Towards a Technical Debt Conceptualization for Serverless Computing

Valentina Lenarduzzi
.LUT University, Finland. valentina.lenarduzzi@lut.fi

Jeremy Daly
AlertMe, USA. jeremy@jeremydaly.com

Antonio Martini
University of Oslo, Norway. antonima@ifi.uio.no

Sebastiano Panichella
Zurich University of Applied Science (ZHAW). Switzerland. panc@zhaw.ch

Damian Andrew Tamburri
TU/e - Jheronimus Academy of Data Science, The Netherlands. d.a.tamburri@tue.nl

***Abstract*—Serverless computing aims at reducing processing and operational units to single event-driven functions for service orchestration and choreography. With its micro-granular architectural characteristics, serverless computing is bound to face considerable architectural issues and challenges in the medium- and long-term; are these bound to become Technical Debt? As known to many, technical debt is a metaphor that reflects the additional long-run project costs connected to immediately-expedient but unsavvy technical decisions. However, what does technical debt mean and how is it expressed in serverless computing and other hybrid compute models? This article represents the first attempt to conceptualize Technical Debt in such a context; we base our arguments over a technical overview of serverless computing concepts and practices and elaborate on them via empirical inquiry. Our results suggest that higher serviceability of serverless technologies is also characterized by the absence of mechanisms to support an adequate maintainability, testability, and monitoring of serverless systems. Indeed, in case of unexpected behaviours, testing and maintenance activities are more complex and more expensive, as mainly based on non-automated, manual tasks.**

SERVERLESS COMPUTING is growing in popularity in the last year [5], [8]; in layman's terms, it means providing a platform wherefore efficient development and deployment of applications to the market can take place in the form of micro-granular functions, without having to manage any underlying infrastructure hence, serverless functions [3], [5]. Different serverless computing platforms have been developed in recent years, that enables developers to concentrate on the business logic, without scaling the infrastructure as the program runs on external servers, but thanks to the support of cloud service providers.

In the last years most organizations migrated or started to migrate from monolithic to Microser-

vices, and some of them to Serverless Functions, to facilitate software system maintenance and adaptability [10]. However, recent studies on microservices migration processes, highlighted that maintenance costs increased after the migration [9], [10], [4].

Considering the similarity between Serverless Functions and Microservice designs, we expect this maintainability problem to be even more challenging in the context of Serverless Functions. Similarly to microservices based systems, the main issue about understanding and handling Technical Debt[1] also apply to systems using Serverless Functions.

The goal of this paper is to investigate what Technical Debt affects Serverless Functions, thus preliminary identifying a set of issues that can be potential predictors of Technical Debt.

To the best of our knowledge, this is the first study that investigates the role of Technical Debt in Serverless computing.

Based on available knowledge on TD and Serverless, we first provide an analysis about which Technical Debts [2] are most likely to affect Serverless Computing. Then we interviewed experts in such domain about which activities lead to accrue Technical Debt.

Combining these two contributions, we identified Technical Debt patterns for Serverless Computing that practitioners and developers can adopt to mitigate Technical Debt. The different items we identify in this work are based on the concepts of Technical Debt contextualized for cloud-native applications, anti-patterns, bad smells proposed in microservices as well as our experience [11], [7].

Moreover, we outline several issues related to Technical Debt that limiting the real potential of Serverless Computing.

Serverless Computing: A Primer

Serverless computing "allows companies to efficiently develop and deploy functions without having to manage any underlying infrastructure" [6]. For example, in the AWS Cloud[2], developing applications that use Lambda functions for compute, are automatically highly-available across multiple Availability Zones within a region, and require no additional infrastructure (e.g. load balancers, automation systems, or other software/components) to gracefully handle massive amounts of load. However, this also reduces the amount of proprietary code that must be written to make your application aware of the complexity of its host infrastructure.

The serverless paradigm is currently receiving enormous interest within the cloud computing domain and the development communities around it. Major cloud providers are promoting such a paradigm as the basis for the next wave of innovation while many application developers are indicating that serverless approaches are going to address key development challenges for many and diverse application scenarios. In the last year, an increasing usage of emerging several serverless computing platforms can be observed (e.g. Amazon Web service (AWS) Lambda[3], Microsoft Azure Functions[4], OpenWhisk[5], and Google Cloud Functions)[6]. These platforms allow developers to focus mainly on the business logic, while the overhead of monitoring, provisioning, scaling and managing the infrastructure are operated by the cloud service providers [8].

Available Serverless technologies can be grouped in two main categories:

Backend-as-a-Service (BaaS) replaces server-side components with off-the-shelf services, allowing developers to outsource aspects the behind the scene their applications such as databases, messages buses, or cloud storage. An example of BaaS is AWS DynamoDB, a fully managed database that manage data components on the developer's behalf [8].

Functions-as-a-Service (FaaS) are environment that enable developers to run software functions. FaaS allows to deploy code that, upon being triggered, is executed in an isolated environment. 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 [8].

[1] Technical Debt is a "design or implementation construct that is expedient in the short term, but sets up a technical context that can make a future change more costly or impossible" and is "limited to internal system qualities, primarily maintainability and evolvability" [1]

[2] https://aws.amazon.com

[3] https://aws.amazon.com/lambda/

[4] https://azure.microsoft.com

[5] openwhisk.apache.org

[6] cloud.google.com

The Berkeley view on Serverless [13] makes the important point that serverless platforms are more than FaaS runtimes; indeed the severless programming model only makes sense if the FaaS layer sits atop a set of high-level and value-based services, i.e., services which only require the addition of modest functionality which is reusable and very cheap computationally—delivered via FaaS—to provide real business value. We elaborate in the next section the limitations of the Serverless programming model.

Serverless Computing: Issues and Challenges

Since Serverless is in its infancy, researchers and practitioners have proposed a few sets of best practices for its adoption and operations as well as patterns for composing and triggering serverless functions [5] together with bad practices that should be avoided [8], [5]. Nupponen et al. [8] and Leitner et al. [5] proposed different bad practices that should be avoided in Serverless applications, such as Asynchronous Calls, Shared Code between functions or too many functions.

Technical Debt in Serverless Computing: A Preliminary Analysis

Serverless can be affected by different types of TD, which can increase serverless applications' costs. Based on the existing academic knowledge on TD and the known features of serverless computing, we elaborated the main types of technical debt that can affect such applications and we provide concrete examples.

Architectural debt: Serveless-based applications are prone to be re-architected, especially because they allow easy experimentation. However, the continuous change in serveless-based applications, if not properly controlled, increases the risk of quickly degrading the architecture [2], For example, a typical architectural problem affecting serveless-based applications is the low separation between the system business logic and its platform-specific interfaces.

Code debt: As any fast development lifecycle, the organizational structure can misbehave creating types of waste including replicated code in the form of extremely similar functions [2]. For example, a typical code problem affecting serveless-based applications is the use of multi-purpose functions, which significantly increase the likelihood of code debt (as consequence also the architectural debt). Indeed, a wide scope of serverless functions leads to less modular components and less flexible reuse of the code

Testing debt: Testing several serverless can become very complex very quickly, which might cause the testing activity to be postponed, which can create a high risk with respect to quality and reliability [2]. For example, a typical test problem affecting serveless-based applications concern the difficulty of testing multi-purpose functions and boilerplate code. Indeed, is is very expensive and usually error-prone the testing boilerplate code and multi-purpose functions in serverless applications.

We discuss how to address these problems in Section "The Path Forward".

Techincal Debt considerations for Cloud-Native Applications: Severless vs. Microservices: based on the aforementioned issues, Serverless Functions face a clear risk of accumulating higher TD than Microservices-based systems. Indeed, the Serverless programming model allows developers to explore different technologies and technical solutions but experimentations in creating "quick and dirty" features that, in the future, are not to be well-maintained and refactored from the company leads 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 too many Serverless Functions. The Integration of a higher number of Serverless Functions implies creating microservices composed of many sub-components, resulting in a mesh with unpredictable architectural and technical characteristics. However, if a distributed microservice monolith system is an issue currently investigated [5], [11], [10] having unmanageable serverless functions is a risk which demands further work.

Exploring the expert point of view

To better conceptualize the aforementioned conjectures and what is Technical Debt in Serverless Computing, we interviewed experts in such domain. We focused our questions on identifying the development activities that might accrue Technical Debt in Serverless-based applications.

Specifically, We interviewed three experts in Serverless Computing (3+ hands-on years experience with the technology). The first involved expert is **Jeremy Daly** (also one of the authors of this paper), an active member of the serverless community, who contributes to a number of serverless projects, including Lambda API, Serverless MySQL, and Lambda Warmer. The second expert is **Ran Ribenzaft**, the Co-Founder & CTO of **Epsagon**, while the third one is **Emrah Şamdan**, one of the two Leader members of **Thundra**. Specifically, **Epsagon** is a serverless monitoring tool that leverages distributed tracing and Artificial Intelligence (AI) technologies to provide a comprehensive, end-to-end view of serverless applications. **Thundra** is a provider that offers an application management, security, and compliance service that pinpoints down to the line of code if there is an issue.

The survey was kept minimalistic by design and consisted in the following questions:

- Which activities are postponed during Serverless based development?
- Which of these postponed activities might accrue Technical Debt?

We collected the information through open-ended questions to allow the respondents to report their experience. The face to face interviews lasted 30 minutes each. Two authors collected the answers separately and then checked possible conflicts. Any disagreement was discussed and clarified with the co-authors.

Technical Debt in Serverless Computing: a Conceptualization

Based on the experts' answers, we grouped the activities that can accrue TD in the following categories.

Unsanctioned technology use

The adoption of the wrong technology can cause a bigger TD in serverless compared to traditional microservices architectures. Serverless is more event-driven than traditional environments, which requires to architect the software correctly and with sanctioned and rigorously-evaluated alternatives, in a way that is possible to exchange the correct type of events over correct managed services, by using the right operational approach. We identified clear examples: "you might start with using an SQS to exchange a message but after a while you discover that some other Lambda function needs to consume the same message. It is typical to see that the second lambda function reads the same message from another queue instead of replacing the SQS with an SNS". This creates an architectural TD which can cause major architectural overhauls rotating around even a single function.

Knowledge churn

Serverless doesn't require any prior knowledge to build large scale applications, and the ability to ship fast but it does force you into a position of capturing and continuously confirming what best-practices and business as well as technical requirements and solutions work best for you (e.g., a very granular vs. a very coarse-grained decomposition). To him "When doing it not-right, you are just getting to an overall TD, but with the right experience and knowledge you can avoid it". This also leads to the necessity to find efficient and effective knowledge exchange mechanisms across teams as well as single individuals.

Process automation not set up

As a second major point, we outlined that **CI/CD processes** are also often delayed when first building an application since most start off small and become more complex over time. Couple this with poor test coverage and it results in low deployment confidence (and doesn't take advantage of blue/green, red/black, or canary deployments).

Not fully setting up an appropriate **Infrastructure-as-Code (IaC)** pipeline increases the deployment overhead since developers need to manually deploy each function independently, and if not using a proper CI/CD the situation would only get worse. Fully setting up infrastructure-as-code (for all of the resources, not just the functions) is therefore a must.

IaC could become a huge source of TD. This is often the case when too many resources are combined into single stacks, and shared resources (like persistence layers, communication channels, and other shared infrastructure) become inter-

twined and hard to decouple. Moreover, not setting up a CD/CI and IaC pipelines can lead to an immediate, constant and continuous TD. Without an automated process, developers should integrate and deploy manually the different serverless applications. As a consequence, developers should allocate time that can lead to postponing other activities.

Secrets management

Secrets management results in the inability (or inflexibility) of easily rotating credentials in external systems. This single issue would lead to broader security implications (discussed later) but in general also force some serverless solutions to actually become service-full applications! This means using hosted services to augment applications capacity in security—think DynamoDB for safe data storage or Mailchimp for more fine-grained email management..

Security

Security is also often delayed or never implemented. Most serverless developers are not "Cloud Engineers" or "Security Experts", so they like bypass IAM to speed development (think "star permissions"). These issues almost never get fixed. Moreover, most FaaS providers require explicit logging, which is perfect for developer abuse, while Secrets management is often overlooked or implemented incorrectly.

We fear that bypassing Identity and Access Management (IAM) to speed development could create considerable TD. For example in the erroneous design, implementation, and/or management of critical system functions as opposed to the management of other more regular functions. At the same time, the lack of knowledge of the developers about Security in the serverless domain and how it shall be managed, can only negatively influence the TD.

Testing activities postponed or not completely performed

Testing is one of the most dire issues. Separation of code into reusable pieces of business logic (e.g., hexagonal architecture) is often an afterthought, which makes writing unit tests more difficult. Integration tests are often "quick and dirty" as well, using sample code to generate events to test functions locally.

In particular during our interviews, all respondents focused the attention on **Integration Testing** and **Load/stress tests** respectively as crucial problems.

Load/stress test simulates "actual user load on any application or website". During normal and high loads it is deputy to check the application behaviour, determining the stability and robustness of the system.

Many times companies do not write Integration Tests because it's hard to accomplish. Instead, they test the integrated architecture in a staging environment by simply running the system with a specific set of inputs. This usually helps but does not verify the system 100%. Once they miss the integration tests, developers never come back and write those.

As reported for Security, testing is also a complex activity. Postponing testing or not testing completely a serlevess application, especially at integration level, does not allow to verify the entire system operability. A clear consequence is that the system does not correctly respond or does not comply with the requirements, and problems that developers do not know exist. This accumulates "invisible" TD that might be more "dangerous" compared with the visible. Load/stress tests. We believe that ot using load/stress tests could led to TD for two main reasons: Developers oversize the system compared with what they really need due to managing a higher load. Serverless despite being a system that automatically scales, some patterns [5], [12] can be used to support a higher load. The systems can anomaly react under stress. Some functions take more time increasing the cost or "hit" the maximum execution time or memory.

Inadequate or verbose logging

We believe that Inadequate or verbose logging can be also considered as a consequence of not completing testing. Developers to avoid this issue use logging in a massive way. However, logging leads to accumulating a lot of data that are unmanageable and require extra effort. Moreover, logging implies debugging that increases the effort for the developers and the possibility to accrue other TD.

TD design patterns for serverless functions

Based on the reported anaysis and expert experience, we identified concrete examples of design patterns Serverless-specific which—as discussed with Jeremy Daly—are useful to minimize TD levels and spread in Serverless applications.

- **TD minimization in Serverless Pipelines**. Utilizing package managers or, in AWS environments, Lambda Layers can facilitate the reuse of boilerplate code to minimize code debt. This allows functions to focus primarily on business logic and allows for that boilerplate code to be updated across multiple functions, iteratively and incrementally following typical DevOps principles.
- **TD minimization using hexagonal architectures**. Separating business logic from platform-specific interfaces using hexagonal architectures, i.e., handling AWS event parsing within a Lambda function and passing a standardized format to underlying components. This can ensure that proprietary business processes remain encapsulated and can be reused or moved even if the interface technology (e.g., Lambda) changes. This architectural device also helps to address testing debt while ensuring that unit tests remain viable regardless of infrastructure or other architectural changes.
- **TD minimization and higher serverless functions testability**. The use of single-purpose functions significantly reduces the likelihood of code debt and architectural debt. indeed, by minimizing the scope of functions, it is possible to create more modular components that enforce the DRY principle (i.e., the "don't repeat yourself" principle wherefore every piece of knowledge must have a single, unambiguous, authoritative representation within a system [15]), as well as allow for more flexible reuse and testability of involved functions. For example, single-purpose functions can be orchestrated via Step Functions—either in AWS or using other orchestration technology—with those same functions being choreographed via events and message buses (e.g., EventBridge in AWS) to achieve complex workflows.

The Path Forward

Complementary to previous work, we found that the Serverless functions adoption is characterized by activities, practices, and aspects that, according also to experts, can lead to higher TD levels. Having analyzed and conceptualized these issues, we identified the aspects that require enhancements and innovations for implementing the vision of more maintainable, evolvable (i.e., TD free) and secure systems, leveraging Serverless Functions:

Serviceability v.s. maintainability: Higher serviceability of serverless technologies has also the side effect of not having to their disposal also mechanisms to support an adequate maintainability and testability of the systems built on top of them. Indeed, in case of unexpected behaviours, testing and maintenance activities are more complex and more expensive, as mainly based on non-automated, manual tasks.

Technical Debt in Serverless computing is characterized by several conceptual antipatterns especially related to **Test Debt**, as indicated by the experts:

- *Unit testing*: system build with serverless functions are difficult to test, because identifying atomic business logic units that can be tested in isolation units is an open problem, leaving business-critical behaviours at system unit level uncovered.
- *Integration testing*: contemporary practices are not supported by automate tools for integration testing, which lead to incomplete or absent integration test code, leaving business-critical behaviours at system level uncovered.
- *CI/CD and IaC practices*: among the typical stages of CD/CI pipelines testing and deployment are highly limited (e.g., by the low testing coverage and not fully setting up of an IaC), which limit the benefits of fully operational CD/CI and IaC Pipelines.

Inadequate debugging and monitoring mechanisms: the actual observability and testability of system build on top of serverless platforms is very limited and contemporary logging, tracing, security practices and technologies are not adequate or "ready" in the serverless context.

Conclusion

This paper offers a first investigation on the existence of Technical Debt affecting serverless applications. Our findings, based on experts' experience on contemporary serveless "status quo" , highlighted several issues limiting the real potential of serverless functions in different industrial contexts and domains specifically issues reflect: (a) testing and testability (including aspects of overall service observability [14]); (b) organisational structure and knowledge management; (c) security. Although the examples we present are anecdotal, they reflect recurring conditions in industry encountered by our interviewees and which deserve further research—both from an organisational and technical perspective—for their resolution.

We conclude that, with respect to the aforementioned specific issues and challenges, technical debt research has put forth several management approaches which are altogether lacking for the serverless domain.

At the same time, the orthogonal nature of the aforementioned challenges seems to suggest entirely new areas of technical debt management theory which still remain completely untapped also for the scope of conventional systems or even systems which combine serverless and regular architectures alike.

Acknowledgment

We thank Ran Ribenzaft (Epsagon), and Emrah Şamdan (Thundra) for taking the time to share their experience and provide us their opinion. Some of the authors' work is partially supported by the European Commission grant no. 0421 (Interreg ICT), Werkinzicht, the European Commission grant no. 787061 (H2020), ANITA, European Commission grant no. 825040 (H2020), RADON, European Commission grant no. 825480 (H2020), SODALITE.

REFERENCES

1. P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering (dagstuhl seminar 16162). In Dagstuhl Reports (Vol. 6, No. 4). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016.
2. Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management. Journal of Systems and Software (Vol. 101). 2015.
3. Casale, G., Artač, M., van den Heuvel, W. et al. RADON: rational decomposition and orchestration for serverless computing. SICS Software.-Intensive. Cyber-Phys. Syst. (2019).
4. S. Soares de Toledo and A. Martini and A. Przybyszewska and D. I. K. Sjøberg. Architectural Technical Debt in Microservices: A Case Study in a Large Company. International Conference on Technical Debt. 2019.
5. P. Leitner, Erik Wittern, J. Spillner and W. Hummer. A mixed-method empirical study of Function-as-a-Service software development in industrial practice. Journal of Systems and Software. 2019
6. W. Lloyd and S. Ramesh and S. Chinthalapati and L. Ly and S. Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. International Conference on Cloud Engineering. pp. 159-169. 2018.
7. D. Neri and J. Soldani and O. Zimmermann and A. Brogi. Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Software-Intensive Cyber-Physical Systems. 2019
8. J. Nupponen and D. Taibi. Serverless: What it Is,What to Do and What Not to Do. International Conference on Software Architecture (ICSA 2020). 2020
9. J. Soldani and D.A. Tamburri and W.J. Van Den Heuvel. The pains and gains of microservices: A Systematic grey literature review. Journal of System and Software. 2018.
10. D. Taibi and V. Lenarduzzi and C. Pahl. Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing. 2017.
11. D. Taibi and V. Lenarduzzi. On the Definition of Microservice Bad Smells. IEEE Software. Vol 35 (3), pp. 56-62. 2018
12. D. Taibi, N. El Ioini, C. Pahl and J. R. Schmid Niederkofler. Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review. 10th International Conference on Cloud Computing and Services Science (CLOSER 2020). 2020
13. E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J.E. Gonzalez, R.A. Popa, I. Stoica, D.A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing. arXiv preprint arXiv:1902.03383. 2019
14. Tamburri, D. A., Bersani, M. M., Mirandola, R. & Pea, G. (2018). DevOps Service Observability By-Design: Experimenting with Model-View-Controller. (ESOCC 2018), Springer.
15. Hunt, A., Thomas, D. (2000). The Pragmatic program-

mer : from journeyman to master. Boston [etc.]: Addison-Wesley.

Valentina Lenarduzzi is a researcher at the LUT University in Finland. Her primary research interest is related to data analysis in software engineering, software quality, software maintenance and evolution, with a special focus on Technical Debt. She obtained her PhD in Computer Science at the Università degli Studi dell'Insubria, Italy, in 2015. She also spent 8 months as Visiting Researcher at the Technical University of Kaiserslautern and Fraunhofer Institute for Experimental Software Engineering (IESE). In 2011 she was one of the co-founders of Opensoftengineering s.r.l., a spin-off company of the Università degli Studi dell'Insubria.

Jeremy Daly is a passionate serverless advocate, an AWS Serverless Hero, and a senior technology leader with more than 20 years of experience building web and mobile applications. He is an active member of the serverless community, creating and contributing to open source serverless projects, and frequently consulting with companies looking to adopt serverless. Jeremy also writes extensively about serverless on his blog (jeremydaly.com), publishes Off-by-none, a weekly email newsletter that focuses on all things serverless (offbynone.io), and hosts the Serverless Chats Podcast (serverlesschats.com). He is currently the CTO of AlertMe.

Antonio Martini is Associate Professor at the University of Oslo and is a part-time researcher at Chalmers University of Technology. The current focus of Antonio's research is on Technical Debt, Architecture, Technical Leadership and Agile software development. Antonio's experience covers Software Engineering and Management in several contexts: large, embedded software companies, small, web companies, business to business companies, startups. His expertise ranges from technical programming to software architecture and software quality, to Agile ways of working and software business. Antonio Martini has worked as Principal Strategic Researcher at CA Technologies for a co-financed project for technology transfer related to Technical Debt and Architecture by the H2020 Marie Skłodowska-Curie grant of the European Union. Antonio has collaborated with several large companies such as Ericsson, Volvo, Saab, Axis, Grundfos, Siemens, Bosch and Jeppesen. He has also started his own consultancy company and has run projects with large companies in north- and central-Europe to manage and visualize Technical Debt. Antonio has been employed as a Postdoc Researcher at Chalmers, after having obtained a PhD in Software Engineering at Chalmers University of Technology, Sweden in 2015.

Sebastiano Panichella is a passionate Senior Computer Science Researcher at Zurich University of Applied Science (ZHAW). His research interests are in the domain of Software Engineering (SE) and cloud computing (CC): DevOps, Machine learning applied to SE, Software maintenance and evolution (with particular focus on Cloud, mobile, and Cyber-physical applications), Mobile Computing. Moreover, he is promoting research on Summarization Techniques for Code, Changes, and Testing. He is author (or co-author) of over several papers appeared in International Conferences (ICSE, ASE, FSE, ICSME, etc.) and Journals (EMSE, TSE, etc.). His research involved studies with industrial companies and open source projects and received best paper awards. He serves and has served as program committee member of various international conferences (e.g., ICSE, ASE, ICPC, ICSME, SANER, MSR). Dr. Panichella was selected as one of the top-20 (second in Switzerland) Most Active Early Stage Researchers (Results of the JSS journal) in Software Engineering (SE). He is a member of the ACM and IEEE. He is Editorial Board Member of Journal of Software: evolution and process (JSEP). He was Editor of a special Issues at EMSE and IST. He is also a Review Board member of the EMSE and TOSEM journals.

Damian Andrew Tamburri is an Associate Professor at the Eindhoven University of Technology and the Jheronimus Academy of Data Science, in s'Hertogenbosch, The Netherlands. His research interests rotate around DevOps and DataOps architectures, properties, and tools from a technical, social, and organisational perspective. Damian has published over 100+ papers in either top Journals or conferences in Software Engineering, Information Systems, and Services Computing. Also, Damian has been an active contributor and lead research in many EU FP6, FP7, and H2020 projects, such as S-Cube, MODAClouds, SeaClouds, DICE, ANITA, DossierCLOUD, ProTECT, and more. In addition, Damian is IEEE Software and ACM TOSEM editorial board member, secretary of the TOSCA TC as well as secretary of the IFIP TC2, TC6, and TC8 WG on "Service-Oriented Computing".