

Remote Procedure Call Approach using the Node2FaaS Framework with Terraform for Function as a Service

Leonardo Rebouças de Carvalho^a and Aleteia P. F. de Araujo^b

Department of Computer Science, University of Brasilia, Brasilia, Brazil

Keywords: Cloud Computing, Multicloud, Orchestrators, Node2FaaS, Function as a Service, Terraform, AWS Lambda, Google Functions, Azure Functions.

Abstract: Cloud computing has evolved into a scenario where multiple providers make up the list of services that process client workloads, resulting in Functions as a Service. Because of this, this work proposes an approach of using RPC based FaaS. Using the Node2FaaS framework as a NodeJS application converter and integrated with Terraform as a cloud orchestrator. So, CPU, memory and I/O overhead tests were performed on a local environment and on the three main FaaS services: AWS Lambda, Google Functions and Azure Functions. The results showed significant runtime gains between the local environment and FaaS services, reaching up to a 99% reduction in runtime when the tests were run on cloud providers.

1 INTRODUCTION

Cloud computing has brought a different way of handling computing resources. Service-oriented clouds deliver multiple computational capacity utilization models, abstracting complexities according to customer needs. Thus, the client decides what level of involvement they want to have with the computational resource they are using.

In this scenario it is important to study mechanisms that exploit the full potential of these available resources to obtain the best possible outcome. This work proposes the adoption of an architecture based on remote procedure calls, in this case Function as a Service services, using the Node2FaaS (Carvalho and Araújo, 2019) Framework for NodeJS application conversion and Terraform (Brikman, 2017) as a public cloud orchestrator.

For this work a NodeJS application was developed, whose functions are intended to apply a controllable stress load on the server computational resources, such as CPU, memory and disk. This application has been converted using Node2FaaS to process its functions in FaaS services and take advantage of all the benefits associated with the cloud computing paradigm, especially the elasticity that allows a better load distribution.

2 FUNCTION AS A SERVICE

Programming models have been evolved over time, driven by software optimization, as well as their maintainability, scalability, and business alignment (Basili et al., 2010). Initially, the software was developed monolithically, meaning the entire application was contained in a single software block. This occasional model is a tight coupling between application components, leading to degradability of maintainability. Today, it is common for system architects to design their applications in a modular fashion, separating different complexity blocks and related feature sets (Larrucea et al., 2018).

To meet the demand for technology platforms that support software development paradigms, whether monolithic or segmented, the traditional infrastructure management model needs to invest heavily in installation and configuration tasks. Enabling an environment to receive source code and execute it properly usually takes many hours of work. To streamline this effort, cloud computing has defined the PaaS (Mell and Grance, 2011) service model, whose delivery consists of a fully configured platform ready to receive source code. However, this type of service still exposes some infrastructure details to the developer who eventually still needs to ensure, among other things, the elasticity of the environment. In addition, the provisioned environment must be maintained during the development period, and this increases the final project cost.

^a  <https://orcid.org/0000-0001-7459-281X>

^b  <https://orcid.org/0000-0003-4645-6700>

In this context, providers are now offering a service model that delivers a completely encapsulated platform for software development and charges only for the processing effectively performed by the platform. This type of service, known as Function as a Service (FaaS) (Spoiala, 2017), has gained much evidence recently as its approach meets the architectural model that has become widespread today the microservices paradigm. Figure 1 shows how to execute modules in monolithic environments compared to those oriented to microservices.

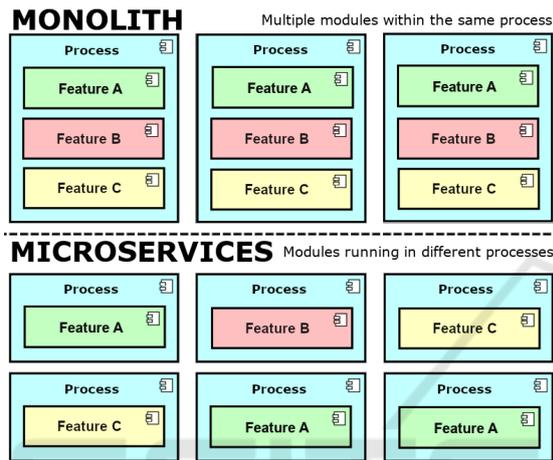


Figure 1: Segmentation of Features with Microservices and Monolithic.

2.1 FaaS Providers

Today's leading public cloud providers have FaaS services in their catalogs. Pioneer Amazon offers the AWS Lambda (Amazon, 2019) service. This service allows processing of NodeJS, Python, Java, Go, Ruby and C# through the .NET Core and provides a monthly package of one million free requests before charging for the service. The provider has several partnerships for deployment, monitoring, code management, and security.

Google offers the Cloud Functions (Google, 2019) service. This service allows processing of functions written in Python, NodeJS and Go and provides an initial quota of two million requests per month and only starts charging when this quota is exceeded.

Microsoft, through its Azure cloud service, makes Azure Functions (Microsoft, 2019) available. This service allows loading of functions written in C#, F#, NodeJS, Java, Python, or PHP (Microsoft, 2019). The provider states in its portal that the functions are available in a Windows environment, although this is transparent to the user. Despite this, there is a forecast of availability of environments using Linux. It offers

the first million service calls for free each month.

Alibaba Cloud offers the Function Compute (Alibaba, 2019) service, which allows processing of code written in NodeJS, Java, Python, Go, PHP and C#. This service also offers a free monthly starting quota of one million requisitions, and charges different rates in each region where it operates.

The IBM Cloud Functions (IBM, 2019) service offered by IBM Cloud handles functions written in NodeJS, Python, Go, Java, PHP, Ruby, Ballerina and Swift (Apache, 2019b) through Apache OpenWhisk¹ containers. With OpenWhisk you can process functions over Docker containers. As such, it is able to extend the possibilities of processing languages to any that can be installed in a Docker container. The provider offers free processing for five million requests per month.

In December 2018, the Oracle Cloud provider announced that it would launch the Oracle Function service in 2019 along the lines of Function as a Service (Oracle, 2019). Like IBM Cloud Functions, Oracle Functions will be supported by an open source processing platform, but unlike IBM, Oracle has decided to use the Fn framework². Since Fn runs over Docker, it can potentially process code in any programming language.

To address this amount of services, as well as all the complexity related to using FaaS services from NodeJS applications, the Node2FaaS Framework (Carvalho and Araújo, 2019) was created, which uses a remote procedure call approach to convert applications written in NodeJS with local processing to work with processing in FaaS services.

3 Node2FaaS FRAMEWORK

The Node2FaaS framework processes the original source code of a NodeJS application offered as input for its execution, and converts it to an application whose functions are performed in a FaaS service. The internal code of the functions is converted and deployed automatically created on the provider. FaaS service API calls, which correspond to the original function code, are swapped in for of the original function definitions. The product of this processing is a

¹Apache OpenWhisk it is an open source, distributed platform that performs functions in response to events at any scale. OpenWhisk manages infrastructure, servers, and sizing using Docker (Apache, 2019b)

²The Fn project is a serverless, open source platform, container native, which can run both in the cloud and locally. It's easy to use, supports all programming languages, is extensible and high-performance (Fn, 2019).

new application, based on the original, whose actual execution of its functions does not occur in the environment where the application is being executed, but in an external FaaS service.

The architecture of the Node2FaaS Framework is presented in Figure 2. It is possible to observe in this figure that in each module of an application there may be functions. Within each function is code that performs some useful operation for the software. Once submitted to Node2FaaS, this code will be published to the cloud provider, and instead, in the converted application, a URI call will appear pointing to the provider-provided REST API during publishing. Thus, the original function code is transferred to the cloud service and then consumed through requests using the HTTP protocol.

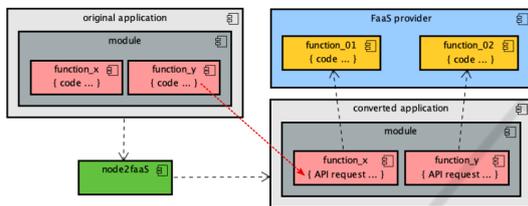


Figure 2: The Node2FaaS Solution Architecture.

To accomplish its mission, Node2FaaS has been internally segmented into modules that fulfill specific tasks and integrate with each other. Thus, working in a coordinated fashion, these modules receive inputs and return results that enable the processing of applications and generation of a new application using the proposed approach.

Figure 3 presents the internal structure of version 1.0 of the framework, and it is possible to verify the existence of a main module, called index, whose role is to coordinate the other modules. In addition to the main module, Node2FaaS consists of the following modules:

- **Common Functions:** Concentrates a set of utilities that are commonly used among the other modules;
- **Preparation:** Ensures that the requirements for proper framework execution are met, and are supported by the following submodules:
 - **Output Structure:** Responsible for creating the target application directory;
 - **Accreditation:** Responsible for obtaining and storing credential information from providers;
 - **Provider Interface:** Responsible for communicating with the providers’ services and abstracting their complexities for the other modules. The version used in this work is integrated with Terraform.

- **Converter:** Coordinates the conversion process and relies on the following submodules:

- **Code Extraction:** Responsible for extracting internal source code from functions;
- **Normalization:** Responsible for making the source code executable in the FaaS service;
- **Assembly:** Responsible for assembling the new definition of functions after publication in the provider;
- **Compression:** Some services advise that code be compressed before publication, this module is responsible for performing this task;
- **Publication:** Responsible for requesting publication to the provider interface module and handling its return.

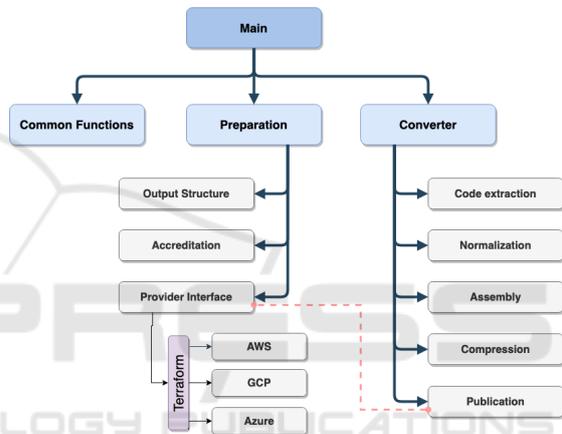


Figure 3: Node2FaaS Framework Composition.

4 EXECUTION FLOW

Once available in the local environment (after installation), the Node2FaaS flow is started from the “node2faas” application. If the framework was installed via NPM, this application will be registered in the machine path and can be run directly from the command: **node2faas -target [path to the app]**. Otherwise it, will be necessary to go to the directory where the framework was downloaded, grant execute permission, and only then run Node2FaaS. Figure 4 presents the activity flow of the Node2FaaS execution process.

Having an active account with the provider is essential because initially the framework looks for cloud access credentials. If it does not find the credentials file, the application prompts the user to enter their username and token for access to cloud services. After that, the system creates the credentials file and will no longer prompts for this in future executions.

Once the credential has been obtained, and an application is provided for conversion by Node2FaaS, it is subjected to a conversion process that will parse the application code for definitions of functions to perform the conversion, as shown in Figure 4.

During the process, if an “include” command is encountered, then the target file is also searched for conversion candidate functions, and this process repeats recursively until no file for inclusion is found.

That way, when the application finds a role, it will effect cloud access to create a new FaaS role. After receiving confirmation of role creation, the application obtains the service access URI, and creates the request within the original role definition. That way, the call to the function remains unchanged and its operation on the cloud platform is made completely transparent.

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’s FaaS service. The converted application maintains the same signature as the original functions, allowing its use to remain unchanged for the requesting processes of the functions. This eliminates the need to make adjustments to the application.

In order to meet the multi-provider provisioning requirement, as advocated by the Multicloud concept, and that the effort to keep the framework up-to-date with the constant updates promoted by the providers in their APIs, it was decided to adopt an orchestrator who could assume this assignment. Several tools were analyzed such as Cloudformation (AWS, 2019), AriaApache (Apache, 2019a), Alien4Cloud (Alien4Cloud, 2019), Celar (Loulloudes, 2019), DICER (Guerriero, 2019), OpenTOSCA (TOSCA, 2019), xOpera (Borovšak, 2019), CloudAssembly (VmWare, 2019) and Cloudify (Cloudify, 2019), however considering the requirements that would allow the Node2FaaS Framework to be incorporated denies the use of Terraform (Brikman, 2017).

5 Terraform

Terraform (Brikman, 2017) is an open source tool created by HashiCorp that allows the definition of infrastructure as code using a simple declarative programming language (Brikman, 2017). It enables deployment and management of this infrastructure across multiple public cloud providers (e.g. AWS, Azure, Google Cloud, and DigitalOcean), as well as on private virtualization platforms (e.g. OpenStack and VMWare) using a few commands (Brikman, 2017).

Terraform currently supports over 30 different

providers. When it comes to servers, Terraform offers several ways to configure them and connect them to existing configuration management tools (Shirinkin, 2017). From a definition file Terraform interacts with a cloud provider and performs the service provisioning according to the requested configuration. The service can be either a virtual machine, a database, a storage service, or whatever the provider supports.

Terraform tracks the momentary state of the infrastructure it creates and applies delta changes when some feature needs to be updated, added, or deleted (Shirinkin, 2017). It also provides a way to import existing resources and target only specific resources. It is also easily extensible with plugins (Shirinkin, 2017).

6 RELATED WORKS

The work (Spillner, 2017) brings an approach of converting Python-written applications to Python deployments in the AWS Lambda service. The application built by Spillner, called Lambada, processes a Python application and converts it to the appropriate code to be instantiated in the cloud. If the user has the AWS client installed and properly configured on the machine, Lambada automatically deploys, but the entire client setup process is up to the user. This limits the use of this converter to users who are able to correctly configure the provider client in their local environment.

Furthermore, that article (Spillner, 2017) is limited to using Lambada for converting single-function applications. In production environment applications, it is common to have multiple functions for the execution of an application. Node2FaaS, on the other hand, enables 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 as Lambada, but for Java applications. In this paper the authors question the economic feasibility of running a Java application entirely using FaaS, and whether there is a possibility to automate the application conversion process. They implemented a tool called Podilizer, and performed experiments using it. The results were classified as promising, however, only for academic purposes, not presenting effective application capacity, since the authors found difficulties in using the applications resulting from the conversion. On the other hand, Node2FaaS is not simply intended for academic experimentation, but for effective use by NodeJS developers.

The Serverless (Serverless, 2019) framework pro-

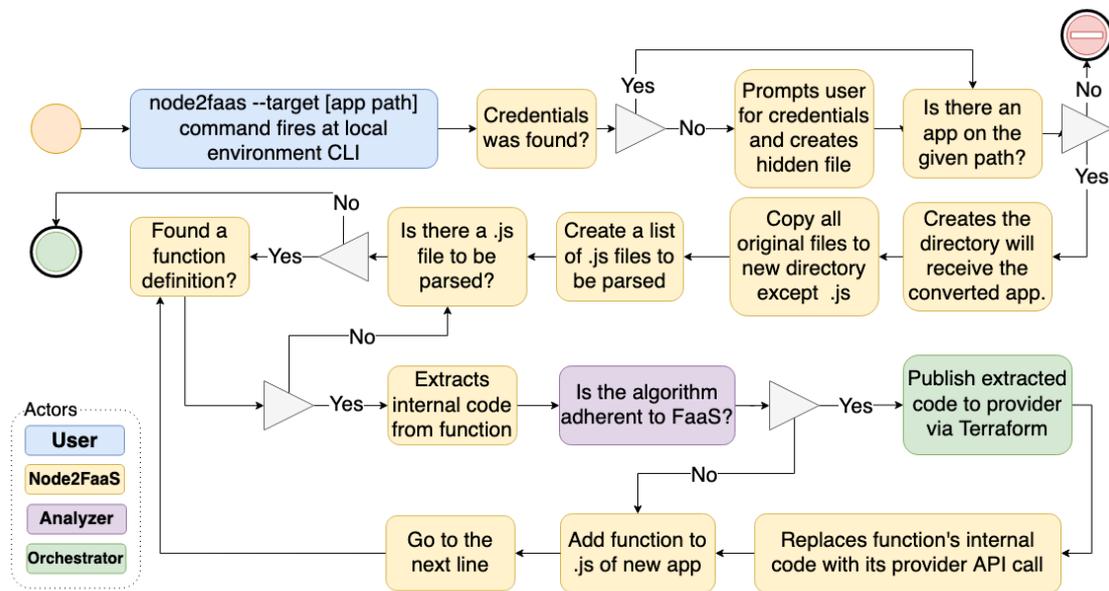


Figure 4: Node2FaaS Application Conversion Process.

vides a platform for deploying applications to major FaaS service providers such as AWS Lambda, Azure Functions, Google Cloud Functions, IBM OpenWhisk, and frameworks like Fn. It abstracts the differences between services and allows deploys to run on FaaS services through a CLI. Written in NodeJS, Serverless needs to be installed directly in the local environment and has templates for interaction with each provider that need to be customized to the user’s needs. It is an open source tool, however it has a paid enterprise version that offers several additional features such as: web dashboard, debugging tools, best practice injection, security simplicity, among others.

Although Serverless is a robust platform for managing FaaS deployments, it works with the various programming languages that each provider has, and so there is no specific focus on NodeJS. Thus, the framework does not perform any assessment of the nature of the code and its adherence to the Function as a Service approach, leaving it to the developer to decide which functionality should be leveraged for FaaS services.

Zeit Now (Zeit, 2019) is a global deployment network built on all existing cloud providers. This makes teams productive by removing servers and configurations, providing a seamless development experience for building scalable, modern web applications (Zeit, 2019). In practice this service acts as a layer above FaaS providers. It allows the creation of web applications through its web platform or integration with code versioning tools such as GitHub³.

³<https://github.com>.

The Zeit Now (Zeit, 2019) platform has a billing model similar to the overlapping providers. It offers a daily quota for executions and resource allocation that allows the user to try the service for free. In addition, it has a CLI interface and Desktop client for Windows and MacOS. It supports a wide range of programming languages and has several plugins that allow integration with other development tools.

Such as Serverless, Zeit Now simply receives the code and passes it on to the provider, acting as an intermediary between the developer and the FaaS service. Because it works with many languages, there is no specific focus on NodeJS, nor is there an assessment of the developer-built algorithm adherence to the serverless paradigm implemented by FaaS services.

Claudia.js (Claudia.js, 2019) is a framework that makes the task of deploying NodeJS projects on AWS Lambda easy. It automates all error-prone deployment and configuration tasks. This framework acts as a pre-AWS Lambda layer, whose function boils down to simplifying interaction with Amazon’s FaaS service. By interacting with only one provider, the Claudia.js framework limits the possibilities of using the cloud model, strengthening vendor lock-in and thus not representing significant gains for its users.

Table 1 presents a comparison between the presented solutions. In this table it is possible to observe that two of these solutions do not have commercial applicability, only academic, and thus do not effectively offer added value through the FaaS model. Among the other solutions, only Node2FaaS offers evaluation

Table 1: Comparative Table of FaaS-oriented Solutions.

| Feature | Lambda | Podlizer | Serverless | Zeit Now | Claudia.js | Node2FaaS |
|-------------------------|----------|----------|------------|----------|------------|-----------|
| Supported Languages | Python | Java | Multiple | Multiple | NodeJS | NodeJS |
| Effective Applicability | Academic | Academic | General | General | General | General |
| Multicloud Support | No | No | Yes | Yes | No | Yes |
| Multicloud Orchestrator | No | No | Hidden | Hidden | No | Yes |
| FaaS Fitting Analysis | No | No | No | No | No | Yes |

execution to classify the adherence of a code to the FaaS approach. Using a multicloud orchestrator is also a differential of Node2FaaS. As such, the framework follows the Orchestrator's evolution and as new integrations with other services and/or providers are added, Node2FaaS is also affected by these enhancements.

7 METHODOLOGY

In order to experiment with the proposed approach, the functions presented in Listing 1, 2 and 3 were developed with the purpose of overloading the CPU, the memory and the machine disk, respectively.

Listing 1: CPU Bound Function.

```
1 exports.cpu = function (req, res) {
2   var result = 0;
3   for (var x in req.params){
4     for (var i = 0;
5         i < parseInt(req.params[x]);
6         i++) {
7       result = parseInt(result)*i
8         +
9         parseInt (req.params[x])*1;
10    }
11  }
12  res.json(result);
13  ;
```

These functions were added to a NodeJS application that was converted to RPC approach in FaaS, using the Node2FaaS framework for the following FaaS providers: AWS Lambda, Google Functions, and Azure Functions. Both the original application and the converted applications were subjected to test batteries that allow the evaluation of their performance in terms of runtime.

Listing 2: Memory Bound Function.

```
1 exports.memory = function (req, res) {
2   var result = new Array();
3   for (var x = 0;
4       x < parseInt(req.params["a"]);
5       x++) {
6     result[x] = new Array();
7     for (var i = 0;
8         i < parseInt(req.params["b"]);
9         i++) {
10      result[x][i] = x+i;
11    }
12  }
13  result = eval(result.join("++"));
14  res.json(result);
15  ;
```

Listing 3: I/O Bound Function.

```
1 exports.io = function (req, res) {
2   var rd = Math.random()*1000;
3   var prefix = Math.floor(rd);
4   var result = 0;
5   const fs = require("fs");
6   for (var x = 0;
7       x < parseInt(req.params["a"]);
8       x++) {
9     for (var i = 0;
10        i < parseInt(req.params["b"]);
11        i++) {
12      fs.writeFileSync("/tmp/"+
13        prefix+
14        x+i,
15        "Node2FaaSTest",
16        function(err) {
17          if(err) {
18            return console.log(err);
19          }
20          console.log("File saved!");
21        });
22      fs.unlinkSync("/tmp/"+
23        prefix+
24        x+i);
25      result++;
26    }
27  }
28  res.json(result);
29  ;
```

Table 2 shows the composition of the test parameters that were performed. In all, 1080 test cases were executed. This test suite was used to plot an average axis for each of the 10 runs of each type (CPU, memory, and I/O).

The environment against which local tests were run had the following configuration:

- **Machine:** Macbook Pro 13" Mid 2012
- **CPU:** Intel Core i5 2,5 GHz
- **Memory:** 16GB 1600 Mhz DDR3
- **Operational System:** MacOSX Mojave(10.14.6)

Two rounds of tests were performed, the first on December 14, 2019 and the second on December 23, 2019. A "shell script" has been developed to command the tests and store the results. At the end, the data were consolidated by calculating the average of the executions of each test case and then the graphs were generated with the results obtained for each provider and for the local environment in each type of test (CPU, memory and I/O).

7.1 Results

The results showed a marked difference in the execution time of the local tests in relation to the times

Table 2: Test Parameters.

| Test | Environment/Providers | Load time | Concurrency | Executions |
|--------|------------------------|-------------|-------------|------------|
| CPU | Local, GCP, AWS, Azure | 1s, 5s, 10s | 10, 50, 120 | 10 |
| Memory | Local, GCP, AWS, Azure | 1s, 5s, 10s | 10, 50, 120 | 10 |
| I/O | Local, GCP, AWS, Azure | 1s, 5s, 10s | 10, 50, 120 | 10 |

registered by the applications adopting the RPC approach. As can be seen from Figure 5, which shows the consolidated average times for CPU test cases, while in local execution the average was around 183 seconds, FaaS services fared much better, reaching 9.62 seconds for AWS. Even the worst-performing provider, Azure, still had a result that represents less than 30% of the result from local processing.

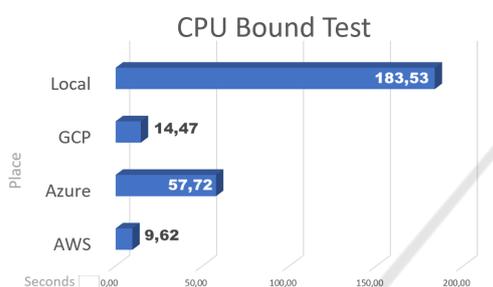


Figure 5: CPU Bound Tests Result.

Figure 6 shows an even more discrepant result between local execution times and FaaS services times, in this case, for tests with high memory consumption. AWS and GCP recorded average times of 2.45 and 6.08 seconds, respectively. The Azure provider recorded a higher result, with an average time of 64.04 seconds, but still much lower than the 247.03 seconds recorded as the average of the tests processed on the local machine.

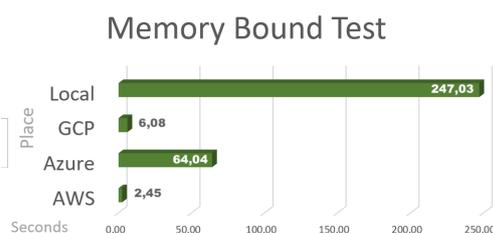


Figure 6: Memory Bound Tests Result.

The test case that aimed to stress disk activity yielded an intriguing result. While AWS and GCP providers recorded averages of 1.73 and 4.17 seconds of runtime and local testing resulted in an average of 55.50 seconds, as can be seen in Figure 7, the Azure provider was unable to finalize any test cases. Considering that the test cases ranged from 10, 50, and 120 concurrent requests and that for a single request

the Azure provider was able to successfully fulfill the request, it can be inferred that Azure’s FaaS service cannot satisfactorily handle situations involving writing and deleting files.

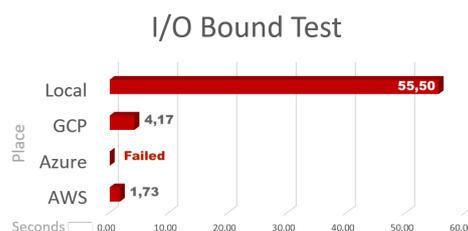


Figure 7: I/O Bound Tests Result.

These results show the effectiveness of the elasticity actuation that cloud computing offers. The perceived discrepancy between local and FaaS execution is due to the queuing of processing that occurs in local execution, while in FaaS execution there is in fact parallel processing, as the provider is in charge of delivering more computational resources when these are perceived as necessary. Table 3 shows the reduction percentages obtained in each provider. It is possible to observe that for the memory test, it was possible to obtain a gain of 99% using AWS provider processing.

Table 3: Runtime Reduction Table.

| Tests | 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% |

8 CONCLUSION

The tests results showed significant differences in the execution time of local functions under stress in relation to the same functions using the proposed approach. For CPU consumption tests the reduction reached 94.75%, while for I/O tests the reduction reached 96.88% and for memory tests there was a significant reduction of 99%.

Given the above, the contribution of Function as a Service services to workloads that require large amounts of computational resources is clear. Because running local workloads that require a lot of compu-

tational resources such as CPU, memory, or I/O consumes more runtime than if they were sent for processing in FaaS services. In addition, considering this context, the Node2FaaS framework proved to be efficient in the task of automatically converting NodeJS-written applications to function using FaaS services through an RPC approach. Similarly, Terraform, acting as the internal orchestrator of Node2FaaS, proved to be very efficient in bringing local application code to the cloud.

However, it is important to note that some provider constraints may make it impossible to use this model in certain provider services, as evidenced by disk stress tests performed on Azure services, which failed in 100% of cases.

In future works it would be important to perform the same tests with real applications that merge the use of computational resources as it occurs in production environments. In that kind of experiment will be possible to see if the same approach yields such significant results when loads are less concentrated on a specific resource. In addition, it is hoped that in the near future the Azure provider will mature their FaaS solution so that disk stress test results can be obtained and properly analyzed.

REFERENCES

- Alibaba (2019). Alibaba functions. <http://alibabacloud.com/products/function-compute>.
- Alien4Cloud (2019). Alien4cloud. <http://alien4cloud.github.io>.
- Amazon (2019). Aws. <https://aws.amazon.com/>.
- Apache (2019a). Apache aria toasca orchestration engine. <https://ariatosca.incubator.apache.org/>.
- Apache (2019b). What is apache openwhisk? <https://openwhisk.apache.org/>.
- AWS (2019). Aws cloudformation. <https://aws.amazon.com/pt/cloudformation/>.
- 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.
- Borovšak, T. (2019). xopera orchestrator. <https://github.com/xlab-si/xopera-opera>.
- Brikman, Y. (2017). *Terraform: Up and Running: Writing Infrastructure as Code*. O'Reilly Media.
- Carvalho, L. and Araújo, A. P. F. d. (2019). Framework node2faas: Automatic nodejs application converter for function as a service. In *Proceedings of the 9th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, pages 271–278. INSTICC, SciTePress.
- Claudia.js (2019). Claudia.js: Serverless javascript, the easy way. <https://claudiajs.com/>.
- Cloudify (2019). Getting started. <https://cloudify.co/getting-started/>.
- Fn (2019). Fn project: Open source. container-native. serverless platform. <https://fnproject.io/>.
- Google (2019). Cloud functions. <https://cloud.google.com/functions/>.
- Guerrero, M. (2019). Dicer. <https://github.com/DICERs/DICER>.
- IBM (2019). IBM cloud functions. <https://www.ibm.com/br-pt/cloud/functions>.
- Larrucea, X., Santamaria, I., Colomo-Palacios, R., and Ebert, C. (2018). Microservices. *IEEE Software*, 35(3):96–100.
- Loulloude, N. (2019). The celar project. <https://github.com/CELAR/c-Eclipse>.
- Mell, P. and Grance, T. (2011). The NIST definition of cloud computing. *National Institute of Standards and Technology*.
- Microsoft (2019). Azure functions. <https://azure.microsoft.com/pt-br/services/functions/>.
- Oracle (2019). Announcing oracle functions. <https://blogs.oracle.com/cloud-infrastructure/announcing-oracle-functions>.
- Serverless (2019). Serverless: Build apps with radically less overhead and cost. <https://serverless.com>.
- Shirinkin, K. (2017). *Getting Started with Terraform*. Packt Publishing.
- 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.
- Spojiala, 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>.
- TOSCA, O. (2019). Open toasca. <http://www.opentosca.org/>.
- VmWare (2019). Introducing vmware cloud assembly, vmware code stream and vmware service broker. <https://blogs.vmware.com/management/2018/08/introducing-cloud-automation.html>.
- Zeit (2019). Zeit now: The global serverless platform. <https://zeit.co>.