Quality aspects of serverless architecture: an exploratory study on maintainability

Louis Racicot[1], Nicolas Cloutier[1], Julien Abt[2], Fabio Petrillo[2]
[1]*École Polytechnique de Montréal, Montréal, Canada*
[2]*Université du Québec à Chicoutimi, Chicoutimi, Canada*
{*louis.racicot,nicolas.cloutier*}*@polymtl.com, julien.abt1@uqac.ca,fabio@petrillo.com*

Keywords: Function as a Service, Serverless, Software architecture, Maintainability, Cloud computing

Abstract: Serverless architecture is emerging more and more popular as the tools are becoming cheaper and more accessible. This way of designing an architecture presents many advantages especially for computing intensive and event-driven applications. Stateless functions are the foundation for these types of architectures, and it might cause an impact on the maintainability of the software. In this paper, we statically analyzed 25 open-source projects using serverless architecture to bring out metrics that applies to the different characteristics of software maintainability. We found out that some characteristics are positively impacted whilst some other seems to be negatively impacted. This paper thus provides findings on the current state of the projects' maintainability using serverless architecture.

1 Introduction

Serverless architecture is an execution model where the provider dynamically manage the resources requested by an application. Despite the name, serverless computing still require servers. In fact, serverless computing can be seen as leasing a server for a very small amount of time before releasing it for another application to use it. Most modern application use serverless computing for CPU heavy task and dedicate smaller server that they can scale horizontally for the routing. Typically, these smaller servers are stateless so that they can send the CPU heavy tasks to the lambda functions asynchronously and continue with the execution until the functions return its results. This create a non blocking pipeline that is very efficient, very scalable and easily maintainable. A big promoter of this architecture is Netflix, proving that the leap of faith is actually fealisable when done correctly.

Since its inception, consumers have discovered a growing number of use cases for the serverless architecture. The most notable use cases are automated backups, scheduled Cron jobs, processing uploaded object (ie: on S3), analyzing log or simply processing and arbitrarily payload. All theses tasks are known to be CPU extensive, but each one of them already had one or many solutions in place. The serverless payper-use business model is interesting for all of theses task. Considering that hosting and compute power is one of the biggest recurring cost for startup aside from workforce, it is interesting for them to know the trade-off of each solution in order to make the most profitable choices. Given all of the possible services that can achieve the same results, it can be expensive to prototype them all to assert which service is the most suitable for a given problem. Serverless architecture is gaining popularity due to its interesting pricing model. The growth in popularity of this model certainly raises concerns as it is used as a replacement for existing service architectures that are at the moment mature, somehow scalable and maintainable (Shadija et al., 2017).

In the recent years, Amazon along with other big technology companies, have started to offer serverless options as part of their cloud systems solutions. Among others, there are known solutions such as AWS Lambda (Amazon, 2017b), Microsoft Azure Functions (Microsoft, 2017) and Google Cloud Functions (Google, 2017).

As serverless architecture is fairly new, it is yet to be proven beneficial as a solution. In this paper, we analyzed the maintainability of different serverless project. Our main research questions is:

RQ: Does the strain of creating serverless functions impact the maintainability?

The purpose of this paper is to identify maintainability aspects in serverless based solutions and to un-

cover those findings for potential future research. Furthermore, we will discuss the current popular methods for deploying serverless solutions and current trends in configuration management for this type of project. To present our results and discussions, we first define what is serverless architecture. Then, we present the related work. After that, present be presented our methodology along with our results. We then discuss our findings. Finally, we present our conclusion and future work.

2 Background

2.1 Serverless and FaaS

Serverless architecture is a model of cloud computing services that cloud providers offer and manage dynamically allocated computational resources, pricing by consumed resources (for example number of requests) rather than on prepaid units of capacity. Sbarski (Sbarski, 2016) defined five principles of serverless architectures:

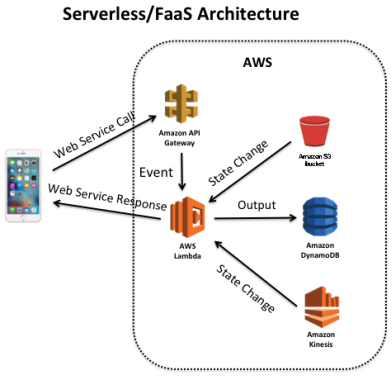
1. Use a compute service to execute code on demand (no servers).
2. Write single-purpose stateless functions.
3. Design push-based, event-driven pipelines.
4. Create thicker, more powerful front ends.
5. Embrace third-party services.

Function as a service (FaaS) is a type of serverless architecture that implies a service written with a set of individual functions which are all entry points in the program. These functions are individually deployed and versioned, and are stateless in the sense that each public function call is run in a new runtime environment. Persistence is only available through external storage solutions such as a database. In fact, Faas is one way to implement serverless architecture.

In general, theses functions are connected to the client services with an independent interface like a REST API. Stateless functions are usually better for short-run time execution even if it is possible to let it run longer. It is usually designed to respond to a query, process some data and give a response. It is not designed be pending and waiting for other queries. The figure 1 is an example of a typical function as a service.

Compared to other solutions such as PaaS, SaaS or IaaS (*Infrastructure as a Service*), FaaS is the solution that allows the most control over the code while allowing no control at all over the infrastructure. That way, developers are able to focus only on the code

Figure 1: Example of a Stateless Function Generic Architecture on AWS Lambda (Golden, 2016)



and its deployment when the FaaS provider takes care of the whole server infrastructure and configuration (Baldini et al., 2017).

Notably, a serverless architecture is more cost-efficient than owning or renting a cluster due to the periods of non-utilization that are not billed to the consumer. A serverless architecture also make the code development easier, since the multi-threading and the scaling is taken care of by the provider. In counterparts, a serverless architecture suffer from some latency when a function has to be initiated after not being used for a long period of time. Also, for very high CPU intensive task, a serverless architecture might not be suited due to the limited resources allocated for each function. Finally, the consumer does not have the control of their server. This can make the deployment, the monitoring harder. This also constraint the user to a small subset of languages.

2.2 Maintainability

Software maintainability can be described by "*the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment*" (Committee et al., 1990).

Furthermore, the international standard ISO/IEC 25010:2011 defines maintainability as the "*degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers* " and breaks it down into five subcategories: *modularity*, *reusability*, *analysability*, *modifiability* and *testability* (ISO/IEC, 2010). It also states that maintainability includes the installation of updates and upgrades, which, in the context of FaaS, can be interpreted as the deployment.

Multiple definitions of maintainability by different software quality models exist. The McCall model (Cavano and McCall, 1978) describes maintainability using four criteria:

- Simplicity
- Conciseness
- Self-descriptiveness
- Modularity

Similarly, the Boehm's quality model (Boehm et al., 1976) defines maintainability as a product of multiple characteristics

- Testability
- Understandability
- Modifiability

Which themselves depends upon multiple sub-characteristics which are *Consistency*, *Accessibility*, *Communicativeness*, *Structuredness*, *Self-descriptiveness*, *Conciseness*, *Legibility* and *Augmentability*.

Finally, ISO 25010 (ISO/IEC, 2010) describes maintainability as a product of a few sub-characteristics:

- Analysability
- Modifiability
- Testability
- Modularity
- Reusability

To help up having a better understanding of what maintainability is, we sum up all the characteristics of different quality models in *Table 1*.

Visser (Visser et al., 2016) used ISO 25010 as quality model to define his metrics. An interesting part of his work is that even though simplicity and conciseness are not even part of ISO 25010, two metrics (*unit complexity* and *unit size*) can account for these characteristics. The understandability is the only characteristic not used in Visser's model. It can be calculated using the three metrics defined in Nazir's paper (Nazir et al., 2010). These metrics are namely *Inheritance*, *Coupling* and *Cohesion*. However, **most of the projects we analyzed don't use classes and therefore the inheritance metric does not apply**. Furthermore, we show in our results that projects using serverless technologies generally have a low coupling and are very small, which is a good sign for a strong cohesion. Because of that, we decided not to analyze understandability in detail, allowing us to use only Visser's model.

Table 1: Relation between maintainability characteristics and quality model

	Quality model		
	McCall	Boehm	ISO 25010
Simplicity	X		
Conciseness	X	X	
Self-descriptiveness	X	X	
Modularity	X		X
Testability		X	X
Understandability		X	
Modifiability		X	X
Analysability			X
Reusability			X

3 Maintainability and FaaS

In this section, we examine how a serverless architecture can impact the aspects of maintainability. For each of them, we formulate a hypothesis. This hypothesis will later on be verified based on the relation between the characteristics of maintainability and the empirically measurable properties of the system, as defined by Visser, as presented in *Table 2*.

Table 2: Relation of sub characteristics and system properties

	Volume	Duplication	Unit complexity	Unit size	Unit interfacing	Module coupling	Component balance	Component independence
Analysability	X	X	X				X	
Modifiability		X		X		X		
Testability	X			X				X
Modularity						X	X	X
Reusability			X		X			

Relation of sub-characteristics and system properties (Visser et al., 2016)

3.1 Modularity

ISO25010 describes modularity as the "degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components."(ISO/IEC, 2010) If we assume that most serverless projects are designed using this guideline, it should imply that serverless architecture complies with the modularity aspect of maintainability. Consequently, we can formulate our first hypothesis:

H1 : Software that uses serverless technologies have a low module coupling, their components are well balanced and are independent. Thus, it complies with the modularity aspect of maintainability.

3.2 Reusability

The reusability is the possibility of an asset to be used by more than one system. Since serverless functions are available by an HTTP request and that it runs completely independently from the software that uses it, we can suppose that serverless architecture enforce the reusability aspect of maintainability. As GC. Fox at al. stated, serverless is good for short-running, stateless and event-driven codes such as micro-services (Fox et al., 2017). Furthermore, because of the nature of a serverless function, it only has one interface available but does not enforce a maximum number of parameters. However, we still suppose that most developers keep a small number of parameters.

H2 : Software that uses serverless technologies have small units of code with a small interface.

3.3 Analysability

The analysability of a project describes how easy it is to diagnose and test the components of the project. Visser states that the metrics that have an impact on the analysability are the size of the code base, the amount of duplicated code, the size of the units of code and the balance of the components. Serverless providers, though they limit the execution time of a function, do not have any restrictions for its volume. It is possible to run large pieces of code as FaaS, but we suppose that most serverless code is quite small. As said previously, we also suppose that the unit size should be small. However, we think that a serverless architecture does not have a positive impact on the code duplication in the case where a project uses multiple serverless functions. Because all functions have to be independent, we expect to find the duplication of libraries and helper functions. Finally, we suppose that serverless architecture does not have an impact on the component balance.

H3: Software that uses serverless architecture has a small serverless code base and small units of code, but it has duplicate code.

3.4 Modifiability

The modifiability of a system represents how easy it is to modify the code without introducing defects or degrading the quality of the existing product (Committee et al., 1990). The modularity and the analysability influence the modifiability because it includes the coding activities as well as the design, the documentation and the validation of the new code. Factors that influence the modifiability are the amount of duplicated code, the complexity of the code units and the coupling between the modules.

H4 : Software that uses serverless technologies tends to be easy to modify because it has simple units of code and low coupling.

3.5 Testability

The testability is the ease to define and execute tests. We already emit hypotheses concerning the volume of the code, the complexity of the code units and the component balance. However, we would like to add some qualitative details about this aspect. While tests can be easily defined and executed locally, the local environment might not behave exactly like the service provider. It is especially true if we want to test the execution time or if we have a lot of environment specific configuration. We can overcome these problems if the deployment on a test environment is easy to do. We will talk about this in the next subsection.

H5 : Software that uses serverless technologies are easy to test because they have a small code base, they have simple units of code and they are made of independent components.

4 Aspects of code management

4.1 Deployment

The ability to easily deploy or install an update is a crucial part of the maintainability. We consider that

even if a code is very easy to modify, developers or release engineers need to be able to deploy their modifications quickly, without breaking the application or introducing deployment-related issues. When we are talking about deploying a serverless function, things can become complicated. Multiple functions can require to be deployed at the same time if their interface or behaviour changes.

How do you deploy the application in those cases? Do you have to deploy the function and the client at the exact same time? What if the function has multiple clients? Does your provider allow to have multiple versions of the function at the same time? And if so, would your application be flexible enough to allow the coexistence of multiple version of the function without producing inconsistencies in your data? These questions cannot be answered by empirical data from the code so instead, we will bring what we have found in the literature to verify our hypothesis.

First of all, there are some tools that can ease the deployment. For example, the *Serverless Framework*(Framework, 2017) offers a range of functionalities to quickly put code into production with a variety of cloud services providers.

Another important part of the deployment is the management of deployed versions. If multiple clients are using FaaS, you might not want to upgrade them all at the same time if you upload a new version of your function that breaks the backward compatibility. Some providers such as AWS have a very straight forward versioning system that allow multiple versions of the same function to coexist. Other providers like Google Cloud and Microsoft Azure don't have such an easy way of managing versions.

In any cases, there seems to have a certain overhead in the management of versions and thus, we formulate the following hypothesis:

H6: The complexity of deployment of serverless software is harmful for its maintainability.

4.2 Configuration management

Because the serverless functions are completely independent from each other and from the client's software, they can be seen as a different program. They can also be developed and maintained by different teams, which makes each serverless function a different project. There is also an explicit dependency between the functions and their clients, and there can be dependencies between the serverless functions themselves. On a large scale, this can lead to complex dependency problems likes the ones that package managers have to handle, and we haven't found a real solution to that problem yet. The technologies are still too young to be confronted to this problem, but it is something that could happen in the future. On a regular scale projects, however, this leads to a wide and still unresolved question that is way beyond the scope of this paper: mono-repository versus multi-repository (Potvin and Levenberg, 2016; Potencier, 2016; Cavale, 2016).

Having only one repository per lambda can help to set up continuous integration tools (Amazon, 2017a). On the other hand, it is much easier to manage each functions of a project in a single repository, especially if all functions are only bound to a single project.

H7: Configuration management for software that uses serverless technologies is still an open question.

5 Study Design

The maintainability of a project has multiple aspects. The most prominent one is the maintainability of the code which can be measured (Visser et al., 2016). We mainly investigated maintainability aspects using the following strategy:

1. Qualitatively discuss possible maintainability and management issues, formulating seven hypothesis (sections 3 and 4)
2. Extract meaningful metrics from FaaS projects to provide empirical results
3. Compare our discussion with empirical results to evaluate the state of maintainability
4. Discuss how code source is managed within configuration management system and deployment tools

To empirically analyze the impacts of FaaS on the code maintainability, we analyzed open-source projects from Github (Just Serverless, 2017) using serverless architecture.

5.1 Practices and tools

To extract the metrics, we used BetterCodeHub, an online static analysis service whose criteria ares base on guidelines for more maintainable code (Visser et al., 2016). BetterCodeHub test various maintainability aspects of project's source code. The results

are organized in a way that the issues are shown to the user of the service with specific reasons why it failed. It has the limitations of analyzing only GitHub projects and the size of the code base is limited to 100 KLOC in the free version. However, it was not a true limitation because of FaaS projects are usually small in terms of lines of code.

BetterCodeHub is used to test our hypothesis because it adopts the same guidelines criteria as SIG for evaluating the maintainability (Software Improvement Group, 2017; Visser et al., 2016). We can link our hypothesis to the guidelines using *Table 2*.

5.2 Analyzed serverless projects

To test these hypotheses, we manually inspected a curated list of serverless projects and we filtered them based on the following criteria: (1) the project must be based on the function-as-a-Service pattern; (2) the project must be of a non-trivial in complexity and in size; (3) the project must be analyzable by BetterCodeHub.

With theses criteria, we found 25 projects (table 3) varying from 1 function to 95 functions. We note that the project with only one function has been accepted due to its complexity. It is an SSH certificate authority with test coverage (Netflix, 2017). The projects are chosen to be as diverse as possible. We need to use projects from different authors and with different scope in order to ensure that the results won't be biased.

To find the number of functions, a manual inspection must be done. Depending on the framework and its version, the information about them is not at the same place. For example, the serverless framework is using a yml configuration file after the version 1.0 to manage every function (Serverless Inc., 2017).

6 Results

All these results have been compiled in *Table 3*. The biggest offenders for the guidelines in the selected projects are the size and simplicity of units, the duplication of codes and the size of the parameters. The modules and components guidelines are nearly complying.

Finally, the sum of all metrics can be reported to the initial characteristics of maintainability as shown on *Table 4* and discussed in the next section of this paper.

Our results show that most of analyzed projects separate concerns in modules, Couple Architecture Components Loosely, keep architecture components balanced, keep the codebase small, and Write Clean Code. However (1) 88% (22/25) of analyzed projects do not write short units of code; (2) 60% of projects (15/25) **write complex units of code** and 52% (13/25)do not use well automated tests; finaly (3) 60% (15/25) tend to have complex unit interfaces.

7 Discussion

With theses results in mind, we discuss the aspects of maintainability and management, evaluating our hypotheses.

7.1 Modularity

H1 : Software that use serverless technologies have a low module coupling, their components are well balanced and are independent. Thus, it complies with the modularity aspect of maintainability.

Serverless Programming is at its core a way to make services. We expect theses small services to increase modularity compared to a monolithic multilayer application (Shadija et al., 2017). They ensure a strong separation of the different behaviours with an interface and that they are all self-contained. Thus we are not surprised that our results seem to confirm a good modularity. All the different criteria for the first hypothesis are validated on the majority of the projects analyzed.

> *Software that use serverless technologies have a low module coupling, their components are well balanced and are independent.*

7.2 Reusability

H2: Software that use serverless technologies have small units of code with a small interface.

It seems that serverless projects tend to have big units of code and the majority of the analyzed projects do not respect the required criteria. However, we note that a bigger project, such as MoonMail, does not mean a greater chance of facing this issue. Serverless architecture should not prevent code reusability. It's still possible to split the code in multiple small units of code that are reusable. But it doesn't enforce or encourage that so it needs to come from within.

> *Serverless projects tend to have big units of code.*

Table 3: BetterCodeHub guidelines applied on serverless projects

	Write Short Units of Code	Write Simple Units of Code	Write Code Once	Keep Unit Interfaces Small	Separate Concerns in Modules	Couple Architecture Components Loosely	Keep Architecture Components Balanced	Keep Your Codebase Small	Automate Tests	Write Clean Code
microapps/MoonMail	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
jonatasschagas/langadventurebackend	✗	✗	✓	✓	✓	✓	✗	✓	✗	✓
craftship/codebox-npm	✗	✓	✗	✗	✓	✓	✓	✓	✓	✓
C0k3/session	✗	✗	✗	✗	✓	✗	✓	✓	✓	✗
agentmilindu/Serverless-Pre-Register	✗	✗	✗	✓	✓	✓	✓	✓	✗	✓
haw-itn/serverless-web-monitor	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓
bart-blommaerts/serverless_garage	✓	✓	✗	✗	✓	✓	✓	✓	✗	✓
michalsanger/serverless-facebook-messenger-bot	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓
laardee/serverless-authentication-boilerplate	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
airbnb/binaryalert	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
airbnb/streamalert	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓
Netflix/bless	✗	✗	✓	✗	✓	✓	✓	✓	✓	✓
apache/incubator-openwhisk	✗	✗	✓	✗	✓	✗	✓	✓	✗	✓
fnproject/fn	✗	✗	✓	✗	✓	✓	✗	✓	✗	✓
capitalone/cloud-custodian	✗	✗	✓	✗	✓	✗	✓	✓	✓	✓
blockstack/blockstack-core	✗	✗	✗	✗	✓	✓	✓	✓	✓	✓
adieuadieu/serverless-chrome	✗	✗	✓	✗	✓	✓	✓	✓	✗	✗
danilop/LambdAuth	✗	✗	✗	✗	✓	✓	✓	✓	✗	✓
serverless-heaven/serverless-webpack	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
bcongdon/corral	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓
awslabs/aws-serverless-auth-reference-app	✗	✓	✗	✗	✓	✓	✓	✓	✗	✓
open-lambda/open-lambda	✗	✗	✓	✗	✓	✓	✗	✓	✗	✗
0x4D31/honeyLambda	✗	✗	✓	✓	✓	✓	✗	✓	✗	✓
amplify-education/serverless-domain-manager	✗	✗	✗	✓	✓	✓	✓	✓	✓	✓
awslabs/serverless-photo-recognition	✗	✓	✓	✓	✓	✓	✗	✓	✗	✓
Successes	3	10	13	9	25	22	20	25	12	22
Failures	22	15	12	16	0	3	5	0	13	3

Table 4: Validation of maintainability characteristics

Analysability	✓
Modifiability	✓
Testability	✓
Modularity	✓
Reusability	✗
Deployment	✓
Configuration management	✗

7.3 Analysability

H3 : Software that use serverless architecture has a small serverless code base and small units of code but do have duplicate code.

As expected, the duplication of code is an issue encountered in our analysis. Half the projects encountered this problem. As described in **H2**, the units of code are bigger than expected. Analysability may be an issue for serverless applications.

Analysability may be an issue for serverless applications.

7.4 Modifiability

H4: Software that use serverless technologies tends to be easy to modify because it has simple units of code and low coupling. However, there is a lot of duplicated code.

It does not seem like serverless technologies tend to be easy to modify. The metric about simple units of code and code only being written once looses by a weak margin. For example, the small authentication service done by Coca-Cola has big units of code containing all the logic of a public function. Since the projects and the use cases are fairly small, it may be tempting for developers to put everything in the same unit and not to split the code base correctly. It may become an issue with small projects which are scaling more than the initial scope.

It is not clear whether serverless technologies tend to be easy to modify.

7.5 Testability

H5: Software that use serverless technologies are easy to test because they have a small code base, they have simple units of code and they are made of independent components.

The code base is small and the components are independent. Still, we see a good advantage for testability in a FaaS application. Another aspect that is not covered by the results is the nature of a serverless application. Since every public function is contained and is run in a new runtime, the tests of a function are simplified because the combinations of possible states for a function are reduced. The dependencies are only coming from the input parameters and from external sources, like a database or from networking.

Serverless technologies are easy to test because they have a small code base and they are made of independent components. Although tests are not always automated

7.6 Deployment

H6: The complexity of deployment of serverless software is harmful for its maintainability.

While we have our worries about deployments, we are seeing solutions implemented by the producers and the frameworks to mitigate theses issues. For example, the serverless framework is offering important features to ease the deployment. They implemented a command line which deploy your functions on your environment. This ease and accelerate the process, which is important, but it permits automation and the use of configuration files reduce the possibilities of human errors during a deployment.

Listing 1: Example of a configuration file used in the Serverless Framework

```
service: service-name
provider:
  name: aws
  stage: beta
  region: us-west-2
```

Another important aspect is the possibility to choose a stage target, the targets can be things like beta, prod, v1, v1_5, v2, etc. This other feature gives the possibility to manager multiple versions of the same API and ensure compatibility between services which are not updated at the same time. (Serverless Inc., 2017).

The deployment of serverless software is in general simplified by offering features to ease the deployment.

7.7 Configuration management

H7: Configuration management for software that uses serverless technologies is still an open question.

As mentioned in the previous section, there is still a large debate on many-repositories versus mono-repository. The projects we analyzed always had all their functions in the same repository. This is in fact much easier to manage by developers and maintainers. Google argues about the reasons why they are using a single repository for most of their code and the specific branching strategy and development workflow they are using in order to make it work (Potvin and Levenberg, 2016). Other companies like Amazon suggest breaking the project repository into smaller ones for large projects. (Workflow, 2017)

All of this seems to point to the facts that this is still an open question. Configuration management has a great impact on the deployment and development process and thus, on many aspects of software quality including maintainability.

Serverless configuration management is an open question, and there is still a large debate on many-repositories versus mono-repository.

7.8 Research opportunities

Serverless is a recent topic, and there are several research opportunities to explore.

Performance This paper was mostly about only one aspect of software quality. Other quality factors such as performance of FaaS in general and between different providers could be a good tool to help the decision-making process of what component to put in a serverless function and with which provider. To go even deeper in performance, a cost benefit analysis (CBA) framework, based on the resources that can be saved, could be made for serverless function to help CTOs decide when to use this kind of technology.

Stability Another important metric beyond the scope of this paper that could influence maintainability is the stability of the serverless functions over time. Raemaekers, Deursen and Visser (Raemaekers et al., 2012) have developed an interesting framework to analyze the stability of software over time that could probably be adapted to serverless functions.

Configuration Management As found previously in this paper, configuration management for serverless technologies is still an open question. It would be interesting to point out the benefits and disadvantages of each configuration management solution for serverless technologies to guide technical managers and developers in the design of their solution.

Continuous Integration Continuous integration has been made easy for monolithic projects with tools such as Jenkins. When it comes to serverless projects, in which multiple functions sometimes written in different languages can be used, it is not that clear on how to configure and perform continuous integration. This could be another interesting research question.

8 Threats of Validity

One of the biggest issues with our analysis is the limited number of public projects using a serverless architecture. A lot of the projects we found are simple demos as trivial as an application printing "Hello World," which are unusable for research like ours. The projects that we analyzed for the most part contains fewer than 10 KLOC of code and may not be a good representation of maintainability on bigger projects. MoonMail was the biggest one we found, but probably companies are making bigger applications which are not open-source. However, this limitation is because an immaturity of serverless domain, and we are investigate the current state of practice in serverless domain. Thus, I plan redo this study in the future to include more mature projects and also in not only open-source projects.

Another issue in our paper is the absence of empirical data on deployment. In this paper we analyzed about deployment in a qualitative way, however, no real data have been found on the subject.

Finally, the quality of your depends of BetterCodeHub outputs. That threats is mitigated because the tool is developed by experts in software quality, but more deep studies should be performed to confirm that claim.

9 Related work

Serverless programming is quite new in the programming model world and there is not yet much academic literature about it and a lot of open questions on the subject.

Fox et al. (Fox et al., 2017) described serverless technologies and their evaluation as *immature* by pointing out the fact that serverless code is complicated to debug and the serverless debugging technologies are non-existent. They also stated that most serverless programs are not easily portable to another FaaS provider. Despite these drawbacks, serverless patterns tend to be much more scalable and cost-effective.

A more recent paper (Baldini et al., 2017) brings up about the current trends and opened problems of serverless computing. At the moment, serverless platforms tend to limit developers to their specific ecosystem. This may change in the future as open source solutions such as *OpenLambda*(OpenLambda, 2016) appears and offer a framework which can deploy the solution of multiples cloud services. According to the article, the best use cases of FaaS are *CPU intensive* and *event-driven* applications. The best programming model for FaaS would be very similar to functional reactive programming, with small, stateless and idempotent functions. Finally, that paper pointed out some of interesting challenges that may influence the maintainability of serverless based solutions. These challenges include *deployment*, *composability* and *code granularity*.

Being relatively recent, serverless computing is just starting to see get some attention by the research community. A lot of the research has been trying to define what is serverless computing and Function-as-as-Service (FaaS) (Jonas et al., 2017)(Spillner, 2017)(Varghese and Buyya, 2017). Some work has been done to benchmark the performance of serverless architectures. It has been shown by (Jonas et al., 2017) that it is possible to build a model that is general enough to implement a number of distributed computing model such as Bulk synchronous parallel. In fact, they show that highly parallelizable operation, such as matrix multiplication, can be parallelized using lambda with very low bottleneck. More precisely, they built their own framework to serialize any highly parallelizable task and send them to an arbitrary number of workers on lambda. They show that the aggregates TFLOPS scale linearly with the number of workers. They also show that the read/write throughput scale also linearly with the number of workers. Finally, they break down the time taken for each phase of the lambda function. They show that the initialization bottleneck of the lambda is 15%. In certain application this bottleneck can be considerable.

Malawski et al. (Malawski et al., 2018) they compare the serverless architecture with the HyperFlow architecture and evaluate their benchmark on AWS, Google Cloud and IBM OpenWhisk. They execute their benchmark on Mersenne Twister and Linpack. They compare the CPU resources allocated with respect to the memory allocated. They observe that the CPU resources allocated on AWS scale linearly with the memory allocated, where the CPU resources allocated does not scale linearly with the memory allocated on Google cloud framework. They also observe that AWS achieve over 30 GFLOPS where Google cloud tops at 17 GFLOPS. This difference is due to the hardware used on each platform.

10 Conclusion

The goal of this paper was to determine whether or not the efforts of using serverless functions is worth in terms of software maintainability. We think that we have achieved this goal by providing an empirical analysis on metrics we could calculate on static code. While taking account of the threats to the validity of this study, we still see a tendency for an increase of maintainability in Function-as-a-Service architectures. We also wanted to promote more research on the subject and thus gave many ideas on possible future researches.

Our results show that 100% (25/25) of analyzed projects separate concerns in modules, Couple Architecture Components Loosely, keep architecture components balanced, keep the codebase small, and Write Clean Code. However (1) 88% (3/25) of analyzed projects do not write short units of code; (2) 60% of projects (15/25) write complex units of code and do not use well automated tests.

In fact, we found that (1) software that use serverless technologies have a low module coupling, their components are well balanced and are independent; (2) serverless projects tend to have big units of code; (3) analysability may be an issue for serverless applications; (4) it is not clear serverless technologies that tend to be easy to modify; (5) serverless technologies are easy to test because they have a small code base and they are made of independent components; (6) the deployment of serverless software is in general simplified by offering features to ease the deployment; and (7) serverless configuration management is an open question, and there is still a large debate on many-repositories versus mono-repository.

We also mentioned about deployment and configuration management, which are in many ways related to maintainability. We discussed on how deployment can be a challenge and what seems to be the possible solutions. We also found out that configuration management is still a debate.

Our major concerns for maintainability are on reusability and on configuration management. In the project we analyzed, there seems to have a lot of duplicated code and heavy units of code with big interfaces. It would be interesting to know if these bad tendencies are also seen in commercial projects that might make a heavier use of FaaS technologies.

We hope that more research will be done in the future on the subject to analyze subsequent, large-scale applications that exist in a serverless way and

in a regular way to compare our results. This way, we could really compare what the advantages and disadvantages brought by serverless architecture are in a more meaningful and precise way.

REFERENCES

Amazon (2017a). Continuous integration deployment for aws lambda functions with jenkins and grunt. https://aws.amazon.com/blogs. (Accessed on 10/17/2017).

Amazon (2017b). Serverless computing - amazon web services. https://aws.amazon.com/serverless/. (Accessed on 09/20/2017).

Baldini, I., Castro, P. C., Chang, K., Cheng, P., Fink, S. J., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R. M., Slominski, A., and Suter, P. (2017). Serverless computing: Current trends and open problems. *CoRR*, abs/1706.03178.

Boehm, B. W., Brown, J. R., and Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA. IEEE Computer Society Press.

Cavale, A. (2016). Our journey to microservices and a mono repository. https://www.shippable.com/index.html.

Cavano, J. P. and McCall, J. A. (1978). A framework for the measurement of software quality. *SIGSOFT Softw. Eng. Notes*, 3(5):133–139.

Committee, I. S. C. et al. (1990). Ieee standard glossary of software engineering terminology (ieee std 610.12-1990). los alamitos. *CA: IEEE Computer Society*, 169.

Fox, G. C., Ishakian, V., Muthusamy, V., and Slominski, A. (2017). Status of serverless computing and function-as-a-service (faas) in industry and research. *arXiv preprint arXiv:1708.08028*.

Framework, S. (2017). https://serverless.com/. (Accessed on 11/19/2017).

Golden, B. (2016). The roadmap to serverless computing: Are you prepared? https://techbeacon.com/roadmap-serverless-computing-are-you-prepared. (Accessed on 11/18/2017).

Google (2017). Cloud functions - serverless environment to build and connect cloud services. https://cloud.google.com/functions/. (Accessed on 11/17/2017).

ISO/IEC (2010). Iso/iec 25010 system and software quality models. Technical report.

Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. (2017). Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA. ACM.

Just Serverless (2017). Github - justserverless/awesome-serverless: Curated list of resources related to serverless architectures and the serverless framework. https://github.com/JustServerless/awesome-serverless#projects--services. (Accessed on 09/23/2017).

Malawski, M., Figiela, K., Gajek, A., and Zima, A. (2018). Benchmarking heterogeneous cloud functions. In Heras, D. B. and Bougé, L., editors, *Euro-Par 2017: Parallel Processing Workshops*, pages 415–426, Cham. Springer International Publishing.

Microsoft (2017). Azure functions—serverless architecture. https://azure.microsoft.com/en-ca/services/functions/. (Accessed on 11/17/2017).

Nazir, M., Khan, R. A., and Mustafa, K. (2010). A metrics based model for understandability quantification. *arXiv preprint arXiv:1004.4463*.

Netflix (2017). Netflix/bless: Repository for bless, an ssh certificate authority that runs as a aws lambda function. https://github.com/Netflix/bless. (Accessed on 10/17/2017).

OpenLambda (2016). https://open-lambda.org/. (Accessed on 09/23/2017).

Potencier, F. (2016). A monorepo vs manyrepos.

Potvin, R. and Levenberg, J. (2016). Why google stores billions of lines of code in a single repository. *Communications of the ACM*, 59(7):78–87.

Raemaekers, S., van Deursen, A., and Visser, J. (2012). Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387. IEEE.

Sbarski, P. (2016). *Serverless Architectures on AWS*. Manning Publications Co.

Serverless Inc. (2017). Serverless - the serverless application framework powered by aws lambda and api gateway. https://serverless.com/. (Accessed on 10/17/2017).

Shadija, D., Rezai, M., and Hill, R. (2017). Towards an understanding of microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)*, pages 1–6.

Software Improvement Group (2017). Better code hub. https://bettercodehub.com/. (Accessed on 09/23/2017).

Spillner, J. (2017). Snafu: Function-as-a-service (faas) runtime design and implementation. *CoRR*, abs/1703.07562.

Varghese, B. and Buyya, R. (2017). Next generation cloud computing: New trends and research directions. *CoRR*, abs/1707.07452.

Visser, J., Rigal, S., van der Leek, R., van Eck, P., and Wijnholds, G. (2016). *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code*. " O'Reilly Media, Inc.".

Workflow, S. F. G. (2017). https://serverless.com/framework/docs/providers/aws/guide/workflow/. (Accessed on 11/19/2017).