Towards a Technical Debt Conceptualization for Serverless Computing

Valentina Lenarduzzi[1], Antonio Martini[2], Sebastiano Panichella[3], and Damian Andrew Tamburri[4]

[1] LUT University, Lahti (Finland)
[2] University of Olso, Oslo (Norway)
[3] University of Zurich, Zurich (Switzerland)
[4] Jheronimus Academy of Data Science, 's-Hertogenbosch (The Netherlands)
`valentina.lenarduzzi@lut.fi`
`antonima@ifi.uio.no`
`panichella@ifi.uzh.ch`
`d.a.tamburri@uvt.nl`

Abstract. Serverless computing is an emergent compute paradigm reducing processing and operational units to single event-driven functions for orchestration. With its micro-granular architectural characteristics, serverless computing is bound to produce a currently inconceivable set of architectural issues and challenges in the medium- and long-term. Therefore, it is going to become more and more important to define Technical Debt (TD) items—that is, a way to measure the additional long-run project cost connected to immediately-expedient technical decisions—for serverless computing as well as hybrid compute models (e.g., Serverless + Microservices hybrids). This article illustrates a conceptualization and research roadmap in the direction of defining such TD items starting from a technical overview on serverless computing.

Keywords: Serverless Computing, Technical Debt, Bad Smells

1 Introduction

Serverless computing is growing its popularity in the last year [**?**],[10]. Serverless computing provides a platform to efficiently develop and deploy applications to the market without having to manage any underlying infrastructure [**?**]. Different serverless computing platforms such as AWS Lambda, Microsoft Azure Functions, and Google Functions have been proposed by the main cloud providers. Such platforms facilitate and enable developers to focus more on business logic, without the overhead of scaling and provisioning the infrastructure as the program technically runs on external servers with the support of cloud service providers [2].

Companies started to migrate from monolithic to Microservices and now to Serverless Function also to facilitate maintenance [14]. However, recent studies that monitored the microservices migration process, highlighted that maintenance costs increase after migration [12],[14],[11].

Considering the similarity between Serverless Functions and Microservices, we expected the same trend in Serverless Functions. The main issue is to understand is the similarity can be adopted also in the context of Technical Debt.

Therefore, the goal of this paper is to investigate Technical Debt in Serverless Functions and to preliminary identify a subset of items (or bad smells), mostly derived from the similarity with microservices, that can be good predictor of Technical Debt.

To the best of our knowledge, this is the first study that start to highlight the role of Technical Debt in Serverless Function.

The different items are based on the concept of Technical Debt for cloud-native applications, anti-patterns and bad smells proposed in microservices, and finally on our experience [13], [9].

2 Serverless Computing: a Primer

Serverless computing is a new paradigm that "*allows companies to efficiently develop and deploy functions without having to manage any underling infrastructure*" [8]. A serverless cloud application is deployed to infrastructure components that are transparent to the application developer. The cloud provider dynamically allocates and provisions servers, and the code is executed in almost-stateless containers that are event-triggered.

The serverless paradigm is currently receiving enormous interest within the cloud-focused developer community at present. The major cloud providers are promoting it as the basis of the next wave of innovation supporting the application developers and many application developers are indicating that serverless approaches do or will address key pain points for developers. As consequence, in the last year we have observed the usage of several emerging serverless computing platforms, such as AWS Lambda, Azure Functions, and Google Cloud Functions have been developed. These platforms allow developers to focus only on the business logic, while the overhead of monitoring, provisioning, scaling and managing the infrastructure are operated by the cloud service providers [2].

The available Serverless technologies can be grouped in two main categories:

Backend-as-a-Service enables to replace server-side components with off-the-shelf services. BaaS enables developers to outsource all the aspects behind a scene of an application so that developers can choose to write and maintain all application logic in the frontend. Examples are remote authentication systems, database management, cloud storage, and hosting. An example of BaaS can be Google Firebase, a fully managed database that can be directly used from an application. In this case, Firebase (the BaaS services) manage data components on our behalf.

Function-as-a-Service which represents an environment within which is possible to run software. Serverless functions are event- driven cloud-based systems where application development relies solely on a combination of third-party services, client- side logic, and cloud-hosted remote procedure calls [2]. FaaS allows developers to deploy code that, upon being triggered, is executed in an isolated environment. Each function typically describe a small part of an entire application.

Serverless functions can be developed in several contexts while, because of its limitations, it might have some issues in other contexts. As an example, long-running functions, such as machine learning training or long-running algorithms might have timeout problems, while constant workloads might result in higher costs compared to indefinitely running on-demand compute services like virtual machines or container runtimes.

More in general, this Serverless movement is in some sense driven by industry, influential figures in the academia are looking at serverless from a more fundamental point of view. The recent *Berkeley view on Serverless* [5] makes the important point that serverless platforms are more than FaaS runtimes; indeed the severless paradigm can only make sense if the FaaS layer sits atop a set of high level services which only require the addition of modest functionality - delivered via FaaS - to provide real business value. We elaborate in the next section the limit of the Serverless practices and movements.

2.1 Fallacies and Pitfalls in Serverless Computing

Since Serverless is in its infancy, researchers and practitioners have proposed a few set of good and best practices for its adoption and operations [2] as well as patterns for composing and triggering serverless functions [**?**] together with bad practices that should be avoided [10][**?**]. As with the success of Kubernetes, serverless open source solutions which have the most traction today are OpenWhisk and Knative.

OpenWhisk was an early mover in this space offering a coherent vision with a clear understanding that event based function invocation was a key characteristic of serverless. Being driven by IBM, it achieved some traction and is now under the auspices of the Apache foundation. The Open Source variant of OpenWhisk is useful but lacks many basic features for production environments, including log management, debugging supports, database integrations, etc. Knative is a newer technology which is closely coupled to the Kubernetes ecosystem: as with OpenWhisk, Knative has a quite comprehensive serverless vision, supporting distribution of functions within kubernetes pods and supporting integration points for ingestion of events. It currently has a large community behind it and has a more open approach than that of OpenWhisk. However, it is worth noting that despite these success stories, a significant issue for the serverless world is that the toolsets and interfaces required for productive developer experiences are only evolving: debugging, logging, tracing etc is still very much a work in progress.

Nupponen et al. [10] and Leitner et al. [**?**] proposed seven different bad practices that should be avoided in Serverless applications, such as *Asynchronous Calls*, *Shared Code between functions* or *Too many functions*.

In summary, several issues are still open in serverless [4].

[5] Jonas, Eric, et al. "Cloud Programming Simplified: A Berkeley View on Serverless Computing." arXiv preprint arXiv:1902.03383 (2019).

– Lack of a *significant serverless tool* sets and interfaces required for improving productivity of developer experiences, with debugging, logging, tracing solutions that are still very much a work in progress.
– Lack of *understanding of the event-driven paradigm*, especially for developers used to develop with different approaches.
– Lack of *solid tools* for deploying and developing functions. Deployment tools are not yet stable and development tools and IDEs do not yet provide a matured specific support.
– *Confusion between functions and microservices.* Some participants claimed that functions should only do only one thing, for a specific business logic.
– *Testing.* Since functions are triggered by well-defined interfaces, unit tests can be easily developed. However, system-level and integration testing become much more complex.

3 Technical Debt for Cloud-Native Architectures

Technical Debt (TD) is a metaphor to represent sub-optimal design or implementation solutions that yield a benefit in the short term [5], but make changes more costly or even impossible in the medium to long term [1]. Part of the overall TD is to be related to architecture sub-optimal decisions, and it is known as Architecture Technical Debt (ADT) [7].

More precisely, ATD is regarded as implemented solutions that are sub-optimal with respect to the quality attributes (internal or external) defined in the desired architecture intended to meet the companies business goals. Understanding the different ATD components is crucial for managing and reducing it [3].

In Cloud-based systems, the impact of migrating from monolith and of postponed activities on TD is still not clear. Only limited number of studies have investigated the impact of postponed activities on TD, especially in the context of microservices [11],[6].

The migration to microservices is a non-trivial task that requires deep re-engineering of the whole system. It is commonly performed on systems that are being actively developed. Therefore, in several cases, the development of new features is prioritized over refactoring of the code, which generates TD and increases the software maintenance effort. Migrating from monolith to Cloud based system should lead as consequence to a significant decrease of the maintenance cost. However, in the long run, TD grows slower after the migration process [6]. One of the possible cause could be the deal with a new system architecture. Developers should consider various aspects such as enabling the legacy system to communicate via Enterprise Service Bus with the microservices, dealing with authentication issues as well as with process-related issues such as the introduction of the DevOps culture, including continuous building and delivery [6].

TD issues related to architectural that can affect MS systems are mainly the existence of too many point-to-point connections among services, the presence of business logic in the communication layer, the lack of standards in the communication among services, weak source code and knowledge management and unnecessarily many different technologies used by the service developers in the communication among services [11].

The main impact of these issues is more encountered as extra cost (called Interest) due to the need of rewriting the code, extra effort to handle different technologies, and coupling between services [11]. Moreover, other factors can affect TD in microservices, such as the definition of a single middleware layer and removal of business logic from the communication layer [11].

4 Towards Technical Debt Issues for Serverless Computing

In this Section, we highlight the different aspects of Serverless that might led to the creation of technical debt. Our proposal will be validated with practitioners and experts in Serverless Computing.

Serverless can imply different types of TD: architectural debt, due to the re-architecting process, that could be "quick and dirty", but also enables to experiment, and therefore to use technical solutions (or shortcuts) that then are never fixed. As it is a fast development system, the system quickly mess-up creating also different type of debt (including code). Test is hard, therefore the testing of several "things" might be postponed, accumulating other debt.

Based on the aforementioned open issues, Serverless Functions can have the risk to accumulated higher Technical Debt than Microservices-based systems. Serverless context allows developers to explore different technologies and technical solutions. However, experimentation can create features "quick and dirty" that in the future will not well maintained and refactored from the company. Moreover, experimentation can led to misunderstanding among the teams that work in a single Serverless function and then have to integrate their work.

Another factor that should be considered is the higher risk to create a distributed monolith system composed by several Serverless Functions. The Integration of more Serverless Functions implicates to create Microservices composed by many components. However, if a Microservice distributed monolith system is an issues currently investigated [**?**],[13],[14] having unmanageable chaos inside the Microservice its self, due to Serverless Functions, is a more dangerous risk.

We propose a set of 13 Technical Debt items (bad smells) (Table 1), considering the aspects and the similarity between Serverless Functions and serverless. The different items are based on the concept of technical debt for cloud-native applications, anti-patterns and bad smells proposed in Serverless Functions, and finally on our experience [13], [9].

5 Road Map and Conclusion

In this paper, we introduce and illustrate a conceptualization of Technical Debt (TD) in Serverless Functions. Moreover, we proposed 13 TD items starting from a technical overview on serverless.

Table 1. TD items (bad smells) for Serverless Computing

TD item	Description
Lack of API Versioning	*If APIs are not versioned, in case of a new version of the APIs, the API consumers can face connection issues* [13].
Hardcoded endpoints	*Hardcoding the IP address and the ports of the services they need to connect bring very soon to problems when needing to change their locations.* [13].
Inappropriate Service Intimacy	*The Serverless Function is keeping on connecting to private data from other services instead of dealing with its own data* [13].
Serverless Function Greedy	In this case, developers often tend to create new Serverless Functions for each feature, even when they are not needed. This issue can generate the explosion of the number of Serverless Functions composing a system, resulting in a useless huge system that will easily become unmaintainable because of its size.
Multiple Services in One Container	*Placing multiple services in one container would constitute an architectural smell for the independent deployability of Serverless Functions.* If two Serverless Functions would be packaged in the same Docker image, spawning a container from such image would result in launching both Serverless Functions [9].
Shared Libraries	*Tightly couples Serverless Functions together, loosing independence between them. Moreover, teams need to coordinate each other when they need to modify the shared library* [13].
Shared Persistency	*It deals with Serverless Functions that share the dame data. This smell highly couples the Serverless Functions connected to the same data, reducing team and service independency* [13].
Single-layer Teams	*The classical approach of splitting teams by technology layers* (e.g., user interface teams, and middle- ware teams, and database teams) *is hence considered an architectural smell, as any change to a Serverless Function may result in a cross-team project having taken time and budgetary approval* [9].
Shared Functions between Serverless Functions	*A function that is used by different Serverless Functions.* In this case, we might have two issues: 1) the function is used by 2 or more Serverless Functions and modified by more than one team (high risk of errors) 2) the function is used by several teams, but only one team modify it (better from maintenance point of view, but increased coupling between teams)
Too many standards	*Despite Serverless Functions allow to use different technologies, the adoption of too many different technologies can be a problem in companies, specially in case of developers turnover.* [13].
Wobbly Service Interactions.	*The interaction of a Serverless Function mi with another Serverless Function m_f is "wobbly" when a failure in m_f can result in triggering a failure also in m_i.* [9].
Wrong Cuts	*Serverless Functions should be splitted based on business capabilities and not on technical layers*(presentation, business, data layers) [13].

We aim at validating the conceptualization of Technical Debt in Serverless Functions and the defined TD items. Therefore, we will design a study to understand what practitioners consider as Technical Debt in Serverless Functions.

We will carry out an exploratory study, structured as a mixed research method, composed by a set of interviews, a focus group, and a final set of group interviews. Based on these results, we will design and conduct detailed case studies, involving companies that develop or used Serverless Functions.

References

1. Avgeriou, P., Kruchten, P., Ozkaya, I., Seaman, C.: Managing Technical Debt in Software Engineering. Dagstuhl Reports 6(4), 110–138 (2016)
2. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., Suter, P.: Serverless Computing: Current Trends and Open Problems (2017)
3. Besker, T., Martini, A., Bosch, J.: Technical debt cripples software developer productivity: A longitudinal study on developers' daily software development work. In: International Conference on Technical Debt. pp. 105–114 (2018)
4. Casale, G., Artač, M., van den Heuvel, W.J., van Hoorn, A., Jakovits, P., Leymann, F., Long, M., Papanikolaou, V., Presenza, D., Russo, A., Srirama, S.N., Tamburri, D.A., Wurster, M., Zhu, L.: Radon: rational decomposition and orchestration for serverless computing. SICS Software-Intensive Cyber-Physical Systems (2019)
5. Cunningham, W.: The wycash portfolio management system. SIGPLAN OOPS Mess. 4(2), 29–30 (Dec 1992)
6. Lenarduzzi, V., Lomio, F., Saarimki, N., Taibi, D.: Does migrate a monolithic system to microservices decrease the technical debt? (2019)
7. Li, W., Shatnawi, R.: An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. J. Syst. Softw. 80(7), 1120–1128 (Jul 2007)
8. Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., Pallickara, S.: Serverless computing: An investigation of factors influencing microservice performance. In: International Conference on Cloud Engineering. pp. 159–169 (2018)
9. Neri, D., Soldani, J., Zimmermann, O., Brogi, A.: Design principles, architectural smells and refactorings for microservices: a multivocal review. Softw.-Inensiv. Cyber-Phys. Syst. (2019)
10. Nupponen, J., Taibi, D.: Serverless: What it is,what to do and what not to do. In: International Conference on Software Architecture (ICSA 2020) (2020)
11. Soares de Toledo, S., Martini, A., Przybyszewska, A., Sjberg, D.I.K.: Architectural technical debt in microservices: A case study in a large company. In: International Conference on Technical Debt. pp. 78–87 (2019)
12. Soldani, J., Tamburri, D.A., Heuvel, W.J.V.D.: The pains and gains of microservices: A systematic grey literature review. J. Syst. Softw. 146, 215 – 232 (2018)
13. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Software 35(3), 56–62 (2018)
14. Taibi, D., Lenarduzzi, V., Pahl, C.: Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing 4(5), 22–32 (2017)