Framework Node2FaaS: Automatic NodeJS Application Converter for Function as a Service

Leonardo Rebouças de Carvalho and Aletéia Patrícia Favacho de Araújo
Computing Science Department, University of Brasília, Campus Darcy Ribeiro, Brasília-DF, Brazil

Keywords: Cloud Computing, FaaS, Function as a Service, NodeJS, Automatic Converter.

Abstract: Cloud computing emerged in the area of computer science as a means to achieve significant cost and time savings when starting projects. Among the various cloud models available, this work highlights Function as a Service - FaaS, and proposes the Node2FaaS framework for automatic conversion of applications written in NodeJS to work in a transparent way with the FaaS model. The experiments demonstrated significant gains of up to 170% at runtime for applications with high file I/O requirements. Applications with high CPU and RAM consumption also have benefits in adopting FaaS after conversion, but only when a threshold of competing processes is reached.

1 INTRODUCTION

With the goal of reducing cost and time expends in new projects, cloud computing has gained significant ground in computer science. Before this proposal became reality, considerable investment was required in the implementation of datacenters to support the processing and storage necessary for projects that demand intense computational resources (Yoo, 2011).

The concept of cloud computing is now a well-established reality. Many providers offer a range of services over the Internet, without major infrastructure investments. Thus the user of a cloud environment is able to have, in moments, access to a volume of computational resources that was restricted, until recently, to large organizations (Malathi, 2011).

The important economic differential of cloud computing lies in the fact that payment occurs on demand, that is, the customer pays only for the effective use of resources, without the need to keep the investment allocated for long periods, as in the past with investments in Datacenters. Now if an organization wants to conduct a short survey that needs clusters with hundreds of machines, simply hire a cloud service for the time needed for processing, and pay only a fraction of what the total infrastructure investment would be.

In addition to the economic benefit, this operating model allows rapid responses to sudden changes in the behavior of an application. At specific times of the year, such as Mother's Day and Christmas, it is common for large retailer websites to experience usage overload, causing slow response times and failure. However, responding quickly to these issues can be the difference between making a sale or losing a customer to a competing site. Cloud computing aids in the prevention and resolution of this type of problem through elasticity, i.e., increasing and decreasing computational capacity according to some pre-set parameter, such as CPU usage.

There are several service delivery models from infrastructure (IaaS) and platforms (PaaS) to complete software (SaaS) (Mell and Grance, 2011). However, cloud providers have constantly matured their services, including promoting integration among them. As service delivery grows, the boundaries envisaged by NIST in 2011 are being redefined. Many providers have renamed their services, especially SaaS, since the term software is something very broad. Thus, models such as database as a service (DBaaS), machine learning as a service (MLaaS), monitoring as a service (MaaS), and others have emerged. In this context, we have used the term XaaS, which expresses "everything as a service" and represents several models implemented by providers (Duan et al., 2015).

In this scenario, applications developed using the monolithic architecture have limitations that hinder scalability gains. One way to mitigate this difficulty is to use cloud services like FaaS (Chapin and Roberts, 2017) to process system segments. FaaS is a service model in which the client loads a piece of code written in a specific language, and the provider provides

Rebouças de Carvalho, L. and Favacho de Araújo, A.
Framework Node2FaaS: Automatic NodeJS Application Converter for Function as a Service.
DOI: 10.5220/0007677902710278
In *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*, pages 271-278
ISBN: 978-989-758-365-0

the processing of that code from a service call, usually via a REST API (Amazon, 2018). This model hides all the complexity involved in the infrastructure.

However, making use of the FaaS model is not trivial because it requires the user to know the form of consumption that each provider offers. In addition, it is necessary for the developer to build the applications with the use of this model in mind, or to invest considerable time adjusting applications already developed to work with FaaS.

In view of the above, this work presents the Node2FaaS framework, whose objective is to automatically and transparently convert applications written in NodeJS to work with the FaaS service model using some of these services offered by public providers. In this article the framework will convert to Amazon's AWS Lambda (Amazon, 2018). The use of the proposed framework abstracts the complexity required for code publication, for example, in Lambda and simplifies the service consumption process. This way, the developer can focus his work on the application's features and not on the provider's details.

2 FUNCTION AS A SERVICE

Programming models have evolved over time, aiming at optimizing the software, as well as its maintenance, evolution capacity and alignment to the business (Basili et al., 2010). Initially, software was developed in a monolithic way, that is, the entire application was contained in a single software block. This model caused high coupling between the components of the application, reducing maintainability. Nowadays, it is common for system architects to design their applications in a modular way, separating the different complexity blocks and grouping sets of related functionalities (Larrucea et al., 2018).

The microservice architecture has gained evidence for providing good scalability as well as improving the software maintenance process (Zimmermann, 2017). In this context, cloud providers have modeled a type of service appropriate to this approach, using FaaS.

The term microservices has been used since 2014 in agile development communities (Zimmermann, 2017). The standards and principles that permeate the concept of microservices include (Zimmermann, 2017):

- Using RestFul API;
- Business-oriented and native cloud-based development;
- Application of multiple paradigms of development, like functional and imperative;
- Applications running on container light services, such as Docker;
- Decentralized continuous delivery;
- Use of DevOps culture (development integrated with the operation).

Therefore, for the construction of software, through the principles of microservices, it is necessary to segment the development of the functionalities of the application (Lewis J, 2018). Figure 1 shows the comparison of the microservice approach with the monolithic approach.

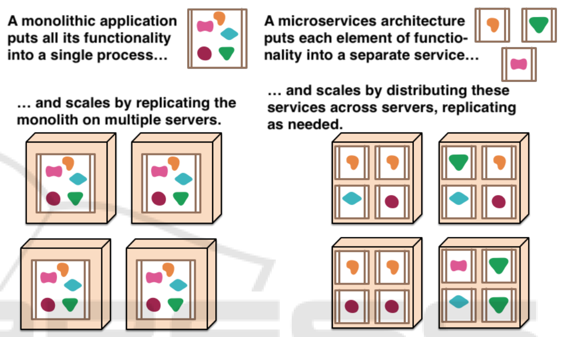
A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers.
... and scales by distributing these services across servers, replicating as needed.

Figure 1: Microservices composition (Lewis J, 2018).

While in the monolithic approach the whole application is placed within a single process, in the microservice approach, each functionality is placed in a different service. Thus, in the monolithic approach the entire application needs to be scaled up, in the case of microservices, only those services with higher demand will be scaled (providing greater rationality in the consumption of resources) (Lewis J, 2018).

Considering that normally the use of the modules is not uniform, that is, each module has a different workload, it is possible that the monolithic model is wasting resources. Although some modules are not always used, they need to be instantiated to enable the most overloaded modules to be scaled up. On the other hand, a different effect occurs in the microservice oriented model, in which only the most used modules will be effectively scaled up(Lewis J, 2018).

In order to meet the demand for infrastructure for applications based on microservices, some cloud providers have come to offer the Function as a Service model. In it the client contracts the execution of a predefined function, loads the code that wants to execute, and receives an access address for the service. Applications using this type of cloud service have been called serverless applications, since the application does not have a specific server and its operation is based on requests made to the provider's API (Savage, 2018).

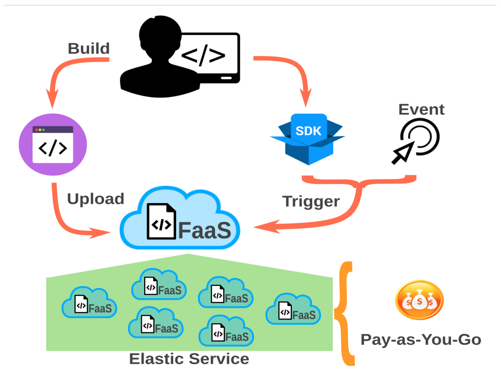

Figure 2: FaaS Workflow.

Figure 2 shows the workflow of Functions as a Service. This workflow demonstrates how the interaction between the developer and the FaaS service occurs, whether it is effectively loading code, querying the Provider Development Kit (SDK) API, or configuring triggers for invoking the service. In addition, Figure 2 shows the elasticity of the service and its pay-as-you-go model.

2.1 Advantages and Disadvantages

The FaaS services offer as main benefits of its adoption (Amazon, 2018):

- Suppressing the need for server management;
- Automatic continuous elasticity;
- Payment on demand.

Other benefits include increased project flexibility and reduced risk (Paula, 2018). Among these benefits, elasticity stands out, since, in critical applications, high availability is a fundamental characteristic. In addition, using FaaS, management focuses on the application and not on the infrastructure (Microsoft, 2018), releasing resources to enhance business logic.

On the other hand, the drawbacks of adopting FaaS-based models include the possible need for adaptations in the application because of the inability to customize features of the operating system (Billock, 2017). In addition, a delay in the response of the first execution of the function may occur after some unused time. This is due to the provider's strategy of leaving the instance that attends the linked function only powered up only for few minutes after the last call (Billock, 2017). Another potential problem is the limits set by the providers for the amount of CPU, memory, and runtime (Spoiala, 2017). The latency caused by the need to pass through the network to perform the processing is another drawback of this model (Paula, 2018).

In addition, one difficulty faced by developers in adopting FaaS is the implementation. Properly understanding the operation of the service can mean the difference between succeeding in adoption and abandoning the approach after some small failures. However, to reach this threshold, there may be enough of a learning curve so as to discourage the adoption of such a model. For this reason, this paper proposes a way to mitigate this difficulty by reducing the gap between the developer and the best results through the use of automated FaaS services, that is, automatically and transparently converting monolithic applications to Function as a Service ones.

2.2 Public Providers of FaaS

Currently, major public cloud providers have FaaS services in their catalogs. The pioneer, Amazon, offers AWS Lambda (Amazon, 2018). This service can process Java, NodeJS, C# and Python, and provides a free monthly processing package before charging for the service. The provider has several partnerships for deployment, monitoring, code management and security.

Google offers the Cloud Functions service (Google, 2018). This service can process functions written in Python and NodeJS and, like Amazon's Lambda, provides an initial quota of requisitions and starts charging only when that quota is exceeded.

Microsoft, through its Azure cloud service, provides the Azure Functions (Microsoft, 2018). This service natively allows you to load functions in C#, including using Microsoft's own tools such as Visual Studio (Microsoft, 2018). The provider specifies through its portal that the functions are only available in a Windows environment, although this is transparent to the user. Despite this, there is a forecast of availability for environments using Linux.

3 NodeJS

As computing evolves, software architectures need to adjust to the needs of the applications in use at the moment.

There was a time when gigantic machines took care of all the processing, and the software ran mostly on the big mainframes. With the popularization of personal computers, the computational power available in these machines became better used and then the software processing was gradually executed in the ”fat clients”.

The client-side programming language that was most prominent in this scenario was JavaScript. Initially used for small validations and simple interactions, today this language has become a standard and several frameworks have made use of it to increase the interactivity of the applications on the Web. The Institute of Electrical and Electronic Engineers (IEEE) published a ranking that lists the most used programming languages in 2018, and in that ranking JavaScript occupies the eighth position (IEEE, 2018).

Due to the success of JavaScript, as early as 2009 Rayan Dahl presented the NodeJS to the World (Mithun Satheesh, 2015). Not limited to being only a JavaScript framework, and without the pretense of being a new programming language, NodeJs is defined as a JavaScript code execution environment (Mithun Satheesh, 2015). The main challenge that NodeJS intends to tackle is the high scalability that Web applications demand today. To do this, it supports the asynchronous execution of processes, preventing other processes from being blocked waiting for the response of a call (Mithun Satheesh, 2015).

NodeJs uses JavaScript for processing on the server side a language already consolidated in client-side processing. With this, it adds a range of web developers to its set of potential users without the need to overcome a new learning curve, as it happens in the normal process of learning a new programming language. In addition, it has a robust support community, which offers a large number of libraries for reuse through its own package manager, NPM (Node Package Manager) (NPM, 2018).

4 FRAMEWORK Node2FaaS

Given the popularization of NodeJS, several systems have been built using this technology, and this is why it was chosen to be used in the design of the proposed framework. In order to use the available FaaS services, the developer must know the details of the APIs of each provider, as well as be able to segment the functions of the applications and convert them into appropriate calls to the service structure. This process can become cumbersome, and discourage developers from adopting the FaaS-based approach. Many professionals can give up the benefits that a cloud-based architecture can offer, such as high availability, resiliency, and cost savings, among others.

In view of the above, this work proposes the Node2FaaS framework. A tool for automatic conversion of monolithic applications, written in NodeJS, to FaaS oriented applications. The tool reads the original code of the target application and converts it to an application whose functions are performed on a FaaS service. The internal code of the functions is converted to *deployments* executed in functions created automatically in the provider. The service request is put in place of the function call. The source code for this work, as well as sample applications, can be found at https://github.com/leonardoreboucas/node2faas. In addition, the Node2FaaS framework can be installed in the local environment using NPM by executing the command: *npm install node2faas*.

4.1 Execution Flow

The flow of Node2FaaS is started from the execution of the node2faas application. If the framework has been installed via npm, this application will be registered in the machine path and can be executed directly from the command: *node2faas [application to be converted path]*. The run stream scans .js files of the original application running the conversion process. At the end of the flow the resulting application will be available in the output directory.

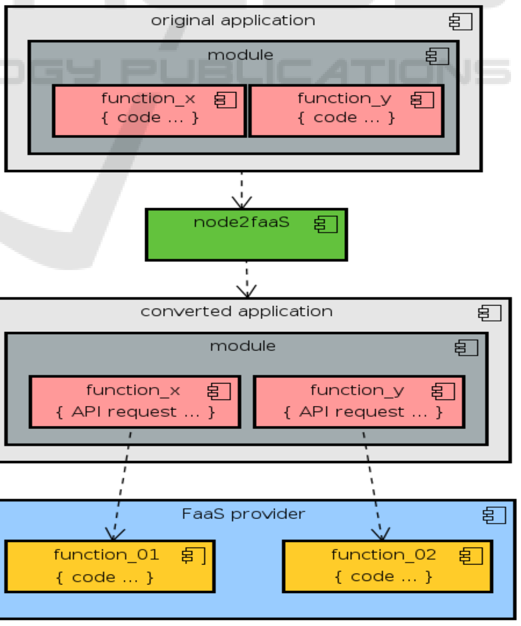

Figure 3: Node2FaaS workflow.

Figure 3 shows the execution flow of Node2FaaS. One can observe that there may be functions inside each module of an application. Within these functions there is code that performs some operation. Once submitted to Node2FaaS, this code will be published to the cloud provider and instead, in the converted application, a request to the cloud provider REST API will appear. The original code of the functions is transferred to the cloud service and then consumed through HTTP requests. The converted application maintains the same signature of the original functions, allowing its use to remain transparent to the requesters of the functions. This avoids the need to make adjustments to the application. For each request of a function, the provider will be responsible for providing an instance for the service.

The design of Node2FaaS does not define which provider should be used, but this work experiments used the Amazon FaaS service, AWS Lambda. After all, Amazon was ranked in 2018 as the leader of IaaS cloud solution in the established Gartner magic quadrant (Gartner, 2018).

4.2 Conversion Process

Firstly, it is essential to have an active account in the provider that will perform the function and the framework searches for the access credentials for the cloud. If it can‘t find the credential file, the application prompts the user to enter their username and token to access the cloud services. After this, the system creates the credential file and no longer prompts in future executions.

Once the credential has been obtained, and an application for conversion is offered, it is submitted to a conversion process that will analyze the application code searching for definitions of functions to perform the conversion, as shown in Figure 4. During the process, if a file is included then the target file is also searched for functions that are candidates for conversion, and this process runs recursively until no new included files are found.

When the application encounters a function, it accesses the cloud and creates a new FaaS function. After receiving confirmation of the function creation, the application obtains the service access URL and creates the request within the original function definition. In this way, the function call remains unchanged and its operation on the cloud platform is done in a totally transparent way.

In the end, Node2FaaS will have generated all the files that should make up the original application, but with the original function code replaced by HTTP calls to the cloud provider FaaS service.

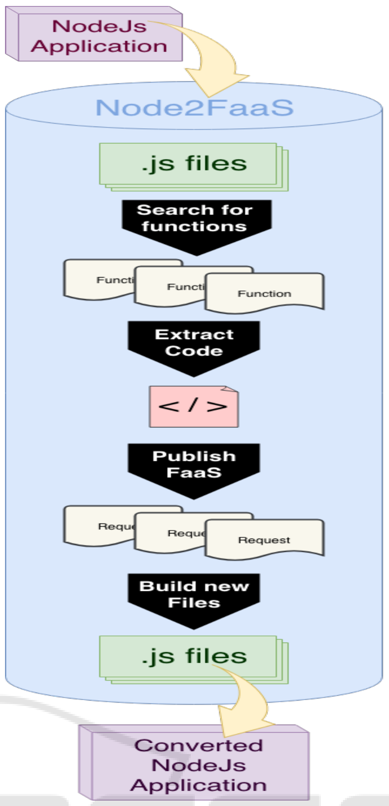

Figure 4: Node2FaaS process.

5 RELATED WORKS

The work (Spillner, 2017) takes a python-written application conversion approach to *deployments* python in the Lambda AWS service. The application built by Spillner, called Lambada, processes a python application and converts it into the appropriate code to be instantiated in the cloud. If the user has the AWS client installed and properly configured on the machine, Lambada performs deploy automatically, but the entire client configuration process is up to the user. This limits the use of this converter to users who are able to properly configure the provider's client in their local environments.

In addition, said article is limited to using Lambada for converting applications with a single function. In productive environment applications it is common to have multiple functions for the execution of an application. The Node2FaaS, proposed in this article, on the other hand, allows a better use in real applications, not being limited to experimental environments, such as Lambada.

Spillner and Dorodko (Spillner and Dorodko, 2017) apply the same approach adopted in Lambada, but for applications developed in Java. In this paper the authors question the economic feasibility of running a Java application entirely using FaaS, and whether there is the possibility of automating the application conversion process. They implemented a tool called Podilizer and performed experiments using it. The results were classified as promising, but only for academic purposes, and did not present effective application capacity, since the authors found difficulties in using the applications resulting from the conversion. On the other hand, Node2FaaS is not intended simply for academic experimentation, but for effective use by NodeJS developers.

6 METHODOLOGY

In order to exploit the potential of a NodeJS application converted to FaaS, using the tool proposed by this work, four test cases were constructed in NodeJS. The first performs a simple addition operation (test 1), the second exploits the server's processing power, consuming CPU resources throw a sequence of nested loops (test 2). The third one consumes significant memory during its execution creating very large arrays (test 3) and the fourth makes successive creations of files in the machine, exploring I/O resources (test 4). Each test case was developed within a different function and all are part of the same application.

The application containing the test cases (*Not Converted*) was hosted on a t2.micro instance of the AWS Elastic Compute Cloud service, EC2. This type of machine has 1 gigabyte of RAM, 1 CPU and uses the RedHat 7.6 operating system. The application was submitted to Node2FaaS for conversion and then the converted application (*Converted*) was hosted in a second t2.micro instance of AWS (with the same configuration). Thus, two applications were left running the test cases. The first performing the processing directly on the EC2 instance, while the second uses the AWS Lambda for processing, according to the architecture shown in Figure 5.

For the experiment a shell script was developed in order to execute simultaneous requests for each test case and collect the results, especially the execution time. Each test case was run simultaneously on both servers. Once the test cases were executed, the data regarding the execution time of the calls were tabulated and graphs were constructed to allow better analysis of the behavior of the applications on each approach.

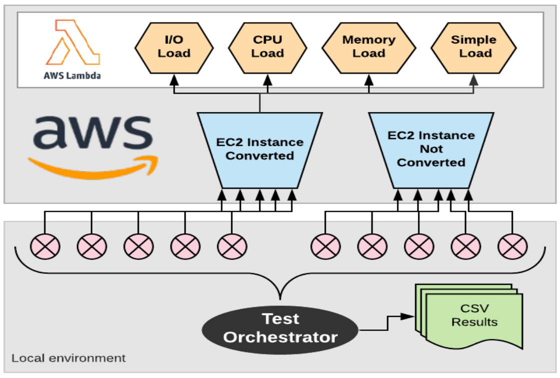

Figure 5: Experiment architecture.

7 RESULTS

The analysis of the graphs allows to infer that as shown in Figure 6 in the first test case (*Simple load*), for most requests, the application running local processing performed better when the execution time was considered. While the average execution time of requests for the *Not Converted* application was 0.46 seconds, the mean of the *Converted* application was 1.45 seconds. That's a difference of 215%. Thus, it is clear that for simple applications the adoption of FaaS does not represent improvement to the execution time.

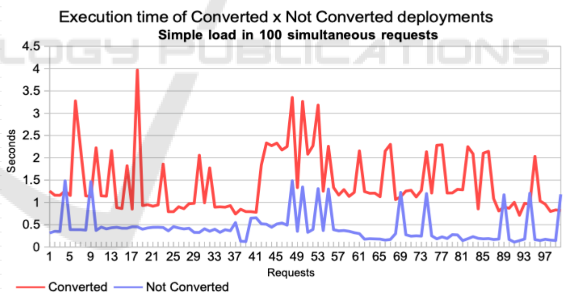

Figure 6: Execution of the simple load test.

The second test case, CPU load, has its output shown in Figure 7. It is possible to observe that the converted application maintained stability in the execution time, varying between 1.12 seconds and 3.37 seconds, while the other application presented continuous degradation, beginning in 0.30 seconds and ending at 2.69 seconds, as presented in Table 1. This demonstrates that for CPU consumption, from a certain point, an application using FaaS presents running times lower than the same application running locally.

The memory-loaded test case, shown in Figure 8, obtained a similar result to the CPU load test. While the application time variation using FaaS remained stable, the application curve without FaaS pointed up-

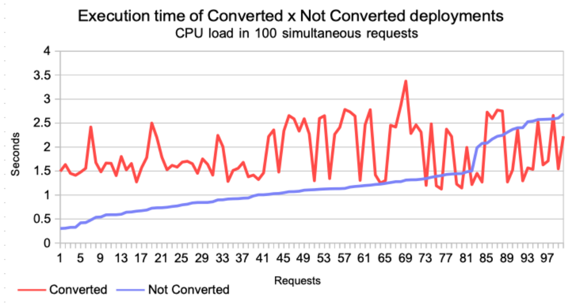

Figure 7: Execution of the CPU load test.

wards. However, the crossing of the curves occurred faster when compared to the CPU test. This shows that the high memory consumption degrades more significantly the execution time than the CPU consumption, in this case.

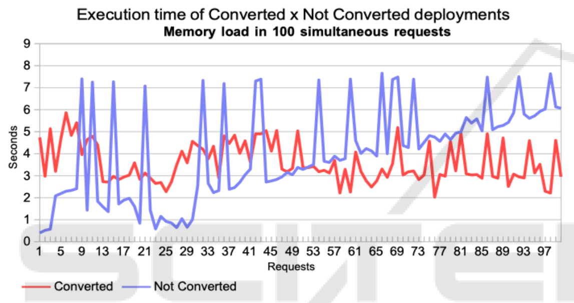

Figure 8: Execution of the memory load test.

In the last test case, shown in Figure 9, the same scenario of tests involving CPU and memory is verified. Even though the execution time of the application without FaaS is initially close to the results obtained by the converted application, its degradation is significantly accentuated. In Table 1 it is possible to verify 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 test, there was a difference of 172% in the worst case.

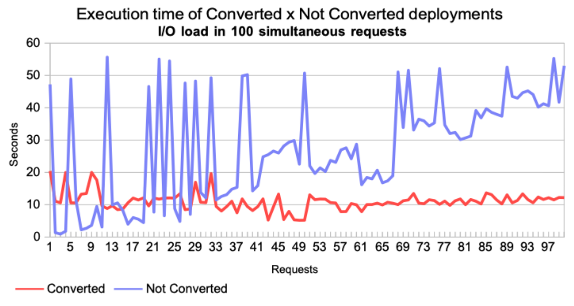

Figure 9: Execution of the I/O load test.

Table 1 presents the consolidated comparison of the results obtained in each type of test. It is possible to observe that the tests with simple load and CPU obtained inferior average times of execution in the server without FaaS. Already the execution with memory overhead and I/O were, on average, faster in servers using FaaS.

Table 1: Node2FaaS execution times.

App	Test	Best	Worse	Average
no FaaS	1-Simple	0,71	3,96	0,43
with FaaS	1-Simple	0,10	1,48	1,45
no FaaS	2-CPU	0,30	2,69	1,22
with FaaS	2-CPU	1,12	3,37	1,89
no FaaS	3-Memory	0,40	7,65	3,99
with FaaS	3-Memory	2,02	5,86	3,59
no FaaS	4-I/O	0,93	55,64	27,54
with FaaS	4-I/O	5,19	20,43	11,04

The analysis of the results allows one to infer, in general, that for a few requests the response time of the application running without FaaS tends to be better (test 1). For applications that consume a lot of CPU resources (test 2) with low competition, applications without FaaS have better results. However beyond a certain threshold, which in the test experiments was around 80 simultaneous requests, FaaS use benefits the runtime. For applications with high memory and I/O consumption (tests 3 and 4), the benefits of using FaaS are real, for any volume. It happens because in monolithic applications the consumption of such type of resource causes processing locks and it delays execution. On the other hand, in FaaS-oriented applications, there is a high parallelism in the consumption of these resources, reducing the effect of the locks. Thus, this type of application presents itself as a candidate for the adoption of the FaaS model for use in the cloud environment.

8 CONCLUSION

Function as a Service itself to be a cloud service model adherent to the current needs. However, the difficulties of structuring and consuming this service model drive the development of tools to improve this adoption process.

Therefore, the framework proposed by this work, Node2FaaS, proved to be efficient in the task of converting monolithic NodeJS applications to work with FaaS. The experiments showed that there were significant gains in the execution time of applications using FaaS, after conversion by Node2FaaS.

For applications that consume a lot of memory and/or perform a lot of I/O files gains after conversion have reached 170%. In addition, FaaS gain did not require investment in application tuning since Node2FaaS did all the code conversion and publishing work on the cloud provider.

However, as shown in the experiments, there are application types that do not benefit from the use of FaaS. The developed tool does not make an evaluation of the inner workings of the original application functions, but rather just reads and converts them. A future task would be to include an analysis in the conversion phase to decide whether to convert the function or to let it run locally.

Another future work is to allow the developer to define functions that should not be converted to FaaS. This increases the flexibility of Node2FaaS, and makes it more suitable for a wider range of applications.

In addition, since the experiment in this work was conducted using Amazon's Lambda service, it is important to conduct experiments with FaaS services from other providers such as Google's Cloud Functions and Microsoft's Azure Functions.

REFERENCES

Amazon (2018). Aws. https://aws.amazon.com/. Accessed: 2018-11-25.

Basili, V. R., Lindvall, M., Regardie, M., Seaman, C., Heidrich, J., Münch, J., Rombach, D., and Trendowicz, A. (2010). Linking software development and business strategy through measurement. *Computer*, 43(4):57–65.

Billock, M. (2017). The pros and cons of aws lambda. https://dzone.com/articles/the-pros-and-cons-of-aws-lambda. Accessed: 2018-12-01.

Chapin, J. and Roberts, M. (2017). *What is Serverless*. Oreilly.

Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C., and Hu, B. (2015). Everything as a service (xaas) on the cloud: Origins, current and future trends. In *2015 IEEE 8th International Conference on Cloud Computing (CLOUD)*, volume 00, pages 621–628.

Gartner (2018). Western europe context: Magic quadrant for cloud infrastructure as a service, worldwide. https://www.gartner.com/doc/3876869?ref=mrktg-srch. Accessed: 2018-11-25.

Google (2018). Google cloud. https://cloud.google.com. Accessed: 2018-11-25.

IEEE (2018). Interactive: The top programming languages 2018. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018. Accessed: 2018-11-25.

Larrucea, X., Santamaria, I., Colomo-Palacios, R., and Ebert, C. (2018). Microservices. *IEEE Software*, 35(3):96–100.

Lewis J, F. M. (2018). Microservices—a definition of this new architectural term. http://martinfowler.com/articles/microservices.html. Accessed: 2018-11-25.

Malathi, M. (2011). Cloud computing concepts. In *2011 3rd International Conference on Electronics Computer Technology*, volume 6, pages 236–239.

Mell, P. and Grance, T. (2011). The nist definition of cloud computing. *National Institute of Standards and Tecnology*.

Microsoft (2018). Azure functions. https://azure.microsoft.com/pt-br/services/functions/. Accessed: 2018-11-25.

Mithun Satheesh, Bruno Joseph D'mello, J. K. (2015). *Web Development with MongoDB and NodeJS*. Packt Publishing.

NPM (2018). About npm. https://docs.npmjs.com/about-npm/. Accessed: 2018-11-25.

Paula, G. S. d. (2018). Avaliação de serviços serverless: um experimento piloto. http://repositorio.roca.utfpr.edu.br/jspui/handle/1/10033. Accessed: 2018-12-01.

Savage, N. (2018). Going serverless. *Commun. ACM*, 61(2):15–16.

Spillner, J. (2017). Transformation of python applications into function-as-a-service deployments. *CoRR*, abs/1705.08169.

Spillner, J. and Dorodko, S. (2017). Java code analysis and transformation into AWS lambda functions. *CoRR*, abs/1702.05510.

Spoiala, C. (2017). Pros and cons of serverless computing. https://assist-software.net/blog/pros-and-cons-serverless-computing-faas-comparison-aws-lambda-vs-azure-functions-vs-google. Accessed: 2018-12-01.

Yoo, C. S. (2011). Cloud computing: Architectural and policy implications. *Springer Science and Business Media*.

Zimmermann, O. (2017). Microservices tenets: Agile approach to service development and deployment. *Comput Sci Res Dev*, 32:301.