.org/
[31] *Getting Started*, Cloudify, New York, NY, USA, 2019. [Online]. Available: https://cloudify.co
[32] *Hot Guide*, OpenStack, Austin TX, USA, 2019. [Online]. Available: https://www.openstack.org
[33] *Apache Aria Tosca Orchestration Engine*, Apache, 2019. [Online]. Available: https://incubator.apache.org/projects/ariatosca.html
[34] *AWS Cloudformation*, AWS, Seattle, WA, USA, 2019. [Online]. Available: https://aws.amazon.com/pt/cloudformation
[35] T. Borovsak, "xOpera orchestrator," XLAB, Liubliana, Eslovénia, Tech. Rep., 2019. [Online]. Available: https://xlab-si.github.io/xopera-docs/
[36] *Introducing VMWare Cloud Assembly, VMWare Code Stream and VMWare Service Broker*, VMWare, Palo Alto, CA, USA, 2019. [Online]. Available: https://docs.vmware.com/en/vRealize-Automation/8.11/Getting-Started-Cloud-Assembly/GUID-D50B687A-1236-4E2E-8C79-995D1762EB85.html
[37] L. R. de Carvalho and A. P. F. de Araujo, "Performance comparison of terraform and cloudify as multicloud orchestrators," in *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID)*, May 2020, pp. 380–389.
[38] C. Spoiala, "Pros and cons of serverless computing," ASSIST Software SRL, Suceava, Romania, Tech. Rep., 2017. [Online]. Available: https://assist-software.net/blog/pros-and-cons-serverless-computing-faas-comparison-aws-lambda-vs-azure-functions-vs-google
[39] O. Zimmermann, "Microservices tenets: Agile approach to service development and deployment," *Comput. Sci.-Res. Develop.*, vol. 32, nos. 3–4, pp. 301–310, Jul. 2017.
[40] D. Flanagan, *JavaScript: O Guia Definitivo*. Porto Alegre, Brazil: Bookman Editora, 2011.
[41] B. Frankston, "The Javascript ecosystem," *IEEE Consum. Electron. Mag.*, vol. 9, no. 6, pp. 84–89, Nov. 2020.
[42] H. Shah and T. Soomro, "Node.js challenges in implementation," *Global J. Comput. Sci. Technol., Netw., Web Secur.*, vol. 17, pp. 1–22, Jun. 2017.
[43] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," *Commun. ACM*, vol. 64, no. 5, pp. 76–84, Apr. 2021.
[44] S. Ristov, C. Hollaus, and M. Hautz, "Colder than the warm start and warmer than the cold start! Experience the spawn start in FaaS providers," in *Proc. Workshop Adv. tools, Program. Lang., PLatforms Implementing Evaluating Algorithms Distrib. Syst.*, Jul. 2022, pp. 35–39.
[45] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing," in *Proc. Int. Conf. Inf. Soc.*, Nov. 2014, pp. 92–97.
[46] *Magic Quadrant for Cloud Infrastructure as a Service, Worldwide*, GARTNER, EUA, Stamford, Connecticut, Jul. 2019. [Online]. Available: https://www.gartner.com/en/documents/3947472
[47] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *J. Mol. Biol.*, vol. 147, no. 1, pp. 195–197, Mar. 1981.
[48] *Serverless: Build Apps With Radically Less Overhead and Cost*, Serverless, San Francisco, CA, USA, 2019. [Online]. Available: https://serverless.com
[49] *Develop. Preview. Ship.*, Vercel, São Francisco, Califórnia, USA, 2020. [Online]. Available: https://vercel.com
[50] *Claudia.js: Serverless Javascript, the Easy Way*, Claudia.js, 2019. [Online]. Available: https://claudiajs.com

LEONARDO REBOUÇAS DE CARVALHO received the M.S. degree in computer science from the University of Brasília, Brazil, in 2020, where he is currently pursuing the Ph.D. degree. He is also a Cloud Automation Manager with a Brazilian government agency. His research interests include Hiper performance computing and cloud computing. He also has experience in computing science with emphasis on software engineering, artificial intelligence, business intelligence, project management, and automatic configuration management. He conducts scientific research and applied innovation concepts in these areas.

ALETÉIA PATRICIA FAVACHO DE ARAÚJO received the master's degree in computer science and computational mathematics from the University of São Paulo, Brazil, and the Ph.D. degree in computer science from the Pontifical Catholic University of Rio de Janeiro, Brazil. She is currently an Associate Professor with the Department of Computer Science, University of Brasília, Brazil. She has experience in computer science with emphasis on parallel processing, distributed systems, working mainly on computational cloud, fog, parallel and distributed algorithms, and serverless computing. She has published several research papers in international journals and conferences.